COMPUTER SECURITY

THIRD EDITION

Dieter Gollmann

Computer Security

THIRD EDITION

Computer Security

THIRD EDITION

Dieter Gollmann

Hamburg University of Technology

A John Wiley and Sons, Ltd., Publication

This edition first published 2011

Registered office
John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex, PO19 8SQ, United Kingdom

Library of Congress Cataloging-in-Publication Data

Gollmann, Dieter.
Computer security / Dieter Gollmann. – 3rd ed.
p. cm.
Includes bibliographical references and index.
ISBN 978-0-470-74115-3 (pbk.)
1. Computer security. I. Title.
QA76.9.A25G65 2011
005.8 – dc22

2010036859

A catalogue record for this book is available from the British Library.

Set in 9/12 Sabon by Laserwords Private Limited, Chennai, India
Printed in Great Britain by TJ International Ltd, Padstow

Contents

Preface

Ég geng í hring
í kringum allt, sem er.
Og utan þessa hrings
er veröld mín

Steinn Steinarr

Security is a fashion industry. There is more truth in this statement than one would like to admit to a student of computer security. Security buzzwords come and go; without doubt security professionals and security researchers can profit from dropping the right buzzword at the right time. Still, this book is not intended as a fashion guide.

This is a textbook on computer security. A textbook has to convey the fundamental principles of its discipline. In this spirit, the attempt has been made to extract essential ideas that underpin the plethora of security mechanisms one finds deployed in today's IT landscape. A textbook should also instruct the reader when and how to apply these fundamental principles. As the IT landscape keeps changing, security practitioners have to understand when familiar security mechanisms no longer address newly emerging threats. Of course, they also have to understand how to apply the security mechanisms at their disposal.

This is a challenge to the author of a textbook on computer security. To appreciate how security principles manifest themselves in any given IT system the reader needs sufficient background knowledge about that system. A textbook on computer security is limited in the space it can devote to covering the broader features of concrete IT systems. Moreover, the speed at which those features keep changing implies that any book trying to capture current systems at a fine level of detail is out of date by the time it reaches its readers. This book tries to negotiate the route from security principles to their application by stopping short of referring to details specific to certain product versions. For the last steps towards any given version the reader will have to consult the technical literature on that product.

Computer security has changed in important aspects since the first edition of this book was published. Once, operating systems security was at the heart of this subject. Many concepts in computer security have their origin in operating systems research. Since the emergence of the web as a global distributed application platform, the focus of

computer security has shifted to the browser and web applications. This observation applies equally to access control and to software security. This third edition of *Computer Security* reflects this development by including new material on web security. The reader must note that this is still an active area with unresolved open challenges.

This book has been structured as follows. The first three chapters provide context and fundamental concepts. Chapter 1 gives a brief history of the field, Chapter 2 covers security management, and Chapter 3 provides initial conceptual foundations. The next three chapters deal with access control in general. Chapter 4 discusses identification and authentication of users, Chapter 5 introduces the principles of access control, with Chapter 6 focused on the reference monitor. Chapter 7 on Unix/Linux, Chapter 8 on Windows, and Chapter 9 on databases are intended as case studies to illustrate the concepts introduced in previous chapters. Chapter 10 presents the essentials of software security.

This is followed by three chapters that have security evaluation as their common theme. Chapter 11 takes the Bell–LaPadula model as a case study for the formal analysis of an access control system. Chapter 12 introduces further security models. Chapter 13 deals with the process of evaluating security products.

The book then moves away from stand-alone systems. The next three chapters constitute a basis for distributed systems security. Chapter 14 gives a condensed overview of cryptography, a field that provides the foundations for many communications security mechanisms. Chapter 15 looks in more detail at key management, and Chapter 16 at Internet security protocols such as IPsec and SSL/TLS.

Chapter 17 proceeds beyond communications security and covers aspects of network security such as Domain Name System security, firewalls, and intrusion detection systems. Chapter 18 analyzes the current state of web security. Chapter 19 reaches into another area increasingly relevant for computer security – security solutions for mobile systems. Chapter 20 concludes the book with a discussion of recent developments in access control.

Almost every chapter deserves to be covered by a book of its own. By necessity, only a subset of relevant topics can therefore be discussed within the limits of a single chapter. Because this is a textbook, I have sometimes included important material in exercises that could otherwise be expected to have a place in the main body of a handbook on computer security. Hopefully, the general coverage is still reasonably comprehensive and pointers to further sources are included.

Exercises are included with each chapter but I cannot claim to have succeeded to my own satisfaction in all instances. In my defence, I can only note that computer security is not simply a collection of recipes that can be demonstrated within the confines of

a typical textbook exercise. In some areas, such as password security or cryptography, it is easy to construct exercises with precise answers that can be found by going through the correct sequence of steps. Other areas are more suited to projects, essays, or discussions. Although it is naturally desirable to support a course on computer security with experiments on real systems, suggestions for laboratory sessions are not included in this book. Operating systems, database management systems, and firewalls are prime candidates for practical exercises. The actual examples will depend on the particular systems available to the teacher. For specific systems there are often excellent books available that explain how to use the system's security mechanisms.

This book is based on material from a variety of courses, taught over several years at master's but also at bachelor's degree level. I have to thank the students on these courses for their feedback on points that needed better explanations. Equally, I have to thank commentators on earlier versions for their error reports and the reviewers of the draft of this third edition for constructive advice.

Dieter Gollmann
Hamburg, December 2010

Chapter 1

History of Computer Security

Those who do not learn from the past will repeat it.

George Santanya

Security is a journey, not a destination. Computer security has been travelling for 40 years, and counting. On this journey, the challenges faced have kept changing, as have the answers to familiar challenges. This first chapter will trace the history of computer security, putting security mechanisms into the perspective of the IT landscape they were developed for.

OBJECTIVES

- Give an outline of the history of computer security.
- Explain the context in which familiar security mechanisms were originally developed.
- Show how changes in the application of IT pose new challenges in computer security.
- Discuss the impact of disruptive technologies on computer security.

1.1 THE DAWN OF COMPUTER SECURITY

New security challenges arise when new – or old – technologies are put to new use. The code breakers at Bletchley Park pioneered the use of electronic programmable computers during World War II [117, 233]. The first electronic computers were built in the 1940s (Colossus, EDVAC, ENIAC) and found applications in academia (Ferranti Mark I, University of Manchester), commercial organizations (LEO, J. Lyons & Co.), and government agencies (Univac I, US Census Bureau) in the early 1950s. *Computer security* can trace its origins back to the 1960s. Multi-user systems emerged, needing mechanisms for protecting the system from its users, and the users from each other. Protection rings (Section 5.6.4) are a concept dating from this period [108].

Two reports in the early 1970s signal the start of computer security as a field of research in its own right. The RAND report by Willis Ware [231] summarized the technical foundations computer security had acquired by the end of the 1960s. The report also produced a detailed analysis of the policy requirements of one particular application area, the protection of classified information in the US defence sector. This report was followed shortly after by the Anderson report [9] that laid out a research programme for the design of secure computer systems, again dominated by the requirement of protecting classified information.

> In recent years the Air Force has become increasingly aware of the problem of computer security. This problem has intruded on virtually any aspect of USAF operations and administration. The problem arises from a combination of factors that includes: greater reliance on the computer as a data-processing and decision-making tool in sensitive functional areas; the need to realize economies by consolidating ADP [automated data processing] resources thereby integrating or co-locating previously separate data-processing operations; the emergence of complex resource sharing computer systems providing users with capabilities for sharing data and processes with other users; the extension of resource sharing concepts to networks of computers; and the slowly growing recognition of security inadequacies of currently available computer systems. [9]

We will treat the four decades starting with the 1970s as historical epochs. We note for each decade the leading innovation in computer technology, the characteristic applications of that technology, the security problems raised by these applications, and the developments and state of the art in finding solutions for these problems. Information technologies may appear in our time line well after their original inception. However, a new technology becomes a real issue for computer security only when it is sufficiently mature and deployed widely enough for new applications with new security problems to materialize. With this consideration in mind, we observe that computer security has passed through the following epochs:

- 1970s: age of the mainframe,
- 1980s: age of the PC,
- 1990s: age of the Internet,
- 2000s: age of the web.

1.2 1970s – MAINFRAMES

Advances in the design of memory devices (IBM's Winchester disk offered a capacity of 35–70 megabytes) facilitated the processing of large amounts of data (for that time). Mainframes were deployed mainly in government departments and in large commercial organizations. Two applications from public administration are of particular significance. First, the defence sector saw the potential benefits of using computers, but classified information would have to be processed securely. This led the US Air Force to create the study group that reported its finding in the Anderson report.

The research programmes triggered by this report developed a formal state machine model for the multi-level security policies regulating access to classified data, the Bell–LaPadula model (Chapter 11), which proved to be highly influential on computer security research well into the 1980s [23]. The Multics project [187] developed an operating system that had security as one of its main design objectives. Processor architectures were developed with support for primitives such as segmentations or capabilities that were the basis for the security mechanisms adopted at the operating system level [92].

The second application field was the processing of 'unclassified but sensitive' data such as personal information about citizens in government departments. Government departments had been collecting and processing personal data before, but with mainframes data-processing at a much larger scale became a possibility. It was also much easier for staff to remain undetected when snooping around in filesystems looking for information they had no business in viewing. Both aspects were considered serious threats to privacy, and a number of protection mechanisms were developed in response.

Access control mechanisms in the operating system had to support multi-user security. Users should be kept apart, unless data sharing was explicitly permitted, and prevented from interfering with the management of the mainframe system. The fundamental concepts for access control in Chapter 5 belong to this epoch.

Encryption was seen to provide the most comprehensive protection for data stored in computer memory and on backup media. The US Federal Bureau of Standards issued a call for a data encryption standard for the protection of unclassified data. Eventually, IBM submitted the algorithm that became known as the Data Encryption Standard [221]. This call was the decisive event that began the public discussion about encryption algorithms and gave birth to cryptography as an academic discipline, a development deeply resented at that time by those working on communications security in the security services. A first key contribution from academic research was the concept of public-key cryptography published by Diffie and Hellman in 1976 [82]. Cryptography is the topic of Chapter 14.

In the context of statistical database queries, a typical task in social services, a new threat was observed. Even if individual queries were guaranteed to cover a large enough query

set so as not to leak information about individual entries, an attacker could use a clever combination of such 'safe' statistical queries to infer information about a single entry. Aggregation and inference, and countermeasures such as randomization of query data, were studied in database security. These issues are taken up in Section 9.4.

Thirdly, the legal system was adapted and data protection legislation was introduced in the US and in European countries and harmonized in the OECD privacy guidelines [188]; several legal initiatives on computer security issues followed (Section 9.6).

Since then, research on cryptography has reached a high level of maturity. When the US decided to update the Data Encryption Standard in the 1990s, a public review process led to the adoption of the new Advanced Encryption Standard. This 'civilian' algorithm developed by Belgian researchers was later also approved in the US for the protection of classified data [68]. For the inference problem in statistical databases, pragmatic solutions were developed, but there is no perfect solution and the data mining community is today re-examining (or reinventing?) some of the approaches from the 1970s. Multi-level security dominated security research into the following decade, posing interesting research questions which still engage theoreticians today – research on non-interference is going strong – and leading to the development of high-assurance systems whose design had been verified employing formal methods. However, these high-assurance systems did not solve the problems of the following epochs and now appear more as specialized offerings for a niche market than a foundation for the security systems of the next epoch.

1.3 1980s – PERSONAL COMPUTERS

Miniaturization and integration of switching components had reached the stage where computers no longer needed to be large machines housed in special rooms but were small enough to fit on a desk. Graphical user interfaces and mouse facilitated user-friendly input/output. This was the technological basis for the personal computer (PC), the innovation that, indirectly, changed the focus of computer security during the 1980s. The PC was cheap enough to be bought directly by smaller units in organizations, bypassing the IT department. The liberation from the tutelage of the IT department resounded through Apple's famous launch of the Macintosh in 1984. The PC was a single-user machine, the first successful applications were word processors and spreadsheet programs, and users were working on documents that may have been commercially sensitive but were rarely classified data. At a stroke, multi-level security and multi-user security became utterly irrelevant. To many security experts the 1980s triggered a retrograde development, leading to less protected systems, which in fairness only became less secure when they were later used outside their original environment.

While this change in application patterns was gathering momentum, security research still took its main cues from multi-level security. Information-flow models and

non-interference models were proposed to capture aspects not addressed in the Bell–LaPadula model. The Orange Book [224] strongly influenced the common perception of computer security (Section 13.2). High security assurance and multi-level security went hand in hand. Research on multi-level secure databases invented polyinstantiation so that users cleared at different security levels could enter data into the same table without creating covert channels [157].

We have to wait for the Clark–Wilson model (1987) [66] and the Chinese Wall model (1989) [44] to get research contributions influenced by commercial IT applications and coming from authors with a commercial background. Clark and Wilson present well-formed transactions and separation of duties as two important design principles for securing commercial systems. The Chinese Wall model was inspired by the requirement to prevent conflicts of interest in financial consultancy businesses. Chapter 12 covers both models.

A less visible change occurred in the development of processor architectures. The Intel 80286 processor supported segmentation, a feature used by multi-user operating systems. In the 80386 processor this feature was no longer present as it was not used by Microsoft's DOS. The 1980s also saw the first worms and viruses, interestingly enough first in research papers [209, 69] before they later appeared in the wild. The damage that could be done by attacking computer systems became visible to a wider public. We will briefly describe two incidents from this decade. Both ultimately led to convictions in court.

1.3.1 An Early Worm

The Internet worm of 1988 exploited a number of known vulnerabilities such as brute force password guessing for remote login, bad configurations (*sendmail* in debug mode), a buffer overrun in the *fingerd* daemon, and unauthenticated login from trusted hosts identified by their network address which could be forged. The worm penetrated 5–10% of the machines on the Internet, which totalled approximately 60,000 machines at the time. The buffer overrun in the *fingerd* daemon broke into VAX systems running Unix 4BSD. A special 536-byte message to the *fingerd* was used to overwrite the system stack:

```
pushl   $68732f         push '/sh, <NUL>'
pushl   $6e69622f       push '/bin'
movl    sp, r10         save address of start of string
pushl   $0              push 0 (arg 3 to execve)
pushl   $0              push 0 (arg 2 to execve)
pushl   r10             push string addr (arg 1 to execve)
pushl   $3              push argument count
movl    sp, ap          set argument pointer
chmk    $3b             do "execve" kernel call
```

The stack is thus set up so that the command `execve("/bin/sh",0,0)` will be executed on return to the *main* routine, opening a connection to a remote shell via

TCP [213]. Chapter 10 presents technical background on buffer overruns. The person responsible for the worm was brought to court and sentenced to a $10,050 fine and 400 hours of community service, with a three-year probation period (4 May 1990).

1.3.2 The Mad Hacker

This security incident affected ICL's VME/B operating system. VME/B stored information about files in *file descriptors*. All file descriptors were owned by the user `:STD`. For classified file descriptors this would create a security problem: system operators would require clearance to access classified information. Hence, `:STD` was not given access to classified file descriptors. In consequence, these descriptors could not be restored during a normal backup. A new user `:STD/CLASS` was therefore created who owned the classified file descriptors. This facility was included in a routine systems update.

The user `:STD/CLASS` had no other purpose than owning file descriptors. Hence, it was undesirable and unnecessary for anybody to log in as `:STD/CLASS`. To make login impossible, the password for `:STD/CLASS` was defined to be the RETURN key. Nobody could login because RETURN would always be interpreted as the delimiter of the password and not as part of the password. The password in the user profile of `:STD/CLASS` was set by patching hexadecimal code. Unfortunately, the wrong field was changed and instead of a user who could not log in, a user with an unrecognizable security level was created. This unrecognizable security level was interpreted as 'no security' so the designers had achieved the opposite of their goal.

There was still one line of defence left. User `:STD/CLASS` could only log in from the master console. However, once the master console was switched off, the next device opening a connection would be treated as the master console.

These flaws were exploited by a hacker who himself was managing a VME/B system. He thus had ample opportunity for detailed analysis and experimentation. He broke into a number of university computers via dial-up lines during nighttime when the computer centre was not staffed, modifying and deleting system and user files and leaving messages from *The Mad Hacker*. He was successfully tracked, brought to court, convicted (under the UK Criminal Damage Act of 1971), and handed a prison sentence. The conviction, the first of a computer hacker in the United Kingdom, was upheld by the Court of Appeal in 1991.

1.4 1990s – INTERNET

At the end of 1980s it was still undecided whether fax (a service offered by traditional telephone operators) or email (an Internet service) would prevail as the main method of document exchange. By the 1990s this question had been settled and this decade became without doubt the epoch of the Internet. Not because the Internet was created

in the 1990s – it is much older – but because new technology became available and because the Internet was opened to commercial use in 1992. The HTTP protocol and HTML provided the basis for visually more interesting applications than email or remote procedure calls. The World Wide Web (1991) and graphical web browsers (Mosaic, 1993) created a whole new 'user experience'. Both developments facilitated a whole new range of applications.

The Internet is a communications system so it may be natural that Internet security was initially equated with communications security, and in particular with strong cryptography. In the 1990s, the 'crypto wars' between the defenders of (US) export restrictions on encryption algorithms with more than 40-bit keys and advocates for the use of unbreakable (or rather, not obviously breakable) encryption was fought to an end, with the proponents of strong cryptography emerging victorious. Chapter 16 presents the communications security solutions developed for the Internet in the 1990s.

Communications security, however, only solves the easy problem, i.e. protecting data in transit. It should have been clear from the start that the real problems resided elsewhere. The typical end system was a PC, no longer stand-alone or connected to a LAN, but connected to the Internet. Connecting a machine to the Internet has two major ramifications. The system owner no longer controls who can send inputs to this machine; the system owner no longer controls what input is sent to the machine. The first observation rules out traditional identity-based access control as a viable protection mechanism. The second observation points to a new kind of attack, as described by Aleph One in his paper on 'Smashing the Stack for Fun and Profit' (1996) [6]. The attacker sends intentionally malformed inputs to an open port on the machine that causes a buffer overrun in the program handling the input, transferring control to shellcode inserted by the attacker. Chapter 10 is devoted to software security.

The Java security model addressed both issues. Privileges are assigned depending on the origin of code, not according to the identity of the user running a program. Remote code (applets) is put in a sandbox where it runs with restricted privileges only. As a type-safe language, the Java runtime system offers memory safety guarantees that prevent buffer overruns and the like. Chapter 20 explores the current state of code-based access control.

With the steep rise in the number of exploitable software vulnerabilities reported in the aftermath of Aleph One's paper and with several high profile email-based virus attacks sweeping through the Internet, 'trust and confidence' in the PC was at a low ebb. In reaction, Compaq, Hewlett-Packard, IBM, Intel, and Microsoft founded the Trusted Computing Platform Alliance in 1999, with the goal of 'making the web a safer place to surf'.

Advances in computer graphics turned the PC into a viable home entertainment platform for computer games, video, and music. The Internet became an attractive new distribution

channel for companies offering entertainment services, but they had to grapple with technical issues around copy protection (not provided on a standard PC platform of that time). Copy protection had been explored in the 1980s but in the end deemed unsuitable for mass market software; see [110, p. 59). In computer security, digital rights management (DRM) added a new twist to access control. For the first time access control did not protect the system owner from external parties. DRM enforces the security policy of an external party against actions by the system owner. For a short period, DRM mania reached a stage where access control was treated as a special case of DRM, before a more sober view returned. DRM was the second driving force of trusted computing, introducing remote attestation as a mechanism that would allow a document owner to check the software configuration of the intended destination before releasing the document. This development is taken up in Sections 15.6 and 20.7.

Availability, one of the 'big three' security properties, had always been of paramount importance in commercial applications. In previous epochs, availability had been addressed by organizational measures such as contingency plans, regular backup of data, and fall-back servers preferably located at a distance from a company's main premises. With the Internet, on-line denial-of-service attacks became a possibility and towards the end of the 1990s a fact. In response, firewalls and intrusion detection systems became common components of network security architectures (Chapter 17).

The emergence of on-line denial-of-service attacks led to a reconsideration of the engineering principles underpinning the design of cryptographic protocols. Strong cryptography can make protocols more exploitable by denial-of-service attacks. Today protocols are designed to balance the workload between initiator and responder so that an attacker would have to expend the same computational effort as the victim.

1.5 2000s – THE WEB

When we talk about the web, there is on one side the technology: the browser as the main software component at the client managing the interaction with servers and displaying pages to the user; HTTP as the application-level communications protocol; HTML and XML for data formats; client-side and server-side scripting languages for dynamic interactions; WLAN and mobile phone systems providing ubiquitous network access. On the other side, there are the users of the web: providers offering content and services, and the customers of those offerings.

The technology is mainly from the 1990s. The major step forward in the 2000s was the growth of the user base. Once sufficiently many private users had regular and mobile Internet access, companies had the opportunity of directly interacting with their customers and reducing costs by eliminating middlemen and unnecessary transaction steps. In the travel sector budget airlines were among the first to offer web booking of flights, demonstrating that paper tickets can be virtualized. Other airlines

followed suit. In 2008, the International Air Transport Association (IATA) abandoned printed airline tickets in favour of electronic tickets as part of its 'Simplifying the Business' initiative.

Similarly, the modern traveller can arrange hotel reservations, car rentals, and conference registrations on the Internet. Other successful commercial applications are the bookseller Amazon, the mail-order business in general, e-banking, and the auction site eBay. The latter is particularly interesting as it enables transactions between private citizens where identities only need to be revealed to the extent of giving a shipping address.

The application-level software implementing the services offered on the web has become a main target for attacks. Major attack patterns are SQL injection (Section 10.5.2), cross-site scripting (Chapter 18), and attacks against the domain name system (Section 17.2). Application software accounts for an increasing number of reported vulnerabilities and real attacks. Attacks have stolen contact data from Gmail users,[1] and a worm spread to over a million users on MySpace.[2] Cross-site scripting overtook buffer overruns as the number one software vulnerability in the Common Vulnerabilities and Exposures list in 2005 and ranked first in the 2007 OWASP Top Ten vulnerabilities.[3] In 2006 SQL injection ranked second in the CVE list.[4]

In line with the growth of commercial activities on the web, the picture of the attacker has changed. The hackers of the 1990s often matched the stereotype of a male in his teens or twenties with limited social skills. One could discuss whether they were laudable whistle blowers exposing flaws in commercial software or whether they were creating wanton damage simply in order to bolster their self-esteem. In rare cases, attacks were made for financial gain. Today, criminal organizations have moved into the web. Criminals have no interest in high profile fast spreading worm attacks. They prefer to place trojans on their victims' machines to harvest sensitive data such as passwords, PINs, or TANs, or to use the victims' machines as part of a botnet.

A further aspect of commercial life has gained momentum because of the availability of the Internet as a high bandwidth global communications infrastructure. Outsourcing, virtual organizations, grid and cloud computing describe facets of a business world where companies merge, split, form joint enterprises, and move part of their activities to subcontractors or subsidiaries abroad on a regular basis. Sensitive information has to be protected among these recurring changes. At the same time information is becoming ever more crucial to a company's success. Security policies have to be

[1] http://jeremiahgrossman.blogspot.com/2006/01/advanced-web-attack-techniques-using.html
[2] http://www.betanews.com/article/CrossSite_Scripting_Worm_Hits_MySpace/1129232391
[3] http://www.owasp.org/index.php/OWASP_Top_Ten_Project
[4] Steve Christey and Robert A. Martin. Vulnerability type distributions in CVE, May 2007. http://cve.mitre.org/docs/vuln-trends/vuln-trends.pdf

defined, enforced, and managed in distributed heterogeneous settings. Policy administration, policy decisions, and policy enforcement become separate activities, potentially carried out at different sites and by different partners. Policy languages should provide support for controlling the effects of merging policies or of importing local policies into an enterprise-wide policy. Policy management systems may present a console for getting a comprehensive view of the various policies coexisting in an enterprise and for setting those policies, so that management rather than the local system owners are in control of policy. Having an accurate and up-to-date view of the current state of a dynamic and global enterprise is a challenge not only for management but also for supervisory authorities. Compliance with regulations that ask management to be truly in control of their companies, e.g. the Sarbanes–Oxley Act, is a major task in today's enterprises.

Efforts in the specification and design of relevant security mechanisms are under way in several areas. Web services security standards address cryptographic protection for XML documents, generic patterns for authentication (SAML), access control (XACML) and much more. The future will show which of these standards have stood the test of time. Federated identity management is a related topic, with applications in heterogeneous organizations but also for single sign-on systems for customers of federated companies. The integration of different authentication, authorization and accounting (AAA) systems, driven in particular by the convergence of Internet and mobile phone services, raises interesting challenges. On access control, research is pursuing ideas first introduced in the work on trust management. In the design of policy languages research is looking for the right balance between the expressiveness of a language and the strength of its formal foundations. Chapter 20 gives an introduction to these developments.

1.6 CONCLUSIONS – THE BENEFITS OF HINDSIGHT

Innovations developed in research laboratories – the mouse, graphical user interfaces, the Internet, the World Wide Web, mobile communications, public-key cryptography, worms, and viruses – have found their way into the mass market. These innovations are, however, not always used as originally envisaged by their inventors. For example, the creators of the Internet were surprised when email turned out to be their most popular service, the PC was turned into an Internet terminal, and SMS was not expected to grow into the major application of the mobile phone network it is today.

There is a lesson for security. Not only inventors are inventive, but also users. Proponents of new technologies are often asked to take a precautionary approach, study the impact of their technology and develop appropriate security mechanisms in advance. This approach can work if the use of the technology follows expectations, but is likely to fail in the face of user innovations. There is the added danger that familiarity with the 'old' security

challenges and their solutions inhibits the appreciation of new challenges where standard state-of-the-art solutions no longer work, and actually would be an impediment to the user. Multi-level secure operating systems and database management systems rarely fit commercial organizations. The recommendation to authenticate all messages in Internet protocols [2] gave way to privacy protection demands.

Innovations research defines disruptive technologies as cheap and comparatively simple technologies that do not meet the requirements of sophisticated users, who would not be seen using them, but are adopted by a wider public that does not need the advanced features [42]. Eventually, the new technology acquires more and more advanced features while remaining cheap as it serves a large enough market. In the end even sophisticated users migrate to the new technology. For example, there was once a market for workstations with much more powerful graphics processors than in a normal PC, but hardly any of the workstation manufacturers have survived. As another example, Internet protocols that were not designed to provide quality of service (QoS) are replacing more and more of the protocols traditionally used in telephone networks. Disruptive technologies may also be a problem for security. Initially, security features are neither required by their users nor by the applications for which they are used, but by the time they are a platform for sensitive applications it becomes difficult to reintegrate security.

1.7 EXERCISES

Exercise 1.1 Examine how end users' responsibilities for managing security have changed over time.

Exercise 1.2 *Full disclosure* asks for all details of a security vulnerability to be disclosed. Does this lead to an increase or a decrease in security?

Exercise 1.3 It has been frequently proposed to make software vendors liable for deficiencies in their products. Who would benefit from such regulations?

Exercise 1.4 Computer security is often compared unfavourably with car safety, where new models have to be approved before they can be brought to market and where vendors recall models when a design problem is detected. Is traffic safety a good model for computer security? Do we need the equivalent of driving licences, traffic rules, and traffic police?

Exercise 1.5 Social networks are a new application that has grown rapidly in recent years. What new security challenges are posed by social networks?

Exercise 1.6 'The net does not forget.' To what extent is it possible to delete information once it has been published on the Internet?

Exercise 1.7 Attacks can come from inside or outside an organization. Are there basic differences in the defences against insider and outsider threats? What is the relative importance of insider threats? Has the relative importance of insider threats changed as the modern IT landscape has been formed?

Exercise 1.8 Examine how security regulations and security mechanisms may be used as trade barriers.

Chapter 2

Managing Security

Before proceeding to the technical content of this book, this chapter will go over some important issues that have to be addressed when implementing security measures in practice. The deployment of security measures (and of IT in general) is a management decision. Technical security measures have to work hand in hand with organizational measures to be effective. Management decisions should be underpinned by some analysis of current risks and threats. We will give a brief introduction to security management and to risk and threat analysis.

OBJECTIVES

- **Set the scene for our discussion of computer security.**
- **Introduce security policies.**
- **Give a brief introduction to security management.**
- **Cover the basics of risk and threat analysis.**

2.1 ATTACKS AND ATTACKERS

When credit card payments over the Internet were first considered, it was thought essential that the traffic between customer and merchant should be protected. After all, the basic Internet protocols offer no confidentiality so parties located between customer and merchant could capture card numbers and use them later for fraudulent purchases. SSL was developed by Netscape to deal with this very problem in the mid 1990s.

However, the real danger may lurk elsewhere. Scanning Internet traffic for packets containing credit card numbers is an attack with a low yield. Badly protected servers at a merchant site holding a database of customer credit card numbers are a much more rewarding target. There is documented evidence that such attacks have occurred,[1] either to obtain credit card numbers or to blackmail the merchant.

Identity theft, i.e. using somebody else's 'identity' (name, social security number, bank account number, etc.) to gain access to a resource or service, exploits a weakness inherent in services that use non-secret information to authenticate requests.

Vulnerabilities in software processing external user input, such as Internet browsers or mail software, may allow external parties to take control of a device. Attackers may corrupt data on the device itself or use it as a stepping stone for attacks against third parties. Worms and viruses make use of overgenerous features or vulnerabilities to spread widely and overload networks and end systems with the traffic they generate. The Internet worm of November 1988 is an early well-documented example of this species (Section 1.3.1) [85]. Denial-of-service attacks against specific targets have started to occur since the late 1990s. Resilience against denial-of-service attacks has become a new criterion in the design of security protocols.

In the scenarios above the attacks came from the outside. Keeping the enemy outside the castle walls is a traditional paradigm in computer security. However, statistics on the sources of attacks often show that attacks from insiders account for a majority of incidents and the largest proportion of damage [220]. There is a suggestion that attacks via the Internet might change this picture, but insider fraud remains a considerable concern in organizations and in electronic commerce transactions.

It has been said that security engineering has as its goal to raise the effort for an attack to a level where the costs exceed the attacker's gains. Such advice may be short-sighted. Not every attacker is motivated by a desire for money. Employees who have been made redundant may want to exact revenge on their former employer. Hackers may want to demonstrate their technical expertise and draw particular satisfaction from defeating

[1] John Leyden, RBS WorldPay breach exposes 1.5 million, *The Register*, 29 December 2008. http://www.theregister.co.uk/2008/12/29/rbs_worldpay_breach/

security mechanisms that have been put in their way. 'Cyber vandals' may launch attacks without much interest in their consequences. Political activists may deface the websites of organizations they dislike.

There is similar variance in the expertise required to break into a system. In some cases insider knowledge will be required to put together a successful plan of attack. In this respect, *social engineering* may be more important than technical wizardry [172]. Hassling computer operators on the phone to give the caller the password to a user account is a favourite ploy. Some attacks require deep technical understanding. Other attacks have been automated and can be downloaded from websites so that they may be executed by *script kiddies* who have little insight into the vulnerabilities or features these attacks are exploiting.

This brief survey of attacks and attackers illustrates the numerous facets that may become relevant when managing security.

2.2 SECURITY MANAGEMENT

Security practitioners know that 'security is a people problem' that cannot be solved by technology alone. The legal system has to define the boundaries of acceptable behaviour through data protection and computer misuse laws. Responsibility for security within organizations, be they companies or universities, resides ultimately with management. Users have to cooperate and comply with the security rules laid down in their organization. Correct deployment and operation of technical measures is, of course, also part of the overall solution.

Protecting the assets of an organization is the responsibility of management. Assets include sensitive information such as product plans, customer records, financial data, and the IT infrastructure of the organization. At the same time, security measures often restrict members of the organization in their working patterns and there may be a potential temptation to flout security rules. This is particularly likely to happen if security instructions do not come from a superior authority but from some other branch of the organization.

It is thus strongly recommended to organize security responsibilities in an organization in a way that makes it clear that security measures have the full support of senior management. A brief *policy* document signed by the chief executive that lays down the ground rules can serve as a starting point. This document would be part of everyone's employment handbook. Not every member has to become a security expert, but all members need to know

- why security is important for themselves and for the organization,
- what is expected of each member, and
- which good practices they should follow.

Security-awareness programmes convey this information. The converse of users ignoring apparently unreasonable security rules are security experts treating apparently unreasonable users as the enemy. Trying to force users to follow rules they regard as arbitrary is not an efficient approach. Involving users as stakeholders in the security of their organization is a better way of persuading them to voluntarily comply with rules rather than to look for workarounds [4].

Organizations developing IT services or products must provide security training for their developers. There is rarely a clear dividing line between the security-relevant components and the rest of a system. It thus helps if developers in general are aware of the environment in which a service will be deployed and of expected dangers, so that they can highlight the need for protection even if they do not implement the protection mechanisms themselves. Developers must also be alert to the fact that certain categories of sensitive data, e.g. personal data, have to be processed according to specific rules and regulations. Finally, developers must keep up to date with known coding vulnerabilities.

2.2.1 Security Policies

Security policies are a core concept in computer security.

> **Security policy** – a statement that defines the security objectives of an organization; it has to state what needs to be protected; it may also indicate how this is to be done.

For example, a policy may regulate access to company premises. Who has permission to enter restricted areas? Must there be security guards to control access? Is everyone required to visibly wear an identification badge? Must visitors be accompanied at all times? Must their bags be checked? When are buildings locked? Who has access to keys?

A policy may regulate access to documents. For example, in the military world secret documents may be handed only to staff with adequate *clearance*. A policy may stipulate who is authorized to approve commercial transactions on behalf of the company. A policy may stipulate that certain transactions must be signed off by more than one person. It may define to what extent employees are allowed to use the company IT system for private purposes (web surfing, email). A policy may declare that senders of offensive emails will face disciplinary action. A policy may state which user actions the employer is allowed to monitor.

A policy may define password formats and renewal intervals. It may declare that only approved machines with the latest security patches installed may connect to the company network. A policy may state that sensitive emails must be encrypted. It may state that users have to give consent when personal data are collected.

There is thus considerable variety in the target of policies and the level of detail they are expressed in. To maintain clarity in our terminology, we follow the definitions laid

out in [216] and distinguish between organizational and automated security policies. A policy has given objectives:

> Security policy objective: A statement of intent to protect an identified resource from unauthorized use.

A policy also has to explain how the objectives are to be met. This can be done first at the level of the organization.

> Organizational security policy: The set of laws, rules, and practices that regulate how an organization manages, protects, and distributes resources to achieve specified security policy objectives.

Within an IT system, organizational policies can be supported by technical means.

> Automated security policy: The set of restrictions and properties that specify how a computing system prevents information and computing resources from being used to violate an organizational security policy.

Automated policies define access control lists and firewall settings, stipulate the services that may be run on user devices, and prescribe the security protocols for protecting network traffic.

2.2.2 Measuring Security

In security engineering we would love to measure security. To convince managers (or customers) of the benefits of a new security mechanism, wouldn't it be nice if you could measure the security of the system before and after introducing the mechanism? Indeed, it is difficult to reach well-founded management decisions if such information cannot be procured. The terms security measurement and security metric are used in this context, but there is no common agreement on their exact meaning. There are, however, two clearly defined phases.

First, values for various security-relevant factors are obtained. In SANS terminology[2] obtaining a value for such a factor is a *security measurement*. Some values can be established objectively, e.g. the number of open network ports, whether the latest patch has been installed for a software product, or the privilege level under which a service program is running. Other values are subjective, e.g. the reputation of a company or the security awareness of its employees.

Secondly, a set of measurements may be consolidated into a single value that is used for comparing the current security state against a baseline or a past state. In SANS terminology, the values used by management for making security comparisons are called

[2]SANS (SysAdmin, Audit, Network, Security) Institute.

security metrics. Other sources do not make this distinction in their terminology [64]. Sometimes, the result of a measurement can directly be used as a metric, e.g. the number of software vulnerabilities flagged by an analysis tool.

Ideally, a security metric gives a quantitative result that can be compared to other results, not just a qualitative statement about the security of the product or system being analyzed.

- A *product* is a package of IT software, firmware and/or hardware, providing functionality designed for use or incorporation within a multiplicity of systems.
- A *system* is a specific IT installation, with a particular purpose and operational environment [58].

Measurements of a product are indicative of its potential security, but even a secure product can be deployed in an insecure manner. An easily guessable password, for example, does not offer much protection. It is thus a task for security management to ensure that the security features provided are properly used.

For a product, you might use the number of security flaws (bugs) detected as a security metric. Tracking the discovery of flaws over time may serve as the basis for predicting the time to the discovery of the next flaw. Relevant methodologies have been developed in the area of *software reliability*. These methodologies assume that the detection of flaws and the invocation of buggy code are governed by a probability distribution given a priori, or a given family of probability distributions where parameters still have to be estimated. You might also measure the *attack surface* of a product, i.e. the number of interfaces to outside callers or the number of dangerous instructions in the code [124].

These proposals are measurements in the sense that they deliver quantitative results. It is open to debate whether they really measure security. How relevant is the number of security flaws? It is sufficient for an attacker to find and exploit a single flaw to compromise security. It is equally open to debate whether such metrics could be the basis for a meaningful security comparison of products, given that it is rare to find two products that serve exactly the same purpose. It has therefore been suggested that these metrics should be treated as *quality* metrics best used for monitoring the evolution of individual products.

Security metrics for a system could look at the actual configurations of the products deployed. In a system with access control features, look at the number of accounts with system privileges or the number of accounts with weak passwords. In a networked system, look at the number of open ports or at the services accessible from outside and whether the currently running versions have known vulnerabilities. Such attributes certainly give valuable *status information* but do not really give the quantitative results desired from a metric.

Specifically for computer networks, you may measure the connectivity of nodes in a network to assess how quickly and how far attacks could spread. You may also measure the time services are unavailable after an attack, or predict recovery times and cost of recovery for a given configuration and class of attacks.

Alternatively, you may try to measure security by measuring the cost of mounting attacks. You could consider

- the time an attacker has to invest in the attack, e.g. analyzing software products,
- the expenses the attacker has to incur, e.g. computing cycles or special equipment,
- the knowledge necessary to conduct the attack.

However, the cost of discovering an attack for the first time is often much larger than the cost of mounting the attack itself. When *attack scripts* are available, attacks can be launched with very little effort or knowledge of the system vulnerabilities being exploited.

As yet another alternative, you could focus on the assets in the system given and measure the risks these assets are exposed to. Section 2.3 gives an overview of risk and threat analysis. In summary, desirable as security metrics are, we have at best metrics for some individual aspects of security. The search for better metrics is still an open field of research.

2.2.3 Standards

Some industry branches have prescriptive security management standards that stipulate what security measures have to be taken in an organization. Typical examples are regulations for the financial sector,[3] or rules for dealing with classified material in government departments.[4]

Other management standards are best described as codes of best practice for security management. The most prominent of these standards is ISO 27002 [129]. It is not a technical standard for security products or a set of evaluation criteria for products or systems. The major topics in ISO 27002 are as follows:

- Security policy. Organizational security policies provide management direction and support on security matters.
- Organization of information security. Responsibilities for security within an enterprise have to be properly organized. Management has to be able to get an accurate view of the state of security within an enterprise. Reporting structures must facilitate efficient communication and implementation of security decisions. Security has to be maintained when information services are being outsourced to third parties.

[3]For example, the Payment Card Industry Data Security Standard (PCI DSS) supported by major credit card organizations.
[4]For example, the US policy stating that the AES encryption algorithm can be used for top secret data with 192-bit or 256-bit keys [68].

- Asset management. To know what is worth protecting, and how much to spend on protection, an enterprise needs a clear picture of its assets and of their value.
- Human resources security. Your own personnel or contract personnel can be a source of insecurity. You need procedures for new employees joining and for employees leaving (e.g. collect keys and entry badges, delete user accounts of leaving members). Enforced holiday periods can stop staff hiding the traces of fraud they are committing. Background checks on staff newly hired can be a good idea. In some sectors these checks may be required by law, but there may also be privacy laws that restrict what information employers may seek about their employees.
- Physical and environmental security. Physical security measures (fences, locked doors, etc.) protect access to business premises or to sensitive areas (rooms) within a building. For example, only authorized personnel get access to server rooms. These measures can prevent unauthorized access to sensitive information and theft of equipment. The likelihood of natural disasters can depend on environmental factors, e.g. whether the area is subject to flooding or earthquakes.
- Communications and operations management. The day-to-day management of IT systems and of business processes has to ensure that security is maintained.
- Access control. Access control can apply to data, services, and computers. Particular attention must be applied to remote access, e.g. through Internet or WLAN connections. Automated security policies define how access control is being enforced.
- Information systems acquisition, development, and maintenance. Security issues have to be considered when an IT system is being developed. Operational security depends on proper maintenance (e.g. patching vulnerable code, updating virus scanners). IT support has to be conducted securely (how do you deal with users who have forgotten their password?) and IT projects have to be managed with security in mind (who is writing sensitive applications, who gets access to sensitive data?).
- Information security incident management. Organizations have to be prepared to deal promptly with security incidents. It must be clear who is in charge of dealing with a given incident, whom to report the incident to, and how to reach a responsible person at all times.
- Business continuity management. Put measures in place so that your business can cope with major failures or disasters. Measures start with keeping backups of important data in a different building and may go on to the provision of reserve computing facilities in a remote location. You also have to account for the loss of key staff members. While incident management deals with the immediate reaction to incidents, business continuity management addresses precautionary measures to be taken in advance.
- Compliance. Organizations have to comply with legal, regulatory, and contractual obligations, as well as with standards and their own organizational security policy. Auditing processes can be put to efficient use while trying to minimize their interference with business processes. In practice, compliance often poses a greater challenge than fielding technical security measures.

Achieving compliance with ISO 27002 can be quite an onerous task. The current state of your organization *vis-à-vis* the standard has to be established and any shortcomings identified have to be addressed. There exist software tools that partially automate this process, again applying best practice, only this time to ensure compliance with the standard.

2.3 RISK AND THREAT ANALYSIS

Risk is associated with the consequences of uncertain events. *Hazard risks* relate to damaging events, *opportunity risks* to events that might have also have a positive outcome, e.g. to a financial investment on the stock exchange. IT risk analysis looks at hazard risks. It can be conducted during the design phase of a system, during the implementation phase, and during operations. It can be applied

- during the development of new *components*, e.g. in the area of software security,
- specifically for the IT *infrastructure* of an enterprise,
- comprehensively for all information assets of an *enterprise* (Figure 2.1).

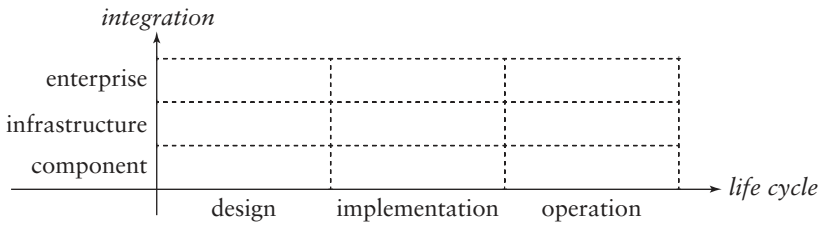


Figure 2.1: Applying Risk Analysis During a System Life Cycle

The literature on risk analysis uses terms such as threat, vulnerability, impact, asset, and attack. These terms are related. For example, an attack exploits a vulnerability to have a negative impact on an asset. Damage to an asset, e.g. a captured password, may facilitate a next attack step and cause damage to a further asset. Without a structured and systematic approach to risk analysis you are in danger of getting lost in the details of particular security problems but failing to establish a comprehensive overview of your risks.

There exist various ways of structuring risk analysis. Risk analysis, threat analysis, or vulnerability scoring are not necessarily different activities. They may be different names for the same objective, but they may also express a different focus when assessing potential damage.

Informally, risk is the possibility that some incident or attack can cause damage to your enterprise. An attack against an IT system consists of a sequence of actions, exploiting vulnerabilities in the system, until the attacker's goals have been achieved. To assess the

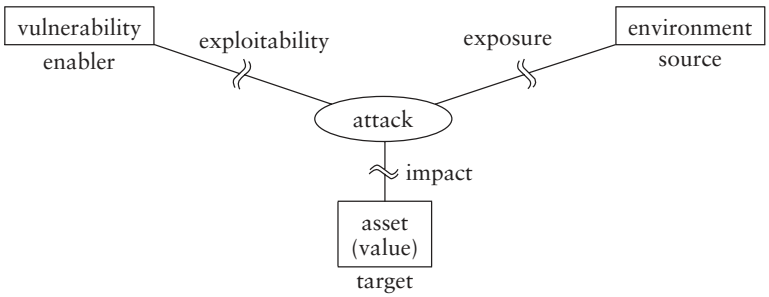


Figure 2.2: Factors in Risk Analysis

risk posed by the attack you have to evaluate the impact of the attack and the likelihood of the attack occurring (Figure 2.2). This likelihood will depend on the exposure of your system to potential attackers and how easily the attack can be mounted (exploitability of vulnerabilities). In turn, this will further depend on the security configuration of the system under attack.

Systems consist of *resources* and of *agents* operating on those resources. In a computer, processes are the agents. In an organization, an agent can be a person given a task to perform. This person may have been authorized to use resources necessary for executing the task. For example, a car from the company car pool may have been assigned for a visit to a customer. A person may be held responsible if the task has not been executed properly. Corruption of a resource can be categorized according to confidentiality, integrity, and availability (Chapter 3). Corruption of an agent can refer to actions that exceed the authority given, and to attempts at avoiding responsibility.

An analysis can already be performed at the design stage. At this point you have a conceptual model of your *assets* (resources) and agents, and perhaps of the environment in which your system will be deployed. You do not yet have implementation vulnerabilities to consider. You can then rate for each threat the potential damage (*impact*) and the exposure in the environment. We will use *threat analysis* specifically for 'risk analysis' at the design stage. Threat analysis may indicate security features that ought to be part of the system design.

2.3.1 Assets

As a first step assets have to be identified and valued. In an IT system, assets include:

- hardware – laptops, servers, routers, mobile phones, netbooks, smart cards, etc.;
- software – applications, operating systems, database management systems, source code, object code, etc.;

- data and information – essential data for running and planning your business, design documents, digital content, data about your customers, etc.;
- reputation.

Identification of assets should be a relatively straightforward systematic exercise. Measurement of asset values is more of a challenge. Some assets, such as hardware, can be valued according to their monetary replacement costs. For other assets, such as data and information, this is more difficult. If your business plans are leaked to the competition or private information about your customers is leaked to the public you have to account for indirect losses due to lost business opportunities. The competition may underbid you and your customers may desert you. Even when equipment is lost or stolen you have to consider the value of the data stored on it, and the value of the services that were running on it. In such situations assets can be valued according to their importance. As a good metric for importance, ask yourself how long your business could survive when a given asset has been damaged: a day, a week, a month?

2.3.2 Threats

A *threat* is an undesirable negative impact on your assets. There are various ways of identifying threats. You can categorize threats by their impact on assets and agents. For example, Microsoft's STRIDE threat model for software security lists the following categories [121]:

- Spoofing identities – an agent pretends to be somebody else; this can be done to avoid responsibility or to misuse authority given to someone else.
- Tampering with data – violates the integrity of an asset; e.g. security settings are changed to give the attacker more privileges.
- Repudiation – an agent denies having performed an action to escape responsibility.
- Information disclosure – violates the confidentiality of an asset; information disclosed to the wrong parties may lose its value (e.g. trade secrets); your organization may face penalties if it does not properly protect information (e.g. personal information about individuals).
- Denial of service – violates the availability of an asset; denial-of-service attacks can make websites temporarily unavailable; the media have reported that such attacks have been used for blackmail.
- Elevation of privilege – an agent gains more privileges beyond its entitlement.

Alternatively, you may identify threats by source. Would the adversary be a member of your organization or an outsider, a contractor or a former member? Has the adversary direct access to your systems or is an attack launched remotely?

2.3.3 Vulnerabilities

Once you move to the implementation stage, you have to examine your system for vulnerabilities. *Vulnerabilities* are weaknesses of a system that could be accidentally or intentionally exploited to damage assets. In an IT system, typical vulnerabilities are:

- accounts with system privileges where the default password, such as 'MANAGER', has not been changed;
- programs with unnecessary privileges;
- programs with known flaws;
- weak access control settings on resources, e.g. having kernel memory world-writable;
- weak firewall configurations that allow access to vulnerable services.

Vulnerability scanners provide a systematic and automated way of identifying vulnerabilities. Their knowledge base of known vulnerabilities has to be kept up to date. Organizations such as SANS or computer emergency response teams (CERTs) provide this information, as do security advisories of software companies.

Risk analysis has to measure the criticality of vulnerabilities. The criticality of a vulnerability depends on the attacks that might exploit it. A vulnerability that allows an attacker to take over a systems account is more critical than a vulnerability that gives access to an unprivileged user account. A vulnerability that allows an attacker to completely impersonate a user is more critical than a vulnerability where the user can only be impersonated in the context of a single specific service.

2.3.4 Attacks

A threat materializes when an attack succeeds. An *attack* is a sequence of steps. It may start innocuously, gathering information needed to move on to gain privileges on one machine, from there jump to another machine, until the final target is reached. To get a fuller picture of its potential impact, a forest of *attack trees* can be constructed. The root of an attack tree is a threat. The nodes in the tree are subgoals that must be achieved for the attack to succeed. Subgoals can be broken into further subgoals. There are AND nodes and OR nodes. To reach an AND node, all subgoals have to be achieved. To reach an OR node, it is enough if one subgoal is achieved.

Figure 2.3 gives a basic attack tree for the attack 'get password'. A password can be obtained by guessing, or by tricking an operator to reveal it, or by spying on the user. Guessing could be on-line or off-line. For off-line guessing, the attacker needs the encrypted password and has to perform a dictionary attack. The attacker could spy on the victim in person (so-called shoulder surfing), direct a camera at the keyboard, or direct a microphone at the keyboard and distinguish by sound the keys being pressed [16].

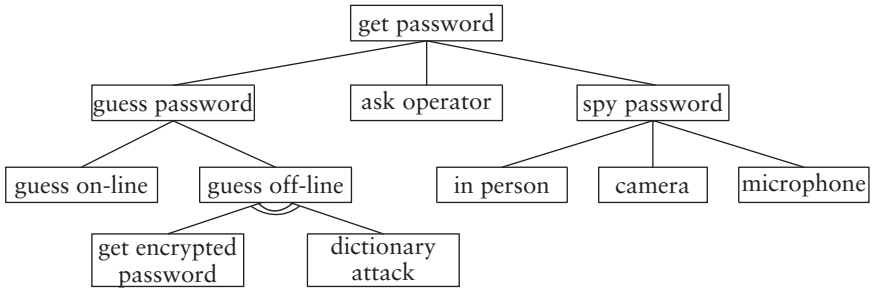


Figure 2.3: Attack Tree for Obtaining Another User's Password

It is possible to assign values to the edges in an attack tree. These values can indicate the estimated cost of an attack, the likelihood that it will occur, the likelihood that it will succeed, or some other aspect of interest. From these values, the cheapest attack, or the most likely attack, or the attack most likely to succeed can be computed.

Attack trees are a formalized and structured method for analyzing threats. Threat assessments become reproducible as the overall assessment of a threat can be traced to the individual assessments of subgoals. If the final result appears implausible, the tree can be consulted to see which subgoals were most critical for the final result, and those individual valuations may be adjusted to more 'sensible' values. This remark explains why the construction of attack trees is more an art than a science. You need experience to know when to readjust your ratings for subgoals, and when to adjust your preconceived opinion of the severity of a threat. You also need experience to know when to stop breaking up subgoals into ever more subgoals, a phenomenon known as *analysis paralysis*.

The severity of an attack depends on the likelihood that it will be launched, the likelihood that it will succeed, and the damage that it might do. Likelihood depends on the difficulty of the attack, on the motivation of the attacker, on the number of potential attackers, and on existing countermeasures. For example, *attack scripts* automate attacks, making it easy to launch the attack. They are also likely to be available to a larger set of attackers. Hence, such an attack would be rated more likely than an individual hand-crafted attack. The DREAD methodology that complements STRIDE demonstrates how the severity of an attack can be measured in a systematic manner [121]:

- Damage potential – relates to the values of the assets being affected.
- Reproducibility – attacks that are easy to reproduce are more likely to be launched from the environment than attacks that only work in specific circumstances.
- Exploitability – captures the effort, expertise, and resources required to launch an attack.

- Affected users – the number of assets affected contributes to the damage potential.
- Discoverability – will the attack be detected? In the most damaging case, you will never know that your system has been compromised. (In World War II, German intelligence refused to believe that many of their encryption schemes had been broken.)

2.3.5 Common Vulnerability Scoring System

The Common Vulnerability Scoring System (CVSS) starts from the vulnerabilities when organizing impact assessment [167]. The basic metric group collects generic aspects of a vulnerability (Table 2.1). The rating considers from where the vulnerability can be exploited (local or remote attacker?), how complex an exploit would have to be (related to exploitability in DREAD), and how many times an attacker would have to be authenticated during an attack (related to exposure and also to lacking social inhibitions due to a *feeling of impunity*). The rating also considers the standard impact categories of confidentiality, integrity, and availability.

Basic metrics		Temporal metrics	Environmental metrics	
Access vector	confidentiality impact	exploitability	collateral damage potential	confidentiality requirement
Access complexity	integrity impact	remediation level	target distribution	integrity requirement
Authentication	availability impact	report confidence		availability requirement

Table 2.1: Metrics in the Common Vulnerability Scoring System

The temporal metrics group captures the current state of exploits and countermeasures. Exploitability is related to reproducibility in DREAD and captures the state of exploits available. The remediation level notes to what extent fixes addressing the vulnerability are available. Report confidence rates the quality of the source announcing the vulnerability.

The environmental metrics group rates the impact on the assets of a given organization. Collateral damage potential covers damage outside the IT system, such as loss of life, loss of productivity, or loss of physical assets. Target distribution measures the number of potential targets within the organization. Environmental metrics rate IT assets according to the standard security requirements of confidentiality, integrity, and availability.

2.3.6 Quantitative and Qualitative Risk Analysis

Having measured the value of assets, the criticality of vulnerabilities, and the likelihood and impact of threats, you now face the tricky task of calculating your risks. Very informally, you can calculate risk as

$$Risk = Assets \times Threats \times Vulnerabilities.$$

This expression overloads the term *threat*. It stands for the potential negative impact on assets, but also for the likelihood that damage will occur.

You often have to deal with subjective ratings. Is the potential damage high or medium? Is an attack likely or very likely? You can attempt to reduce the effect of subjectivity by taking many ratings on detailed issues and then apply a combination algorithm to get the final result.

Quantitative risk analysis takes ratings from a mathematical domain such as a probability space. For example, you can assign monetary values to assets and probabilities to the likelihood of attacks, and then calculate the expected loss. This method has the pleasing feature of having a well-established mathematical theory as its basis, but the considerable drawback that your inputs are often just educated guesses. In short, the quality of the results you obtain cannot be better than the quality of the inputs provided. You could consider other mathematical frameworks, such as fuzzy theory, to make some provisions for the imprecise nature of your ratings. There are areas of risk analysis where quantitative methods work, but more often the lack of precision in the inputs does not justify a mathematical treatment.

Qualitative risk analysis takes values from domains that do not have an underlying mathematical structure:

- Assets could be rated on a scale of critical – very important – important – not important.
- Criticality of vulnerabilities could be rated on a scale of has to be fixed immediately – has to be fixed soon – should be fixed – fix if convenient.
- Threats could be rated on a scale of very likely – likely – unlikely – very unlikely.

CVSS follows this approach. The individual ratings are then mapped to weights that serve as input to the combination algorithm. DREAD uses a finer granularity for its ratings, i.e. numerical values from 1 to 10. The average of the five DREAD ratings is the final risk value. Whatever scheme you are using, guidance on how to assign ratings consistently is essential.

Encoding the combining function as a table has its disadvantages. The table has to be reworked when new risk factors emerge. For large tables it may be difficult to justify individual entries, or get management to approve your choices of weights. The Mehari methodology breaks up the combining function into small tables in a systematic manner [67]. All risk factors are rated with values between 1 and 4. Four-by-four tables combine the ratings for pairs of factors. The final risk indicators are computed by repeatedly applying such combination tables. Figure 2.4 gives a hypothetical example for such an evaluation. First, damage potential (of an attack) is calculated as a function of impact and the number of targets affected; likelihood is calculated as a function of exploitability

	$n = 1$	$n = 2$	$n = 3$	$n = 4$
$i = 4$	2	2	3	4
$i = 3$	1	2	2	3
$i = 2$	1	1	2	2
$i = 1$	1	1	1	2

damage potential as a function of impact and number of targets

	$f = 1$	$f = 2$	$f = 3$	$f = 4$
$e = 4$	1	2	2	3
$e = 3$	1	1	2	3
$e = 2$	1	1	2	2
$e = 1$	1	1	1	1

likelihood as a function of exploitability and feeling of impunity

	$l = 1$	$l = 2$	$l = 3$	$l = 4$
$d = 4$	2	3	4	4
$d = 3$	2	2	3	3
$d = 2$	1	2	2	3
$d = 1$	1	1	1	2

risk as a function of likelihood and damage potential

Figure 2.4: Computing Risk with Evaluation Tables

and feeling of impunity. The risk posed by the attack is a function of likelihood and damage potential.

Terminology in IT security is by no means standardized. You will encounter different conceptual frameworks than the one sketched here, possibly using the same terms but with different meanings. For example, CVSS defines threat as 'the likelihood or frequency of a harmful event occurring'. You may find vulnerability scanners that are marketed as risk analysis tools. Some may indeed give a rating for the vulnerabilities they detect. The burden is ultimately on you to find out what any 'risk analysis tool' is actually offering, and then place it in the framework of your choice.

2.3.7 Countermeasures – Risk Mitigation

The result of a risk analysis is a prioritized list of threats, together with recommended countermeasures to mitigate risk. Risk analysis tools usually come with a knowledge base of countermeasures for the threats they can identify. Risk analysis is also used for calculating the *return on security investment* (ROSI). ROSI compares for given security measures the expected reduction in risk with the costs of fielding the security measures.

It might seem trivially true that one should first go through a risk analysis before deciding on which security measures to implement. However, there are two reasons why this ideal approach may not work. Conducting a risk analysis for a larger organization will take

time, but the IT system in the organization and the world outside will keep changing. So, by the time the results of the analysis are presented, they are already somewhat out of date. Secondly, the costs of a full risk analysis may be difficult to justify to management.

For these reasons, organizations may opt for *baseline protection* as an alternative. This approach analyzes the security requirements for typical cases and recommends security measures deemed adequate. One of the best-known IT security baseline documents is maintained by the German Information Security Agency [49].

2.4 FURTHER READING

Anderson's book on security engineering gives an excellent insight into the full extent of the challenges faced in security [10]. A good discussion of security management, and of IT security in general, can be found in [212]. A discussion on the various meanings of the term 'security policy' is given in [216]. The observations on defining security policies in commercial organizations made in [212] are still valid today. The management of information security risks is discussed in [5], security measurement in [64]. Scoring tools for CVSS are available on the web.

2.5 EXERCISES

Exercise 2.1 Define a security policy for an examination system. Examination questions are set by the teacher and checked by an external examiner. Students sit the exam. Then their papers are marked, marks are approved by the examinations committee, results are published, and students may see their own papers. Which assets need to be protected? Who may get access to the documents used in this examination system?

Exercise 2.2 How would you measure the effectiveness of a security-awareness programme?

Exercise 2.3 Should a risk analysis of a computer centre include flooding damage to computing equipment even when the centre is in a high and dry location?

Exercise 2.4 Consider the theft of a central server from a university department. Which assets could be damaged if this happens? Construct an attack tree for this threat.

Exercise 2.5 Conduct a risk and threat analysis for a mobile phone service, taking into account that calls are transmitted over a radio link between mobile phone and base station, and that with international roaming a subscriber can use

the service in visited networks when away from home. Conduct your analysis from the subscribers' and the network operators' viewpoint.

Exercise 2.6 Bank customers can withdraw cash from automated teller machines (ATMs) using a cash card and a personal identification number (PIN). Build an attack tree for PIN compromise and rate the likelihood of the attacks.

Exercise 2.7 Conduct a risk and threat analysis for ATM cash withdrawals, both from the customer's and the bank's viewpoint.

Exercise 2.8 Perform your own CVSS scoring for a recently announced vulnerability.

Chapter 3

Foundations of Computer Security

We cannot start a meaningful exploration of computer security without defining the subject itself. We should not start such an exploration without some general guidelines that can help to bring order into the multitude of concepts and security mechanisms that you can encounter today. Thus, our first task is the search for a definition of computer security. To get away from discussing individual security systems in isolation, we will propose a set of general engineering principles that can guide the design of secure information processing systems. You are encouraged to keep looking out for these principles in the various security systems presented throughout this book.

OBJECTIVES

- Approach a definition of computer security, introducing confidentiality, integrity, and availability.
- Explain the fundamental dilemma of computer security.
- Mention some general design decisions that have to be made when constructing secure systems.
- Point out that computer security mechanisms have to rely on physical or organizational protection measures to be effective.

3.1 DEFINITIONS

Software may crash, communication networks may go down, hardware components may fail, human operators may make mistakes. As long as these failures cannot be directly attributed to some deliberate human action they would not be classified as security issues. Accidental failures count as *reliability* issues, operating mistakes as *usability* issues. Security is concerned with *intentional* failures. There may not always be a clear intent to achieve a particular goal, but there is at some stage a decision by a person to do something that person is not supposed to do. The root cause of security problems is thus human nature.

In proper academic tradition we start our investigations by defining the object of our studies. At least, we will try to do so. Computer security deals with the techniques employed to maintain security within a computer system. We will not attempt to distinguish between computer systems – loosely speaking, boxes with processors and memory in them – and information technology (IT) systems – roughly speaking, closely coupled networks of computer systems. Technology keeps moving too fast. Modern computers are already closely coupled networks of components. Software that was once an application program can become part of the operating system. Web browsers are a prominent example of this trend. Software running on your machine need not be stored on your machine. It can come from a local server, or maybe from a web server somewhere on the Internet. Hence, you may use computer security and IT security as synonyms without risking too much confusion.

At first glance, we seem to have a clear map for the road ahead as 'security' appears to be a rather obvious concept. However, security is one of those unfortunate notions that retreat further and further when you try to pin down their precise meaning. Much effort has gone into drafting definitions of computer security, and into later revisions of these definitions. The editors of these documents are almost inevitably accused of either being too narrow or trespassing into areas of computer science outside of computer security proper.

3.1.1 Security

Security is about the protection of assets. This definition implies that you have to know your assets and their value. This general observation is of course also true in computer security, and we have already mentioned the role of risk analysis in Section 2.3. Our focus now turns specifically to protection measures in computer systems. A rough classification of protection measures distinguishes between the following:

- prevention – taking measures that prevent your assets from being damaged;
- detection – taking measures that allow you to detect when an asset has been damaged, how it has been damaged, and who has caused the damage;

- reaction – taking measures that allow you to recover your assets or to recover from damage to your assets.

To illustrate this point, consider the protection of valuable items kept in your private home.

- Prevention: Locks at the door and window bars make it more difficult for a burglar to break into your home. A wall round the property, or the moat of a medieval castle, adds another layer of protection.
- Detection: You will detect when something has been stolen if it is no longer there. A burglar alarm goes off when a break-in occurs. Closed circuit television cameras[1] can provide information that leads to the identification of an intruder.
- Reaction: You can call the police. You may decide to replace the stolen item. The police may retrieve a stolen item and be able to return it to you.

Examples from the physical world can help to explain principles in computer security. However, it is not always possible or advisable to draw parallels between physical security and computer security. Some terms are actually quite misleading when used in an IT context. To take an example closer to our area, consider the use of credit card numbers when placing orders over the Internet. A fraudster could use your credit card number to make purchases that will be charged to your card. How can you protect yourself?

- Prevention: Use encryption when placing an order. Rely on the merchant to perform some checks on the caller before accepting a credit card order. Don't use your card number on the Internet.
- Detection: A transaction that you did not authorize appears on your credit card statement.
- Reaction: You can ask for a new card number. The cost of the fraudulent transaction may have to be covered by the card holder, the merchant where the fraudster made the purchase, or the card issuer.

In this example, the fraudster has 'stolen' your card number, but you still possess it. A legal system may treat this case differently from the case where your card has been stolen – you are still in possession of your card number – so that the fraudster cannot be charged for stealing your credit card number. When existing laws were found to fall short when faced with new kinds of undesirable behaviour, new computer misuse laws had to be passed to address new threats.

To continue this line of inquiry, consider your options for protecting confidential information. Perhaps you will only detect that your secret has been compromised when it is disclosed. In some cases, the damage may then be irretrievable. Your competitors may

[1] CCTV cameras have become a popular tool for securing public spaces. It is, however, a matter of debate how much security is actually achieved [202].

have got hold of a product design you had spent years developing, reached the market before you, and are reaping all the profits while you are going out of business. In such a situation, prevention is your only sensible method of protecting your assets. This also explains why historically computer security has paid a lot of attention to preventing the disclosure of confidential information.

There is not always a direct trade-off between prevention and detection. Practice shows that the more you invest in prevention, the more you may have to invest in detection (measuring security) to be certain that prevention works.

3.1.2 Computer Security

In a first attempt to capture the notion of computer security, we examine how information assets can be compromised. The definition most frequently proposed covers three aspects:

- confidentiality – prevention of unauthorized disclosure of information;
- integrity – prevention of unauthorized modification of information;
- availability – prevention of unauthorized withholding of information or resources.

You can immediately start a discussion on the priority of these topics and make a case for reordering these items. Alternatively, you can argue that the list is incomplete – as lists are never complete – and add further points such as *authenticity* if you have communications in mind, or *accountability* and *non-repudiation* if your interest is in applications such as electronic commerce.

Even at this general level, you will find disagreement about the precise definition of some aspects of security. We will pick definitions from documents important to the history of computer security such as the US Trusted Computer System Evaluation Criteria (Orange Book [224]), the European Information Technology Security Evaluation Criteria (ITSEC [70]), both covered in Chapter 13, and the International Standard ISO 7498-2 [125], the ISO/OSI security architecture for communications security, now superseded by ISO 10181, but still quite influential. It is often helpful to appreciate the context a definition has been taken from. The definitions above, for example, have been taken from ITSEC.

3.1.3 Confidentiality

Historically, security and secrecy were closely related. Even today, many people still feel that the main objective of computer security is to stop unauthorized users *reading* sensitive data. More generally, unauthorized users should not *learn* sensitive information. Confidentiality (privacy, secrecy) captures this aspect of computer security. The terms *privacy* and *secrecy* sometimes distinguish between the protection of personal data (privacy) and the protection of data belonging to an organization (secrecy). Confidentiality is a well-defined concept and research in computer security often concentrated on this topic, not least because it raised new issues that had no counterpart in physical security. Sometimes security and confidentiality are even used as synonyms.

Once you delve deeper into confidentiality issues you will face the question of whether you only want to hide the content of a document from unauthorized view, or also its existence. To see why one might take this extra step, consider traffic analysis in a communications system. The adversary simply looks at who is talking to whom and how often, but not at the content of the messages passed. Even so, an observer may derive useful information about the relationship between the corresponding parties. In the context of traffic analysis, you might require the *unlinkability* of certain events.

> **Unlinkability** – Two or more *items of interest* (messages, actions, events, users) are *unlinkable* if an attacker cannot sufficiently distinguish whether they are related or not.

If you want to hide who is engaging in a given action, you could ask for a property such as *anonymity*.

> **Anonymity** – A subject (user) is *anonymous* if it cannot be identified within a given *anonymity set* of subjects.

In a world of paper documents kept in safe storage, you could control read access to a document simply by specifying the list of people who were allowed to ask for it. Perhaps surprisingly, it sometimes necessary also to police write operations when enforcing confidentiality. You will read more on this topic in Section 11.2.

3.1.4 Integrity

It is not easy to give a concise definition of integrity. In general, integrity is about making sure that everything is as it is supposed to be. (This is perhaps a rather unhelpful definition, but it reflects reality.) Within the confinements of computer security, we may settle for the definition quoted in Section 3.1.2 and declare that integrity deals with the prevention of unauthorized *writing*. When this interpretation is used with information-flow policies (see Chapter 12), integrity is the dual of confidentiality and similar techniques can often be used to achieve both goals.

However, further issues such as 'being authorized to do what one does' or 'following the correct procedures' have also been subsumed under integrity. This approach is taken in the influential paper by Clark and Wilson [66], which declares integrity to be the property that

> no user of the system, even if authorized, may be permitted to modify data items in such a way that assets or accounting records of the company are lost or corrupted.

If we define integrity to be the prevention of all unauthorized actions, then confidentiality becomes a part of integrity.

So far we have captured security by specifying the user actions that have to be controlled. From a systematic point of view, you are better off by concentrating on the *state* of the

system when defining integrity. The Orange Book definition of 'data integrity' is precisely of this nature [224]:

> The state that exists when computerized data is the same as that in the source documents and has not been exposed to accidental or malicious alteration or destruction.

Here, integrity is a synonym for *external consistency*. The data stored in a computer system should correctly reflect some reality outside the computer system. This property is of course highly desirable, but it cannot be guaranteed entirely by mechanisms internal to the computer system.

To add to the confusion, other areas of information security have their own notions of integrity. For example, in communications security, integrity refers to the *detection* and *correction* of modifications to, insertions in, deletion, or replay of transmitted data. This includes both *intentional* manipulations and random transmission errors. You could view intentional modification as a special case of unauthorized modification, when nobody is authorized to modify. However, there is not much to gain from taking such a position because the presence, or absence, of an authorization structure has an impact on the nature of the problem that has to be solved, and on the respective security mechanisms.

Integrity is often a prerequisite for other security properties. For example, an attacker could try to circumvent confidentiality controls by modifying the operating system or an access control table referenced by the operating system. Hence, we have to protect the integrity of the operating system or the integrity of access control data structures to achieve confidentiality.

Finally, note that there exist even more general definitions of integrity, which treat security and availability as parts of integrity.

3.1.5 Availability

We take the definition given in ISO 7498-2 [125]:

> **Availability** – The property of being accessible and usable upon demand by an authorized entity.

Availability is very much a concern beyond the traditional boundaries of computer security. Engineering techniques used to improve availability often come from other areas such as fault-tolerant computing. In the context of security, we want to ensure that a malicious attacker cannot prevent legitimate users from having reasonable access to their systems. That is, we want to prevent *denial of service*. Again, we refer to ISO 7498-2 for a definition:

> **Denial of service** – The prevention of authorized access to resources or the delaying of time-critical operations.

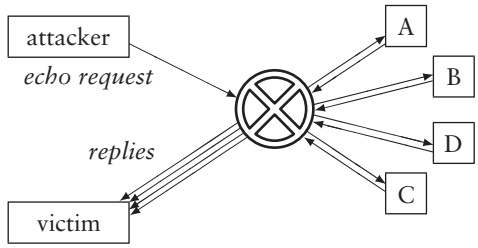


Figure 3.1: The *smurf* Denial-of-Service Attack

There have been *flooding* attacks on the Internet where an attacker effectively disables a server by overwhelming it with connection requests. Figure 3.1 shows *smurf*, one of the first denial-of-service attacks. The attacker sends an Internet Control Message Protocol (ICMP) echo request to the broadcast address of some network with the victim's address as the sender address (address spoofing). The echo request gets distributed to all nodes in that network. Each node replies back to the spoofed sender address, flooding the victim with reply packets. The amplification provided by the broadcast address works to the attacker's advantage.

In many situations, availability may be the most important aspect of computer security, but there is a distinct lack of security mechanisms for handling this problem. As a matter of fact, security mechanisms that are too restrictive or too expensive can themselves lead to denial of service. Designers of security protocols now often try to avoid imbalances in workload that would allow a malicious party to overload its correspondent at little cost to itself.

3.1.6 Accountability

We have now covered the three traditional areas of computer security. Looking back, you can see that they all deal with different aspects of access control and put their emphasis on the *prevention* of unwelcome events. We have to accept the fact that we will hardly ever be able to prevent all improper actions. First, we may find that authorized actions can lead to a security violation. Second, we may find a flaw in our security system that allows an attacker to find a way past our controls. Therefore, we may add a new security requirement to our list. Users should be held responsible for their actions. This requirement is particularly important in the context of electronic commerce, but can already be found in such historic documents as the Orange Book [224]:

> **Accountability** – Audit information must be selectively kept and protected so that actions affecting security can be traced to the responsible party.

To be able to do so, the system has to *identify* and *authenticate* users (see Chapter 4). It has to keep an *audit trail* of security-relevant events. If a security violation has occurred,

information from the audit trail may help to identify the perpetrator and the steps that were taken to compromise the system.

3.1.7 Non-repudiation

Non-repudiation is related to accountability.

> **Non-repudiation** – Non-repudiation services provide *unforgeable evidence* that a specific action occurred.

This definition is meaningful when analyzing the security services cryptographic mechanisms can provide. Digital signatures (Section 14.4) provide non-repudiation. Typical non-repudiation services in communications security are *non-repudiation of origin* providing evidence about the sender of a document, and *non-repudiation of delivery*, providing evidence that a message was delivered to a specific recipient.

Discussions about non-repudiation are prone to suffer from imprecise language. Should one talk about non-repudiation of *receipt* when mail has been delivered to a mailbox? Even more fundamental misunderstandings about non-repudiation prevail. It is sometimes said that non-repudiation services provide 'irrefutable evidence' about some event, and that such evidence will be 'accepted by any court in the world'. It is naïve to assume that mathematical evidence can make it impossible for a person to deny involvement in a disputed event. The concept of irrefutable evidence is alien to most legal systems. We will briefly return to this topic in Section 15.5.6.

3.1.8 Reliability

A discussion of security has several reasons for mentioning other areas of computing such as *reliability*, relating to (accidental) failures, and *safety*, relating to the impact of system failures on their environment, which also deal with situations where a system has to perform properly in adverse conditions. The first is an overlap in terminology. Depending on your preferred point of view, security is an aspect of reliability or vice versa. IFIP WG 10.4 has tried to escape from this dilemma by introducing *dependability* as the unifying concept and treating security, reliability, integrity, and availability as aspects of dependability [148]:

> **Dependability** – The property of a computer system such that reliance can justifiably be placed on the service it delivers. The **service** delivered by a system is its behaviour *as it is perceived* by its user(s); a **user** is another system (physical, human) which *interacts* with the former.

Furthermore, an application may have to address more than one issue at the same time. Consider, for example, a computer system in a safety-critical application. Security controls intended for stopping malicious actions might attempt to identify an intrusion by looking for unfamiliar patterns of behaviour. A reaction to an emergency may also appear unfamiliar – hopefully emergencies are rare events – so intrusion detection may

misread legitimate actions in a critical situation as an attack and potentially compound the problem by triggering security mechanisms that interfere with the actions of the emergency team. In general, you must not address security independently of the other requirements on the application you want to secure.

Finally, similar engineering methods are used in these areas. For example, standards for evaluating security software and for evaluating safety-critical software have many parallels and some experts expect that eventually there will be only a single standard.

3.1.9 Our Definition

This book will adopt the following operational definition of security.

> Computer security deals with the *prevention* and *detection* of *unauthorized* actions by users of a computer system.

With this definition, proper authorization and access control are essential to computer security. Proper authorization assumes the existence of a *security policy*, as explained in Section 2.2.1. The *correction* of the effects of improper actions will only play a minor role in our further discussions. The definition above describes what we do in computer security. When looking at the root causes for why we do it, we might adopt a definition such as:

> Computer security is concerned with the measures we can take to deal with *intentional* actions by parties behaving in some unwelcome fashion.

This definition does not mention unauthorized actions. It does not refer to attacks either but draws the boundary wider to include issues such as spam email. Sending an unsolicited email is not necessarily an attack. (If you are looking for a job, you might send your CV to companies that might find your skills useful, even if no vacancies are advertised.) Similarly, receiving an unsolicited email need not be an unwelcome event. This will change according to the number and nature of the emails one receives on a daily basis. When this misuse of email becomes too much of a nuisance, the fight against spam becomes a security issue.

Lesson

The main conclusions of this introductory discussion on terminology are as follows:

1. There is no single definition of security.
2. When reading a document, be careful not to confuse your own notion of security with that used in the document.
3. A lot of time is spent (and wasted) in trying to define unambiguous notations for security.

3.2 THE FUNDAMENTAL DILEMMA OF COMPUTER SECURITY

As the number of users relying on computer security has grown from a few organizations dealing with classified data to everyone connected to the Internet, the requirements on computer security have changed radically. This change has given rise to a fundamental dilemma.

> *Security-unaware* users have specific security requirements but usually no security expertise.

A security-unaware user cannot make educated decisions about security products and will have to pick standard 'best practice' solutions. Standard solutions may not address the user's specific requirements. When eliciting those requirements from the user, questions have to be asked that need awareness of security issues to be answered . . .

Compared to this fundamental dilemma, the conflict between security and ease of use is a straightforward engineering trade-off. The impact of security on performance is manifold.

- Security mechanisms need additional computational resources. This cost can be quantified easily.
- Security interferes with the working patterns users are accustomed to. Clumsy or inappropriate security restrictions lead to loss of productivity.
- Effort has to be put into managing security. Buyers of security systems therefore often opt for the product that has the best management features (which is often the one with the best graphical user interface).

3.3 DATA VS INFORMATION

Computer security is about controlling access to *information* and *resources*. However, controlling access to information can sometimes be elusive and is therefore often replaced by the more straightforward goal of controlling access to *data*. The distinction between data and information is subtle but it is also the root of some of the more difficult problems in security. Data represents information. Information is the (subjective) interpretation of data:

> **Data –** Physical phenomena chosen by convention to represent certain aspects of our conceptual and real world. The meanings we assign to data are called information. Data is used to transmit and store information and to derive new information by manipulating the data according to formal rules [46].

When there is a close link between information and corresponding data, the two concepts may give very similar results. However, this is not always the case. It may be possible to transmit information over a *covert channel* (see Section 11.2.5). There,

the data are 'grant' and 'deny' replies to access requests while the information received is the contents of a sensitive file. Another example is the problem of *inference* in statistical databases (Section 9.4). For a brief look at this issue, consider a database of tax returns. This database is used by tax inspectors processing individual records. It is also used by officials who need access to statistical summaries of tax returns but have no business reading individual records. Assume that the database management system allows statistical queries only on sufficiently large data sets to protect individual records. It would still be possible to combine the results from two queries over large data sets which differ only by a single record. Thus, even without accessing the data directly, information about an individual record can be derived.

3.4 PRINCIPLES OF COMPUTER SECURITY

You are likely to come across claims that computer security is a very complex issue, 'like rocket science'. Do not let such opinions frighten you off. If you are given the chance to implement the security features of a computer system in a systematic way, a disciplined approach to software (systems) development and a good understanding of a few essential security principles will carry you a long way. However, you certainly will struggle if you add on security to an already complex system as an afterthought, when you are constrained by design decisions that have been taken without any consideration of their security implications. Unfortunately, too often the latter is the case.

We will now propose a few fundamental design parameters of computer security. These design decisions provide the framework for structuring the presentations in this book. Figure 3.2 illustrates the main dimensions in the design space for computer security. The horizontal axis represents the focus of the security policy (Section 3.4.1). The vertical axis represents the layer of the computer system where a protection mechanism is implemented (Section 3.4.2).

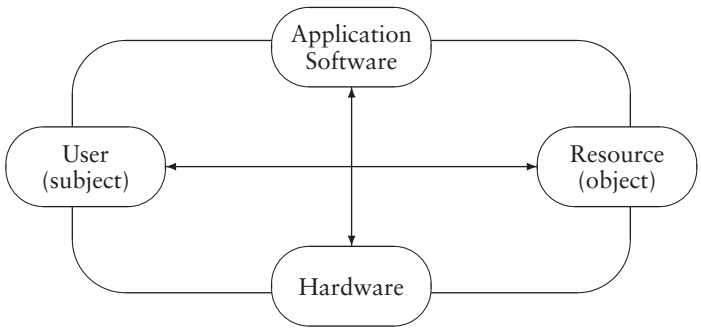


Figure 3.2: The Dimensions of Computer Security

3.4.1 Focus of Control

We could rephrase the definitions of integrity given in Section 3.1.4 and say that integrity has to do with compliance with a given set of rules. We can have rules on:

- the format and content of data items – for example, a rule could state that the balance fields in an accounts database have to contain an integer; such integrity rules define internal consistency properties and do not depend on the user accessing the data item or on the operation performed on the data item.
- the operations that may be performed on a data item – for example, a rule could state that only operations 'open account', 'check balance', 'withdraw', and 'deposit' have access to the balance fields in an accounts database and that only bank clerks are allowed to execute 'open account'; such rules may depend on the user and on the data item.
- the users who are allowed to access a data item – for example, a rule could state that only the account holder and bank clerks have access to balance fields in an accounts database.

We have just made an important general observation and arrived at our first design decision.

> **First Design Decision:** In a given application, should the protection mechanisms in a computer system focus on
>
> - data,
> - operations,
> - or users?

It is a fundamental design decision choosing which of these options to take when applying security controls. Operating systems have traditionally focused on protecting data (resources). In modern applications, it is often more relevant to control the users' actions.

3.4.2 The Man–Machine Scale

Figure 3.3 presents a simple layered model of a computer system. This model is only intended for general guidance. You should not expect to find all the layers in every computer system you analyze, nor should you be surprised to find systems where you can identify more than the five layers of our model.

- Users run *application programs* that have been tailored to meet specific application requirements.
- The application programs may make use of the *services* provided by a general purpose software package such as a database management system (DBMS), an object reference broker (ORB), or a browser.
- The services run on top of the *operating system*, which performs file and memory management and controls access to resources such as printers and I/O devices.

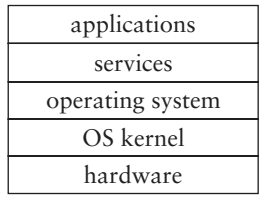


Figure 3.3: Layers of an IT System

- The operating system may have a *kernel* (micro-kernel, hypervisor) that mediates every access to the processor and to memory.
- The *hardware*, i.e. processors and memory, physically stores and manipulates the data held in the computer system.

Security controls can sensibly be placed in any of these layers. We have now explained the dimensions of our second fundamental security principle.

> **Second Design Decision:** In which layer of the computer system should a security mechanism be placed?

When you investigate fielded security products, you will observe security mechanisms at every layer of this model, from hardware to application software. It is the task of the designer to find the right layer for each mechanism, and to find the right mechanisms for each layer.

Take a second look at our new design decision, visualizing the security mechanisms of a computer system as concentric protection rings, with hardware mechanisms in the centre and application mechanisms on the outside (Figure 3.4). Mechanisms towards the centre tend to be more generic, more computer-oriented, and more concerned with controlling access to data. Mechanisms at the outside are more likely to address individual user requirements. Combining our first two design decision, we will refer to the *man–machine*

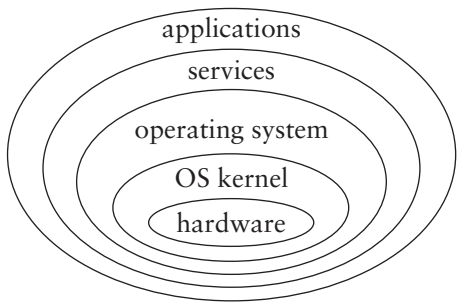


Figure 3.4: The Onion Model of Protection Mechanisms

Figure 3.5: The Man–Machine Scale for Access Control Mechanisms

scale for placing security mechanisms (Figure 3.5). This scale is related to the distinction between data (machine-oriented) and information (man-oriented).

3.4.3 Complexity vs Assurance

Frequently, the location of a security mechanism on the man–machine scale is closely related to its complexity. To the right of the scale you find simple generic mechanisms, while applications often clamour for *feature-rich* security functions. Hence, there is yet another decision you have to take.

> **Third Design Decision:** Do you prefer simplicity – and higher assurance – to a feature-rich security environment?

To achieve a high degree of assurance, the security system has to be examined in close detail and as exhaustively as possible. Hence, there is a trade-off between complexity and assurance. The higher an assurance level you aim for, the simpler your system ought to be. As an immediate consequence, you can observe that

> feature-rich security systems and high assurance do not match easily.

It will not come as a surprise that high assurance requires adherence to systematic design practices. In fact, computer security is one of the areas that early on adopted formal methods as a tool in their quest for the highest assurance levels.

This decision is also linked to the fundamental dilemma of computer security. A simple generic mechanism may be unable to enforce specific protection requirements, but to choose the right options in a feature-rich security environment users have to be security experts. Security-unaware users are left in a no-win situation.

3.4.4 Centralized or Decentralized Controls

Within the domain of a security policy, the same controls should be enforced. If there is a single central entity in charge of security, then it is easy to achieve uniformity, but this central entity may become a performance bottleneck. Conversely, a distributed solution may be more efficient but we have to take additional care to guarantee that the different components define and enforce the policy consistently.

Fourth Design Decision: Should the tasks of defining and enforcing security be given to a central entity or should they be left to individual components in a system?

This question arises naturally in distributed systems security and you will see examples of both alternatives. However, this question is also meaningful in the context of mainframe systems as demonstrated by the mandatory and discretionary security policies of the Bell–LaPadula model covered in Section 11.2.

3.5 THE LAYER BELOW

So far we have briefly touched on assurance but have predominantly explored options for expressing the most appropriate security policies. It is now time to think about attackers trying to bypass your protection mechanisms. Every protection mechanism defines a *security perimeter* (*boundary*). The parts of the system that can malfunction without compromising the protection mechanism lie outside this perimeter. The parts of the system that can be used to disable the protection mechanism lie within this perimeter. This observation leads to an immediate and important extension of the second design decision from Section 3.4.2:

Fifth Design Decision: How can you prevent an attacker from getting access to a layer below the protection mechanism?

An attacker with access to the 'layer below' is in a position to subvert protection mechanisms further up. For example, if you gain systems privileges in the operating system, you are usually able to change programs or files containing the control data for security mechanisms in the services and applications layers. If you have direct access to the physical memory devices so that you can manipulate the raw data, the logical access controls of the operating system have been bypassed. We give further examples below to illustrate this point. The fact that security mechanisms have a soft underbelly and are vulnerable to attacks from lower layers should be a reason for concern, but not for despair. When you reach the stage where you cannot apply computer security mechanisms or do not want to do so, you still can put in place physical or organizational security measures (Figure 3.6).

We will now give a few examples of access to the layer below. The explanations assume some basic understanding of the way computer systems work.

Recovery Tools

If the logical organization of the memory is destroyed due to some physical memory fault, it is no longer possible to access files even if their physical representation is still intact. Recovery tools, like Norton Utilities, can help to restore the data by reading the (physical) memory directly and then restoring the file structure. Such a tool can, of course, be used to circumvent logical access control as it does not care for the logical memory structure.

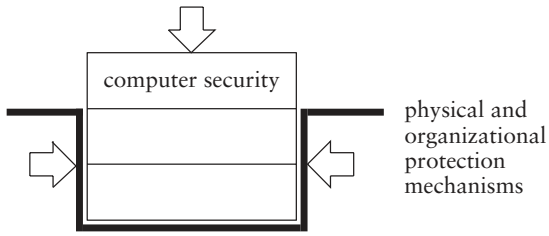


Figure 3.6: Physical and Organizational Security Measures Controlling Access to the Layers Below

Unix Devices

Unix treats I/O devices and physical memory devices like files. The same access control mechanisms can therefore be applied to these devices as to files. If access permissions are defined badly, e.g. if read access is given to a disk which contains read protected files, then an attacker can read the disk contents and reconstruct the files. More information on Unix security follows in Chapter 7.

Object Reuse (Release of Memory)

A single-processor multiprogramming system may execute several processes at the same time but only one process can 'own' the processor at any point in time. Whenever the operating system deactivates the running process to activate the next, a *context switch* is performed. All data necessary for the later continuation of the execution are saved and memory is allocated for the new process. *Storage residues* are data left behind in the memory area allocated to the new process. If the new process could read such storage residues, the logical separation between processes the operating system should provide has been breached. To avoid this problem, all memory locations that are released could be overwritten with a fixed pattern or the new process could be granted read access only to locations it has already written to.

Buffer Overruns

In a buffer overrun attack, a value is assigned to a variable that is too large for the memory buffer allocated to that variable, so that memory allocated to other variables is overwritten. This method of modifying variables that should be logically inaccessible is further explained in Section 10.4.1.

Backup

A conscientious system manager will perform regular backups. Whoever can lay their hands on the backup media has access to all the data on the tape and logical access control is of no help. Thus, backup media have to be locked away safely to protect the data.

Core Dumps

When a system crashes, it creates a *core dump* of its internal state so that the reasons for the crash can be more easily identified. If the internal state contains sensitive information, such as cryptographic keys, and if core dumps are stored in files that can be read by everyone, an attacker could intentionally crash a multi-user system and look in the core dump for data belonging to other users.

3.6 THE LAYER ABOVE

There is, however, also another conclusion you can draw from the observations just made. As you will discover in this book, it is neither necessary nor sufficient to have a secure infrastructure, be it an operating system or a communications network, to secure an application. An application may take care of its own security requirements. The security services provided by the infrastructure may be irrelevant for the application. The infrastructure cannot defend against attacks from the layer above.

The Fundamental Fallacy of Computer Security

Do not believe that you *must* secure the infrastructure to protect your applications.

Of course, security guarantees provided by the infrastructure may be useful when securing an application. Still, you have to take great care when checking that those guarantees really are relevant for the application. Chapter 18 will illustrate this point.

3.7 FURTHER READING

For a second opinion on computer security there are a number of books you could consult. A very readable introduction to the subject is provided by [195]. At the other end of the spectrum, [8] covers the theoretical elements of computer security. Another comprehensive treatment of information security, with many valuable pointers for further reading, is presented in [191].

Books on the security features of specific operating systems tend to be expensive, concentrate on issues relevant to someone managing such a system, such as drop-down menus and their options, but do not provide much further insight into the way security is implemented. Notable exceptions to this rule, although a little dated, are Park's book on AS/400 [189] and Brown's book on Windows security [47].

3.8 EXERCISES

Exercise 3.1 Conduct a search for further definitions of the security concepts defined in this chapter. Starting points may be the Common Criteria [58] or the websites of the US TCSEC programme[2] and of the Common Criteria Scheme. Many of the major IT companies also have pages on security on their websites.

Exercise 3.2 The *Parkerian hexad* divides security into the six categories: confidentiality, possession or control, integrity, authenticity, availability, and utility. Identify situations where *possession or control* of a container holding some information is a relevant requirement that cannot be captured by the other properties. Identify situations where the *utility* of information is a relevant requirement that cannot be captured by the other properties.

Exercise 3.3 Examine the relationship between unlinkability and anonymity.

Exercise 3.4 Write a short essay discussing the difference between data and information and find your own examples demonstrating that controlling access to data does not necessarily imply controlling access to information.

Exercise 3.5 On the computing system you are using, identify the software components that potentially could incorporate security mechanisms.

Exercise 3.6 A good graphical user interface is an appropriate criterion for purchasing a security product. Discuss.

Exercise 3.7 Look for further examples where a security mechanism in one layer can be bypassed by an attacker who has access to a layer below.

Exercise 3.8 Identify the security perimeters that may be applicable when analyzing personal computer security. In your analysis, consider when it is appropriate to assume that the room the PC is placed in, the PC itself, or some security module within the PC lies within the security perimeter.

[2] Available, at the time of writing, at `http://csrc.nist.gov/publications/secpubs/rainbow/`

Chapter 4

Identification and Authentication

In a secure system you might need to track the identities of users requesting its services. Authentication is the process of verifying a user's identity. You have two reasons for authenticating a user:

- The user identity is a parameter in access control decisions.
- The user identity is recorded when logging security-relevant events in an audit trail.

Chapter 20 will explain why it is not always necessary or desirable to base access control on user identities. There is a much stronger case for using identities in audit logs. This chapter deals with identification and authentication of users as it is standard in current operating systems.

OBJECTIVES

- Visit a familiar security mechanism to learn some general lessons.
- Get an introduction to password protection.
- Appreciate that security mechanisms rely on administrative measures to be effective.
- Understand the dangers when using abstractions in computer security.

4.1 USERNAME AND PASSWORD

Literally, you make your first contact with computer security when you log on to a computer and are asked to enter your *username* and *password*. The first step is called *identification*: you announce who you are. The second step is called *authentication*: you prove that you are who you claim to be. To distinguish this use of the word 'authentication' from other interpretations, we can specifically refer to:

Entity authentication – The process of verifying the identity claimed by some system entity.

Once you have entered your username and password, the computer compares your input against the entries stored in a password file. Login will succeed if you enter a valid username and the corresponding password. If username or password is incorrect, login fails. Usually, the login screen will be displayed again and you can start your next attempt. Some systems keep a count of failed login attempts and prevent or delay further attempts when a certain threshold has been reached. To reduce the chance of an attacker using an unattended machine where another user is logged on, authentication may be demanded not only at the start of a session but also at intervals during the session (*repeated authentication*). You may also choose to *lock* the screen or to close a session automatically if a machine is idle for too long.

> **Lesson**
> Repeated authentication addresses a familiar problem in computer security, known as TOCTTOU (time of check to time of use). The operating system *checks* a user's identity at the start of a session but *uses* the identity to make access control decisions later on during the session.

Once upon a time, you would have entered username and password on a screen containing a friendly welcome message and some information on the system you were about to access. Today, cautious systems managers will not make too much information available to the outside world and replace the welcome message by a warning telling unauthorized persons to stay out. For example, Windows has the option of displaying a *legal notice* dialog box. Users have to acknowledge this warning message before logon can proceed.

Today, most computer systems use identification and authentication through username and password as their first line of defence. For most users, this mechanism has become an integral part of the routine of starting a session on their computer. We thus have a mechanism that is widely accepted and not too difficult to implement. On the other hand, managing password security can be quite expensive and obtaining a valid password is a common way of gaining unauthorized access to a computer system. Let us therefore

examine the actual security of authentication by passwords. First, a password has to be set for the user account; otherwise the attacker can enter unchecked. The attacker may

- *intercept* the password at the time a new user account is created,
- try to *guess* the password,
- get the password from the user through *phishing* or *spoofing*, or by *keyloggers*,
- get the password from the system by compromising the password file or by *social engineering*.

When looking for defences, do not forget the user's role in password protection.

4.2 BOOTSTRAPPING PASSWORD PROTECTION

Passwords are meant to be secrets shared between the user and the system authenticating the user. So, how do you bootstrap a system so that the password ends up in the right places, but nowhere else? In an enterprise, users could be asked to come to an office and collect their password personally. If this is not feasible, the password could be conveyed by mail, email, or phone, or entered by the user on a web page. You now have to consider who might intercept the message and, most importantly, who might actually pick it up. For example, a letter containing the password for an online bank account might be stolen or an impersonator may phone in asking for another user's password. How do you authenticate a remote user when the user has not got a password yet? To address these issues:

- do not give the password to the caller but call back an authorized phone number from your files, e.g. from an internal company address book;
- call back someone else, e.g. the caller's manager or local security officer;
- send passwords that are valid only for a single login request so that the user has to change immediately to a new password (intercepting the first password is thus of limited value);
- send mail by courier with personal delivery;
- request confirmation on a different channel to activate the user account, e.g. enter the password on a web page and send confirmation by SMS (phone).

When setting up a new user account you might tolerate some delay in getting your password. When you are in the middle of an important task and just realize that you have forgotten your password you need an instant remedy. The procedures for resetting a password are pretty much the same as those mentioned above, but now an organization has to staff a hot desk at all times requests may come in. In global organizations such a hot desk has to be available round the clock. Proper security training has to be given to personnel at the hot desk. Thus, password support can become a non-negligible cost factor.

Lesson
Security mechanisms may fail to give access to legitimate users. Your overall security solution should be able to handle such situations efficiently.

4.3 GUESSING PASSWORDS

Password choice is a critical security issue. While you cannot eliminate the risk of an attacker guessing a valid password, you can try to keep low the probability of such an event. An attacker may follow two basic guessing strategies:

- exhaustive search (brute force) – try all possible combinations of valid symbols, up to a certain length;
- intelligent search – search through a restricted name space, e.g. try passwords that are somehow associated with a user such as name, names of friends and relatives, car brand, car registration number, phone number, etc., or try passwords that are generally popular.

A typical example of the second approach is a *dictionary attack*-trying all passwords from a dictionary. So, what are your defences?

- Change default passwords: when systems are delivered, they often come with default accounts such as 'system' with default password 'manager'. This helps the field engineer to install the system, but if the password is left unchanged the attacker has an easy job getting into the system. In this particular example, the attacker even gets access to a privileged account.
- Password length: to thwart exhaustive search, a minimal password length should be prescribed. Standard Unix systems, however, have a *maximal* password length set to eight characters only.
- Password format: mix upper and lower case symbols and include numerical and other non-alphabetical symbols in your password. The size of the password space is at least $|A|^n$ where n is the minimal password length and $|A|$ is the size of the character set used for constructing passwords.
- Avoid obvious passwords: do not be surprised to find out that attackers are equipped with lists of popular passwords and be aware that dictionary attacks have extended the scope of 'obvious' quite substantially. Today, you can find an on-line dictionary for almost every language.

How can the system further help to improve password security?

- Password checkers: as a system manager, you can use tools that check passwords against some dictionary of 'weak' passwords and prevent users from choosing such passwords. This imitates – and pre-empts – dictionary attacks against the system.
- Password generation: some operating systems include password generators producing random but pronounceable passwords. Users are not allowed to pick their own password but have to adopt a password proposed by the system.
- Password ageing: an expiry date for passwords is set, forcing users to change passwords at regular intervals. There may be additional mechanisms to prevent users from choosing previous passwords, e.g. a list of the last ten passwords used. Still, determined users will be able to revert to their favourite password by making a sufficient number of changes until their old password is accepted again.
- Limit login attempts: the system monitors unsuccessful login attempts and reacts by locking the user account completely or at least for a certain period of time to prevent or discourage further attempts. The time the account is locked could be increased in proportion to the number of failed attempts.

Given what has just been said, it would seem that security is highest if users must have long passwords, mixing upper and lower case characters and numerical symbols, probably generated for them by the system, and changed repeatedly. Will this approach really work? Will you get the desired security in practice?

Users are unlikely to memorize long and complicated passwords. Such passwords will be written down on a piece of paper kept close to the computer, where it is most useful both for the legitimate user and a potential intruder. It is a standard step in security reviews to look out for passwords on notes posted on computer terminals. Similar considerations apply when passwords are changed very frequently. Users finding it difficult to comply with the rigour of such a password management scheme may be tempted to use passwords that can be more easily memorized, and therefore more easily guessed. They may revert quickly to their favourite password or make simple and predictable changes to this password. If you have to change the password every month, just add the month (two digits, from 1 to 12, or three characters from JAN to DEC, the choice is yours) to your chosen password and you have passwords that you can remember. Of course, an attacker who has found one of those passwords gets a good idea what to expect next.

Experience shows that people are best at memorizing passwords they use regularly. Hence, passwords work reasonably well in situations where they are entered quite frequently, but not with systems used only occasionally. When changing your password,

it is good advice to type it immediately several times. It is equally good advice not to change passwords before weekends or holidays.

Lesson
Do not look at security mechanisms in isolation. Putting too much emphasis on one security mechanism may actually weaken the system, not least because users will find ways of circumventing security if they cannot do their job properly when the security mechanisms are inappropriate. With passwords, you have observed a trade-off between the complexity of passwords and the faculties of human memory.

4.4 PHISHING, SPOOFING, AND SOCIAL ENGINEERING

Identification and authentication through username and password provide *unilateral authentication*. A user enters a password and the computer verifies the user's identity. But does the user know who has received the password? So far, the answer is no. The user has no guarantees about the identity of the party at the other end of the line.

In phishing and spoofing attacks the user voluntarily sends the password over a channel, but is misled about the end point of the channel. In a *spoofing attack*, the attacker runs a program that presents a fake login screen on a machine and leaves the machine. An unsuspecting user comes to this 'idle' machine and tries to log in. The victim sees what appears to be the normal login menu. When entering username and password, the inputs are collected by the attacker's program. Login is then aborted with a (fake) error message and the spoofing program terminates. Control returns to the operating system which now prompts the user with a genuine login request. The user tries again, succeeds on this second attempt, and may remain completely unaware of the fact that the password has been compromised. What can be done about such a spoofing attack?

- Displaying the number of failed logins may indicate to the user that an attack has happened. If your first login fails but you are told at your second attempt that there have been zero login attempts since your last session, you should become suspicious.
- Trusted path: guarantee that the user communicates with the operating system and not with a spoofing program. For example, Windows has a *secure attention sequence* CTRL+ALT+DEL which invokes the Windows operating system logon screen. The

user should press such a secure attention key when starting a session, even when the logon screen is already displayed.

- Mutual authentication: if users require stronger guarantees about the identity of the system they are communicating with, e.g. in a distributed system, the system could be required to authenticate itself to the user.

Phishing attacks ask users for their password (or other sensitive data) under some false pretence. For example, the message could claim to come from a service you are using, tell you about an upgrade of the security procedures, and ask you to enter your username and password at the new security site that will offer you stronger protection[1]. Users should take care to enter their passwords only at the 'right' site, but in practice it is not always easy to recognize the right site.

The attacker may impersonate a user and trick a system operator into releasing the password to the attacker. Such *social engineering* attacks are more successful when they better understand the psyche of the target [172]. Is this a person that can be bullied? Is this a person that is very supportive of struggling users?

4.4.1 Password Caching

Beyond spoofing attacks, an intruder may have other ways of 'finding' a password. Our description of login has been quite abstract. The password travels directly from the user to the password checking routine. In reality, it will be held temporarily in intermediate storage locations such as buffers, caches, or even a web page. The management of these storage locations is normally beyond the control of the user and a password may be kept longer than the user expects.

This issue is illustrated nicely by a problem encountered by the developers of an early web-based on-line banking service [12]. Web browsers cache information to enable users to scroll back to pages they have recently visited. To use the on-line banking service, you enter your password on a web page. You conduct your business, close the banking application, but do not terminate the browser session. The next user on the terminal can scroll back to the page with your password and log on as you.

As a precaution, it was recommended to exit the browser after the banking transaction. Note that users are now asked to participate in a memory management activity they would otherwise not be involved in. This is another instance of *object reuse* (Section 3.5).

[1] In 2009, criminals succeeded with such an attack in getting access to the Emissions Trading Registry and trade in emission rights.

Lesson
Abstraction is useful and dangerous at the same time. It is useful to discuss password security in abstract terms. You can examine policies on password formats or ageing without knowing how passwords are processed in your IT system. It is dangerous to discuss password security only at such an abstract level. Implementation flaws can compromise the best security policies.

4.5 PROTECTING THE PASSWORD FILE

To verify a user's identity, the system compares the password entered by the user against a value stored in the *password file*. The password can be intercepted by a *keylogger* at the machine at the user's end; it can be intercepted in transit so it should be sent through a secure tunnel (see Chapter 16); and the password file might be compromised. An attacker can directly impersonate a user when the contents of an unencrypted password file are disclosed or when entries in the password file can be modified. Even the disclosure of encrypted passwords may be a concern. Dictionary attacks can then be conducted off-line and protection measures such as limiting the number of unsuccessful login attempts would not come into play. To protect the password file, you have the following options:

- cryptographic protection;
- access control enforced by the operating system;
- a combination of cryptographic protection and access control, possibly with further enhancements to slow down dictionary attacks.

For cryptographic protection, we do not even need an encryption algorithm. A one-way function will do the job. For now, the following working definition will do:

> A one-way function is a function that is relatively easy to compute but significantly harder to undo or reverse. That is, given x it is easy to compute $f(x)$, but given $f(x)$ it is hard to compute x.

Chapter 14 has more details on one-way functions (cryptographic hash functions). One-way functions have been used to protect stored passwords for quite some time [234, p. 91ff]. Instead of the password x, the value $f(x)$ is stored in the password file. When a user logs in and enters a password, say x', the system applies the one-way function f and then compares $f(x')$ with the expected value $f(x)$. If the values match, the user has been successfully authenticated. If f is a proper one-way function, it is not feasible to reconstruct a password x from $f(x)$.

The password file could now be left world-readable but for off-line dictionary attacks. In a dictionary attack, the attacker hashes all words in a dictionary and compares the results against the hashed entries in the password file. If a match is found, the attacker knows that user's password. One-way functions can be chosen to slow down dictionary attacks. This consideration has governed the choice of the one-way function crypt(3) used in Unix systems, which repeats a slightly modified DES encryption algorithm 25 times, using the all-zero block as start value and the password as key [173]. Of course, there is a slight performance penalty for legitimate users at login, but if you optimize the one-way function for speed you also improve the performance of dictionary attacks.

Access control mechanisms in the operating system restrict access to files and other resources to users holding the appropriate privileges. Only privileged users may have write access to the password file. Otherwise, an attacker could get access to the data of other users simply by changing their password, even if it is protected by cryptographic means. If read access is restricted to privileged users, passwords in theory could be stored unencrypted. If the password file contains information that is also required by unprivileged users, then the password file must contain encrypted passwords. However, such a file can still be used in dictionary attacks. A typical example is /etc/passwd in Unix. Therefore, many versions of Unix store enciphered passwords in a file that is not publicly accessible. Such files are called *shadow password files*.

A weak form of read protection is provided by proprietary storage formats. For example, Windows NT did store encrypted passwords in a proprietary binary format. An unsophisticated user will be defeated but a determined attacker will obtain or deduce the information necessary to be able to detect the location of security-relevant data. On its own 'security by obscurity' is not very strong, but it can add to other mechanisms such as password encryption.

There is, however, the danger that a successful breach of such a peripheral defence may be blown out of all proportion. In early 1997, there was a flurry of claims that Windows NT password security had been broken. Sounds really serious, doesn't it? The actual fact behind these stories was the announcement of a program that converted encrypted passwords from binary format to a more readable presentation. Not a big deal after all the excitement.

If you are worried about dictionary attacks but cannot hide the password file, you may consider *password salting*. When a password is encrypted for storage additional information, the *salt*, is appended to the password before encryption. The salt is then stored with the encrypted password. If two users have the same password, they will therefore have different entries in the file of encrypted passwords. Salting slows down dictionary attacks as it is no longer possible to search for the passwords of several users simultaneously.

Lesson
You have seen three security design principles.

- A combination of mechanisms can enhance protection. Encryption and access control are used to guard password files.
- Security by obscurity only protects against casual intruders. Do not place much trust in this strategy.
- If you can, separate security-relevant data from data that should be openly available. In Unix, /etc/passwd contains both types of data. Shadow password files achieve the desired separation.

4.6 SINGLE SIGN-ON

Passwords have separated friend from foe for centuries. In an IT environment, they control access to computers, networks, programs, files, etc. As a user, you would not find it particularly convenient if you had to enter passwords over and over again when navigating through cyberspace to a bit of information. Sitting at your workstation and needing some information from a database held on a server on the network, would you be pleased if you had to

- enter a first password at the workstation,
- enter a second password to get out on the network,
- enter a third password to access the server,
- enter a fourth password to access the database management system,
- enter a fifth password to open a table in the database?

Forget about the problem of potentially having to remember five different passwords and picking the right one at each occasion; having to re-enter the same password five times is bad enough.

A *single sign-on service* solves this problem. You enter your password once. The system may store this password and whenever you have to authenticate yourself again, the system will take the password and do the job for you. Such a single sign-on service adds to your convenience but it also raises new security concerns. How do you protect the stored password? Some of the techniques mentioned previously will no longer work because the system now needs your password in the clear.

Lesson
System designers have to balance convenience and security. Ease of use is an important factor in making IT systems really useful. Unfortunately, many practices which are convenient also introduce new vulnerabilities. This is not the last time the *curse of convenience* will haunt you.

4.7 ALTERNATIVE APPROACHES

If you are dissatisfied with the level of security provided by passwords, what else can you do? The following general options are open to you. As a user, you can be authenticated on the basis of

- something you know,
- something you hold,
- who you are,
- what you do,
- where you are.

Something you know

The user has to know some 'secret' to be authenticated. You have already seen a first example of this mode of authentication. A password is something you know. Another example is the personal identification number (PIN) used with bank cards and similar tokens. As a third example, consider the situation when you make a telephone query about your bank account. The clerk dealing with your call may ask you for further personal information such as home address, date of birth, and name of spouse before releasing any information.

In this mode of authentication, anybody who obtains your secret 'is you'. On the other hand, you leave no trace if you pass your secret to somebody else. When there is a case of computer misuse in your organization where somebody has logged in using your username and password, can you prove your innocence? Can you prove that you did not divulge your password?

> **Lesson**
> A password does not authenticate a person, successful authentication only implies that the user knew a particular secret. There is no way of telling the difference between the legitimate user and an intruder who has obtained that user's password.

Something you hold

The user has to present a physical token to be authenticated. A key that opens a lock is something you hold. A card or an identity tag used to control access to a company's premises are other examples of such a token. Driven by the cost of password management, large organizations have introduced *smart cards* for user authentication.

A physical token can be lost or stolen. As before, anybody who is in possession of the token has the same rights as the legitimate owner. To increase security, physical tokens are often used in combination with something you know: bank cards come with a PIN, or they contain information identifying the legitimate user, such as a photo. However, not

even the combination of mechanisms can totally prevent a fraudster from obtaining the information necessary to impersonate a legitimate user, nor does it stop a user from passing on that information voluntarily.

Who you are

Biometric schemes that use unique physical *characteristics* (traits, features) of a person such as face, fingerprints, iris patterns [77], hand geometry, or possibly even DNA at some time in the future, may seem to offer the ultimate solution for authenticating a person. Biometric schemes are used for two purposes:

- identification – a 1:*n* comparison that tries to identify the user from a database of *n* persons;
- verification – a 1:1 comparison that checks whether there is a match for a given user.

We will use fingerprints as an example to sketch how biometric authentication works. The pattern of ridges in a fingerprint serves as the unique characteristic. First, samples of the user's fingerprint are collected. A *sample* is an analog or digital representation of a biometric characteristic. Biometric *features* are then extracted from the samples and stored as *reference templates*. The features used by a typical fingerprint recognition system are so-called *minutiae*, i.e. positions where ridges end, positions where ridges bifurcate, positions where ridges form a triangle, and the like.

For greater accuracy, several templates may be recorded, possibly for more than one finger. These templates are stored in a secure database. This process is called *enrolment*. The *failure-to-enrol rate* (FER) gives the frequency with which the system fails to enrol a user, e.g. because the skin on the fingers is so worn down that no good quality templates can be obtained.

When the user logs on, a new reading of the fingerprint is taken and compared against the reference template. Authentication by password gives a clear reject or accept at each authentication attempt. In contrast, with biometrics the stored reference template will hardly ever match precisely the template derived from the current measurements. A *matching algorithm* measures the similarity between reference template and current template. The user is accepted if the similarity is above a predefined threshold. Thus, we have to face up to new problems, *false positives* and *false negatives*. Accepting the wrong user (false positive) is clearly a security problem. Rejecting a legitimate user (false negative) creates embarrassment and potential availability problems.

Technology analysis of a biometric scheme is based on (given) databases of biometric samples. This analysis measures the performance of the algorithms extracting and comparing biometric characteristics. By setting the threshold for the matching algorithm,

we can trade off a lower *false match rate* (FMR),

$$\text{FMR} = \frac{\text{number of successful false matches}}{\text{number of attempted false matches}},$$

against a higher *false non-match rate* (FNMR),

$$\text{FNMR} = \frac{\text{number of rejected genuine matches}}{\text{number of attempted genuine matches}},$$

or vice versa (Figure 4.1). Designers of biometric authentication systems have to find the right balance between those two errors. It depends very much on the application where this balance will be found. The *equal error rate* (EER) is given by the threshold value where FMR and FNMR are equal. Currently, the best state-of-the-art fingerprint recognition schemes have an EER of about 1–2%. Iris pattern recognition has a superior performance.

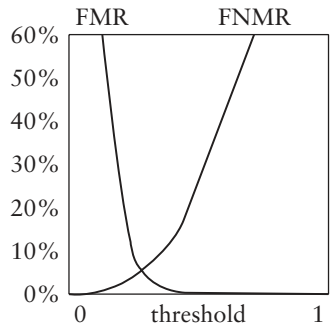


Figure 4.1: Typical Values of FMR and FNMR as a Function of Matching Threshold

Scenario analysis records error rates in actual field trials. It also measures the performance of the fingerprint reader (hardware and software) capturing templates at login time. The *failure-to-capture rate* (FTC) gives the frequency of failing to capture a sample; the *failure-to-extract rate* (FTX) gives the frequency of failing to extract a feature from a sample. The *failure-to-acquire rate* (FTA) gives the frequency of failing to acquire a biometric feature:

$$\text{FTA} = \text{FTC} + \text{FTX} \cdot (1 - \text{FTC}).$$

The *false accept rate* (FAR) for the entire biometric scheme is then

$$\text{FAR} = \text{FMR} \cdot (1 - \text{FTA}),$$

the *false reject rate* (FRR),

$$\text{FRR} = \text{FTA} + \text{FNMR} \cdot (1 - \text{FTA}),$$

and the *false positive identification rate* (FPIR) for a database with n persons

$$\text{FPIR} = (1 - \text{FTA}) \cdot (1 - (1 - \text{FMR})^n).$$

Next, the problem of 'forged' fingers must be considered. Fingerprints, and biometric traits in general, may be unique but they are not secrets. You leave your fingerprints in many places and it has been demonstrated in the past that it is not too difficult to construct rubber fingers that defeat most commercial fingerprint recognition systems [226, 161]. If biometric authentication takes place in the presence of security personnel this might be a minor issue. However, when authenticating remote users additional precautions have to be taken to counteract this type of fraud.

Overall, the industry is just gaining experience with large-scale deployment of biometric schemes. It remains to be seen whether results from experiments conducted in controlled environments are a good indicator of practical performance.

There is a final issue. Will users accept such a mechanism? They may feel that they are treated like criminals if their fingerprints are taken. They may not like the idea of a laser beam scanning their retina.

What you do

People perform some mechanical tasks in a way that is both repeatable and specific to the individual. Hand-written signatures have long been used in banking to confirm the identity of users when signing cheques and credit card payment slips. Forgeries are relatively easy to perpetrate for skilled criminals. For greater security, users could sign on a special pad that measures attributes like writing speed and writing pressure. On a keyboard, typing speed and intervals between key strokes are being used to authenticate individual users. As before, the authentication system has to be set up so that false positives and false negatives are reduced to levels acceptable for the intended application.

Where you are

When you log on, the system may also take into account where you are. Some operating systems already do so and grant access only if you log on from a certain terminal. For example, a system manager may only log on from an operator console but not from an arbitrary user terminal. Similarly, as a user you may be only allowed to log on from the workstation in your office. Decisions of this kind will be even more frequent in mobile and distributed computing. If the precise geographical location has to be established during authentication, a system may use the services of the Global Positioning System (GPS). Identifying the location of a user when a login request is made may also help to resolve later disputes about the true identity of that user.

4.8 FURTHER READING

The history of Unix password security is told in [173], where quite interesting statistics on typical password choices can be also found, and is taken further in [93]. Practically every book on computer security contains extensive advice on proper password choice and on the importance of password security. You can find quite a number of password *crackers* on the Internet. Analyzing one of these programs will give you a good idea of the type of passwords current crackers search for and of the size and sophistication of the dictionaries they are using. Running such a program without explicit authorization may bring you into conflict with the disciplinary rules of your organization and with the criminal law of many countries.

4.9 EXERCISES

Exercise 4.1 Check the password scheme on your own computer system. Are there any rules on password length, password format, or password expiry? How are passwords stored in your system?

Exercise 4.2 Assume that you are only allowed to use the 26 letters of the alphabet to construct passwords.

- How many different passwords are possible if a password is at most $n = 4, 6, 8$ characters long and there is no distinction between upper case and lower case characters?
- How many different passwords are possible if a password is at most $n = 4, 6, 8$ characters long and passwords are case-sensitive?

Exercise 4.3 Assume that passwords have length 6 and all alphanumerical characters, upper and lower case, can be used in their construction. How long will a brute force attack take on average if

- it takes one tenth of a second to check a password?
- it takes a microsecond to check a password?

Exercise 4.4 Assume that you are only allowed to use the 26 letters of the alphabet to construct passwords of length n. Assume further that you are using the same password in two systems where one accepts case-sensitive passwords but the other does not. Give an upper bound at the number of attempts required to guess the case-sensitive version of a password.

Exercise 4.5 You are shipping WLAN access points. Access to these devices is protected by password. What are the implications of shipping all access points with the same default password? What are the implications of shipping each access point with its individual password?

Exercise 4.6 Passwords are entered by users and checked by computers. Thus, there has to be some communications channel between user and computer. So far we have taken a very abstract view of this channel and assumed that it exists and that it is adequately secure. When is this assumption justified? When is it not justified?

Exercise 4.7 If you are required to use several passwords at a time, you may consider keeping them in a *password book*. A password book is a protected file containing your passwords. Access to the password book can again be controlled through a *master* password. Does such a scheme offer any real advantages?

Exercise 4.8 There exists a time–memory trade-off in password guessing described in [115]. Let N be the number of possible passwords. In a precomputation step using N trial encryptions, a table with $N^{2/3}$ entries is constructed. If you later want to find a given encrypted password, you need $N^{2/3}$ trial encryptions. How much memory space do you need when passwords of length 6 are chosen from a 5-bit character set? How quickly will you find the password if a trial encryption takes one millisecond?

Exercise 4.9 Conduct a security analysis of authentication based on personal information such as date of birth and other details from a person's private life.

Exercise 4.10 Conduct a survey of commercially available biometric authentication systems. What are the false acceptance, false rejection, and equal error rates for those systems?

Chapter 5

Access Control

You have now logged on to the system. You create new files and want to protect them. Some of them may be public, some only intended for a restricted audience, and some may be private. You need a language for expressing your intended access control policy and you need mechanisms that enforce access control. This chapter introduces the vocabulary for talking about access control. Chapters 11 and 12 will look into specific access control policies.

OBJECTIVES

- Introduce the fundamental model of access control.
- Look at a few sets of access operations and warn of the danger of substituting your intuition for the actual definition of terms.
- Present essential access control structures, independent of specific security policies.
- Define partial orderings and lattices, mathematical concepts often used when expressing security policies.

5.1 BACKGROUND

Before immersing yourself in the details of access control, consider the way computer systems – and the use of computer systems – have developed over the last few decades. Computer systems manipulate data and mediate access to shared resources such as memory and printers. They have to provide access control to data and resources, although primarily for reasons of integrity and not so much for confidentiality. Traditional multi-user operating systems offer general infrastructure services to a considerable variety of users. By their very nature, these operating systems have simple and generic access operations and are not concerned with the meaning of the files they handle. Modern desktop operating systems support individual users in performing their job. In this scenario, you find more complex application-specific access operations. Users are not interested in the lower-level details of the execution of their programs. Not surprisingly, it may be quite difficult to map their high-level security requirements to low-level security controls. In a nutshell, you are witnessing the transition from *general purpose* computer systems to (flexible) *special purpose* computer systems. Keep this trend in mind when comparing the different access control models covered in this book.

5.2 AUTHENTICATION AND AUTHORIZATION

To discuss access control, we first have to develop a suitable terminology. The very nature of 'access' suggests that there is an active entity, a *subject* or *principal*, accessing a passive *object* with some specific *access operation*, while a *reference monitor* (see Chapter 6) grants or denies access. Figure 5.1 captures this view of access control.

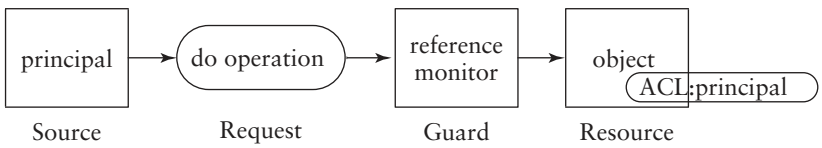


Figure 5.1: The Fundamental Model of Access Control

Access control then consists of two steps, *authentication* and *authorization*. In the words of Lampson *et al.* [145]:

> If *s* is a statement *authentication* answers the question 'Who said *s*?' with a principal. Thus principals make statements; this is what they are for. Likewise, if *o* is an object *authorisation* answers the question 'Who is trusted to access *o*?' with a principal.

The security literature has two terms for the entity making an access request, *subject* and *principal*, but does not distinguish between those two concepts in a consistent way. The relationship between subjects and principals on one side, and the human users of a

computer system on the other can further confuse the picture. To separate the meaning of these two terms we take our cue from earlier work on operating system security:

> Subjects operate on behalf of *human users* we call *principals*, and access is based on the principal's name bound to the subject in some unforgeable manner at authentication time. Because access control structures identify *principals*, it is important that principal names be globally unique, human-readable and memorable, *easily and reliably associated with known people* [98].

This quote reflects traditional *identity-based access control* where security policies ultimately refer to human users. This is still the most common type of access control supported by commercial operating systems but, as discussed in Chapter 20, no longer the only paradigm in access control. In our general framework we refer to *principals* when discussing security policies and to *subjects* when discussing the operational systems that should enforce a security policy. This terminology is consistent with the traditions quoted above.

> **Definition.** A *principal* is an entity that can be granted access to objects or can make statements affecting access control decisions [99]. A *subject* is an active entity within an IT system.

For the purpose of access decisions, subjects have to be bound to principals. When a subject requests access to a protected object the reference monitor checks whether the principal bound to the subject has the right to access the object. We might thus say that the subject 'speaks for' a principal. A typical example of a principal in an operating system is a *user identity*. The principals permitted to access a given object could be stored in an *access control list* (ACL) attached to the object (Figure 5.1). A typical example of a subject is a process running under a user identity (the principal). However, principals need not represent human users or attributes of human users. In Java one of the main parameters for access control is the *code source* (Section 20.5) and the relation between principals and subjects is defined as follows [104]:

> The term principal represents a *name* associated with a subject. Since subjects may have multiple names, a subject essentially consists of a collection of principals.

The objects of access control are files or resources, such as memory, printers, or nodes in a computer network. There is not meant to be a clear distinction between subjects and objects in the sense that every entity in the system has to be either a subject or an object. Depending on circumstances, an entity can be a subject in one access request and an object in another. The terms *subject* and *object* merely distinguish between the active and passive party in an access request.

Principals and objects present two options for focusing control. You can either specify

- what a principal is allowed to do, or
- what may be done with an object.

This is an instance of the first design principle from Section 3.4.1. Operating systems provide an *infrastructure* for managing files and resources, i.e. objects. In such a setting, you will encounter mostly access control mechanisms taking the second approach. On the other hand, *application*-oriented IT systems, such as database management systems, offer services directed at the end user. Such systems may well incorporate mechanisms for controlling the actions of principals.

5.3 ACCESS OPERATIONS

Depending on how you look at a computer system, access operations vary from reading and writing to physical memory to method calls in an object-oriented system. Comparable systems may use different access operations and, even worse, attach different meanings to operations of the same name. We will now examine some typical sets of access operations taken from important early contributions in this area.

5.3.1 Access Modes

On the most elementary level, a subject may observe an object or alter an object. There are thus two *access modes*:

- Observe – look at the contents of an object;
- Alter – change the contents of an object.

Although most access control policies could be expressed in terms of Observe and Alter, such policy descriptions will often be too far removed from the application-level operations, making it difficult to check whether the correct policy has been implemented. Hence, you usually find a richer set of access operations.

5.3.2 Access Rights of the Bell–LaPadula Model

At the next level of complexity, you find the *access rights* of the Bell–LaPadula security model discussed in Chapter 11, and the *access attributes* of the Multics operating system [187], two of the milestones in the history of computer security.

The Bell–LaPadula model has four access rights: execute, read, append (sometimes also referred to as blind write), and write. Figure 5.2 gives the relation between these access rights and the two basic access modes Observe and Alter.

	execute	append	read	write
Observe			X	X
Alter		X		X

Figure 5.2: Access Rights in the Bell–LaPadula Model

To understand the rationale for this definition, consider how a multi-user operating system controls access to files. A user has to *open* a file before access is granted. Usually, files can be opened for read access or for write access. In this way, the operating system can avoid conflicts such as two users simultaneously writing to the same file. For reasons of efficiency, write access usually includes read access. For example, a user editing a file should not be asked to open it twice, once to read and once to write. Hence, it is meaningful to define the write right so that it includes Observe and Alter mode.

Few systems actually implement the append operation. Allowing users to alter an object without observing its content is in general not a useful operation. Audit logs are one instance where the append right is useful. A process writing to the log file has no need to read the file, and probably should not read it at all.

With respect to the execute right, which includes neither Observe nor Alter mode, you may ask how a computer could execute a program without reading the program's instructions. You would of course be right and the Multics execute attribute indeed requires execute and read rights. However, there exist operations where the contents of an object are used in an execution without being read. Consider a cryptographic engine holding a master key in a special tamper-resistant register (Figure 5.3). There is physically no way the master key can be read out but access control rules may govern who is allowed to use this key for encryption. We can invoke this key without reading it and the execute right is just what is needed to address such a situation.

Lesson
Beware of using your own intuition when interpreting access operations someone else has defined!

The Multics operating system distinguishes between access attributes for files (known as data segments) and access attributes for directories. It is common practice to interpret a given set of access rights differently depending on the type of object. The terms 'read', 'write' and 'execute' are used again to name access attributes, but not in exactly the same

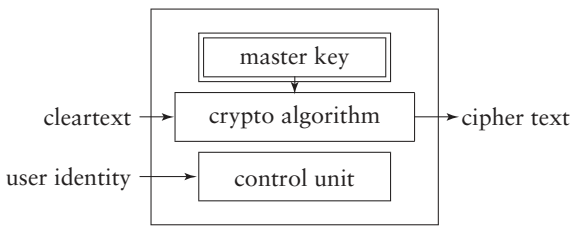


Figure 5.3: A Cryptographic Engine

files:		directories:	
read	r	status	r
execute	e,r	status and modify	w
read and write	w	append	a
write	a	search	e

Figure 5.4: Access Attributes in Multics

meaning as in the Bell–LaPadula model. To maintain some clarity in our presentation, we will denote the Bell–LaPadula access rights by e, r, a, w. Figure 5.4 maps the Multics access attributes to Bell–LaPadula access rights.

Access rights in current operating systems are covered in Chapter 7 on Unix and Chapter 8 on Windows.

5.3.3 Administrative Access Rights

In the Unix operating system access control policies are expressed in terms of three operations:

- read – reading from a file;
- write – writing to a file;
- execute – executing a (program) file.

As with Multics, Unix write access does not imply read access. When applied to a directory, the access operations take the following meaning:

- read – list directory contents;
- write – create or rename a file in the directory;
- execute – search the directory.

Thus, Unix controls who can create and delete files by controlling write access to the file's directory. The access rights specified for a file are changed by modifying the file's entry in its directory. Other operating systems have a special operation for deleting files.

Similarly, Unix defines the right to modify access rights to files and directories via access rights to the parent directory. Other operating systems have special operations for modifying access rights. Operations for manipulating a subject's access rights tend to be called grant and revoke when the subject's rights are modified by some other party, and assert and deny when the subject changes its own access rights. Operations of this nature are of interest in *delegation* policies, where one subject invokes another subject and the rights of the invoked subject have to be established.

Windows, for example, has specific access rights for modifying security settings. The *standard permissions* include:

- delete;
- write DACL (modify access control list);
- write owner (modify owner of a resource).

5.4 ACCESS CONTROL STRUCTURES

Next, we have to state which access operations are permitted. We have to decide on the structures to use for capturing security policies, whilst facing two competing requirements:

- The access control structure should help to express your desired access control policy.
- You should be able to check that your intended policy has been captured correctly.

In the following, let S be a set of subjects, O a set of objects, and A a set of access operations. There is no need yet to be more specific about any of these sets.

5.4.1 Access Control Matrix

At a basic level, access rights can be defined individually for each combination of subject and object simply in the form of an *access control matrix* (*table*)

$$M = (M_{so})_{s \in S,\, o \in O} \quad \text{with} \quad M_{so} \subseteq A.$$

The entry M_{so} specifies the set of access operations subject s may perform on object o. This approach goes back to the early days of computer security [144]. Access control matrices are also known as *access permission matrices*. The Bell–LaPadula model employs an access control matrix to model the discretionary access control policies of the Orange Book (Section 11.2). Figure 5.5 gives a simple example of an access control matrix for two users and three files.

- bill.doc may be read and written to by Bill, while Alice has no access at all.
- edit.exe can be executed both by Alice and Bill but otherwise they have no access.
- fun.com can be executed and read by both users, while only Bill can write to the file.

The access control matrix is an abstract concept and not very suitable for direct implementation if the number of subjects and objects is large or if the sets of subjects and objects change frequently. In such scenarios, intermediate levels of control are preferable.

	bill.doc	edit.exe	fun.com
Alice	-	{execute}	{execute, read}
Bill	{read, write}	{execute}	{execute, read, write}

Figure 5.5: An Access Control Matrix

5.4.2 Capabilities

There are two fundamental options for implementing an access control matrix. Access rights can be kept with the subjects or with the objects. In the first case, every subject is given a *capability*, an unforgeable token that specifies this subject's access rights. This capability corresponds to the subject's row in the access control matrix. The access rights of our previous example given as capabilities are as follows.

Alice's capability: edit.exe: execute; fun.com: execute, read;
Bill's capability: bill.doc: read, write; edit.exe: execute; fun.com: execute, read, write;

Typically, capabilities are associated with discretionary access control (Section 5.5). When a subject creates a new object, it can give other subjects access to this object by granting them the appropriate capabilities. Also, when a subject (process) calls another subject, it can pass on its capability, or parts thereof, to the invoked subject.

Capabilities are by no means a new concept, but hitherto they have not become a widely used security mechanism. This is mainly due to the complexity of security management and to the traditional orientation of operating systems towards managing objects:

- It is difficult to get an overview of who has permission to access a given object.
- It is difficult to revoke a capability – either the operating system has to be given the task or users have to keep track of all the capabilities they have passed on. This problem is particularly awkward when the rights in the capability include the transfer of the capability to third parties.

The advent of distributed systems has rekindled interest in capability-based access control where security policies have to deal with users roaming physically or virtually between nodes in a computer network.

When you decide to employ capabilities, you also have to give some consideration to their protection. Where do you store the capabilities? If capabilities are only used within a single computer system, then it is feasible to rely only on integrity protection by the operating system (see Chapter 6). When capabilities travel over a network, you also need cryptographic protection (see Chapter 14).

5.4.3 Access Control Lists

An *access control list* stores the access rights to an object as a list with the object itself. An ACL therefore corresponds to a column of the access control matrix and states who may access a given object. The entries in an ACL are called *access control entries* (ACEs). ACLs are a typical security feature of commercial operating systems. The access rights of our previous example, given in the form of ACLs, are as follows.

ACL for bill.doc:	Bill: read, write;
ACL for edit.exe:	Alice: execute; Bill: execute;
ACL for fun.com:	Alice: execute, read; Bill: execute, read, write.

Management of access rights based only on individual subjects can be rather cumbersome. It is therefore common practice to place users in *groups* and to derive access rights also from a user's group. The Unix access control model is based on simple ACLs each having three entries that assign access rights to the principals *user*, *group*, and *others* (Section 7.5).

ACLs are a fitting concept for operating systems that are geared towards managing access to objects. If, however, you want an overview of the permissions given to an individual user, e.g. to revoke that user's permissions, you face a laborious search through all ACLs.

No matter how you implement the access control matrix, managing a security policy expressed by such a matrix is a complex task in large systems. In particular, it is tedious and error-prone to establish that all entries in such a matrix are as desired. Moreover, access control based only on subjects and objects supports a rather limited range of security policies. Further information, which may be appropriately included in an access control decision, may refer to the program the subject invokes to access the object. This is not a novel idea at all, as you can see from the following comment on access control in the Titan operating system, developed in Cambridge in the early 1960s [179]:

> In particular, it was possible to use the identity of a program as a parameter for access-control decisions as well as, or instead of, the identity of the user, a feature which Cambridge people have ever since regarded as strange to omit.

Lesson
Don't think that new technologies necessarily create new security problems. More often than not, the 'new' problems are reincarnations of old problems and the principles for their solution are already known.

5.5 OWNERSHIP

When discussing access control, we must state who is in charge of setting security policies. There are two fundamental options:

- We can define an *owner* for each resource and let the owner decree who is allowed to have access. Policies defined by the owner are called *discretionary* because access control is at the discretion of the owner.

- A system-wide policy decrees who is allowed to have access. Policies imposed by the system are called *mandatory*.

Most operating systems support the concept of *ownership* of a resource and consider ownership when making access control decisions. They may include operations that redefine the ownership of a resource. Historically, *discretionary access control* and *mandatory access control* referred specifically to policies used in the defence sector. The definitions in the Orange Book [224] follow this tradition.

Access control based on user identities was called discretionary.[1] These policies happened to be at the discretion of the owner. We may highlight the focus on user identities by adopting the term *identity-based access control* (IBAC) instead. Referring to individual users in a policy works best within closed organizations. In general, IBAC incurs an identity management overhead and does not scale well.

Access control based on policies that refer to security labels for classified documents, e.g. confidential, top secret, was called mandatory access control. There were strict rules everyone had to follow. There are few applications for mandatory access control outside the defence sector. Discretionary and mandatory access control have survived in computer security textbooks, but not very much in the wild.

5.6 INTERMEDIATE CONTROLS

In computer science, problems of complexity are solved by indirection (David Wheeler). This principle can also be applied to access control. We introduce intermediate layers between users and objects to represent policies in a more manageable fashion.

5.6.1 Groups and Negative Permissions

The following discussions will be built around a simple example. A teacher wants to give students access to course material. Instead of putting all students individually into an ACL for each piece of course material, the teacher could put all students into a *group* and put this group into the respective ACLs.

Groups are thus a means of simplifying the definition of access control policies. Users with similar access rights are collected in groups and groups are given permission to access objects. Some security policies demand that a user can be the member of one group only, others allow membership in more than one group.

Figure 5.6 captures a situation where all access permissions can be mediated through group membership. Often, security policies have *exceptions* from general rules where

[1] In fact, discretionary access control was characterized as access control for policies captured in an access control matrix.

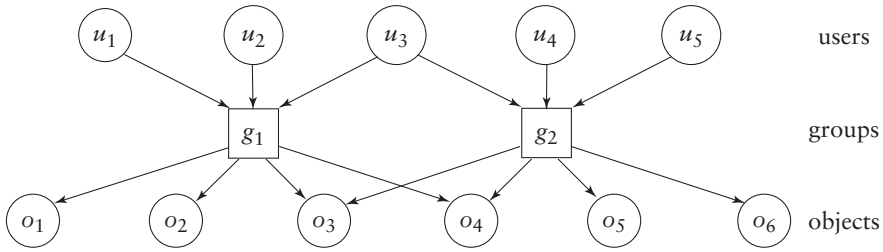


Figure 5.6: Groups Serve as an Intermediate Access Control Layer

some user should get a permission for an object directly, or where a user should be denied a permission that normally follows from membership in some group. A *negative permission* is an entry in an access control structure that specifies the access operations a user is not allowed to perform. In Figure 5.7, user u_1 is denied access to object o_1 and user u_3 is directly granted access to object o_5.

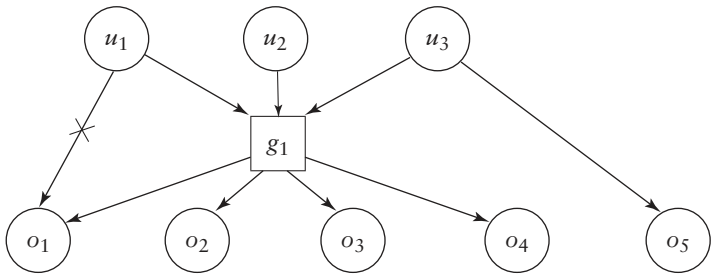


Figure 5.7: Access Control with Negative Permissions

The negative permission given to user u_1 contradicting the positive permission given to group g_1 is an example of a *policy conflict*. When specifying a policy, you have to know how conflicts will be resolved by the reference monitor. If policies are defined by ACLs, a simple and widely used algorithm just processes the list until the first entry matching the principal given in the access request is found, and makes the decision based on this information only. Any conflicting entries later in the list are ignored.

5.6.2 Privileges

A policy could refer to the operations a user is allowed to execute. We let the term *privilege* stand for the right to execute certain operations. This follows frequent usage of this term, where privileges are associated with operating system functions; there can be privileges for system administration, backup, mail access, or network access. Privileges can be viewed as an intermediate layer between subjects and operations (Figure 5.8).

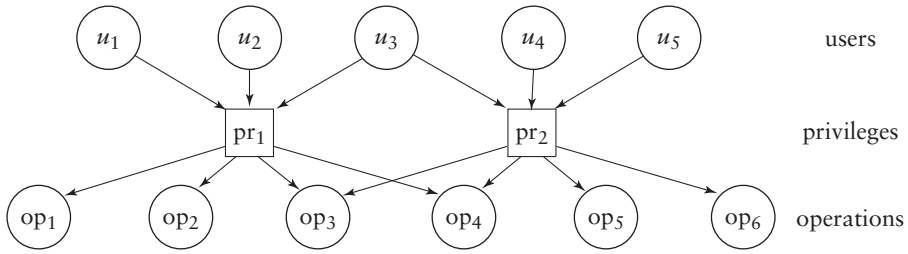


Figure 5.8: Privileges as an Intermediate Layer between Subjects and Operations

5.6.3 Role-Based Access Control

At a technical level, a *role* can be defined as a collection of application-specific operations (*procedures*). Subjects derive their access rights from the role they are performing. Role-based access control (RBAC) introduces roles, procedures, and possibly *data types* as intermediate layers between subjects and objects.

- *Roles*: a role is a collection of procedures. Roles are assigned to users. A user assigned to a role can execute the procedures defined for that role. A user can have more than one role and more than one user can have the same role [200].
- *Procedures*: procedures are 'high-level' access control methods with a more complex semantic than read or write. Procedures can only be applied to objects of certain data types. As an example, consider a funds transfer between bank accounts.
- *Data types*: each object is of a certain data type and can be accessed only through the procedures defined for this data type. Controlling access to an object by restricting the procedures that may access this object is a general programming practice. It is a fundamental concept in the theory of abstract data types.

When a user logs in, the process started derives its permissions from the roles assigned to that user. Frequently, a user has to take an explicit action to activate the roles. The least-privilege principle suggests that only roles necessary for the current task should be activated.

In the example of Section 5.6.1 the teacher could create a role *Student* for the students on his course and assign the permission to read course material to this role. A role *Teacher* could be given the permission to edit the course material.

Role hierarchies define relationships between roles. A senior role can do anything the junior role can do. This helps policy administration. The definition of the senior role need not list all the procedures the senior role has permission to execute; it suffices to state the relationship to junior roles and add those procedures that may only be executed by a

user in the senior role. In our example, a junior role *Teaching Assistant* could be given permission to edit material for exercise classes. The role hierarchy must not be confused with the hierarchy of positions (superior–subordinate) in an organization. These two hierarchies need not always correspond.

Separation of duties is an important general security principle. There exist tasks where certain steps must be executed by distinct users. For example, in a purchasing department the person approving a payment must not be the person who issued the purchase order. The administrator in charge of assigning access rights must not exercise those rights himself. There exist numerous flavours of static and dynamic separation-of-duties policies. In a static separation-of-duties policy, the roles that may be assigned to a user are fixed and have to take into account separation-of-duties requirements. For example, a user can either issue purchase orders or approve payments. In dynamic separation-of-duties policies, the roles that may be assigned to a user depend on the current task. In this case, the user who has issued a particular purchase order may not approve payment for that order, but may approve payments for orders issued by someone else.

The National Institute of Standards and Technology (NIST) has published a widely used classification of RBAC levels [198]. The levels are defined incrementally. Each level includes the features of the previous level.

- Flat RBAC: users are assigned to roles, permissions are assigned to roles, users get permissions via role membership; *user-role reviews* are supported. In our example, assume there is a student *Alice*. The user-role review would tell whether she is just a *Student* or has been appointed *Teaching Assistant*.
- Hierarchical RBAC: adds support for role hierarchies. In our example, the *Teacher* role can be defined as senior to *Teaching Assistant*.
- Constrained RBAC: adds support for separation-of-duties policies. In our example there could be a rule that students cannot be teaching assistant on a course they are taking.
- Symmetric RBAC: adds support for *permission-role reviews*. Conducting such a review can be difficult in large distributed systems. A permission-role review in our example might tell which roles have write access to course material.

In many organizations roles are a suitable intermediate layer for setting security policies. There exist 'roles' with well-defined tasks. Staff members are assigned to those roles so it is natural to define what a user in a role must and should be able to do. However, this does not imply that 'role' as a concept will be used precisely in the way it has been defined above. There exist other approaches for using organizational roles when constructing access control systems.

> The term RBAC itself does not have a generally accepted meaning, and it is used in different ways by different vendors and users [198].

5.6.4 Protection Rings

Protection rings are a particularly simple example of an intermediate layer of hardware-based access control for subjects and objects. Each subject (process) and each object is assigned a number, depending on its 'importance'. In a typical example, processes are assigned one of the following numbers:

0 – operating system kernel;
1 – operating system;
2 – utilities;
3 – user processes.

Access control decisions are made by comparing the subject's and object's numbers. (The outcome of the decision depends on the security policy you try to enforce using protection rings.) These numbers correspond to concentric *protection rings*, with ring 0 in the centre giving the highest degree of protection (Figure 5.9). If a process is assigned the number *i*, we say the process 'runs in ring *i*'.

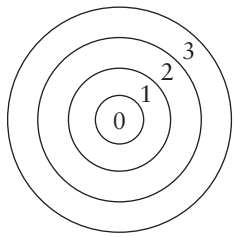


Figure 5.9: Protection Rings

Protection rings were introduced for integrity protection. They had already been used in the Multics operating system, and special hardware was developed to support this security mechanism [207]. Current processors provide similar features at the machine language level. Unix uses two levels with root and operating system running in ring 0 and user processes running in ring 3. An operating system with a microkernel could assign software components to protection rings as follows:

- microkernel runs in ring 0;
- process manager runs in ring 1;
- all other programs run in ring 3.

Memory locations containing sensitive data, such as the operating system code, can only be accessed by processes that run in ring 0 or 1. A typical policy based on protection rings is given in Section 6.3.5. More examples can be found in [187, Chapter 4] and [191, Section 7.2].

5.7 POLICY INSTANTIATION

When *developing* software you will rarely be in a position to know your eventual users. At this stage, security policies cannot refer to specific user identities, but can perhaps refer to generic *placeholder* principals such as *Teacher* and *Student* in the running example, or *owner*, *group*, *others* in Unix, or *friends* in a social network. A customer *deploying* the software would know its authorized users and can thus instantiate the generic policy with the appropriate user identities. The reference monitor has to resolve the placeholder principals to concrete user identities when processing an actual request.

5.8 COMPARING SECURITY ATTRIBUTES

When evaluating a security policy, the reference monitor compares the access rights granted to the subject with the access rights demanded by the policy. The most basic comparison is an equality check, e.g. between the user identity a process is running under and the user identity in an ACE. Real-life policies may use security attributes with a richer set of comparison operators.

5.8.1 Partial Orderings

Protection rings are a very simple example where we can decide for any two rings i and j which is the innermost. In general, this need not be the case. Consider an extension of the example of Section 5.6.1. The department creates a group *Year_1* for first year students to manage access for resources specifically dedicated to them. There is also a group *Year_2* for second year students, *Year_3* for third year students, etc. The group of first year students would be contained in the group of all students, but there is no such relation between groups *Year_1* and *Year_2*. The best we can aim for is a *partial ordering*.

Definition. A partial ordering $\leq$ ('less or equal') on a set (of security levels) L is a relation on $L \times L$ which is:

- *reflexive* – for all $a \in L$, $a \leq a$ holds;
- *transitive* – for all $a, b, c \in L$, if $a \leq b$ and $b \leq c$, then $a \leq c$;
- *antisymmetric* – for all $a, b \in L$, if $a \leq b$ and $b \leq a$, then $a = b$.

If two elements $a, b \in L$ are not comparable, we write $a \not\leq b$.

Typical examples of partial orderings are:

- $(\mathcal{P}(X), \subseteq)$, the power set of a set X with the subset relation as partial ordering.
- $(N, |)$, the natural numbers with the 'divides' relation as partial ordering.
- The strings over an alphabet Σ with the prefix relation as a partial ordering. A string β is a prefix of a string α if there exists a string γ such that we can write $\alpha = \beta\gamma$. In this case, we write $\beta \leq \alpha$.

Hasse diagrams are a graphical representation of partially ordered sets (posets). A Hasse diagram is a directed graph in which the nodes are the elements of the set. The edges in the diagram give a 'skeleton' of the partial ordering. That is, for $a, b \in L$ we place an edge from a to b if and only if

- $a \leq b$ and $a \neq b$, and
- there exists no $c \in L$ such that $a \leq c \leq b$ and $a \neq c$, $c \neq b$.

With this definition, $a \leq b$ holds if and only if there is a path from a to b. Edges in the graph are drawn to point upwards. The Hasse diagram for the partially ordered set $(\mathcal{P}(\{a, b, c\}), \subseteq)$ is given in Figure 5.10.

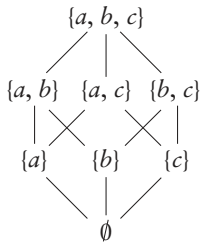


Figure 5.10: The Poset (Lattice) $(\mathcal{P}(\{a, b, c\}), \subseteq)$

5.8.2 Abilities in the VSTa Microkernel

The capabilities of the VSTa microkernel may illustrate the use of partial orderings. As they are not quite capabilities as defined in Section 5.4.2, let us use *abilities* instead. An ability is a finite string of positive integers. To separate integers a dot is placed in front of each integer. So, an ability is a string $.i_1.i_2.\cdots.i_n$ for some value n where $i_1, \ldots, i_n$ are integers. There is no limit on the length n of such a string. Indeed, n may be equal to 0. Examples of abilities are .1.2.3, .4, or .10.0.0.5.

Abilities ordered through the prefix relation constitute a partial ordering. In our running example, the department might assign ability .3.1.101 to the group of students on course CS101. Course material for CS101 might be labelled with .3.1.101, general material for first year students with .3.1, general second year material with .3.2, and .3 would be used for general material for all students. These abilities are related by $.3 \leq .3.1 \leq .3.1.101$ but $.3.1 \not\leq .3.2$.

For a policy that grants access if the object's label is a prefix of the subject's label, CS101 students will get access to their own course material, year 1 material, and general information for students.

For a moment, consider the dual of the above policy. Access is granted if the subject's ability is a prefix of the object's ability. In this case, the ability '.', a dot followed by no

integers, defines a superuser who has access to all objects, as the empty string ε is the prefix of any ability. Thus, by not assigning an ability to a subject you would grant that subject access to all objects.

Lesson
Access control algorithms compare attributes of subjects and objects. You always have to check what happens if an attribute is missing. Fail-safe behaviour would suggest that access should be denied. Often this is not the case and you could be in for an unpleasant surprise.

5.8.3 Lattice of Security Levels

Returning to the original policy in our example, if two groups of students should have access to a document, the department could use the longest common prefix of the abilities assigned to the two groups to label the document. For example, if *Year_1* and *Year_2* should get access to a document, ability .3 could be used as a label. On the other hand, if we have two objects labelled (say) with .3.1 and .3.2 and want to assign a label to a subject that has access to both, we could not do it in our current system.

In general, given the standard confidentiality policy where a subject may observe an object only if the subject's security level is higher than the object's security level, we may wish to have unique answers to the following two questions:

- Given two objects at different security levels, what is the minimal security level a subject must have to be allowed to read both objects?
- Given two subjects at different security levels, what is the maximal security level an object can have so that it still can be read by both subjects?

The mathematical structure that allows us to answer these two questions exists. It is called a *lattice*. Formally, it can be defined as follows.

Definition. A lattice $(L, \leq)$ consists of a set L and a partial ordering $\leq$. For every two elements $a, b \in L$ there exists a *least upper bound* $u \in L$ and a *greatest lower bound* $l \in L$, i.e.

$$a \leq u, b \leq u, \text{ and } \forall v \in L : (a \leq v \wedge b \leq v) \Rightarrow (u \leq v),$$
$$l \leq a, l \leq b, \text{ and } \forall k \in L : (k \leq a \wedge k \leq b) \Rightarrow (k \leq l).$$

In security, we say 'a is dominated by b' or 'b dominates a' if $a \leq b$. The security level dominated by all other levels is called *System Low*. The security level dominating all other levels is called *System High*. For example, the partially ordered set $(\mathcal{P}(\{a, b, c\}), \subseteq)$ of Figure 5.10 is a lattice with the empty set $\emptyset$ as *System Low* and the set $\{a, b, c\}$ as *System High*.

Whenever you meet a security system where security attributes are compared in some way you are likely to find that it is convenient if these attributes form a lattice. It is not necessary to understand lattices to grasp the essential facts of computer security. Nonetheless, it helps to understand lattices when reading the research literature on this subject.

5.8.4 Multi-level Security

Much security research in the 1970s and 1980s was driven by the demands of protecting classified information. Research built on the existing policies regulating physical access to classified documents. Documents were assigned *security levels*. A user's *clearance* dictated which documents the user could obtain. The *mandatory access control* policies and *multi-level security* policies of the Orange Book make use of security levels and adapt these policies to IT systems. In their most elementary version, these policies refer to a linearly ordered hierarchy of four security levels: *unclassified*, *confidential*, *secret*, and *top secret* (Figure 5.11).

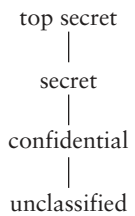


Figure 5.11: Security Levels in Linear Order

With a linear ordering of security levels you can only express a limited set of security policies. You could not, for example, restrict access to documents relating to a secret project X just to the people working on X. Anyone at level *secret* would have access. To be able to state such *need-to-know policies* controlling access to the resources of specific projects, the following lattice of security levels was introduced.

- Take H, a set of *classifications* with a hierarchical (linear) ordering $\leq_H$.
- Take a set C of *categories*, e.g. project names, company divisions, academic departments, etc. A *compartment* is a set of categories.
- A *security label (security level)* is a pair (h, c), where $h \in H$ is a security level and $c \subseteq C$ is a compartment.
- The partial ordering $\leq$ of security labels is defined by

$$(h_1, c_1) \leq (h_2, c_2) \text{ if and only if } h_1 \leq_H h_2 \text{ and } c_1 \subseteq c_2.$$

Figure 5.12 illustrates this construction. There are two hierarchical levels, *public* and *private*, and two categories, *personnel* (PER) and *engineering* (ENG). In the ensuing lattice, the following relations hold, for example:

(*public*, {PER}) $\leq$ (*private*, {PER}),
(*public*, {PER}) $\leq$ (*public*, {PER, ENG}),
(*public*, {PER}) $\not\leq$ (*private*, {ENG}).

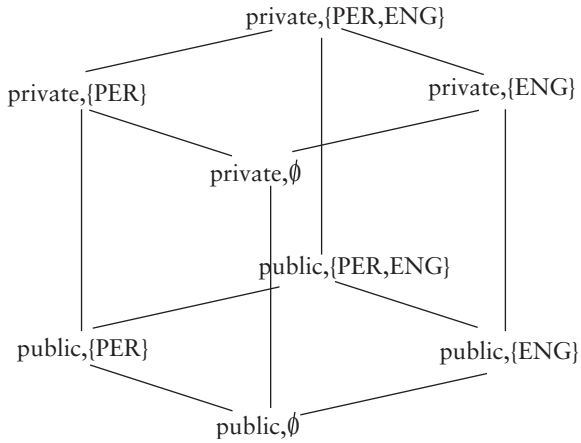


Figure 5.12: A Lattice of Security Labels

To see how this lattice of security labels can be used to implement mandatory *need-to-know* (*least privilege*) policies, look at the lattice of Figure 5.12 in the light of the simple confidentiality policy mentioned above. A subject with security label (*private*, ENG) will not be able to read any object that has the category PER in the compartment of its label. Thus, even an object labelled (*public*,{PER, ENG}) will be out of bounds.

The discussion of security lattices started with the simple hierarchical lattice of Figure 5.11, typical of military multi-level security policies. We then added compartments to express a greater variety of policies. Systems have been built to enforce such policies with very high levels of assurance. Today, we find applications that use high-assurance multi-level security systems but have no hierarchical component in their security levels. For example, a firewall could use the lattice from Figure 5.13 to achieve a strict separation between the inside and the outside of a network.

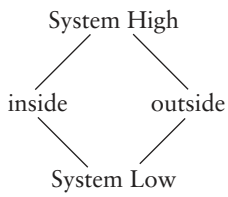


Figure 5.13: A Lattice for a Firewall

5.9 FURTHER READING

The fundamental access control structures and security lattices are covered in [8, 80, 191]. Influential early papers on access control (protection) are [144, 196]. Wilkes [234] has more to say about access control in operating systems developed in the 1960s. More examples of security policies for protection rings are given in [191, 181]. A good survey of role-based access control (RBAC) has been published in [200]. Further information about lattice-based access control models together with a description of how they are used to deal with confidentiality and integrity can be found in [199].

5.10 EXERCISES

Exercise 5.1 You are given two bits to capture access operations on a directory. How would you use the four operations available to you? How would you control the creation and deletion of files? How would you implement the concept of *hidden* files with these access operations? (Hidden files are only visible to authorized subjects.)

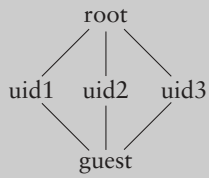
Exercise 5.2 Consider a system with the four access operations read, write, grant, and revoke. You can use grant not only to give other subjects read and write access, but also to grant them the right to grant access to the objects you own. Which data structure and algorithm would you use to implement the grant and revoke operation so that you can revoke *all* access to an object you own?

Exercise 5.3 In a social network, what access rights would you give to *friends*? What access rights would you give to the *friends* of your *friends*? How would your policy depend on the length of such a recommendation chain?

Exercise 5.4 Discuss the differences between groups and roles, if any.

Exercise 5.5 Explain why the partial ordering of abilities as defined in Section 5.8.2 does not constitute a lattice. Try to convert the partial ordering into a lattice by adding any further elements you need to the set of abilities.

Exercise 5.6 You are given a security policy stating that a subject has access to an object if and only if the security level of the subject dominates the security level of the object. What is the effect of using the lattice below with this policy?



Exercise 5.7 Let $(L, \leq)$ be a lattice of security levels where L is a finite set. Show that unique elements *System Low* and *System High* must exist in such a lattice.

Exercise 5.8 Construct the lattice of security labels for the security levels *public*, *confidential*, and *strictly confidential*, and for the categories ADMIN, LECTURERS, and STUDENTS. Which objects are visible to a subject with security label (*confidential*,{STUDENTS}) in a need-to-know policy? How many labels can be constructed from n security levels and m categories? By way of illustration, consider the values $n = 16$ and $m = 64$.

Exercise 5.9 You are given a security policy that uses the lattice of compartments as security labels. Access is granted only when the subject's label is a subset of the object's label. With the categories ADMIN, LECTURERS, and STUDENTS, which objects can be accessed by a subject with label {STUDENTS}? Why is a subject with label {ADMIN, STUDENTS} more constrained than a subject with label {STUDENTS}? Interpret the roles of the labels $\emptyset$ and {ADMIN, LECTURERS, STUDENTS} in this policy.

Exercise 5.10 You are given a set of categories. Implement a lattice-based *need-to-withhold* policy where you selectively withdraw access rights from subjects.

Chapter 6

Reference Monitors

The previous chapter introduced elementary concepts that are useful when writing access control policies. We now move on to the basics of enforcing such policies. More refined policy descriptions and more refined access control systems are the subject of later chapters. We will present the core mechanisms for protecting the integrity of the operating system and for controlling access to memory, focusing on access control at the bottom levels of a layered system architecture. Along the way, we will point out a few more general lessons for the design of secure systems.

OBJECTIVES

- **Introduce fundamental concepts of access control, such as *reference monitor* and *trusted computing base*.**
- **Discuss different options for the design of reference monitors.**
- **Introduce status and controlled invocation as two important security primitives.**
- **Understand the motivation for enforcing security at a low system layer, and get an overview of the security mechanisms available at the bottom system layers.**

6.1 INTRODUCTION

There are three fundamental concepts in computer security that are sufficiently closely related to create confusion but deserve to be kept apart. We take our definitions from the Glossary of the Orange Book [224].

> *Reference Monitor:* An access control concept that refers to an abstract machine that mediates all accesses to objects by subjects.
>
> *Security Kernel:* The hardware, firmware, and software elements of a trusted computing base that implement the reference monitor concept. It must mediate *all* accesses, be protected from modification, and be verifiable as correct.
>
> *Trusted Computing Base (TCB):* The totality of protection mechanisms within a computer system – including hardware, firmware, and software – the combination of which is responsible for enforcing a security policy. A TCB consists of one or more components that together enforce a unified security policy over a product or system. The ability of the TCB to correctly enforce a security policy depends solely on the mechanisms within the TCB and on the correct input by system administrative personnel of parameters (e.g. a user's clearance) related to the security policy.

So, the reference monitor is an abstract concept, the security kernel its implementation, and the TCB contains the security kernel among other protection mechanisms. Core requirements on the implementation of a reference monitor were formulated in the Anderson report [9]:

- The reference validation mechanism must be tamper-proof.[1]
- The reference validation mechanism must always be invoked.[2]
- The reference validation mechanism must be small enough to be subject to analysis and tests to be sure that it is correct.

The common view of reference monitors and security kernels is very much coloured by the original research agenda laid out in [9]. There, the security kernel includes the implementation of the reference validation mechanism, access control to the system itself, and components for managing the security attributes of users and programs. A strong case is made for implementing the security kernel in the bottom layers of the architecture from Figure 3.3. Conversely, the term security kernel sometimes stands generally for the security mechanisms at those bottom layers.

[1] Today, *tamper-resistant* has replaced *tamper-proof* in the security literature, so as not to create the impression that a security component is unbreakable.
[2] This requirement is known as *complete mediation*.

6.1.1 Placing the Reference Monitor

In principle, reference monitors can be placed anywhere in a layered architecture. Indeed, examples of all possible design decisions can be found.

- In hardware: access control mechanisms in microprocessors are covered in Section 6.3.
- In the operating system kernel: a *hypervisor* is a virtual machine that exactly emulates the host computer on which it is running. It can be used to separate users, or applications for that matter, by providing each with a separate virtual machine.
- In the operating system: access control in Unix and Windows is examined in Chapters 7 and 8, respectively. The reference monitor of the Multics operating system is described in Section 11.3.
- In the services layer: illustrated by access control in database management systems (Chapter 9), the Java Virtual Machine, `the.NET` Common Language Runtime (Chapter 20), the CORBA middleware architecture, or in web browsers.
- In the application: applications with specific security requirements may include security checks in the application code rather than invoking security services from a lower systems layer. In Web 2.0 applications, reference monitors can appear within web pages.

We can also ask where to place the reference monitor with respect to the program it should control. The reference monitor can be provided by a lower systems layer, as depicted in Figure 6.1a. This is the typical pattern of access control in an operating system. Application programs request access to protect resources. The reference monitor is part of the operating system kernel and mediates all access requests. Access control in CORBA follows the same pattern [39].

Alternatively, the program could be run within an interpreter. The interpreter mediates all access requests by the program. The byte code interpreter in the Java Virtual Machine exemplifies this approach. The program is placed within the reference monitor, as shown in Figure 6.1b.

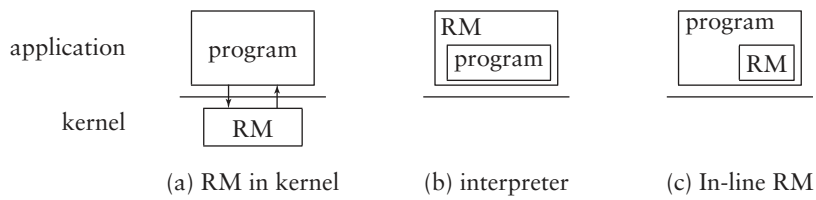


Figure 6.1: Placing the Reference Monitor (RM)

In the third approach, the program is rewritten to include the access control checks. The in-line reference monitors introduced in [90] are an example of this option. Now the reference monitor is placed within the program, as in Figure 6.1c.

6.1.2 Execution Monitors

In a running system, the reference monitor receives requests from subjects. Subjects are processes executing programs. We can differentiate between reference monitors on the basis of their ability to inspect and modify those programs [205].

- An *execution monitor* [205] only looks at the history of execution steps but does not try to predict the outcome of possible future executions. This variant is typically found in operating systems, which only keep a finite (and small) amount of information about the history of an execution, and where the reference monitor does not consider the program when making access decisions.
- The reference monitor could consider all possible future executions of the program when making its decision. *Static type checking* is an example of this approach.
- The reference monitor could rewrite the program (in-line checks into the program) to ensure that granting the access request cannot lead to violations of the security policy.

6.2 OPERATING SYSTEM INTEGRITY

Assume you have an operating system that can enforce all your access control policies. Unauthorized access to resources is impossible as long as the operating system works as intended. Attackers may then direct their attention to the operating system itself and try to disable security controls by modifying the operating system. You now definitely face an integrity problem, no matter what your original concerns were. The operating system is not only the arbiter of access requests, it is also itself an object of access control. The new security policy is:

> Users must not be able to modify the operating system.

The threat model in operating system security assumes that the attacker has access to the operating system command line, but not to the physical hardware. When securing an operating system, two competing requirements have to be addressed:

- Users should be able to use (invoke) the operating system.
- Users should not be able to misuse the operating system.

Two important concepts commonly used to achieve these goals are *modes of operation* and *controlled invocation* (also called *restricted privilege*). These concepts can be used in any layer of a computing system, be it application software, operating system, or hardware. However, remember that these mechanisms can be disabled if the attacker gets access to a lower layer.

6.2.1 Modes of Operation

The first prerequisite for an operating system to be able to protect itself from users is the ability to distinguish between computations 'on behalf of' the operating system and computations 'on behalf of' a user.

> **Mode of operation** – The mode of operation defines which actions, e.g. machine instructions, function calls, connections to network ports, may be performed on a system.

In *dual-mode operation* a system can work in

- *user* mode (a.k.a. *protected* mode), where instructions that are not critical for security may be performed, or in
- *supervisor* mode (a.k.a. *kernel, monitor, root, system* mode); *privileged instructions* are instructions that can only be executed in supervisor mode.

A *status flag* in the processor's control register can record the current mode at hardware level. For example, the Intel 80x86 processor has two status bits, thereby supporting four modes. The Unix operating system distinguishes between *supervisor* (root) and *user* mode.

Why are such modes useful? For example, the operating system could grant write access to memory locations only if the processor is in supervisor mode to stop users from writing directly to memory and corrupting the logical file structure.

6.2.2 Controlled Invocation

When a user wants to execute an operation requiring supervisor mode, e.g. a write to a memory location, the processor has to switch between modes – but how should this switch be performed? Simply changing the status bit to supervisor mode would give all privileges associated with this mode to the user without any control of what the user actually does. Therefore, it is desirable that the system only performs a predefined set of operations in supervisor mode and then returns to user mode before handing control back to the user. We refer to this process as *controlled invocation*.

> **Controlled invocation** – Invocation of a function that executes privileged instructions to provide a limited, well-defined functionality and then returns to user mode.

6.3 HARDWARE SECURITY FEATURES

Hardware is the lowest layer in an IT architecture. Hardware is also a place where computer security can link in with physical security. Security mechanisms at the hardware level are therefore a natural starting point for our investigations. This section looks at the security features of microprocessors, using the Motorola 68000 and the Intel 80x86 as historic examples.

6.3.1 Security Rationale

There are two good reasons for placing security in one of the lower system layers (Figure 6.2). A security mechanism in any given layer can be compromised if an attacker gets in at a layer below. To evaluate the security of a system, you therefore have to check that your security mechanisms cannot be bypassed. The more complex your system is, the more difficult this check becomes. At the core of your system, you can hope to find reasonably simple structures that are amenable to thorough analysis. This argument points to a first reason for placing security in the core.

It may be possible to evaluate security to a higher level of assurance.

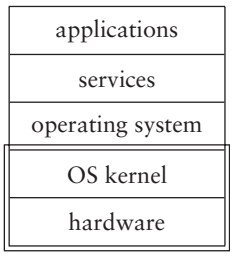


Figure 6.2: Protection in the Security Kernel

Microprocessor design is very much the science of establishing which set of operations is most useful to the majority of users. The right choice and the efficient implementation of generic operations determine overall performance. You can follow the same route when implementing security. Decide on the generic security mechanisms and put them into the core of your system. This is the second reason for placing security in the core.

Putting security mechanisms into the core of the system reduces the performance overheads caused by security checks.

All the arguments we have put forward to bolster the case for putting security mechanisms into the core of the system have pushed us to the machine end on the man–machine scale (Figure 6.3). The consequences are predictable.

Access control decisions made by reference monitors are far removed from access control decisions made by applications.

6.3.2 A Brief Overview of Computer Architecture

We assume that the reader is familiar with the fundamental concepts of computer architecture. If necessary, this background can be acquired from any good textbook on this topic, e.g. [116]. For our purposes, the simple schematic description of a computer in Figure 6.4 suffices. We have a central processing unit (CPU), memory, a bus connecting CPU and memory, and some input/output devices. In real life, all three entities can have a much more refined structure.

Figure 6.3: The Place of the Security Kernel on the Man–Machine Scale

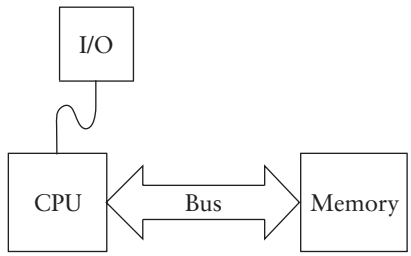


Figure 6.4: Schematic Picture of a Computer

The Central Processing Unit

The core CPU components are:

- *Registers* – which can be categorized as general purpose registers and *dedicated* registers. Important dedicated registers are
  - the *program counter*, which points to the memory location that contains the next instruction to be executed;
  - the *stack pointer*, which points to the top of the system stack;
  - the *status register*, which allows the CPU to keep essential state information.
- *Arithmetic Logic Unit (ALU)* – executes instructions given in a machine level language; executing an instruction may also set bits in the status register.

The *system stack* is a specially designated part of the memory. The stack can be accessed by *pushing* data on to its top or by *popping* data from its top. To switch between different programs, the CPU performs a *context switch* and saves the state of the current process – program counter, status register, etc. – on the stack before giving control to the new process.

Input/Output

Input devices such as keyboards and output devices such as monitors facilitate user interaction. For entering security-sensitive data such as username and password, a

trusted path from the I/O device to the TCB is required. An example is the secure attention sequence CTRL+ALT+DEL in Windows. In applications where users sign documents, we would like to be sure that the document displayed is also the document that is actually being signed.

Memory Structures

The following brief survey examines the security characteristics of different memory structures.

- RAM (random access memory): this is read/write memory; you cannot rely on such memory to guarantee integrity or confidentiality.
- ROM (read-only memory): there is a built-in integrity guarantee so you would only have to add your own confidentiality mechanisms; ROM may be a good location for storing (parts of) the operating system.
- EPROM (erasable and programmable read-only memory): may be used for storing parts of the operating system or cryptographic keys; technologically more sophisticated attacks may pose a threat to security.
- WROM (write-once memory): this memory structure comes with a mechanism for freezing the memory contents once and for all. In hardware, this can be achieved by blowing a fuse that has been placed on the write line, but you may also come across 'logical fuses'. WROM may be a good location for storing cryptographic keys; write-only disks are being used for recording audit trails.

There exists a further distinction between *volatile* memory and *non-volatile* (*permanent*) memory. Volatile memory loses its content if power is switched off. Physically, this process is neither instantaneous nor complete. If power is switched on immediately after having been switched off, the old data may still be held in memory. Even if power has been switched off for some time, it is possible that the old memory contents can be reconstructed by special electronic techniques. To delete data completely, the memory locations that held the data have to be overwritten repeatedly with suitable bit patterns that depend on the storage medium [225]. To prevent sensitive data leaving imprints at the electronic level, keep moving it round in memory.

Permanent memory keeps its content if power is switched off. If sensitive data, such as encryption keys, are stored in permanent memory and if attackers have direct access to memory bypassing the CPU, further measures such as cryptographic or physical protection have to be implemented. For example, a light sensor placed in a tamper-resistant module may detect an attempted manipulation and trigger the deletion of the data kept in the module. Physical protection is a topic in its own right that falls outside the scope of this book. We restrict our attention to the situation where users get access to memory only through the CPU and investigate how the CPU can enforce

confidentiality and integrity. For example, what can be done to prevent a computer virus from overwriting a clean version of the operating system with an infected version?

Keep in mind that 'memory' in Figure 6.4 is another abstraction. Logically, memory may consist of main memory, caches for quick access, buffers, etc. Even backup media could be included in this list. Hence, a data object may exist simultaneously in more than one location in this memory hierarchy. Besides a persistent copy in secondary memory, there will be temporary copies. Usually, the location and lifetime of these temporary copies are not under the user's control. Security controls on a data object can be bypassed if one of the temporary copies is held in an unprotected memory area.

6.3.3 Processes and Threads

A process is a program in execution. A process is an important unit of control for the operating system, and of course also for security. Roughly, a process consists of

- executable code,
- data,
- the execution context, e.g. the contents of certain relevant CPU registers.

A process works in its own address space and can communicate with other processes only through primitives provided by the operating system. This logical separation between processes is a useful basis for security. On the other hand, a context switch between processes is an expensive operation as the operating system has to save the current execution context on the stack.

Threads are strands of execution within a process. All threads share the process address space, thereby avoiding the overheads of a full context switch, but also avoiding control by a potential security mechanism.

6.3.4 Controlled Invocation – Interrupts

Processors are equipped to deal with interruptions in execution, created by errors in the program, user requests, hardware failure, etc. The mechanisms to do so are variously called *exceptions*, *interrupts*, and *traps*. The different terms may refer to different types of events but, as ever, there are competing classifications; see e.g. [116] for further reading.

We will use *trap* as the generic term and explain how traps can be used for security purposes. A trap is a special input to the CPU which includes an address, called an *interrupt vector*, in an *interrupt vector table*. The interrupt table gives the location of the program which deals with the *condition* specified in the trap. This program is called the *interrupt handler*. When a trap occurs, the system saves its current state on the stack and then executes the interrupt handler (Figure 6.5). In this way, control is taken away from the user program. The interrupt handler has to make sure that the system is restored to

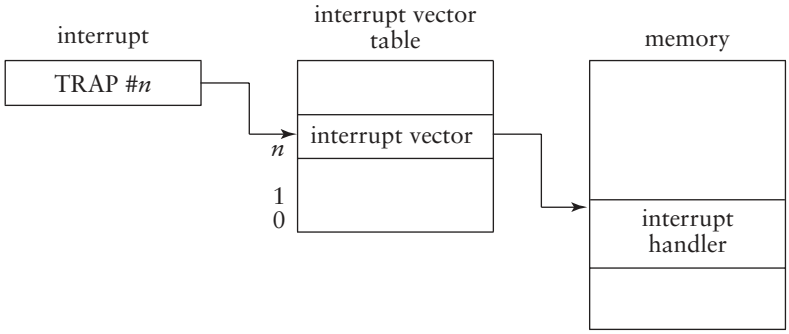


Figure 6.5: Processing of an Interrupt

a proper state, e.g. by clearing the supervisor status bit, before returning control to the user program.

It is possible for a further interrupt to arrive while the processor deals with a current interrupt. The processor may then have to interrupt the current interrupt handler. Improper handling of such a situation can lead to security flaws. On early PCs, a user could interrupt the execution of a program by pressing CTRL-C so that the processor returns to the operating system prompt with the status bit of the current process. A user could then enter supervisor mode by interrupting the execution of an operating system call. It is therefore important that before executing a program, the interrupt table is set up so that interrupts will be handled in an appropriate way.

From this discussion, it should be clear that the interrupt table is an interesting point of attack and has to be protected adequately. Changing an entry in the interrupt table so that it points to attack code, which is then executed before jumping to the proper interrupt handler, is one of the strategies used by virus writers (Figure 6.6). Using tables as a layer of indirection is a useful and frequently employed design technique. However, whenever such indirections are used, you should look out for attacks of the type just described.

Lesson
Redirecting pointers is a very efficient attack method.

6.3.5 Protection on the Intel 80386/80486

The Intel 80x86 architecture will illustrate some fundamental protection techniques at the processor level. There is a two-bit field in the status register defining four *privilege*

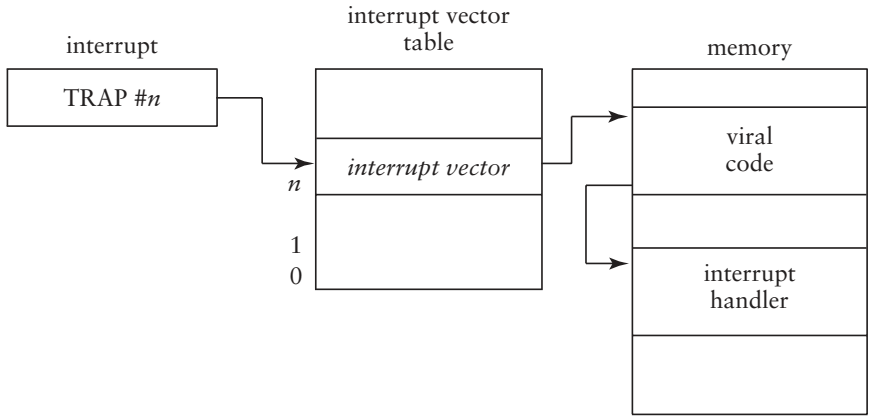


Figure 6.6: Inserting Viral Code by Redirecting an Interrupt Vector

levels (protection rings; see Section 5.6.4). The privilege level can only be changed by a single instruction (POPF), which has to be executed at level 0. Not all operating systems make use of all four levels. For example, Unix only uses levels 0 and 3. The 80x86 implements the following security policy.

> Procedures can only access objects in their own ring or in outer rings. Procedures can invoke subroutines only within their own ring.

Information about system objects such as memory segments, access control tables, and gates is stored in *descriptors*. Descriptors are stored in the *descriptor table* and accessed via *selectors*. The privilege level of an object is stored in the DPL field of its descriptor. A selector is a 16-bit field containing an index pointing to the object's entry in the descriptor table and also a *requested privilege level* (RPL) field (Figure 6.7). The use of the RPL field will be explained in a moment. Only the operating system has access to selectors.

The system objects containing information about subjects, i.e. about processes, also have descriptors and selectors. When a subject requests access to an object, the relevant selectors are loaded into dedicated segment registers. For example, the privilege level of the current process, called the *current privilege level* (CPL), is the privilege level of the selector stored in the code segment (CS) register.

Not surprisingly, we again face the problem of having to manage access to operations which require higher privileges. Assume that an application program in ring 3 needs a service from an operating system routine in ring 1. On the 80x86, this problem is solved by using *gates*. A gate is a system object that points to a procedure (in some code segment) where the gate has a privilege level different from that of the code to which it

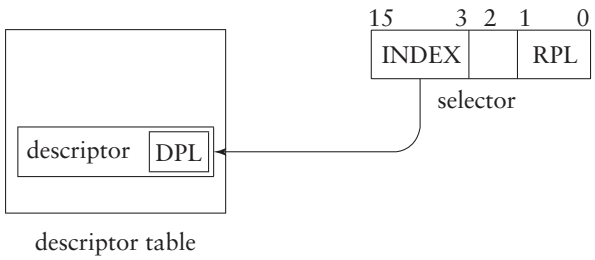


Figure 6.7: Selectors and Descriptors

points. Gates allow execute-only access to a procedure in an inner ring. Restrictions on outward calls are still enforced.

For a procedure to use a gate, the gate has to be in the same ring as the procedure. When invoking a subroutine through a gate, the current privilege level changes to the level of the code to which the gate is pointing. When returning from the subroutine, the privilege level is restored to that of the calling procedure. A subroutine call also saves the information indicating the state of the calling procedure or the return address on a stack. To determine the appropriate privilege level of the stack, remember that the calling procedure cannot write to an inner ring. However, leaving the stack in the outer ring is unsatisfactory for security reasons as it leaves the return address rather unprotected. Therefore, part of the stack (how much is described in the gate's descriptor) is copied to a more privileged stack segment.

6.3.6 The Confused Deputy Problem

Allowing outer-ring procedures to invoke inner-ring procedures creates a potential security loophole. The outer-ring procedure may ask the inner-ring procedure to copy an object residing in the inner ring to the outer ring.[3] This will not be prevented by any of the mechanisms presented so far, nor does it actually violate the security policy we have stated. We may therefore wish to extend the original security policy to be able to take into account not only the current privilege level but also the level of the calling process.

In the 80x86, such policies can be supported using the RPL field in the selector and the *adjust requested privilege level* (ARPL) instruction. The ARPL instruction changes the RPL fields of all selectors to the CPL of the calling procedure. The system can then compare the RPL (in the selector) and the DPL (in the descriptor) of an object and refuse to complete the requested operation if they differ (Figure 6.8).

[3]Today, the term 'confused deputy problem' is often used to describe situations where an unprivileged entity invokes an entity with higher privileges to perform actions that violate the security policy.

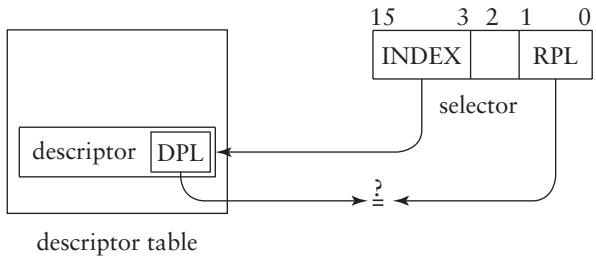


Figure 6.8: Comparing RPL and DPL

Lesson
For better precision in access control it may be beneficial to consider some aspect of the execution history when deciding on an access request.

6.4 PROTECTING MEMORY

Operating systems manage access to data and resources. They are usually not involved in the interpretation of user data. Multitasking operating systems interleave the execution of processes belonging to different users. Then, the operating system not only has to protect its own integrity but also has to prevent users from accidentally or intentionally accessing other users' data. The integrity of the operating system itself is preserved by separating user space from operating system space. Logical separation of users prevents accidental and intentional interference between users. Separation can take place at two levels:

- file management, dealing with logical memory objects;
- memory management, dealing with physical memory objects.

As far as security is concerned, this distinction is important. To see why, consider the two main ways of structuring memory, segmentation and paging. *Segmentation* divides data into logical units. Each segment has a unique name. Items within a segment are addressed by giving the segment name and the appropriate offset within the segment. The operating system maintains a table of segment names with their true addresses in memory. The Multics operating system used segmentation for logical access control.

\+ Segmentation is a division into logical units, which is a good basis for enforcing a security policy.

– Segments have variable length, which makes memory management more difficult.

Paging divides memory into pages of equal size. Addresses again consist of two parts, the page number and the offset within a page.

+ Paging allows efficient memory management. Segments in Multics are actually paged.
− Paging is not a good basis for access control as pages are not logical units. Thus, one page may contain objects requiring different protection.

Even worse, paging may open a covert channel. Logical objects can be stored across page boundaries. When such an object is accessed, the operating system will at some stage require a new page and a *page fault* will occur. If page faults can be observed, as is the case in most operating systems, then a user is provided with information in excess of the proper result of the access request.

As an example, consider a password scheme. The user enters a password which is scanned character by character and compared with a reference password stored in memory. Access is denied the moment an incorrect match is found. If a password is stored across a page boundary, then an attacker can deduce from observing a page fault that the piece of the password on the first page had been guessed correctly. If the attacker can control where the password is stored on the page, password guessing becomes easy, as sketched in Figure 6.9. In the first step, the password is placed in memory such that the first character is on a page separate from the rest of the password. The attacker now tries all values for the first character until a page fault occurs, indicating that the guess was correct. The password is then realigned in memory so that the first two password characters are on a page separate from the rest. The attacker already knows the first character and now tries all values for the second character until a page fault occurs. By continuing with this ploy, the attacker can search for each password character individually.

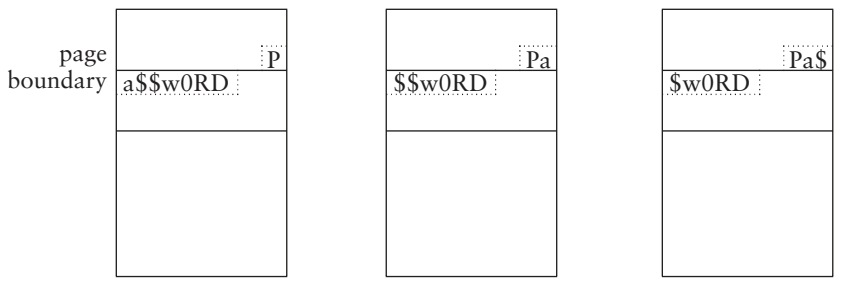


Figure 6.9: Using Page Faults as Covert Channels to Guess a Password

6.4.1 Secure Addressing

When you want the operating system to protect its own integrity and to confine each process to a separate address space, then one of your tasks is to control access to data objects in memory. Such a data object is physically represented as a collection of bits

stored in certain memory locations. Access to a logical object is ultimately translated into access operations at machine language level. At this level, you can pursue three options for controlling access to memory locations:

- the operating system modifies the addresses it receives from user processes;
- the operating system constructs the effective addresses from relative addresses it receives from user processes;
- the operating system checks whether the addresses it receives from user processes are within given bounds.

Address sandboxing is an example of the first approach. An address consists of a *segment identifier* and an *offset*. When the operating system receives an address, it sets the correct segment identifier. Figure 6.10 shows how this can be done with two register operations. First, a bitwise AND of the address with *mask_1* clears the segment identifier. Then a bitwise OR with *mask_2* sets the segment identifier to the intended value, SEG_ID.

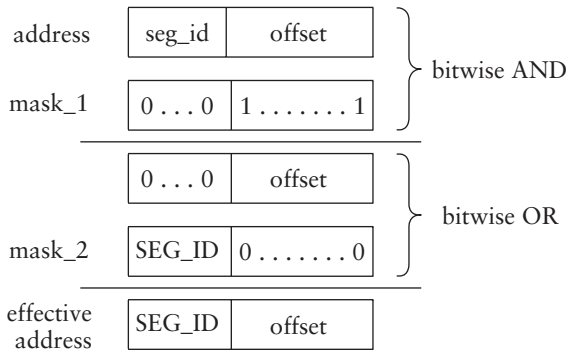


Figure 6.10: Address Sandboxing

In the second approach, clever use of addressing modes keeps processes out of forbidden memory areas. If you need more background on addressing modes, consult [116] or other books on operating systems or computer architecture. Of the various addressing modes, relative addressing is of particular interest to us.

Relative addressing: the address is specified by an offset relative to a given base address.

Relative addressing allows position-independent coding. Thus a program can be stored anywhere in memory, giving greater flexibility to the memory management utilities. It also facilitates the use of *fence registers*. The fence register contains the address of the end of the memory area allocated to the operating system. Addresses in a user program are interpreted as relative addresses (*offset, displacement*). The operating system then uses relative addressing with respect to the fence register (*base register addressing*) to calculate

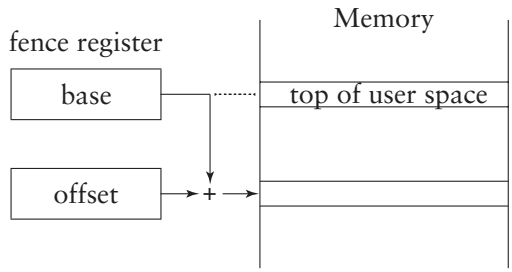


Figure 6.11: Base Register Addressing

the effective addresses (see Figure 6.11). In this way, only locations outside the operating system space can be accessed by user programs. Similar methods can be employed by the operating system to separate the memory areas allocated to different users.

This approach can be refined by defining the memory space allocated to a process through *base registers* and *bounds registers*. One can even go a step further and introduce base and bounds registers for a user's program space and data space, respectively. To make proper use of such a facility, the processor must be able to detect whether a given memory location contains data or program code. The Motorola 68000 processor did support such a separation through *function codes* (Figure 6.12). Function codes signal the processor status to the address decoder, which may use this information to select between user memory and supervisor memory or between data and programs.

Lesson
The ability to distinguish between data and programs is a very useful security feature. It provides a basis for protecting programs from modification.

FC2	FC1	FC0	
0	0	0	(undefined, reserved)
0	0	1	user data
0	1	0	user program
0	1	1	(undefined, reserved)
1	0	0	(undefined, reserved)
1	0	1	supervisor data
1	1	0	supervisor program
1	1	1	interrupt acknowledge

Figure 6.12: Motorola 68000 Function Codes

From a more abstract point of view, memory has been divided into different regions. Access control can then refer to the *location* a data object or a program comes from.

Lesson
You now have an example of location-based access control in microcosm. In distributed systems or computer networks, you often require location-based access control in macrocosm.

Most instruction sets have no means of checking the type of operands. In such a situation, type information can be supplied in the program by specifying the address registers to be used when loading data of a given type. This solution requires proper programming discipline.

In contrast, in a *tagged architecture*, each data item has a tag specifying its type. Before execution, the CPU can detect type violations directly from the value stored in memory. These tags could also be used to enforce security policies. Historically, tagged architectures have been popular in theoretical considerations rather than in actual implementations. The few examples of tagged architectures include the Burroughs B6500-7500 system and the IBM System/38 [32]. (For the reader with an interest in the history of computing, von Neumann in his *First Draft of a Report on the EDVAC* in 1945 describes a tagged architecture [228].) Figure 6.13 shows a tagged architecture which indicates the type of memory objects, e.g. integer (INT), bit string (STR), or operand (OP). Tags can also be used to indicate which access operation may be performed on a memory location, e.g. read, write, or execute.

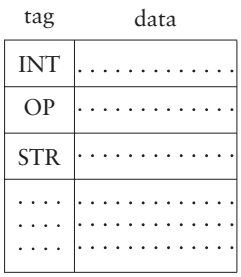


Figure 6.13: A Tagged Architecture

6.5 FURTHER READING

The fundamental access control paradigms are due to Lampson [144]. The original case for reference monitors is made in the Anderson report [9]. The outcome

of computer security research aimed at the first multi-user operating systems is treated comprehensively in [80]. A further survey of protection techniques is compiled in [146]. An excellent account of the techniques used in the design of secure multi-user operating systems is [97], which is out of print but available on the web. This book contains many useful pointers to technical reports in this area. Address sandboxing and related techniques are described in [229].

The early history of the development of secure computer systems has been collated in [158]. Opposing standpoints in the discussion of whether TCBs (security kernels) are still an appropriate paradigm in the construction of secure systems are taken in [38] (against) and [19] (in favour). Theoretical foundations for the design of reference monitors are explored in [205].

6.6 EXERCISES

Exercise 6.1 Microprocessors on smart cards used to have their entire card operating system in ROM, but the field has moved on towards microprocessors where part of the operating system can be downloaded into EEPROM. What are the advantages and disadvantages of keeping the operating system in ROM? What are the security implications of moving parts of the operating system into EEPROM?

Exercise 6.2 Can you have security without security kernels? Discuss the advantages and disadvantages of having a security kernel as the TCB.

Exercise 6.3 Look for examples that show how the following three principles are applied in building secure systems: separation of duties, abstract datatypes, and atomic operations. (An atomic operation has to be executed in its entirety to preserve security. If it is interrupted, the system may end up in an insecure state.)

Exercise 6.4 So-called parasitic viruses infect executable programs. How can the ability to distinguish between programs and data help to construct a defence against such viruses?

Exercise 6.5 Some buffer overrun attacks put the code they want to be executed on the call stack. How can the ability to distinguish between programs and data help to construct a defence against this particular type of buffer overrun attack?

Exercise 6.6 Anti-virus software scans files for attack signatures. How could a virus intercept the read requests to memory and hide its existence?

Exercise 6.7 Consider a system that writes event numbers to its audit log and uses a table to translate these numbers into messages. What is the potential advantage of using this level of indirection in log file entries? What are the potential dangers?

Exercise 6.8 As a case study, examine how type enforcement is implemented in SELinux.

Chapter 7

Unix Security

So far, we have looked at individual security mechanisms in isolation. In an actual implementation, they rely on each other. For example, access control and authentication have to work together – each would not be effective without the other. Hence, we now turn to the security mechanisms provided by an operating system. Unix is a first example that gives us the chance to inspect the mechanics of security at a fair level of detail. Security mechanisms in Linux are sufficiently similar to their counterparts in Unix for this chapter to serve also as an introduction to Linux security.

OBJECTIVES

- Understand the security features provided by a typical operating system.
- Introduce the basics of Unix security.
- See how general security principles are implemented in an actual system.
- Appreciate the task of managing security in a continuously changing environment.

7.1 INTRODUCTION

Operating systems combine building blocks such as identification and authentication, access control, and auditing to provide a coherent set of security controls. Once support for flexible and 'feature-rich' security policies is desired, security mechanisms become increasingly complex. In these circumstances, the TCB will be too large to fit into a small security kernel. In our layered model, we now look at the security controls provided at the operating system level. The following questions may serve as a guide when assessing the security of an operating system.

- Which security features have been implemented?
- How can these security features be managed?
- What assurances are there that the security features will be effective?

There is a general pattern to the way security controls are organized in most operating systems. Information about users (principals) is stored in *user accounts*. *Privileges* granted to a user can be stored in this account. *Identification* and *authentication* verify a user's identity, allowing the system to associate the user's privileges with any process (subject) started by the user. *Permissions* on resources (objects) can be set by the system manager or the owner of a resource. When deciding whether to grant or deny an access request, the operating system may refer to the user's identity, the user's privileges, and the permissions set for the object.

Security not only deals with the *prevention* of unauthorized actions but also with their *detection*. We have to face up to the fact that attackers may find their way round protection mechanisms. Provisions have to be made to keep track of the actions users have performed to be able to investigate security breaches or to trace attempted attacks. Therefore, operating systems keep (and protect) an *audit log* (*audit trail*) of security-relevant events.

Finally, the best security features of an operating system are worthless if they are not used properly. A system has to be started in a secure state, thus *installation* and *configuration* of the operating system are important issues. Inadequate default settings can be a major security weakness. Operating systems are highly complex and continually evolving software systems. Hence, there is always the chance that vulnerabilities are detected and removed or accidentally introduced in a new release. Alert systems managers have to stay in touch with current developments.

We have outlined a framework that structures operating system security along the following lines:

- principals, subjects, and objects;

- access control;
- audit, configuration and management.

This chapter examines the security features of the Unix operating system. Because of its design history, Unix did not have a great reputation for reliability or security – see [169] – but it does offer a set of security features that can be effective if used properly, and attitudes towards Unix security have changed considerably. Different releases of Unix and Linux may differ in some technicalities and in the way some security controls are enforced. Commands and filenames used in this chapter are thus indicative of typical use but may be different on your particular system. The POSIX 1003 series of standards defines common interfaces in an attempt to standardize Unix. POSIX 1003.6 deals with security mechanisms.

This chapter is by no means intended as a complete introduction to Unix security or an instruction manual on how to set up your Unix system securely. Rather, we will limit ourselves to presenting the basics of Unix security and to highlighting security features that illustrate points of general interest.

7.1.1 Unix Security Architecture

While most secure operating systems have a *security architecture* explaining how security is enforced and where security-relevant data are kept, Unix has a history of diverging and converging versions. This is a fair reflection of the fact that security features were added into Unix whenever the necessity arose, rather than being an original design objective.

Unix was originally designed for small multi-user computers in a networked environment and later scaled up to commercial servers, and scaled down to PCs. Like the Internet, Unix was developed for friendly environments such as research laboratories or universities, and security mechanisms were weak. As Unix developed, new security controls were added to the system and existing controls were strengthened. When deciding on how to implement a new feature, designers were guided very much by the desire to interfere as little as possible with the existing structures of Unix. The Unix design philosophy assumes that security is managed by a skilled administrator, not by the average computer user. Hence, support for security management often comes in the form of scripts and command line tools.

7.2 PRINCIPALS

The principals are so-called *user identities* (UIDs) and *group identities* (GIDs). UIDs and GIDs were originally 16-bit numbers. Modern systems support 32-bit identifiers. Some

−2	nobody
0	root
1	daemon
2	uucp
3	bin
4	games
9	audit
567	tuomaura

Table 7.1: Examples of User IDs

UID values have special meanings, which may differ between systems, but the superuser (root) UID is always 0. UIDs were conceived as principals for local policies. There is no distinction between UIDs defined on different systems. Moving principals and rules between policy domains is not possible. Table 7.1 gives examples of UIDs.

7.2.1 User Accounts

Information about principals is stored in *user accounts* and *home directories*. User accounts are stored in the `/etc/passwd` file. Entries in this file have the format

username:password:UID:GID:ID string:home directory:login shell

The *username* is a string up to eight characters long. It identifies the user when logging in but is not used for access control. Unix does not distinguish between users having the same UID. The *password* is stored encrypted (Section 7.3.1). The field *ID string* contains the user's full name. The last two fields specify the user's *home directory* and the Unix *shell* available to the user after successful login. Further user-specific settings are defined in the `.profile` file in the user's home directory. The actions taken by the system when a user logs in are specified in the file `/etc/profile`. Displaying the password file with `cat /etc/passwd` or `less /etc/passwd` will produce entries like

```
dieter:RT.QsZEEsxT92:10026:53:Dieter Gollmann:/home/staff/
dieter:/usr/local/bin/bash
```

7.2.2 Superuser (Root)

In every Unix system there is a user with special privileges. This *superuser* has UID 0 and usually the username *root*. The root account is used by the operating system for essential tasks like login, recording the audit log, or access to I/O devices.

All security checks are turned off for the superuser, who can do almost everything. For example, the superuser can become any other user. The superuser can change the system clock. The superuser can find a way round some of the few restrictions imposed on

him. For example, a superuser cannot write to a filesystem mounted as read-only but can dismount the filesystem and remount it as writable. The superuser cannot decrypt passwords because `crypt` is a one-way function.

7.2.3 Groups

Users belong to one or more *groups*. Collecting users in groups is a convenient basis for access control decisions. For example, one could put all users allowed to access email in a group called `mail`, or all operators in a group `operator`. Every user belongs to a *primary group*. The GID of the primary group is stored in `/etc/passwd`. The file `/etc/group` contains a list of all groups. Entries in this file have the format

group name:group password:GID:list of users

For example, the entry

```
infosecwww:*:209:chez,af
```

tells us that group `infosecwww` has the password disabled, has GID 209, and two members, `chez` and `af`. Table 7.2 lists GIDs with special meanings.

0	system/wheel
1	daemon
2	uucp
3	mem
4	bin
7	terminal

Table 7.2: Special Group IDs

In older versions of System V Unix, a user could only be in one group at a time. Modern Unix and Linux versions follow Berkeley Unix by letting users reside in more than one group. In addition to the *primary group*, several *supplementary groups* can be specified in the user account.

7.3 SUBJECTS

The subjects are processes. Each process has a *process ID* (PID). New processes are created using `exec` or `fork`. Each process is associated with a *real* UID/GID and an *effective* UID/GID. The real UID is inherited from the parent process. Typically it is the UID of the user who is logged in. The effective UID is inherited from the parent process or from the file being executed (Section 7.5.1). POSIX compliant versions also keep a *saved* UID/GID. The following example illustrates the use of real and effective UID/GID and the Unix logon process.

Process	UID real	UID effective	GID real	GID effective
/bin/login	root	root	system	system

User diego logs on; the logon process verifies username and password and changes UID and GID:

/bin/login	diego	diego	staff	staff

The logon process executes the user's login shell:

/bin/bash	diego	diego	staff	staff

From the shell the user executes the command `ls`:

/bin/ls	diego	diego	staff	staff

The user executes command `su` to start a new shell as root:

/bin/bash	diego	root	staff	system

7.3.1 Login and Passwords

Users are identified by *usernames* and authenticated by *passwords*. When the system is booted, the login process is started, running as root. When a user logs in, this process verifies username and password. If the verification succeeds, UID/GID are changed to that of the user, and the user's login shell is executed. Root login can be restricted to terminals nominated in `/etc/ttys`. The last time a user has logged in is recorded in `/usr/adm/lastlog` and can be displayed, e.g. with the `finger` command.

On many Unix systems, passwords are limited to eight characters. There exist tools that support good practice in choosing passwords by preventing the use of weak passwords. Passwords are enciphered (hashed, to be precise) with the `crypt(3)` algorithm, which repeats a slightly modified DES algorithm 25 times, using the all-zero block as start value and the password as key. The encrypted passwords are stored in the `/etc/passwd` file.

When the password field for a user is empty, the user does not have to provide a password on login. When the password field starts with an asterisk, the user cannot log in because such values can never be the result of applying the hash function to a cleartext password. This is a common method of disabling a user's account.

Passwords are changed using the `passwd(1)` command. You are asked to supply your old password first to guard against someone else changing your password when you are logged in but leave your machine unattended (regarded as bad practice anyway). Since characters are never displayed on the screen when a password is entered, you are asked to enter a new password twice, to ensure that you really typed what you thought you typed. After a change, you can confirm the effect of the change with a new login or with the `su(1)` (set user) command.

7.3.2 Shadow Password File

Security conscious versions of Unix offer further provisions for password security. The file `/etc/passwd` is world-readable as it contains data from user accounts that are needed by many programs. Thus, an attacker can copy the password file and then search for passwords in an off-line dictionary attack. To remove this vulnerability, passwords are stored in a *shadow password file*, e.g. `/.secure/etc/passwd`, that can only be accessed by root. This file can also be used for password ageing and automatic account locking. File entries have nine fields:

- username
- password
- days since password was last changed
- days left before user may change password
- days left before user is forced to change password
- days to issue password expiry warning
- days left after password expiry until account is disabled
- days the account has been disabled
- reserved.

Password salting is another method for slowing down dictionary attacks. The *salt* is a random 12-bit value that is added to the password proper, and stored in the clear.

7.4 OBJECTS

The objects of access control include files, directories, memory devices, and I/O devices. For the purpose of access control, all are treated uniformly as *resources*. Resources are organized in a tree-structured filesystem.

7.4.1 The Inode

Each file entry in a directory is a pointer to a data structure called an *inode*. Table 7.3 gives fields in the inode that are relevant for access control. Each directory contains a pointer to itself, the file '.', and a pointer to its parent directory, the file '..'. Every file has an owner, usually the user who has created the file. Every file belongs to a group. Depending on the version of Unix, a newly created file belongs either to its creator's group or to its directory's group.

Inspecting a directory with the command `ls -l` produces listings like

```
-rw-r--r-- 1 diego staff 1617 Oct 28 11:01 adcryp.tex
drwx------ 2 diego staff  512 Oct 25 17:44 ads/
```

which contain the following information:

- The first character gives the type of the file: a '-' indicates a file, '`d`' a directory, '`b`' a block device file, and '`c`' a character device file.

mode	type of file and access rights
uid	user who owns the file
gid	group which owns the file
atime	access time
mtime	modification time
itime	inode alteration time
block count	size of file
	physical location

Table 7.3: Fields in the Inode Relevant for Access Control

- The next nine characters give the *file permissions*, to be discussed below.
- The following numerical field is the *link counter*, counting the number of links (pointers) to the file.
- The next two fields are the *name* of the owner and the *group* of the file.
- Then follows the size of the file in bytes.
- The time and date is *mtime*, the time of the last modification. `ls -lu` displays *atime*, the time of last access. `ls -lc` displays *itime*, the time of last modification of the inode.
- The last entry is the name of the file. The '/' after `ads` indicates a directory. The filename is stored in the directory, not in the inode.

The *file permissions (permission bits)* are grouped in three triples that define read, write, and execute access for *owner*, *group*, and *other* (also called *world*) respectively. A '-' indicates that a right is not granted. Thus `rw-r--r--` gives read and write access to the owner and read access to group and other, `rwx------` gives read, write, and execute access to the owner and no rights to group and other.

File permissions in Unix are also specified as octal numbers by splitting the nine permissions into three groups of three. Each access right is represented by a bit, which, if set, grants access. These numbers are shown in Table 7.4. A combination of rights is the sum of the corresponding numbers. For example, the permission `rw-r--r--` is equivalent to 644. The permission 777 gives all access rights to owner, group, and others.

7.4.2 Default Permissions

Unix utilities, such as editors or compilers, typically use default permissions 666 when creating a new file and permissions 777 when creating a new program. These permissions can be further adjusted by the `umask`. The `umask` is a three-digit octal number specifying the rights that should be withheld. Thus, `umask` 777 denies all access, while `umask` 000 adds no further restrictions. Sensible default settings are:

022 all permissions for the owner, read and execute permission for group and world;

400	read by owner
200	write by owner
100	execute by owner
040	read by group
020	write by group
010	execute by group
004	read by world
002	write by world
001	execute by world

Table 7.4: Octal Representation of Access Permissions

037 all permissions for the owner, read permission for group, no permissions for world;

077 all permissions for the owner, no permissions for group and world.

The actual default permission is then derived by *masking* the default permissions of a Unix utility with the `umask`. A logical AND of the bits in the default permission and of the inverse of the bits in the `umask` is computed. For example, given default permission 666 and `umask` 077 we compute 666 AND NOT(077) which results in 600, giving the owner of the file read and write access while all other access is denied. The `umask` can be changed by the command

```
umask [−S] [mask]
```

where the flag `-S` indicates symbolic mode. When no mask is specified, the current `umask` is displayed.

The `umask` in `/etc/profile` defines a system wide default setting. These default settings can be overruled for individual users by putting `umask` into the user's home directory in files such as `/etc/profile`, `.profile`, `.login`, or `.cshrc`, depending on the way a particular Unix installation has been set up. It is not possible to define individual default permissions for directories and let files inherit their permissions from their directory.

When a new file is created using the copy command `cp`, the permissions of the file are derived from the `umask`. When a new file is created by renaming an existing file using the command `mv`, the existing permissions are retained.

7.4.3 Permissions for Directories

Directories are created with the `mkdir` command. To put files and subdirectories into a directory, a user needs the correct file permissions for the parent directory.

- Read permission allows a user to find which files are in the directory, e.g. by executing `ls` or similar commands.

- Write permission allows a user to add files to and remove files from the directory.
- Execute permission is required for making the directory the current directory and for opening files inside the directory. You can open a file in the directory if you know that it exists but you cannot use `ls` to see what is in the directory.

Thus, to get access to your own files, you need execute permission in the directory. To prevent other users from reading your files, you could either set the access permission accordingly or you could prevent access to the directory. To delete a file, you need write and execute access to the directory. You do not need any permission on the file itself. It can even belong to another user. To quote a systems manager on this feature: 'A real pain if you try and install a permanent file in someone's directory.'

A remnant from earlier versions of Unix is the sticky bit. Its original purpose was to keep the text segment of a program in virtual memory after its first use. The system thus avoided transferring the program code of frequently accessed programs into the paging area. Today, the sticky bit is used to restrict the right to delete a file. For example, job queues are often world-writable so that anyone can add a file. However, in this case everyone would be able to delete files as well. When a directory has the sticky bit set, an entry can only be removed or renamed by a user if the user is the owner of the file, the owner of the directory, and has write permission for the directory, or by the superuser.

When `ls -l` displays a directory with the sticky bit set, `t` appears instead of `x` as the execute permission for world.

7.5 ACCESS CONTROL

Access control is based on attributes of subjects (processes) and of objects (resources). Standard Unix systems associate three sets of access rights with each resource, corresponding to *owner*, *group*, and *other*. *Superusers* are not subject to this kind of access control. Unix treats all resources in a uniform manner by making no distinction between files and devices. The permission bits are checked in the following order.

- If your `uid` indicates that you are the owner of the file, the permission bits for *owner* decide whether you can get access.
- If you are not the owner of the file but your `gid` indicates that your group owns the file, the permission bits for *group* decide whether you can get access.
- If you are neither the owner of the file nor a member of the group that owns the file, the permission bits for *other* decide whether you can get access.

It is therefore possible to set permission bits so that the owner of a file has less access than other users. This may come as a surprise but is also a valuable general lesson. For any access control mechanism you have to know precisely the order in which different access criteria are checked.

7.5.1 Set UserID and Set GroupID

We return to controlled invocation. Unix requires superuser privilege to execute certain operating system functions, for example only root can listen at the *trusted ports* 0–1023, but users should not be given superuser status. A way has to be found to meet both demands. The solutions adopted in Unix are *set userID* (SUID) and *set groupID* (SGID) programs. Such programs run with the effective user ID or group ID of their owner or group, giving temporary or restricted access to files not normally accessible to other users. When `ls -l` displays an SUID program, then the execute permission of the owner is given as `s` instead of `x`:

```
-rws--x--x 3 root bin 16384 Nov 16 1996 passwd*
```

When `ls -l` displays an SGID program, the execute permission of the group is given as `s` instead of `x`. In the octal representation of permissions a fourth octet placed in front of the permissions for owner, group, and others is used to indicate SUID and SGID programs, and directories with the sticky bit set (Table 7.5).

4000	set user ID on execution
2000	set group ID on execution
1000	set sticky bit

Table 7.5: Octal Representation of SUID and SGID Programs

If, as is often the case, root is the owner of an SUID program, a user who is executing this program will get superuser status *during execution*. Important SUID programs are:

`/bin/passwd`	change password
`/bin/login`	login program
`/bin/at`	batch job submission
`/bin/su`	change UID program

We have to add a customary warning. As the user has the program owner's privileges during execution of an SUID program, this program should only do what the owner intended. This is particularly true for SUID programs owned by root. An attacker who can change the behaviour of an SUID program, e.g. by interrupting its execution, may not only embark on actions requiring superuser status during the attack but also be able to change the system so that superuser status can be obtained on further occasions. In this respect, danger comes from SUID programs with user interaction. All user input, including command line arguments and environment variables, must be processed with extreme care. A particular pitfall are *shell escapes* which give a user access to shell commands while running as superuser. Programs should have SUID status only if it is really necessary. The systems manager should monitor the integrity of SUID programs with particular care.

7.5.2 Changing Permissions

The permission bits of a file are changed with the `chmod` command which can be run only by the owner of the file or by the superuser. This command has the following formats:

`chmod [-fR] absolute_mode file`	specifies the value for all permission bits;
`chmod [-fR] [who]+permission file`	adds permissions;
`chmod [-fR] [who]-permission file`	removes permissions;
`chmod [-fR] [who]=permission file`	resets permissions as specified.

In *absolute mode*, the file permissions are specified directly by an octal number. In *symbolic mode*, the current file permissions are modified. The *who* parameter can take the following values:

`u` ... changes the owner permissions;
`g` ... changes the group permissions;
`o` ... changes the other permissions;
`a` ... changes all permissions;

The *permission* parameter can take the values

`r` read permission;
`w` write permission;
`x` execute permission for files, search permission for directories;
`X` execute permission only if file is a directory or at least one execute bit is set;
`s` set-user-ID or set-group-ID permission;
`t` save text permission (set the sticky bit).

The option `-f` suppresses error messages, the option `-R` applies the specified change recursively to all subdirectories of the current directory.

The SUID permission of a program can be set as follows:

`chmod 4555 file`	set SUID flag;
`chmod u+s file`	set SUID flag;
`chmod 555 file`	clear SUID flag;
`chmod u-s file`	clear SUID flag.

The GUID permission is set using `g` instead of the `u` option.

The command `chown` changes the owner of a file, `chgrp` changes the group of a file. The `chown` command could be a potential source of unwelcome SUID programs. A user

could create an SUID program and then change the owner to root. To prevent such an attack, some versions of Unix only allow the superuser to run `chown`. Other versions allow users to apply `chown` to their own files and have `chown` turn off the SUID and SGID bit. Similar considerations apply to `chgrp`.

7.5.3 Limitations of Unix Access Control

Files have only one owner and one group. Permissions only control read, write, and execute access. All other access rights, e.g. the right to shut down the system or the right to create a new user, have to be mapped to the basic file access permissions. Operations other than read, write, execute have to be left to the applications. In general, it is often impractical to implement more complex security policies with the Unix access control mechanisms. In this respect, Unix security lies more towards the machine end of the man–machine scale (Figure 7.1).

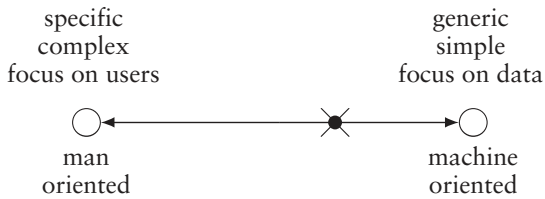


Figure 7.1: Unix Security on the Man–Machine Scale

7.6 INSTANCES OF GENERAL SECURITY PRINCIPLES

In this section, we will demonstrate how some of our general security principles find their expression in the context of Unix.

7.6.1 Applying Controlled Invocation

A sensitive but freely accessible resource such as a web server can be protected by controlled invocation patterns that combine the concepts of ownership, permission bits, and SUID programs:

- Create a new UID 'webserver' that owns the resource and all the programs that need access to the resource.
- Give access permission to the resource only to its owner.
- Define all the programs that access the resource as SUID to *webserver* programs.

Lesson
Beware of overprotection. If you deny users direct access to a file they need to perform their job, you have to provide indirect access through SUID programs. A flawed SUID program may give users more opportunities for access than wisely chosen permission bits. This is particularly true if the owner of the resource and the SUID program is a privileged user like root.

In this case we see an example of a technique that is often used in the design of security mechanisms. An abstract attribute is represented by a data structure in the system. This data structure is then reused by another security mechanism for a different purpose. The UID was introduced as the representation of real users in the system. Now, the UID is used for a new kind of access control where it no longer corresponds to real users.

7.6.2 Deleting Files

Objects exist in logical and in physical memory. What happens if we remove (delete) a file from the filesystem? Does it still exist in some form?

Unix has two ways of copying files. The command `cp` creates an identical but independent copy owned by the user running `cp`. The commands `link` and `ln` create a new filename with a pointer to the original file and increase the *link counter* of the original file. The new file shares its contents with the original. If the original file is deleted with `rm` or `rmdir`, it disappears from its parent directory but the contents of the file as well as its copy still exist. Hence, users may think that they have deleted a file whereas it still exists in another directory, and they still own it. To make sure of getting rid of a file, the superuser has to run `ncheck` to list all the links to that file and then delete those links. Also if a process has opened a file which is then deleted by its owner, the file will remain in existence until that process closes the file.

Once a file has been deleted the memory space allocated to this file becomes available again. However, until these memory locations have actually been used again they still contain the file's contents. To avoid such *memory residues*, you should *wipe* the file by overwriting its contents with all-zeros or another pattern appropriate to the storage medium before deleting it. Even then your file may not have been deleted completely as advanced filesystems, e.g. a defragmenter, may move files around, leaving more copies of the file on disk.

7.6.3 Protection of Devices

The next issue still relates to the distinction between logical and physical memory structures. Unix treats devices like files. Thus, access to memory or access to a printer can be controlled like access to a file through setting permission bits. Devices are created

using the `mknod` command which should only be executable by root. A small sample of the devices commonly found in the directory `/dev` is:

`/dev/console`	console terminal
`/dev/mem`	main memory map device (image of the physical memory)
`/dev/kmem`	kernel memory map device (image of the virtual memory)
`/dev/tty`	terminal

Attackers can bypass the controls set on files and directories if they can get access to the memory devices holding these files. If the read or write permission bits for world are set on a memory device, an attacker can browse through memory or modify data in memory without being affected by the permissions defined for the files stored in this memory. Almost all devices should therefore be world-unreadable and world-unwritable.

Commands like the *process status* command `ps` display information about memory usage and therefore require access permissions for the memory devices. Defining `ps` as an SUID to root program allows `ps` to acquire the necessary permissions but a compromise of the `ps` command would leave an attacker with root privileges. A more elegant solution has `ps` as an SGID program and lets the group `mem` own the memory devices.

The `tty` terminal devices are another interesting example. When a user logs in, a terminal file is allocated to the user who becomes owner of the file for the session. (When a terminal file is not used, it is owned by root.) It is convenient to make this file world-readable and writable so that the user can receive messages from other parties. However, this also introduces vulnerabilities. The other parties are now able to monitor the entire traffic to and from the terminal, potentially including the user's password. They can send commands to the user's terminal, e.g. reprogramming a function key, and have these commands executed by the unwitting user. In some systems, intelligent terminals can automatically execute commands. This gives an attacker the opportunity to submit commands using the privileges of another user.

7.6.4 Changing the Root of the Filesystem

Access control can be implemented by constraining suspect processes in a *sandbox*. Access to objects outside the sandbox is prevented. In Unix, the *change root* command `chroot` restricts the part of the filesystem available to an unauthorized user. This command can only be executed by root. The command

`chroot` <*directory*> <*command*>

changes the root directory from / to *directory* when *command* executes. Only files below the new root are accessible thereafter. When employing this strategy, you have to make sure that user programs find all the system files they need. These files are 'expected' to be in directories such as `/bin`, `/dev`, `/etc`, `/tmp`, or `/usr`. New directories of the same

names have to be created under the new root and populated with the files the user will need by copying or linking to the respective files in the original directories.

7.6.5 Mounting Filesystems

When you have different security domains and introduce objects from another domain into your system, you may have to redefine the access control attributes of these objects.

The Unix filesystem is built by linking together filesystems held on different physical devices under a single root, denoted by '/'. This is done with the `mount` command. In a networked environment, remote filesystems (NFS) can be mounted from other network nodes. Similarly, users could be allowed to mount a filesystem from their own storage medium (`automount`).

If you are a security expert, warning bells should start to ring. The mounted filesystems could contain all sorts of unwelcome files, e.g. SUID to root programs sitting in an attacker's directory. Once the filesystem has been mounted, the attacker could obtain superuser status by running such a program. Danger also comes from device files which allow direct access to memory, where the permissions have been set so that an attacker has access to these files. Therefore, the command

`mount` [`−r`] [`−o` *options*] *device directory*

comes with a `-r` flag specifying read-only mount and options such as:

`nosuid`	turns off the SUID and SGID bits on the mounted filesystem;
`noexec`	no binaries can be executed from the mounted filesystem;
`nodev`	no block or character special devices can be accessed from the filesystem.

Again, different versions of Unix implement different options for `mount`.

Lesson
UIDs and GIDs are *local* identifiers that need not be interpreted the same way on different Unix systems (from different vendors). When mounting remote filesystems, clients may misinterpret these identifiers. Hence, *globally unique* identifiers ought to be used across networks.

7.6.6 Environment Variables

Environment variables are kept by the shell and are normally used to configure the behaviour of utility programs. Table 7.6 lists some environment variables for the *bash* shell. A process inherits the environment variables by default from its parent process and

PATH	searchpath for shell commands
TERM	terminal type
DISPLAY	name of display
LD_LIBRARY_PATH	path to search for object and shared libraries
HOSTNAME	name of Unix host
PRINTER	default printer
HOME	path to home directory
PS1	default prompt
IFS	characters separating command line arguments

Table 7.6: Environment Variables for the *Bash* Shell

a program executing another program can set the environment variables for the program called to arbitrary values.

This is a problem as the invoker of SUID/SGID programs is in control of the environment variables these programs are given. An attacker could try to take control of execution by setting the environment variables to dangerous values. Furthermore, many libraries and programs are controlled by environment variables in obscure or undocumented ways. For example, an attacker may set IFS to unusual values to circumvent protection mechanisms that filter out dangerous inputs to SUID/SGID programs (more in Chapter 10). As a countermeasure, an SUID/SGID program could erase the entire environment and then reset a small set of necessary environment variables to safe values.

Lesson
Inheriting things you don't want or don't know about can become a security problem.

7.6.7 Searchpath

Our final favourite is the execution of programs taken from a 'wrong' location. Unix users interact with the operating system through a *shell* (a command line interpreter). As a matter of convenience, a user can run a program simply by typing its name without specifying the full *pathname* that gives the location of the program within the filesystem. The shell will then look for the program following a *searchpath* specified by the `PATH` environment variable given in the `.profile` file in the user's home directory. (Use `ls -a` to see all files in your home directory and `more .profile` to see your profile.) When a directory is found which contains a program with the name specified, the search stops and that program will be executed. A typical searchpath looks like this:

```
PATH = .:$HOME/bin:/usr/ucb:/bin:/usr/bin:/usr/local:/usr/new:
       /usr/hosts
```

In this example, directories in the searchpath are separated by ':'; the first entry '.' is the current directory. It is now possible to insert a Trojan by giving it the same name as an existing program and putting it in a directory which is searched earlier than the directory containing the original program.

To defend against such attacks, call programs by giving their full pathname, e.g. `/bin/su` instead of `su`. Also, make sure that the current directory is not in the searchpath of programs executed by root.

7.6.8 Wrappers

The access control and audit mechanisms presented so far are not very sophisticated. They adhere to the traditions of operating system security and concentrate on controlling access to resources. It is possible to implement controls at 'intermediate levels' by judicious use of the basic access control mechanisms. Alternatively, we can modify Unix itself to achieve this goal. The challenge here is to find a component of Unix that can be changed in a way so that useful security controls are added while the rest of the operating system remains unaffected.

TCP wrappers very elegantly demonstrate this design approach. Unix network services such as telnet or ftp are built upon the following principle. The `inetd` daemon listens to incoming network connections. When a connection is made, `inetd` starts the appropriate server program, and then returns to listening for further connections. The `inetd` daemon has a configuration file that maps services (port numbers) to programs. Entries in this configuration file have the format

service type protocol waitflag userid executable command-line

For example, the entry for telnet could be

```
telnet stream tcp nowait root /usr/bin/in.telnetd in.telnet
```

When `inetd` receives a request for a service it handles, it consults the configuration file and creates a new process that runs the *executable* specified. The name of this new process is changed to the name given in the *command-line* field.

Usually, the name of the executable and the name given in *command-line* are the same. This redundancy opens the door for a nice trick. Point the `inetd` daemon to a *wrapper program* instead of the original executable and use the name of the process to remember the name of the original executable, which you want to run after the wrapper has performed its security controls. In our example, the configuration file entry for telnet could be replaced by

```
telnet stream tcp nowait root /usr/bin/tcpd in.telnet
```

The program executed is now `/usr/bin/tcpd`. This is the TCP wrapper executable. The process executing the wrapper will still be called `in.telnet`. Within this wrapper,

you can perform all the access control or logging you want. In the original application, wrappers were used for IP address filtering (Chapter 17). Because the wrapper knows the directory it is in, i.e. `/usr/bin`, and its own name, i.e. `in.telnet`, it can then call the original server program, i.e. `/usr/bin/in.telnet`. The user will see no difference and receives exactly the same service as before.

Lesson
Adding another level of indirection is a powerful tool in computer science. In security, it can be used to attack systems and to protect systems. By inserting a TCP wrapper between the `inetd` daemon and the server program, you are able to add security controls without changing either the source code of the daemon or the source code of the server program.

The beauty of this example is its generality. The same principle can be used to protect a whole set of Unix network services.

Lesson
TCP wrappers combine a fundamental design principle, controlled invocation, and an elegant trick that makes it possible to add security checks to services without having to change the programs that call these services. This is the ideal situation when you have to retrofit security into an existing system.

7.7 MANAGEMENT ISSUES

We will quickly run through some issues that are relevant to managing the operational security of Unix systems.

7.7.1 Managing the Superuser

The root account is used by the operating system to perform its own essential tasks but also for certain other system administration tasks. Because superusers are so powerful, they are also a major weakness of Unix. An attacker achieving superuser status can effectively take over the entire system.

An attacker who is able to edit `/etc/passwd` can become superuser by changing its UID to 0, so the files `/etc/passwd` and `/etc/group` have to be write protected. To reduce the impact of a compromise, separate the duties of the systems manager, e.g. by having

special users such as `uucp` or `daemon` to deal with networking. If one of these special users is compromised, not all is lost. Systems managers should not use root as their personal account. When necessary, change to root can be requested by typing `/bin/su` (without specifying a username). The operating system will not refer to a version of `su` that has been put in another directory. Record all `su` attempts in the audit log together with the user (account) who issued the command.

7.7.2 Trusted Hosts

In a friendly environment, it may be sufficient to authenticate a user only once although a number of different machines are being accessed. *Trusted hosts* in Unix support this mode of operation. Users from a trusted host can log on without password authentication. They only need to have the same username on both hosts. Trusted hosts of a machine are specified in `/etc/hosts.equiv`. Trusted hosts of a user are specified in the `.rhosts` file in the user's home directory.

Usernames must be synchronized across hosts, a task that becomes tedious as the number of hosts grows. (Vendor-specific configuration tools exist.) Once a host has been entered into `/etc/hosts.equiv`, all users on this host have access. Exceptions are difficult to configure.

7.7.3 Audit Logs and Intrusion Detection

Once a system has been installed and is operational, its security mechanisms should prevent illegal user actions. However, the protection mechanisms may not be adequate or flawed. Undesirable security settings may be mandatory to keep the system running. Therefore, further mechanisms are desirable to detect security violations or other suspicious events when they are happening or after they have happened. Some security-relevant events are recorded automatically in Unix log files:

- `/usr/adm/lastlog` – records the last time a user has logged in; this information can be displayed with the `finger` command.
- `/var/adm/utmp` – records accounting information used by the `who` command.
- `/var/adm/wtmp` – records every time a user logs in or logs out; this information can be displayed with the `last` command. To prevent this file from taking over all available memory, it may be pruned automatically at regular intervals.
- `/var/adm/acct` – records all executed commands; this information can be displayed with the `lastcomm` command.

The precise name and location of these files may be different on your Unix system. Accounting, turned on by the `accton` command, can also be used for auditing purposes. Further commands for observing a Unix system are `find`, `grep`, `ps`, `users`.

Most security-relevant events recorded in the log files above refer to a user. Hence, the log entry should include the UID of the process causing the event. How is auditing then affected by SUID programs? Such programs run with the UID of their owner, not with the UID of the user running the program. Hence, log entries should also include the real UID of the process.

Lesson
User identifiers are a security attribute that is used for two purposes: access control and accountability. It is not always possible to employ the same attribute for both purposes at the same time. As long as UIDs correspond to 'real' users, access controls based on permissions and auditing complement each other. Once you create special user identities to protect access to resources through SUID or SGID programs, you get an attribute that is of limited use in auditing.

7.7.4 Installation and Configuration

A crucial point in the life of an operating system is its installation. Operating systems have many security features and features affecting security, all of which may not be well documented. Historically, default settings favoured smooth installation and operation, giving too many privileges to the maintenance engineer or to the system manager. It is appropriate to restrict the system manager in the same way as any other user, and to separate the roles of system manager and security manager. Complex and badly documented features may make it very difficult to set up the system so that it effectively enforces the intended security policy. Unix by itself does little to ease the system manager's job.

- System managers have to be knowledgeable about all security-relevant files and about dangerous default settings that have to be changed after installation.
- When a system is being set up, security-relevant parameters are defined with standard Unix editing commands. Permissions on resources are set at a level closer to the operating system than to an application. For example, users are installed by editing files such as `/etc/passwd`. Protection of the `passwd` program is effected through commands like

```
chmod 4750 /bin/passwd chgrp staff /bin/passwd
```

- When auditing a system, Unix search commands are used. For example, the following instruction scans for accounts without a password:

```
awk -F: 'length($2) < 1 print $1' < /etc/passwd
```

SUID and SGID are found by

```
find  -type f ( -perm 2000 -o -perm 4000 ) -exec ls -ld ;
```

(Experienced Unix users take pride in crafting such commands but the average user would find it difficult to mange systems in such a fashion.)

- Access control policies are supported through simple discretionary access control. Structured protection can be implemented based on *group* membership and by using accounts with login disabled.

Thus, there is a place for add-on Unix security products, both for managing security features and for checking the current security status. Such tools search for known flaws such as weak passwords, bad permissions on files and directories, or malformed configuration files. These tools can be used by systems managers to detect vulnerabilities in the systems they manage, but they are not universally popular as they can be used by an attacker for the very same purpose.

7.8 FURTHER READING

This chapter has taken a snapshot of Unix security. For a fuller picture, there is an abundance of books and websites on Linux and Unix security. If you need specific information about a particular Unix version, consult the documentation provided by the manufacturers and the on-line documentation (*manual pages*) provided on your system.

Multi-level secure Unix systems are discussed in [197]. Security Enhanced (SE) Linux adds mandatory access control based on domains (types), with roles as a layer of indirection between users and domains [211]. Solaris has been supporting RBAC since version 8.

7.9 EXERCISES

Exercise 7.1 Check the on-line documentation for security-relevant commands. Find your own entry in `/etc/passwd` and check the permission settings on your files and directories.

Exercise 7.2 Create a subdirectory in your home directory and put a file `welcome.txt` with a short message in this subdirectory. Set the permission bits on the subdirectory so that the owner has `execute` access. Try to:

- make the subdirectory the current directory with `cd`;
- list the subdirectory;
- display the contents of `welcome.txt`;
- create a copy of `welcome.txt` in the subdirectory.

Repeat the same experiments first with `read` permission and then with `write` permission on the subdirectory.

Exercise 7.3 Which Unix command will list all world-writable files in your directories?

Exercise 7.4 How would you protect a `tty` device from other users?

Exercise 7.5 Can you capture Unix access control through UID, GID, and permissions within the framework of the VSTa abilities (Section 5.8.2)?

Exercise 7.6 Implement the Chinese Wall model and the Clark–Wilson model (Section 12.3) with the Unix security mechanisms.

Exercise 7.7 How would you set up the backup procedure to reduce security exposures?

Exercise 7.8 In your assessment, what are the strong and weak points of Unix security?. Write a short report on this topic (1000 words).

Chapter 8

Windows Security

Unix access control treats all objects uniformly as resources. In contrast, access control in Windows can be tailored to individual object types. We will therefore use Windows as an example to show how a finer-grained approach to access control might be structured. The emphasis is on 'might' as we do not use any particular version of Windows for reference. The fundamental principles of the Windows security architecture have been stable during its evolution. Features that have been added are the inheritance of ACEs in Windows 2000 (Section 8.4.2), User Account Control in Windows Vista (Section 8.5.2), and *OwnerRights* in Windows 7 (Section 8.3.2).

OBJECTIVES

- Introduce the basics of Windows security.
- Show how to use indirection to make access control more manageable.
- Show how inheritance of access rights in a directory can be managed.
- Move from identity-based access control to code-based access control.

8.1 INTRODUCTION

We will not try to give a complete overview of Windows security or guidance on how to make the best use of all its security features. Our main goal is to contrast security in Unix and Windows and to highlight features of general interest that illustrate fundamental issues of computer security. Windows provides considerable support for managing security. We will focus on principles and mention other aspects such as graphical interfaces for managing the security features only in passing. The efforts to secure Windows are reflected in successful certifications according to Common Criteria EAL4 (see Section 13.6). For example, a certificate[1] for various versions of Windows Server 2003 and Windows XP was issued in February 2008.

8.1.1 Architecture

The Windows system architecture is given in Figure 8.1. There is a separation between *user mode* (typically protection ring 3) and *kernel mode* (protection ring 0), just as in Unix. The *Hardware Abstraction Layer* (HAL) provides the interfaces to the computer hardware. The core operating system services, comprising the *Windows executive*, run in kernel mode. The executive includes the *Security Reference Monitor* (SRM), which is in charge of access control.

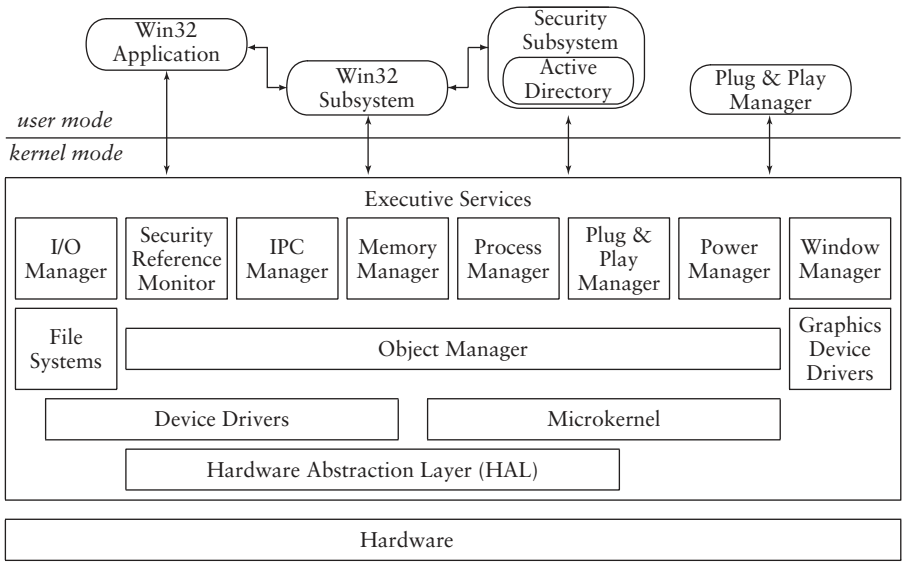


Figure 8.1: Windows 2000 System Architecture

[1] Validation Report CCEVS-VR-VID10184-2008.

User programs make application program interface (API) calls to invoke operating system services. Context switch and transition from ring 3 to ring 0 are handled by the Local Procedure Call facility. Device drivers (often third party products) run in kernel mode and are thus also security-relevant. Vulnerabilities in their code, such as buffer overruns, can be exploited by an attacker to take over a Windows system. The components of the *security subsystem* running in user mode are:

- *Logon process* (*winlogon*) – the process that authenticates a user when logging on.
- *Local security authority* (LSA) – involved at logon when it checks the user account and creates an *access token* (more of this in Section 8.2.2); it is also responsible for auditing functions.
- *Security account manager* (SAM): maintains the user account database that is used by the LSA during user authentication for local logon.

Passwords are stored in the SAM in hashed or encrypted form. Encryption has to be used if the system is to be able to automatically authenticate the user on remote machines.

8.1.2 The Registry

The *registry* is the central database for Windows configuration data. Entries in the registry are called *keys* (not to be confused with cryptographic keys). The tool for inspecting and modifying the registry is the *Registry Editor* (Regedit.exe or Regedt32.exe). At the top level, the registry has five important predefined keys:

- HKEY_CLASSES_ROOT – contains file extension associations; e.g. you could specify that .doc files are handled by Word.
- HKEY_CURRENT_USER – configuration information for the user currently logged on.
- HKEY_LOCAL_MACHINE – configuration information about the local computer.
- HKEY_USERS – contains all actively loaded user profiles on the system.
- HKEY_CURRENT_CONFIG – information about the hardware profile used by the local computer at system startup.

A registry *hive* is a group of keys, subkeys, and values in the registry. Security-relevant hives are:

- HKEY_LOCAL_MACHINE\ SAM;
- HKEY_LOCAL_MACHINE\Security;
- HKEY_LOCAL_MACHINE\Software;
- HKEY_CURRENT_CONFIG;
- HKEY_USERS\DEFAULT.

The system can be tailored to user requirements in the registry and default protections are set. An attacker could modify the behaviour of the operating system by modifying registry entries. For example, registry keys can point to locations where the operating system automatically looks for certain executable files. In Windows this is called a *path*. If the permissions set for such a key are weak, e.g. write permission for *Everyone*, then an attacker can insert malicious software by modifying the path. Protecting the integrity of registry data is therefore a necessity. Removing the Registry Editor from all machines not used for system management is a first line of defence. Some security-relevant keys should not even be changed via the Registry Editor but only through specific utilities.

A potential pitfall when setting policies are undefined keys. Consider, for example, the key that specifies which users and groups can access the register remotely (Windows KB 314837):

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
SecurePipeServers\Winreg.

If the key exists, it will be consulted when a user requests remote access to the registry. If the key does not exist, no checks specific to remote access will be performed. Remote access will be treated exactly like local access to the registry.

8.1.3 Domains

Stand-alone Windows machines are usually administered locally by their users. Within organizations a more structured approach to system and security management is essential. As a user on a computer network, you would not like to log on over and over again when accessing a resource or a service on another machine. As an administrator in charge of a computer network, you would not want to configure the security settings individually on each and every machine. Windows uses *domains* to facilitate single sign-on and centralized security administration.

A domain is a collection of machines sharing a common user accounts database and security policy. Domains can form a hierarchy. Users then do not require accounts with individual machines but with the domain.

In a domain, one server acts as the *domain controller* (DC). Other computers then join the domain. *Domain admins* create and manage domain users and groups on the DC. The domain controller authority has information about user passwords and can act as a trusted third party when a user (or some other principal) authenticates itself to another entity. This constitutes a design decision for a centralized authentication (password management) service.

A domain can have more than one domain controller. Updates may be performed at any DC and are propagated using the *multimaster replication* model. Here, decentralization of services is used as a design principle to achieve better performance.

8.2 COMPONENTS OF ACCESS CONTROL

Access control in Windows is more complex than access control in a typical filesystem. It applies to objects such as files, registry keys, and Active Directory objects. Means for structuring policies are groups, roles, and inheritance. The next sections will identify the principals, subjects, and objects in Windows, explain where the access rules can be found, and how they are evaluated.

8.2.1 Principals

Principals are the active entities in a security policy. They are the entities that can be granted or denied access. In Windows principals may be local users, aliases, domain users, groups, or machines. Principals have a human-readable name (username) and a machine readable identifier, the *security identifier* (SID).

Users created by the LSA are local users. *Local principals* are administered locally and visible only to the local computer. Examples are the local system, i.e. the operating system, and local users. The human-readable name of a local user or alias has the form

principal = MACHINE\principal,

e.g. 'TUHH-688432\Administrators'. Local users and aliases can be displayed from the command line[2] with the commands

> *net user*
> *net localgroup*

Domain principals are administered by domain admins on a domain controller. They are seen by all computers on the domain. Examples are domain users and the *Domain Admins* alias. The human-readable name of a domain user, group, alias, or machine has the form

principal@domain = DOMAIN\principal,

e.g. 'diego@europe.microsoft.com = EUROPE\diego'. Domain users, groups and aliases can be displayed with the commands

> *net user /domain*
> *net group /domain*
> *net localgroup /domain*

There exist also *universal principals*, e.g. the *Everyone* alias. Information about principals is stored in accounts and user profiles. Local accounts are in the registry (under HKEY_USERS). Domain accounts are at the domain controller but cached locally. The user profile is stored in the filesystem under *Documents and Settings*\. Some pre-defined principals need not be stored anywhere.

[2]Click on *Start*, then on *Run*, enter *cmd*, and type the command.

Groups and Aliases

A SID is an individual principal. A (global) *group* is a collection of SIDs managed by the domain controller. A group has its own group SID, so groups can be nested. The members of a group can make use of privileges and permissions given to the group. Groups constitute an intermediate layer of control. Permissions for an object are given to a group. Users are then given access to the object by becoming members of this group.

An *alias* (local group) is a collection of user and group SIDs managed by a domain controller or locally by the LSA. Aliases cannot be nested. Aliases are placeholder principals implementing logical roles. An application developer can refer to an alias *Student*. At deployment time appropriate SIDs are assigned to this alias. Note, though, that aliases do not fit the technical RBAC definition of 'roles' as defined in Section 5.6.3.

Security Identifiers

The general format of a SID is S-R-I-S-S-. . . -RID, starting with the letter S followed by

- R: revision number (currently 1),
- I: identifier authority (48-bit),
- S: one to fourteen subauthority fields (32-bit),
- RID: relative 32-bit identifier, unique in authority's name space.

This data structure supports deeply nested organizational structures. In practice, such sophisticated structures were not encountered and the standard format of a SID is S-R-I-SA-SA-SA-RID. A newly created issuing authority gets a SID with identifier authority 5, followed by the number 21 and a 96-bit random number put into three subauthority fields.

The design principle is that authorities have (statistically) unique identifiers, SIDs include the identifier of the issuing authority (domain), so a SID cannot by mistake represent access rights in the scope of some other domain.

SIDs for users and groups are unique and cannot be assigned again to another user or group. The SID is constructed when a user account is created and is fixed for the lifetime of the account. As a pseudo-random input (clock value) is used in its construction, you will not get the same SID if you delete an account and then re-create it with exactly the same parameters as before. Hence, a principal cannot by mistake get permissions of a previous principal. Conversely, the new account will not retain the access permissions given to the old account.

The following list gives a few typical principals together with their SID.

- Everyone (World): S-1-1-0.
- SYSTEM: S-1-5-18; the operating system on a machine runs locally as S-1-5-18; to other machines in the domain the machine is known under a separate domain-specific SID.

- Administrator: S-1-5-21-<*local authority*>-500; a user account created during operating system installation.
- Administrators: S-1-5-32-544; built-in group with administrator privileges; it contains initially only the Administrator account.
- Domain admins: S-1-5-21-<*domain authority*>-512; global group that is a member of the Administrators alias on all machines in the domain.
- Guest: S-1-5-21-<*authority*>-501; the field <*authority*> is a 96-bit unique machine or domain identifier created when Windows or a domain controller is installed.

When a domain is created, a unique SID is constructed for this domain. When a workstation or a server joins a domain, it receives a SID that includes the domain's SID. (Machines use their SIDs to check whether they are in the same domain.) As SIDs cannot be changed, moving a domain controller between domains is not a simple administrative process. The machine has to be completely reinstalled and logically become a 'new' machine to receive a new SID and become a controller in a new domain.

8.2.2 Subjects

Subjects are the active entities in an operational system. In Windows, processes and threads are subjects. Security credentials for a process or a thread are stored in an *access token*:

User SID
Group & Alias SIDs
Privileges
Defaults for New Objects
Miscellaneous

The SIDs serve as identity and authorization attributes. The token also contains the union of all privileges assigned to these SIDs. The defaults for new objects include parameters like owner SID, group SID, and DACL (explained in Section 8.2.4). The miscellaneous entries include the logon session ID and the token ID. Some fields in a token are read-only, others may be modified. A new process gets a copy of the parent's token, with possible restrictions.

Once a token has been created it will not change even if a group membership or a privilege that had been valid at the time of creation is revoked. This makes for better performance and better reliability because a process can decide in advance whether it has sufficient access rights for a given task. However, policy changes only take effect when the token has expired. The access rights at time of check need not be the same as the access rights at time of use. TOCTTOU (recall the lesson in Section 4.1) is a well-known issue in computer security.

Privileges

Privileges control access to system resources. A privilege is uniquely identified by its programmatic name, e.g. *SeTcbPrivilege*, and also has a display name, e.g. 'Act as part of the operating system'. Privileges are assigned to users, groups and aliases on a per-machine basis. They are cached in tokens as locally unique identifiers (LUIDs). Privileges are different from access rights, which control access to *securable objects* (Section 8.2.4). Typical privileges are:

- backing up files and directories;
- generating security audits;
- managing and auditing security logs;
- taking ownership of files and other objects;
- bypassing traverse checking;
- enabling computer and user accounts to be trusted for delegation;
- shutting down the system.

User Authentication – Interactive Logon

Users can be authenticated by username and password, but other options are supported too, e.g. authentication using a smart card. When a user logs on to a machine, authentication is initiated by pressing the *secure attention sequence* CTRL+ALT+DEL which invokes the Windows operating system logon screen and provides a *trusted path* from the keyboard to the logon process (*winlogon.exe*). The login dialog box is generated by the Graphical Identification and Authentication (GINA) dynamic-link library (DLL). The logon process runs permanently (under the principal SYSTEM).

To thwart spoofing attacks always press CTRL+ALT+DEL when starting a session, even when the logon screen is already displayed. The secure attention key sequence generates calls to low-level functions that cannot be duplicated by application programs. This trusted path is only present when a machine is actually running Windows. A machine running some other operating system could simulate the Windows logon screen and mount a spoofing attack.

A *legal notice* may optionally be displayed as a warning message. Users have to acknowledge this message before logon can proceed. Users are then prompted for username and password. Username and password are gathered by the logon process and passed on to the LSA (*lsass.exe*). For local logon, the local LSA calls an *authentication package* that compares username and password against the values stored in the account database. When a match is found, the SAM returns to the LSA the user's SID and the security ID of any group the user belongs to. Domain logon uses Kerberos and the user is authenticated by the LSA on a domain controller. The LSA then creates an access token containing the user's SIDs and privileges and passes the token to the logon process.

Creating Subjects and Network Logon

In the next step, the logon process starts a shell (*explorer.exe*) in a new *logon session* under the user (principal) that has been authenticated and attaches the access token to this process. The shell spawns processes to the same logon session (Figure 8.2). These processes are the *subject* for access control purposes. Logging off destroys the logon session and all processes in it.

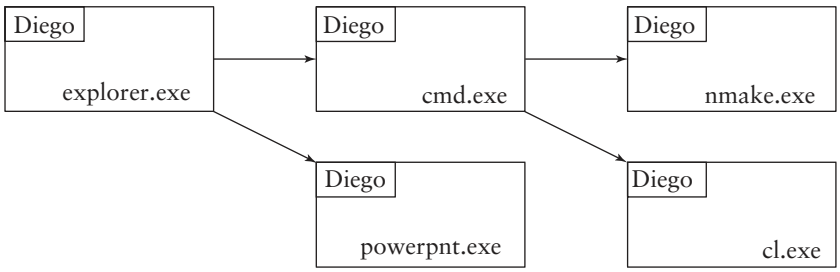


Figure 8.2: Example: Processes in a User Session

A process can spawn a new local process (subject) by calling *CreateProcess*. The new process gets a copy of parent's token. Each process has its own token. Different processes within a logon session can have different tokens. Threads can be given different tokens.

The user's network credentials, e.g. the password, are cached in the *interactive logon session*. Processes can then create *network logon* sessions for that user at other machines, i.e. automatically authenticate the user on a remote machine. Network logon sessions do not normally cache credentials. In Windows machines are principals and can have a machine account with password in a domain. Thus, a domain controller can also authenticate machines.

8.2.3 Permissions

A *permission* is an authorization to perform a particular operation on an object. *Access rights* correspond to the operations that can be performed on an object. The *standard access rights* applying to most types of objects are:

- DELETE – delete the object;
- READ_CONTROL – read access (to security descriptor) for owner, group, DACL;
- WRITE_DAC – write access to DACL;
- WRITE_OWNER – write access to owner;
- SYNCHRONIZE – allows a process to wait for an object to enter the signalled state.

GENERIC_EXECUTE	FILE_READ_ATTRIBUTES STANDARD_RIGHTS_EXECUTE SYNCHRONIZE
GENERIC_READ	FILE_READ_ATTRIBUTES FILE_READ_DATA FILE_READ_EA STANDARD_RIGHTS_READ SYNCHRONIZE
GENERIC_WRITE	FILE_APPEND_DATA FILE_WRITE_ATTRIBUTES FILE_WRITE_DATA FILE_WRITE_EA STANDARD_RIGHTS_WRITE SYNCHRONIZE

Table 8.1: Mapping of Generic Access Rights for Files and Directories

To display permissions for a file, click on *File*, then on *Properties*, and finally on *Security*. In Windows specific access rights can be tailored to each class of objects. In this way, access rights can be adapted to the application requirements, but developers would have to remember numerous specific rights. To address this issue, *generic access rights* provide an intermediate description level. Each class of objects has a mapping from the generic access rights onto real access rights so there is no need to remember class-specific permissions. Table 8.1 shows the mapping of generic access rights for files and directories. The generic access rights are:

- GENERIC_READ
- GENERIC_WRITE
- GENERIC_EXECUTE
- GENERIC_ALL

Access Mask

The desired operation in an access request (requested rights) and the granted access rights are internally given as 32-bit *access masks*. The bits in an access mask are assigned as follows:

0–15	specific rights defined for the object type the request refers to
16-22	standard rights
23	access system security, required to access a SACL
24–27	reserved
28	generic all
29	generic execute
30	generic write
31	generic read

The mapping of the standards rights is given as:

16	DELETE
17	READ_CONTROL
18	WRITE_DAC
19	WRITE_OWNER
20	SYNCHRONIZE

Extended Rights

In Active Directory, it is also possible to define access control on operations. Such access controls are called *extended rights*. Typical examples are *Send-As* and *Receive-As*, allowing the user to send mail or receive mail at a given mailbox. Extended rights are not defined by an access mask but by a *globally unique identifier* (GUID) corresponding to a *controlAccessRight* object. The list of extended rights is not fixed. Developers can create new extended rights for custom operations.

8.2.4 Objects

Objects are the passive entities in an access operation. In Windows there are *executive objects* such as processes or threads, Active Directory objects, filesystem objects, registry keys, devices such as printers, and more. In addition, developers can define their own *private* objects. Securable objects have a *security descriptor*. Security descriptors for built-in objects are managed by the operating system. Security descriptors for private objects have to be managed by the application software. Creating private securable objects can be a tedious task but enables highly granular access control.

Active Directory

Active Directory can be viewed as a tree of *typed* objects (Figure 8.3). *Containers* are objects that may contain other objects. Active Directory can be dynamically extended by adding new object types or new properties to existing object types. Thus you can tailor the object types to your own requirements. Each *object type* has specific properties and a

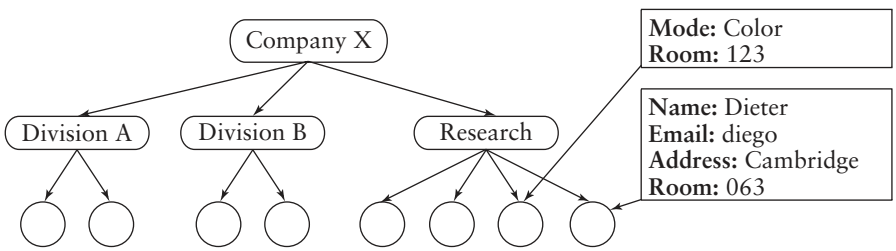


Figure 8.3: A Directory Tree

unique GUID; each property has its own GUID. In Figure 8.3 an 'employee' object with properties **name**, **email**, **address**, and **room** and a 'printer' object with properties **mode** and **room** are highlighted.

Logically related objects of different type can be put in the same container. This is useful for managing resources in an organization because you do not need different structures for different types of objects, and because you can use the directory structure to define general access policies for containers and let the objects within the container inherit the policy (see Section 8.4.2).

Security Descriptor

The security descriptor has the structure:

Owner SID
Primary Group
DACL
SACL

The *Owner SID* field indicates the owner of the object. Objects get an owner when they are created. Ownership can also be obtained via the privilege 'Take ownership of files and other objects' (*SeTakeOwnershipPrivilege*). In earlier versions of Windows, the owner always had READ_CONTROL and WRITE_DAC permission. In Windows 7, the *OwnerRights* SID was introduced to have more flexibility in assigning rights to the owner. The *Discretionary Access Control List* (DACL) determines who is granted or denied access to the object. The *System Access Control List* (SACL) defines the audit policy for the object. The *Primary Group* is included for POSIX compliance.

8.3 ACCESS DECISIONS

Each object type has an *object manager* that handles the creation of objects and verifies that a process has the right to use an object. Active Directory, for example, is the object manager for directory objects. For access control, the object manager calls an access decision function implemented by the Security Reference Monitor. The SRM, the *policy decision point*, returns a yes/no answer to the object manager, the *policy enforcement point*.

In general, access control decisions consider the subject requesting access, the object to which access is requested, and the desired access right. Not all three parameters need be considered in all circumstances. The credentials of the subject, including its principal, are stored in its token. The security attributes of an object are stored in its security descriptor. The desired access operation is given as an access mask.

The desired access is compared against the subject's token and the object's security descriptor when a handle to the object is being created, not at access time (TOCTTOU). Thus, changing a file DACL does not affect open file handles. This design decision leads to better performance and better reliability as all access control checks are made in advance, before the process starts a task.

8.3.1 The DACL

The DACL in the security descriptor is a list of *access control entries* (ACEs). The ACE format is:

type
inheritance and audit flags
access mask
ObjectType
InheritedObjectType
SID: principal the ACE applies to

The type determines whether the ACE is used for allowing, denying, or monitoring access requests. The generic types are:

- Access-denied (in DACL to deny access);
- Access-allowed (in DACL to allow access);
- System-audit (in SACL to monitor access).

There exist also object-specific types for allowing, denying, and monitoring access. Object-specific types can be applied to objects, properties, property sets, or extended rights. *ObjectType* is a GUID defining an object type. Applications can now include the *ObjectType* of objects in their access requests. For a given request, only ACEs with a matching *ObjectType* or without an *ObjectType* are evaluated. For example, to control read/write access on object property, put the GUID of the property in *ObjectType*. To control create/delete access on objects, put the GUID of the object type in *ObjectType*. Object-specific ACEs are also used for controlling inheritance of ACEs (Section 8.4.2).

An ACE for a web directory is shown below that grants users permission to set their own homepage. This policy applies to all users so we use the PRINCIPAL_SELF SID (S-1-5-10) as a placeholder. The application creating the homepage will supply the current user's SID when making an access request.

ACE1	
Type:	ACCESS_ALLOWED_OBJECT_ACE
ObjectType:	GUID for web homepage
InheritedObjectType:	GUID for User Account objects
Access rights:	write
Principal SID:	PRINCIPAL_SELF

The next ACE allows Server Applications to create RPC end points in any container of type RPC Services. The ACE will be inherited into any container of type RPC Services.

ACE2	
Type:	ACCESS_ALLOWED_OBJECT_ACE
ObjectType:	GUID for RPC Endpoint
InheritedObjectType:	GUID for RPC Services
Access rights:	create child
Principal SID:	Server Applications

8.3.2 Decision Algorithm

When a subject requests access to an object, the Security Reference Monitor takes the subject's token and the object's ACL and the desired access mask to determine whether the requested access should be granted. It first checks if a DACL exists. If there is no DACL, i.e. a so-called NULL DACL, no further checks are performed and access is granted.

Otherwise, the algorithm next checks whether the token lists the owner of the object. If this is the case but if there are no ACEs for the *OwnerRights* SID, the owner is granted READ_CONTROL and WRITE_DAC rights. If these rights are sufficient for the request, access is granted and the decision algorithm terminates. If there are ACEs for the *OwnerRights* SID, these ACEs will be considered in due order by the decision algorithm and define the rights of the owner. The DACL is then processed entry by entry. Permissions are *accumulated* by building a *GrantedAccess* mask. The SID in an ACE is compared with the subject's SIDs. The following three cases are possible:

1. The SID in the ACE does not match a SID in the token; the ACE will be skipped.
2. The SID in the ACE matches a SID in the token, the ACE is of type *Access-denied*, and the access mask contains a requested access right; access will be denied and no further checks take place.
3. The SID in the ACE matches a SID in the token and the ACE is of type *Access-allowed*; the access mask in the ACE is added to *GrantedAcesss*. If *GrantedAcess* now contains all the permissions in the requested access mask, access is granted and no further checks take place; otherwise, the search goes on.

Access is denied if the end of the DACL is reached and the granted mask is not equal to the requested mask. Thus, access will always be denied if there is an empty DACL and access will always be granted if there is no DACL.

For negative ACEs to take precedence over positive ACEs, they must be placed at the top of the DACL. As you will see in the next section, to achieve a finer granularity of access control one might place negative ACEs also after positive ACEs.

Lesson
Operating systems often store access control information in different places. It is important to know in which order checks are performed. Sometimes, as in Unix, only the first matching access control entry is consulted. At other times, more specific entries coming later can overrule a previous entry. Finally, you have to know how the operating system reacts if it finds no entry matching an access request.

8.4 MANAGING POLICIES

Windows access control can be used in several ways, with varying levels of granularity and complexity. Access control based on the principal requesting access is known as *impersonation* because the process 'impersonates' the user SID of its token. This is a coarse method but simple to implement. Impersonation is a typical operating systems concept and does not work well at the application level. *Role-centric* access control uses groups and aliases to give a process suitable access rights for its task. In *object-centric* access control, application-level objects get a security descriptor. This method facilitates fine-grained access control, but matters can get complex at the same time.

8.4.1 Property Sets

To ease administration, it is possible to collect the properties of an object type in property sets. Instead of ACEs for all the properties of an object type, you need only one ACE that refers to the property set. A property set is identified by its GUID. In an access request, a list of properties can be passed to the reference monitor and a single check against the property set returns the result for each property. As a further advantage, changes to the properties of an object type do not force us to change the ACL.

8.4.2 ACE Inheritance

It would be tedious to specify access rules by hand each time a new object is created. Default settings for security attributes automate this process. In Windows, default settings are derived from the subject making the request (Section 8.2.2) and from the container (directory) the object is being placed in. In Active Directory a container may contain objects of different types, therefore a selective inheritance strategy is supported.

Inheritance in Active Directory is controlled through inheritance flags that indicate whether an ACE has been inherited, and through the *InheritedObjectType*. ACEs are inherited from the container at the time a new object is created. When a new object is created only ACEs with a matching *InheritedObjectType* or without an *InheritedObjectType* are copied into its ACL. This is illustrated in Figure 8.4.

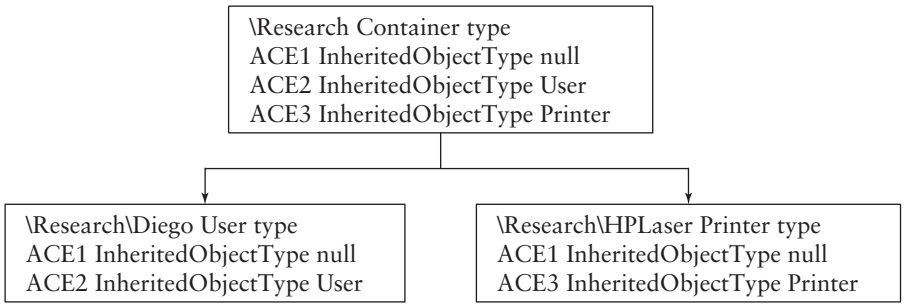


Figure 8.4: Type-Specific Inheritance within a Container

Changes to the container made later have no immediate effect on objects already in it. Such a strategy is known as *static inheritance*. If you change access permissions on a container and want to let the changes filter through to its content, you have to run a *propagation algorithm*. This algorithm should be *idempotent*: applying the propagation algorithm a second time causes no changes in access rights. Static inheritance has performance advantages and more predictable behaviour compared to dynamic inheritance, where any changes to the container would automatically be applied to all its objects.

When setting permissions on a container you usually do not know yet which objects will eventually be placed in it. In particular, you might not know the owners of these objects, but many access policies give the owner of an object specific rights. Hence, you have to be able to specify the rights an owner would inherit without already naming the owner. For this purpose, Windows has a special placeholder SID, CREATOR_OWNER. This SID is replaced in an inherited ACE by the owner's SID when a new object is created.

Inheritance Flags

Inheritance flags further specify how ACEs are inherited. The following flags are defined:

- INHERITED_ONLY_ACE – ACE can propagate to child objects, ignored by access checking mechanism;
- OBJECT_INHERIT_ACE – inherited by all sub-objects that are not containers;
- CONTAINER_INHERIT_ACE – inherited by sub-objects that are containers;
- NO_PROPAGATE_INHERIT – inherited only by children, but not propagated further.

Exceptions to Rules

In practice, there will be exceptions even for the best-thought-out general rules. So, we want a scheme where it is easy both to define and apply general rules, and to define

exceptions to those rules. As ACEs are evaluated in the order they appear in the DACL, placing locally added ACEs in front of inherited ACEs implements a policy where specific entries take precedence over more generic entries. In consequence, ACEs from closer containers are placed in front of more distant containers. Thus, it is possible for a positive ACE to appear before a matching negative ACE, as shown in Figure 8.5. There, ACE2 is defined directly for *Letter_A* in the container *Documents* and therefore appears before the inherited ACE. Because ACE1 is only inherited to objects of type *Letter*, it is not inherited by *Invoice_A*.

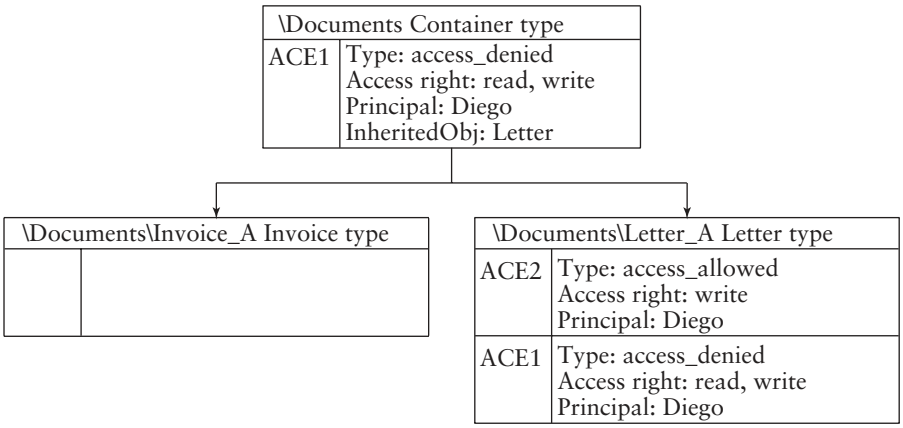


Figure 8.5: Locally Added ACEs are Placed in Front of Inherited ACEs

Placing more specific entries in front of more general entries is one way of implementing exceptions to general rules. Windows has a further mechanism for creating exceptions. Setting the flag SE_DACL_PROTECTED in the security descriptor of an object blocks inheritance of ACEs altogether, and you are starting with a clean slate when setting permissions on an object. In Figure 8.6, this flag is set in the descriptor of *Letter_A* and its DACL only contains a locally defined ACE2.

8.5 TASK-DEPENDENT ACCESS RIGHTS

So far access control has (implicitly) referred to users. The SIDs in the security token of a process are initially set when a user logs in and then passed on to other processes within the session. Access control based only on user identities makes life cumbersome for privileged users. The powerful privileges assigned to them are necessary for certain tasks, but might be dangerous in other circumstances. For example, a system administrator does not need and should not use administrator privileges when performing standard user actions such as surfing on the web (to learn about the latest security vulnerabilities). A conscientious

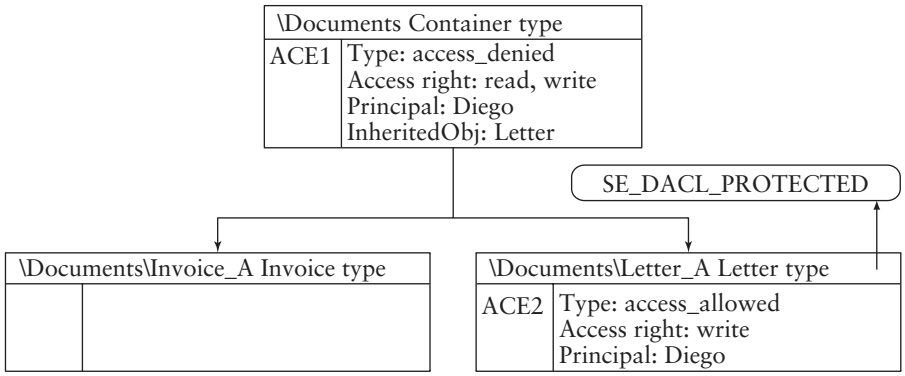


Figure 8.6: SE_DACL_PROTECTED Blocks Inheritance of ACEs

system administrator would thus log on as a standard user and explicitly switch to the administrator role (log in again, use features such as *Run As*) when necessary.

To avoid this hassle, access control should also refer to the task being performed. It is today quite usual but not particularly helpful to refer to tasks (and programs) executed with limited privileges as 'untrusted'. This misses the point. We are just adhering to good security engineering practices and follow the principle of *least privilege*. Code is only running with the permissions it needs to do its job.

8.5.1 Restricted Tokens

Task-dependent least privilege can be achieved with *restricted tokens*. Restricted tokens are a hypothetical extension of Windows access control proposed in [218]. They are constructed by removing privileges from a given access token. This could be done by:

- removing privileges from the token;
- disabling groups – group SIDs are not deleted but marked as USE_FOR_-DENY_ONLY;
- adding a *restricted* SID to the token – a process with a restricted token gets access only if the SID and the restricted SID are granted access.

Disabling groups can be useful when a server thread *impersonates* a client, i.e. runs in the context of the client's access token. A deny-only SID would disable access rights granted to the client that might be dangerous when performing the task at hand. In Figure 8.7 a process with restricted token requests read access to objects with three different DACLs. In case (a) both the principal SID *Diego* and the restricted SID *MyApp* have read

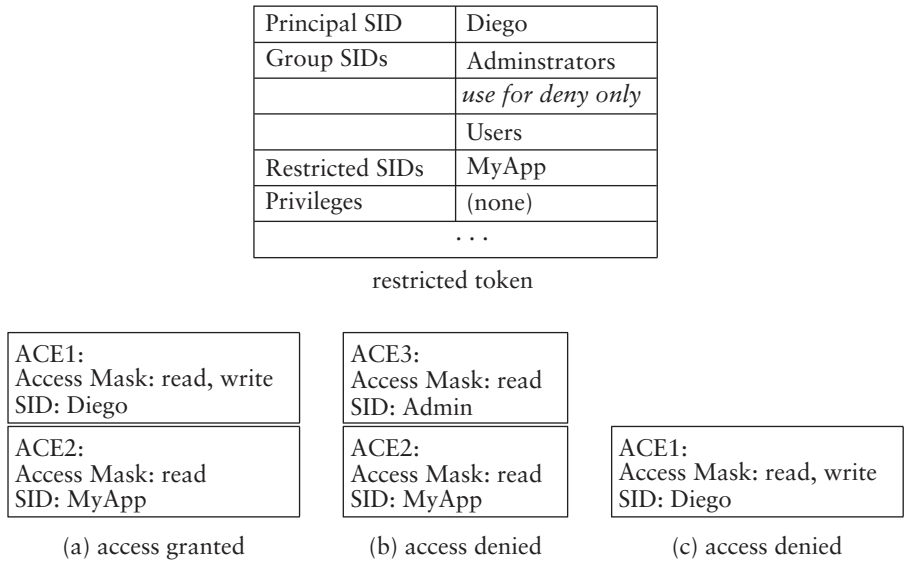


Figure 8.7: Access with Restricted Tokens

permission, so access is granted. In case (b) the *Admin* group SID has been marked for deny-only, so ACE3 will be skipped. Access will be denied because the principal does not have the required right. In case (c) the restricted SID *MyApp* has no read permission, so access is denied although the principal would have the required right.

To restrict the access rights of a program we may create a restricted SID representing this program. Our example uses a restricted SID for *MyApp*. This SID has to be entered into the DACLs of all objects the program should have access to. Alternatively, restricted SIDs may be created for each object type and be added to the restricted tokens of subjects that are permitted access.

8.5.2 User Account Control

Restricted tokens would – in theory – be an option for limiting the access rights of a user depending on the application that is running. In practice, sensible security policies may be hard to come by and restricted tokens might be a tool too difficult to master. Vista implements only a limited version of task-dependent access control. *User Account Control* (UAC) is intended for administrators who also perform tasks as a standard user. When a user in an administrator group logs in, two tokens (admin, user) are created. When the session starts, the standard user token is used. The administrator token is used only when the user attempts to perform an administrative task.

Lesson
Less can be more. Too many options for setting policies can be confusing.

8.6 ADMINISTRATION

We conclude this chapter with a few further remarks on managing Windows security.

8.6.1 User Accounts

The SAM keeps security-relevant information about users in the user account database. User accounts are edited using the *User Manager for Domains* utility. They can be displayed from the command line with net user *username*. Among others, the following fields can be defined:

- *Username* – the unique name used for logon.
- *Full name* – the name of the user owning the account.
- *Expiration date* – by default, accounts do not have an expiration date.
- *Password dates* – time password was last changed, time password expires (you can force users to change their password at the next logon by expiring the current password), time from when password can be changed; you can also indicate whether users may change their passwords.
- *Logon hours* and *workstations* – you can specify when the user is allowed to log in and from which machines the user is allowed to log in. The *forcibly disconnect remote users from server when logon hours expire* setting determines whether a user is thrown out outside hours or allowed to continue with an existing session. In the latter case, only new logons are prevented.
- *User profile path* and *logon script name* – the profile defines the user's desktop environment, i.e. program groups, network connections, screen colours, etc. The logon script is a batch file or executable file that runs automatically when a user logs on.
- *Home directory* – you can also specify whether the home directory is on the local machine or on a network server.
- *Local and global groups* – groups the user is member of.

8.6.2 Default User Accounts

Windows supports security management through default accounts. You have already seen some examples in Section 8.2.1. There exist three types of default user and group accounts:

- *predefined accounts* – installed with the operating system;
- *built-in accounts* – installed with the operating system, application, and services;
- *implicit accounts* – created implicitly when accessing network resources.

Default users and groups created by the operating system can be modified, but not deleted. *LocalSystem* is a built-in account used for running system processes and handling system-level tasks. Users cannot log in to this account, but certain processes can do so.

Administrator and *Guest* are predefined accounts installed locally. The *Administrator* account cannot be removed or disabled. It has complete access to files, directories, services, and other facilities. Although access permissions on files and directories can be set so that *Administrator* does not have access, the *Administrator* would still be able to change the access permissions and then get access after all. By default, the *Administrator* account for a domain is a member of the groups *Administrators*, *Domain Admins*, *Domain Users*, *Enterprise Admins*, *Schema Admins*, and *Group Policy Creator Owners*.

In a domain, the local Administrator account is primarily used when the system is first installed. Once installation is completed, the actual administrators can be made members of the *Administrators group*. Thus, individual administrator privileges can be revoked more easily.

The *Guest* account is intended for users who only need occasional access. Permissions can be given to this account as to any other user account. When Windows is installed, the *Guest* account is disabled.

Built-in groups have predefined user rights and permissions and provide another level of indirection when assigning access rights to users. A user obtains standard access rights by becoming a member of such a built-in group. Typical examples of built-in groups are *Administrators*, *Backup Operators*, *User*, or *Guests*. System managers are advised to stick to the built-in groups when implementing their security policies and to define groups with different permission patterns only if there are strong reasons for doing so.

A number of *predefined groups* are installed with Active Directory domains. Furthermore, there exist *implicit groups* that can also be used for efficient definition of access permissions:

- *Everyone* (S-1-1-0) – contains all local and remote users, including *Guest*; this group can be used to grant or deny permissions to all users.
- *Interactive* (S-1-5-4) – contains all users logged on locally.
- *Network* (S-1-5-2) – contains all users logged on over the network.

- *LocalSystem* (S-1-5-18) – the operating system.
- *CreatorOwner* (S-1-3-0) – placeholder in an inheritable ACE, replaced with the SID of the current owner.

8.6.3 Audit

Windows records security-relevant events in the security log. Security-relevant events to be logged when access to an object is requested can be defined in the SACL of the object. Entries in the log file are generated by the Security Reference Monitor. Security-relevant events typically include valid and invalid logon attempts, privilege use, and events such as creating, deleting, or opening a resource (file). The events to be logged can be selected and displayed using the *Event Viewer*. A maximum size of the audit log can be set, as can the number of days entries have to be kept in a log. There are three *wrapping options* for defining the retention method when the log reaches its maximal size:

- *Overwrite events as needed* – any record can be overwritten to make room for new entries.
- *Overwrite events by days* [*x*] – entries older than the specified number of days may be overwritten.
- *Do not overwrite events* – old records are never overwritten; the log must be cleared manually before new events can be logged.

With the last two options, a system with the *Audit: Shut down system immediately if unable to log security audits* setting enabled shuts down automatically when the log is full. This is a way of meeting Orange Book C2 and Common Criteria certification requirements demanding that auditable events must never go unrecorded.

8.6.4 Summary

With its range of conceptual tools that support security administration, and in particular for managing security policies at the application level, Windows can be placed closer to the man-oriented end of the man–machine scale than Unix (Figure 8.8).

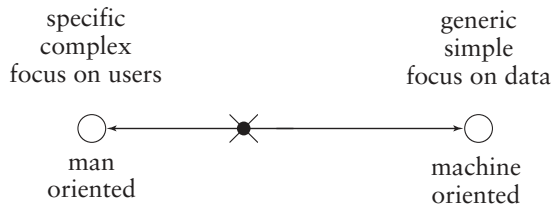


Figure 8.8: Windows Security on the Man–Machine Scale

8.7 FURTHER READING

Most security handbooks devoted to a specific operating system do not go below the surface of the security system and concentrate on existing features and their management. For references of this type, you have to check the latest publication lists, and the material on Microsoft's *technet.microsoft.com* website. An explanation of Windows access control that goes into greater depth is given in [47].

The reader interested in operating systems security case studies beyond Unix and Windows may turn to smart phone operating systems. An analysis of access control in Android can be found in [185]. Access control at the middleware layer (CORBA) is the subject of [39, 147].

8.8 EXERCISES

Exercise 8.1 Create a container for a lecture course. The container should hold objects of different types: course description, lecture notes, exercise solutions, student projects. Define groups and DACLs on the container and on objects so that:

- the course leader has access to all resources;
- students enrolled on the course have read access to lecture notes as well as read/write access to their own project;
- all students have read access to the course description;
- a student is nominated as tutor and gets read access also to exercise solutions;
- one given exercise solution is made available to all students on the course.

Exercise 8.2 In Unix and Windows access rights are defined for users and groups. To facilitate better security management, users are placed into groups. How do the two operating systems decide on an access request when users have fewer privileges than their group? How are access rights that have been given to a group withheld from individual members?

Exercise 8.3 Examine the 'controls at an intermediate layer' used in Windows.

Exercise 8.4 Unix UIDs can be set by the administrator. Windows creates randomized SIDs. Discuss the design rationale for taking control from the administrator and having randomized identifiers defined by the system.

Exercise 8.5 Default Windows accounts cannot be deleted. Examine the design rationale for this design decision.

Exercise 8.6 Examine the Windows documentation for event logs and describe how customized read access to event logs can be granted.

Exercise 8.7 You have created an object with a NULL DACL everyone should have access to. How could placing this object in a container affect your plan? How can you make sure that your intended policy is implemented? Find out what is needed to create an object with a NULL DACL.

Exercise 8.8 Design a strategy for archiving log files. Specify the retention method to use and discuss the main parameters a system manager has to consider when following your strategy.

Chapter 9

Database Security

A database does not merely store *data*, it provides *information* to its users. Database security is therefore not only concerned with the protection of sensitive data, but also provides mechanisms that allow users to retrieve information in a controlled manner. This remark highlights the two topics that distinguish database security from operating system security. You should control access to information more than access to data, and you should definitely focus control on the principals requesting access, notwithstanding the fact that the protection of data remains an important issue.

OBJECTIVES

- Analyze the security issues that are specific to database systems.
- Understand how views can be used for access control in a relational database.
- Appreciate the problem of protecting information in statistical databases.
- Give a brief outlook on the privacy issues that arise when processing personal data.

9.1 INTRODUCTION

A database is a collection of data, arranged in some meaningful way. A *database management system* (DBMS) organizes the data and gives users the means to retrieve information. If access to information were completely uncontrolled, a database would render a less useful service because it is quite likely that you would (be forced to) refrain from putting certain data into the database. For example, databases often hold information about individuals, be it employee records in a company, student records in a university, or tax records with the Inland Revenue. Many countries have enacted *privacy* legislation putting an organization maintaining such a database under an obligation to protect personal data. Therefore, from early on database security had an important place within computer security. It had a special place because database security is different from operating systems security. Here is the argument to back up this claim.

Operating systems manage *data*. Users invoke operating systems functions to create a file, to delete a file, or to open a file for read or write access. None of these operations considers the content of a file. Quite appropriately, the same is true for access control decisions made by an operating system. Decisions depend on the identity of the user, permissions defined for the file, access control lists, security labels, etc., but not on the content of the file. This is not due to some fundamental security theorem, it is simply a reasonable engineering decision.

Entries in a database carry *information*. Database users perform operations that consider the content of database entries. The most typical use of a database is perhaps a database search. Hence, it is fitting that the access control decisions made by a database management system also consider the content of database entries. A popular example is a salaries database where salaries above a given threshold have to be kept confidential. In summary, database security is placed more towards the user end of the man–machine scale (Figure 9.1).

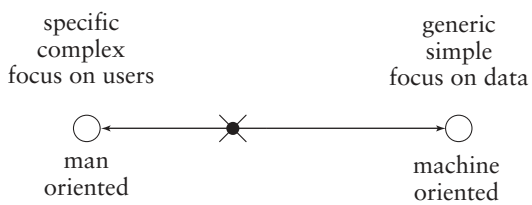


Figure 9.1: The Location of Database Security on the Man–Machine Scale

At first sight, protecting sensitive information in a database looks easy. In the salaries database, you simply add to the query statement a condition that checks the amount of the salary. If you know which data to protect, this approach is certainly feasible.

However, an intruder may be interested in many different pieces of information. The following list indicates the range of possible sources of information:

- exact data – the values stored in the database;
- bounds – lower or upper bounds on a numerical value such as a salary can already be useful information;
- negative result – e.g. if a database contains numbers of criminal convictions, then the information that a particular person does not have zero convictions is sensitive;
- existence – the existence of data may already be sensitive information;
- probable value – being able to guess some information from the results of other queries.

In the end, you have to guard against all eventualities. Protecting information becomes even trickier if the database permits statistical queries. A statistical query would, for example, return the sum of all salaries or the mean of all salaries. A clever combination of such queries can reveal the information you want to protect. This topic is taken up in Section 9.4.

You have been warned about the many routes via which sensitive information may leak from your database. You should, of course, take security seriously but not lose sight of the fact that your database has to serve a useful purpose. Overly restrictive policies denying access to data even though no sensitive information is disclosed reduce the value of the database. You thus have to strive for *precision*, i.e. protecting sensitive information while revealing as much non-sensitive information as permissible.

Database entries carry information about entities external to the computer system, such as warehouse stock levels, students' examination results, bank account balances, or available seats on a flight. Database entries should correctly reflect these external facts. Database security incorporates application-specific integrity protection to achieve internal and external consistency:

- *Internal consistency* – the entries in the database obey some prescribed rules. For example, stock levels cannot fall below zero.
- *External consistency* – the entries in the database are correct. For example, the stock levels indicated in the database match the stock levels in the warehouse. The database management system can help to avoid mistakes when updating the database, but you cannot rely on the DBMS alone to keep the database in a *consistent state*. This property is also called *accuracy*.

In the layered model of Section 3.4, the database management system can be placed in the services layer on top of the operating system. The DBMS has to meet database-specific security requirements that are not dealt with by the operating system. The DBMS can enforce security in conjunction with protection mechanisms within the operating system

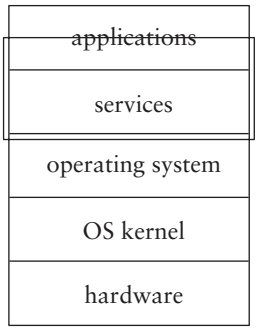


Figure 9.2: Location of Database Security

or on its own when there are no adequate controls in the operating system or when it becomes too cumbersome to involve the operating system. Moreover, the DBMS can be a tool for defining security controls in the application layer. Figure 9.2 captures the fact that database security includes security mechanisms at quite different layers of abstraction.

9.2 RELATIONAL DATABASES

The relational model is today the most widely used model for organizing a database. Once more, we assume that the reader is familiar with the underlying concepts and give only a brief introduction to relational databases. A detailed exposition is given in [76].

> A *relational database* is a database that is perceived by its users as a collection of *tables* (and tables only).

This definition of a relational database refers to its perception by users and not to its physical organization. This is also the appropriate level of abstraction for discussing database security.

Formally, a relation R is a subset of $D_1 \times \cdots \times D_n$ where $D_1, \ldots, D_n$ are the *domains* on n *attributes*. The elements in the relation are n-tuples $(v_1, \ldots, v_n)$ with $v_i \in D_i$, i.e. the value of the ith attribute has to be an element of D_i. The elements in a tuple are often called *fields*. When a field does not contain any value, we represent this by entering a special *null* value in this position. The meaning of null is 'there is no entry' and not 'the entry is unknown'.

The relations in Figure 9.3 could be part of a travel agent's database. The relation *Diary* has four attributes, *name*, *day*, *flight*, and *status*, with the following domains:

- *name* – all valid customer names;
- *day* – the days of the week, Mon, Tue, Wed, Thu, Fri, Sat, Sun;

Name	Day	Flight	Status
Alice	Mon	GR123	private
Bob	Mon	YL011	business
Bob	Wed	BX201	
Carol	Tue	BX201	business
Alice	Thu	FL9700	business

Flight	Destination	Departs	Days
GR123	THU	7:55	1-4--
YL011	ATL	8:10	12345-7
BX201	SLA	9:20	1-3-5-
FL9700	SLA	14:00	-2-4-6-
GR127	THU	14:55	-2-5-

Figure 9.3: The Relations *Diary* and *Flights*

- *flight number* – flight numbers, two characters and up to four numerals;
- *status* – business or private.

The standard language for retrieving and updating information in a relational database is SQL, the *Structured Query Language* [127]. The SQL operations for data manipulations are as follows:

- SELECT – retrieves data from a relation. The operation

```
SELECT Name, Status
       FROM  Diary
       WHERE Day = 'Mon'
```

  returns the result

Name	Status
Alice	private
Bob	business

- UPDATE – updates fields in a relation. The operation

```
UPDATE Diary
       SET Status = private
       WHERE Day = 'Sun'
```

  marks all journeys on a Sunday as private trips.
- DELETE – deletes tuples from a relation. The operation

```
DELETE FROM Diary
       WHERE Name = 'Alice'
```

  deletes all of Alice's journeys from *Diary*.
- INSERT – adds tuples to a relation. The operation

```
INSERT INTO Flights (Flight,Destination,Days)
       VALUES ('GR005', 'GOH', '12-45-')
```

  inserts a new tuple into *Flights* where the field *Departs* is still unspecified.

In all cases, more complicated constructions are possible. It is not the purpose of this book to explain all the intricacies of SQL and we will only give one example for demonstration. To find out who is going to Thule, run

```
SELECT Name
       FROM Diary
       WHERE Flight IN
             ( SELECT Flight
                      FROM Flights
                      WHERE Destination = 'THU' )
```

Relations are often visualized as *tables*. Attributes correspond to the columns in the table, using the names of the attributes as *headings* for the columns. The rows in the table correspond to the tuples (*records*) in the database. In the relational model, a relation cannot contain links or pointers to other tables. A relationship between tables (relations) can only be given by another relation. In a relational database, different kinds of relations may exist.

- *Base relations*, also called real relations, are named and autonomous relations; they exist in their own right, are not derived from other relations, and have 'their own' stored data.
- *Views* are named, derived relations, defined in terms of other named relations; they do not have stored data of their own.
- *Snapshots*, like views, are named, derived relations, defined in terms of other named relations; they have stored data of their own.
- *Query results* may or may not have a name; they have no persistent existence in the database *per se*.

For example, a snapshot of the *Diary* table that tells who is travelling when is defined by

```
CREATE SNAPSHOT Travellers
    AS SELECT name, day
       FROM   Diary
```

9.2.1 Database Keys

In each relation, you have to be able to identify all tuples in a unique way. Sometimes, a single attribute may be used as such an identifier. There may even be a choice of attributes which could serve this purpose. On the other hand, it also may happen that you need more than one attribute to construct such a unique identifier.

A *primary key* of a relation is a unique and minimal identifier for that relation. A primary key K of a relation R has to fulfil two conditions.

1. *Uniqueness*. At any time, no tuples of R have the same value for K.
2. *Minimality*. If K is composite, no component of K can be omitted without destroying uniqueness.

In the relation *Diary*, the combination of *name* and *day* can serve as the primary key (assuming that customers only go on one journey per day). In the relation *Flights*, the primary key is the flight number.

Every relation must have a primary key as no relation may contain duplicate tuples. This follows directly from the formal definition of a relation. When the primary key of one relation is used as an attribute in another relation, then it is a *foreign key* in that relation. In our example, the flight number as the primary key in the relation *Flights* is a foreign key in the relation *Diary*.

9.2.2 Integrity Rules

Within a relational database, you can define integrity rules that enforce *internal consistency* and help to maintain *external consistency (accuracy)*. Most of these rules will be specific to the application, but there are two rules that are inherent to the relational database model.

Entity Integrity Rule. No component of the primary key of a base relation is allowed to accept nulls.

This rule allows you to find all tuples in the base relations. The tuples in the base relations correspond to 'real' entities and we would not represent such an entity in the database if we could not identify it.

Referential Integrity Rule. The database must not contain unmatched foreign key values.

A foreign key value represents a *reference* to an entry in some other table. An unmatched foreign key value is a value that does not appear as a primary key in the referenced table. It is a reference to a non-existing tuple.

In addition to these two rules, further application-specific integrity rules may be desirable. Those integrity rules are important because they keep the database in a useful state. Typically, you would use them to do the following:

- *Field checks* – to prevent errors on data entry. In our example, we can guard against the insertion of arbitrary values in the *status* attribute of the *Diary* relation through a rule checking that the value entered is either *business* or *private*.
- *Scope checks* – in statistical databases, you may want rules for checking that the results of queries are computed over a sufficiently large sample. Looking ahead to the *Students* relation of Figure 9.4, you could define a rule that returns a grade average 67 by default if the sample size is not larger than three.

- *Consistency checks* – entries in different relations may refer to the same aspect of the external world and should therefore express a consistent view of this aspect. In our example, we can check the day a customer travels against the scheduled departure days of their flights. Alice's flight on Monday on GR123 is consistent with the fact that this flight departs on Mondays and Thursdays. Carol's booking on Tuesday on BX201 is inconsistent with the fact that this flight leaves on Mondays, Wednesdays, and Saturdays. An integrity rule comparing the respective fields could have stopped the travel agent from making this mistake.

Integrity rules of this kind are controls in the application layer. The DBMS provides the infrastructure for specifying and enforcing such rules. For example, an *integrity trigger* is a program that can be attached to an object in the database to check particular integrity properties of that object. When an UPDATE, INSERT, or DELETE operation tries to modify such an object, this program is triggered and performs its check.

We will pursue this topic no further than pointing to potential clashes between confidentiality and integrity. When the evaluation of an integrity rule requires access to sensitive information, you face the dilemma of either evaluating the rule incompletely (and incorrectly) to protect the sensitive information, or leaking some sensitive information to maintain consistency of the database.

9.3 ACCESS CONTROL

To protect sensitive information, the DBMS has to control how users can access the database. To see how controls could be implemented, you should remember that access to a database can be considered at two levels:

- data manipulation operations on base relations;
- compound operations such as views or snapshots.

Go back for a moment to Section 3.4.1. You can look at access control from two directions:

- restricting the operations available to a user, or
- defining the protection requirements for each individual data item.

In a DBMS, controls on compound operations regulate how users can work with the database. On the other hand, checks on the manipulation of base relations protect the entries in the database. By deciding on the type of access operations you want to control, you also influence the focus of the policies to be enforced. Conversely, the focus of your policies will suggest which type of operations to control. Whatever option you choose, there are two properties you ought to aim for:

- *completeness* – all fields in the database are protected;
- *consistency* – there are no conflicting rules governing the access to a data item.

A security policy is consistent if there is no element in the database that can be accessed in different ways which result in different access control decisions. Legitimate access requests should not be prevented, nor should there be ways of circumventing the specified access policy.

9.3.1 The SQL Security Model

The basic SQL security model follows a familiar pattern. It implements discretionary access control based on three entities:

- *users* of the database – the *user identity* is authenticated during a logon process, while the DBMS may run its own logon or accept user identities authenticated by the operating system;
- *actions*, including SELECT, UPDATE, DELETE, and INSERT;
- *objects* – tables, views, and columns (attributes) of tables and views; SQL3 further includes user-defined constructs.

Users invoke actions on objects. The DBMS has to decide whether to permit the requested action. When an object is created, a user is designated as its owner and initially only the owner has access to the object. Other users first have to be issued with a *privilege*. The components of a privilege are

(granter, grantee, object, action, grantable)

The two mainstays of the SQL security model are privileges and views. They provide the framework for defining application-oriented security policies.

9.3.2 Granting and Revocation of Privileges

In SQL, privileges are managed with the operations GRANT and REVOKE. Privileges refer to particular actions and can be restricted to certain attributes of a table. In our example, we allow two travel agents, *Art* and *Zoe*, to inspect and update parts of the *Diary* table:

```
GRANT SELECT, UPDATE (Day,Flight)
      ON TABLE Diary
      TO Art,Zoe
```

Privileges can be selectively revoked:

```
REVOKE UPDATE
       ON TABLE Diary
       FROM Art
```

A further feature is the granting of the right to grant privileges, implemented in SQL by the GRANT *option*. For example, having been granted privileges on table *Diary*, with

```
GRANT SELECT
      ON TABLE Diary
      TO Art
      WITH GRANT OPTION
```

travel agent *Art* may in turn grant privileges on the same table to *Zoe*, with

```
GRANT SELECT
      ON TABLE Diary
      TO Zoe
      WITH GRANT OPTION
```

When the owner of table *Diary* revokes the privileges granted to *Art*, all the privileges granted by *Art* also have to be revoked. Thus, revocation has to *cascade* and the information necessary to do so has to be maintained by the database system.

You should also note that once other users have been granted access to data, the owner of the data cannot control how information derived from these data is used, even if there is still some control over the original data. You can read data from a table and copy it into another table without requiring any 'write' access to the original table.

9.3.3 Access Control through Views

Views are derived relations. The SQL operation for creating views has the format

```
CREATE VIEW view_name [ ( column [, column ]... ) ]
       AS subquery
       [ WITH CHECK OPTION ];
```

You could implement access control in a relational database by granting privileges directly for the entries in base relations. However, many security policies are better expressed through views and through privileges on those views. The subquery in the view definition can describe quite complex access conditions. As a simple example, we construct a view that includes all business trips in the example relation *Diary*.

```
CREATE VIEW business_trips AS
       SELECT * FROM Diary
       WHERE Status = 'business'
       WITH CHECK OPTION;
```

Access control through views can justifiably be placed in the application layer. The DBMS only provides the tools for implementing the controls. Views are attractive for several reasons:

- Views are flexible and allow access control policies to be defined at a level of description that is close to the application requirements.
- Views can enforce context-dependent and data-dependent security policies.
- Views can implement controlled invocation.
- Secure views can replace security labels.
- Data can be easily reclassified.

Application-oriented access control can be expressed through views such as

```
CREATE VIEW Top_of_the_Class AS
       SELECT * FROM Students WHERE Grade <
       (SELECT Grade FROM Students WHERE Name =
       current_user());
```

to display only those students whose grade average is less than that of the person using the view, or

```
CREATE VIEW My_Journeys AS
       SELECT * FROM Diary
       WHERE Customer = current_user());
```

to display only those journeys booked by the customer using the view. Discretionary access control can be implemented by adding the access control table to the database. Views can now refer to this relation. In this way, you can also express access control based on group membership as well as policies regulating the users' rights to grant and revoke access rights.

Furthermore, views can define or refer to security labels. In our example, business flights to Thule can be marked as confidential by creating the view

```
CREATE VIEW Flights_at_CONFIDENTIAL AS
       SELECT * FROM Diary
       WHERE Destination = 'THU' AND Status = 'business';
```

Controlling read access through views poses no particular technical problem other than capturing your security policy correctly. The situation is different when views are used by INSERT or UPDATE operations to write to the database. First, there exist views that are not updatable because they do not contain relevant information needed to maintain the integrity of the corresponding base relation. For example, a view that does not contain the primary key of an underlying base relation cannot be used for updates. Secondly, even if a view is updatable, some interesting security issues remain. A travel agent who has access to the *Diary* database only through the view *business_trips* sees

Name	Day	Flight	Status
Bob	Mon	YL011	business
Carol	Tue	BX201	business
Alice	Thu	FL9700	business

Should the travel agent be allowed to update the view with the following operation?

```
UPDATE business_trips
       SET  Status = 'private'
       WHERE Name = 'Alice' AND Day = 'Thu'
```

The entry for `Alice` would then disappear from view. As a matter of fact, in this case the update will not be permitted because the definition of the view has specified the CHECK option. If the definition of a view includes WITH CHECK OPTION, then UPDATE and INSERT can only write entries to the database that meet the definition of the view. If the CHECK option is omitted, *blind writes* are possible.

A view is not only an object in the SQL security model, but can also be seen as a program. When a view is evaluated with the privileges of its owner rather than with the privileges of the user invoking the view, you have another method of implementing controlled invocation.

The access conditions in a view have to be specified within the limits of SQL. If this proves too restrictive, software packages (stored procedures) written in a more expressive language are another option for the DBMS to provide controlled access to the database. Users are granted execute privilege on the package which runs with the privileges of its owner.

Lesson
Controlled invocation can be found at any layer of a computer system. It is a principle equally useful in microprocessors as in database management systems.

So far we have presented the aspects that make views a useful security mechanism. Naturally, views have also their disadvantages:

- Access checking may become rather complicated and slow.
- View definitions have to be checked for 'correctness'. Do they really capture the intended security policy?
- Completeness and consistency are not achieved automatically; views may overlap or may fail to capture the entire database.
- The security-relevant part of the DBMS (the TCB) will become very large.

Views are suitable in a 'normal commercial' environment. They can be tailored to the application and require no modification of the DBMS. Definition of views then is part of the general process of defining the structure of the database so that it best meets the business requirements.

It may, however, become difficult to determine for individual data items who has access. Therefore, views are less suitable in situations where it is deemed necessary to protect the data items, rather than control the users' actions.

9.4 STATISTICAL DATABASE SECURITY

Statistical databases raise security issues not yet investigated in this book. The distinctive feature of a statistical database is that information is retrieved by means of *statistical (aggregate) queries* on an attribute (column) of a table. The aggregate functions in SQL are:

- COUNT – the number of values in a column;
- SUM – the sum of the values in a column;
- AVG – the average of the values in a column;
- MAX – the largest value in a column;
- MIN – the smallest value in a column.

The *query predicate* of a statistical query specifies the tuples that will be used to compute the aggregate, and the *query set* are the tuples matching the query predicate. In a nutshell, statistical databases pose the following security problem:

- The database contains data that are individually sensitive. Direct access to data items is therefore not permitted.
- Statistical queries to the database are permitted. By their very nature, these queries read individual data items.

In such a setting it becomes possible to *infer* information. We will show that it is no longer sufficient to police access requests individually. We are also taking a more pragmatic view of information flow. The confidentiality models in Chapters 11 and 12 try their best to stop any information flow whatsoever. In a statistical database, there must be some information flow from the data to their aggregate. We can only try to reduce it to an acceptable level.

The *Students* database of Figure 9.4 provides the examples in this section. Statistical queries on all attributes are allowed but individual entries in the *Units* and *Grade Ave*. Columns cannot be read directly. The statistical query

```
Q1 :  SELECT AVG(Grade Ave.)
      FROM   Students
      WHERE  Programme = 'MBA'
```

computes the grade average of all MBA students. The *query predicate* in the example is `Programme = 'MBA'`.

Name	Sex	Programme	Units	Grade Ave.
Alma	F	MBA	8	63
Bill	M	CS	15	58
Carol	F	CS	16	70
Don	M	MIS	22	75
Errol	M	CS	8	66
Flora	F	MIS	16	81
Gala	F	MBA	23	68
Homer	M	CS	7	50
Igor	M	MIS	21	70

Figure 9.4: The *Students* Relation

9.4.1 Aggregation and Inference

Two important concepts in statistical database security are aggregation and inference. *Aggregation* refers to the observation that the sensitivity level of an aggregate computed over a group of values in a database may differ from the sensitivity levels of the individual elements. You will mostly meet scenarios where the sensitivity level of the aggregate is lower than the levels of the individual elements. The reverse would be true when the aggregate is sensitive executive information derived from a collection of less sensitive business data.

The aggregate is another relation in the database, e.g. a view, so you can use the security mechanisms proposed in this chapter to control access to the aggregate. However, an attacker can exploit the difference in sensitivity levels to obtain access to the more sensitive items. The *inference problem* refers to the derivation of sensitive information from non-sensitive data. The following types of attack have to be considered:

- *Direct attack* – the aggregate is computed over a small sample so that information about individual data items is leaked.
- *Indirect attack* – this combines information relating to several aggregates.

- *Tracker attack* – a particularly effective type of indirect attack.
- *Linear system vulnerability* – a step beyond tracker attacks, using algebraic relations between query sets to construct equations which yield the desired information.

9.4.2 Tracker Attacks

We will now demonstrate how to employ statistical queries to elicit sensitive information from our *Students* relation. Assume that we know that Carol is a female CS student. By combining the legitimate queries

```
Q1 :  SELECT  COUNT(*)
      FROM    Students
      WHERE   Sex = 'F' AND Programme = 'CS'

Q2 :  SELECT  AVG(Grade Ave.)
      FROM    Students
      WHERE   Sex = 'F' AND Programme = 'CS'
```

we learn from Q1 that there is only one female CS student in the database so the value 70 returned by Q2 is precisely her grade average. The problem here is that the selection criteria define a set containing only one element. You could therefore allow a statistical query only if it covers a sufficiently large subset. However, we could simply query the complement by negating the selection criteria and obtain the same result as before from the difference between the result of the query applied to the entire database and the result of the query applied to the complement of the set we are really interested in. You therefore have to demand that not only the set of tuples considered by a query but also its complement are sufficiently large.

Unfortunately, even this is not good enough. Assume that each query set and its complement must contain at least three elements. The sequence of queries

```
Q3 :  SELECT  COUNT(*)
      FROM    Students
      WHERE   Programme = 'CS'

Q4 :  SELECT  COUNT(*)
      FROM    Students
      WHERE   Programme = 'CS' AND Sex = 'M'

Q5 :  SELECT  AVG(Grade Ave.)
      FROM    Students
      WHERE   Programme = 'CS'
```

```
Q6 :  SELECT  AVG(Grade Ave.)
      FROM    Students
      WHERE   Programme = 'CS' AND Sex = 'M'
```

returns the values Q3: 4, Q4: 3, Q5: 61, Q6: 58. All queries take into account a sufficiently large set of tuples, so they are not prohibited. However, by combining the four results we compute Carol's grade average as $4 \cdot 61 - 3 \cdot 58 = 70$.

You might feel that we were lucky in this case and were only able to construct this set of queries because Carol was the single female CS student. We will now show how to set up an attack in a systematic way. First, we need a tracker.

> A query predicate T that allows information to be tracked down about a single tuple is called an *individual tracker* for that tuple. A *general tracker* is a predicate that can be used to find the answer to any inadmissible query.

Let T be a general tracker and let R be a predicate that uniquely identifies the tuple r we want to probe. In our example, the predicate is `Name = 'Carol'`. We make two queries to the database using the predicates $R \vee T$ and $R \vee \text{NOT}\ (T)$. Our target r is the only tuple used by both queries. To make sure that both queries are admissible, we choose T so that the query set and its complement are large enough for the query to be permitted. A final query over the entire database gives us all the data to complete the attack. In our example

```
Sex = 'F' AND Programme = 'CS'
```

is an individual tracker for Carol, and `Programme = 'MIS'` is one of many general trackers. We proceed with the queries

```
Q7 :  SELECT  SUM(Units)
      FROM    Students
      WHERE   Name = 'Carol' OR Programme = 'MIS'

Q8 :  SELECT  SUM(Units)
      FROM    Students
      WHERE   Name = 'Carol' OR NOT (Programme = 'MIS')

Q9 :  SELECT  SUM(Units)
      FROM    Students
```

and obtain Q7: 75, Q8: 77, and Q9: 136. So Carol must have passed $(75 + 77) - 136 = 16$ units. Experience has shown that almost all statistical databases have a general tracker.

9.4.3 Countermeasures

The analysis of statistical inference attacks dominated the early literature on database security. Since then, researchers have shifted their attention to other areas, less because

complete and waterproof solutions have been found and implemented, but rather because they had to acknowledge the limits of countermeasures against inference attacks. Given those limitations, what can you realistically do about the inference problem?

First, you would suppress obviously sensitive information. This implies that you know which information is sensitive, which hopefully is the case, and that you know how this information can be derived. You would at least check the size of the query set before releasing the result of a statistical query.

Next, you could disguise the data. You could randomly swap entries in the database so that an individual query will give an incorrect result, even though the statistical queries would still be correct. Alternatively, you could add small random perturbations to the query result so that the value returned is close to the real value but not quite correct. As a drawback, these techniques reduce precision and usability. You would not want to randomly swap values in a medical database.

Some aggregation problems can be eased by taking care with the design of the database schema [156]. A *static* analysis of the structure of the database can reveal sensitive relationships between attributes. Such attributes are then placed in separate tables. A user with access to one table only is no longer able to correlate the attributes. Of course, a user with access to all relevant tables is still in a position to do so, but as the database manager you can be more *precise* when allocating privileges. In our example, the relationship between names and academic performance is sensitive. We replace the table *Students* by two tables, linked by a student identity number:

Name	ID
Alma	B13
Bill	C25
Carol	C23
Don	M38
Errol	C12
Flora	M22
Gala	B36
Homer	C10
Igor	M20

ID	Sex	Programme	Units	Grade Ave.
B13	F	MBA	8	63
C25	M	CS	15	58
C23	F	CS	16	70
M38	M	MIS	22	75
C12	M	CS	8	66
M22	F	MIS	16	81
B36	F	MBA	23	68
C10	M	CS	7	50
M20	M	MIS	21	70

The first table can now be classified at a sufficiently high level so that only authorized users can link names and academic performance.

Finally, observe that inference problems are caused not so much by single queries but by a clever combination of several queries. You could therefore track what the user knows. Possibly this gives the best security but it is also the most expensive option. User actions are recorded in an audit log and you perform a *query analysis* to step in if a suspicious

sequence of queries is detected. In the first place you have to know what constitutes suspicious behaviour. To tighten your protection even further, your query analysis will have to consider what two users, or a group of users, know together.

9.5 INTEGRATION WITH THE OPERATING SYSTEM

When you look at a database from the position of the operating system, you see a number of operating system processes and the memory resources that store the data entries. In many respects, the DBMS has similar duties to the operating system. It has to stop users from interfering with each other, and with the DBMS.

If you do not want to duplicate effort, you could give these tasks to the operating system. In such a set-up, the DBMS runs as a set of operating system processes. There are system processes for general database management tasks and each database user is mapped to a separate operating system process (Figure 9.5). Now the operating system can distinguish between users, and if you store each database object in its own file, then the operating system can do all the access control. The DBMS only has to translate user queries into operations the operating system understands.

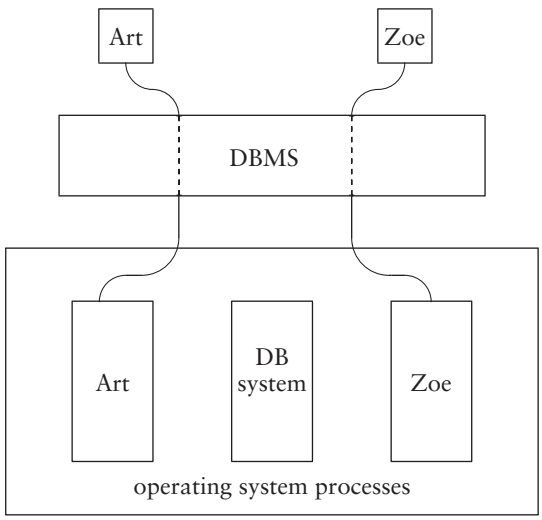


Figure 9.5: Isolation of Database Users by the Operating System

Allocating an individual operating system process to every database user wastes memory resources and does not scale up to large user numbers. Hence, you need processes that handle the database requests of several users (Figure 9.6). You save memory but the

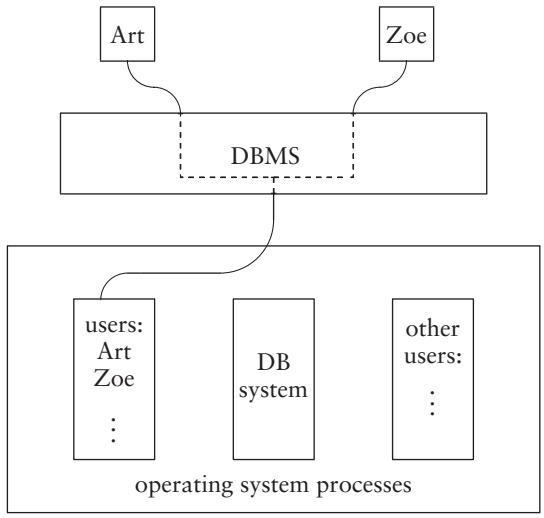


Figure 9.6: Isolation of Database Users by the DBMS

responsibility for access control now rests firmly with the DBMS. Similar considerations apply to the storage of database objects. If the objects are too small, having a separate file for each object is wasteful. Once the operating system has no access control functions with respect to database users, you are free to collect several database objects in one operating system file.

9.6 PRIVACY

Many organizations will for legitimate reasons store personal data on their customers, such as name, address, age, credit card numbers, meal preferences or other consumer habits. Data of this kind is usually protected by law. Compliance with legal and regulatory constraints is an important issue when maintaining such data.

International recommendations for the protection of personal data are the *OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data*. These guidelines state eight protection principles where the *data subject* is the individual the data refer to and the *data controller* maintains the database.

1. Collection limitation principle: There should be limits to the collection of personal data. Data should be obtained by lawful and fair means and, where appropriate, with the knowledge or consent of the data subject.
2. Data quality principle: Personal data should be relevant to the purposes for which they are to be used, accurate, complete and up to date.

3. Purpose specification principle: The purposes for which personal data are collected should be specified.
4. Use limitation principle: Personal data should not be disclosed or used for purposes other than those specified, except with the consent of the data subject or by the authority of law.
5. Security safeguards principle: Personal data should be protected by reasonable security safeguards.
6. Openness principle: There should be a general policy of openness about developments, practices and policies with respect to personal data.
7. Individual participation principle: An individual should have the right to obtain confirmation of whether or not the data controller has data relating to him, to challenge data relating to him, and to be given reasons if such a request is denied.
8. Accountability principle: A data controller should be accountable for complying with measures which give effect to the principles stated above.

Another important document is the EU Directive[1] on the protection of individuals with regard to the processing of personal data and on the free movement of such data. EU directives are not laws but have to be transformed into national law by EU member states. The Directive has a strong emphasis on user consent. Users should 'opt in' to indicate that they agree that their data are stored. The alternative is an 'opt out' policy where data subjects have to state explicitly that data should not be kept.

In the US, data protection is addressed by sectoral laws. For example, the Health Insurance Portability and Accountability Act 1996 is a federal law protecting patients' medical records and other health information provided to health plans, doctors, hospitals and other health care providers. The Act took effect in 2003 and had a considerable impact on database security, and on IT security in general.

Platform for Privacy Preferences

Websites may advertise their privacy policies. Users may, in theory, examine those policies before releasing personal data. In practice, this hardly happens. The Platform for Privacy Preferences [74] was therefore launched to automate the process of checking a user's privacy preferences against the stated policy of a website. Websites would express their data-collection and data-use practices in a standardized machine-readable XML format known as a P3P policy. Policies would explain what data are collected, for what purpose data will be used, and any opt-out or opt-in options for data uses.

On the client side, P3P user agents would be built into browsers or browser plug-ins, or run on proxy servers. The user agents would automatically make decisions on behalf of

[1] EU Directive 95/46/EC of the European Parliament and of the Council of 24 October 1995.

the user comparing a site's P3P policy with the privacy preferences set by the user. Users would not need to read the privacy policies at every site they visit.

P3P is a descriptive language. Users have to 'trust' that a website adheres to its stated policies. Data protection laws can stipulate that they have to. There is, however, a fundamental concern. Can automated actions by a user agent ever imply the user's consent as required by data protection laws? In the end, P3P had not found sufficient support from browser vendors to be widely adopted.

9.7 FURTHER READING

If you need to revise the relational database model, consult [76]. A substantial collection of material on database security has been compiled in [56]. An earlier but still very useful book on database security is [80]. In particular, this book is a good reference on statistical database security. A further useful source on this topic is [156]. All major database vendors maintain web pages with information on their products and with good introductions to database security.

9.8 EXERCISES

Exercise 9.1 Consider an accounts database with records *(customer name, account number, balance, credit rating)* and three types of user: *customer*, *clerk*, *manager*. Define an access structure, e.g. through views, so that

- customers can read their own account;
- clerks can read all fields other than *credit rating* and update *balance* for all accounts;
- managers can create new records, read all fields, and update *credit rating* for all accounts.

Exercise 9.2 Consider a database of student records that contains student names and the marks for all courses in the programme. Lecturers are provided with a view that shows all students on their course whose paper has not yet been marked. Should this view be defined WITH CHECK OPTION? Suggest general criteria for deciding whether to use the CHECK OPTION.

Exercise 9.3 All statistical queries on the *Students* relation (Figure 9.4) must have at least three tuples in their query set. Only AVG queries on the attribute *Grade Ave*. are allowed. Find a new general tracker and construct a tracker attack on Homer's grade average.

Exercise 9.4 In database security, you have seen examples of security controls placed in the application layer. What are the problems with this approach?

Exercise 9.5 Consider a database where aggregates are placed at a higher sensitivity level than the data items they are derived from. A user privileged to access the data items can potentially compute the aggregate by accessing the data items individually. How can you defend against such an attack?

Exercise 9.6 You are given a database where access rights can be defined separately for each row in a table. Access control will use the security mechanisms of the operating system. What are the consequences of this design decision with respect to the way the database objects are stored in operating system files? (As a yardstick, consider an operating system that uses 100 bytes of administrative data for each file and a database with 10 million records.) Is this a viable design decision?

Chapter 10

Software Security

Computer security makes the headlines when a critical vulnerability in some widely used software product is discovered. Attacks may target memory management flaws using a *buffer overrun* to manipulate the control flow at a layer below the programming abstractions used by the software developer. Attacks may target applications written in a *scripting language* and insert their own commands via user input. In both cases, the attacker manages to execute code with elevated privileges. The programming errors exploited are sometimes quite simple and their eradication might seem a trivial job. In some instances this assessment is true, but building complex systems is a challenging task and there is a long history of security bugs in software systems.

This chapter will analyze the causes of software vulnerabilities and defence options at a general level. Detailed instructions on writing secure code are beyond our scope and the reader is referred to the references listed at the end of the chapter.

OBJECTIVES

- Explain the basic causes that lead to software security failures.
- Discuss the dangers of abstractions.
- Present the defence options when designing software systems.
- Give an introduction to security testing.

10.1 INTRODUCTION

On a stand-alone machine, a personal computer in the true sense of the word, you are in control of the software components sending inputs to each other. On a machine connected to the Internet, hostile parties can provide input. Networking software is a popular target for attacks. It is built to receive external input and involves low-level manipulations of buffers. Dynamic web applications are a popular target as they construct code from external inputs. Software is secure if it can handle intentionally malformed input. Secure software is not the same as software with added security features.

10.1.1 Security and Reliability

Software security is related to software quality and software reliability, but the focus differs. Reliability deals with accidental failures. Failures are assumed to occur according to some given probability distribution. Improvements in reliability can be calculated based on this distribution. To make software more reliable, it is tested against typical usage patterns: 'It does not matter how many bugs there are, it matters how often they are triggered.' In security the attacker picks the distribution of the inputs. Hence, traditional testing methods are not geared towards finding security bugs. To make software more secure, it has to be tested against atypical usage patterns (but there are typical attack patterns).

10.1.2 Malware Taxonomy

Software that has a malicious purpose is called *malware*. There are different types of malware, and computer security has adopted anthropomorphic metaphors for their classification. A *computer virus* is a piece of self-replicating code attached to some other piece of code, with a *payload*. The payload can range from the non-existent via the harmless, e.g. displaying a message or playing a tune, to the harmful, e.g. deleting and modifying files. A computer virus *infects* a program by inserting itself into the program code. A *worm* is a replicating but not infecting program. Media reports on computer security incidents do not always make this distinction between worms and viruses. You might read about a virus attack when the code that is spreading would be better described as a worm.

A *Trojan horse* is a program with hidden side effects that are not specified in the program documentation and are not intended by the user executing the program. A *logic bomb* is a program that is only executed when a specific trigger condition is met.

10.1.3 Hackers

Originally, a hacker was an expert familiar with the intricate details of a computer system, able to use the system in a way beyond the grasp of ordinary users. Over time, the term hacker has acquired a negative connotation, describing a person who illicitly

breaks into computer systems. There is still a distinction between *white hats* who use their skills to help software developers, and *black hats* who break into systems intent on creating mischief and damage. Recent years have seen an increased amount of plain criminal activity on the Internet.

Many attacks exploit well-known security weaknesses (or design features) in an automated and efficient manner, needing neither ingenuity nor deep technical knowledge. Attackers who run attacks with tools acquired from someone else are known as *anklebiters* or *script kiddies*.

Experimenting with dangerous code at home may be an intellectual challenge but is fraught with danger. Anti-virus researchers have learned the importance of (physically) separating experimental from operational systems. Otherwise, there is the danger of code escaping out of the laboratory into the wild. Distributing code that performs actions on other people's machines is likely to bring you into conflict with the law. So, the familiar warning applies: don't try this at home.

10.1.4 Change in Environment

Change is one of the biggest enemies of security. You may have a system that offers perfectly adequate security. You change a part of the system. You may be aware of the security implications of this change and still get it wrong. Even worse, you may feel that the change has nothing to do with security, only to wake up to an unpleasant surprise.

10.1.5 Dangers of Abstraction

Abstraction is an important concept we cannot do without when designing and understanding complex systems. High-level descriptions of a system hide unnecessary details. Software developers use abstractions all the time when writing code in high-level programming languages. However, software security problems arise when intuitive properties of an abstraction do not match its concrete implementation. This chapter gathers together broken abstractions.

10.2 CHARACTERS AND NUMBERS

Broken abstractions can already be found with elementary concepts such as characters and integers. When describing their binary representation in memory or in messages we will often use hexadecimal values. Hexadecimal values in C have the prefix `0x`; hexadecimal values in a URL have the prefix %. We will use examples from both worlds.

10.2.1 Characters (UTF-8 Encoding)

A software developer writes an application that should restrict users to a specific subdirectory `A/B/C`. Users enter a filename as `input`. The application constructs the

pathname to the file as `A/B/C/input`. This attempt to constrain the users can be bypassed easily. An adversary could step up in the directory tree using `../` and access the password file by entering

`../../../../../etc/passwd`.

As a countermeasure, the developer performs *input validation* and filters out the offending character combination '`../`'. However, this is not the end of the story.

The UTF-8 encoding of the Unicode character set (RFC 2279) was defined for using Unicode on systems that were designed for ASCII. ASCII characters (`U0000–U007F`) are represented by ASCII bytes (`0x00–0x7F`). Non-ASCII characters are represented by sequences of non-ASCII bytes (`0x80–0xF7`). The following encoding rules are defined:

`U000000–U00007F`: `0xxxxxxx`

`U000080–U0007FF`: `110xxxxx 10xxxxxx`

`U000800–U00FFFF`: `1110xxxx 10xxxxxx 10xxxxxx`

`U010000–U10FFFF`: `11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`

The `xxx` bits are the least significant bits of the binary representation of the Unicode character. For example, the UTF-8 encoding of the copyright sign `U00A9 = 1010 1001` is `11000010 10101001 = 0xC2 0xA9`. Only the shortest possible UTF-8 sequence is valid for any Unicode character, but many UTF-8 decoders also accept longer variants. When multi-byte UTF-8 formats are accepted, a character has more than one representation. Here are three ways of writing '/'.

	format	binary	hex
1 byte	`0xxx xxxx`	`0010 1111`	`2F`
2 byte	`110x xxxx`	`1100 0000`	`C0`
	`10xx xxxx`	`1010 1111`	`AF`
3 byte	`1110 xxxx`	`1110 0000`	`E0`
	`10xx xxxx`	`1000 0000`	`80`
	`10xx xxxx`	`1010 1111`	`AF`

Once, a version of Microsoft's IIS was vulnerable as illegal Unicode representations of single-byte characters were accepted but not checked for during input validation. A URL starting with

{IPaddress}`/scripts/..%c0%af../winnt/system32/`

referenced the directory `C:\winnt\system32` because `%c0%af` is the two-byte UTF-8 encoding of /. Hence, `..%c0%af../` becomes `../../`. There is a further twist to this story. Consider the URL

{IPaddress}`/scripts/..%25%32%66../winnt/system32/`.

To see how this URL is processed, write the sequence `%25%32%66` in binary. You get `00100101 00110010 01100110`, i.e. the ASCII characters `%2f`. The URL is thus decoded to IPaddress`/scripts/..%2f../winnt/system32/`. No problem yet, but if the URL is decoded a second time, `%2f` is read as `/` and the URL references the directory `C:\winnt\system32`.

Lesson
Beware of mistranslations that change the meaning of texts. Decoding UTF-8 is a translation between different levels of abstraction.

10.2.2 The rlogin Bug

The `-f` option in the Unix `login` command

```
login [-p] [-h<host>] [[-f]<user>]
```

'forces' login and the user does not have to enter a password. The `rlogin` command

```
rlogin [-l<user>] <machine>
```

allows users to log in on remote machines. The *rlogin* daemon sends a login request to the machine named in the second argument for the user given in the first argument. Some versions of Linux and AIX did not check the syntax of the name field. The request

```
rlogin -l -froot machine
```

then results in

```
login -froot machine
```

i.e. forced login as root at the designated machine. This problem is caused by the composition of two commands. Each command on its own is not vulnerable. Syntax checking by the *rlogin* daemon can prevent this attack.

10.2.3 Integer Overflows

In mathematics integers form an infinite set. On a computer system, integers are represented as binary strings of fixed length (precision), so there are only a finite number of 'integers'. Programming languages have signed and unsigned integers, short and long (and long long) integers. If the result of a computation gets too large for the chosen representation, a *carry overflow* occurs. In this case, the familiar rules for integer arithmetic no longer apply. For example, with unsigned 8-bit integers arithmetic operations are performed modulo 256 and you get $255 + 1 = 0$, $16 \times 17 = 16$, and $0 - 1 = 255$.

Signed integers are represented as 2's complement numbers. The most significant (left-most) bit indicates the sign of the integer. If the sign bit is zero, the number is positive and

is given in normal binary representation. If the sign bit is one, the number is negative. To calculate the 2's complement representation of $-n$:

- Invert the binary representation of n by changing all ones to zeros and all zeros to ones. For 8-bit integers, this step computes $255 - n$.
- Add one to the intermediate result. For 8-bit integers, this step computes $255 - n + 1 = 256 - n$. The number 256 corresponds to the carry bit.

Computing with signed integers can again give unexpected results. For example, for signed 8-bit integers you get $127 + 1 = -128$ and $-128/-1 = -1$. Switching between signed and unsigned integers may turn a large positive value into a negative value: `0xFF` $= 2^8 - 1$ (unsigned) $= -1$ (signed).

Conversion between integer representations can cause security problems. Let the guard command

```
if (size < sizeof(buf))
```

compare a signed integer variable `size` with the result of `sizeof(buf)` that returns a result of data type `size_t`, i.e. an unsigned integer. If size is negative and if the compiler casts the result of `sizeof(buf)` to a signed integer, the buffer can overflow.

Integer truncation is another potential source of problems. Once, a Unix version had the following vulnerability. A program received a UID as a signed integer and checked UID$\neq$0 to prevent root access. The UID was later truncated to an unsigned short integer. Input `0x10000` became `0x0000` (root!).

In mathematics, the axioms defining the integers imply results such as $a + b \geq a$ if $b \geq 0$. With computer integers, such obvious 'facts' are no longer true. Such discrepancies between the abstract model and the actual implementation can lead to buffer overruns (Section 10.4.1). Consider the following code snippet that copies two character strings into a buffer and checks that the two strings (plus a terminating symbol) fit into the buffer:

```
char buf[128];
combine(char *s1, size_t len1, char *s2, size_t len2)
{
      if (len1 + len2 + 1 <= sizeof(buf)) {
      strncpy(buf, s1, len1);
      strncat(buf, s2, len2);
      }
}
```

On a 32-bit system, an attacker could set `len1 < sizeof(buf)` and `len2 = 0xFFFFFFFF`. Then `strncat` will be executed and the buffer will overrun as

$$\texttt{len1} + \texttt{0xFFFFFFFF} + 1 = \texttt{len1} + 2^{32} - 1 + 1 = \texttt{len1} \bmod 2^{32}.$$

Computation of array indices uses integer arithmetic. If you do not check that the result of an index calculation does not exceed the length of the array, a memory location above the array will be accessed (a typical beginner's mistake in introductory programming courses). Wrap-around to lower addresses occurs when arithmetic operations give negative results or when the effects of modular addition are not considered by the programmer. When computing array indices, you must check both upper and lower bounds.

Computer integers do not implement the mathematical abstraction 'integers', but integers modulo 2^w, where w is the number of bits chosen for their representation. This break in abstraction can cause programming mistakes of the kind just discussed.

> Many programmers appear to view integers as having arbitrary precision, rather than being fixed-sized quantities operated on with modulo arithmetic [14].

Lesson
Declare all integers as unsigned integers unless you really need negative numbers. If you are measuring the size of objects in memory, you do not need negative numbers. If your compiler flags a signed–unsigned mismatch, check whether you really need two different representations. If you do, pay attention to the checks you are implementing.

10.3 CANONICAL REPRESENTATIONS

Names (identities, identifiers) are a widely used abstraction. We give names to files and directories so that we can find them later. Network nodes have identifiers at the different levels of the protocol stack (DNS names, IP addresses, ...). Names are also used in security decisions. A firewall may let traffic pass to and from certain nodes only. Security policies define which files a user may access. When an entity has more than one name or when names have equivalent representations, an attacker may try to bypass security controls by giving an alternative name that had not been considered when setting the policy. Problems of this nature are encountered frequently. Filenames, URLs, and IP addresses can all be written in more than one way.

- Filename: `c:\x\data = c:\y\z\..\..\x\data = c:\y\z\%2e%2e\%2e\%2e\x\data`

- Dotless IP: $a.b.c.d = a \cdot 2^{24} + b \cdot 2^{16} + c \cdot 2^8 + d$
- Symbolic links: filename pointing to another file

The problem caused for security enforcement can be illustrated at the level of colloquial English. The file sharing service Napster had been ordered by court to block access to certain songs. Napster implemented a filter that blocked downloads based on the name of the song. Napster users bypassed this control by using variations of the names of songs. This is a particularly awkward problem because the users decide which names are equivalent.

Next, consider case-insensitive filenames. In this case, *myfile* and *MyFile* are equivalent names for the same file. Combine a filesystem with case-insensitive filenames with a security mechanism that uses case-sensitive filenames. (A vulnerability of this kind once involved the Apache web server and the HFS+ filesystem.) Assume that permissions are defined for one version of the name only, e.g. for *MyFile*. The attacker requests access to *myfile*, no restrictions are set, so the security mechanism grants the request and the filesystem gives access to the resource that should have been protected.

To address this problem, perform *canonicalization* before making access control decisions. Canonicalization resolves the various equivalent forms of a name to a single standard name. The single standard name is known as the *canonical name*. Canonicalization is relevant whenever an object has different but equivalent representations.

Lesson
Do not rely on the names received as user input; convert them – correctly – to your standard representation. 'Do not trust your inputs' is the battle cry in [121]. Use full pathnames. Do not let the system generate the full pathnames automatically. Preferably, do not make decisions based on names at the application level but use the operating system access controls.

10.4 MEMORY MANAGEMENT

Ancient cities and medieval castles had rings of protective walls to keep attackers out. To defeat these defences, attackers tunnelled under walls to make them collapse. Similarly, software developers may have designed impressive logical defences. If the attacker can tunnel through to memory, the logical defences are undermined from below. Flaws in memory management can open the cracks that let the attacker penetrate the system.

The programming languages C and C++ were designed to let developers perform their own memory management. There are situations where this feature is called for but it is

then up to the developers to get memory management right. Languages such as Java or C# take these tasks away from the developer and thus remove a possible source of errors.

Figure 10.1 shows a typical memory configuration. The runtime *stack* using the highest memory addresses contains the stack frames of the processes currently on the call stack. A stack frame contains information such as return address, local variables, and function arguments. The stack grows downwards in memory. System libraries are stored at the bottom of memory. Above is the *heap*. The heap is a memory area dynamically allocated by the application. The heap grows upwards.

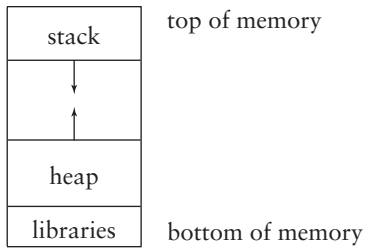


Figure 10.1: Memory Configuration

10.4.1 Buffer Overruns

When writing a program you will use *variables* to hold the values you want to manipulate. When the program is executed, memory sections (*buffers*) are allocated to those variables. A *buffer overrun* (overflow) occurs when the value assigned to a variable exceeds the size of the buffer allocated. Memory locations not allocated to this variable are overwritten. If the memory location overwritten had been allocated to some other variable, the value of that other variable is changed. Unintentional buffer overruns can make software crash. Intentional buffer overruns may enable an attacker to modify security-relevant data by assigning a deliberately malformed value to some other variable. Attractive targets are return address (specifies the next piece of code to be executed) and security settings. Buffer overruns have been the source of security vulnerabilities for some time. We add another historic example to the story of the Internet worm (Section 1.3.1).

VMS Login

One version of Digital's VMS operating system had a bug in the login procedure. Users could specify the device they wanted to log in to by giving their username as

username`/DEVICE =` <*machine*>

In one version of the operating system the length of the argument *machine* was not checked. A device name of more than 132 bytes overwrote the privilege mask of the process started by login. Users thus could set their own privileges by giving an appropriate 'machine name'.

10.4.2 Stack Overruns

Buffer overrun attacks on the call stack are known as *stack overruns*. Figure 10.2 (left) gives an outline of the frame that would be pushed on the stack when calling a function `void function(int a, int b, int c)` with local variables x and y. First come the input values in reverse order, followed by the return address and the saved frame pointer that points to the top of the previous stack frame. Finally, buffers for local variables defined within the function are allocated. (The precise layout of a stack frame is specific to operating system and compiler.)

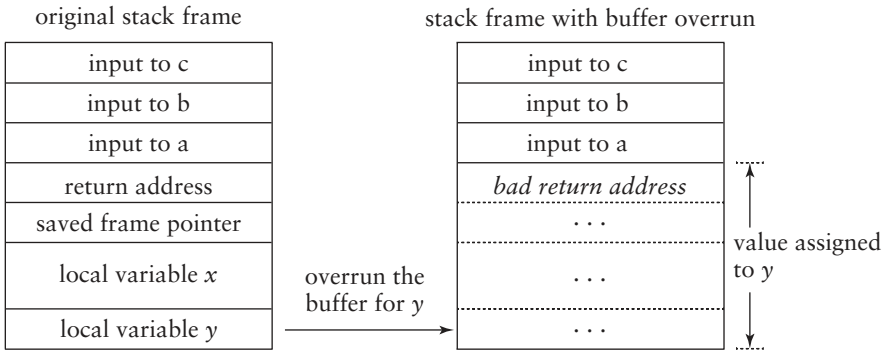


Figure 10.2: Stack Smashing Attack

Assume that the function does not check the size of the value assigned to y. An attacker could then pick a value that overruns the buffer allocated to y, overwriting memory locations above y. Specifically, an attacker can overwrite the return address with the start address of the attacker's code (*shellcode*) as shown in Figure 10.2. The attacker's code would then run with the privileges of the current process. This attack would be impossible if the stack grew upwards in memory [138].

The final task for the attacker is to find a way of getting the shellcode into the system. There are two major methods for doing so:

- *argv*[] method – the shellcode is passed as an argument to the vulnerable function and gets stored on the stack; this method requires an executable stack.
- *return-to-libc* method – the attacker calls a system library function; system functions that execute user-provided commands are likely candidates.

When the attacker cannot predict precisely where the input variable and thus the attack code will be placed, a *landing pad* of NOP (no operation) instructions at the start of the code can compensate for variations in the location the code will be found in. Section 10.7 will explain how to defend against stack overrun attacks.

10.4.3 Heap Overruns

Because of the prominence of stack overrun attacks, more effort has been devoted to protecting the stack than to protecting the heap. Buffer overrun attacks on the heap are known as *heap overruns*. It is more difficult to determine how to overwrite a specific buffer on the heap and to determine which other buffers will be overwritten in the process. If you are an attacker, you may not want to crash the system before you have taken over, but even attacks that do not succeed all the time are a threat to the defender.

To take control of execution, an attack has to overwrite a parameter that has influence on the control or data flow. On the heap, such parameters are pointers to (temporary) files and function pointers. Take a vulnerable program that defines a temporary file and writes to it. Create a buffer overrun that overwrites the pointer to the temporary file with a pointer to the target file, e.g. the password file. The attacker can now use the program to write to the target file. A function pointer `int (*funcptr)(char *str)` allows a programmer to dynamically modify the function to be called. On execution the function pointed to by the pointer will be called. When a buffer overrun overwrites the function pointer, a function defined by the attacker will be executed. Exploiting function pointers requires an executable heap.

Lesson
Redirecting pointers is a great way of attacking systems.

10.4.4 Double-Free Vulnerabilities

Stack overrun attacks modify the return address. The attacker can fairly easily guess the location of this pointer relative to a vulnerable buffer. Equally, the defender knows which target to protect. Overwriting arbitrary pointers with arbitrary values is a more powerful attack. There are more targets; hence the attack is more difficult to defend against.

The attacker does not even have to *write* to the pointer. Instead, the operating system can be lured into *reading* malformed input and then doing the job for the attacker. The following attack targets memory allocation in Unix. Our description follows Doug Lea's malloc but other allocators follow similar principles. Memory is allocated calling

```
void * malloc (size_t size)
```

which returns a pointer to a newly allocated chunk of `size` bytes. The contents of the chunk are not initialized. The call returns a null pointer if a chunk cannot be allocated. Memory is deallocated by calling

```
void free (void *ptr)
```

where `*ptr` must have been returned by a previous call to *malloc*(), *calloc*(), or *realloc*(). If `*ptr` is null, no operation is performed. Otherwise, the behaviour of *free* is undefined when applied twice to the same pointer.

Memory is divided into chunks. Free chunks are placed in double-linked lists called *bins*. Chunks contain user data and control data, so-called *boundary tags* that include the size of the chunk and the size of the previous chunk in memory (Figure 10.3). The boundary tag of a free chunk also contains forward and backward pointers to its neighbours in the bin. A lower order bit of `size` is used as a *control flag* that indicates whether the previous chunk is free. When a chunk is freed it is coalesced into a single chunk with neighbouring free chunks. There are no adjacent free chunks in memory. Free chunks that have been coalesced are taken out of their bin using the `unlink()` command.

```
#define unlink(P, BK, FD) {
[1] FD = P->fd;
[2] BK = P->bk;
[3] FD->bk = BK;
[4] BK->fd = FD;
}
```

`unlink()` takes a pointer `P` out of a list. Lines 1 and 2 save the backward and forward pointers of `P`. Line 3 updates the backward pointer of the next chunk in the list; the address located at `FD` plus 12 bytes (offset of *backward* field in the boundary tag) is overwritten with the value stored in `BK`. The forward pointer is updated in similar fashion.

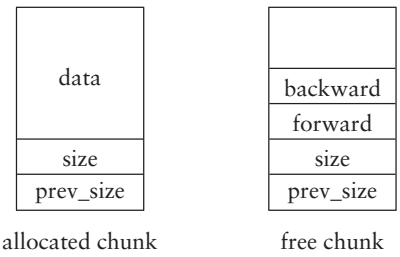


Figure 10.3: Allocated and Free Chunks

The programming error to look out for in a *double-free* attack is a function that may free a chunk twice but does not set the respective pointer to null the first time. The attacker calls the vulnerable function and proceeds as follows:

1. Let the function allocate a memory block *A* that may be freed twice.
2. The function frees *A* a first time (but does not set the pointer to null); the attacker had prepared the memory so that when *A* is freed consolidation with neighbouring free chunks below *A* creates a large block.

3. Ask the function to allocate a large block in the hope of getting the large chunk just freed.
4. Let the function write user-provided input into this block next to where *A* had been; the user input is a fake unallocated chunk; the *forward* field contains the target address that should be overwritten (minus the offset 12), the *backward* field the value to be written.
5. Continue the vulnerable function until it frees *A* a second time.

When *A* is freed the second time, the fake chunk next to *A* is treated as a free chunk that should be coalesced with *A* and taken out of its (non-existing) bin. Therefore `unlink()` is applied to the fake chunk and the target address is overwritten in step 3.

Tables at known locations that influence the execution flow (such as the Global Offset Table in Unix) are an attractive target. The double-free vulnerability came to prominence with an exploit for the *zlib* compression library in 2002. Double-free is an example of an *uninitialized memory corruption* vulnerability where a command reads from a memory location that has not been initialized and happens to contain data left there by the attacker. The attacker has to predict correctly which memory location the command will read from, but can increase the chances of success by *heap spraying*, i.e. leaving data in a large number of locations.

10.4.5 Type Confusion

Programs written in a type-safe (memory-safe) language cannot access memory in inappropriate ways. A well-known example is the Java programming language. Each Java object is an instantiation of a class. The Java Virtual Machine (JVM) keeps track of objects in memory by using pointers that are tagged with the class of the object. Operators must only accept operands of the correct type. Static and dynamic type checking should prevent any access that violates the abstractions provided by objects and classes. Automatic garbage collection takes care of further memory management tasks.

A *type-confusion* attack manipulates the pointer structure so that two pointers, one with a wrong class tag, point to the same object. Such an object can then be manipulated by accessing the 'wrong' type. Consider, for example, a trusted object *A* of type *Tr*. When an attacker manages to have the pointer for another untrusted object *X* of type *Un* point to the same buffer in memory, the trusted object *A* can be modified as if it were of type *Un* (Figure 10.4). Often (but not always) a type-confusion attack can be extended to compromise the entire system and modify objects at will. Type-confusion attacks are rare, but they do occur. Sun Security Bulletin no. 00218 (18 March 2002) refers to a problem in a JVM implementation. An attack on a mobile phone version of Java[1] was reported in 2004.

[1] Adam Gowdiak, Java 2 Micro Edition Security Vulnerabilities, Hack in the Box Security Conference 2004, Kuala Lumpur, Malaysia.

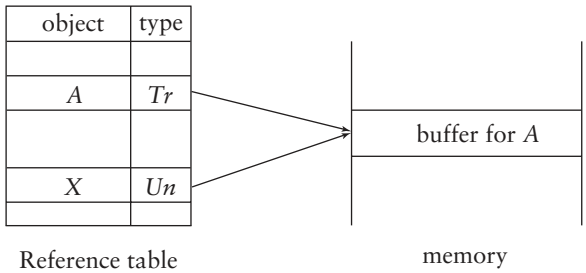


Figure 10.4: Type-Confusion Attack

An old version of Netscape Navigator had a bug that allowed a simple type-confusion attack [163]. Java allows a program that uses type T also to use type *array of* T. Array types are defined for internal use only. Their name begins with the character `[`. Programmer-defined classnames are not allowed to start with this character. Hence, there should be no danger of conflict. However, a Java byte code file could declare its own name to be a special array type name, thus redefining one of Java's array types.

An ingenious type-confusion attack using random memory errors is described in [106]. The strategy is to create a memory layout where a random bit error is likely to change a reference so that it points to an object of the wrong type. Create two classes A and B

```
class A {
  A a1;
  A a2;
  B b;
  A a4;
  A a5;
  int i;
  A a7;
};
```

```
class B {
  A a1;
  A a2;
  A a3;
  A a4;
  A a5;
  A a6;
  A a7;
};
```

and a program

```
A p;
B q;
void write(int target, int value)
{
  p.i = target;
  q.a6.i = value;
}
```

This program type checks correctly. The attack proceeds as follows.

1. Fill the memory with lots of objects of type *B* and a single object of type *A*, (say) at location *x*; let all the *A* fields in the *B* objects point to this single *A* object.
2. Keep scanning the *B* fields until a memory error is detected; shining a light bulb on the memory chip is a proven way of inducing memory errors.
3. Call the program above with pointers `p` and `q` pointing to the same memory location, with `target` giving the address to be overwritten, minus the offset $6 \cdot 4$, and `value` giving the value to be written.

Suppose a bit flips in a *B* object in a position referencing the sole *A* object in memory. The position affected now contains the value $x \oplus 2^i$ for some value *i*. The modified pointer is likely to point to an *A* field somewhere in memory. Dereferencing its *b* field is likely to hit one of the many *A* fields containing the address *x* of the sole *A* object. You now have type confusion. There is a pointer of type *A* containing *x* and a pointer of type *B* containing *x*. Calling the program with the parameters stated will write a value chosen by the attacker to an address chosen by the attacker.

Lesson
Attacks at the hardware layer can undermine security controls in the layer above.

10.5 DATA AND CODE

Data and code are important abstractions when designing a system that executes programs. When input presented as data gets executed, the abstraction is broken and attacks may be possible. The following examples have in common that data received as input may not mean what the programmer expected.

10.5.1 Scripting

Scripting languages construct commands (scripts) from predefined code fragments and from user input. The script is then passed to another software component (browser, database server, operating system) where it is executed. An attacker may try to hide additional commands in user input. The defender has to check and *sanitize* user input. Both have to be aware of certain technical details of the component executing the script:

- symbols that terminate command parameters;
- symbols that terminate commands;

- dangerous commands, e.g. commands for executing the commands they receive as input (*eval*, *exec*, *system*, ...).

Examples of scripting languages are Perl, PHP, Python, Tcl, Safe-Tcl, and JavaScript. As a first example, a script for a Unix server that sends a file to a client may contain the instruction

```
cat thefile | mail clientaddress
```

where `thefile` is the name of the file and `clientaddress` is the mail address of the client. When a malicious user enters `user@address | rm -rf /` as the mail address, the server will execute

```
cat thefile| mailuser@address | rm –rf /
```

mailing the file to the user and deleting all files the script has permission to delete.

10.5.2 SQL Injection

SQL is the standard database query language. Strings in SQL commands are placed between single quotes. When a script constructs SQL queries as strings put together from query fragments and user input, the attacker may be able to change the logic of a query by entering a single quote in the input, followed by SQL instruction fragments. Consider the following query to a client database where `$name` is input provided by the user:

```
$sql = "SELECT * FROM client WHERE name = '$name'"
```

It is intended to create queries of the form `select * from client where name = 'Bob'`. However, when the attacker enters `Bob' OR 1 = 1--`, the query becomes

```
SELECT * FROM client WHERE name = 'Bob' OR 1=1--'
```

The argument of the WHERE clause is read as the logical disjunction of "`name = 'Bob'`" and "`1=1`", and `--` is read as the start of a comment. The second clause evaluates to TRUE so the disjunction evaluates to TRUE and the entire client database is selected.

Assume further that the database uses semicolons to terminate commands. The attacker could insert additional commands with inputs like `Bob'; drop table client --`. There are two classes of countermeasures against code injection attacks:

- Input validation – make sure that no unsafe input is used in the construction of a command; this is a general topic covered in Section 10.7.4.
- Change the *modus operandi* – modify the way commands are constructed and executed so that unsafe input can do no harm.

Parametrized queries with *bound parameters* (DBI placeholders in Perl) follow the second approach. Scripts are compiled with placeholders instead of user input. Commands are called by transmitting the name of the procedure and the parameter values. During execution, the placeholders are replaced by the actual input. Attempts to smuggle in commands through user input will not work. User input will be treated as data by the component executing the script. This defence reaches its limit when the parametrized procedure contains *eval*() statements that accept user inputs as arguments. User input to the script will be passed to such statements.

10.6 RACE CONDITIONS

Race conditions can occur when multiple computations access shared data in such a way that their results depend on the sequence of accesses. This could happen when multiple processes or multiple threads in a multi-threaded process access the same variable. An attacker can try to exploit a race condition to change a value after it has been checked by the victim but before it is used. This issue is known in the security literature as TOCTTOU (see Section 4.1). We give a historic example from the 1960s relating to CTSS, one of the early time-sharing operating systems. This is the start of our story [72]:

> Once, a user found that the password file was given as the 'message of the day'.

What happened? In CTSS, every user had a unique home directory. When a user invoked the editor, a scratch file was created in this directory. This scratch file had a fixed name, (say) SCRATCH, independent of the name of the file that was edited. This was a reasonable design decision. A user could only run one application at a time. No one else could work in another user's directory. Therefore, there was no need to provide more than one scratch file for the editor. So far so good. Furthermore, *system* was a user with its own directory. At some stage, several users were working as system managers and it seemed convenient that they should be allowed to work (access the system directory) concurrently. This feature was implemented. Now (Figure 10.5),

1. one system manager starts editing the message of the day: SCRATCH:=MESS,
2. then a second system manager starts editing the password file: SCRATCH:=PWD,
3. and the first manager stores the edited file, so MESS:= SCRATCH = PWD.

Lesson
Atomic transactions are an important abstraction when an operation has to execute as a single unit. The operation should either execute as a whole, or have no effect at all.

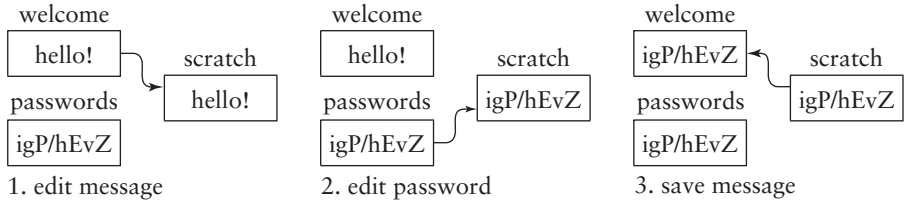


Figure 10.5: Sharing the Scratch File in CTSS

This abstraction is implemented using locking mechanisms that protect access to the resources used by the transaction. An example of a locking mechanism is the *synchronized* keyword in Java. Protecting atomic transactions is important in a multi-threaded language like Java, but Java leaves this task to the programmer. The disadvantage of adding synchronization is a loss of performance.

10.7 DEFENCES

We could treat the problems encountered individually and look for specific solutions, often limited to a given programming language or runtime system. This would amount to *penetrate-and-patch* at a meta-level. Alternatively, we can look for general patterns. In the case of insecure software, the pattern repeated is that of familiar programming abstractions such as *variable, array, integer, data & code, address (resource locator)*, or *atomic transaction* being implemented in a way that can break the abstraction.

Software security can be addressed in the processor architecture, in the way code is executed, in the programming language we are using, in the coding discipline we adhere to, through checks added at compile time, during software development, and during deployment.

10.7.1 Prevention: Hardware

Buffer overrun attacks overwrite control information with user input. They can be prevented by hardware features that protect control information. Intel's Itanium processor has a separate register for the return address. A processor architecture with a separate *secure return address stack* (SRAS) is described in [151]. When adding protection mechanisms at the hardware layer, there is no need to rewrite or recompile programs. Only certain processor instructions have to be modified. However, there are also drawbacks beyond the need for new hardware. Existing software, e.g. code that uses multi-threading, may work no longer.

10.7.2 Prevention: Modus Operandi

A *non-executable stack* stops shellcode from being run from the stack. Code does not have to be recompiled, but existing software that requires an executable stack will work no longer.

It is to the attacker's advantage when memory usage is predictable and elements needed for an attack are always in the same location. *Address space layout randomization* can then decrease the attacker's chance of success. This technique has been used in BSD for the stack layout to defend against argv[] attacks and in Windows for systems libraries as a defence against return-to-libc attacks.

10.7.3 Prevention: Safer Functions

When developing software in C or C++, you have to avoid writing code susceptible to buffer overruns. C is infamous for its unsafe string handling functions such as `strcpy`, `sprintf`, or `gets`. Take the specification of `strcpy` as an example:

```
char * strcpy( char * strDest, const char * strSource );
```

An exception is raised if source or destination buffer are null. The result is undefined if strings are not null-terminated, and there is no check whether the destination buffer is large enough. You are advised to replace the unsafe string functions with safer functions where the number of bytes or characters to be handled can be specified, such as `strncpy`, `_snprintf`, or `fgets`. The specification of the function `strncpy` is

```
char * strncpy( char * strDest, const char * strSource, size_t count );
```

Using safer string handling functions does not eradicate buffer overruns by itself. You still have to get your byte counts and your arithmetic right. You have to know the correct maximal size of your data structures. This is straightforward when data structures are used only within a function but may be difficult if data structures are shared between programs. If you underestimate the required length of the buffer your code may become unreliable and crash.

Besides safer string handling functions, there are also safer integer libraries that flag carry overflows during arithmetic operations. Some compilers can be configured to check for unsafe functions.

10.7.4 Prevention: Filtering

You can distinguish between 'good' and 'bad' inputs in two ways:

- White lists only allow good values. This is a conservative approach, but if you forget about some specific legal input a legitimate action might be blocked.

- Black lists, blocking all dangerous values such as <, >, &, = , %, :, ', '' that might be used to insert code. If you forget about a dangerous input, attacks may get through.

Watertight black lists are difficult to get:

1. You must know all dangerous inputs, e.g. all *escape characters* – escape characters mark the transition from one context into another; the single quote in SQL starts/ends the parsing of a string.
2. You must know all the encodings of characters a browser will accept, e.g. illegal but syntactically correct UTF-8 encodings of characters or the more obscure UTF-7 format that was used in some XSS attacks.
3. You must know all the characters a component might convert to a dangerous character; helpful components might convert language-specific characters to similar-looking ASCII characters; e.g. Unicode characters 2039 (single left quotation mark in French) and 304F (hiragana character 'ku') could be mapped to < (start of an HTML tag).

White lists work well when valid inputs can be characterized by clear rules, preferably expressed as *regular expressions*. This is not always feasible. In a bulletin board application that only permits alphanumeric characters it is not possible to insert script tags. However, neither is it possible for users to include image tags and post holiday snaps with their entries. Filtering rules could also refer to the *type* of an input. For example, an `is_numeric()` check could be applied when an integer is expected as input.

Dangerous characters can be sanitized by using safe encodings. For example, in HTML <, > and & should be encoded as `<`, `>`, and `&`. (There are more characters that need HTML encoding.) *Escaping* places a special symbol, often a backslash, in front of the dangerous character. For example, escaping single quotes will turn d'Hondt into d\'Hondt. In PHP, you can call `addslashes()` for escaping user inputs.

Escaping has its pitfalls. Take this example[2] that builds on `addslashes()` and the GBK character set for simplified Chinese. GBK has one-byte and two-byte characters. `0xbf27` is not a valid two-byte character; interpreted as single-byte characters, we get `0xbf` followed by `0x27` ('), displayed in GBK as ¡'. Adding a backslash (`0x5c`) in front of the single quote gives `0xbf5c27`. This happens to be the two-byte GBK character 縗 (`0xbf5c`) followed by a single quote. The backslash has become part of another character, the single quote has survived.

To apply filtering you have to know all sources of malicious input. Input to a software component can come via input parameters, *environment variables* inherited from the caller, or out of files read by the component. In web applications (Chapter 18) inputs can

[2]From `http://shiflett.org/blog/2006/jan/addslashes-versus-mysql-real-escape-string`

come from GET parameters, POST parameters, URLs, referer, cookies, from a database, from the filesystem, etc. Filters can be implemented in a wrapper (Section 7.6.8) to protect legacy code you do not want to touch.

Lesson
Filtering remains a difficult problem, despite all the support for validating inputs available in scripting languages. While it may be fairly straightforward to plug the obvious hole, perfect defences are often difficult to achieve.

10.7.5 Prevention: Type Safety

You can avoid software bugs by using a programming language that stops you from making mistakes. *Type safety* guarantees the absence of untrapped errors [55]. Safety guarantees rely on static checks and on dynamic runtime checks. *Static type checking* at compile time examines whether the arguments an operand may get during execution are always of the correct type. Static type checking is more complicated than dynamic type checking during runtime, but it results in faster execution because the hard work is done in advance.

In practice, type safety often means just memory integrity. The ultimate goal in software security, though, is execution integrity [89]. With type safety we are getting close to 'provably secure' software, but there are some caveats. Type safety is difficult to prove completely and problems may hide in the actual implementation (see Section 10.4.5). An even greater challenge is the precise definition of what execution integrity means for a given software component (e.g. an operating system). Hence, the type safety properties you usually get are useful to have for security, but do not imply 'security'.

10.7.6 Detection: Canaries

A change to the return address might be detected by placing a (random) check value as a *canary*[3] in the memory position just below the return address [73]. Before returning, the system checks that the canary still has the correct value (Figure 10.6). To make use of this technique, code has to be recompiled so that the insertion and subsequent check of the canary are added to the object code. This technique can also be used on the heap for protecting the boundary tag in chunks [237, 236].

10.7.7 Detection: Code Inspection

Manual code inspection is tedious and error prone, so automation is desirable. Code inspection tools scan the code looking for potential security problems. A tool that uses

[3]Canaries were used in coal mines to detect gas leaks. The death of a canary was a warning for the miners to evacuate the pit.

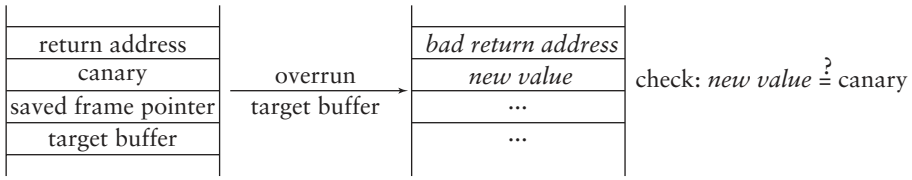


Figure 10.6: Canary Indicating an Attempted Buffer Overrun Attack on the Stack

meta-compilation for C source code is described in [14, 112]. It works as an expert system that incorporates rules for known security issues. The rules look for patterns of the form

$$\text{untrustworthy source} \rightarrow \text{sanitizing check} \rightarrow \text{trust sink}$$

and raises an alarm if untrustworthy input gets to a sink without proper checks. This pattern can also be used to learn about new rules. If, say, code analysis indicates a check but no sink, then probably the rules are not aware of a particular kind of sink. If this is the case and a new sink is identified, corresponding rules can be added to the rule base. Code inspection is good at catching known types of problem but does not guarantee that there are no vulnerabilities, or that everything flagged as a problem is indeed a flaw. For practical deployment of code analysis tools, it is essential to keep the false alarm rate low.

Taint analysis marks inputs from untrusted *sources* as tainted and stops execution if a security-critical function (*sink*) receives tainted input. *Propagating functions* propagate taint. *Sanitizing functions* produce clean output from tainted input. Table 10.1 gives the information relevant when analyzing PHP scripts for SQL injection. *Static taint* analysis is applied to source code at compile time so problems can be eliminated before deploying the code. *Dynamic taint analysis* is performed at runtime and can capture data-dependent taint propagation but can significantly reduce performance. The checks necessary are normally included by the compiler.

taint sources	uninitialized data if `register_globals` is enabled, superglobals arrays $_GET, $_POST, $_COOKIES, $_SERVER, data from internal sources such as database and files
propagating functions	string functions `substr()`, `str_replace()`, `preg_replace()`, ... `mysql_fetch_array()`, `mysql_fetch_assoc()`, `mysql_fetch_row()`, `file()`, `fread()`, `fscanf()`, ...
sanitizing functions	`mysql_escape_string()`, `mysql_real_escape_string()`, `int type cast`
sinks	`mysql_query()`, `mysqli_query()`

Table 10.1: PHP Sources, Propagators, Sanitizers, and Sinks for SQL Injection

10.7.8 Detection: Testing

Testing is an integral part of software development normally used to demonstrate the correctness of functionality. In security testing you have to show the correctness of security functionality. This implies that you must have some idea about the potential threats. The results of a threat analysis should provide input to security testing.

It is often said that testing cannot prove the absence of errors. This is correct, but nor can security proofs. Proofs only guarantee the absence of the errors you have been looking for in the abstract model used in the proof. The attacks presented in this chapter break the abstractions relied on by software developers. In the same way, attackers break the abstractions security proofs rely on. In software security, proofs amount to *symbolic testing*.

In *white-box testing* the tester has access to source code and can use this information for test planning. *Black-box testing* is performed without looking at the source code. A famous instance of black-box testing is the security analysis of SNMP implementations carried out by the Oulu University Secure Programming Group (see CERT advisory CA-2002-03).

Lesson
You do not need source code for observing how memory is used or for testing whether inputs are properly checked. Access to a high-level specification of a software component and of its interfaces may actually be more helpful than access to the source code.

An important general technique for security testing is *data mutation* (*fuzzing*) [121]. Data mutation sends malformed inputs to the program interfaces. Random inputs are not particularly useful for security testing. They test how unexpected inputs are handled, but usually not much code will be exercised as such inputs will be caught by simple input checks or crash the code. (When code crashes, check whether the return address has been overwritten by input data. This indicates that the code is vulnerable to buffer overruns.) Partially incorrect inputs try to get past the first line of defence and exercise more of the code.

For data containers data mutation might explore the following cases. The container the program is creating already exists; software may crash if it tries to create a new temporary file that already exists. The container the program tries to access does not exist. Consider, for example, a NULL DACL in Windows; if there is no DACL, the access permissions are not checked and access is granted. The program does not have access to the container, or only restricted access. Set permissions so that the tested component

has no access to the container, or only insufficient access rights. Test what happens if the name of the container changes and test the code with filenames that are links to other files.

For data, the test cases should include null (no value; what happens if an input field is empty?), zero, wrong sign (which brought down the Ariane 5 rocket[4]), wrong type, out-of-bounds input, and combinations of valid and invalid data (valid data to get past input checks so that invalid data may actually be processed), character sets and encodings. Test cases should include inputs that are too long and inputs that are too short. When the application expects a fixed size input but gets less data, security problems may arise; see, for example, the Sun tarball vulnerability described in [107]. For network code, test cases should include replay of messages from previous protocol runs, out-of-sync messages, fragmented messages, partially correct protocol runs (to check for something like TCP SYN flooding attacks) and high traffic volumes.

10.7.9 Mitigation: Least Privilege

The principle of least privilege applies in two ways. When writing code, be sparing with requiring privileges to run the code. If code running with few privileges is compromised, the damage is limited. Least privilege comes into play again when deploying software systems. Do not give your users more access rights than necessary. Do not activate options you do not need.

The Unix `sendmail` program may illustrate the latter problem. System managers configuring mail systems need to establish that messages arrive at their destination. It would therefore help if the mail configuration on a network node could be checked and modified remotely, without requiring the system manager to log in at that node. For this purpose, `sendmail` includes a *debug* option. When this option is switched on at the destination, a set of commands can be sent as the username in a mail message, which will be executed by `sendmail`. This feature was one point of attack for the Internet worm (Section 1.3.1).

Lesson
In the past, software was shipped in open configurations, with generous access permissions and all features activated. It was up to the user to harden the system by removing features and restricting access rights. Today, software is more likely to be shipped in a locked-down configuration. Users must switch on the features they want to use.

[4] `http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html`

10.7.10 Reaction: Keeping Up to Date

When a software vulnerability is discovered, the affected code has to be fixed, the revised version has to be tested, a patch has to be made available and has to be installed by the users for the problem to be fixed. In other words, it is not just the software vendors who have to react to problems found in their products. The user community must also take action, and be aware of the problem in the first place. The investigations in [11] show that most exploits of a given vulnerability occur well after countermeasures have been made available. Figure 10.7 is taken from this report.

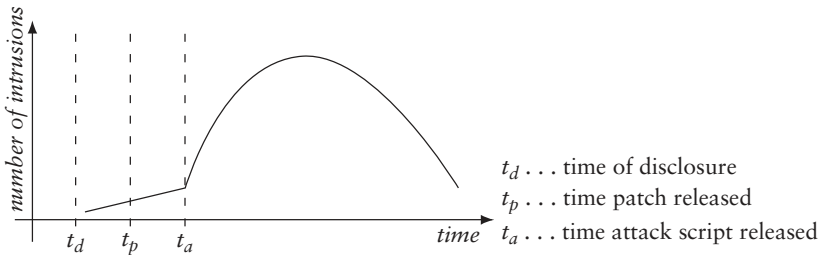


Figure 10.7: Life Cycle of Intrusions

Patches might become a resource for potential attackers as they may give information about the vulnerability that has been removed, which may still persist in other places. By comparing previous and patched versions, attackers may learn about new vulnerability classes. Patches may thus be obfuscated to hide the problem that has been fixed. There is a lively discussion about the best strategies for vulnerability disclosure and patch distribution.

10.8 FURTHER READING

There is a good choice of books on writing secure software, e.g. [121, 227, 107, 119, 122]. Microsoft's Security Development Lifecycle is described in [123]. Technical details of various buffer overflow attacks are covered in [95]. *Phrack* magazine covers technical details of software vulnerabilities. For those interested in the history of the field, the first paper on computer worms was published in 1982 [209]. A starting point for papers on race conditions in modern operating systems is [37]. You will find an excellent introduction into the realities of providing and using software security tools in [33].

If you are interested in the stories behind real attacks, try [217, 208, 84]. These are factual accounts of attacks written for a general audience. You can learn about new attacks through CERT advisories, mailing lists such as BugTraq (maintained at `www.securityfocus.com`), and security bulletins from software vendors.

10.9 EXERCISES

Exercise 10.1 Will this while loop terminate? If yes, after how many iterations? Does the result change if the variables are unsigned integers instead of signed integers?

```
int i = 1;
int c = 1;
while (i > 0)
{
i = i * 2;
c++;
}
```

Exercise 10.2 Business software stores dates in the binary-coded decimal (BCD) format. Decimal digits are encoded in four bits by their binary representation. A byte stores two decimal digits. For which days of the month, for which months of the year, and for which years is the binary value stored in a byte different from its BCD value?

Exercise 10.3 Give a fix for the flaw in the code for concatenating two strings discussed in Section 10.2.3.

Exercise 10.4 An alternative design for a canary mechanism places the NULL value just below the return address. What is the rationale for this design decision? When does this method meet its objectives, and what are its limitations?

Exercise 10.5 What is the flaw in the code snippet below that fills a buffer with zeros? How can the problem be fixed?

```
char* buf;
buf = malloc(BUFSIZ);
memset ( buf, 0, BUFSIZ );
```

Exercise 10.6 You are given chunks whose length is a multiple of 16 bytes. For each size of chunk, there is a separate bin. The following instruction takes the size of a chunk in bytes and calculates the index of the bin the chunk belongs to:

```
#define bin_index(size)
        ( ( ( (unsigned int)(size) ) >> 3) - 2)
```

Why is this instruction vulnerable to a buffer overrun attack?

Exercise 10.7 Unix systems use *environment variables* to configure the behaviour of utility programs. A program executing another program can set

the environment variables for that program. How could this fact be used in an attack? What defences should programmers apply?

Exercise 10.8 Consider a memory system that stores free blocks in a double-linked list `IDX` in ascending order of size. A free block `P` of size `S` is inserted using the `frontlink` command given below. Blocks are taken from the list using `unlink`. How does this system react when a block that is already in the free list is added again, and removed later?

```
#define frontlink(A, P, S, IDX, BK, FD)
...
[1] FD = start_of_bin(IDX);
[2] while ( FD != BK && S < chunksize(FD) )
          {
[3]         FD = FD->fd;
          }
[4] BK = FD->bk;
[5] P->bk = BK;
[6] P->fd = FD;
[7] FD->bk = BK->fd = P;
}
```

Exercise 10.9 Analyze how `frontlink` could be exploited in a memory corruption attack.

Exercise 10.10 Design a scalable protection mechanism for chunks that uses canaries without having to store reference values for all canaries.

Exercise 10.11 You are given an archiving function that writes to a zip archive storing files with their directory path. Extraction from the archive restores files at the location given. How could an attacker working in user mode who has no access to the root directory or to `/etc`, and who is not permitted to archive `/etc/passwd`, overwrite this file?

Exercise 10.12 Define test cases for `frontlink` and `unlink`.

Exercise 10.13 Determine the sanitization functions and trust sinks in a taint analysis of PHP scripts that addresses cross-site scripting attacks (Section 18.4).

Exercise 10.14 It has been claimed that full disclosure of software vulnerabilities will improve software security. When a vulnerability is found, what steps have to be taken so that the problem is eventually fixed at the end user's site? Give a step-by-step analysis discussing important issues that should be taken into account. What are the advantages and/or disadvantages of full disclosure?

Chapter 11

Bell–LaPadula Model

What is your security policy? What rules decide who gets access to your data? To formulate a security policy, you have to describe the entities governed by the policy and state the rules that constitute the policy. This could be done informally in a natural language document. In practice, such documents too often suffer from ambiguities, inconsistencies, and omissions. To avoid these problems, you might prefer a formal statement of your security policy. A security model does just that.

Security models play an important role in the design and evaluation of high-assurance security systems. Their importance was noted in the Anderson report [9]. The design process starts from a formal specification of the policy the system should enforce, i.e. the security model, and a high-level specification of the system itself. By adding more details to this high-level specification you can arrive at a series of lower-level specifications. Then you have to show that the high-level specification implements the desired policy. For high assurance, a formal proof may be required. You also have to show that the lower-level specifications are consistent with your policy, but at these levels it becomes increasingly difficult to conduct formal proofs.

This chapter is intended as a case study to demonstrate this approach. The Bell–LaPadula model was developed to capture multi-level security policies for classified data. We will describe this model and use the Multics operating system to show how the model can be used when analyzing a system designed to enforce these policies. Chapter 12 will give a survey of other important security models.

OBJECTIVES

- **Demonstrate how security policies can be formalized.**
- **Introduce the Bell–LaPadula model, and discuss its scope and limitations.**
- **Show how a formal model can be used when analyzing a security system.**
- **Present some important milestones in the history of computer security.**

11.1 STATE MACHINE MODELS

State machines (automata) are a popular tool for modelling many aspects of computing systems. We assume that the readers of this book are already familiar with this topic. State machines are also the basis for some important security models. The essential features of a state machine model are the concepts of a *state* and of state changes occurring at discrete points in time. A state is a representation of the system under investigation at one moment in time, which should capture exactly those aspects of the system relevant to our problem. The *state transition* function defines the next state depending on the present state and input. An output may also be produced.

A simple example of a state machine is a light switch. It has two states, *on* and *off*, and one input, *press*, that moves the system from state *on* to state *off*, and from state *off* to state *on*. A ticket vending machine is a more sophisticated example. Its state has to record the ticket requested and the money still to be paid. The inputs are ticket requests and coins. The outputs are the tickets and any change returned. An example from computer science is a microprocessor. The state of the machine is given by its register contents and the inputs are the machine instructions.

If you want to talk about a specific property of a system, such as security, using a state machine model, you must first identify all the states that fulfil this property. You then have to check whether all state transitions *preserve* this property. If this is the case and if the system starts in an *initial state* having this property, then you can prove by induction that the property will always hold.

11.2 THE BELL–LAPADULA MODEL

The *Bell–LaPadula model* (BLP) is probably the most famous of the security models. It was developed by Bell and LaPadula at the time of the first concerted efforts to design secure multi-user operating systems. If those systems were to process classified information at different security levels, they had to enforce the multi-level security (MLS) policy described in Section 5.8.4. Users may only get information they are entitled to according to their clearance.

The BLP model is a state machine model capturing the confidentiality aspects of access control [24]. Access permissions are defined both through an access control matrix and through security levels. Security policies prevent information flowing downwards from a high security level to a low security level. The BLP model only considers the information flow that occurs when a subject observes or alters an object.

11.2.1 The State Set

Our description of the BLP model uses the notation introduced in Chapter 5. We have:

- a set of *subjects* S;
- a set of *objects* O;
- the set of *access operations* $A = \{\mathsf{execute}, \mathsf{read}, \mathsf{append}, \mathsf{write}\}$ that directly mirror the access rights of Section 5.3.2;
- a set L of *security levels* with a partial ordering $\leq$.

We want to use the state of the system to check its security, so the state set of our model has to capture all current instances of subjects accessing objects and all current permissions.

We can use a table to record which subject has access to which object at a given point in time. The rows in the table are indexed by subjects, the columns by objects, and an entry in the table gives the access operations the subject currently performs on the object. In mathematical notation, such a table corresponds to a collection of tuples (s, o, a), indicating that subject s currently performs operation a on object o. Tuples are elements of the set $S \times O \times A$ (the Cartesian product of the sets S, O, A), so a table corresponds to an element of the power set $\mathcal{P}(S \times O \times A)$. It is customary to use the symbol b to denote the table of current access operations in the BLP model, and B to denote the set of all such tables.

The current access permission matrix is written as $M = (M_{so})_{s \in S, o \in O}$. We use $\mathcal{M}$ to denote the set of all access permission matrices. The BLP model uses three functions for assigning security levels to subjects or objects:

- $f_S : S \to L$ gives the maximal security level each subject can have;
- $f_C : S \to L$ gives the current security level of each subject;
- $f_O : O \to L$ gives the *classification* of all objects.

The current level of a subject cannot be higher than its maximal level, hence $f_C \leq f_S$ or, in words, f_S dominates f_C. You will see the reason for introducing f_C in a moment. The maximal security level is sometimes called the subject's *clearance*. Other sources use clearance only to denote the security levels of users.

For convenience, we write f for the triple (f_S, f_C, f_O) and use $F \subset L^S \times L^S \times L^O$ to denote the set of all possible security level assignments. All of this leaves us with a rather complicated state set $B \times \mathcal{M} \times F$. An individual state is given by the three components (b, M, f). Defining the state set is indeed the major task in the BLP model. We do not have to describe inputs, outputs, or the precise structure of state transitions, to give the BLP security properties.

11.2.2 Security Policies

The BLP model defines security as the property of states. Multi-level security policies allow a subject to read an object only if the subject's security level dominates the object's classification. These multi-level security policies are also called *mandatory security policies*.

The Simple Security Property

A first obvious property is the *simple security property* (*ss-property*):

> A state (b, M, f) satisfies the *ss-property* if, for each element $(s, o, a) \in b$ where the access operation a is `read` or `write`, the security level of the subject s dominates the classification of the object o, i.e. $f_O(o) \leq f_S(s)$.

In the ss-property subjects act as observers. This policy captures the traditional *no read-up* security policy that applies when a person requests access to a classified document. However, we are in a computer system where subjects are processes. Thus, subjects have no 'memory' like a person but they have access to memory objects. Subjects can act as channels by writing into one memory object data they are reading from another memory object. In this way, data may be declassified improperly (Figure 11.1). For example, an attacker might insert a high-level Trojan horse that copies the content of higher-level objects into low-level objects.

The ∗-Property

Transferring a policy from the pen-and-paper world into IT systems, we have struck on an issue that did not exist before. In response we might also control write access, but

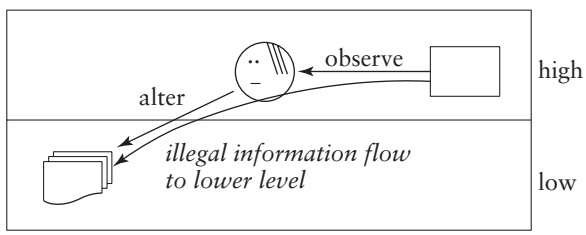


Figure 11.1: Declassification of an Object Using a Subject as Channel

simply preventing subjects from altering objects at lower levels raises a new problem. With such a policy a high-level subject is unable to send any message to a low-level subject. We can escape from this restriction in two ways:

- Temporarily downgrade a high-level subject. This is the reason for introducing the current security level f_C.
- Identify a set of subjects permitted to violate the $*$-property. These subjects are called *trusted subjects*.

The first approach assumes that a subject forgets all it knew at a higher security level the moment it is downgraded. This looks implausible if you view subjects as human beings, but BLP is about modelling computers. There, subjects (processes) have no memory of their own. The only things they 'know' are the contents of the objects (files) they are allowed to observe. In this situation, a temporary downgrade does indeed solve the problem. In an alternative interpretation, f_S specifies a user's clearance. Users are allowed to log in below their clearance and f_C indicates at which level a user actually has logged in.

The BLP model includes a *no-write-down* policy that refers to the current security level f_C, the so-called $*$-*property* (star-property[1]):

> A state (b, M, f) satisfies the $*$-*property* if, for each element $(s, o, a) \in b$ where the access operation a is `append` or `write`, the current level of the subject s is dominated by the classification of the object o, i.e. $f_C(s) \leq f_O(o)$.
>
> Furthermore, if there exists an element $(s, o, a) \in b$ where the access operation a is `append` or `write`, then we must have $f_O(o') \leq f_O(o)$ for all objects o' where $(s, o', a') \in b$ and a' is `read` or `write`.

The illegal information flow downwards in Figure 11.1 is blocked by the $*$-property.

When adopting the second approach, the no-write-down policy only applies to subjects that are not trusted. By definition, a *trusted subject* may violate the security policy. Indeed, to focus your mind you may well use the adjective *trusted* precisely as an indicator for system components that can hurt you. In contrast, if you have convinced yourself that a subject will not hurt you, then call it *trustworthy*.

The Discretionary Security Property

The Orange Book uses *discretionary access control* for policies that control access based on named users and named objects. Subjects holding an access permission may pass that permission on to other subjects. In the BLP model, such policies are expressed by an access control matrix and captured by the *discretionary security property* (*ds-property*):

> A state (b, M, f) satisfies the *ds-property* if, for each element of $(s, o, a) \in b$, we have $a \in M_{so}$.

[1] The first version of the model did not include this property yet, and the symbol $*$ was reputedly used as a placeholder until a proper name for the policy would be found.

11.2.3 The Basic Security Theorem

A state (b, M, f) is called *secure* if the ss-, $*$-, and ds-property are satisfied. A transition from state $v_1 = (b_1, M_1, f_1)$ to state $v_2 = (b_2, M_2, f_2)$ is called secure if both v_1 and v_2 are secure. To see which checks must be performed to determine whether the new state is secure, consider, for example, the ss-property. A state transition *preserves* the ss-property if and only if:

1. each $(s, o, a) \in b_2 \setminus b_1$ satisfies the ss-property with respect to f_2 ($b_2 \setminus b_1$ denotes the set difference between b_2 and b_1);
2. if $(s, o, a) \in b_1$ does not satisfy the ss-property with respect to f_2, then $(s, o, a) \notin b_2$.

Preservation of the $*$-property and of the ds-property can be described in a similar way. We are now in a position to state an important property of the BLP model.

> **Basic security theorem**. If all state transitions in a system are secure and if the initial state of the system is secure, then every subsequent state will also be secure, no matter what inputs occur.

A formal proof of this theorem would proceed by induction over the length of input sequences. The proof would build on the fact that each state transition preserves security but would not refer to the specific BLP security properties.

Lesson
The basic security theorem is a consequence of state machine modelling, not a consequence of the specific security properties chosen in the BLP model.

In practice, the basic security theorem limits the effort needed to verify the security of a system. You can check state transitions individually to show that they preserve security and you have to identify a secure initial state. As long as you start your system in this secure initial state, it will remain secure.

11.2.4 Tranquility

In a paper in 1987 [164], McLean triggered a heated debate about the value of the BLP model by putting forward a system that contained a state transition which

- downgraded all subjects to the lowest security level,
- downgraded all objects to the lowest security level,
- entered all access rights in all positions of the access control matrix M.

The state reached by this transition is secure according to the BLP definitions. Should such a state be regarded as secure? As the BLP model says that this is the case, does BLP capture security correctly? There are two opinions.

- The case against BLP (McLean): Intuitively, a system that can be brought into a state where everyone is allowed to read everything is not secure. Therefore, the BLP model has to be improved.
- The case for BLP (Bell): If the user requirements call for such a state transition, then it should be allowed in the security model. If it is not required, then it should not be implemented. This is not a BLP problem but a problem of correctly capturing the security requirements.

At the root of this disagreement is a state transition that changes access rights. Such changes are certainly possible within the general BLP framework, but the originators of the model were actually contemplating systems where access rights are fixed. The property that security levels and access rights never change is called *tranquility*. Operations that do not change access rights are called *tranquil*.

11.2.5 Aspects and Limitations of BLP

BLP is a very significant security model. It played an important role in the design of secure operating systems and almost every new model was compared to it. In this context it is helpful to separate several features of the BLP model.

1. The descriptive capabilities of the model: the BLP state set describes all current access operations and all current access permissions.
2. The security policies are based on security levels and an access control matrix. It is easy to introduce other structures in their place. For example, to model access control in a situation where a subject is allowed to access objects only through certain programs, an $S \times S \times O$ access control structure is more appropriate (cf. Section 8.5.1).
3. The actual security properties: the BLP model has the ss-property, $*$-property, and ds-property. The Biba model (Section 12.1) differs from BLP mainly in its security properties.
4. The specific solution: e.g. the state transitions in the Multics interpretation (Section 11.3).

The fact that the BLP model defines security in terms of access control is a major reason for its popularity. Therefore, it is not too difficult to express the actions of an operating system or a database management system in terms of BLP. However, although it is an important security model, BLP does not cover all aspects of security. It has been criticized for:

- only dealing with confidentiality, not with integrity;
- not addressing the management of access control;
- containing covert channels.

The absence of integrity policies is a feature of the BLP model, rather than a flaw. As you will see in the next chapter, it is quite reasonable for a security model to limit its

ambitions. BLP has no policies regulating the modification of access rights. As a matter of fact, BLP was originally intended for systems where there is no change of security levels.

A *covert channel* is an information flow that is not controlled by a security mechanism [53]. Object names are a blatant covert channel if low-level subjects may see high-level object names and are only denied access to the contents of the objects. In the BLP model, you could use the access control mechanism itself to construct a covert channel. Information could flow from a high security level to a low security level as follows:

- A low-level subject creates an object dummy.obj at its own level.
- Its high-level accomplice (a Trojan horse?) either upgrades the security level of dummy.obj to high or leaves it unchanged.
- Later, the low-level subject tries to read dummy.obj. Success or failure of this request discloses the action of the high-level subject. One bit of information has been transmitted from high to low.

Telling a subject that a certain operation is not permitted constitutes information flow. This leads to interesting solutions in database security (*polyinstantiation*), where an object may have different values at different security levels to avoid this kind of problem [79, 157].

Lesson
Sometimes it is not sufficient to hide only the contents of objects. Their existence may also have to be hidden.

11.3 THE MULTICS INTERPRETATION OF BLP

The Multics (Multiplexed Information and Computing Service) operating system was the object of an ambitious research project aiming to develop a secure, reliable, etc., multi-user operating system [25, 187]. Much research on security, like BLP, was linked to the Multics project. An overview of the protection mechanisms in Multics is given in [187, Chapter 4]. Because of its wide-ranging goals and security requirements, Multics became too cumbersome for some project members, who then created something much simpler, viz. Unix. The history of the two systems highlights the balance between usability and security as far as commercial success is concerned. The last system running Multics was decommissioned in 2000.

Studying Multics gives us a chance to see how a security model, the Bell–LaPadula model in this case, is used in the design of a secure operating system. As a formal model for access control, the BLP model is very well suited to capturing the security requirements

of operating systems. As a matter of fact, it was developed just for that purpose. The inductive definition of security in the BLP model makes it relatively easy to build a secure system. We only need to define state transitions properly to guarantee security. To prove that Multics is secure, we have to find a description of Multics that is consistent with the BLP model. We will follow by and large the presentation given in [25] to show how BLP concepts are mapped into Multics.

11.3.1 Subjects and Objects in Multics

The *subjects* in Multics are processes. Each subject has a *descriptor segment* that contains information about the process, including information about the objects the process currently has access to. For each of these objects, there is a *segment descriptor word* (SDW) in the subject's descriptor segment. The format of the SDW is given in Figure 11.2. The SDW contains the name of the object, a pointer to the object, and *indicator* flags for read, execute, and write access. These indicators refer to the access attributes specified in Section 5.3.2. The security levels of subjects are kept in a *process-level table* and a *current-level table*. The *active segment table* keeps track of all active processes. Only active processes have access to an object.

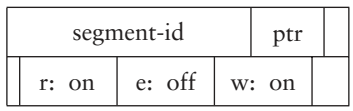


Figure 11.2: Multics Segment Descriptor Word

Objects in Multics are memory segments, I/O devices, etc. Objects are organized hierarchically in a directory tree. Directories are again segments. Information about an object, like its security level or its access control list (ACL), are kept in the object's parent directory. To change an object's access control parameters and to create or delete an object requires write or append access right to the parent directory.

To access an object, a process has to traverse the directory tree from the *root* directory to the target object. If a directory in this path is not accessible to the process, the target object is not accessible either. In other words, an unclassified object in a secret directory cannot be read by an unclassified user. Hence, it makes little sense to place objects into directories with a higher security level and we always require that the security level of an object dominates the security level of its parent directory. This property is called *compatibility*. You have to deal with the same issue in modern operating systems. If you want to make your files accessible to other users, you also have to get the access control settings on the directory path right.

We now have all the necessary information to identify the components of the BLP state set with data in the Multics systems tables and descriptor segments.

- The current access b: stored in the SDWs in the descriptor segments of the active processes; the active processes are found in the *active segment table*.
- The access control matrix M: represented by the ACLs. For each object, the ACL is stored in the object's parent directory; each ACL entry specifies a process-id and the access rights the process-id has on that object.
- The level function f: the security levels of subjects are stored in the *process-level table* and the *current-level table*; the security level of an object is stored in its parent directory.

11.3.2 Translating the BLP Policies

Section 5.3.2 has already introduced the Multics access attributes for data segments and directory segments. We also explained how these access attributes correspond to the access rights of the Bell–LaPadula model. As a reminder, we restate the access attributes for data segments:

access attribute	access right
read	r
execute	e, r
read and write	w
write	a

The BLP security properties now have to be rephrased in terms of the security levels of processes and data segments and of the indicators stored in the SDWs. For example, the $*$-property is written as follows:

> For any SDW in the descriptor segment of an active process, the current level of the process:
>
> - dominates the level of the segment if the read or execute indicator are on and the write indicator is off;
> - is dominated by the level of the segment if the read indicator is off and the write indicator is on;
> - is equal to the level of the segment if the read indicator is on and the write indicator is on.

Figure 11.3 indicates how compliance with the $*$-property is verified. The security level L_c of the current process is held in the *current-level table*. The contents of the *descriptor segment base register* (DSBR) point to the head of the descriptor segment of the current process. This descriptor segment happens to contain the SDW for an object where the access attribute is write only. Hence, the write indicator is on and the read indicator is off. The object's security level L_o is taken from its parent directory and compared with L_c to check that $L_c \geq L_o$ holds.

11.3.3 Checking the Kernel Primitives

Finally, a set of *kernel primitives* has to be specified. These kernel primitives are the state transitions in an abstract model of the Multics kernel. We have to show that they

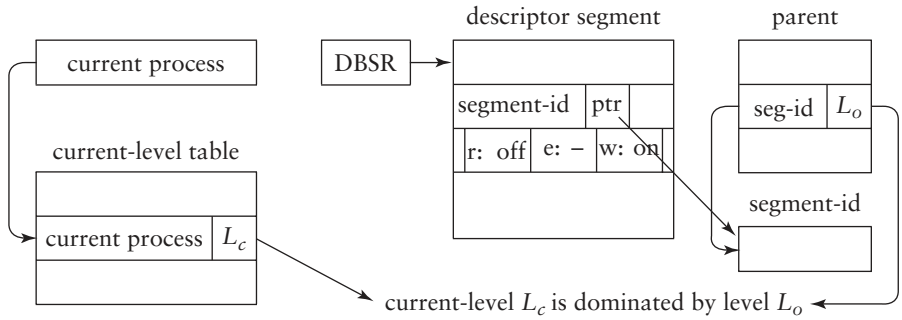


Figure 11.3: The ∗-Property for Access Attribute Write (only)

preserve the BLP security policies. Then the preconditions of the basic security theorem hold and we have a proof for the 'security' of Multics. Of course, this is not a complete proof of security. We would still have to show that the implementation of the kernel primitives, and in the end their execution on a given hardware platform, complies with their specification.

We choose get-read to look at a kernel primitive in detail. The get-read primitive takes as its parameters a process-id and a segment-id. The operating system has to check whether:

- the ACL of segment-id, stored in the segment's parent directory, lists process-id with read permission;
- the security level of process-id dominates the security level of segment-id;
- process-id is a trusted subject or the current security level of process-id dominates the security level of segment-id.

If all three conditions are met, access is permitted. If no SDW for segment-id exists, a corresponding SDW is added to the descriptor segment of process-id with the read indicator turned on. If a SDW for segment-id already exists in the descriptor segment of process-id, the read indicator in this SDW is turned on. If any of the three conditions is not met, access is denied.

We list some further primitives that were proposed for implementation in the Multics kernel:

- release-read – a process releases an object; the read flag in the corresponding SDW is turned off; if thereafter no indicator is on, the SDW is removed from the descriptor segment.
- give-read – a process grants read access to another process (discretionary access control).

- rescind-read – a process withdraws a read permission given to another process.
- create-object – a process creates an object; the operating system has to check that write access on the object's directory segment is permitted and that the security level of the segment dominates the security level of the process.
- delete-object – when deleting an object, the same checks are performed as in create-object.
- change-subject-current-security-level: the operating system has to check that no security violations are created by the change; this kernel primitive, along with the primitive change-object-security-level, was not intended for implementation (tranquillity).

Ideally, processors are developed so that their instruction sets dovetail with the kernel primitives of the operating system. Conversely, kernel primitives can be designed to match the support provided by existing processors.

11.4 FURTHER READING

The original reports on the Bell–LaPadula model have been republished in the *Journal of Computer Security* [24]. A framework for policies on changing access rights in the Bell–LaPadula model is discussed in [165]. The research literature is full of contributions extending the scope of the BLP model while still enforcing its MLS policies.

The outcome of computer security research aimed at the first multi-user operating systems is treated comprehensively in [80]. A further survey of protection techniques is compiled in [146]. Comments on Multics security and, in particular, on the complexity of security management and the complexity of assessing the correctness of the design can be found in [196]. Lessons learned during the work on Multics security are discussed in [138].

11.5 EXERCISES

Exercise 11.1 Describe the $*$-property in terms of the basic access modes Alter and Observe.

Exercise 11.2 Identify further covert channels in the BLP model.

Exercise 11.3 Write a short essay stating your position in the Bell vs McLean debate.

Exercise 11.4 Bell–LaPadula does not specify policies for changing access rights. What policies would you suggest?

Exercise 11.5 Rewrite the ss-property for the Multics operating system.

Exercise 11.6 Specify the checks that have to be made for get-write and release-write kernel primitives.

Exercise 11.7 Specify the checks that have to be made for create-object and delete-object kernel primitives.

Exercise 11.8 Specify the checks that have to be made for the change-subject-current-security-level kernel primitive.

Chapter 12

Security Models

The Bell–LaPadula model was designed to capture one specific security policy. It was, however, so successful that for a time it was treated as 'the model of security' in general. Not surprisingly it was found lacking in this respect, but this should not be taken as a criticism of the model itself. Rather, security requirements are application-dependent and there exist applications that are substantially different from the military environment multi-level security comes from.

This chapter will take a wider look at security models. We will add models for integrity policies (Biba, Clark–Wilson) and for policies that, in contrast to the tranquility assumption of the BLP model, dynamically change access rights (Chinese Wall). We will also use the term 'security model' in a wider sense. The Clark–Wilson model, for example, does not formalize a single specific policy but provides a descriptive framework that can serve as a blueprint for formalizing a wider class of security policies. Furthermore, we will discuss models that are of interest primarily from a theoretical point of view as they provide a basis for proving some fundamental facts about access control.

OBJECTIVES

- Present a wider range of modelling techniques for access control.
- Introduce basic concepts relevant to commercial security policies.
- Provide theoretical foundations for analyzing access control problems.
- Appreciate that some decision problems in security are inherently undecidable.

12.1 THE BIBA MODEL

Consider integrity policies that label subjects and objects with elements from a lattice $(L, \leq)$ of integrity levels and that prohibit the corruption of 'clean' high-level entities by 'dirty' low-level entities. Information may only flow downwards in the integrity lattice. As in the BLP model, we will only contemplate information flows caused directly by access operations. 'Clean' and 'dirty' are used as shorthand for high integrity and low integrity. The concrete meaning of integrity levels would depend on the given application.

The Biba model [35] formalizes this type of integrity policy. It is a state machine model similar to BLP, and we will use the mathematical notations introduced in the previous chapter. The assignment of integrity levels to subjects and objects is given by the functions $f_S : S \to L$ and $f_O : O \to L$. Unlike BLP, there is no single high-level integrity policy. Instead, there are a variety of approaches. Some even yield mutually incompatible policies.

12.1.1 Static Integrity Levels

Mirroring the tranquillity property of the BLP model, we can state policies where integrity levels never change. The following two policies *prevent* clean subjects and objects from being contaminated by dirty information.

Simple integrity property (no write-up). If subject s can modify (Alter) object o, then $f_S(s) \geq f_O(o)$.

Integrity $*$-property (no read-down). If subject s can read (Observe) object o, then s can have write access to some other object o' only if $f_O(o') \leq f_O(o)$.

These two integrity properties are the dual of the mandatory BLP policies and are the basis for claims that integrity is the dual of confidentiality.

12.1.2 Dynamic Integrity Levels

The next two integrity properties automatically *adjust* the integrity level of an entity if it comes into contact with dirty information. The integrity level $\inf(f_S(s), f_O(o))$ is the greatest lower bound of $f_S(s)$ and $f_O(o)$. It is well defined as we have a lattice of integrity levels.

Subject low watermark property. Subject s can read (Observe) an object o at any integrity level. The new integrity level of the subject is $\inf(f_S(s), f_O(o))$, where $f_S(s)$ and $f_O(o)$ are the integrity levels before the operation.

Object low watermark property. Subject s can modify (Alter) an object o at any integrity level. The new integrity level of the object is $\inf(f_S(s), f_O(o))$, where $f_S(s)$ and $f_O(o)$ are the integrity levels before the operation.

These are examples of policies with dynamically changing access rights. As integrity levels can only be lowered, there is a danger that all subjects and objects eventually sink to the lowest integrity level. Note that organizations working with MLS policies observed a corresponding tendency. Objects had a way of percolating up to System High over time.

12.1.3 Policies for Invocation

The Biba model can be extended to include an access operation invoke. A subject can *invoke* another subject, e.g. a software tool, to access an object. This is a step towards formulating access control at intermediate layers. What kind of policy should govern invocation? To make sure that invocation does not bypass the *mandatory integrity policies* we could add the

> *Invoke property*. Subject s_1 can invoke subject s_2 only if $f_S(s_2) \leq f_S(s_1)$.

Subjects are only allowed to invoke tools at a lower level. Otherwise, a dirty subject could use a clean tool to access, and contaminate, a clean object (see Section 6.3.6).

Alternatively, we may use tools for this very purpose: dirty subjects may have access to clean objects, but only if they use a clean tool to do so (controlled invocation). This tool may perform a number of consistency checks to ensure that objects remain clean. Integrity protection mechanisms in operating systems that use *protection rings* (Section 5.6.4) fall into this category. In this scenario, a more privileged subject should not use less privileged tools and we get the

> *Ring property*. A subject s can read objects at all integrity levels. It can only modify objects o with $f_O(o) \leq f_S(s)$; it can invoke a subject s' only if $f_S(s) \leq f_S(s')$.

Quite obviously, the last two properties are contradictory. It will depend on the application which property is more appropriate.

12.2 CHINESE WALL MODEL

The Chinese Wall model by Brewer and Nash captures access rules in a consultancy business. Analysts have to avoid conflicts of interest when dealing with different clients [44]. Informally, conflicts arise because clients are direct competitors in the same market or because of the ownerships of companies. Analysts are governed by the following security policy:

> There must be no information flow that causes a conflict of interest.

The state set of the Bell–LaPadula model needs some slight adaptations to address this policy.

- The set of companies is denoted by C.
- The analysts are the *subjects* and S is the set of subjects.
- The *objects* are items of information. Each object refers to a single company. The set of objects is denoted by O.

- All objects concerning the same company are collected in a *company data set*. The function $y : O \to C$ gives the company data set of each object.
- *Conflict of interest classes* indicate which companies are in competition. The function $x : O \to \mathcal{P}(C)$ gives the conflict of interest class for each object, i.e. the set of all companies that should not learn about the contents of the object.
- The *security label* of an object o is the pair $(x(o), y(o))$.
- *Sanitized information* has been purged of sensitive details and is not subject to access restrictions. The security label of a sanitized object is $(\emptyset, y(o))$.

Conflicts of interest arise not only from objects currently accessed but also from objects that have been accessed in the past. We therefore need a data structure that records the history of the subjects' actions. This purpose is served by a Boolean $S \times O$ matrix N, with

$$N_{s,o} = \begin{cases} \text{TRUE, if the subject } s \text{ has had access to object } o, \\ \text{FALSE, if the subject } s \text{ has never had access to object } o. \end{cases}$$

Setting $N_{s,o} = \text{FALSE}$ for all $s \in S$ and all $o \in O$ gives an initial state that fulfils the security properties below.

The first security policy deals with direct information flow. We want to prevent a subject from being exposed to a conflict of interest. Therefore, access is granted only if the object requested belongs to

- a company data set already held by the user, or
- an entirely different conflict of interest class.

Formally, we can express this as follows.

> *ss-property*. A subject s is permitted to access an object o only if for all objects o' with $N_{s,o'} = \text{TRUE}$, $y(o) = y(o')$ or $y(o) \notin x(o')$.

On its own, this property does not guarantee compliance with the stated security policy. Indirect information flow is still possible. Consider the following example (Figure 12.1). Two competitors, *A* and *B*, have their accounts with the same *Bank*. *Analyst_A*, dealing with *A* and the *Bank*, updates the *Bank* portfolio with sensitive information about company *A*. *Analyst_B*, dealing with company *B* and the *Bank*, now has access to information about a competitor's business. Therefore, also write access is regulated.

> $*$-*property*. A subject s is granted write access to an object o only if s has no read access to an object o' with $y(o) \neq y(o')$ and $x(o') \neq \emptyset$.

Write access to an object is only granted if no other object belonging to a different company data set that contains unsanitized information can be read. In the example of Figure 12.1 both write operations are blocked by the $*$-property. The $*$-property stops unsanitized information from flowing out of a company data set.

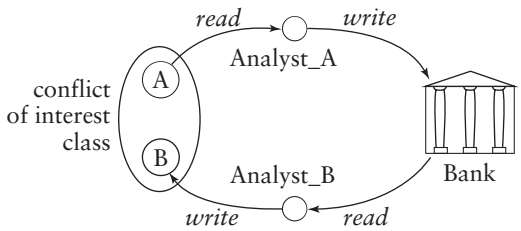


Figure 12.1: Indirect Information Flow within a Conflict of Interest Class

In contrast to the BLP model, where the assignment of access rights is usually assumed to be static, we have here a model where access rights have to be reassigned in every state transition.

12.3 THE CLARK–WILSON MODEL

Clark and Wilson address the security requirements of commercial applications [66]. They argue that these requirements are predominantly about (data) integrity, i.e. about preventing unauthorized modification of data, fraud, and errors. This is a rather wide definition of integrity. In fact, the authors even include issues of concurrency control, which are beyond our scope of security. Integrity requirements are divided into two parts:

- *internal consistency*, which refers to properties of the internal state of a system and can be enforced by the computing system;
- *external consistency*, which refers to the relation of the internal state of a system to the real world and has to be enforced by means outside the computing system, e.g. by auditing.

The general mechanisms for enforcing integrity are as follows:

- *Well-formed transactions* – data items can be manipulated only by a specific set of programs; users have access to programs rather than to data items.
- *Separation of duties* – users have to collaborate to manipulate data and to collude to circumvent the security system.

Separation of duties is pervasive in the operation of a secure system. It is reasonable to require that different persons develop, test, certify and operate a system. In turn, it may be required that during operation different persons have to collaborate to enable a transaction.

The Clark–Wilson model uses programs as an intermediate layer between subjects and objects (data items). Subjects are authorized to execute certain programs. Data items can

be accessed through specific programs. Defining the set of programs that may access data of a certain type is a general mechanism in software engineering (see *abstract data types*, *object-oriented programming*), which can be gainfully employed in constructing secure systems. It is testimony to the influence of the BLP model that Clark and Wilson write about 'labelling subjects and objects with programs instead of security levels'.

In the Clark–Wilson model, integrity means 'being authorized to apply a program to a data item that may be accessed through this program'. Clark and Wilson stress the difference between military and commercial security requirements. There is some truth in the observation that the relative importance of confidentiality and integrity is not the same in these two worlds, but there will be military applications with integrity requirements and commercial applications with confidentiality requirements. For us, there is a much more relevant distinction. The access operations in the Clark–Wilson model are programs performing complex application-specific manipulations. Access operations in the BLP model are simple and generic, as befits an operating system. We are observing the difference between a general purpose operating system (BLP) and an application-oriented IT system (Clark–Wilson).

The following points are considered in the Clark–Wilson model (see Figure 12.2):

1. Subjects have to be identified and authenticated.
2. Objects can be manipulated only by a restricted set of programs.
3. Subjects can execute only a restricted set of programs.
4. A proper audit log has to be maintained.
5. The system has to be certified to work properly.

In a formalization of this model, the data items governed by the security policy are called constrained data items (CDIs). Inputs to the system are captured as unconstrained data

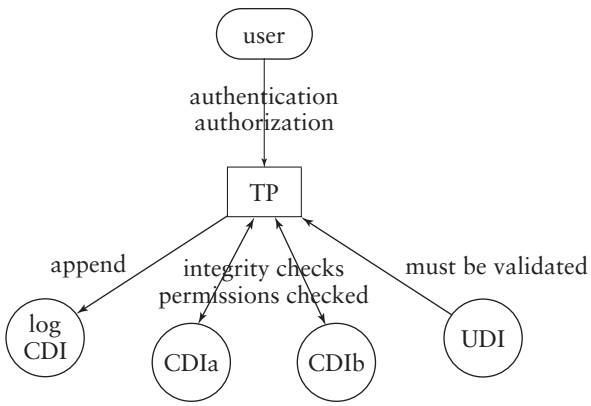


Figure 12.2: Basic Principles of Access Control in the Clark–Wilson Model

items (UDIs). Conversion of UDIs to CDIs is a critical part of the system which cannot be controlled solely by the security mechanisms in the system. CDIs can be manipulated only by transformation procedures (TPs). The integrity of an item is checked by integrity verification procedures (IVPs).

Security properties are defined through five *certification rules*, suggesting the checks that should be conducted so that the security policy is consistent with the application requirements:

CR1 IVPs must ensure that all CDIs are in a valid state at the time the IVP is run (integrity check on CDIs).

CR2 TPs must be certified to be valid, i.e. valid CDIs must always be transformed into valid CDIs; each TP is certified to access a specific set of CDIs.

CR3 The access rules must satisfy any separation-of-duties requirements.

CR4 All TPs must write to an append-only log.

CR5 Any TP that takes a UDI as input must either convert the UDI into a CDI or reject the UDI and perform no transformation at all.

Four *enforcement rules* describe the security mechanisms within the computer system that should enforce the security policy. These rules have some similarity with discretionary access control in the BLP model.

ER1 For each TP, the system must maintain and protect the list of entries (CDI*a*, CDI*b*, ...) giving the CDIs the TP is certified to access (capability of the TP).

ER2 For each user the system must maintain and protect the list of entries (TP1, TP2, ...) specifying the TPs the user can execute (capability of the user).

ER3 The system must authenticate each user requesting to execute a TP.

ER4 Only a subject that may certify an access rule for a TP may modify the respective entry in the list. This subject must not have execute rights on that TP.

The Clark–Wilson model is a framework and guideline ('model') for formalizing security policies rather than a model of a specific security policy. It stresses the importance of management approval of the processes and of the security policies to be followed in an organization. The model refers to this step as *certification*.

12.4 THE HARRISON–RUZZO–ULLMAN MODEL

The Bell–LaPadula model does not state policies for changing access rights or for the creation and deletion of subjects and objects. The Harrison–Ruzzo–Ullman (HRU)

model defines *authorization systems* that address these issues [114]. In the HRU model, there is:-

- a set of *subjects* S;
- a set of *objects* O;
- a set of *access rights* R;
- an access matrix $M = (M_{so})_{s \in S,\, o \in O}$; the entry $M_{so} \subseteq R$ specifies the rights subject s has on object o.

There exist six *primitive operations* for manipulating the set of subjects, the set of objects, and the access matrix:

- **enter** r into M_{so}
- **create subject** s
- **create object** o
- **delete** r from M_{so}
- **delete subject** s
- **delete object** o

The HRU model has a simple programming language for writing commands. Commands have the format

command $c(x_1, \ldots, x_k)$
  if r_1 in M_{s_1,o_1} **and**
  if r_2 in M_{s_2,o_2} **and**
    $\vdots$
  if r_m in M_{s_m,o_m}
  then
    op_1
    op_2
    $\vdots$
    op_n
end

The indices $s_1, \ldots, s_m$ and $o_1, \ldots, o_m$ are subjects and objects that appear in the parameter list $(x_1, \ldots, x_k)$. The conditions check whether particular access rights are present. The list of conditions may be empty. If all conditions hold, the sequence of basic operations is executed. Each command contains at least one operation. For example, the command

command *create_file*(s, f)
    create f
    enter $\underline{\text{o}}$ into $M_{s,f}$
    enter $\underline{\text{r}}$ into $M_{s,f}$
    enter $\underline{\text{w}}$ into $M_{s,f}$
end

is used by subject s to create a new file f so that s is the owner of the file (access right $\underline{\text{o}}$) and has read and write permission to the file (access rights $\underline{\text{r}}$ and $\underline{\text{w}}$). The owner s of file f grants read access to another subject p with

command *grant_read*(s, p, f)
  if $\underline{\circ}$ in $M_{s,f}$
  then enter $\underline{r}$ in $M_{p,f}$
end

An authorization system is defined by a set of commands and by its state, captured by the access matrix. The effect of a command is recorded as a change to the access matrix. We denote the modified access control matrix by M'. The HRU model can capture security policies regulating the allocation of access rights. To verify that a system complies with such a policy, you have to check that there exists no way for undesirable access rights to be granted.

> A state of an authorization system is said to *leak* the right r if there exists a command c that adds the right r into a position of the access matrix M that previously did not contain r. More formally, there exist s and o so that $r \notin M_{s,o}$ but $r \in M'_{s,o}$.
>
> A state of an authorization system, i.e. an access matrix M, is said to be *safe* with respect to the right r if no sequence of commands can transform M into a state that leaks r.

Verifying compliance with a security policy in the HRU model thus comes down to verifying safety properties (see also Section 12.6). The following theorem holds.

> **Theorem.** Given an authorization system with access matrix M and a right r, verifying the safety of M with respect to the right r is an undecidable problem [114].

You now find yourself in the unenviable position of not being able to tackle the safety problem in its full generality. You have to restrict the HRU model to have a better chance of success. For example, you could only allow *mono-operational* systems in which each command contains a single operation.

> **Theorem.** Given a mono-operational authorization system, an access matrix M, and a right r, verifying the safety of M with respect to the right r is decidable [114].

Limiting the size of the authorization system is another way of making the safety problem tractable.

> **Theorem.** The safety problem for arbitrary authorization systems is decidable if the number of subjects is finite [155].

These results on the decidability of the safety problem reveal glimpses of the third design principle (Section 3.4.3). If you design complex systems that can only be described by complex models, it becomes difficult to find proofs of security. In the worst case (undecidability), there does not exist a universal algorithm that verifies security for all problem instances. If you want verifiable security properties, you are better off limiting the complexity of the security model. Such a model may not describe all desirable security properties, but you may gain efficient methods for verifying 'security'. In turn, you would

be advised to design simple systems that can be adequately described in the simple model. If there is too wide a gap between system and model, proofs of security in the model will not carry much weight.

Lesson
The more expressive a security model is, both with respect to the security properties and the systems it can describe, the more difficult it usually is to verify security properties.

12.5 INFORMATION-FLOW MODELS

In the Bell–LaPadula model, information can flow from a high security level to a low security level through a covert channel. *Information-flow models* consider any kind of information flow, not only the direct information flow through access operations modelled in BLP. Informally, a state transition causes an information flow from an object x to an object y, if we may learn more about x by observing y. If we already know x, no information can flow from x. Information flow may be explicit or implicit:

- Explicit information flow – observing y after the assignment `y:= x;` tells you the value of x.
- Implicit information flow – observing y after the conditional statement `IF x=0 THEN y:=1;` may tell you something about x even if the assignment `y:= 1;` has not been executed. For example, if $y = 2$, you know that $x \neq 0$.

12.5.1 Entropy and Equivocation

Information theory can give precise and quantitative definitions of information flow. The amount of information derivable from an observation is formally defined by the entropy of the object (variable) we are observing. Let $\{x_1, \ldots, x_n\}$ be the values a variable x can take, and let $p(x_i)$ be the probability of x taking the value x_i, $1 \leq i \leq n$. The *entropy* $H(x)$ of x is defined as

$$H(x) = -\sum_{i=1}^{n} p(x_i) \log_2 p(x_i).$$

For example, let x take all values between 0 and $2^w - 1$ with equal probability. Then

$$H(x) = -\sum_{i=1}^{2^w} \frac{1}{2^w} \log_2 \left(\frac{1}{2^w} \right) = w.$$

That is, a binary word of length w carries w bits of information if all words of length w are equally likely.

The information flow from x to y is measured by the change in the equivocation (conditional entropy) of x given the value of y. Let x and y be two variables that can take the values $\{x_1, \ldots, x_n\}$ and $\{y_1, \ldots, y_m\}$ with probabilities $p(x_i)$ and $q(y_j)$. Let $p(x_i, y_j)$ be the joint probability of x and y taking the values x_i and y_j, and let $p(x_i \mid y_j)$ be the conditional probability of x taking the value x_i if y takes the value y_j. The *equivocation* $H_y(x)$ of x given the value of y is defined as

$$H_y(x) = -\sum_{i=1}^{n}\sum_{j=1}^{m} p(x_i, y_j)\log_2 p(x_i|y_j).$$

From $p(x_i, y_j) = p(x_i|y_j)q(y_j)$ we get

$$H_y(x) = -\sum_{j=1}^{m} q(y_j)\sum_{i=n}^{m} p(x_i|y_j)\log_2 p(x_i|y_j).$$

As an example, consider the assignment `IF x=0 THEN y:=1;` from above. Let x and y be binary variables, with y initially set to 0 and both values of x equally likely. If $y = 0$ still holds after the assignment, x must have been 1; if $y = 1$, x must have been 0. We get

$$p(0|0) = p(1|1) = 0 \text{ and } p(1|0) = p(0|1) = 1, \text{ hence } H_y(x) = 0.$$

Indeed, after performing the assignment and observing y, we know the exact value of x. All information in x has flowed to y. If x can take the values 0, 1, 2 with equal probability, we get $q(0) = \frac{2}{3}, q(1) = \frac{1}{3}$,

$$p(0|0) = p(1|1) = p(2|1) = 0,\ p(1|0) = p(2|0) = \frac{1}{2},\ p(0|1) = 1, \text{ and } H_y(x) = \frac{2}{3}.$$

12.5.2 A Lattice-Based Model

The components of the information-flow model are:

- a lattice $(L, \leq)$ of *security labels*;
- a set of labelled *objects*;
- the security policy – information flow from an object with label c_1 to an object with label c_2 is permitted only if $c_1 \leq c_2$, and any information flow violating this rule is illegal.

A system is called *secure* if there is no illegal information flow. The advantage of such a model is that it covers all kinds of information flow. The disadvantage is that it becomes more difficult to design secure systems. For example, it has been shown that checking whether a given system is secure in the information-flow model is an *undecidable* problem.

We can further distinguish between *static* and *dynamic* enforcement of information-flow policies. In the first case, the system (program) is considered as a static object. The second case considers the system under execution. We may find that some information

flow may be possible in theory (and therefore should be detected in the static analysis) but will never occur during execution. Therefore, static analysis tends to produce too restrictive systems.

Non-interference models are an alternative to information-flow models. They provide a different formalism to describe what a subject knows about the state of the system. Subject s_1 does not interfere with subject s_2 if the actions of s_1 have no influence on s_2's view of the system. To prove that there can be no information flow between security levels – we will use *high* and *low* to keep the presentation simple – you have to show that for every execution involving *high* and *low* subjects there exists another execution involving only *low* subjects, and the effects of the two executions are indistinguishable for *low* subjects.

Currently, information-flow and non-interference models are areas of research rather than the basis of a practical methodology for the design of secure systems.

12.6 EXECUTION MONITORS

The previous two sections have shown that certain security problems are undecidable. There cannot be a general algorithm that solves all instances of these problems. Now our theoretical investigations will follow a different route. We will start from the typical access control mechanisms in use today and characterize the policies these mechanisms can enforce. After all, a policy is useful in practice only if it can be enforced reasonably efficiently. We consider three classes of security policies [205]:

- access control policies – define restrictions on the operations principals can perform on objects;
- information-flow policies – restrict what principals can infer about objects from observing system behaviour (see Section 12.5);
- availability policies – restrict principals from denying others the use of a resource.

Access control should prevent insecure behaviour of a *target system*. The mechanisms deployed today in firewalls, operating systems, middleware architectures such as CORBA, or in web services have in common that they monitor the execution of that target system and step in if an execution step is prohibited by the given security policy. The term *execution monitoring* (EM) was introduced in [205] for enforcement mechanisms that monitor the execution steps of a target system and terminate the target's execution if a violation of the security policy is about to occur.

Execution monitors have two important limitations. First, they do not have a model of the target system, so they cannot predict the outcomes of possible continuations

of the execution they are observing. Compilers and theorem-provers, for example, work by analyzing a static representation of the target and can deduce information about all of its possible executions. These methods are therefore not EM mechanisms. Secondly, EM mechanisms cannot modify a target before executing it. In-line reference monitors and reflection in object-oriented systems thus do not fall into the execution monitor category.

12.6.1 Properties of Executions

Executions of the target system are sequences of steps. The precise nature of these steps depends on the actual target. Typical examples are memory access operations and file access operations. Let Ψ denote the set of all finite and infinite sequences of steps, and Σ_S the sequences representing executions of the target system S. A security policy p is a predicate on the set of executions. A target S satisfies the security policy p if $p(\Sigma_S)$ equals TRUE. The safety property of the HRU model, but also the BLP, Biba, and Chinese Wall policies, are all examples of security policies.

Let Σ denote a set of executions. A security policy p that can be enforced by an execution monitor must be specified by a predicate of the form

$$p(\Sigma) : (\forall \sigma \in \Sigma : \hat{p}(\sigma))$$

where $\hat{p}$ is a predicate on individual executions. This observation provides a link to the literature on linear-time concurrent program verification [7]. There, a set $\Gamma \subset \Psi$ of executions is called a *property* if membership of an element is determined by the element alone, not by other members of the set. A security policy must therefore be a property to have an enforcement mechanism in EM.

Not every security policy is a property. Some security policies cannot be defined as a predicate on individual executions. For example, to check compliance with information-flow policies you have to show that given execution is indistinguishable from another which is guaranteed to contain no information flow (see the discussion at the end of Section 12.5).

Furthermore, not every property is EM enforceable. Enforcement mechanisms in EM cannot look into the future when making decisions on an execution. Consider an execution σ that complies with the security policy but has a prefix σ' that does not. Informally, the execution goes through an 'insecure' state but would be permissible in the end. As a simple example, consider a policy that requires a matching 'close file' for every 'open file' command. An execution monitor has to prohibit an insecure prefix and stop executions that would be secure. For such policies, EM would be a *conservative* approach that stops more executions than necessary.

12.6.2 Safety and Liveness

Among the properties of executions there are two broad classes of particular significance.

- Safety properties – nothing bad can happen. (The 'safety' property of access matrices in the HRU model meets this description.)
- Liveness properties – something good will happen eventually.

There exists a close relationship between safety and the policies that can be enforced by execution monitors. We formally define safety properties by characterizing their complements. In the definition, the first i steps of a sequence $\sigma \in \Psi$ will be denoted by $\sigma[..i]$. A property Γ is called a safety property if, for every finite or infinite execution σ,

$$\sigma \notin \Gamma \Rightarrow \exists i(\forall \tau \in \Psi : \sigma[..i]\tau \notin \Gamma)$$

holds [143]. If an execution σ is unsafe, the execution has to have some point of no return i after which it is no longer possible to revert to a safe continuation of the execution.

If the set of executions for a security policy is not a safety property, then there exists an unsafe execution that could be extended by future steps into a safe execution. As discussed above, such properties (policies) do not have an enforcement mechanism from EM. So, if a policy is not a safety property, it is not EM enforceable. Put the other way round, execution monitors enforce security policies that are safety properties. However, not all safety properties have EM enforcement mechanisms. This leads us to the following classification.

- Information-flow policies do not define sets of executions that are properties; thus, information flow cannot be a safety property and in turn cannot be enforced by EM.
- Availability policies define properties but not safety properties; any partial execution could be extended so that the principal would get access to the resource in the end.
- Availability policies that refer to a maximum waiting time (MWT) [100] are safety properties; once an execution has waited beyond the MWT, any extension will naturally also be in violation of the availability policy; MWT policies cannot be enforced by EM as they refer to time.
- Access control policies define safety properties; partial executions ending with an unacceptable operation being attempted will be prohibited.

12.7 FURTHER READING

Surveys of research on security models are given in [146] and [166]. The original paper by Clark and Wilson is highly recommended reading [66]. An implementation of the Clark–Wilson model using capabilities is described in [136]. A slight extension of the Biba model providing mandatory integrity controls that can be used to implement Clark–Wilson is proposed in [152].

A detailed treatment of the decidability properties of the HRU model and of information-flow models, with definitions, proofs, and more theorems, is given in [80]. For non-interference models, refer to Goguen and Meseguer's seminal paper [101]. Applications of security models in the security evaluation of smart cards are described in [36, 204].

12.8 EXERCISES

Exercise 12.1 The Biba model can capture a variety of integrity policies. Give examples of application areas where

- a policy with static integrity labels,
- a policy with dynamically changing integrity labels,
- the ring property

is appropriate.

Exercise 12.2 Can you use Bell–LaPadula and Biba to model confidentiality and integrity simultaneously? Can you use the same security labels for both policies?

Exercise 12.3 Can you fit the Chinese Wall Model into the Bell–LaPadula framework?

Exercise 12.4 Should the $*$-property in the Chinese Wall model refer to current read access only or to any past read access?

Exercise 12.5 Give a formal model that describes the Clark–Wilson enforcement rules.

Exercise 12.6 Let x be a 4-bit variable that can take all values between 0 and 15 with equal probability. Given the assignment `IF x>7 THEN y:=1;` and the initial value $y = 0$, compute the conditional entropy $H_y(x)$.

Exercise 12.7 Develop a security model for documents that are declassified after 30 years.

Exercise 12.8 In a medical information system that controls access to patient records and prescriptions:

- doctors may read and write patient records and prescriptions;
- nurses may read and write prescriptions only but should learn nothing about the contents of patient records.

How can you capture this policy in a lattice model that prevents information flow from patient records to prescriptions? In your opinion, which security model is most appropriate for this policy?

Chapter 13

Security Evaluation

Users of a security-sensitive system need some kind of assurance that they are using adequately secure products. They need answers to two questions. Is the security service provided by the system adequate for meeting the given security requirements? This question refers to the *functionality* of the system. Have the security services been implemented properly so that one can rely on them? This question refers to the *assurance* that the system is providing the expected service. For answers to these questions, users could

1. rely on the word of the manufacturer/service provider,
2. test the system themselves,
3. or rely on an impartial assessment by an independent body (*evaluation*).

Users have to be security experts to be able to take the second option. Most users are not in this position. So some kind of security evaluation is the only alternative to taking a security product on trust. This chapter will explore security evaluation and discuss whether current evaluation schemes have any benefits to offer.

OBJECTIVES

- Appreciate the fundamental problems any security evaluation process has to address.
- Propose a framework for analyzing evaluation criteria.
- Give an overview of the major evaluation criteria.
- Assess the merits of evaluated products and systems.

13.1 INTRODUCTION

An organization aware of the need to protect itself has to decide which security systems to deploy. This is ultimately an executive decision, but the decision-makers are unlikely to have profound security expertise themselves. They will need advice, both on the functionality of the security services to deploy and on suitable products. Advice could come in the form of a best practice recommendation for a business sector. It will tell decision-makers in which direction to turn when selecting security services. Advice can come in the form of test reports on security products. A report should inform the reader about the findings of the tests, but also about the tests conducted.

We will structure our discussion of security evaluation by asking six questions.

What is the *Purpose* of the Exercise?

We follow the terminology of the Orange Book and distinguish between:

- *evaluation* – assessing whether a product has the security properties claimed for it; this is the test of a security product;
- *certification* – assessing whether an (evaluated) product is suitable for a given application; this is the best practice recommendation;
- *accreditation* – deciding that a (certified) product will be used in a given application; this is the executive decision.

These are the terms from the Orange Book. Other sources may use different terms or use the same terms differently. The names given to the various activities are therefore less important than the fundamental differences in their respective goals.

We will focus on the security evaluation of products and systems. Systematic tests need a test plan. *Evaluation criteria* capture the procedures to follow when performing security tests. The *Trusted Computer Security Evaluation Criteria* (TCSEC, Orange Book) [224] were the first evaluation criteria to gain wide acceptance. Several criteria have since been developed to react to perceived shortcomings of the Orange Book and to the changes in the use of IT systems. The desire to unify the different criteria that have arisen has led to the Common Criteria [58]. Important milestones in the development of security evaluation criteria have been the

- Information Technology Security Evaluation Criteria (ITSEC) [70];
- Canadian Trusted Computer Product Evaluation Criteria (CTCPEC) [53];
- Federal Criteria [177].

What is the *Target* of the Evaluation?

Evaluation criteria refer to *products*, i.e. off-the-shelf components that can be used in a variety of applications and have to meet generic security requirements (e.g. an operating

system), and to *systems*, i.e. collections of products assembled to meet the specific requirements of a given application. In the first case, one has to agree on a set of generic requirements. The *security classes* of the Orange Book and the *protection profiles* of the Federal and Common Criteria try to achieve just that. In the second case, requirements capture and analysis becomes part of each individual evaluation. ITSEC was suited to the evaluation of systems.

The distinction between products and systems highlights a fundamental dilemma in security evaluation: users are not security experts but have specific security requirements. Evaluation of off-the-shelf products with respect to generic criteria capturing typical requirements can be a useful decision criterion for a non-expert, but the products may not address the actual security requirements. Evaluation of customized systems will address the perceived requirements, but you now ask the non-expert user to confirm that the security requirements have been properly captured. If further help is provided at this stage, you start to cross the borderline between a security evaluation intended for the general public and the job of a security consultant who advises a particular client.

What is the *Method* of the Evaluation?

The credibility of evaluation very much hinges on the methods used in evaluation. An evaluation method should prevent two situations from arising:

1. An evaluated product is later found to contain a serious flaw.
2. Different evaluations of the same product disagree in their assessment of the product.

Repeatability (re-evaluation by the same team gives the same result) and *reproducibility* (re-evaluation by a different team gives the same result) are therefore often stated as requirements of an evaluation methodology.

Security evaluation can be product-oriented or process-oriented. *Product-oriented* (*investigational*) methods examine and test the product. They may tell more about the product than process-oriented methods, but different evaluations may well give different results. Is this a problem for credibility?

Process (*audit*) *oriented* methods look at documentation and the process of product development. They are cheaper and it is much easier to achieve repeatable results, but the results themselves may not be very valuable. The first version of the European *Information Technology Security Evaluation Manual* [71] was a prime example of repeatability overpowering content. Is this a problem for credibility?

What is the *Organizational Framework* of the Evaluation Process?

Security evaluations should arrive at independent, commonly accepted verdicts on the properties of products. An independent *evaluation facility* can either be a government

agency (the approach taken originally in the US) or a properly accredited private enterprise (e.g. in Europe [219]). In both schemes, a government body backs the evaluation process and issues the certificates. Accredited evaluation facilities might issue certificates themselves. You might even imagine schemes where empirical evidence for the expertise of the evaluators *de facto* replaces a formal accreditation.

A government body may charge for an evaluation or conduct evaluations as a free public service. If all evaluations are conducted by a single government agency, then it will hardly be necessary to create further organizational overheads to ensure the consistency of evaluations. However, there is still the danger of *interpretation drift* (*criteria creep*) over time. Evaluations may be slow due to lack of competition and limited resources at the evaluation facility. There may also be a problem of staff mobility when experienced evaluators leave for higher salaries in the private sector.

In an environment with private evaluation facilities, *certification agencies* have to enforce the consistency of evaluations (repeatability, reproducibility) between different facilities and may confirm the verdict of the evaluation facility. The precise formulation of the criteria becomes more important to avoid differing interpretations. Evaluations are paid for and commercial pressures should lead to faster evaluations and the resources required from the sponsor of an evaluation may be more predictable. On the other hand, precautions have to be taken so that commercial pressures do not lead to incorrect results.

Further organizational aspects concern the contractual relationship between the sponsor of an evaluation, the product manufacturers and the evaluation facility. Moreover, there have to be appropriate procedures for the start of an evaluation, for the issuing of evaluation certificates, and for the re-evaluation of modifications to evaluated products.

What is the *Structure* of the Evaluation Criteria?

Security evaluation aims to give assurance that a product/system is secure. Security and assurance may be related to:

- *functionality* – the security features of a system, e.g. discretionary access control, mandatory access control, authentication, auditing.
- *effectiveness* – are the mechanisms used appropriate for the given security requirements? For example, is user authentication by password sufficient or does the application require a cryptographic challenge-response protocol?
- *assurance* – the thoroughness of the evaluation.

The Orange Book defines evaluation classes for a given set of typical Department of Defense requirements. Therefore, all three aspects are considered simultaneously in the definition of its evaluation classes. ITSEC provided a flexible evaluation framework that can deal with new security requirements. Therefore, the three aspects were addressed independently.

What are the *Costs* and *Benefits* of Evaluation?

In addition to any fee paid for an evaluation, indirect costs also have to be considered, such as the time devoted to producing the evidence required for evaluation, to the training of evaluators, and to liaising with the evaluation team. When considering the cost of evaluation, one may again distinguish between the evaluation of off-the-shelf products and of customized systems. In the first case, the evaluation sponsor can potentially spread the cost between a larger number of customers. In the second case, the sponsor, or a single customer, may have to bear all the costs on their own.

Evaluations may be required by government procurement guidelines or be mandated by law or certain industry standards. Evaluations may also improve a product in the users' perception.

13.2 THE ORANGE BOOK

Work towards security evaluation guidelines started in the US in 1967. It led to the *Trusted Computer Security Evaluation Criteria* (Orange Book), the first guidelines for evaluating security products (operating systems). Although these efforts were concentrated in the 'national security' sector, the authors of the Orange Book wanted to create a more generally applicable document that provides:

- a yardstick for users to assess the degree of trust that can be placed in a computer security system;
- guidance for manufacturers of computer security systems;
- a basis for specifying security requirements when acquiring a computer security system.

Security evaluation examines the security-relevant part of a system, i.e. the trusted computing base (TCB; Section 6.1). The access control policies of the Orange Book are already familiar from the Bell–LaPadula model (Section 11.2), discretionary access control and mandatory access control based on a lattice of security labels. A *reference monitor* verifies that subjects are authorized to access the objects they request.

High assurance is linked to simple TCBs, structured design methodologies, formal methods, and competent and properly supported systems management. Bell–LaPadula is an obvious candidate for a formal model capturing the Orange Book security policies, but other models have also been used in Orange Book evaluations. It is assumed that greater simplicity in the TCB will allow more comprehensive analysis. Complex systems will therefore generally fall into the lower evaluation classes. The Orange Book was used for security evaluations performed by a national security organization.

The *evaluation classes* of the Orange Book address typical patterns of security requirements that existed at the time the criteria were drafted. *Specific security feature*

requirements and *assurance requirements* are combined in the definition of evaluation classes. There are four security divisions and seven security classes. The security classes are defined incrementally. All requirements of one class are automatically included in the requirements of all higher classes. Products in higher security classes provide more security mechanisms and higher assurance through more rigorous analysis. The four divisions are as follows:

D Minimal Protection

C Discretionary Protection ('need to know')
- **C1** Discretionary Security Protection
- **C2** Controlled Access Protection

B Mandatory Protection (based on 'labels')
- **B1** Labelled Security Protection
- **B2** Structured Protection
- **B3** Security Domains

A Verified Protection
- **A1** Verified Design

C2 systems make *users individually accountable for their actions*, enforcing discretionary access control at the granularity of single users. C2 was regarded as the most reasonable class for commercial applications [91] although intrinsically giving rather weak assurance guarantees. Most major vendors offered C2-evaluated versions of their operating systems or database management systems. Class C2 has played an important role in establishing a baseline for operating system security.

Division B is intended for products that enforce mandatory access control policies on classified data. Testing and documentation have to be much more thorough than for division C. An informal or formal model of the security policy is required. All flaws uncovered in testing must be removed. Class B1 is not very demanding with respect to the structure of the TCB. Hence, complex software systems such as multi-level secure Unix systems – System V/MLS (from AT&T) and operating systems from vendors like Hewlett-Packard, DEC, and Unisys – or database management systems – Trusted Oracle 7, INFORMIX-Online/Secure, and Secure SQL Server (from Sybase) – received B1 certificates.

Class B2 increases assurance requirements. A formal model of the security policy and a Descriptive Top Level Specification (DTLS) of the system are required, as is a modular system architecture. The TCB shall provide distinct address spaces to isolate processes, with support at the hardware level. A *covert channel analysis* has to be conducted and events potentially creating a covert channel have to be audited. Security testing shall establish that the TCB is *relatively resistant to penetration*. The Trusted XENIX operating system from Trusted Information Systems was rated B2.

B3 systems are *highly resistant to penetration*. Many of the new elements in class B3 have to do with security management. For higher assurance, a *convincing argument* shall establish the consistency between the formal model of the security policy and the informal DTLS. Various versions of XTS-300 (and XTS-200) from Wang Government Services had been rated B3. XTS-300 was a multi-level secure operating system (STOP) running on a Wang proprietary x86 hardware base.

Class A1 is functionally equivalent to B3. It achieves the highest assurance level through the use of formal methods. Formal specification of policy and system, together with consistency proofs, show with a high degree of assurance that the TCB is correctly implemented. Evaluation for class A1 requires:

- a formal model of the security policy;
- a Formal Top Level Specification (FTLS), including abstract definitions of the functions of the TCB;
- consistency proofs between model and FTLS (formal, where possible);
- that the TCB implementation has informally shown to be consistent with the FTLS;
- a formal analysis of covert channels (informal for timing channels) – continued existence of covert channels has to be justified and bandwidth may have to be limited.

A1 rated products include the SCOMP operating system and network components such as MLS LAN (from Boeing) and the Gemini Trusted Network Processor. When the Orange Book was written, consideration was given to defining even higher assurance classes *beyond A1*, with more requirements on system architecture, testing, formal specification and verification, and trusted design environment. Given the difficulties of evaluating complex software products even to lower assurance levels, there was little incentive to progress work in this direction.

13.3 THE RAINBOW SERIES

The Orange Book is part of a collection of documents on security requirements, security management and security evaluation published by the US National Security Agency and National Computer Security Center, originally developed for the evaluation of systems that process classified government data. The documents in this series are known by the colour of their cover and, as there are plenty of them, they became known as the rainbow series. The concepts and terminology introduced in the Orange Book were adapted to the specific aspects of database management systems and of computer networks in the *Trusted Database Management System Interpretation* (Lavender/Purple Book) [176] and in the *Trusted Network Interpretation* (Red Book) [175].

13.4 INFORMATION TECHNOLOGY SECURITY EVALUATION CRITERIA

The harmonized European Information Technology Security Evaluation Criteria [70] were the result of Dutch, English, French and German activities in defining national security evaluation criteria. A first draft was published in 1990 and the Information Technology Security Evaluation Criteria (ITSEC) were formally endorsed as a Recommendation by the Council of the European Union on 7 April 1995. As a European document, ITSEC exists in a number of translations, which adds to the difficulties of uniformly interpreting the criteria.

ITSEC is a logical progression from the lessons learned in various Orange Book interpretations. The Orange Book was found to be too rigid, and ITSEC strives to provide a framework for security evaluation that can deal with new sets of security requirements when they arise. The link between functionality and assurance is broken. The criteria apply to *security products* as well as to *security systems*. The term *Target of Evaluation (TOE)* was introduced in ITSEC. It stands for the product or system submitted for security evaluation.

The *sponsor* of the evaluation determines the operational requirements and threats. The *security objectives* for the TOE further depend on laws and other regulations. They establish the required security functionality and evaluation level. The *security target* specifies all aspects of the TOE that are relevant for evaluation. It describes the security functionality of the TOE, possibly also envisaged threats, objectives, and details of security mechanisms to be used. The security functions of a TOE may be specified individually or by reference to a predefined *functionality class*.

Seven *evaluation levels*, E0 to E6, express the level of confidence in the correctness of the implementation of security functions. E0 stands for inadequate confidence. For each evaluation level, the criteria enumerate items to be delivered by the sponsor to the evaluator. *The evaluator shall ensure that these items are provided, taking care that any requirements for content and presentation are satisfied, and that the items clearly provide, or support the production of, the evidence that is called for*. Close cooperation between the sponsor/developer and the evaluator is recommended.

European security evaluation criteria responded to the problems exposed by the Red Book, and also the Trusted Database Interpretation, by separating function and assurance requirements and considering the evaluation of entire security systems. The flexibility offered by ITSEC may sometimes be an advantage, but it also has its drawbacks. Remember the fundamental dilemma highlighted in Section 3.4.3. How can users who are not security experts decide whether a given security target is right for them?

13.5 THE FEDERAL CRITERIA

The next link in the evolutionary chain of evaluation criteria are the US Federal Criteria [177]. They took the next logical step, giving more guidance in the definition of evaluation classes but retaining some degree of flexibility. They stick to the evaluation of products and to the linkage between function and assurance in the definition of evaluation classes. They try to overcome the rigid structure of the Orange Book through the introduction of product-independent *protection profiles*. A protection profile has five sections:

- *Descriptive elements* – the 'name' of the protection profile, including a description of the information protection problem to be solved.
- *Rationale* – the fundamental justification of the protection profile, including threat, environment, and usage assumptions, a more detailed description of the information protection problem to be solved, and some guidance on the security policies that can be supported by products conforming to the profile.
- *Functional requirements* – these establish the protection boundary that must be provided by the product, such that expected threats within this boundary can be countered.
- *Development assurance requirements* – for all development phases from the initial design through to implementation, including the development process, the development environment, operational support and development evidence.
- *Evaluation assurance requirements* – specify the type and intensity of the evaluation.

13.6 THE COMMON CRITERIA

For security evaluation to be commercially attractive, evaluation results should be recognized as widely as possible. A first step in this direction is agreement on a common set of evaluation criteria. Thus, various organizations in charge of national security evaluations came together in the Common Criteria Editing Board (CCEB) and produced the Common Criteria [58] in an effort to *align* existing and emerging evaluation criteria such as TCSEC, ITSEC, CTCPEC, and the Federal Criteria. In 1999, the Common Criteria (CC) became the international standard (ISO 15048). The CCEB has been succeeded by the CC Implementation Board (CCIB).

The CC merge ideas from their various predecessors. (As an unfortunate consequence of this merger, the reader is faced with a very voluminous document.) The CC are applicable for the security evaluation of products or systems. The generic term 'Target of Evaluation' is used again. The CC abandon the strict separation of functionality classes and assurance levels adopted in ITSEC and follow the Federal Criteria in using protection profiles similar to predefined security classes. The Security Target (ST) expresses security requirements for a specific TOE, e.g. by reference to a protection profile. The ST is the

basis for any evaluation. The Evaluation Assurance Level (EAL) defines what has to be done in an evaluation.

13.6.1 Protection Profiles

To guide decision-makers, information about security objectives, rationale, threats and threat environment, and further application notes are collected in a Protection Profile (PP). This is a (reusable) set of security requirements that meet specific user needs. Figure 13.1 gives the structure of a PP. User communities should develop their own PPs to capture their typical security requirements. There exists a process for evaluating new profiles and for maintaining an official register of PPs.

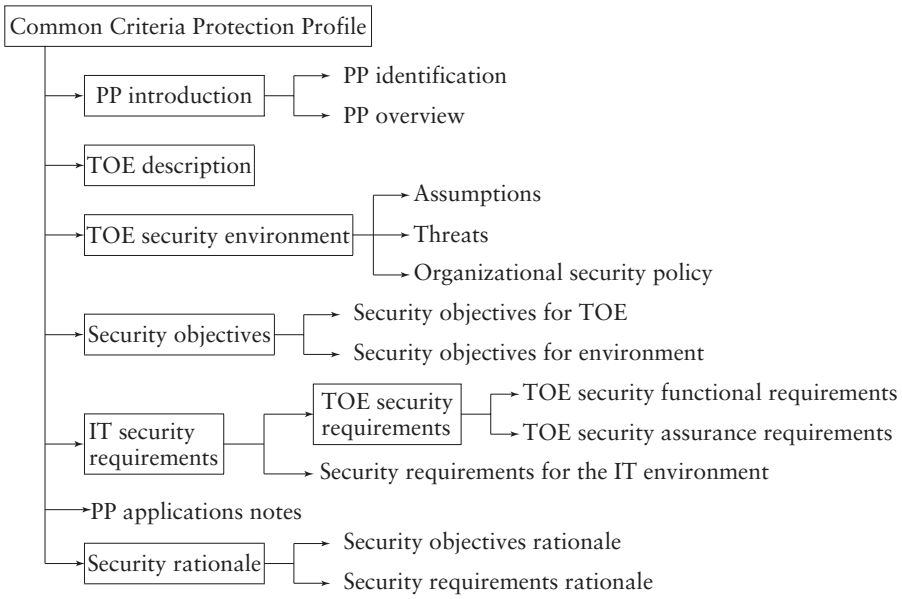


Figure 13.1: Common Criteria Protection Profile

Today, there exist PPs for a wide variety of systems. They can be generic, e.g. for an infrastructure product such as an operating system, or specific to a single application. The CC evaluation of Windows mentioned in Chapter 8 used the *Controlled Access Protection Profile version 1.d* that has its origins in the Orange Book class C2. At the other end of the spectrum, there are PPs for taxi on-board computers (in Dutch) or for electronic health cards (some in German). The scope of PPs spans single-level and multi-level operating systems, database management systems, firewalls, intrusion detection systems, trusted platform modules, biometric verification mechanisms, postage meters, automatic cash dispensers, electronic wallets, secure signature-creation devices, machine readable travel documents, and several aspects of smart card security.

13.6.2 Evaluation Assurance Levels

EALs specify the duties of the developer of a TOE and of the evaluator. There are seven incrementally defined EALs:

EAL1 – functionally tested The tester receives the TOE, examines the documentation and performs some tests to confirm the documented functionality. Evaluation should not require any assistance from the developer. The outlay for evaluation should be minimal.

EAL2 – structurally tested The developer provides test documentation and test results from a vulnerability analysis. The evaluator reviews the documentation and repeats some of these tests. The effort required from the developer is small and a complete development record need not be available.

EAL3 – methodically tested and checked The developer uses configuration management, documents security arrangements for development, and provides high-level design documentation and documentation on test coverage for review. This level is intended for developers who already follow good development practices but do not want to implement further changes to their practices.

EAL4 – methodically designed, tested, and reviewed The developer provides low-level design documentation and a subset of security functions (TCB) source code for evaluation. Secure delivery procedures have to be in place. The evaluator performs an independent vulnerability analysis. Usually EAL4 is the highest level that is economically feasible for an existing product line. Developers have to be ready to incur additional security-specific engineering costs.

EAL5 – semiformally designed and tested The developer provides a formal model of the security policy, a semiformal high-level design and functional specification as well as the full source code of the security functions. A covert channel analysis has to be conducted. The evaluator performs independent penetration testing. For evaluation at this level, it helps if the TOE has been designed and developed with the intention of achieving EAL5 assurance. The additional costs of evaluation beyond the costs of the development process itself ought not to be large.

EAL6 – semiformally verified design and tested The source code must be well structured and the access control implementation (reference monitor) must have low complexity. The evaluator has to conduct more intensive penetration testing. The cost of evaluation should be expected to increase.

EAL7 – formally verified design and tested The developer provides a formal functional specification and a high-level design. The developer has to demonstrate or prove correspondence between all representations of the security functions. The security functions must be simple enough for formal analysis. This level can typically only be achieved with a TOE that has a tightly focused security functionality and is amenable to extensive formal analysis.

13.6.3 Evaluation Methodology

The Common Evaluation Methodology (CEM) specifies all the steps that have to be followed when validating the assurance requirements in an ST [57]. The Common Criteria Recognition Agreement (CCRA) provides recognition of evaluations performed in another country. The CEM addresses assurance levels EAL1 to EAL4. Only these assurance levels are mutually recognized. Higher assurance levels are not necessarily accepted in other countries. In the US, the Common Criteria Evaluation and Validation Scheme (CCEVS) is the national programme for performing security evaluations according to the Common Criteria. There is a validation body that approves participating security testing laboratories, provides technical guidance and validates the results of security evaluations.

The CEM establishes the general evaluation framework. Evaluation methods can be specific to individual areas. For example, a security evaluation of a smart card might examine physical tamper resistance and resilience to side-channel attacks. In an evaluation of an operating system such an analysis would be out of place. Within the CC evaluation process, industry working groups may codify the state of the art in area-specific evaluation methods. This is, for example, the case in the smart card sector.

13.6.4 Re-evaluation

Certificates apply to a particular version and a particular configuration of a product. In an actual installation, it is likely that a different configuration and probably already a different version is used so that, strictly speaking, the certificate offers no direct security guarantees. Hence, there are continually attempts to develop evaluation methodologies that make it easy to re-evaluate a new version of a previously evaluated product with reduced cost and effort. The RAMP scheme in the rainbow series is one such example.

13.7 QUALITY STANDARDS

The ultimate step towards audit-based evaluation would be to assess how a product is developed without any reference to the product itself. A company then becomes a ‘certified producer of secure systems’. Such an approach has proven popular in the area of quality control. Standards like ISO 9000 advise organizations on how to put into place internal quality management and external quality assurance to vouch for the quality of their products. Some vendors claim that being registered under an ISO 9000 quality seal is a better selling argument than a security certificate for a particular product and that security evaluation should move in this direction.

The attractions such a proposal has for companies developing secure systems are evident. The costs of evaluation are much reduced. If the developers of secure systems win in this proposal, will the users of secure systems lose out? This is not a foregone conclusion. After all, a certificate is no guarantee that a system cannot be broken. Therefore, you

have to assess each evaluation scheme on its own merits to decide whether individually evaluated products offer more security than products from accredited developers.

System operators have to strike a balance between cost, productivity, and security. Any two of these factors tend to pull against the third. IT security is not a mere technological issue. Concentrating efforts on security evaluation whilst neglecting the operational management of security will not increase security. Such considerations may persuade users to look into quality standards as an alternative to security evaluation.

13.8 AN EFFORT WELL SPENT?

Security evaluation according to the CC is required in several countries by public sector customers. Major operating system and database management system vendors offer evaluated products. However, outside the government sector there has been little enthusiasm for evaluated products. Security evaluation has been criticized as an expensive government-driven process [91]. It has been noted that the CC evaluation process has a poor record of actually detecting security vulnerabilities in products.[1] You will be stretched to find many security advisories on Windows or Linux that have come out of CC evaluations.

There are exceptions, though, and in certain markets CC evaluations are pursued by most vendors. At the time of writing, the smart card sector is one example. Here, CC security evaluation has been a success story. One can come across comments from industry that investment in evaluation has been worth the effort. The following factors may have contributed to this situation.

Security is a primary application of smart cards. Their functionality is relatively fixed. It is not part of the normal business model to patch the software on cards deployed in the field. Hence, evaluation results are not out of date very quickly and there is time to amortize the cost of evaluation. Re-evaluation of a product may be cheaper than a full evaluation when the TOE is not too complex and changes are incremental. High-assurance evaluation has revealed flaws in products. There is a general market demand as smart cards are often deployed in regulated sectors. Examples are government regulations for citizen cards, and self-regulation in the credit card sector. All of this creates a virtuous cycle. Industry sees the value of the process and contributes to it, e.g. by leading the design of PPs for smart cards. In turn, this increases industry acceptance.

In contrast, the evaluation of operating systems has not been a success story. Operating systems provide an infrastructure service so security competes with other criteria. Functionality is complex and evolving and there is a wide range of user requirements.

[1] Conference report on ACSAC 2004 in the IEEE *Cipher* newsletter:
http://www.ieee-security.org/Cipher/ConfReports/2005/CR2005-ACSAC.html

Security evaluation is thus addressing a moving target. Evaluations are out of date almost by default. The code base comes from different sources and there is security-relevant interaction with other software components, e.g. with the browser. Running an evaluated operating system may on its own tell little about the security of the IT system it is part of. High-assurance evaluation is hardly feasible. Lower-assurance evaluation hardly finds flaws. In such a situation, it is not surprising when users conclude that quality assurance about vendors is more telling than the evaluation of their products.

13.9 SUMMARY

We have considerable experience in evaluating security components, and a growing list of evaluated products. Evaluation of products works best for well-defined, limited security functionalities. At the time of the Orange Book, these were provided by discretionary and mandatory access control. Today, such a situation is found in the area of smart cards. However, security problems can arise in parts of the system that do not appear security-critical at first sight, written by developers with no security expertise.

We have some experience in managing security in security-aware organizations. We have audit teams with expertise in evaluating the status of an organization. However, can end users manage their end systems?

The targets for security evaluation are moving to the application layer. In terms of technology, note that current attacks such as SQL injection, XSS, XSRF or JavaScript hijacking (Chapter 18) are exploiting flaws in application software. In terms of organization, there is the challenge of managing application security components. Applications are heavily customized. You therefore need a rapid and adaptable methodology for evaluating application software. When moving to the application layer, security is moving closer to the end user. Security evaluation then has to consider unsophisticated end users when assessing usability.

Organizations are more often interested in the accreditation of their IT infrastructure. Is it fit for purpose? Compliance with laws and regulations is of growing importance. When security requirements are application-driven and when there is so much variability in security policies, traditional security evaluation may no longer be feasible. The task at hand is that of a security consultant advising an organization on the quality of the measures in place for handling security risks.

13.10 FURTHER READING

The early history of security evaluation and of formal security modelling is told in [158]. Practical aspects of security evaluation with an overview of the Orange Book classes are covered in [65]. The developments in IT that led to

the Orange Book and beyond are narrated in [203]. A brief description of the A1 evaluated Blacker system can be found in [232]. A prototype that was developed to meet A1 requirements but did not become a commercial product is described in [137]. For a case study on assurance and covert channel aspects of multi-level secure systems, see [135].

The Canadian Trusted Computer Product Evaluation Criteria have the reputation of being the most concise and readable of the evaluation criteria. Websites containing evaluation criteria, ancillary documents and lists of evaluated products are:

- `http://www.radium.ncsc.mil/tpep/library/rainbow/` for the rainbow series;
- `http://csrc.nist.gov/cc` and `http://www.commoncriteriaportal.org/` for the Common Criteria.

13.11 EXERCISES

Exercise 13.1 Security evaluation has to deal with moving targets. Product development does not stand still while one particular version is being evaluated. How could evaluation certificates be kept up to date? Consult schemes such as RAMP when drafting your proposal.

Exercise 13.2 Security products have to hit moving targets. The threat environment will change during the lifetime of a fielded product. How would you set up a scheme for the evaluation of anti-virus products that keeps certificates up to date in a changing threat environment? Are there any components of your scheme that should be included in the evaluation of operating systems?

Exercise 13.3 It is sometimes claimed that evaluated products are mainly used as an insurance against the accusation of not following established best practice, and not because they offer better security. What do you expect from a security evaluation scheme that does provide added value?

Exercise 13.4 Evaluation criteria exist to help security-unaware users meet specific security requirements. Are protection profiles the right solution for this problem?

Exercise 13.5 ITSEC covers the security evaluation of systems. Consultants advise clients on solutions to their security problems. Where would you draw

the boundary between consultancy and evaluation? Does evaluation have any advantage over contracting consultants?

Exercise 13.6 Write a protection profile for web browsers.

Exercise 13.7 Examine the options for blocking and monitoring covert channels. How is the usability of a system affected by blocking covert channels?

Chapter 14

Cryptography

Once upon a time – at the high tide of research on multi-level security in the 1980s – you could hear claims that there was nothing cryptography had to offer computer security. Computer security was about TCBs, reference monitors, discretionary and mandatory access control, and formal verification of security models and system specifications. In this view, the contributions of cryptography appeared to be peripheral indeed. One-way functions to store passwords were the only obvious instance of a cryptographic mechanism used in secure operating systems.

By the mid 1990s, the mood had swung round to the other extreme. Cryptography was seen as the miraculous cure that would solve all computer security problems. Secure operating systems were dismissed as a thing of the past, too expensive, too restrictive, too far away from user demand, doomed to extinction like the dinosaurs. Export restrictions on strong cryptographic algorithms were regarded as the main obstacle that needed to be overcome to make computers secure. A further decade on, a more sober assessment had been reached of the contributions cryptography can offer for computer security. Cryptographic techniques remain an essential component in securing distributed systems.

OBJECTIVES

- Appreciate the variety of applications that use cryptography with quite different intentions.

- **Introduce the basic concepts of cryptography.**
- **Understand the type of problems cryptography can address, and the types of problem that need to be addressed when using cryptography.**
- **Indicate the computer security features that are required to support cryptography.**

14.1 INTRODUCTION

In its traditional definition cryptography is the science of secret writing. Cryptanalysis is the science of analyzing and breaking ciphers. Cryptology encompasses both subjects. Once they were the domain of spies and secret agents. These origins still endow cryptography with a certain mystique.

Modern cryptography is very much a mathematical discipline. It is outside the scope of this book to present the mathematical background necessary to understand the finer points of cryptography. Instead, we will explain how cryptography can be used in computer security and point out that very often computer security is a prerequisite to make cryptography work.

14.1.1 The Old Paradigm

Cryptography has its roots in communications security. Communications security addresses the situation depicted in Figure 14.1. Two parties *A* and *B* communicate over an insecure channel. The antagonist is an *intruder* who has full control over this channel, being able to read their messages, delete messages, and insert messages. The two parties *A* and *B* trust each other. They want protection from the intruder. Cryptography gives them the means to construct a secure logical channel over an insecure physical connection. In this respect, cryptography is fundamentally different from the computer security mechanisms discussed so far. All of them are vulnerable to compromise from the 'layer below'. However, access to the physical communications link does not compromise cryptographic protection.

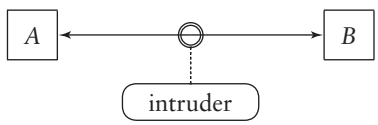


Figure 14.1: Communications Security

In distributed systems, the traffic between clients and servers is a point of attack for would-be intruders. Vulnerabilities introduced by insecure communications links can

naturally be counteracted by services and mechanisms from communications security. Such services include:

- data confidentiality – encryption algorithms hide the content of messages;
- data integrity – integrity check functions provide the means to detect whether a message has been changed;
- data origin authentication – message authentication codes or digital signature algorithms provide the means to verify the source and integrity of a message.

Data origin authentication includes data integrity. A message that has been modified in transit no longer comes from its original source. Conversely, if the sender's address is part of the message, you also have to verify the source of a message when verifying its integrity. In such a setting, data integrity and data origin authentication are equivalent concepts. A separate notion of data integrity makes sense in other applications, e.g. for file protection in anti-virus software.

The traditional view of friend and foe has its place in computer security, but it is no longer the major force driving applications of cryptography in computing. Unfortunately, it still dominates the public perception of cryptography. This view is also reflected in those verification tools for cryptographic protocols, whose axioms assume that *A* and *B* will behave according to the rules of the protocol and only consider the effects of the intruder's actions.

14.1.2 New Paradigms

Let us take a fresh look. In electronic commerce, a customer enters into a business transaction with a merchant. Neither party expects the other to cheat, but disputes are possible and it is always better to have rules agreed in advance than to solve problems in an ad hoc fashion. Customer and merchant therefore have reasons to run a protocol that does not assume that the other party can be trusted in all circumstances. The antagonist is now a misbehaving *insider*, rather than an intruder. The third party in Figure 14.2 is no longer the intruder but a *trusted third party* (TTP), e.g. an arbitrator. *Non-repudiation* services generate the evidence the arbitrator will consider when resolving a dispute.

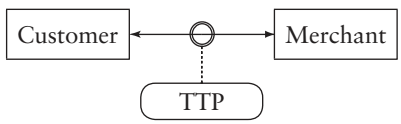


Figure 14.2: Electronic Commerce Security

Many countries have laws specifying when and how a *law enforcement agency* (LEA) can get an interception warrant that obliges a telecommunications service provider to

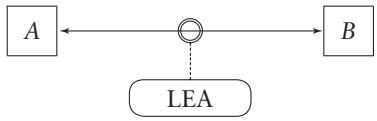


Figure 14.3: Communications Security and Law Enforcement

give access to communications between particular users. The third party in Figure 14.3 is now a client of the telecommunications operator who has to be provided with a *legal intercept* service. In this context, *key escrow* services that reveal the key used to encrypt the traffic were once a topic discussed with great passion.

14.1.3 Cryptographic Keys

Cryptography has adopted the lock as its favourite icon to signal the services it renders to the public. A quick look at the user interfaces of 'security enabled' web browsers or email products will confirm this observation. Analogies are fraught with danger, and you should not take them too far, but there are some important concepts that carry over from locksmiths to cryptographers. To lock and unlock a door, you need a key. Locks differ in strength. Some are easy to pick while others are so strong that intruders will resort to brute force attacks to break through a door or choose a different path altogether and break into a house through a window instead.

Cryptographic algorithms use keys to protect data. There are again variations in strength, ranging from schemes that can be broken with simple statistical methods to those that are far beyond the current grasp of mathematical analysis and computational abilities. *Brute force attacks* exhaustively search the entire key space and give an upper bound for the strength of an algorithm.

Modern cryptography does not rely on the secrecy of its algorithms. The key used in a cryptographic transformation should be the only item that needs protection. This principle was postulated by Kerckhoffs in the nineteenth century. It is particularly appropriate in the setting of our new security paradigms where a large user community with competing interests has to be supported. *De facto* standardization and open evaluation of public algorithms is a natural process in such a situation, giving each party the chance to conduct its own security assessment and making it easier for new participants to join in.

Key management, in the most general meaning of the word, is thus of paramount importance for the security of cryptographic schemes. You have to address questions such as the following:

- Where are keys generated?
- How are keys generated?

- Where are keys stored?
- How do they get there?
- Where are the keys actually used?
- How are keys revoked and replaced?

At this point, the circle closes and we return to computer security. Cryptographic keys are sensitive data stored in a computer system. Access control mechanisms in the computer system have to protect these keys. When access control fails, cryptographic protection is compromised. In most security systems currently fielded, the cryptographic algorithms are the strongest part and wily attackers will look for other vulnerabilities rather than wasting their time on cryptanalysis.

Lesson
Cryptography is rarely, if ever, the solution to a security problem. Cryptography is a translation mechanism, usually converting a communications security problem into a key management problem and ultimately into a computer security problem. Hopefully, the resulting problem is easier to solve than the original problem. In summary, cryptography can enhance computer security, but it is not a substitute for computer security.

14.1.4 Cryptography in Computer Security

If you have data that need to remain secret, a *vault* for locking away those secrets would come in useful. The vault has to be unlocked with a key when putting data in or taking data out. Such a vault is implemented by *symmetric encryption* mechanisms.

There is also the *transparent vault* you might see in a public lottery draw. Everyone can see what is in the vault. Only an authorized person may fill the vault, needing a *private key* to do so. If the vault has a unique serial number, everyone can refer to documents in the vault by this serial number. In the parlance of cryptography this is a *public key*. Such a transparent vault can be used for creating protected name spaces. In this case, the public key is like a database key for organizing and addressing documents.

You might want to have a *private letter box*. Anybody can drop documents into the letter box. Only the owner can open it. The owner needs a *private key* for taking documents out. The letter box needs a serial number so that you can distinguish between letter boxes. This and the previous feature can be implemented using *public-key cryptography*.

When a document leaves your control, you might save a *fingerprint* so that you could detect any eventual later changes. This strategy is used, for example, by anti-virus products. The fingerprint of a program is computed in a clean environment and stored

in a place where it cannot be modified, e.g. on a CD-ROM. To check the status of the program, the fingerprint is recomputed and compared with the value stored. If an attacker is able to change the program without changing the fingerprint, the change would go undetected and the scheme would be broken. Having two documents with the same fingerprint is called a *collision*. *Collision-resistant hash functions* implement fingerprints that can be used for integrity protection. Protection of the fingerprints themselves is also important. Fingerprint computation does not require any secret information. Hence, anybody can create a valid fingerprint for a given document.

A system authenticating users by password may store only *fingerprints* (*hashes*) of the passwords (Section 4.5). Reconstructing passwords from their fingerprints must not be feasible. Otherwise, little would have been gained for security. In this case, the function for computing fingerprints must also be a *one-way function*.

14.2 MODULAR ARITHMETIC

Many modern cryptographic algorithms build on algebraic principles. They can be defined on exciting algebraic structures such as elliptic curves or Galois fields. We will stay more down to earth and only use integers in our description.

Let m be an integer. In the following, we will call m the modulus. The 'mod m' equivalence relation on the set of integers is defined by

$$a = b \bmod m \text{ if and only if } a - b = \lambda \cdot m \text{ for some integer } \lambda.$$

We say 'a is equivalent to b modulo m'. You can check that mod m is indeed an equivalence relation that divides the set of integers into m equivalence classes

$$(a)_m = \{b \mid a = b \bmod m\}, \quad 0 \leq a < m.$$

It is more customary to designate the equivalence class by a mod m and we will follow this convention. You can verify the following useful properties:

- $(a \bmod m) + (b \bmod m) = (a + b) \bmod m$;
- $(a \bmod m) \cdot (b \bmod m) = (a \cdot b) \bmod m$;
- for every $a \neq 0 \bmod p$, p prime, there exists an integer a^{-1} so that $a \cdot a^{-1} = 1 \bmod p$.

For a prime modulus p, the multiplicative order modulo p is defined as follows:

> Let p be a prime and a an arbitrary integer. The *multiplicative order* of a modulo p is the smallest positive integer n such that $a^n = 1 \bmod p$.

Fermat's little theorem states that the multiplicative order modulo p of any non-zero element must be a factor of $p - 1$.

> **Theorem.** For every $a \neq 0 \bmod p$, p prime, we have $a^{p-1} = 1 \bmod p$.

This fact is used in the construction of quite a few cryptographic algorithms. The security of these algorithms is often related, and on a few occasions equivalent, to the difficulty of one of the following problems from number theory:

- Discrete logarithm problem (DLP). Given a prime modulus p, a basis a, and a value y, find the *discrete logarithm* of y, i.e. an integer x such that $y = a^x \bmod p$.
- nth root problem. Given integers m, n and a, find an integer b such that $a = b^n \bmod m$. The solution b is the nth root of a modulo m.
- Factorization. Given an integer n, find its prime factors.

With the right choice of parameters, these problems are a suitable basis for many cryptographic algorithms. However, not all instances of these problems are difficult to solve. Obviously, if p or n are small integers these problems can be solved by exhaustive search within reasonable time. At the time of writing, 1024-bit integers are barely regarded as long enough, 2048-bit integers are a more common recommendation, and you can use longer integers if you can tolerate the decrease in performance as arithmetic operations will take longer. Length is not the only aspect you have to consider. The difficulty of these problems also depends on the structure of p and n. (To pursue this topic further, you will need to turn to more specialized mathematical sources.)

14.3 INTEGRITY CHECK FUNCTIONS

A cryptographic hash function h maps inputs x of arbitrary bit length to outputs $h(x)$ of a fixed bit length n. This is known as the *compression* property. Hash functions tend to be faster and less resource-consuming than the other cryptographic mechanisms mentioned; the *ease of computation* property demands that, given x, it is easy to compute $h(x)$.

14.3.1 Collisions and the Birthday Paradox

A *collision* exists if there are two inputs x, x', $x \neq x'$, with $h(x) = h(x')$. The probability of finding a collision by brute force search depends on the bit length of the hash. For an n-bit hash y, the expected number of tries before an x with $h(x) = y$ is found is 2^{n-1}. If you are just looking for arbitrary collisions, a set of about $2^{n/2}$ inputs is likely to contain a pair causing a collision. This result is based on the *birthday paradox*. Put m balls numbered 1 to m into an urn, draw a ball, list its number, and put it back. Repeat this experiment. For $m \to \infty$, the expected number of draws before a previously drawn number appears converges to $\sqrt{\frac{1}{2}m\pi}$.

14.3.2 Manipulation Detection Codes

Manipulation detection codes (MDCs), also called *modification detection codes* or *message integrity codes*, detect changes to a document. Depending on the given application,

the requirements put on MDCs may differ. The security properties one might demand from a hash functions h include:

- *Pre-image resistance (one-way)* – given a value y, it is computationally infeasible to find a value x such that $h(x) = y$.
- *Second pre-image resistance (weak collision resistance)* – given an input x and $h(x)$, it is computationally infeasible to find another input x', $x \neq x'$, with $h(x) = h(x')$.
- *Collision resistance (strong collision resistance)* – it is computationally infeasible to find any two inputs x and x', $x \neq x'$, with $h(x) = h(x')$.

MDCs come in two flavours [168]:

- a *one-way hash function* (OWHF), with the compression, ease-of-computation, pre-image resistance, and second pre-image resistance properties;
- a *collision-resistant hash function* (CRHF), with the compression, ease-of-computation, second pre-image resistance, and collision resistance properties.

The result of applying a hash function is varyingly called a

- *hash value*,
- *message digest*,
- *checksum*.

The last term leaves ample room for confusion. In communications theory, checksums refer to error correcting codes, typically a cyclic redundancy check (CRC). A CRC is a linear function, so creating collisions is easy. Checksums in security, on the other hand, must not be computed with a CRC but with a cryptographic hash function (MDC).

The *discrete exponentiation* function $f(x) := g^x \bmod p$ is a one-way function when the parameters p and g are chosen judiciously. To invert discrete exponentiation, you have to solve the discrete logarithm problem introduced in Section 14.2. Discrete exponentiation is a useful primitive in the construction of cryptographic schemes, as you will see later in this chapter. However, discrete exponentiation is not a particularly fast operation, so you have to turn to other algorithms when processing large quantities of data at high speed.

Fast hash functions tend to be constructed along similar design patterns. At the core of the hash function is a *compression function* f that works on inputs of fixed length. An input x of arbitrary length is broken up into blocks $x_1, \ldots, x_m$ of a given block size, with padding added to the last block. The hash of x is then obtained by repeated application of the compression function. Let h_0 be a (fixed) *initial value*. Compute

$$h_i = f(x_i || h_{i-1}), \quad \text{for } i = 1, \ldots, m$$

(where the symbol || denotes concatenation), and take h_m as the hash value of x (Figure 14.4).

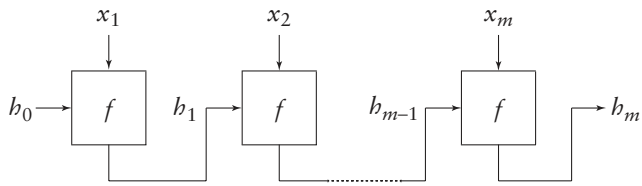


Figure 14.4: Construction of a Hash Function

14.3.3 Message Authentication Codes

Message authentication codes provide assurance about the source and integrity of a message (data origin authentication). A message authentication code is computed from two inputs, the message and a secret cryptographic key. Therefore, message authentication codes are sometimes called keyed hash functions. Formally, a message authentication code is a family of functions h_k parametrized by the secret key k. Each member of the family has the compression and the ease-of-computation property. An additional *computation resistance* property must hold:

> For any fixed value of k unknown to the adversary, given a set of values $(x_i, h_k(x_i))$, it is computationally infeasible to compute $h_k(x)$ for any new input x.

To authenticate a message, the receiver has to share the secret key used to compute the message authentication code with the sender. A third party that does not know the key cannot validate the message authentication code. A message authentication code algorithm can be derived from a MDC algorithm h using the following HMAC construction [26, 140]. For a given key k and message x, compute

$$HMAC(x) = h(k||p_1||h(k||p_2||x))$$

where p_1 and p_2 are bit strings (padding) that extend k to a full block length of the compression function used in h.

14.3.4 Cryptographic Hash Functions

Although we have been at pains to elaborate the different properties that may be required of a hash function, it is in practice assumed that a strong cryptographic hash function meets all requirements, and that a hash function that has weaknesses in one respect should be treated with suspicion.

Once MD4 and MD5 were popular hash functions. MD5 was the standard choice in Internet protocols. It was then shown to be computationally feasible to find meaningful collisions; see [83] on MD4. Once one collision has been found, it may be possible to use this collision to create multiple collisions for other document types.[1] MD4 and MD5 are no longer recommended.

[1] At the rump session of Eurocrypt 2005, M. Daum and S. Lucks presented a generic attack for constructing pairs of postscript documents that have the same MD5 hash.

The Secure Hash Algorithm (SHA-1) designed to operate with the US Digital Signature Standard (DSA) processes 512-bit blocks and generates a 160-bit hash value. Collision attacks on SHA-1 well below the brute-force bound of 2^{80} operations have been reported [230]. RIPEMD-160 is a hash function frequently used by European cryptographic service providers, designed along the same principles as SHA-1. These hash functions are still in use but the development of new hash functions is under way, triggered by the attack on SHA-1 but also by the fear that 160-bit hash values may prove too short to resist brute force attacks in the not so distant future. SHA-256 is the current choice when longer hash values are desired.

14.4 DIGITAL SIGNATURES

In Figure 14.1, message authentication codes help parties *A* and *B* detect fraudulent messages inserted by the intruder on the communications channel. However, message authentication codes do not constitute evidence a third party could use to decide whether *A* or *B* sent a particular message. They are therefore of little use in the electronic commerce scenario of Figure 14.2 when customers need assurance that merchants cannot fake orders and merchants need assurance that customers will honour the orders they have made. These situations call for a *digital signature*.

A digital signature scheme consists of a key generation algorithm, a signing algorithm, and a verification algorithm. A digital signature of a document is a value depending on the contents of the document and on some secret only known to the signer, i.e. a private signature key. The signature associates the document with a public verification key. The verification algorithm usually takes the document and the public verification key as input. In exceptional cases the document – or parts of the document – can be recovered from the signature and the document does not have to be provided for signature verification. Figure 14.5 gives a schematic representation of a typical digital signature scheme where the private signature key is applied to a hash of the document. The following *verifiability* property characterizes digital signature schemes:

> A third party can resolve disputes about the validity of a digital signature without having to know the signer's private key.

Digital signatures support non-repudiation. Public-key cryptography (Section 14.5) is a natural source for digital signature schemes. In such a scheme, the connection between the private signature key and the public verification key has the property that it is computationally infeasible to derive the signature key from the verification key. Despite the similarities in the underlying mathematical techniques, you should draw a clear distinction between digital signatures and public-key encryption algorithms. These two schemes meet fundamentally different purposes. Encryption protects the confidentiality of a message and has to be invertible. Digital signatures provide data origin authentication and non-repudiation. A digital signature algorithm need not be invertible. In fact, invertibility causes some additional security concerns.

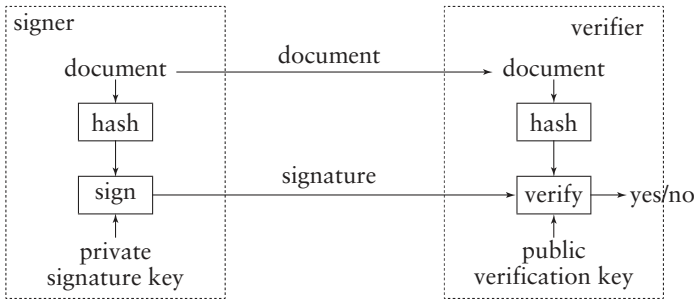


Figure 14.5: Schematic Representation of a Generic Digital Signature Scheme

14.4.1 One-Time Signatures

You do not need fancy mathematics to construct a signature scheme. To obtain *one-time signatures* you only need a cryptographic hash function h [142]. To sign an n-bit document, you pick your private key by choosing at random $2n$ values x_i^0, x_i^1 and publish the values (commitments) $y_i^0 = h(x_i^0), y_i^1 = h(x_i^1)$, $1 \leq i \leq n$, as your public key. The ith bit of the signature s for a document m is then given by

$$s_i = \begin{cases} x_i^0 & \text{if } m_i = 0, \\ x_i^1 & \text{if } m_i = 1. \end{cases}$$

Evidently, you cannot use your private key again, hence the name one-time signatures. The verifier has your public key and checks

$$\begin{aligned} y_i^0 &= h(s_i) \quad \text{if } m_i = 0, \\ y_i^1 &= h(s_i) \quad \text{if } m_i = 1. \end{aligned}$$

You will have spotted that the verifier needs additional evidence to confirm that the values y_i^0, y_i^1 are indeed your public key. We will return to this topic in Section 15.5.1.

Instead of relying on the difficulty of a mathematical problem or on the strength of some other cryptographic primitive, you could rely on the difficulty of compromising a *tamper-resistant* hardware device. The device contains a secret signature key and/or *secret* verification keys. The device is constructed so that it cannot use the signature key for verification or a verification key for signing. To sign a document, the device uses its signature key to construct a message authentication code, which it then attaches to the document. To verify this signature, the verifier's device has to hold the signer's signature key as a verification key and uses this key to construct a message authentication code and compare it with the signature received.

14.4.2 ElGamal Signatures and DSA

The ElGamal signature scheme [86] shows that signing is not 'encryption with a private key'. Let p be a large and appropriately chosen prime number. Let g be an integer of a

large prime order q modulo p. Note that q must be a factor of $p-1$. Let a be the private signature key of user A and $y = g^a \bmod p$ the corresponding public verification key.

Assume that the document to be signed is an integer m, $0 \leq m < p$. Otherwise, you can apply a suitable hash function and sign the digest of the document. To sign m, user A picks a random number k, $0 \leq k < p$, such that $\gcd(k, p-1) = 1$, computes $r = g^k \bmod p$, and solves the equation

$$a \cdot r + k \cdot s = m \bmod q$$

in the unknown s. The pair (r, s) is A's signature on m. The verifier needs A's verification key y and checks

$$y^r \cdot r^s \stackrel{?}{=} g^m \bmod p.$$

For a correct signature, the equation

$$y^r \cdot r^s = g^{ar+ks} = g^m \bmod p$$

holds. The security of this scheme is closely related but not equivalent to the discrete logarithm problem. A number of more secure and more efficient signature schemes have been derived from the ElGamal signature scheme. One such signature algorithm is the Digital Signature Algorithm (DSA) [222]. In this scheme, the private and public key of a user A are generated as follows.

1. Select a prime q such that $2^{159} < q < 2^{160}$.
2. Choose an integer t, $0 \leq t \leq 8$, and a prime p, $2^{511+64t} < p < 2^{512+64t}$, such that q divides $p-1$.
3. Select α, $1 < \alpha < p-1$, and compute $g = \alpha^{(p-1)/q} \bmod p$. If $g = 1$, try again with a new α. (This step computes a generator g of order q modulo p.)
4. Select a such that $1 \leq a \leq q-1$.
5. Compute $y = g^a \bmod p$.
6. A's public key is (p, q, g, y); the private key is the value a.

To sign a document m, A computes the hash value $h(m)$ with SHA-1 and converts $h(m)$ into an integer. Then A

1. randomly selects an integer k, $1 \leq k \leq q-1$,
2. computes $r = (g^k \bmod p) \bmod q$,
3. computes $k^{-1} \bmod q$,
4. computes $s = k^{-1}(h(m) + ar) \bmod q$.

A's signature on m is the pair (r, s). The signature is checked with A's public key (p, q, g, y) by

1. verifying $1 \leq r \leq q$ and $1 \leq s \leq q$,
2. computing $w = s^{-1} \bmod q$,
3. computing $u_1 = w \cdot h(m) \bmod q$ and $u_1 = r \cdot w \bmod q$,
4. computing $v = (g^{u_1} y^{u_2} \bmod p) \bmod q$,

and accepted if and only if $v = r$. The ECDSA algorithm is similar to DSA but works with points of an elliptic curve instead of integers modulo the prime p. ECDSA has been specified in the standard ANSI X9.62.

14.4.3 RSA Signatures

The RSA algorithm [194], named after its inventors Rivest, Shamir, and Adleman, can be equally used for signing and for encryption. This very specific property of RSA can be blamed for many of the prevailing misconceptions about digital signatures and public-key cryptography. In the RSA signature scheme, a user A picks two prime numbers p and q, and a private signature key e with $\gcd(e, p-1) = 1$ and $\gcd(e, q-1) = 1$. The public verification key consists of the product $n = p \cdot q$ and an exponent d with

$$e \cdot d = 1 \bmod \operatorname{lcm}(p-1, q-1).$$

The document to be signed is an integer m. The document is hashed with a suitable hash function h so that $1 \leq h(m) < n$. To sign m, A forms the signature

$$s = h(m)^e \bmod n.$$

The verifier needs A's verification key (n, d) and checks

$$s^d \stackrel{?}{=} h(m) \bmod n.$$

For a correct signature, this equation holds because of

$$s^d = h(m)^{e \cdot d} = h(m) \bmod n.$$

The security of RSA is closely related but not equivalent to the difficulty of factoring. To forge a signature on a given document m, the attacker has to compute the dth root of $h(m)$ modulo n. The *weak RSA assumption* captures the assumption that this is not feasible. An attacker might also take an arbitrary value $\tilde{s}$ and look for a document $\tilde{m}$ such that $\tilde{s}^d = h(\tilde{m}) \bmod n$. This would, however, violate the one-way property of the hash function. From the verifier's perspective, the hash of the document is evidence that can be used for identifying genuine documents.

The security of any given implementation of RSA depends on further factors. For example, the high-level description given above does not explain how $h(m)$ is encoded as an integer modulo n. Bad choices of the encoding function can introduce vulnerabilities. Padding a 160-bit SHA-1 hash with leading zeros to get a 1024-bit integer would be a bad choice. The currently recommended way of implementing RSA is known as RSA-PSS (probabilistic signature scheme).

14.5 ENCRYPTION

We reserve the term *encryption* for algorithms that protect the confidentiality of data. An encryption algorithm, also called a *cipher*, enciphers *plaintext (cleartext)* under the control of a cryptographic key. We write $eK(X)$ to denote that plaintext X is encrypted under key K. *Decryption* with the appropriate decryption key retrieves the plaintext from the ciphertext. We write $dK(X)$ to denote that ciphertext X is decrypted under key K. A *deterministic* encryption algorithm always maps a plaintext to the same ciphertext for a fixed key. A *probabilistic* encryption algorithm gives different results for different encryptions of the same plaintext under the same key.

Some encryption algorithms provide the means for detecting integrity violations, but this is not always the case. You may even see signature algorithms described as 'encryption with a private key', but this is often wrong and always misleading.

Encryption algorithms come in two flavours. In *symmetric algorithms*, the same key is used for encryption and decryption. This key has to be kept secret. All parties sharing the same key can read data encrypted under that key. To set up private communications with different parties, you need a new key for each party. Maintaining a large number of shared secret keys can become a quite onerous management task.

In *asymmetric encryption algorithms*, also called *public-key algorithms*, different keys are used for encryption and decryption. The encryption key can be made public, while the decryption key has to remain private. Obviously, the two keys are algorithmically related, but it should not be feasible to derive the private key from its public counterpart. To differentiate between symmetric and public-key cryptosystems, we will use *secret key* only in the context of symmetric systems and *private key* only in the context of asymmetric systems.

With secret key cryptosystems, the security management task of getting the right key into the right place is evident. Public-key cryptography seems to make management a lot easier. After all, public keys are public and need no protection. Or do they? When you use public-key encryption to encipher a document, you probably will need to know who will be able to read the enciphered document. More generally, private keys authenticate a principal or serve as capabilities carrying access rights, e.g. to read a document. Now, it becomes a major task to guarantee the link between a public key and the access rights or principals associated with the corresponding private key. We will return to this topic in Section 15.5.1.

Encryption algorithms can also be divided into *block ciphers* and *stream ciphers*. There exist two criteria for making this distinction.

- Block size: A block cipher encrypts larger blocks of data, typically 64-bit blocks, with a complex encryption function. Security of block ciphers depends on the design of the

encryption function. A stream cipher encrypts smaller blocks of data, typically bits or bytes, with a simple encryption function, e.g. bitwise exclusive-OR. As you can imagine, this distinction becomes blurred at the edges. Is a 16-bit block still a large block? When is an encryption algorithm simple?

- Key stream: A block cipher encrypts blocks belonging to the same document all under the same key. A stream cipher encrypts under a constantly changing key stream. Security of stream ciphers relies on the design of the key stream generator. With this definition, DES in a feedback mode (see below) is classified as a stream cipher.

14.5.1 Data Encryption Standard

The Data Encryption Standard (DES) is an important milestone in the history of computer security and cryptography. DES was developed in the 1970s as a US government standard for protecting non-classified data and was published as a Federal Information Processing Standard [221]. DES encrypts 64-bit plaintext blocks under the control of 56-bit keys. Each key is extended by a parity byte to give a 64-bit working key. Like many block cipher algorithms, DES is based on the Feistel principle. Feistel ciphers iterate the same basic step in a number of rounds. The input to round i is divided into two halves, L_i and R_i, and the output is computed as

$$L_{i+1} = R_i$$
$$R_{i+1} = L_i \oplus F(K_i, R_i)$$

where F is some non-linear function and K_i is the subkey for that round (Figure 14.6). The inverse of this operation can be computed by the same circuit,

$$R_i = L_{i+1}$$
$$L_i = R_{i+1} \oplus F(K_i, L_{i+1})$$

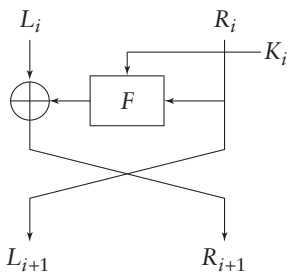


Figure 14.6: The Feistel Principle

The non-linear function F in DES expands the 32-bit input R_i into a 48-bit block and computes the bitwise exclusive-OR with the 48-bit subkey K_i. This intermediate result is divided into eight 6-bit blocks, which serve as input to eight *substitution boxes* (*S-boxes*).

Each S-box converts its 6-bit input into a 4-bit output. The output of the S-boxes is put through a *permutation box* (*P-box*) that performs a bit permutation on its 32-bit input to give the result of *F*.

DES has 16 such *rounds*. Each round uses a different 48-bit subkey K_i, derived from the 56-bit DES key. The input to the first round is processed by an *initial permutation IP*, the output of the last round by the inverse permutation IP^{-1}. We omit further details of the key scheduling algorithm, the expansion scheme, the S-boxes and the permutations.

When DES became a standard in the 1970s, it was given a 'shelf life' of 15 years. DES has aged remarkably well and is still in use, particularly in the commercial and financial sector, despite having been superseded by the Advanced Encryption Standard (AES) [223]. The major challenge to the security of DES did not come from new cryptanalytic techniques but from its key size. Today, exhaustive search through a 56-bit key space is feasible without dedicated equipment. The performance of workstations has kept increasing over the years and networking lets the amateur cryptanalyst harness even more resources. With such opposition, you would expect a 56-bit key to survive for days or weeks, rather than for decades or centuries. Multiple encryption extends the key size without changing the algorithm. The favoured option is *triple DES* using three 56-bit keys. A variation popular because of its backward compatibility with single DES uses two 56-bit DES keys, K_1 and K_2, to encipher a plaintext P by $C = eK_1(dK_2(eK_1(P)))$.

The Rijndael algorithm [75] was adopted as the AES, a US Federal Standard, in 2001. The algorithm can be used with key sizes of 128 bits, 192 bits, and 256 bits. AES works with 128-bit data blocks. Rijndael is specified to work also with 192-bit and 256-bit data blocks, where the block length can be chosen independently of the key length.

14.5.2 Block Cipher Modes

Block ciphers can be used in a variety of encryption modes. In *electronic code book* (ECB) *mode*, each plaintext block is enciphered independently under the same key. This mode may leak information about the plaintext. If a plaintext block is repeated, this will show up in the ciphertext. Furthermore, there is very limited integrity protection. Decryption will not detect whether the sequence of ciphertext blocks has been changed, whether some blocks are missing, or whether blocks have been duplicated.

In *cipherblock chaining* (CBC) *mode* (Figure 14.7), the previous ciphertext block C_{i-1} is added (bitwise exclusive-OR) to the next plaintext block P_i before encryption, i.e.

$$C_i = eK(P_i \oplus C_{i-1}).$$

Hence, repeated plaintext blocks will not show up as repeated ciphertext blocks. For the first plaintext block P_1, an initialization vector is used as C_0. The initialization vector is usually kept secret, although in many applications this would not be a necessary security

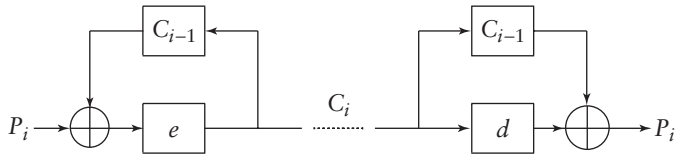


Figure 14.7: Cipherblock Chaining Mode

condition. The initialization vector should be changed for every message to make sure that an observer cannot detect that two plaintexts start with the same blocks. The initialization vector is required for decryption of the first ciphertext block. Ciphertext block C_i is decrypted by

$$P_i = C_{i-1} \oplus dK(C_i).$$

If a ciphertext block is corrupted, only two plaintext blocks are affected. Assume that $\tilde{C}_i$ is used instead of C_i. After

$$\tilde{P}_i = C_{i-1} \oplus dK(\tilde{C}_i)$$
$$P_{i+1} = \tilde{C}_i \oplus dK(C_{i+1})$$

normal decryption resumes.

Output feedback mode (Figure 14.8) uses the block cipher as the key stream generator for a stream cipher. In this mode, the plaintext can be processed in chunks that are smaller than the block size of the cipher algorithm. A register stores the input to the encryption function. The initial contents of this register are defined by an initialization vector. To encrypt a plaintext chunk, add it to a sub-block from the output of the encryption function (bitwise exclusive-OR). The output of the encryption function is fed back to the shift register. Decryption is exactly the same process as encryption. The initialization vector must change for every message but need not be kept secret. An error in the transmission of a ciphertext block C_i only affects the corresponding plaintext

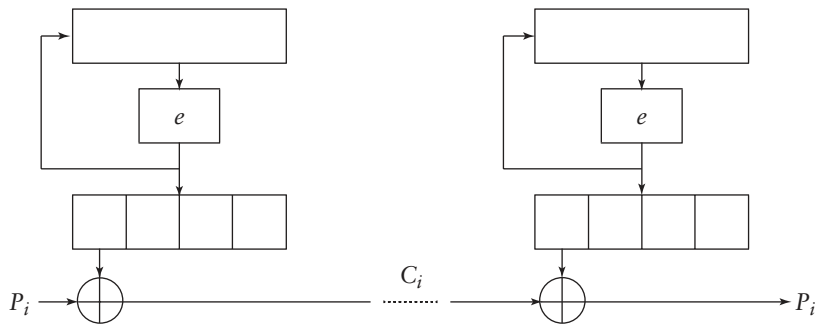


Figure 14.8: Output Feedback Mode

block P_i. An attacker can therefore selectively modify plaintext bits by changing the ciphertext in the corresponding positions.

Cipher feedback (CFB) *mode* (Figure 14.9) uses the block cipher to generate a data-dependent key stream. Again, the plaintext can be processed in chunks that are smaller than the block size of the cipher algorithm. In this mode, previous ciphertext blocks are fed back into a shift register. The contents of the register are encrypted and a sub-block of this ciphertext is added (bitwise exclusive-OR) to the next plaintext chunk. Decryption is exactly the same process as encryption. In this mode, the initialization vector must change for every message. The initialization vector is required for decryption of the first ciphertext block and need not be kept secret. A transmission error, or modification, of a ciphertext block affects decryption until the modified block has left the shift register feeding the encryption function at the receiver's end.

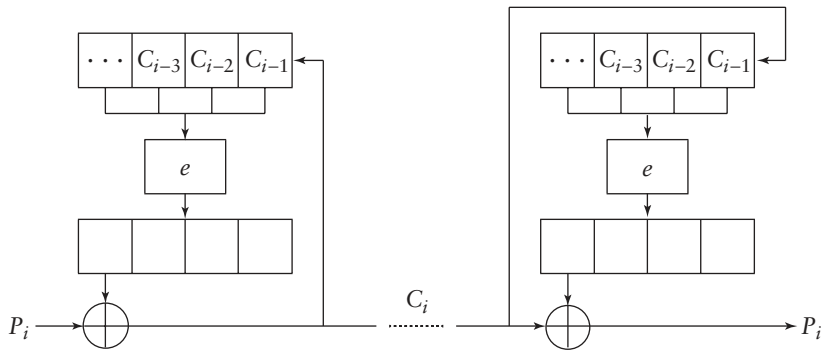


Figure 14.9: Cipher Feedback Mode

14.5.3 RSA Encryption

The set-up is already familiar from the RSA signature scheme. When RSA is used as a public-key encryption algorithm, user A picks two prime numbers p and q, and a private decryption exponent d with $\gcd(d, p-1) = 1$ and $\gcd(d, q-1) = 1$. The public encryption key consists of the product $n = p \cdot q$ and an exponent e with

$$e \cdot d = 1 \bmod \operatorname{lcm}(p-1, q-1).$$

Messages have to be divided into blocks so that each block is an integer less than n. To send a message block m to A, the sender computes

$$c = m^e \bmod n.$$

The receiver A uses the private decryption key d to obtain

$$c^d = m^{e \cdot d} = m \bmod n.$$

Do not be deceived by the simplicity of this algorithm. Proper implementation can be tricky and simplistic implementations of 'textbook RSA' are likely to be insecure.

Padding

RSA is a block cipher. According to the key length, messages are broken into blocks of 1024 (or 2048, 4096, ...) bits. When encrypting a message, padding may have to be added to make the message length a multiple of the block length. Padding can defeat some attacks. When decrypting a message, the receiver can check the padding data and discard plaintexts with syntactically incorrect padding. On the other hand, padding might be exploited for an attack. Once, the standard PKCS #1 v1.5 recommended padding a data value D as follows:

00	02	*PS*	00	*D*

The first two padding bytes have the values 0 and 2, respectively. *PS* is a string of pseudo-randomly generated non-zero bytes of length $|n| - |D| - 3$ ($|.|$ gives length in bytes). Then there is another byte of value 0 before the data D.

Bleichenbacher found an attack against RSA with this padding scheme that requires approximately 2^{20} chosen ciphertexts to get the plaintext if the receiver signals whether decryption fails or succeeds [41]. A typical setting for this attack would be the SSL protocol (Section 16.5). The data value to be retrieved is a session key, the receiver is a server. The attacker intercepts an encrypted session key. The attacker then sends a chosen ciphertext to the server. The server replies with an error message when decryption fails. No error signals success and narrows the interval containing the session key. The attacker keeps repeating this game and after about 2^{20} attempts the key is uniquely defined (in cryptography, a million can be a small number).

Because of this attack, Optimal Asymmetric Encryption Padding (OAEP) was adopted as the new standard for padding RSA messages. OAEP was attractive because it came with a formal proof of its security [28]. The proof, however, was found to be flawed but has been partially fixed [210]. New padding attacks have appeared [159]. At the time of writing, research on secure padding schemes is still ongoing.

Lesson
Even if cryptographic algorithms are secure at an abstract mathematical level, it may be possible to find attacks against concrete implementations. Thus, additional security analysis at lower levels of abstraction is called for.

14.5.4 ElGamal Encryption

In the ElGamal public-key algorithm, p is a large and appropriately chosen prime number. Let g be an integer of large order modulo p, let a be the private decryption key of user A, and $y_a = g^a \bmod p$ the corresponding public encryption key. Messages have to be

divided into blocks so that each block is an integer less than p. To send a message block m to A, the sender picks a random number k, computes $r = g^k \bmod p$, and sends the ciphertext

$$(c_1, c_2) = (r, my_a^k)$$

to A. With its private decryption key a, A obtains

$$c_2 \cdot c_1^{-a} = m \cdot y_a^k \cdot r^a = m \cdot g^{ak} \cdot g^{-ak} = m.$$

In this scheme, a ciphertext block is twice as long as a plaintext block. On the other hand, if two plaintext blocks are equal, the corresponding ciphertext blocks will still be different. The random number k must be used for one encryption only. Reuse of random numbers seriously weakens the cipher.

14.6 STRENGTH OF MECHANISMS

Measuring the strength of cryptographic algorithms is an imprecise art, sometimes resting on firm mathematical foundations, sometimes relying on intuition and experience. A cryptographic algorithm can be

- empirically secure,
- provably secure,
- unconditionally secure.

An algorithm is *empirically secure* if it has withstood the test of time. Prolonged analysis has found no serious weakness and, although there is no proof that the algorithm may not eventually fall to a new and ingenious attack, the algorithm has found acceptance within the cryptographic community. Taking aside its key length, DES was the prime example of an empirically secure algorithm. New analytical methods such as differential cryptanalysis have strengthened rather than weakened the perceived security of DES.

Provably secure algorithms seem to offer what computer security has been longing for, i.e. provable security. Provable security is expressed within the framework of complexity theory. An algorithm is secure if breaking the algorithm is at least as difficult as solving another problem that is known to be hard. This sounds wonderful, but it pays to read the small print.

Being 'at least as difficult' is an asymptotic concept. It holds for problem instances that are 'sufficiently large'. The theory will not tell you what is sufficiently large. Instead, you have to assess the current power of computing equipment and the progress in algorithm design. For example, the size of numbers we can factor has constantly increased over the years. This is an empirical argument. Even worse is to come. Cryptography's favourite hard problems are factoring and the discrete logarithm problem. There is actually no proof that these problems are necessarily difficult to solve. Again, cryptography relies

on empirical arguments that no fast algorithms have been found so far and that a major breakthrough looks extremely unlikely. To be more positive, this theory has also led to constructive results that give lower bounds on the effort required to break a cryptographic scheme in relation to the difficulty of breaking some other cryptographic primitive.

Provably secure algorithms can be broken by an attacker with sufficient computing resources. Of course, you would hope that the resources necessary are beyond the capacity of any attacker. *Unconditionally secure* algorithms cannot be broken even by attackers with unlimited computing power. Unconditional security is expressed in terms of information theory. An algorithm is secure if an attacker does not gain additional information about the plaintext from observing the ciphertext.

The standard example of an unconditionally secure algorithm is the *one-time pad*. Sender and receiver share a truly random key stream which they use only once. The ciphertext is the bitwise exclusive-OR of plaintext and key stream. The receiver exclusive-ORs the same key stream to the ciphertext to retrieve the plaintext:

$$ciphertext \oplus key\ stream = plaintext \oplus key\ stream \oplus key\ stream = plaintext.$$

Because every key is equally probable, the attacker cannot guess anything about the plaintext that could not have been guessed before seeing the ciphertext.

Be wary! Even unconditionally secure ciphers have been broken. When operators cut corners and use the same key stream twice, an attacker overlaying the two ciphertexts will see the combination of two plaintexts:

$$\begin{aligned} ciphertext_1 \oplus ciphertext_2 &= plaintext_1 \oplus keystream \oplus plaintext_2 \oplus keystream \\ &= plaintext_1 \oplus plaintext_2. \end{aligned}$$

Making sense of two overlaying plaintext messages is not exactly a hard cryptanalytic problem. The Venona project documents that such incidents happened even at the height of the Cold War.

The last remark highlights a crucial fact. More often than not, cipher systems are broken because of bad key management rather than because of an inherent weakness in the algorithm. The Enigma machine of World War II is the most famous illustration of this point. The security of key management protocols is therefore of utmost importance. Key management protocols are covered in Chapter 15.

14.7 PERFORMANCE

To give an indication of the execution times for cryptographic algorithms, we summarize some of the performance data reported in [192], where very detailed studies for many algorithms and platforms can be found. The measurements we quote refer to (not

optimized) software implementations of the algorithms, running on a Pentium III processor under Linux. Execution times vary between different compilers. Table 14.1 gives performance measurements for hash functions, stream ciphers, and block ciphers as the number of instruction cycles per byte. Table 14.2 compares RSA and two ECDSA variants. Measurements are given as the number of cycles (in millions) per invocation. RSA encryption and RSA signatures use the public exponent $e = 3$ to demonstrate the maximum gain achievable using short public exponents. The key length of the algorithms is given for reference.

Algorithm	cycles/byte
RC4	7–8
MD5	7–8
SHA-1	15
SHA2-512	83
Rijndael-128	25–30
DES	60

Table 14.1: Performance Measurements for Hash Functions and Symmetric Key Algorithms

algorithm	operation	cycles per invocation	key set up (cycles)	key length (bits)
RSA-OAEP	encrypt	2.026 M	1.654 M	1024
	decrypt	42.000 M		
RSA-PSS	sign	42.000 M	1.334 M	1024
	verify	2.029 M		
ECDSA-GF(p)	sign	4.775 M	4.669 M	160
	verify	6.085 M		
ECDSA-GF(2^m)	sign	5.061 M	4.825 M	163
	verify	6.809 M		

Table 14.2: Performance Measurements for Asymmetric Encryption and Digital Signature Algorithms

14.8 FURTHER READING

If you are interested in the history of secret communications, [134] is the book for you. For an up-to-date professional reference on cryptography, use [168]. A nice and non-mathematical introduction to modern cryptography, together with an extensive bibliography and collection of cryptographic algorithms, can be found in [206]. In these books, you will find all the details and further information on

the algorithms presented in this chapter. New cryptological results are posted to the eprint server of the International Association for Cryptologic Research `http://eprint.iacr.org`.

The founding paper of public-key cryptography is [82]. The wraps have been removed from even earlier classified research by CESG on public-key systems [87]. Prime numbers are an important ingredient in many public-key algorithms. An excellent explanation of deterministic primality testing can be found in [109]. For developments on new block cipher modes, consult the NIST special publications 800-38A, 800-38B, and 800-38C. The Venona documents can be found at `http://www.nsa.gov/venona/`.

14.9 EXERCISES

Exercise 14.1 Cryptographic protocols are intended to let agents communicate securely over an insecure network. Is this statement correct?

Exercise 14.2 Cryptography needs physical security. To what extent is this statement correct?

Exercise 14.3 Assuming that it is computationally infeasible to launch attacks that require 2^{80} computations of hash values, how long should the hash values be to achieve weak and strong collision resistance, respectively?

Exercise 14.4 For DSA, show that the condition $v = r$ holds for valid signatures.

Exercise 14.5 Given a modular exponentiation algorithm for n-bit integers that needs about n^3 operations, how much does performance deteriorate by moving from 1024-bit to 2048-bit RSA?

Exercise 14.6 Consider the RSA signature algorithm without a hash function, i.e. $s = m^e \bmod n$. Explain how, and to what extent, an attacker could forge signatures if there are no redundancy checks on the message m.

Exercise 14.7 When a document is too long to be processed directly by a digital signature algorithm, a hash of the document is computed and then signed. Which properties do you require from this hash function to prevent an attacker from forging signatures?

- Distinguish between situations where the attacker only knows messages signed by the victim and situations where the attacker can choose messages the victim will sign.

- Distinguish between *selective* forgeries, where the attacker has control over the content of the forged message, and *existential* forgeries, where the attacker has no control over the content of the forged message.
- Consider the specific requirements of hash functions used with an invertible signature algorithm such as RSA.

Exercise 14.8 In the DSA signature algorithm, why should the verifier make sure that g is an element of large order?

Exercise 14.9 In the ElGamal signature scheme, show how the private signature key can be compromised if the random value k is used in signing two different documents.

Exercise 14.10 Are NP-complete problems a suitable basis for constructing cryptographic algorithms?

Chapter 15

Key Establishment

Cryptography transforms (communications) security problems into key management problems. To use encryption, digital signatures, or message authentication codes, the parties involved have to hold the 'right' cryptographic keys. With public-key algorithms, parties need authentic public keys. With symmetric key algorithms, parties need shared secret keys. This could be achieved by sending letters through the mail – a common method of distributing PINs for credit cards – or by couriers travelling between sites and delivering keys. These proposals are either not very secure or not very cheap. Ideally, we would like to conduct key management over the existing communications infrastructure.

When two parties negotiate a new key, be it because they are communicating for the first time or because they are starting a new session, they may also have to prove who they are. Proving identity and establishing keys were once both called authentication. This chapter will discuss and separate these two issues.

OBJECTIVES

- Clarify the possible meanings of authentication.
- Present some major key establishment protocols.
- Discuss how public keys can be linked to user identities.
- Show how cryptographic protocols are being applied in computer security.

15.1 INTRODUCTION

Public-key algorithms tend to be computationally more expensive than symmetric key algorithms. Cost factors include computation time and bandwidth. Both depend on key length. Furthermore, it is desirable to use long-term keys only sparingly, in order to reduce the 'attack surface'. This is a precaution against attacks that need to collect large amounts of encrypted material. As a solution for both problems, long-term keys are used to establish short-term *session keys*.

It is good cryptographic practice to restrict the use of keys to a specific purpose. Examples of *key usage* are encryption, decryption, digital signature (for authentication), and non-repudiation in communications security. Other examples are key encrypting keys and data encrypting keys in key management, e.g. as used in digital rights management. Similarly, master keys and transaction keys have been used in hierarchical key management schemes in the financial sector. In public-key cryptography, you are strongly advised not to use a single key pair both for encryption and digital signatures.

15.2 KEY ESTABLISHMENT AND AUTHENTICATION

Once upon a time, protocols establishing a session key were called authentication protocols. After all, it is their purpose to let you know 'whom you are talking to'. In the literature, in particular in older sources, you may still find this convention. Today the terminology used in cryptology distinguishes between authentication and key establishment. The separation of key establishment from authentication can be traced in the development of international standards. In the 1980s, the influential ISO/OSI framework (ISO 7498-2, [125]), still had a session-oriented view of entity authentication.

> Peer entity authentication: The corroboration that a peer entity in an association is the one claimed. This service is provided for use at the establishment of, or at times during, the data transfer phase of a connection to confirm the identities of one or more of the entities connected to one or more of the other entities.

By the early 1990s, ISO/IEC 9798-1 [126] defined entity authentication in a way that no longer included the establishment of a secure session:

> Entity authentication mechanisms allow the verification, of an entity's claimed identity, by another entity. The authenticity of the entity can be ascertained only for the instance of the authentication exchange.

In this interpretation, entity authentication checks whether an entity is alive. This property is related to *dead peer detection* in communications networks.

In *unilateral* authentication only one of the entities is authenticated. In *mutual* authentication, the identities of both entities are verified.

15.2.1 Remote Authentication

Whenever you change your environment, you have to reappraise the suitability of established security mechanisms. Moving from a centralized system to a distributed system definitely has an impact on security. To see how change can affect you, have another look at authentication by password (Chapter 4). Passwords are useful when a user works on a terminal that has a fixed link to a host. Here, you may have valid reasons to believe that the connection between terminal and host is secure and that it is not possible to eavesdrop on passwords, change or insert messages, or to take over a session. In a distributed setting, this fundamental assumption on the security of the communications link is unlikely to be justified.

Unprotected passwords transmitted over networks are an obvious vulnerability. Exploitation of this vulnerability can easily be automated. *Password sniffers* are programs that listen to network traffic and extract packets containing passwords and other security-relevant information. Still, passwords are a popular authentication mechanism in distributed systems. Take the HTTP protocol as an example. It is run between a client and a server. The client sends HTTP requests to the server. To authenticate the client, HTTP has built-in authentication mechanisms based on a shared secret (the password) known by client and server. There are two types of authentication, basic access authentication and digest access authentication.

In basic access authentication, the client just has to provide the password. When the client asks for a protected resource, the server replies with the 401 Unauthorized response code. The client then sends authentication information (the base64 encoded password) to the server, and the server checks whether the client is authorized to access the resource. All authentication data is sent in the clear. A protocol run looks like this:

- Client: `GET /index.html HTTP/1.0`
- Server: `HTTP/1.1 401 Unauthorized`
  `WWW-authenticate Basic realm="SecureArea"`
- Client: `GET /index.html HTTP/1.0`
  `Authorization: Basic am9ldXNlcjphLmIuQy5E`
- Server: `HTTP/1.1 200 Ok` (and the requested document)

In contrast, digest access authentication does not send passwords in the clear. A cryptographic hash function h is used in a *challenge-response* protocol. The WWW-authenticate parameters sent by the server include a unique challenge, called a *nonce*. It is the server's responsibility to make sure that every 401 response comes with a unique, previously unused nonce value. The client replies with an authorization response containing the plaintext username, the nonce value it just received, and the so-called request-digest, computed as

$$\text{request-digest} = h(h(\text{username}||\text{realm}||\text{password}) \,||\text{nonce}||h(\text{method}||\text{digest-uri}))$$

where 'digest-uri' relates to the requested uniform resource identifier (URI) and 'method' gives the HTTP request method. A detailed description of this protocol with all its options is given in RFC 2617.

The term nonce was proposed in [180] to denote a unique value that is used only once. Nonces play an important role in protocol design. A nonce can be a counter value, a time stamp, or a random number. A nonce is not necessarily unpredictable, but some protocols require unpredictable nonces. It depends on the particular security goals which type of nonce should be used.

A further example of remote user authentication by password is the Remote Authentication Dial-In User Service protocol (RADIUS, RFC 2865). Like HTTP digest access authentication, RADIUS includes a challenge-response option where the password is not transmitted in the clear.

15.2.2 Key Establishment

The process whereby a shared secret becomes available to two or more parties for later cryptographic use is today called *key establishment*. This process involves the parties wishing to establish the shared keys, often called the *principals*, and possibly third parties such as authentication servers. Principals in key establishment protocols are not necessarily the same as principals in access control (Chapter 5). When the third party could violate the security goals of the protocol, it is called a *Trusted Third Party* (TTP). The principals have to trust the TTP when they invoke its services.

Sometimes, an analysis of the mathematical details of a cryptographic protocol reveals the existence of a subset of *weak keys* that would allow an insider to cheat. When designing a key exchange protocol you therefore should answer two questions:

- Which party will suffer if a weak key is established?
- Which parties can control the choice of key?

If a misbehaving insider can influence key generation so that a weak key is chosen, there may be a scope for insider attacks. This issue is known as *key control*. Key establishment services can be further distinguished according to the contributions each principal makes to the new key, and to the actual security guarantees provided [168].

- Key transport: One party creates the secret value and securely transfers it to the other(s).
- Key agreement: Both parties contribute to the generation of the secret value so that no party can predict the outcome.
- Key authentication: One party is assured that no other party aside from a specifically identified second party may gain access to a particular secret key.

- Key confirmation: One party is assured that a second (possibly unidentified) party has possession of a particular secret key.
- Explicit key authentication: Both key authentication and key confirmation hold.

Figure 15.1 shows how the meaning of entity authentication has evolved over time. Other factors to consider when assessing a protocol are third party requirements. Is a TTP involved? Would it be off-line or on-line?

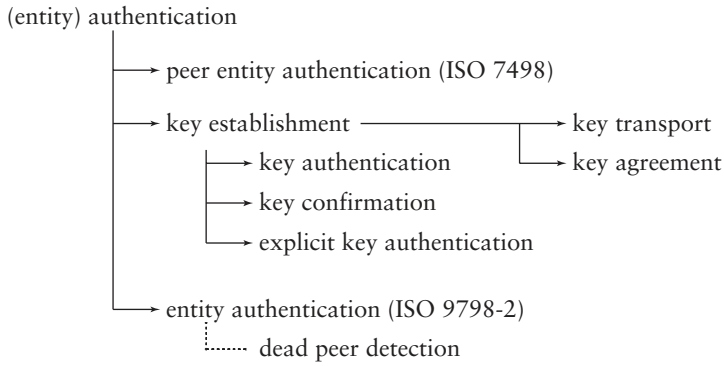


Figure 15.1: Terminology of Authentication and Key Establishment

15.3 KEY ESTABLISHMENT PROTOCOLS

Cryptographic protocols that establish keys for use by other protocols are known as key establishment protocols. The research literature offers an extensive choice of such protocols. The following protocols have been selected to illustrate important design techniques.

15.3.1 Authenticated Key Exchange Protocol

Our first protocol, Authenticated Key Exchange Protocol 2 (AKEP2, [27]), uses 'cheap' hash functions instead of encryption and does not rely on a TTP. Two principals, A and B, share two long-term symmetric keys, K and K', and in each protocol run generate fresh random nonces, n_a and n_b, respectively. The protocol uses a keyed hash function (message authentication code), h_K, and a keyed one-way function, $h'_{K'}$. AKEP2 is a three-pass protocol:

$$\begin{aligned} &1.\ A \to B: \quad n_a \\ &2.\ B \to A: \quad B, A, n_a, n_b, h_K(B, A, n_a, n_b) \\ &3.\ A \to B: \quad A, n_b, h_K(A, n_b) \end{aligned}$$

In the first step, A sends a challenge n_a. In the second step, B responds with $h_K(B, A, n_a, n_b)$ and sends its own challenge n_b. The shared key is $k = h'_{K'}(n_b)$. In the third step, A responds to this challenge with $h_K(A, n_b)$. AKEP2 provides mutual entity authentication and (implicit) key authentication.

15.3.2 The Diffie–Hellman Protocol

The Diffie–Hellman protocol is a key agreement protocol [82]. Principals A and B do not share a secret in advance. They agree on a group G of prime order q, and on a generator g of this group. The group G could be defined as a subgroup of order q in the group of integers modulo p, where p is a large and appropriately chosen prime number. It could also be defined by an elliptic curve.

Principal A picks a random number x and sends $X = g^x$ to B. Principal B picks a random number y, sends $Y = g^y$ to A, and computes X^y. On receipt of Y, A uses its own secret x to compute Y^x. Because of

$$X^y = g^{xy} = g^{yx} = Y^x$$

both parties now share the secret g^{xy}. They can compute shared keys by taking the required number of bits from the hash $h(g^{xy})$. The security of this protocol depends on the difficulty of the discrete logarithm problem (Section 14.2) in group G. An attacker able to compute discrete logarithms could obtain x and y from g^x and g^y. It is not known whether the security of the Diffie–Hellman protocol is equivalent to the discrete logarithm problem.

Man-in-the-Middle Attack

There is one problem left. Neither party knows whom it shares the secret with. This can be exploited by a man-in-the-middle attack. The attacker M inserts itself on the communications path between A and B, replies to A when A initiates a protocol run and at the same time starts a protocol run with B where M pretends to be A (Figure 15.2). Principals A and B might believe they have established a shared key. In fact M shares key g^{xu} with A and key g^{yv} with B, and can read all traffic between A and B while acting as a relay.

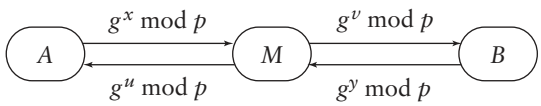


Figure 15.2: Man-in-the-Middle Attack against the Diffie–Hellman Protocol

MQV Protocol

To make the Diffie–Hellman protocol resilient against man-in-the-middle attacks, authentication has to be added. The Menezes–Qu–Vanstone (MQV) protocol modifies the key derivation phase of the Diffie–Hellman protocol to let the shared key depend on the participants' public keys [149]. Let a be the private key of A and g^a its public key; let b be the private key of B and g^b its public key. Set $l = \lceil (\log_2 q)/2 \rceil$, i.e. half the bit length of the group order q.

A and B run the Diffie–Hellman protocol to exchange *ephemeral* Diffie–Hellman values X and Y. Both parties then compute exponents d and e of length $l + 1$. Variants of MQV differ in the choice of exponents. Some variants include the names (identifiers) of A and B, id_A and id_B, in the computation and use a hash function h' with hash values of length l:

- $d = 2^l + (X \bmod 2^l), e = 2^l + (Y \bmod 2^l)$ (MQV) – the Diffie–Hellman values X and Y are truncated to l bits and the most significant bit is set to 1.
- $d = h'(X, id_B), e = h'(Y, id_A)$ (HMQV) – *hashed* MQV includes the names of the principals and applies a hash function; with this modification, the protocol can be formally proven to be secure [139].
- $d = h'(X, Y, id_A, id_B), e = h'(Y, X, id_A, id_B)$ (FHMQV) – *fully hashed* MQV extends the hashes over all parameters shared in a protocol run [201]; computing hashes over all values shared in a protocol run is a frequently used technique in the design of security protocols.

In all variants, principal A uses its private key a and ephemeral secret x to compute $(Y \cdot B^e)^{x+da}$; B uses its private key b and ephemeral secret y to compute $(X \cdot A^d)^{y+eb}$. As

$$(Y \cdot B^e)^{x+da} = (g^y \cdot g^{be})^{x+da} = g^{(y+be)(x+da)} = g^{(x+ad)(y+eb)} = (X \cdot A^d)^{y+eb}$$

both parties now share an authenticated secret and can derive shared keys from the hash of this secret. The MQV protocol provides mutual key authentication. Attacks on MQV and HMQV (despite the security proof) have been reported.

15.3.3 Needham–Schroeder Protocol

The Needham–Schroeder protocol is a key transport protocol [180]. Two parties A and B obtain their session key from a server S. Both principals share a secret key with the server in advance. A symmetric cipher is used for encryption. Nonces (random challenges) are included in the messages to prevent replay attacks. The following conventions are used in the description of the protocol:

- K_{as}, a secret key shared by A and S;
- K_{bs}, a secret key shared by B and S;
- K_{ab}, a session key created by S for use between A and B;
- n_a, n_b, nonces generated by A and B, respectively;

Figure 15.3 shows the steps that take place when A requests from the server S a session key K_{ab} that is intended for communication with B. In the first three protocol steps, A obtains the session key from S and forwards it to B. By checking the nonce n_a returned in the server's message, A can verify that the session key has been issued in response to its recent request and is not a replay from a former protocol run. In the last two steps, B verifies that A is currently using the same session key. In steps 4 and 5, A performs a unilateral entity authentication of B.

1. $A \rightarrow S:\ A, B, n_a$
2. $S \rightarrow A:\ eK_{as}(n_a, B, K_{ab}, eK_{bs}(K_{ab}, A))$
3. $A \rightarrow B:\ eK_{bs}(K_{ab}, A)$
4. $B \rightarrow A:\ eK_{ab}(n_b)$
5. $A \rightarrow B:\ eK_{ab}(n_b - 1)$

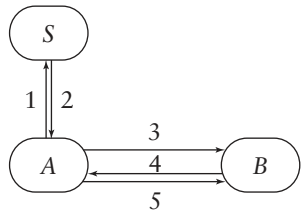


Figure 15.3: Message Flow in the Needham–Schroeder Protocol

The Denning–Sacco Attack

The Needham–Schroeder protocol achieves its goals under the standard assumption that the long-term keys K_{as} and K_{bs} are not compromised, and if just a single protocol run is considered. Denning and Sacco discovered a *replay attack* where the adversary M impersonates A reusing a compromised session key K_{ab} [81]. The adversary starts at step 3 of the protocol and replays to B the third message from the protocol run when the compromised session key was established. B decrypts the message and reuses the compromised session key.

$$\begin{aligned} 3.\ M \rightarrow B:&\quad eK_{bs}(K_{ab}, A) \\ 4.\ B \rightarrow M:&\quad eK_{ab}(n'_b) \\ 5.\ M \rightarrow B:&\quad eK_{ab}(n'_b - 1) \end{aligned}$$

This is a *known key attack*, using a compromised old session key to compromise a future session. As far as B is concerned, the Needham–Schroeder protocol does not provide *key freshness*.

> A key is *fresh* (from the viewpoint of one party) if it can be guaranteed to be new [168].

Key freshness helps to protect against replay attacks.

Perfect Forward Secrecy

When analyzing protocols, it makes sense to treat the compromise of past session keys differently from the compromise of long-term secret keys. When a long-term key is compromised, we can no longer protect future sessions. However, we would still like past sessions to remain secure. A protocol achieves *perfect forward secrecy* if the compromise of session keys or long-term keys does not compromise past session keys. The term 'forward secrecy' indicates that the secrecy of old session keys is carried forward into the future.

15.3.4 Password-Based Protocols

In the Needham–Schroeder protocol, client and server already share secret keys when they start a protocol run. Thus, both systems must be equipped to store keys securely. If the client is a person accessing the server via an 'untrusted' device we can only rely on

shared secrets the users can memorize, i.e. we are back to something like passwords. We could then use a password P to encrypt a randomly generated session key K_s, and use the session key to encrypt further data.

$$\begin{aligned} &1.\ A \rightarrow B: \quad eP(K_s) \\ &2.\ B \rightarrow A: \quad eK_s(\text{data}) \end{aligned}$$

This naïve protocol has a problem. It is vulnerable to an *off-line dictionary attack*. The attacker guesses the password P, decrypts the first message and gets a candidate session key K'_s, which is then used to decrypt the second message. When meaningful text emerges it is likely that the password had been guessed correctly.

The Encrypted Key Exchange (EKE) protocol avoids this problem [31]. It uses a symmetric encryption algorithm to encrypt data with the password P as the key, and also a public-key encryption system. In a protocol run, principal A generates a random public/private key pair K_a, K_a^{-1}. In the first message A sends the public key K_a to B, encrypted under P (symmetric encryption). In the second message B sends the randomly generated session key K_s to A, encrypted first under K_a (public-key encryption) and then under P (symmetric encryption).

$$\begin{aligned} &1.\ A \rightarrow B: \quad eP(K_a) \\ &2.\ B \rightarrow A: \quad eP(eK_a(K_s)) \end{aligned}$$

It is left as an exercise to show that this protocol is not vulnerable to an off-line dictionary attack. Generating fresh key pairs for each protocol run is not a trivial exercise. There has been further work on the design of more efficient password-based key establishment protocols, and on their formal analysis; see, for example, [111].

15.4 KERBEROS

Kerberos was developed at MIT within the Athena project in the 1980s. Athena provided computing resources to students across and beyond the MIT campus and included additional administrative functionalities such as accounting. The risks and threats addressed by Kerberos, as stated in [170], are:

> The environment is not appropriate for sensitive data or high risk operations, such as bank transactions, classified government data, student grades, controlling dangerous experiments, and such. The risks are primarily uncontrolled use of resources by unauthorized parties, violations of integrity of either system's or user's resources, and wholesale violations of privacy such as casual browsing through personal files.

Kerberos has since found wide acceptance. Several industry standards have adopted Kerberos for distributed systems authentication, notably the Internet RFC 4120. Kerberos authenticates clients to services in a distributed system. Authentication is built around the concepts of *tickets* and central security servers.

Kerberos has its origin in the Needham–Schroeder key exchange protocol (Section 15.3.3). A symmetric cipher system is used for encryption. Users are authenticated by username and password, but passwords are not transmitted over the network. RFC 4120 gives a detailed specification of Kerberos Version 5, including the error messages that are issued when a protocol run cannot proceed. Our simplified description omits some of the data fields in the Kerberos messages and deals only with the case where a protocol run is successfully completed.

Kerberos lets a user A at a client machine C get access to a server B, involving an authentication server S. User A shares a secret key K_{as} with the authentication server S. This key is derived from the user's password with a one-way function. K_{bs} is a secret key shared by B and S, K_{ab} is the session key created by S for use between A and B, n_a is a nonce generated by the client for A's protocol run, and T_c is a time stamp referring to the client's clock. The ticket for B is $\text{ticket}_B = eK_{bs}(K_{ab}, A, L)$, where L is the lifetime of the ticket. The core protocol works as follows.

$$
\begin{aligned}
&1.\ C \to S: && A, B, n_a \\
&2.\ S \to C: && eK_{as}(K_{ab}, n_a, L, B), \text{ticket}_B \\
&3.\ C \to B: && \text{ticket}_B, eK_{ab}(A, T_c) \\
&4.\ B \to C: && eK_{ab}(T_c)
\end{aligned}
$$

To start a session, user A logs on at client C, entering username and password. The client sends A's request for authentication to B to S. The first message contains A's identity, the name of the server, and a nonce, all sent in the clear. In the second step, S looks up A's key K_{as} in its database (the protocol stops if A is unknown to S), generates the session key K_{ab} and ticket_B. S sends the session key, in a data structure encrypted under K_{as}, and the ticket to C. The client derives the key K_{as} from A's password and decrypts the first part of the reply, obtains the session key K_{ab}, and verifies the nonce. In the third step, C sends ticket and *authenticator* $eK_{ab}(A, T_c)$ to B. B decrypts the ticket with K_{bs} and obtains the session key K_{ab}. B checks that the identifiers in ticket and authenticator match, that the ticket has not expired, and that the time stamp is valid. The validity period for time stamps has to consider the skew between the local clocks of C and B. In the fourth step, B returns the time stamp T_c encrypted under the session key K_{ab} to C.

Kerberos is traditionally deployed using ticket granting servers in conjunction with an authentication server. The Kerberos authentication server (KAS) authenticates principals at logon and issues tickets, which in general are valid for one login session and enable principals to obtain other tickets from ticket granting servers. The authentication server is sometimes called the key distribution centre (KDC). A ticket granting server (TGS) issues tickets that give principals access to network services demanding authentication. Figure 15.4 shows the steps that take place in a protocol run involving a KAS and a TGS. Here, $K_{a,tgs}$ is a session key created by the KAS for use between C and the TGS. There is a second nonce n'_a and time stamp T'_a. L_1 and L_2 are the lifetimes of the two tickets.

1. $C \rightarrow KAS$: A, TGS, n_a
2. $KAS \rightarrow C$: $eK_{as}(K_{a,tgs}, n_a, L_1, TGS)$, $\text{ticket}_{A,\,TGS}$
3. $C \rightarrow TGS$: $\text{ticket}_{A,TGS}, eK_{a,tgs}(A, T_a), B, n'_a$
4. $TGS \rightarrow C$: $eK_{a,tgs}(K_{ab}, n'_a, L_2, B)$, ticket_B
5. $C \rightarrow B$: $\text{ticket}_B, eK_{ab}(A, T'_a)$
6. $B \rightarrow C$: $eK_{ab}(T'_a)$

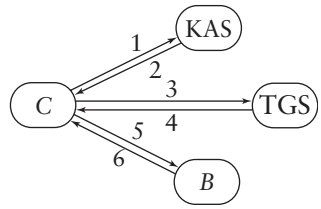


Figure 15.4: The Message Flow in the Kerberos Authentication Protocol

The *ticket granting ticket* $\text{ticket}_{A,TGS}$ is constructed as

$$\text{ticket}_{A,TGS} = eK_{tgs}(K_{a,tgs}, A, L_2)$$

where eK_{tgs} is a key shared by KAS and TGS.

15.4.1 Realms

A Kerberos realm is a single administrative domain that controls access to a collection of servers. A KAS is at the heart of a Kerberos realm. To get Kerberos up and running, principals have to be registered with the KAS, the TGSs have to receive access control information, and all the necessary keys have to be put in place by the security administrator.

Kerberos has the advantages of a centralized security system. A single security policy is enforced by a limited number of security servers. Thus, it is relatively easy to check that the system set-up complies with the security policy and to implement changes if so desired.

A realm often corresponds to a single organization. To facilitate access to services in other organizations, you need inter-realm authentication. This requires a 'trust relationship' between the authentication servers in different realms. In this case, 'trust' is a shared secret key. Between organizations, key sharing is often underpinned by contractual agreements. Is trust transitive? If there is trust between realms R_1 and R_2, and between realms R_2 and R_3, can a client in R_1 get access to a server in R_3? There is no universal answer. The outcome depends on prior agreements between the realms.

As an example, consider a user A in realm R_1 who requests from its authentication server KAS_1 a ticket for a server B in realm R_3. The user, or some discovery service, has found out that KAS_1 has a trust relationship with KAS_2 and that KAS_2 has a trust relationship with KAS_3, and that these trust relationships are transitive. On A's request, KAS_1 generates a ticket granting ticket (TGT) for realm R_2 and forwards this TGT together with A's request to KAS_2. In turn, KAS_2 generates a TGT for realm R_3, and forwards this TGT together with A's request to KAS_3. KAS_3 creates the ticket for B and sends it to A. The client where A has logged on presents this ticket when requesting a service from B.

15.4.2 Kerberos and Windows

Kerberos has become the authentication protocol of choice in Windows. Windows domains correspond to Kerberos realms. Domain controllers act as KDCs. The principals that may run Kerberos are users but also machines. In Windows, authentication is the basis for later access control decisions. The principals in Windows access control are the SIDs. (Here, we finally have a clash between the two definitions of principal.) Therefore, a principal's SIDs have to be stored in the ticket. A Kerberos ticket, as defined in RFC 4120, contains *inter alia* a mandatory field *cname* for the client name and an optional *authorization-data* field. In Windows, *cname* holds the principal's name and realm, e.g. diego@tuhh.de, and *authorization-data* holds the group SIDs. Details of Microsoft's implementation of Kerberos can be found in [47].

15.4.3 Delegation

In distributed systems, controlled invocation literally takes on a new dimension. A user may log on at a local node and then execute a program on a remote node. To obtain access to resources at the remote node, the process executing the program will need the relevant access rights. Typically, the program would be endowed with the access rights of the user and then run with these access rights on the remote node. This process is called *delegation*.

The process on the remote node now runs with all the access rights delegated by the user. In a distributed system, users may not feel too comfortable about releasing all their rights to a node they have little control over. If there is weak protection on the remote node, an attacker may grab the user's access rights and use them for illicit purposes. It may be desirable for users to be able to control which rights they delegate, to have accountability mechanisms that monitor the use of delegated access rights.

The Kerberos implementation in Windows supports different modes of delegation. When explaining these modes, we follow the popular convention of calling the parties involved Alice and Bob. When Alice needs a service from Bob, where Bob has to access servers on her behalf, and when she knows in advance what Bob is going to need, she applies for *proxy tickets* for the relevant servers and gives the tickets and the corresponding session keys to Bob. These server tickets are marked as proxy tickets and must contain special authorizations that limit how Bob can use Alice's credentials, e.g. state the name of a file Bob is allowed to print. If she does not know in advance what Bob is going to need, Alice applies for a *forwarded* TGT for Bob and transfers this ticket and corresponding session key to Bob. In this way, Alice delegates her identity to Bob. Bob can now apply for tickets on her behalf. Bob can impersonate Alice. In [47], this is aptly called 'the fast and loose way to delegate credentials'. Principals can be nominated as OK-AS-DELEGATE to have some control over the delegation of credentials (identities).

Is delegating identities a good idea at all? At the level of subjects this is quite reasonable. SIDs give access rights, and a process running on behalf of a user cannot constantly

go back to that user and ask whether it is all right to pass on a SID to another process. However, when Alice delegates her identity to another user, she is doing the equivalent of giving her password away (for the time the forwarded ticket is valid). Sharing of passwords is usually frowned upon and may violate Alice's corporate security policy.

Concepts from operating system security may not always be appropriate at the application layer. Calling both users and processes Alice and Bob does not add to clarity either. In general, anthropomorphic metaphors such as 'Alice talks to Bob' or 'Bob verifies Alice's identity' can be very misleading. In computer security, the entities are computers, not human beings, they send messages over a network, they do not talk, and you have to decide what you mean by 'verifying *A*'s identity'. There is definitely no visual contact between computers.

Lesson
Alice and Bob are semantic sugar. They allow you to tell nice stories but they also invite you to think at the wrong level of abstraction.

15.4.4 Revocation

How can access rights be revoked from a principal? The system administrator of the KAS and the TGS have to update their databases so that these access rights are no longer available to the principal. The access rights are thus revoked for the next session, i.e. the next time the principal logs on or requests a ticket from the TGS. The tickets the principal already possesses are, however, valid until they expire. For example, KAS tickets usually have a lifetime of about one day. This is another instance of the TOCTTOU problem.

You now face a trade-off between convenience and security. If the TGS issues tickets with a distant expiry date, the principal has no need to access the TGS that often and the TGS may occasionally be off-line without too much impact on the users. However, revocation of an access right will take effect with a longer delay. If the TGS issues tickets with short lifetimes, principals have to update their tickets more regularly and the availability of the security servers becomes more important for system performance.

15.4.5 Summary

For a full assessment of Kerberos, you have to go beyond analyzing the authentication protocol and the strength of the underlying cryptographic algorithms. You also have to examine its non-cryptographic security features. The following points are part of such an investigation.

- Timeliness of messages is confirmed by checking time stamps. Therefore, reasonably synchronous clocks are required throughout the whole system and the clocks themselves

have to be protected against attacks. Secure clock synchronization in itself may require authentication.

- Checking of time stamps allows for some clock skew. The typical acceptance window of five minutes is rather large and can be exploited by replay attacks.
- Servers have to be on-line. The KAS is needed on-line at login, and the TGS is needed when a ticket is requested. Requirements on the availability of a TGS may be relaxed as discussed above.
- Session keys (for a symmetric cipher) are generated by Kerberos servers (authentication and ticket granting servers). As the session keys are used in subsequent communications between principals, trust in the servers has to encompass trust that servers will not misuse their ability to eavesdrop.
- Password guessing and password spoofing attacks are possible [235].
- Keys and tickets are held on the client's machine. Therefore, you rely on the protection mechanisms on that node for the security of Kerberos. As long as Kerberos users worked from simple terminals, this was not much of an issue. The situation changed once users ran Kerberos on a PC or a multi-user workstation.
- The initial client request is not authenticated. An attacker sending spoofed authentication requests would get tickets in return. This could constitute a denial-of-service attack against the authentication server or be an attempt to collect material for a cryptanalytic attack. As a countermeasure, the server could ask the user for authentication in a *pre-authentication* phase before generating any tickets.

Furthermore, it is important to distinguish the security of the protocol itself from the security of implementations of Kerberos. For example, one implementation of Kerberos Version 4 reportedly used a weak random-number generator for key generation, so that keys could be found easily by exhaustive search.

15.5 PUBLIC-KEY INFRASTRUCTURES

The description of the MVQ protocol omitted one important detail. How do A and B know that the verification keys they are using for authentication indeed correspond to the right party? This is a crucial problem in public-key cryptography. We rarely want to run protocols between cryptographic keys but between principals that have meaningful names (identities) at a higher protocol layer, e.g. usernames of clients or DNS names of servers. There has to be a reliable source that links those identities with cryptographic keys. Note that with symmetric ciphers, the parties A and B often trust a server to create this connection.

15.5.1 Certificates

Diffie and Hellman envisaged a public directory where one could look up the public keys of users, just as in a phone directory [82]. In a student project in 1978, Kohnfelder

implemented this directory as a set of digitally signed data records containing a name and a public key. He coined the term *certificate* for these records. Certificates originally had a single function, binding between names and keys. Today, the term has a wider use and you can find explanations like the following.

> Certificate: A signed instrument that empowers the Subject. It contains at least an Issuer and a Subject. It can contain validity conditions, authorization and delegation information [88].

In this spirit, we can define a certificate as a digitally signed document that binds a subject to some other information. Subjects can be people, keys, names, etc. Subjects in this context are not necessarily the same as subjects in access control (Chapter 5). Identity (ID) certificates bind names to keys. Sometimes this is still the default interpretation of the term 'certificate'. Attribute certificates bind names to authorizations. Authorization certificates bind keys to authorizations.

15.5.2 Certificate Authorities

The binding between subject and key, or some other information, is established by the party that issues (signs) the certificate. This party is known as the issuer. Certificate authority (CA) is just another name for issuer. When you issue certificates for your own use, you are a CA. Sometimes 'CA' is used more narrowly for organizations issuing ID certificates. The application determines the technical and procedural trust requirements a CA has to meet. The CA has to protect its own private key. It may have to check that the subject is who it claims to be and that the attributes in the certificate are correct. Checks may be performed at various levels of thoroughness. The VeriSign certificate classes may serve as an example. At the most elementary level, the CA just checks that subject names are unique. At the highest level, the subject has to appear in person with government-approved identity documents. Some of these checks can be performed by a registration authority (RA) and the CA's main task is the issuance of certificates.

Public-key infrastructure (PKI) is the somewhat imprecise term used to describe the system for issuing and managing certificates. Depending on your source, a PKI may be:

- software for managing digital certificates;
- a system of hardware, software, policies and people providing security assurances;
- the technology for securing the Internet;
- a worldwide system of digital ID cards.

There is no 'correct' definition. Whenever you encounter a so-called PKI, you have to establish which interpretation is intended in the given context before drawing any further conclusions.

15.5.3 X.509/PKIX Certificates

Today, X.509 version 3 (v3) is the most commonly used PKI standard. The original ITU-T Recommendation X.509 [128] was part of the X.500 Directory [59], which has since also been adopted as ISO 9594-1. X.500 was intended as something akin to a

global on-line telephone directory. X.509 certificates would bind public keys (originally passwords) to X.500 pathnames (distinguished names) to note who has permission to modify X.500 directory nodes. X.500 was geared towards identity-based access control:

> Virtually all security services are dependent upon the identities of communicating parties being reliably known, i.e. authentication.

This view of the world pre-dates the web and many new e-commerce scenarios, where a different kind of access control is more appropriate.

Compared to previous versions, the X.509 v3 certificate format (Figure 15.5) includes extensions to increase flexibility. Extensions can be marked as *critical*. If a critical extension cannot be processed by an implementation, the certificate must be rejected. Non-critical extensions may be ignored. Critical extensions can be used to standardize policy with respect to the use of certificates.

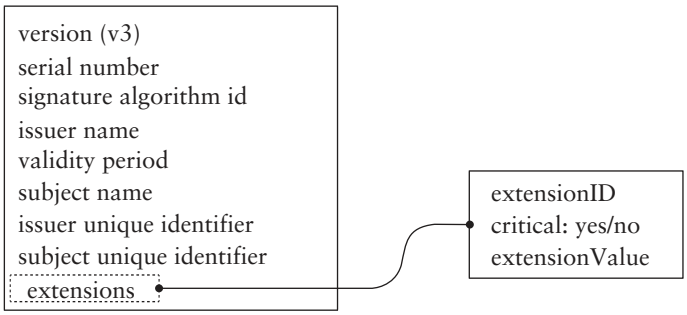


Figure 15.5: X.509 v3 Certificate Format

PKIX is the Internet X.509 public-key infrastructure [120]. It adapts X.509 v3 to the Internet by specifying appropriate extensions. A *public-key certificate* (PKC) contains a subject's public key and some further information. An *attribute certificate* (AC) contains a set of attributes for a subject. Attribute certificates are issued by attribute authorities. A PKI is the set of hardware, software, people, policies and procedures needed to create, manage, store, distribute, and revoke PKCs. A privilege management infrastructure (PMI) is a collection of ACs, with their issuing attribute authorities, subjects, relying parties, and repositories.

X.509 and PKIX are name-centric PKIs. To associate a key with access rights, you have to know the subject the key belongs to. Authorization attributes are linked to a cryptographic key via a common name in a PKC and an AC. An alternative is the simple public-key infrastructure (SPKI) [88], a key-centric PKI. SPKI certificates bind keys directly to attributes (access rights).

Lesson
Like user identities, certificates can serve two purposes. They can identify an entity associated with a cryptographic key or they can specify the access rights to be given to the holder of a cryptographic key (without indicating the holder's identity).

15.5.4 Certificate Chains

Technically, it is wrong to claim that a certificate is needed to verify a digital signature. An authentic copy of the verification key is needed. Verification keys may be stored in certificates, but can also be stored in protected memory. However, there are systems that require all verification keys to be stored in certificates. The Java 2 security model is an example.

To check a certificate, another verification key is needed, which might be vouched for by a certificate. This creates a certificate chain. Ultimately, you need a *root verification key* whose authenticity cannot be guaranteed by a certificate. Typically a set of root verification keys is installed in browsers/email programs. When you get messages like 'You do not trust this certificate' then there is no chain rooted in one of your root verification keys. In a system that requires certificates for signature verification you can use self-signed certificates to store root keys. A self-signed certificate can be verified with the public key contained in it.

Certificates have expiry dates. It is wrong to believe that a certificate cannot be used after it has expired. Deciding what should be done with expired certificates is a policy decision. In the world of passports, for example, an EU passport is valid for travel within the EU for a year after it has expired. There are two main policies for considering expiry dates and revocation status when evaluating a certificate chain.

In the *shell model* all certificates must be valid at the time of evaluation. If a top-level certificate expires or is revoked, all certificates signed by the corresponding private key have to be reissued under a new key. A CA should thus only issue certificates that expire before its own certificate. This model may fit when certificate subjects are addresses from a hierarchical address space and the purpose is *authenticating the source of traffic*. We want to be sure that the address is valid now but will not check again some time in the future. An address may become invalid due to a reorganization at some level of the address hierarchy. Hence, we should confirm that all links through the address space are valid at the time of authentication.

In the *chain model* the issuer's certificate has to be valid at the time the certificate was issued. If a top-level certificate expires or is revoked, certificates signed by the corresponding private key remain valid. This model may fit when certificates are used

for *authenticating documents*. A signed contract may have to be checked after the time a certificate has expired. The chain model also may also be appropriate with certain attribute certificates. For example, when a certificate confirms that its subject has student status it is important that the person issuing the certificate has the authority to do so. It is not required that this very person still has this authority when the certificate is presented later.

The chain model requires a time-stamping service that reliably establishes when a certificate was issued. A time stamp authority (TSA) is a TTP that provides proof of existence for a particular datum at an instant in time. A TSA does not check the documents it certifies. TSP, the PKIX time stamp protocol, is described in RFC 3161.

15.5.5 Revocation

A certificate may have to be revoked if a corresponding private key has been compromised or if a fact the certificate vouches for no longer is valid. Certificate revocation lists (CRLs) distributed at regular intervals or on demand are the solution proposed in X.509. CRLs make sense if on-line checks are not possible or too expensive. When on-line checks are feasible, CRLs can be queried on-line. But when on-line checks are feasible, certificate status can be queried on-line and CRLs may become superfluous. Instead, the current status of a certificate could be queried with a protocol like the Online Certificate Status Protocol (OCSP, RFC 2560). The German electronic signature infrastructure, for example, requires checks against positive lists of valid certificates. Short-lived certificates are an alternative to revocation.

15.5.6 Electronic Signatures

Hand-written signatures play an important part in commercial and legal transactions. (Not because they are difficult to forge, but because they signal intent.) When such transactions are executed electronically, an equivalent of hand-written signatures is required. Digital signatures have been proposed as the solution. However, a digital signature is just a cryptographic mechanism for associating documents with verification keys. The security service that associate documents with persons is usually called *electronic signature*. Electronic signature services often use digital signatures as a building block, but could be implemented without them.

There have been numerous efforts to integrate electronic signatures into legal systems. A prominent example is the EU Electronic Signature Directive (Directive 1999/93/EC of 13 December 1999 on a Community framework for electronic signatures). The Directive uses electronic signatures as a technology-neutral term, but so-called *advanced electronic signatures* have de facto to be implemented with digital signatures. Advanced electronic signatures are vouched for by *qualified certificates*. Further requirements on certification service providers (CAs and the like) and signature creation devices (e.g. smart cards)

apply. Figure 15.6 gives a schematic overview of all the components of a secure electronic signature service, and of the basis for their security.

Lesson
In the end, cryptographic protection has to be anchored in a non-cryptographic base.

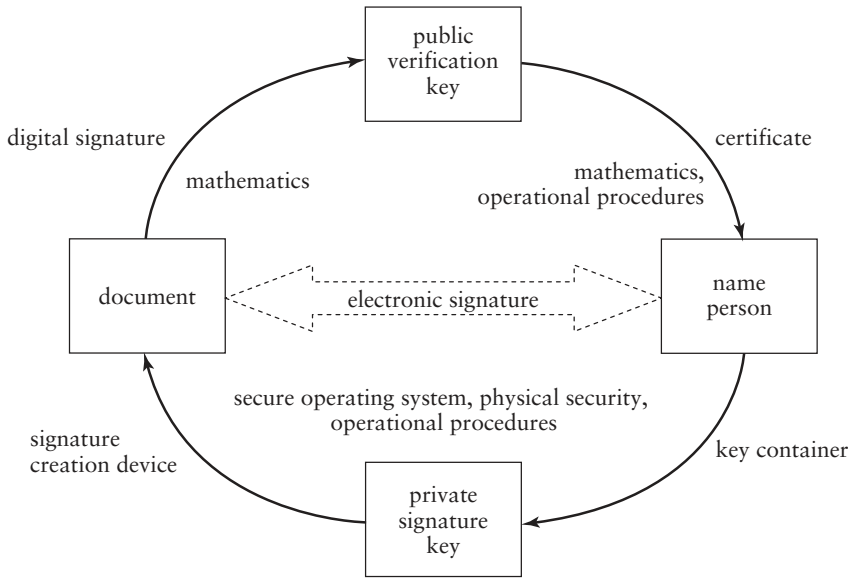


Figure 15.6: Components of an Electronic Signature Service Using Digital Signatures

15.6 TRUSTED COMPUTING – ATTESTATION

Let us switch the focus back from people to machines. There may be situations where we might only engage in a transaction with a remote platform if we know the exact hardware and software configuration of that machine. We might request this information from the remote platform, but how can we be sure that the information we get is correct? This issue has been addressed by the Trusted Computing Group (TCG). The process of vouching for the accuracy of information is called *attestation*.

The Trusted Platform Module (TPM) is the hardware component at the core of the security architecture specified in the TCG. The TPM can store integrity check values for hardware and software components in platform configuration registers (PCRs).

Furthermore, the TPM holds an endorsement key (EK) that cannot be removed. The EK is a 2048-bit RSA key pair installed by the manufacturer. The public key identifies the TPM. Attestation uses the private key for signing.[1] The TPM manufacturer may issue a certificate confirming that the EK belongs to a genuine TPM. In practice, though, such certificates are rarely issued.

The TPM could sign the PCR contents with its EK, but if all attestations are signed by the same key, an observer could link them all. The TPM may therefore create fresh signature key pairs, so-called attestation identity keys (AIKs), to make attestations unlinkable. The TPM needs the services of a *privacy CA* (pCA) to get a certificate that confirms that the AIK belongs to a genuine TPM. The following protocol was once considered for obtaining such a certificate. The TPM sends its public endorsement key EK and the public part of the attestation identity key AIK_i to the pCA. The pCA checks that EK belongs to a genuine TPM, stores the mapping between EK and AIK, and returns a certificate Cert_{pCA} to the TPM. In an attestation the TPM signs the PCR contents with the private part of AIK_i and includes Cert_{pCA} in the message sent to the verifier.

1. TPM $\rightarrow$ pCA: EK, AIK_i
2. pCA $\rightarrow$ TPM: Cert_{pCA}
3. TPM $\rightarrow$ Verifier: $\text{AIK}_i, \text{sAIK}_i(\text{PCR}), \text{Cert}_{pCA}$

You might have noted that the attempt to make attestations unlinkable has failed. In the first message all attestation keys are linked to EK, and thus all attestations can still be linked.

Direct Anonymous Attestation

Completely anonymous attestation is not desirable. It should be possible to recognize attestations from TPMs that are known to be compromised. Several advanced cryptographic techniques are used in the construction of a direct anonymous attestation protocol that provides unlinkable attestation for 'good' TPMs while being able to detect the attestations from known 'bad' TPMs [45, 51]. A detailed description of this protocol is beyond the scope of this book. We will just sketch some of the ideas that have contributed to its design.

Blind signatures had been introduced by Chaum for e-cash protocols [61]. In a blind signature scheme, a user lets someone else sign a document without revealing the document to the signer. Blind signature schemes can be implemented with RSA. Let (n, e) be the signer's public key, d the private key, and m the document to be signed. The user generates a random blinding factor r with $\gcd(r, n) = 1$ and sends $x = r^e m \bmod n$ to the signer. The signer returns

$$t = x^d = (r^e m)^d = r m^d \bmod n.$$

[1] This is in contrast to the TCG Glossary, which states that the private key is used for decrypting messages sent to the TPM.

The user finally computes the signature s of m as $s = r^{-1}t \bmod n$. In *group signature schemes* signers can remain hidden in a group. Only group members can generate valid signatures. Anyone can verify a signature, but not the individual signer. In the context of attestation, the group is formed by all valid TPMs. In a *zero-knowledge proof* the prover demonstrates knowledge of a secret without revealing any information about the secret. Such a secret could, for example, be a discrete logarithm.

The TPM gets from its issuer a Camenisch–Lysyanskaya signature [52] on its secret x, without revealing that secret to the issuer. This signature is a certificate that x belongs to a TPM. When signing a message, e.g. an AIK, the TPM shows in a zero-knowledge proof that it has such a certificate. The TPM further picks a generator g from a group where the DLP is hard and constructs a pseudonym $N_V = g^x$, together with a zero-knowledge proof that N_V has been properly formed with the secret x vouched for by the certificate. The message being signed is an input parameter to this zero-knowledge proof, used both by signer and verifier.

To detect a compromised TPM the verifier gets g and compares N_V with g^x for all secrets x known to belong to a compromised TPM. When g is chosen at random, pseudonyms cannot be linked. When g is fixed, all pseudonyms can be linked. When g is derived as a hash from the verifier's name, pseudonyms for different verifiers cannot be linked but pseudonyms used with an individual verifier can be linked. A verifier might find it suspicious if the same TPM is used too often and check how often a pseudonym is being presented.

15.7 FURTHER READING

A detailed specification of Kerberos is contained in the Internet RFC 4120. An analysis of Kerberos security, in the context of the environment for which it was developed, is given in [30]. Extensions to Kerberos refine its access control features, e.g. through *privilege attribute certificates* (PACs) in SESAME [15], OSF DCE, or PERMIS. The PKIX roadmap [13] and the Internet Draft on the Simple Public Key Infrastructure [88] provide a discussion of the theory of certificates worth reading. Lessons learned while deploying a PKI in an international company are reported in [3].

15.8 EXERCISES

Exercise 15.1 In the HTTP basic authentication protocol, analyze the security gains (if any) when the client sends a hash of the password instead of a base64 encoding of the password.

Exercise 15.2 Show that the AKEP2 protocol provides mutual entity authentication and implicit key authentication.

Exercise 15.3 Show that the MVQ protocol provides mutual entity authentication and key authentication. Justify the claim that the MVQ protocol is not vulnerable to the same man-in-the-middle attack as the Diffie–Hellman protocol.

Exercise 15.4 Consider this simple password-based challenge-response protocol run between a user A and a server S. P_A denotes A's password, n is a random nonce generated by the server, and h is a known cryptographic hash function.

$$\begin{aligned} &1.\ S \rightarrow A: \quad eP_A(n) \\ &2.\ A \rightarrow S: \quad eP_A(h(n)) \end{aligned}$$

Show that this protocol is vulnerable to an off-line password guessing attack.

Exercise 15.5 Justify the claim that the EKE protocol is not vulnerable to an off-line password guessing attack.

Exercise 15.6 Modify the Needham–Schroeder key exchange protocol so that both parties A and B can contribute input to the generation of the session key.

Exercise 15.7 Design an extension of Kerberos that allows access between domains. What administrative arrangements have to be in place to make such a scheme feasible? What additional steps would you introduce in the protocol?

Exercise 15.8 A company keeps an archive of signed electronic records, together with the relevant certificates. Should the shell model or the chain model be used when checking the validity of documents in the archive?

Exercise 15.9 Consider a scheme where certificates are used to delegate access rights. Should the shell model or the chain model be used when deciding on access requests?

Exercise 15.10 Consider a scheme where certificates are used to delegate access rights. Should a subject be able to delegate rights it cannot exercise itself? In your answer, distinguish between name-centric and key-centric PKIs.

Chapter 16

Communications Security

As long as data are within a machine, you can place a reference monitor as a guard and rely for protection on classic measures from computer security. When data leaves the machine, protection has to be extended along the connection link to the next safe place. This is the task of communications security. Cryptography plays an important role in implementing communications security services. The recipient of cryptographically protected data may have to do some processing before being able to decide whether incoming traffic is junk to be discarded. Denial-of-service attacks exploit this by forcing the victim to perform 'expensive' computations. The design of communications security protocols has to be aware of this issue.

OBJECTIVES

- **Give an overview of the security challenges specific to communications systems.**
- **Introduce the design of network security protocols, using the basic Internet security protocols IPsec and SSL/TLS as examples.**
- **Understand that the Internet is not a cloud.**
- **Explain why tunnels are sometimes placed within another tunnel.**

16.1 INTRODUCTION

Computer networks are the communications infrastructure for transmitting data between nodes in a distributed system. Data to be sent by an application in one node have to prepared for transport, transmitted as a sequence of electronic or optical signals, and reassembled and presented to an application program at the receiver's end. Network protocols have to find a route from sender to receiver, deal with the loss or corruption of data, and also with the loss of connections, e.g. when builders cut through a telephone cable. It is good engineering practice to address these concerns one at a time and use a layered architecture, with application protocols at the top and protocols that physically transmit encoded information at the bottom.

The ISO/OSI security architecture [125] defines *security services* to combat communications security threats. Security services are implemented by *security mechanisms*. The mechanisms providing these services mostly come from cryptography. Typical examples are encryption, digital signatures, and integrity check functions. Cryptographic protection has a nice property: a secure protocol in layer N will not be compromised when it is run on top of insecure protocols at the layers below. There is one exception to this rule. When your goal is anonymity and you take precautions to hide the identities of participants in one layer, then the data added by lower-layer protocols may still reveal information about the source and destination of messages.

16.1.1 Threat Model

Attackers have access to the communications link between two end points. Messages can be seen and modified by anyone bent on doing so. The job of a communications security service is done once data has been delivered to an end point. This is the 'old' secret service threat model. It is captured in formal models for protocol analysis that put the attacker in charge of all communications. In a metaphor from the world of pen and paper, the sender writes a note on a piece of paper and drops it on the floor. Later, the receiver goes through his garbage bin to see what has arrived.

The attacker can be *passive* or *active*. A passive attacker just listens to traffic. When the attacker is interested in the content of messages, we talk about *eavesdropping*, *wiretapping*, or *sniffing*. *Traffic analysis* tries to identify communications patterns and may be possible even when the attacker cannot read individual messages. An attacker might try to identify messages coming from the same source (linkability) or find out who is talking to whom, and how often. In mobile services, the attacker might also be interested in a user's location.

An active attacker may modify messages, insert new messages, or corrupt network management information such as the mapping between DNS names and IP addresses. In *spoofing attacks* messages come with forged sender addresses. In *flooding* (*bombing*) attacks a large number of messages is directed at the victim. In *squatting* attacks, the

attacker claims to be at the victim's location. Active attacks are not necessarily more difficult to mount than passive attacks. For example, in practice it is much easier to send an email with a forged sender address than to intercept an email intended for someone else.

16.1.2 Secure Tunnels

A secure tunnel (channel) is a secure logical connection between two end points that crosses an insecure network. Typical security guarantees are data integrity, confidentiality, and data origin authentication. The end points might be machines named by IP addresses or by domain names; the end points might be specific software components hosted at a client or server. Confusion about the precise nature of the end point authenticated can lead to 'security services that do not provide any security at all'. If the tunnel does not end where the user expects, the attacker may wait at the other side of the tunnel.

Secure tunnels do not provide security services once data are received. Secure tunnels are usually not intended for providing a non-repudiation service. They are mainly a matter for their immediate end points. Secure tunnels tend to be built in the following steps:

- An authenticated key establishment protocol establishes a fresh, shared secret between the end points; sometimes only one party is authenticated, sometimes mutual authentication is performed.
- In a key derivation phase, symmetric keys for message authentication code (MAC) and traffic encryption are derived from the shared secret.
- Further traffic is protected using the derived keys.

Expensive asymmetric cryptography is only used for entity authentication and key establishment. (Keyed) pseudo-random functions, typically built from standard cryptographic hash functions, are employed in key derivation. Symmetric key algorithms (MAC, encryption) are used for protecting data sent through the tunnel. Non-cryptographic mechanisms are nonces and time stamps used for freshness in entity authentication, and sequence numbers for preventing replay attacks. Fast rekeying and reuse of suspended sessions are supported in some cases.

16.2 PROTOCOL DESIGN PRINCIPLES

The seven-layer model of the ISO/OSI architecture (Figure 16.1) is a familiar framework for layering network protocols. Layered models provide a useful abstraction for discussing network security. Layered models also return us to a topic familiar from Section 3.4.2. Security services at the top can be tailored to a specific application. However, each application needs its own security protocols. Security services at the bottom can protect traffic from all higher layers, relieving application protocol designers from security concerns. However, some applications may find that this protection does not meet their requirements too well.

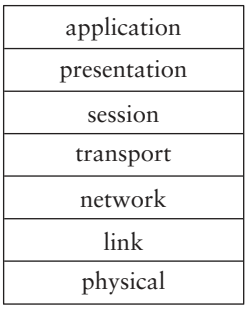


Figure 16.1: The ISO/OSI Seven-Layer Model

In a layered model, *peer entities* in a layer N communicate using an (N)-protocol. Protocols at layer $N + 1$ see a virtual connection at layer N and need not consider aspects of any lower layer (Figure 16.2). The (N)-protocol in turn builds on protocols from lower layers. There exists a general pattern for passing data to the lower layers. Messages in the (N)-protocol are called (N)-*protocol data units* (PDUs). The (N)-protocol transmits an (N)-PDU by invoking *facilities* at layer $N - 1$. At this stage, the (N)-PDU may be fragmented and otherwise processed. The results are then equipped with headers and trailers to become $(N - 1)$-PDUs. The recipient of these $(N - 1)$-PDUs uses information from the headers and trailers to reassemble the (N)-PDU. Figure 16.3 gives a simplified view of this process.

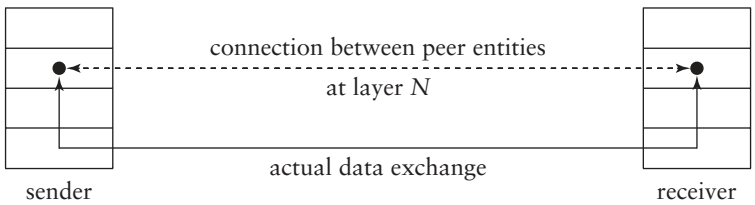


Figure 16.2: Virtual Connection at Layer *N*

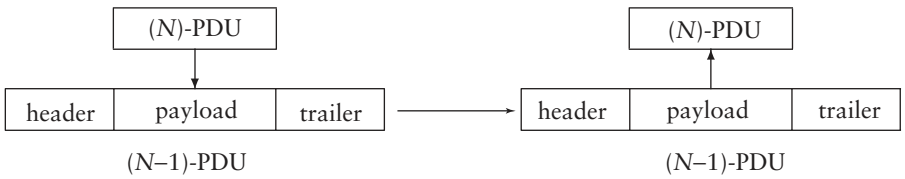


Figure 16.3: Processing an (N)-PDU

There are two principal options for implementing security services at layer $N-1$ that are called by an (N)-protocol. The upper-layer protocol can be *aware* of the security services at the lower layer, or the security services could be *transparent*. In the first case, the upper-layer protocol has to change its calls so that they explicitly refer to the security facilities provided. In the second case, the upper-layer protocol does not have to change. In both cases, the headers in the $(N-1)$-PDU are a convenient location for storing security-relevant data.

The Internet protocol stack has four layers (Figure 16.4). At the application layer there are protocols such as Telnet, FTP, HTTP, Simple Mail Transfer Protocol (SMTP), or Secure Electronic Transaction (SET). Protocols at the transport layer are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). At the Internet layer, there is the Internet Protocol (IP). TCP and UDP use *port numbers* to indicate the application protocol a PDU belongs to. Common port numbers are 21 (FTP), 23 (Telnet), 25 (SMTP, sending email), 110 (POP3, collecting email), 143 (*imap*, collecting email), 80 (HTTP), 443 (HTTPS, secure web pages), or 53 (DNS, name look-ups). The protocols at the link (and physical) layer are specific to the network technology.

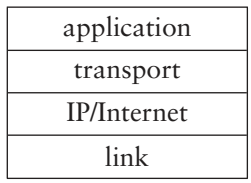


Figure 16.4: The Internet Layers

TCP and IP are at the heart of the Internet, together with UDP and the ICMP management protocol. Originally, these protocols were designed for friendly and cooperating users linked by an unreliable network, so security was no concern at all. Today, the use of TCP/IP is widespread and the demand for security is strong. The Internet Engineering Task Force (IETF) has standardized security protocols at the Internet and transport layer in RFCs (Requests for Comment). Within the IETF there are numerous ongoing activities regarding the revision of the existing security protocols and the development of new security protocols.

16.3 IP SECURITY

The Internet Protocol is a connectionless and stateless protocol that transmits IP packets (datagrams). These are the PDUs at the Internet layer. The core IP specification provides a best-effort service. Each packet is treated as an independent entity, unrelated to any other IP packet. There is no guaranteed delivery of packets, no mechanism for maintaining the

order of packets, and no security protection. IP version 4 was published as RFC 791 in 1981. Since then, the Internet has kept growing and, in consequence, IP has had to be adapted to cope with new demands. IP version 6 (IPv6) was specified in RFC 1883. We will refer to this version when discussing IP security mechanisms.

The security architecture for IP (IPsec) is given in RFC 4301. IPsec is optional for IPv4 and mandatory for IPv6. IPsec includes two major security mechanisms, the IP Authentication Header (AH) described in RFC 4302 and the IP Encapsulating Security Payload (ESP) covered in RFC 4303. The IP security architecture does not include mechanisms to prevent traffic analysis.

16.3.1 Authentication Header

The Authentication Header protects the integrity and authenticity of IP packets but does not protect confidentiality. In the 1990s, export restrictions on encryption algorithms were a political reason for having an authentication-only mechanism. These export restrictions have by and large been lifted and it is now recommended to use ESP only, to simplify implementations of IPsec. There is one important difference between the integrity protection afforded by AH and ESP. In ESP the original header of an IP packet is not considered when computing the authenticator (Figures 16.6 and 16.7). The integrity checksum in AH covers the original IP header, with the exception of mutable fields.

16.3.2 Encapsulating Security Payloads

Encapsulating Security Payload can be used to provide confidentiality, data origin authentication, data integrity, some replay protection, and limited traffic flow confidentiality. An ESP packet (Figure 16.5) contains the following fields:

- Security parameters index (SPI) – 32-bit field, used for identifying the *security association* for this IP packet.
- Sequence number – unsigned 32-bit field containing a monotonically increasing counter value; must be included by the sender but is processed at the receiver's discretion.
- Payload data – variable-length field containing the transport-layer PDU.
- Padding – optional field containing padding data for the encryption algorithm; the length of the data to be encrypted has to be a multiple of the algorithm's block size.
- Pad length.
- Next header – type of the transport-layer PDU.
- Authentication data – a variable number of 32-bit words containing an integrity check value (ICV) computed over the ESP packet without the authentication data.

The SPI and sequence number constitute the ESP header. The fields after the payload are the ESP trailer. ESP can be used in two modes. In *transport mode* (Figure 16.6), an upper-layer protocol frame, e.g. from TCP or UDP, is encapsulated within the ESP. The

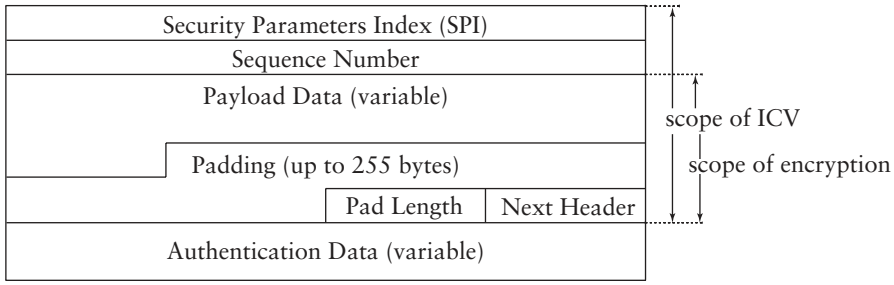


Figure 16.5: ESP Packet

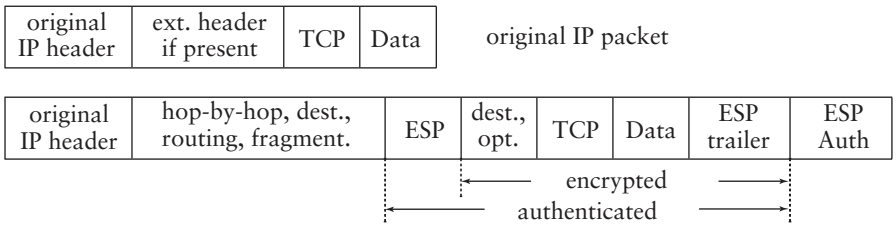


Figure 16.6: Applying ESP in Transport Mode to IPv6 Packet

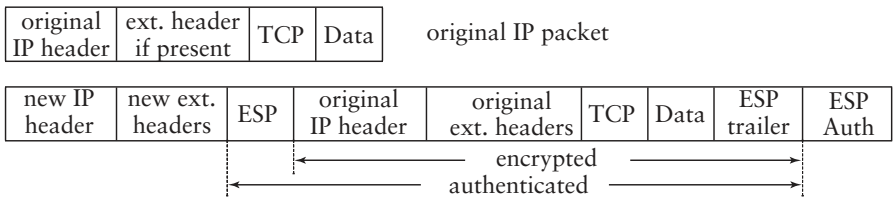


Figure 16.7: Applying ESP in Tunnel Mode to an IPv6 Packet

IP header is not encrypted. Transport mode provides end-to-end protection of packets exchanged between two end hosts. Both nodes have to be IPsec aware.

In *tunnel mode* (Figure 16.7) an entire IP packet plus security fields is treated as a new payload of an outer IP packet. The original inner IP packet is encapsulated within the outer IP packet. IP tunnelling can therefore be described as IP within IP. This mode can be used when IPsec processing is performed at security gateways on behalf of end hosts. The end hosts need not be IPsec aware. The gateway could be a perimeter firewall or a router. This mode provides gateway-to-gateway security rather than end-to-end security. On the other hand, you get traffic flow confidentiality as the inner IP packet is not visible to intermediate routers and the original source and destination addresses are hidden.

16.3.3 Security Associations

To generate, decrypt, or verify an ESP packet a system has to know which algorithm and which key (plus initialization vector) to use. This information is stored in a *security association* (SA). The SA is the common state between two hosts for communication in one direction. Bidirectional communication between two hosts requires two SAs, one in each direction. Therefore, SAs are usually created in pairs.

A security association is uniquely identified by an SPI (carried in AH and ESP headers), source address, destination address and the security protocol (AH or ESP). It contains the relevant cryptographic data such as algorithm identifiers, keys, key lifetimes, and possibly initialization vectors. There is a sequence number counter and an anti-replay window. The SA also indicates the protocol mode (tunnel or transport). The list of active SAs is held in the security association database (SAD). SAs can be combined, e.g. for multiple levels of nesting of IPsec tunnels. Each tunnel can begin and end at different IPsec gateways along the route. Figure 16.8 shows a typical configuration where a remote host has security associations with a gateway and with an internal host.

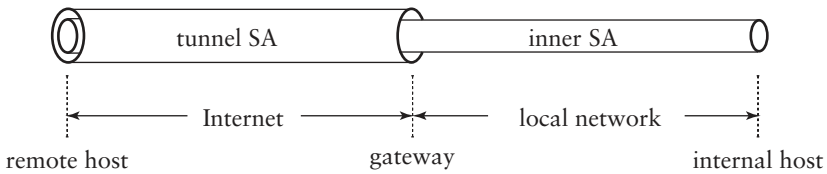


Figure 16.8: Combining Security Associations

16.3.4 Internet Key Exchange Protocol

Manual creation of SAs works only if the number of nodes is small but does not scale to reasonably sized networks of IPsec-aware hosts. An automated process supported by a key exchange protocol is thus needed. The first version of the Internet Key Exchange protocol (IKEv1, RFC 2409) was criticized as being too flexible and too complex, with too many options.

There is an underlying general strategic problem. For a security protocol to be widely used it has to be standardized. But a protocol has to be widely used to learn which features are essential and should be standardized, and which options are hardly ever used. Based on this experience, the standard can be revised. A new version of the Internet Key Exchange protocol, IKEv2, is specified in RFC 4306. IKEv2 establishes an IKE SA and a variable number of child SAs. It consists of two phases and three message exchanges, IKE_SA_INIT, IKE_AUTH (first phase), and CREATE_CHILD_SA. IKEv2 uses Diffie–Hellman as its single key establishment method.

Prudent security engineering advice in the mid 1990s recommended that all messages in a security protocol should be authenticated [2]. This defends against replay attacks where an attacker uses intercepted messages from one protocol run in another run. The underlying security paradigm is a closed organization. Parties want protection from outsiders. Inside the organization identities are readily revealed.

The Internet is not a closed organization. Internet users may not wish to disclose their identities to all and sundry all the time. Privacy concerns create a new requirement: neither an observer nor the communicating partner get evidence that proves that a conversation took place (*plausible deniability*). In a protocol, identities should then be hidden as long as possible. In this case, there can be no authentication. This is reflected in the design of IKEv2.

Lesson
Security requirements depend on the application.

IKE_SA_INIT Exchange

IKEv2 starts with an IKE_SA_INIT exchange between initiator and responder. Components in brackets are optional.

$$\begin{aligned} &1.\ I \rightarrow R: \quad HDR, SAi1, KEi, Ni \\ &2.\ R \rightarrow I: \quad HDR, SAr1, KEr, Nr, [CERTREQ] \end{aligned}$$

The parameters are the header HDR containing the SPI and other information, security associations SA, Diffie–Hellman values $KEi = g^i$, $KEr = g^r$ in the chosen group, nonces Ni, Nr, and an optional certificate request $[CERTREQ]$. The initiator states which cryptographic algorithms are supported ($SAi1$), the responder picks a suite ($SAr1$). Initiator and responder have thereby negotiated a shared but unauthenticated SA ($SAr1$) and can compute a shared but unauthenticated master key $SKEYSEED$ from the nonces and from g^{ir}. The shared suite of cryptographic algorithms and the shared key(s) are used to protect the messages in the next exchange. No identities are disclosed in the IKE_SA_INIT exchange, other than the IP addresses in the IP headers.

IKE_AUTH Exchange

In the IKE_AUTH exchange messages are cryptographically protected under a key SK derived from the master key. We denote encryption under SK by $SK\{\ldots\}$.

$$\begin{aligned} &1.\ I \rightarrow R: \quad HDR, SK\{IDi, [CERT,][CERTREQ,][IDr,]AUTH, SAi2, TSi, TSr\} \\ &2.\ R \rightarrow I: \quad HDR, SK\{IDr, [CERT,]AUTH, SAr2, TSi, TSr\} \end{aligned}$$

The parameters are the identities IDi, IDr of initiator and responder, the authenticator \em AUTH, and traffic selectors TS that are outside the scope of this book. The authenticator is a digital signature or MAC over the message. Authenticators must be

verified. In this exchange a new authenticated SA ($SAr2$) is negotiated. The shared suite of cryptographic algorithms from $SAr2$ and the shared keys are used to protect the messages in a third exchange. Identities are only included in encrypted messages.

CREATE_CHILD_SA Exchange

Messages are cryptographically protected with keys derived after the first phase.

$$\begin{aligned} &1.\ I \rightarrow R: \quad HDR, SK\{[N], SA, Ni, [KEi,][TSi, TSr]\} \\ &2.\ R \rightarrow I: \quad HDR, SK\{SA, Nr, [KEr,][TSi, TSr]\} \end{aligned}$$

The notify flag $[N]$ is used to indicate the SA when an SA is being rekeyed. This exchange negotiates SAs for AH and ESP. There are different SAs for each combination of

$$\{\text{ESP,AH}\} \times \{\text{tunnel,transport}\} \times \{\text{sender, receiver}\}.$$

Optionally new keys can be established for the CHILD_SA.

16.3.5 Denial of Service

Strong cryptography can be a weakness. An attacker may send bogus message to the victim, who performs 'expensive' cryptographic operations, e.g. signature verifications. Such denial-of-service attacks cannot be prevented. To mitigate their effect, protocols should be designed in a way that makes the attacker's cost comparable to the victim's cost.

A denial-of-service attack against the initiator could be launched during IKE_SA_INIT by responding to the first message with a bogus response. If the initiator uses the first response received to set up a connection, no usable SA and key will be established and IKE fails. As a defence, the initiator accepts multiple responses to its first message, continues the protocol, and discards all invalid half-open connections when a valid cryptographically protected response is received to any one of its requests.

A denial-of-service attack against the responder could send a flood of session initiation requests from forged IP addresses, exhausting the responder's state and CPU. As a defence, the responder could check that requests come from the claimed source address (authentication) before proceeding further. This check can be implemented by returning a *cookie* to the initiator who has to repeat the request, now including the cookie. IKE_SA_INIT with cookies works as follows:

$$\begin{aligned} &1.\ I \rightarrow R: \quad HDR(A, 0), SAi1, KEi, Ni \\ &2.\ R \rightarrow I: \quad HDR(A, 0), N(COOKIE) \\ &3.\ I \rightarrow R: \quad HDR(A, 0), N(COOKIE), SAi1, KEi, Ni \\ &4.\ R \rightarrow I: \quad HDR(A, B), SAr1, KEr, Nr, [CERTREQ] \end{aligned}$$

The field A in the header is the SPI assigned by the initiator, the field B the SPI assigned by the responder. To counter state exhaustion, the responder must use a stateless authentication mechanism. The cookie is not stored, but the responder must be able to check that a cookie received is valid. Cookies are only used locally, so they raise

no interoperability issues and there is no need to standardize cookie generation. The responder could use a local secret, changed regularly and shared with nobody else, when creating a cookie. RFC 4306 suggests computing the cookie as

$$VersionIDofSecret, Hash(Ni, IPi, SPIi, secret)$$

The responder may keep a window of secrets to be able to accept cookies that are slightly out of date.

16.3.6 IPsec Policies

IPsec policies determine the security processing that should be applied to an IP packet. IPsec-aware hosts have a security policy database (SPD). The SPD is consulted for each outbound and inbound packet. The fields in an IP packet are matched against fields in SPD entries. Figure 16.9 illustrates the processing of outbound IP packets. Matches can be based on source and destination addresses (and ranges of addresses), transport-layer protocol, port numbers, etc. Once an applicable policy has been found indicating an action to be taken, we have to check the security association database to see whether there is an SA for performing that action. For inbound IPsec-protected packets, for example, the search finds the entry that matches the longest SA identifier.

1. Search the SAD for a match on {SPI, destination address, source address}; if a SAD entry matches, process the inbound packet with that entry.
2. Otherwise, search the SAD for a match on {SPI, destination address}; if a SAD entry matches, process the inbound packet with that entry.

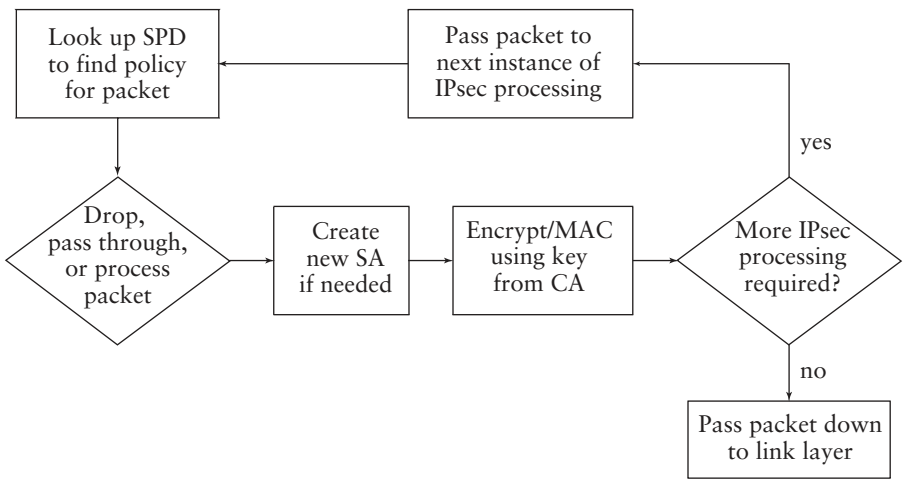


Figure 16.9: Processing Outbound IPsec Packets

3. Otherwise, search the SAD for a match on only {SPI} if the receiver has chosen to maintain a single SPI space for AH and ESP, or on {SPI, protocol} otherwise; if a SAD entry matches, process the inbound packet with that entry.
4. Otherwise, discard the packet and log an auditable event (a new SA may be needed).

16.3.7 Summary

IPsec provides transparent security for all users of IP, without changing the interface to IP. Upper-layer protocols need not be re-engineered to invoke security and need not even be aware that their traffic is protected at the IP layer (Figure 16.10). However, there is not much scope for tuning the level of protection to the requirements of the application. IP is concerned about its performance as a communications protocol and cannot spend much time on checking application-specific data to pick a security association. IPsec provides host-to-host security, but not user-to-user or application-to-application security. IPsec violates one of the original design principles of the layered Internet architecture. The network layer is supposed to be stateless and unreliable. However, the order of data in a secure channel may be crucial. This is difficult to maintain if IP packets are dropped or reordered. Moreover, parties protecting their traffic with IPsec now need to maintain shared state at the network layer.

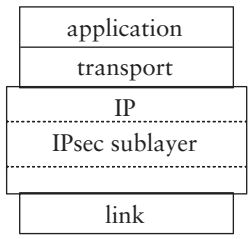


Figure 16.10: IP Security

16.4 IPSEC AND NETWORK ADDRESS TRANSLATION

Protocols at different Internet layers are intertwined in manifold ways. TCP and UDP compute (non-cryptographic) checksums that depend on IP source and destination addresses. AH and ESP compute cryptographic checksums over their payloads (TCP, UDP, ...). This works as long as the addresses in an IP header do not change on the way from sender to receiver. This assumption matches the metaphor of the Internet as a cloud. Packets enter the cloud at some point and emerge unchanged somewhere else, were it not for the potential interference by an attacker. However, the metaphor

is deceptive. The Internet is not a cloud, and there are entities that legitimately modify addresses in IP headers. We will briefly discuss the implications for IPsec.

Network address translation (NAT) was invented in order to cope with the shortage of IPv4 addresses (RFC 3022). NAT maps private IP addresses to routable addresses in the public network, reducing the need for global IP addresses. NAT is today seen as a security feature; local hosts are not directly addressable from the external network. Port address translation (PAT) modifies TCP/UDP source and destination ports. NAT and PAT recalculate IP and TCP/UDP header checksums.

Middleboxes such as routers, firewalls, NAT and PAT devices may rely on information that is not available when IPsec is used. Conversely, IPsec authentication may refer to fields in the IP header that are modified by a middlebox. For example, both in tunnel or transport mode, AH is incompatible with NAT. AH only works when source and destination networks are reachable without translation. ESP works with NAT. The outer IP header is not included in the hash value computation for authentication.

IKE has problems when NAT devices transparently modify outgoing packets. If incoming packets in an IKE negotiation are expected from UDP port 500 and if a NAT device is introduced, the actual packet port may not be the expected port. Thus IKE negotiation will not even begin. When IKE includes IP addresses as part of the authentication process, changes made by a NAT device will cause IKE negotiation to fail.

NAT traversal (NAT-T) (RFC 3947 and 3948) deals with the problems that arise when the source address of a packet does not match the true source IP address needed for cryptographic checks. NAT-T adds a UDP header that encapsulates the ESP header, i.e. sits between ESP header and outer IP header (Figure 16.11). NAT-T puts the sender's original IP address into a NAT original address (NAT-OA) payload during the IKE exchange. When processing a packet, the UDP header is a marker triggering an editing step that modifies the new IP header so that any disruptive NAT changes are undone and checksums are successfully verified.

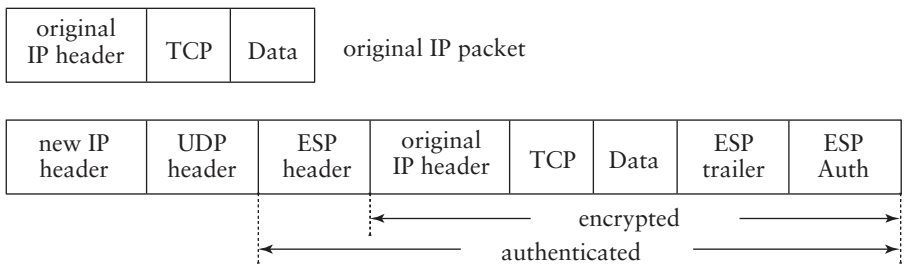


Figure 16.11: Tunnel Mode ESP Encapsulation in NAT-T

16.5 SSL/TLS

TCP provides a reliable byte stream between two nodes. TCP is a stateful connection-oriented protocol that detects when packets are lost, when packets arrive out of order, and discards repeated data. TCP performs address-based entity authentication when establishing a session between two nodes, but chose a vulnerable implementation of this protocol as highlighted in Section 17.1.2. TCP lacks strong cryptographic entity authentication, data integrity or confidentiality. These services were introduced in the *Secure Socket Layer* (SSL) protocol developed by Netscape, mainly to protect World Wide Web traffic. The *Transport Layer Security* protocol (TLS) is by and large identical with SSL version 3 (SSLv3) so the protocol has become known as SSL/TLS. The latest version at the time of writing, TLSv1.2, is specified in RFC 5246.

Within the IP stack, SSL sits between the application layer and TCP (Figure 16.12). Hence, SSL can rely on the properties guaranteed by TCP and, for example, need not concern itself with the reliable delivery of data. Like TCP, SSL is stateful and connection-oriented. The SSL *session state* contains information required for the execution of cryptographic algorithms, such as a session identifier, the specification of the cipher suite, shared secret keys, certificates, random values used by protocols, etc. To contain the overheads caused by key management, one SSL session can include multiple connections. The characteristic example is an HTTP 1.0 session between a client and a server, where a new connection is made to transfer each part of a composite document. Only a subset of the state information has to change for each connection.

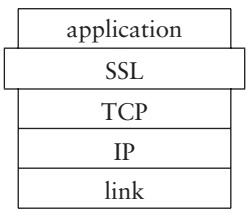


Figure 16.12: The SSL Security Layer

SSL has two components, the *SSL Record Layer* and the *SSL Handshake Layer*. The SSL Record Layer takes blocks from an upper-layer protocol, fragments these blocks into *SSL plaintext records*, and then applies the cryptographic transformation defined by the *cipher spec* in the current session state. The SSL Record Layer thus provides a service similar to IPsec. The parallels between IPsec security associations and the SSL state are by no means accidental.

The *SSL Handshake Protocol* sets up the cryptographic parameters of the session state.

1. Client → Server: ClientHello
2. Server → Client: ServerHello,[Certificate],[ServerKeyExchange], [CertificateRequest], ServerHelloDone
3. Client → Server: [Certificate], ClientKeyExchange, [CertificateVerify], ChangeCipherSpec, Finished
4. Server → Client: ChangeCipherSpec, Finished

To illustrate this protocol, we will step through a run where the client authenticates the server. The client initiates the protocol run with a *ClientHello* message, containing a 28-byte random number, a list of suggested ciphers, ordered according to the client's preference, and a suggested compression algorithm.

M1: ClientHello:

ClientRandom[28]
Suggested Cipher Suites: TLS_RSA_WITH_IDEA_CBC_SHA TLS_RSA_WITH_3DES_EDE_CBC_SHA TLS_DH_DSS_WITH_AES_128_CBC_SHA
Suggested Compression Algorithm: NONE

The server selects the cipher TLS_RSA_WITH_3DES_EDE_CBC_SHA from the suggested suite. RSA will be used for key exchange, triple DES in CBC mode for encryption, and SHA as the hash function. The server replies with a *ServerHello* message and a certificate chain.

M2: ServerHello:

ServerRandom[28]
Use Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA
Session ID: 0xa00372d4XS

Certificates:

subjectAltName: SuperStoreVirtualOutlet PublicKey: 0x521aa593 . . . Issuer: SuperStoreHQ
subjectAltName: SuperStoreHQ PublicKey: 0x9f400682 . . . Issuer: Verisign

Server Done:

NONE

In our example, no certificate is requested from the client. The client verifies the certificate chain referring to the *subject alternative name* extensions in the certificates, and then locally creates a random 48-byte *PreMasterSecret*. The *MasterSecret* is the first 48 bytes of

PRF(PreMasterSecret, "master secret", ClientRandom || ServerRandom).

Here, PRF is shorthand for a more complex function based on MD5 and SHA that takes as inputs a secret, a label and a seed. (The symbol || denotes concatenation.) The *MasterSecret* serves as input to the construction of a *key block* of the form

PRF(MasterSecret, "key expansion", ClientRandom || ServerRandom).

All required MAC and encryption keys for client and server are extracted from the key block. They keys protecting traffic from client to server are different from the keys protecting traffic from server to client. Thus, the parties can easily distinguish between messages they send and messages they receive, and they are not subject to *reflection attacks* where a message is replayed to its sender.

The client now transmits the *PreMasterSecret* to the server, using the key management algorithm specified in the selected cipher suite and the server's certified public key. The client should then immediately destroy the *PreMasterSecret*. In our example, the algorithm is RSA and the public key is 0x521aa593 . . . . The ChangeCipherSpec message indicates that subsequent records will be protected under the newly negotiated ciphersuite and keys. The client then ties the third message to the first two through two hashes constructed with MD5 and SHA.

M3:	A: ClientKeyExchange:	RSA_Encrypt(ServerPublicKey,PreMasterSecret)
	B: ChangeCipherSpec:	NONE
	C: Finished	MD5(M1 || M2 || M3A)
		SHA(M1 || M2 || M3A)

The server decrypts the *PremasterSecret* and computes from it the *MasterSecret*, the *key block*, and all derived secret keys valid for this session with the client. The server verifies the hash appended to the client's message and replies as follows.

M4	A: ChangeCipherSpec:	NONE
	B: Finished	MD5(M1 || M2 || M3A || M3C)
		SHA(M1 || M2 || M3A || M3C)

The client verifies the hash in the server's message. Both parties have now established shared secret keys which they can use to protect application traffic.

16.5.1 Implementation Issues

SSL/TLS is a mature security protocol that has been thoroughly analyzed. Formal proofs of its properties provide a high degree of assurance that it achieves its stated goals.

However, implementing a secure protocol securely is still a formidable challenge. We will look at the three areas where mistakes have been made.

SSL/TLS makes use of random numbers during the handshake protocol but also when creating the keys for clients and servers. Flaws in the random number generation can then undermine the security of the protocol. In an early implementation of SSL, a low quality pseudo-random number generator had led to predictable session keys [102]. More recently, Debian's OpenSSL had a serious security problem.[1] There was insufficient randomness in the generation of public/private key pairs. The flaw had been introduced in May 2006 and was discovered in May 2008. The cause of the flaw was a tool-based code analysis looking for software security vulnerabilities. The tool had flagged a read from uninitialized memory in an OpenSSL library. The offending instruction was commented out. Normally, reading from uninitialized memory is a programming mistake. In this case, uninitialized memory was intentionally used as a source of randomness. Removing the offending instruction removed a main source of randomness in key generation.

Error messages can help to decipher encrypted messages. When an error message, or the length of an error message, or the time an error message is sent, depends on the state the internal processing of a message has reached, the attacker may learn something about the message being processed.

Response times by a server that differ in the event of a padding failure or of a MAC failure, in conjunction with an analysis of padding method for CBC mode have led to the recovery of SSL protected passwords [54]. Timing attacks have been used to discover private keys. An attacker in the same LAN segment as an OpenSSL server can derive the server's private key from its response times [48].

Note that these are attacks against particular implementations but not against the protocol at the conceptual level. Insecure application of SSL/TLS will be discussed in Chapter 18.

16.5.2 Summary

In the SSL/TLS handshake protocol, client and server agree on a cipher suite, establish the necessary keying material, and authenticate each other. Today, SSL is the most widely used Internet security protocol, supported by all major web browsers. SSL adds a security layer between application protocols and TCP, so applications explicitly have to ask for security. Thus, application code has to be changed, but the required changes are not much more than edit operations, e.g. replacing a TCP **connect** call in the pre-SSL application with an **SSL connect** call. The **SSL connect** call will initialize the cryptographic state parameters and make the original TCP **connect** call.

[1] Debian Security Advisory 1571: OpenSSL Predictable Random Number Generator, May 2008.

Client and server have to protect the parameters of the security contexts (or IPsec security associations) they have established. Otherwise, the security provided by SSL (or IPsec) will be compromised. This brings us back to computer security.

Lesson
Cryptographic protection cannot be compromised from the layer below in the communications network. It can, however, be compromised from the layer below in the operating system of a network node.

16.6 EXTENSIBLE AUTHENTICATION PROTOCOL

The Extensible Authentication Protocol (EAP) defines authentication protocols at the level of abstract message flows called *methods*. Methods can be implemented with a variety of underlying mechanisms. Thus, the authentication mechanism can be chosen independently of the application and can be changed without having to rewrite the application. The application calling an authentication protocol just sees the abstract message flow. Other examples of this approach are the SAML profiles for web services (Section 18.7.2) and previously the General Security Services API (GSS-API).

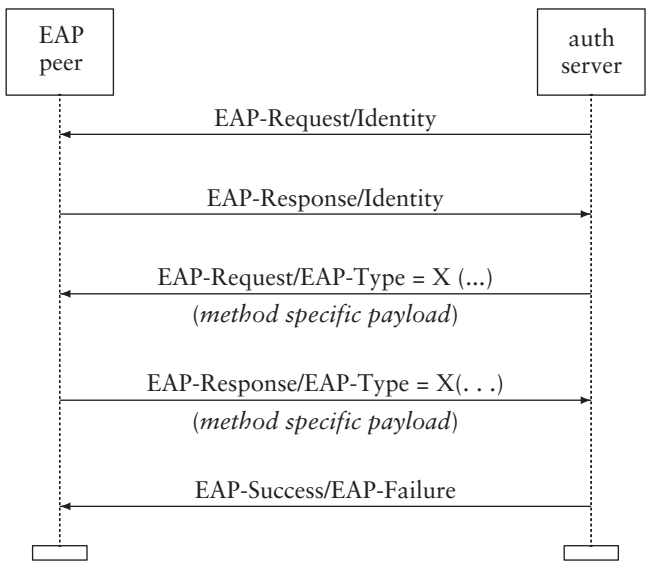


Figure 16.13: EAP Generic Message Flow

EAP has been specified in IEEE 802.1X and RFC 3748. It has its origin in WLAN authentication and has been proposed by the 3rd Generation Partnership Project for WLAN interworking (e.g. EAP-AKA). Windows supports various EAP methods for remote access. Figure 16.13 shows the generic EAP message flow. The identity exchange messages can be skipped if the peer's identity is already known. An EAP method can have several request/response rounds. Generic EAP messages exchange identities and encapsulate authentication protocol messages. Users are identified by their Network Access Identifier (NAI). These identifiers are specified in RFC 4282. EAP methods provide at least one-way EAP peer authentication to EAP authentication servers. Many EAP methods have been proposed to meet a variety of application security requirements.

EAP Tunnelled TLS (EAP-TTLS) illustrates how EAP is being deployed today. EAP-TTLS is intended for a setting where a user connects to a server from a client machine. The server has a certificate. The client uses TLS to authenticate the server and establish

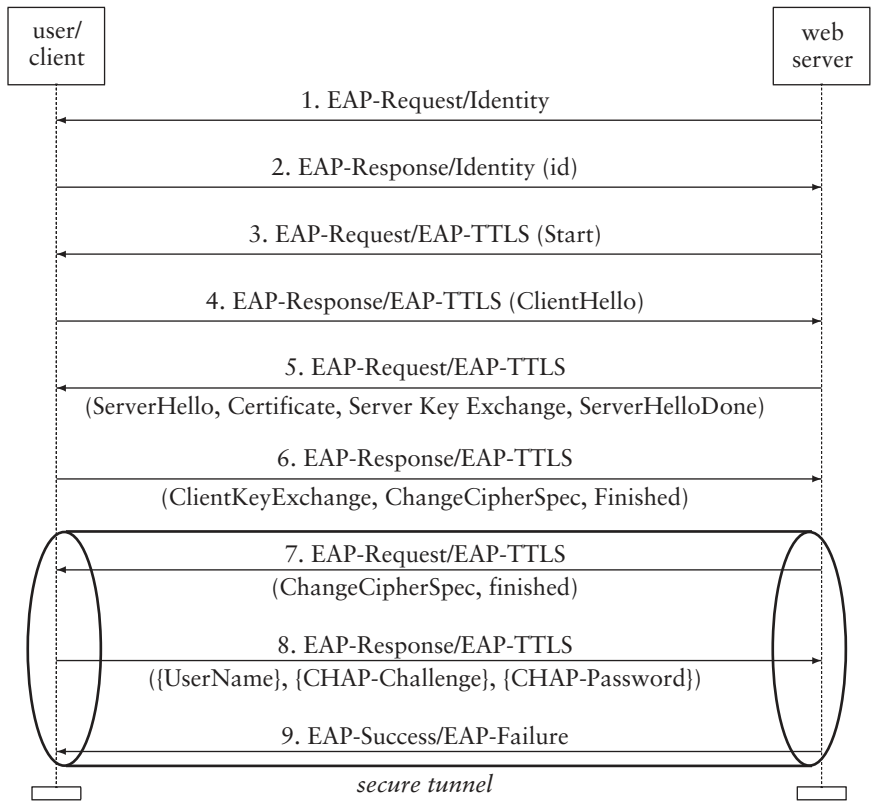


Figure 16.14: EAP Tunnelled TLS: EAP-TTLSv0 with CHAP

a secure tunnel to the server. Users are authenticated by password at the server, e.g. using the CHAP protocol (RFC 1994). EAP-TTLSv0 is specified in RFC 5281 and supports inner legacy methods (such as CHAP). Most computationally expensive tasks are located at the server. In the first round of EAP exchanges, the TLS protocol is run with unilateral authentication of the server (steps 3 to 6 in Figure 16.14). From step 7 on, messages are sent in a secure TLS tunnel. The final EAP round (steps 7 and 8) executes password-based authentication. EAP-TTLS prevents man-in-the-middle attacks under the assumption that the TLS tunnel has been established with the intended server. This assumption will be investigated in Chapter 18.

16.7 FURTHER READING

This chapter has only briefly sketched the issues and techniques in communications security. The best source of material on Internet security is of course the Internet. The website of the Internet Engineering Task Force is `http://www.ietf.org`. Documents on IPsec can be found at `http://www.ietf.org/html.charters/ipsec-charter.html`. The IETF specifications are, like networking technology, continuously changing. This is very much the case for work on Internet security, so you should not be disappointed if some aspects of IPsec or SSL/TLS have changed by the time you read this book.

16.8 EXERCISES

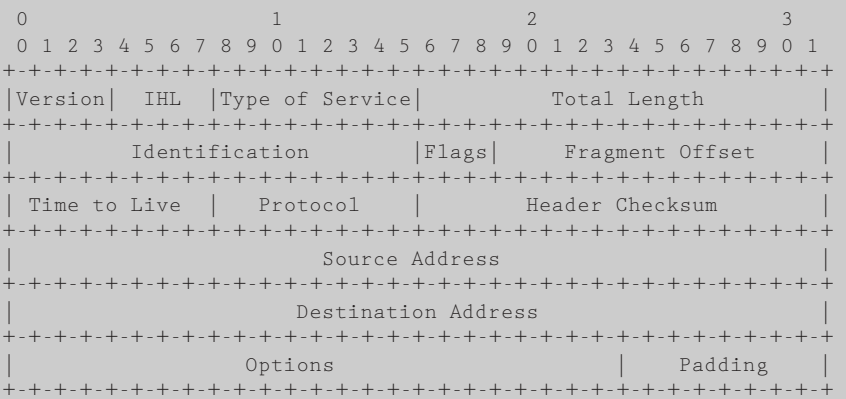
Exercise 16.1 The 32-bit sequence numbers in AH and ESP headers may be inconveniently short in the case of heavy traffic. Sequence numbers must not be reused and, when the available numbers are exhausted, rekeying is triggered. Describe a solution that extends the range of sequence numbers without adding to the size of the AH and ESP headers.

Exercise 16.2 For IPsec and SSL, the nodes running the protocol are assumed to be secure. What additional security mechanisms do you need at these nodes to make this assumption true?

Exercise 16.3 Parties that have established a secure tunnel may wish to switch to new session keys. Suggest a rekeying protocol that runs faster than the key establishment protocol used when setting up the tunnel. Justify the design decisions taken.

Exercise 16.4 IKEv2 has specific rekeying sub-protocols. TLS performs a new handshake for rekeying. Examine the differences in the overheads and security properties of these two strategies.

Exercise 16.5 An IPv4 header has the following format:



An attacker has intercepted an IPv4 packet protected by ESP in tunnel mode for confidentiality only (the confidentiality of the inner header and payload are protected). The attacker happens to know the inner destination address the packet was sent to. Encryption uses a 64-bit block cipher in CBC mode. How can the attacker use the gateway at the end of the tunnel in an attempt to learn the payload? What are the chances of the attack succeeding? For your analysis, assume that IPv4 only checks the header checksum. Would a switch to a 128-bit block cipher defeat your attack?

Exercise 16.6 A gateway processes a packet that is protected by ESP in tunnel mode and finds an error after decryption when checking the inner packet. Examine the security issues that arise when constructing the error message the gateway replies with.

Exercise 16.7 Routers fragment IP packets when the packet size exceeds the maximum transmission unit (MTU) of the outgoing interface. Adding IPsec headers may cause IP fragmentation. Path MTU discovery (PMTUD) dynamically determines the MTU between two end points:

- The source first sets the path MTU to the (known) MTU of the first hop and sends packets with the 'don't fragment' (DF) bit set.
- If a router along the path would have to fragment a packet, it discards the packet and sends an ICMP destination unreachable message to the source indicating 'fragmentation needed and DF set'.
- The source then reduces the path MTU and tries again.

Assume that all traffic from a gateway to a given destination subnet is protected by ESP in tunnel mode. A source in the gateway's subnet runs PMTUD. The gateway

copies the DF bit from the original IP packet into the IPsec header. Assume further that an intermediate router in the backbone requires fragmentation of the IPsec-encapsulated packet. The DF bit is set on the packet so it cannot be fragmented. The backbone router must drop the packet and send an ICMP unreachable message back to the source. However, the source address in the path MTU message is that of the IPsec end point (the gateway). The path MTU message carries an SPI pointing to an IPsec SA, but a single SA is used for all hosts on the subnet, so the gateway does not know which host to send the path MTU message to.

How would you solve this problem?

Exercise 16.8 Let there be two NAT devices behind a security gateway. The two NAT devices assign the same local address to two different machines. What problem will occur at the gateway when both machines run IPsec with NAT-T? How would you address this problem?

Chapter 17

Network Security

Secure tunnels can be built across insecure networks. Devices in the network may, however, turn out to be stumbling blocks where tunnels collapse. This chapter will look at devices on the Internet. You need Domain Name System (DNS) servers to look up the current address of a host you want to connect to. Firewalls and intrusion detection systems provide non-cryptographic security services. We will switch to a network perspective on security. The attacker is not in control of the network. The attacker has access to the network from a node in the network, and only sees traffic that passes that node.

OBJECTIVES

- Give an overview of the security challenges specific to networks.
- Examine the security of the Domain Name System.
- See how network boundaries can serve as security perimeters.
- Understand the principles and limitations of firewalls and intrusion detection systems.

17.1 INTRODUCTION

An attacker may try to learn about the internal structure of your network and use this information to launch an attack. Information gleaned from network management protocols that collect diagnostics about the load and availability of nodes can become security-sensitive. At the same time, these protocols are needed to use the network efficiently. By being overprotective, you can easily diminish the quality of services provided by a network.

Access control in a network regulates how traffic may cross network boundaries. Firewalls implement access control at the network level. Network managers want to know whether their defences are effective, whether their network is under attack, and what attacks are currently faced. Intrusion detection systems provide this information. In protocol design, you might further find that entities in the network, such as firewalls or network address translators, get in the way of Alice and Bob running their protocol, and need special attention.

17.1.1 Threat Model

The security protocols in Chapter 15 were described at an abstract level. Messages travelled directly between principals and we did not consider the precise nature of this exchange. The Internet is often described as a cloud, where all internal details are hidden. Such an abstract model is not necessarily always the best way to address security issues. You might assume a less powerful and potentially more realistic adversary.

Botnets are characteristic of the threats you are facing from the Internet. A *bot* (*drone*) is a program that receives commands from a *bot controller*. A malicious bot controller may install bots surreptitiously on the machines of unwitting Internet users, communicate with the bots using protocols such as Internet Relay Chat (IRC) and HTTP, and launch spam or denial-of-service attacks from the bots. Botnets illustrate the principle of attack *amplification*.

To take out a botnet, you have to take out the bot controller, e.g. by blocking its IP address. In *fast flux networks*, the bots know their controller by its domain name. A bot controller then just moves to a new IP address when its current address is blocked. In *fast domain flux networks*, the domain name of the bot controller can be changed dynamically. The bot controller registers a new domain name when its current name is blacklisted.

Botnet attacks do not target communications links. On the other hand, you must no longer assume that the end points of links are safe harbours. To reason about botnet attacks and defences, you do not have to consider an adversary that is in charge of the entire Internet. The *net adversary* is a malicious network node which has the capacity to

- read messages directly addressed to it,
- spoof arbitrary sender addresses,
- try to guess fields sent in unseen messages.

The next two sections show attacks by the net adversary.

17.1.2 TCP Session Hijacking

To open a TCP session with a server B, a client machine A initiates the following three-way handshake protocol.

$$\begin{array}{ll} 1.\ A \rightarrow B: & \text{SYN, ISSa} \\ 2.\ B \rightarrow A: & \text{SYN|ACK, ISSb, ACK(ISSa)} \\ 3.\ A \rightarrow B: & \text{ACK, ACK(ISSb)} \end{array}$$

SYN and ACK indicate that respective bits have been set. ISSa and ISSb are 32-bit sequence numbers. The acknowledgements are computed as ACK(ISSa) = ISSa+1 and ACK(ISSb) = ISSb+1. Under the assumptions made above, an attacker trying to impersonate A has to guess the sequence number ISSb sent to A. This protocol is then secure as long as the sequence numbers are sufficiently random.

However, RFC 793 specified that the 32-bit counter be incremented by 1 in the low-order position about every 4 microseconds. Even worse, Berkeley-derived Unix kernels incremented the counter by 128 every second, and 64 for each new connection. There is not much randomness left to confound an attacker.

The attack made possible by such implementation decisions was described as early as 1985 [174] and later generalized [29]. The attacker M first opens a genuine connection to its target B and receives a sequence number ISSb. The attacker then impersonates A, sending a packet with A's address in the source field,

$$M(A) \rightarrow B\text{: SYN, ISSc.}$$

B replies to the legitimate A with

$$B \rightarrow A\text{: SYN|ACK, ISSb}', \text{ ACK(ISSc).}$$

M does not see this message but uses ISSb to predict the current value ISSb′ and sends

$$M(A) \rightarrow B\text{: ACK, ACK(ISSb}').$$

If the guess is right B assumes that it has a connection with A, when in fact M is sending the packets. M cannot see the output from this session, but it may be able to execute commands with A's privileges on server B. This attack could be run in a Unix environment where the attacker spoofs messages from a trusted host A (Section 7.7.2). Protocols such as Remote Shell (RSH) are vulnerable as they employ address-based authentication, assuming that users logging in from a trusted host have already been authenticated.

To defend against this attack, a firewall could block all TCP packets arriving from the Internet with a local source address. This scheme works if all your trusted hosts are on the local network. If trusted hosts also exist on the Internet, the firewall has to block all protocols that use TCP with address-based authentication. As a better solution, avoid address-based authentication entirely. Cryptographic authentication is preferable.

17.1.3 TCP SYN Flooding Attacks

After responding to the first SYN packet, server *B* stores the sequence number ISSb so that it can verify the ACK from the client. In a TCP SYN flooding attack, the attacker *M* initiates a large number of TCP open requests (SYN packets) to *B* without completing the protocol runs, until *B* reaches its half-open-connection limit and cannot respond to new incoming requests. Modifications of the TCP handshake protocol that allow the server to remain stateless are left as an exercise. As part of a TCP session hijacking attack, *M* could launch a SYN flooding attack against *A* so that *A* does not process the SYN-ACK packet from *B* and would not tear down the connection the attacker wants to open.

17.2 DOMAIN NAME SYSTEM

Hosts on the Internet are usually known by their DNS name. To connect to a host you need the IP address currently corresponding to the DNS name. The DNS is a distributed directory service for domain names (host names). It is used for looking up IP addresses for host names, and host names for IP addresses (reverse look-up). It is also the basis for the same-origin policies applied by web browsers (Chapter 18). Anti-spam systems such as Sender Policy Framework use DNS records to identify valid mail servers.

DNS knows various types of resource records. The binding of host names to IP addresses is given in A records. Host names and IP addresses are collected in zones. A zone is managed by its *authoritative name server*. Authoritative name servers provide the mapping between host names and IP addresses for their zone. Protocols such as BIND, MSDNS, PowerDNS and DJBDNS resolve host names to IP addresses.

We will explain name resolution at a general, simplified level. There are 13 (logical) root servers on the Internet. All name servers are configured with the IP addresses of these root servers. Global Top Level Domain (GTLD) servers are in charge of top-level domains such as *.com*, *.edu*, *.net*, *.org*, *.cc*, *.cn*, *.tv* and *.uk*. There can be more than one GTLD server per top-level domain (TLD). Root servers know about GTLD servers. GTLD servers know the authoritative name servers in their TLD. Recursive name servers pass name resolution requests on to other name servers and cache answers received. IP address look-up then works as follows (Figure 17.1).

1. A client asks its local resolving name server (resolver) to resolve a host name (target).
2. The resolver checks whether it has a valid IP address for the target in its cache. If it does, this address is returned; otherwise, the resolver refers the request to one of the root servers.
3. The root server returns a list of GTLD servers for the target's TLD, and also their IP addresses (in so-called glue records).
4. The resolver refers the request to one of the GTLD servers.
5. The GTLD server returns a list of authoritative name servers for the target's domain, together with their IP addresses.
6. The resolver refers the request to one of the authoritative name servers.
7. The authoritative name server returns an authoritative answer with the target's IP address and time-to-live (TTL) for the binding.
8. The resolver sends the answer to the client and caches the binding.

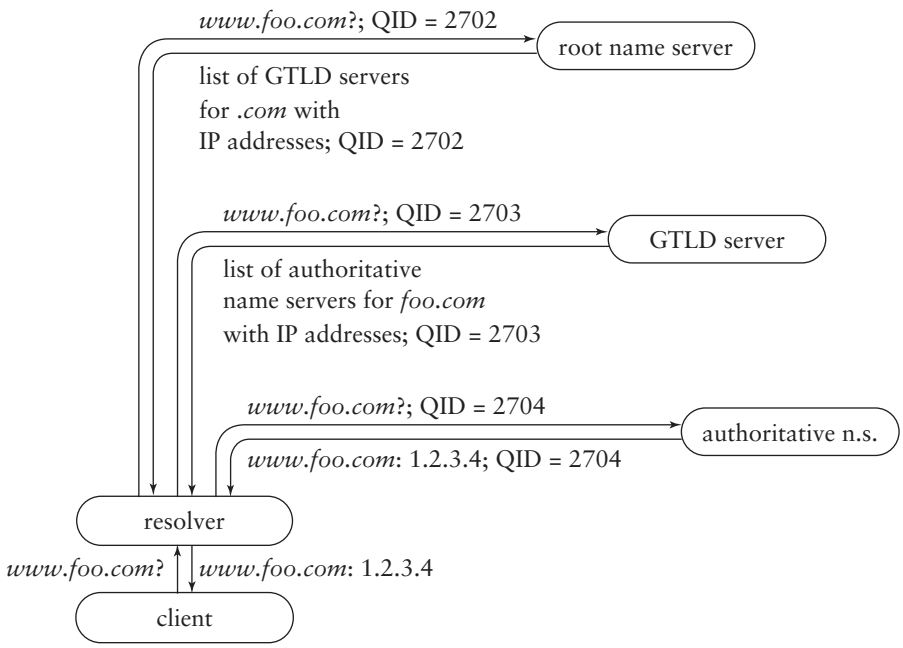


Figure 17.1: Name Resolution in the Domain Name System

The answer remains in the cache until it expires. Note that the time-to-live of the answer is set by the authoritative name server. This reflects the fact that the authoritative name server would know how stable address bindings are in its zone.

17.2.1 Lightweight Authentication

The resolver uses a challenge-response protocol to authenticate the origin of the replies it receives. A 16-bit query ID (QID) and the UDP port that should receive the answer are included in each request. The responding name server copies the QID into its answer and replies on the port indicated. The resolver caches the first answer received for a given QID and host name on the nominated port. The resolver then discards this QID and drops all answers that do not match an active QID.

If the query is not passed by mistake to the attacker, her probability of correctly guessing an answer is 2^{-16}. This assumes that the root server entries at the resolver are correct, that the routing tables in the root servers are correct, that the routing tables in the GTLD servers are correct, that the cache entries at the resolver are correct, and that routing from local name server to authoritative name server is correct. In this case, guessing QIDs is the only attack method left for subverting cache entries.

17.2.2 Cache Poisoning Attack

The attacker asks a resolver to resolve a host name the attacker wants to take over. This host name must not have a valid entry in the resolver's cache. The attacker immediately floods the resolver with spoofed answers that map the host name to an IP address of the attacker's choice. The spoofed answers contain guessed QIDs with a long TTL and are sent on a guessed UDP port. If a spoofed answer with the correct QID arrives on the correct port before the genuine answer, the attacker's value is cached and the correct answer is dropped. If anyone queries the resolver for the target's host name, the resolver will give the address provided by the attacker from its cache, until the attacker's TTL expires. If the authoritative answer comes first it will be cached. The attacker is blocked until the authoritative TTL expires.

The probability of the attack succeeding depends on the following factors:

- The difficulty of guessing the QID – this is particularly easy when the QID is generated by a counter (as in Figure 17.1).
- The difficulty of guessing the port number – this is particularly easy if a fixed port number is used.
- The width of the time window until the authoritative answer arrives – the window of opportunity can be increased by running a simultaneous denial-of-service attack against the authoritative name server.

Good implementations of DNS will address the first two issues.

17.2.3 Additional Resource Records

Name servers send additional resource records to resolvers where these records are cached, just in case they might prove useful in the future. This is a performance

optimization that might save round trips during name resolution later. A malicious name server might provide resource records for other domains when being queried for a host in its domain. Therefore, the resolver performs *bailiwick checking*: additional resource records that do not belong to the queried domain, i.e. records 'out of bailiwick', are not accepted by the resolver.

17.2.4 Dan Kaminsky's Attack

The attacker is in a race with the authoritative name server. If the authoritative answer comes first, the attacker's next attempt has to wait until the TTL expires. An attacker intent on poisoning the cache for *www.foo.com* can escape from this restriction by sending a query for a random host *rand123.foo.com* instead of a query for the intended target. When the resolver's cache has no entry for this random host a new name resolution request is triggered. This ploy defeats TTL as a measure to slow down an attacker, but TTL was not intended as a security mechanism in the first place.

The authoritative name server for *foo.com* is unlikely to have an entry for *rand123.foo.com*. An NXDOMAIN answer will be sent, indicating that the host does not exist. As before, the attacker floods the resolver with spoofed answers for *rand123.foo.com* with guessed QIDs on guessed ports. The spoofed answers come with an additional resource record for *www.foo.com*, which is in bailiwick. If the attacker wins the race against the NXDOMAIN answer, the attacker's entry for *www.foo.com* is cached with a TTL set by the attacker (Figure 17.2).

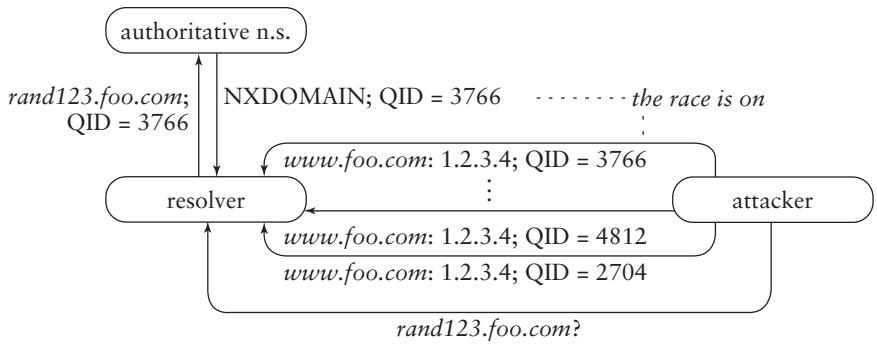


Figure 17.2: Dan Kaminsky's Attack

If the authoritative name server wins, the attack is restarted with a new random host name in the domain *www.foo.com*. The attacker increases her chances of success by creating a large window of opportunity glued together from many small windows. Such attacks have reportedly succeeded in practice within 10 seconds. This is a very serious attack

that could even be launched against TLDs. The attacker would then become the name server for domains of her choice. The following countermeasures have been suggested:

- Increase the search space for the attacker and run queries on random ports – the attacker now must guess QID and port number.
- Restrict access to local resolvers – a zone should use a recursive name server for internal queries to resolve (external) host names and a separate non-recursive authoritative name server for resolving external queries for host names in the zone (split-split name server).
- Give up on simple challenge-response authentication, and use cryptographic authentication instead – apply digital signatures to resource records (DNSSec).

17.2.5 DNSSec

DNSSec, short for DNS Security Extensions, protects the authenticity and integrity of resource records with digital signatures. DNSSec was originally specified in 1999 (RFC 2535). This RFC was superseded by RFCs 4033–4035 in 2005. Several new resource record types were introduced. RRSIG resource records contain signatures of other resource records. DNSKEY resource records contain the public keys of zones. DS (Delegation Signer) resource records contain hashes of DNSKEY resource records.

Authentication chains are built by alternating DNSKEY and DS resource records (Figure 17.3). The public key in a DNSKEY resource record is used to verify the signature on the next DS resource record. The hash in the DS resource record provides the link to the next DNSKEY resource record, and so on. Verification in the resolver has to find a *trust anchor* for the chain (root verification key).

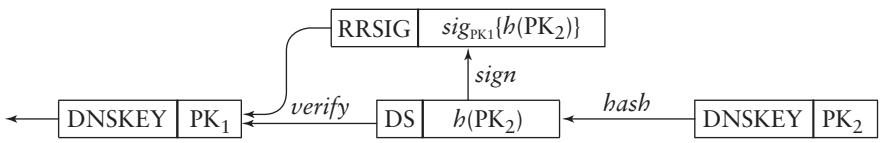


Figure 17.3: DNSSec Authentication Chains

An attacker might not only inject a wrong binding into the cache of a resolver but could also wrongly claim that a host does not exist. DNSSec therefore also provides *authenticated denial of existence*. NSEC ('next') resource records are used to create a daisy chain of hosts in a zone. The NSEC record for a host gives the host next in the zone in alphabetic order. (The NSEC record for the last host points back to the first entry.) To prove that a host does not exist, the NSEC resource record for the host in the zone that would come immediately before the host queried is sent. This is the evidence that no hosts exists between this host and the next host from the NSEC resource record.

This solution makes *zone walking* possible. All hosts in a zone can be enumerated by following the NSEC resource records. Some take the view that the DNS is a public database so there is no problem. Others see security and privacy issues and are opposed to deploying NSEC. The new NSEC3 resource record (RFC 5155) uses hashes of host names and a more complex *closest encloser* proof for authenticated denial of existence.

The infrastructure for supporting DNSSec is still in its infancy. Verification of authentication chains needs trust anchors. It is conceivable that the hierarchy of signers will mirror the hierarchy of the DNS name space. That is, the root would sign the public keys of TLDs, TLDs would sign the public keys of their domains, and so on. For a global system, this raises tricky political questions. At the time of writing, few TLDs have their own DNSSec keys. Moreover, DNSSec makes changes to the DNS protocols. Hence, all machines running DNS protocols will eventually have to be upgraded. Finally, there is the general challenge of bootstrapping a new service when it is only supported by parts of the infrastructure.

17.2.6 DNS Rebinding Attack

When resolving a host name, the resolver 'trusts' the authoritative DNS server and relies on the information received. Trust is bad for security. The authoritative DNS server might lie. In a DNS rebinding attack, the attacker binds a host in its own domain to the victim's IP address. Same-origin policies (Chapter 18) authorize actions based on the domain of the host at which the action is performed. The attacker may then get access to the victim's machine because it appears to be in the attacker's domain. DNSSec is no defence against this type of attack.

The attacker prepares a DNS rebinding attack by putting a web page that contains a malicious script on a host in its domain *attacker.org*. The script will request a connection to the victim when the page is visited. When a user visits this page the user's browser needs an IP address for the host and asks the authoritative DNS server for domain *attacker.org*. The attacker's DNS server is authoritative for this domain. The attacker may lie:

- The DNS server binds *attacker.org* to two addresses, to the attacker's and to the victim's IP address. When the script issues its request to the victim, the browser will allow the request as it appears to go back to *attacker.org*. This vulnerability in an early version of Netscape was fixed by letting the same-origin policy refer to the IP address instead of the domain name [78].
- The DNS server binds *attacker.org* to the attacker's IP address with a short time-to-live. The script waits before sending a request to *attacker.org* until the browser's binding of the host name has expired. The authoritative DNS server is then asked again. Now, *attacker.org* is bound to the victim's address. To fix this problem, the browser should not trust the DNS server on the time-to-live of a binding but set its own by *pinning*

the host name to the IP address the web page was loaded from.[1] Pinning overrules the TTLs set by authoritative name servers.

- The attacker takes its web server off-line after the page has been loaded. The script waits before sending a request to *attacker.org*. At that time the browser's connection attempt fails and the pinning is dropped. The script asks again and the browser performs a new DNS look-up. The browser now gets the victim's IP address [132]. This is an instance of a more widespread problem in security: *error handling* procedures implemented without duly considering their security implications.

Plug-ins extend browser functionality but can introduce new DNS rebinding vulnerabilities [130]. Plug-ins may do their own pinning when they do not want to rely on the browser. An attacker may then use the client browser as a proxy to connect to the victim by resolving *attacker.org* to the attacker's IP address for one plug-in and to the victim's IP address by another. The two plug-ins may communicate in the browser because they appear to connect to the same host. Having one pinning database for all plug-ins prevents this attack.

Lesson
Centralizing security mechanisms has its advantages.

The same-origin policy may be made more flexible by letting a DNS server tell which IP addresses are valid for a host in its domain. This can be useful for providers hosting content on several sites who want to direct their customers to the most convenient site. With respect to security, we are literally back to square one. The DNS server might lie and nominate the victim's IP address in its policy. As a defence, we might not only ask the authoritative DNS server of a domain, but also cross-check with the host whose IP address has been given to check whether it agrees to be associated with *attacker.org*. This check could be implemented as an extension of reverse DNS look-up [130]. There is a parallel to the bombing attacks covered in Section 19.4.

17.3 FIREWALLS

Cryptographic mechanisms protect the confidentiality and integrity of data in transit. Authentication protocols verify the source of data. To control what traffic is allowed to enter your network (ingress filtering) or leave your network (egress filtering) you may deploy a *firewall*.

A firewall is a network security device controlling traffic flow between two parts of a network.

[1] J. Roskind. Attacks against the Netscape browser. Invited talk, RSA Conference, 2001.

Firewalls became popular in the early 1990s, a time when end hosts were often PCs that could not defend themselves at all. Then, it made sense to concentrate security enforcement at the network boundary. Firewalls are often installed between the network of an entire organization and the Internet, but could also be installed in an intranet to protect individual departments. For example, a university could put firewalls between the subnets of academic departments and the main campus network.

Firewalls defend a protected network against parties who try to access services from outside the network that are intended to be available only internally. Firewalls can also restrict access from inside to external services that are deemed dangerous or unnecessary for the work of an organization. All traffic has to go through the firewall for protection to be effective. Dial-in lines (in the distant past), wireless LANs, laptops, and USB sticks are notorious examples of unprotected entry points into the network behind a firewall.

A firewall can decide to route sensitive traffic via a *virtual private network* (VPN). A VPN establishes a secure connection between the gateways of subnets of an organization that are not directly connected. All traffic between the subnets has to go through these gateways where cryptographic protection is added to extend the security perimeter. A firewall can also perform network address translation, hiding internal machines with private addresses behind public IP addresses, and translating public addresses to private addresses for internal servers. Hiding the internal structure of a network reduces the attack surface. Fewer targets are known to the attacker.

Firewalls implement access control. Parameters that could be used for access control can be found at each network layer. At OSI layer 3 you have source and destination IP addresses. At OSI layer 4 you have TCP and UDP port numbers. Note that the port number does not necessarily define the service running at that port. At OSI layer 7 there is information related to various applications: email addresses, email contents, web requests, executable files, viruses and worms, images, usernames, and passwords, to name just a few.

17.3.1 Packet Filters

Packet filters work at OSI layers 3 and 4. Rules specifying which packets are allowed through the firewall and which are dropped are applied to packets individually. Typical rules specify source and destination IP addresses, and source/destination TCP and UDP port numbers. Rules for traffic in both directions can be defined. Such a firewall can be implemented by a TCP/IP packet filtering router which examines the TCP/IP headers of every packet going through and can drop packets.

Only static rules can be enforced and certain common protocols are difficult to handle. For example, when a client sends an FTP request to an FTP server, the firewall cannot link the data packets coming back from the server to this request. We can have blanket

rules for all packets coming in on port 20, or all packets on port 20 from IP addresses nominated in advance, but we cannot have dynamically defined rules.

17.3.2 Stateful Packet Filters

Stateful (dynamic) packet filters understand requests and replies. For example, they would know about the (SYN, SYN-ACK, ACK) pattern of a TCP open sequence. Rules are usually only specified for the first packet in one direction, and a new rule is created dynamically after the first outbound packet. Further packets in the communication are then processed automatically. Stateful firewalls can support policies for a wider range of protocols than simple packet filter, e.g. FTP, IRC, or H323.

Packet filtering can be done by routers, giving high performance at lower cost. Moreover, it is easier to configure securely platforms that offer only limited functionality. Linux systems use *iptables* as the data structure for defining packet filtering rulesets. The filtering policies that can be enforced are limited by the parameters that can be observed in the TCP and IP headers, but can be specified more easily.

17.3.3 Circuit-Level Proxies

Circuit-level proxies have rules similar to packet filters but do not route packets. Rules determine which connections are allowed and which will be blocked. Allowed connections generate a new connection from firewall to destination. This type of firewall is mentioned for the sake of completeness. It is rarely used in practice as the functionality is similar to that of stateful packet filters but the performance is lower.

17.3.4 Application-Level Proxies

For each application protocol the firewall should police, a proxy implements the server and client part of the protocol on the firewall. When a client connects to the firewall, the proxy at the firewall acts as the server and validates the request. A mail proxy, for example, could filter out viruses, worms and spam. If the client request is allowed, the proxy acts as a client and connects to the destination server. Responses come back through the firewall and are again processed and checked by the proxy. The proxy server is the only entity seen by the outside world and it appears transparent to the internal users except for filtering, e.g. removing email attachments. Proxies can be seen as another instance of *controlled invocation*.

Application-level proxies typically run on a hardened PC. They can provide close control over the content of incoming and outgoing traffic. In this respect, application-level proxies offer a high level of security, provided the configuration is appropriate. On the downside, the amount of processing per connection is large and configuration more complicated. In this respect, application-level proxies are less secure, and security vulnerabilities in firewall products have been reported. Performance is lower and cost is higher compared

with packet filters. Moreover, you need a proxy server for each service you want to protect. Therefore, this approach does not scale too easily with the growing number of Internet services on offer.

The actions of packet filters have been compared to telephone call barring by number. This blocks calls to certain numbers, e.g. premium rate numbers. In contrast, application-level proxies are like telephone call monitoring by listening to the conversations.

17.3.5 Firewall Policies

Permissive policies allow all traffic but block certain dangerous services, such as Telnet or *snmp*, or port numbers known to be used by an attack. If you forget to block something you should, it is allowed, and might be exploited for some time without you realizing it. *Restrictive* policies block all traffic and allow only traffic known to meet a useful purpose, such as HTTP, POP3, SMTP, or SSH. This is the more secure option. If you block something that is needed, someone will complain and you can then allow the protocol. A policy is usually represented as an ACL with positive and negative entries. A typical firewall ruleset could look like this:

- Allow from internal network to Internet: HTTP, FTP, SSH, DNS
- Allow from anywhere to mail server: SMTP only
- Allow from mail server to Internet: SMTP, DNS
- Allow from inside to mail server: SMTP, POP3
- Allow reply packets
- **Block everything else.**

Defining and managing rulesets is an important issue when deploying firewalls in practice.

17.3.6 Perimeter Networks

Where should a mail server be placed in relation to the firewall? A mail server requires external access to receive mail from outside, so it should be on the inside of the firewall. Only then can the firewall protect access to the mail server from outside. A mail server also requires internal access to receive mail from the internal network, so it should be outside the firewall. You might want to stop worms and viruses spreading from your network or prevent confidential documents leaving your network. As a solution, you could create a perimeter network, also known as a demilitarized zone (DMZ) for servers which require (selective) access from both inside and outside of the firewall (Figure 17.4). Besides mail servers, the DMZ would also be the place to put web servers and DNS servers.

17.3.7 Limitations and Problems

Firewalls do not protect against insider threats. Blocking services may create inconveniences for users. Network diagnostics may be harder. Some protocols are hard to support. Packet filtering firewalls do not provide content-based filtering. If email is

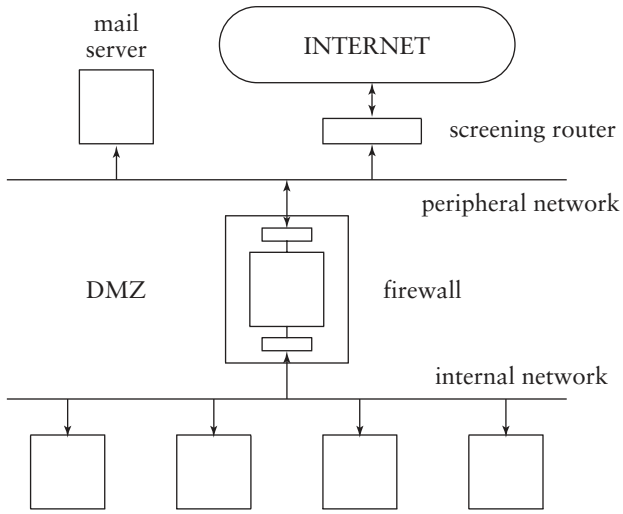


Figure 17.4: Creating a Demilitarized Zone

allowed through, then emails containing viruses are allowed through. Even application-proxy firewalls may not perform thorough checks on content. Firewalls do not know about operating system or application vulnerabilities. A lot of services today use HTTP at port 80 so it becomes increasingly difficult to decide which traffic on this port is legitimate.

Protocol tunnelling, i.e. sending data for one protocol through another protocol, negates the purpose of a firewall. As more and more careful administrators block almost all ports but have to leave port 80 open, more and more protocols are tunnelled through HTTP to get through the firewall. Another candidate for tunnelling is the SSH protocol.

Encrypted traffic cannot be examined and filtered, so protocols such as HTTPS and SSH that provide end-to-end cryptographic protection cannot be monitored by the firewall. The alternative is to have proxies for such protocols in the firewall, but then you lose end-to-end security. These developments have led some to announce the near-demise of firewalls as a component of security architectures and to predict that security services would move back from the network into the end hosts. *Personal firewalls* are moving access control on network traffic back to the end systems.

17.4 INTRUSION DETECTION

It would be nice to prevent all attacks, but in reality this is rarely possible. New types of attacks occur, such as denial-of-service attacks (where cryptography may make the problem worse). Perimeter security devices such as firewalls mainly *prevent* attacks by

outsiders. They may fail to do so. A firewall may be misconfigured, a password may be sniffed off the network, a new type of attack may emerge. Moreover, these devices do not *detect* when an attack is under way or has taken place. To detect network attacks, an intrusion detection system (IDS) can be fielded.

An IDS consists of a set of *sensors* gathering data, either located on the hosts or on the network. The sensor network is managed from a central *console*. There, data are analyzed, intrusions reported, and possibly reactions triggered. There are two approaches for detecting intrusions, misuse detection and anomaly detection. The communications between sensors and console should be protected, as well as the storage of the signature database and of the logs generated. There should also be a secure scheme for getting signature updates from the IDS vendor. Otherwise, the IDS itself could be attacked and manipulated. There have been incidents where vulnerabilities in IDS systems have been exploited for attacks.

17.4.1 Vulnerability Assessment

Vulnerability assessment examines the security state of a network. Information about open ports, software packages running (which version, patched?), network topology, etc. is collected, and a prioritized list of vulnerabilities is compiled. Vulnerability assessment is only as good as the knowledge base being used, which has to be updated constantly to handle new threats. Several organizations track security vulnerabilities and lists of available patches. Good sources for this information are the Computer Emergency Response Teams (CERTs), e.g. the CERT at Carnegie Mellon University (`http://www.cert.org/`), SANS (`http://www.sans.org/`), Security Focus (`http://www.securityfocus.com/`) where the BugTraq archive is maintained, and the websites of major software and hardware manufacturers.

17.4.2 Misuse Detection

Misuse detection looks for *attack signatures*. Attack signatures are patterns of network traffic or activity in log files that indicate suspicious behaviour. Example signatures might include the number of recent failed login attempts on a sensitive host, a pattern of bits in an IP packet indicating a buffer overflow attack, or certain types of TCP SYN packet that indicate a SYN flooding attack. The IDS may also consult the security policy and a database of known vulnerabilities and attacks for the system monitored.

These systems are only as good as the information in the database of attack signatures. New vulnerabilities are constantly being discovered and exploited. Vendors need to keep up to date with the latest attacks and issue database updates. Customers need to install the updates. The database of known vulnerabilities and exploitation methods can become large and unwieldy and may slow down the IDS. At the time of writing commercial IDS products are still based on misuse detection. These systems are also known as knowledge-based IDSs.

17.4.3 Anomaly Detection

Statistical anomaly detection (or behaviour-based detection) uses statistical techniques to detect potential intrusions. First, the 'normal' behaviour is established as a baseline. During operation, a statistical analysis of the data monitored is performed and the deviation from the baseline is measured. If a threshold is exceeded, an alarm is raised. Such an IDS does not need to know about security vulnerabilities in a particular system. The baseline defines normality. So, there is a chance of detecting novel attacks without having to update a knowledge base.

On the other hand, anomaly detection just detects anomalies. Suspicious behaviour does not necessarily constitute an intrusion. A burst of failed login attempts at a sensitive host could be due to an attack or to the administrator having forgotten the password. Some interesting insights into this problem can be gleaned from [22]. Attacks are not necessarily anomalies. A careful attacker might just 'fly under the radar' of the IDS and remain undetected. This is particularly true when the baseline is adjusted dynamically and automatically. A patient attacker may be able to gradually shift 'normality' over time until his planned attack no longer generates an alarm. Thus, we have to be concerned about *false positives* (false alarms) when an attack is flagged although none is taking place, and *false negatives* when an attack is missed because it falls within the bounds of normal behaviour.

17.4.4 Network-Based IDS

A network-based IDS (NIDS) looks for attack signatures in network traffic. Typically, a network adapter running in promiscuous mode monitors and analyzes all traffic in real time as it travels across the network. The attack recognition module uses network packets as the data source. There are three common techniques for recognizing attack signatures: pattern, expression or bytecode matching; frequency or threshold crossing (e.g. to detect port scanning activity); and correlation of lesser events (not yet widely used in commercial products). *Snort* is a popular NIDS developed in the open-source community. The main challenge in NIDSs is the extraction of relevant information out of the vast number of events logged, preferably in near real time.

17.4.5 Host-Based IDS

A host-based IDS (HIDS) looks for attack signatures in log files of hosts. It can also verify the checksums of key system files and executables at regular intervals. Some products can use regular-expressions to refine attack signatures (e.g. **passwd** program executed AND **.rhosts** file changed). Some products listen to port activity and generate alerts when specific ports are accessed, providing limited NIDS capability. There is a trend towards host-based intrusion detection. The most effective IDSs combine NIDS and HIDS.

Due to the near real-time nature of IDS alerts, an IDS can be used as a response tool, but automated responses are not without dangers. An attacker might trick the IDS into responding, with the response aimed at an innocent target (say, by spoofing source IP address). Users can be locked out of their accounts because of false positives. Repeated email notifications become a denial-of-service attack on the administrator's email account.

17.4.6 Honeypots

Honeypots are systems used to track attackers and to learn and gather evidence about *novel* attack techniques. Honeypots mimic real systems but do not contain, and therefore reveal, real operational data. By definition, every activity monitored on the honeypot is unauthorized.

> A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource [214].

Honeypots can engage in a session with an attacker at different levels of interaction. Low-interaction honeypots offer basic emulations of some services and of the operating system. There is not much an attacker can do on such a honeypot so there is a limit to the adversarial behaviour the honeypot can log. Moreover, an attacker might quickly recognize the honeypot for what it is and walk away. There already exist tools for detecting honeypots. The more sophisticated the emulations become, the more types of behaviour can be observed. High-interaction honeypots offer real services, with fake data. The more interactions made possible, the greater is the danger that an attacker can misuse the honeypot as a staging post for launching attacks against other machines.

The three main challenges in this area are the constructing of convincing honeypots and honeynets, in particular when adversarial behaviour at the application level should be investigated, the securing of honeypots, and the extraction of novel attacks from the behaviour monitored. Most attacks seen will already be known.

17.5 FURTHER READING

This chapter has only been able to give a brief sketch of the issues and techniques in network security. Network security is covered comprehensively in [215]. RFC 3833 gives a threat analysis for the Domain Name System. Good books on firewalls are [63, 238]. Technical issues that arise in network intrusion detection are discussed in [193]. New research in intrusion detection is published in the RAID conference series.

17.6 EXERCISES

Exercise 17.1 The Address Resolution Protocol (ARP) associates hardware addresses with IP addresses. This association may change over time. Each network node keeps an ARP cache of corresponding IP and hardware addresses. Cache entries expire after a few minutes. A node trying to find the hardware address for an IP address that is not in its cache broadcasts an ARP request that also contains its own IP and hardware address. The node with the requested IP address replies with its hardware address. All other nodes may ignore the request. How could ARP spoofing be performed? What defences can be used against spoofing?

Exercise 17.2 Develop a stateless implementation of the TCP handshake protocol that is not vulnerable to TCP SYN flooding attacks.

Exercise 17.3 Consider a DNS resolver that does not keep track of host names it is currently trying to resolve. Several queries for the same host name may thus be active at the same time. How can this situation be exploited by a cache poisoning attack? What is the probability of success of your attack?

Exercise 17.4 What security features do you expect from a secure email system, and from the machines running a secure email system? Which protocol layer is most appropriate for such a security service? In your answer, distinguish between services that want to offer anonymity and those that do not.

Exercise 17.5 Examine the implications of tunnelling IP through IP on the design of a packet filtering firewall.

Exercise 17.6 Why is dynamic port allocation a potential problem for packet filtering firewalls? Suggest a solution for requests coming from the internal network that expect answers on a dynamically allocated port. Suggest a solution for protocols where the responder specifies the port number where further queries are expected to arrive.

Exercise 17.7 End-to-end encryption is a potential problem for application-level proxies. Suggest a solution so that a protocol encrypting its payloads can traverse an application-level proxy.

Exercise 17.8 Firewalls protect an internal network from the outside. Can firewalls protect against virus infections? Consider the different types of firewalls in your answer. How does cryptographic protection at the TCP/IP layer or at the application layer affect a firewall's ability to protect against viruses?

Exercise 17.9 A company allows its employees to use laptops at home and when travelling. Propose a security architecture to protect the laptops and the company's intranet.

Exercise 17.10 A DNS resource record can indicate that a host is a designated mail server. Design a spam defence that makes use of this feature.

Chapter 18

Web Security

The web is an IT infrastructure consisting of standardized protocols, document formats, and software components. The browser is the core client-side component. Its counterpart is the web server. Vulnerabilities in this infrastructure have become a major concern in software security. SQL injection and cross-site scripting figure prominently in current vulnerability statistics. The web is also the multitude of applications offered on this IT infrastructure. Web applications are becoming more and more dynamic, both in the way they are reacting to user input and in the way applications can be composed.

Web security includes aspects of network security. Tunnels built at different conceptual layers have to fit at their end points. This plumbing job has to be done with care. Web security includes aspects of software security. Code-injection attacks target web applications. Web security includes aspects of access control. Security policies stipulate how data may be shared between websites. The security attributes (evidence) policies referred to must be authenticated. Web security also refers to the security framework for web services developed by OASIS. This framework supports the collaboration between organizations that is facilitated by the web as a common distributed computing platform.

OBJECTIVES

- **Point to the issues that arise when combining tunnels at different conceptual layers.**

- **Describe the role of the browser as a security component.**
- **Discuss service-oriented access control.**
- **Introduce the basics of web services security.**

18.1 INTRODUCTION

We start with a generic description of the web as a technical infrastructure. We will omit many details of the protocols, data formats and software components involved and focus on general principles. We will also abstract from specific browser behaviour. While the browsers on the market handle many issues in the same way, there exist some differences. Further, browsers may change their modus operandi in the future when faced with new security challenges.

The term Web 1.0 has become shorthand for web applications that deliver static content. Figure 18.1 shows the basic information flow in such a setting. At the client side, interaction with the application is handled by the *browser*. At the server side, the *web server* receives the client requests. Scripts at the web server extract input from the client data and construct requests to a *back-end server*, e.g. a database server. The web server receives the result from the back-end server and returns *HTML result pages* to the client. Web server and back-end server are separate logical components but may reside on the same physical machine.

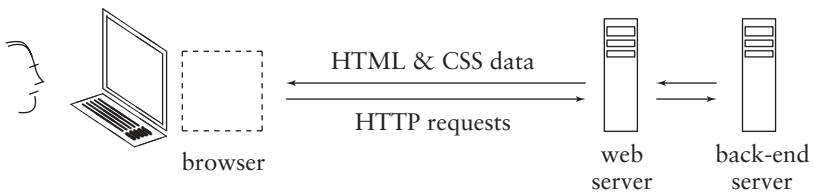


Figure 18.1: Web 1.0 Application

18.1.1 Transport Protocol and Data Formats

The transport protocol used between client and server is the hypertext transfer protocol (HTTP). HTTP/1.1 is specified in RFC 2616. HTTP is located in the application layer of the Internet protocol stack. This *network application layer* must not be confused with the *business application layer* in the software stack. The client sends HTTP *requests* to the server. A request states a *method* to be performed on a resource held at the server.

www.wiley.com/WileyCDA/Section/id-302475.html?query=computer%20security

Figure 18.2: Reading Host and URI from a Browser Bar

The GET method retrieves information from a server. The resource is given by the Request-URI and Host fields in the request header. The syntax of a URI is specified in RFC 3986. Figure 18.2 shows how host and Request-URI are displayed in a typical browser bar. You start at the delimiter between host and URI, read the host from right to left, then jump to the start of the URI and read the URI from left to right. Attacks have exploited this idiosyncrasy by constructing host names that contain a character that looks like a slash. A user parsing the browser bar will interpret the string to the left of this character as the host name. The actual delimiter used by the browser is, however, far out to the right in what the user reads as the URI. To defend against this attack, browser vendors are pursuing two strategies:

- Block dangerous characters – this method reaches its limit when the dangerous symbol is a legal character in the alphabet that host names may be written in.
- Display to the user where the browser splits host name from URI – the user's abstraction can thus be aligned with the browser's implementation.

The POST method specifies the resource in the Request-URI and puts the action to be performed on it into the *body* of the HTTP request. The POST method was intended for posting messages, annotating resources, and sending large data volumes that would not fit into the Request-URI. However, in principle it can be used for any other actions that can be requested by using the GET method. Side effects may differ depending on whether an action was requested by GET or by POST.

The server sends HTTP *responses* to the client. Web pages in a response are written in HyperText Markup Language (HTML). The elements that can appear in a web page include *frame* (subwindow), *iframe* (in-lined subwindow), *img* (embedded image), *applet* (Java applet), *form* (interactive element specifying an action to be performed on a resource when triggered by a particular event; *onclick* is such an event). The server may use Cascading Style Sheets (CSS) to give further information on how to display the web page.

18.1.2 Web Browser

The client browser performs several functions:

- Displaying web pages – the Domain Object Model (DOM) is an internal representation of a web page used by browsers [150]; JavaScript requires this particular representation.
- Managing sessions (Section 18.2).
- Performing access control when scripts within a web page are executed (Section 18.3).

When the browser receives an HTML page it parses the HTML into the `document.body` of the DOM. Objects like `document.URL`, `document.location`, and `document.referrer` get their values according to the browser's view of the current page.

18.1.3 Threat Model

The security analysis of web applications does not assume the standard threat model of communications security where the attacker is 'in control of the network' and can read, modify, delete and insert messages (Section 16.1.1) or the standard threat model of operating system security where the attacker has access to the operating system command line (Section 6.2). The *web adversary* is a malicious end system. This attacker only sees messages addressed to him *and data obtained from compromised end systems accessed via the browser*. The attacker can also guess predictable fields in unseen messages.

The communications network is 'secure'. End systems may be malicious or may be compromised via the browser. We do not consider attacks that exploit vulnerabilities in the implementation of other network protocols.

18.2 AUTHENTICATED SESSIONS

Since HTTP/1.1, client and server have been able to establish a communications session based on TCP. Such sessions are not authenticated. When application resources are subject to access control, the user at the client has to be authenticated as the originator of requests. This is achieved by establishing an authenticated session. Authenticated sessions exist at three conceptual layers:

- at the business application layer, as a relationship between user (subscriber) and service provider;
- at the network application layer, between browser and web server;
- at the transport layer, between client and server.

Authenticated sessions at the transport layer can be established with SSL/TLS. For users in possession of a certificate and a corresponding private key, TLS with mutual authentication can be used (Section 16.5). When user and service provider share a password, a protocol such as EAP-TTLS is suitable (Section 16.6). Running HTTP over TLS in the HTTPS protocol is specified in RFC 2818.

At the network application layer, the server may create a *session identifier* (SID) and transmit it to the client. Note that in our threat model the SID can be captured once it is stored in an end system but not during transit. The client includes the SID in subsequent requests to the server. Requests are authenticated as belonging to a session if they contain the correct SID. The server may have authenticated the user before the SID was issued and may encode this fact in the SID. The server may have issued the SID without prior user authentication and may use it for checking that requests belong to the same session.

There are three methods for transferring session identifiers:

- Cookie – sent by the server in an HTTP response in a `Set-Cookie` header field (RFC 2965). The browser stores the cookie in `document.cookie` and includes it in requests with a domain matching the cookie's origin.
- URI query string – the SID is included in Request-URIs for resources of the application.
- POST parameter – the SID is stored in a hidden field in an HTML form.

At the business application layer, the server can again send an authenticator to the client. This authenticator has to be stored in the private space of the application at the client side. In JavaScript, a script running in the browser may use a *private object* for this purpose.

18.2.1 Cookie Poisoning

If SIDs are used for access control, you have to consider that malicious clients and outside attackers may try to elevate their permissions by modifying a SID (cookie). Such attacks are known as *cookie poisoning*. Outside attackers may further try educated guesses about a client's cookie, maybe after having contacted the server themselves, and try to use their guess to impersonate the user. They could also try to steal the cookie from client or server. Cookie stealing is described in Section 18.4.1. The web threat model imposes two requirements on session identifiers: they must be unpredictable, and they must be stored in a safe place. The server can prevent modification of a SID by embedding a cryptographic message authentication code in the SID constructed from a secret only held at the server.

18.2.2 Cookies and Privacy

When cookies were first introduced in the 1990s, there were fears about their impact on user privacy. To address these fears, cookies were defined to be domain-specific. Servers are only sent cookies belonging to their domain. Thus, cookies do not disclose information to the server other than that someone had visited a site in this domain before. Attacks on user privacy could still be performed within a domain by creating client profiles, combining information from cookies placed by different servers put artificially in the same domain, or by observing client behaviour over time.

Users can protect their privacy by configuring their browsers to control cookie placement. The browser could ask for permission before storing a cookie, which easily becomes a nuisance, or block cookies altogether. There is also the option of deleting cookies at the end of a session.

An interesting legal conundrum about cookies is explained in [118]. Earlier versions of P3P (Section 9.6) could only express policies about retrieving cookies. This is reasonable from a technical point of view but not in accordance with the EU Data Protection

Directive. The Directive asks for user consent at the time personal data is written. This addresses a privacy concern originally related to databases holding personal data. When data about a person is recorded on systems belonging to someone else, it makes sense to ask for consent when data is written. Cookies, however, store data pertaining to a user on that user's machine. The sensitive operation is read access by some other party.

Lesson

Legislation may enshrine old technology. Laws regulating IT are passed to meet the challenges posed by the technology of the time they were drafted. It may happen that lawmakers incorporate assumptions about the use of technology that, with the benefit of hindsight, turn out to apply only to the specific applications of their time. A law may thus prescribe not only the protection goal, which remains unchanged, but also the protection mechanism, which may not be the best option in some novel application.

18.2.3 Making Ends Meet

In the EAP-TTLS scenario, the client first establishes a transport-layer session where the server is authenticated to the client. Within this session, an application-layer session is established where the user is authenticated to the server. This approach is secure as long as both sessions have the same end point. However, when a user is tricked into opening a TLS session with an attacker, a man-in-the-middle attack becomes possible (Figure 18.3). The user opens a TLS tunnel to the attacker. The attacker opens a TLS tunnel to the server. The server asks for the user's credentials. The attacker passes the request to the user. The user replies, sending the credentials in the 'secure' tunnel to the attacker. The attacker successfully completes authentication at the server. The server creates a *user authenticator* (UAC), e.g. a cookie, and sends it to the user via the attacker. The attacker can now impersonate the user.

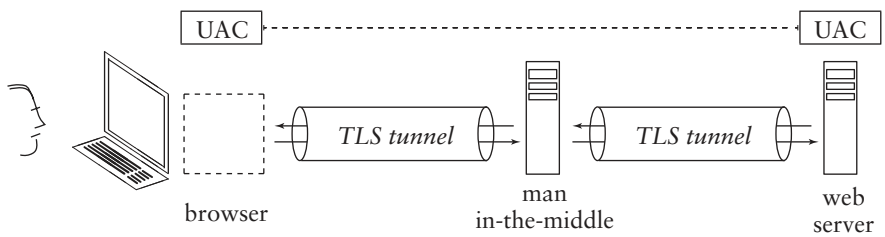


Figure 18.3: Man-in-the-Middle Attack Breaking Application-Layer Sessions

As a defence, bind the UAC not only to user credentials but also to the TLS session in which the credentials are being transferred to the server. The server can then detect

whether requests are sent and received in the same TLS session. If the sessions differ, it is likely that a man-in-the-middle sits between client and server. More details can be found in [186] and RFC 2617.

Lesson
'Client' and 'server' are dangerous abstractions. You must know precisely at which layer a tunnel ends.

The man-in-the-middle attack in Figure 18.3 combines two TLS tunnels in space. A man-in-the-middle may also combine two TLS tunnels in time (Figure 18.4, [160]). This attack exploits a weakness in the way web servers use TLS for user authentication. A user may first establish an anonymous TLS tunnel to the server. User authentication is triggered once access to a protected resource is requested. The server then sends a TLS Hello Request asking the client to renegotiate the TLS session. In the new negotiation the user is authenticated using public-key cryptography. The TLS specification makes no

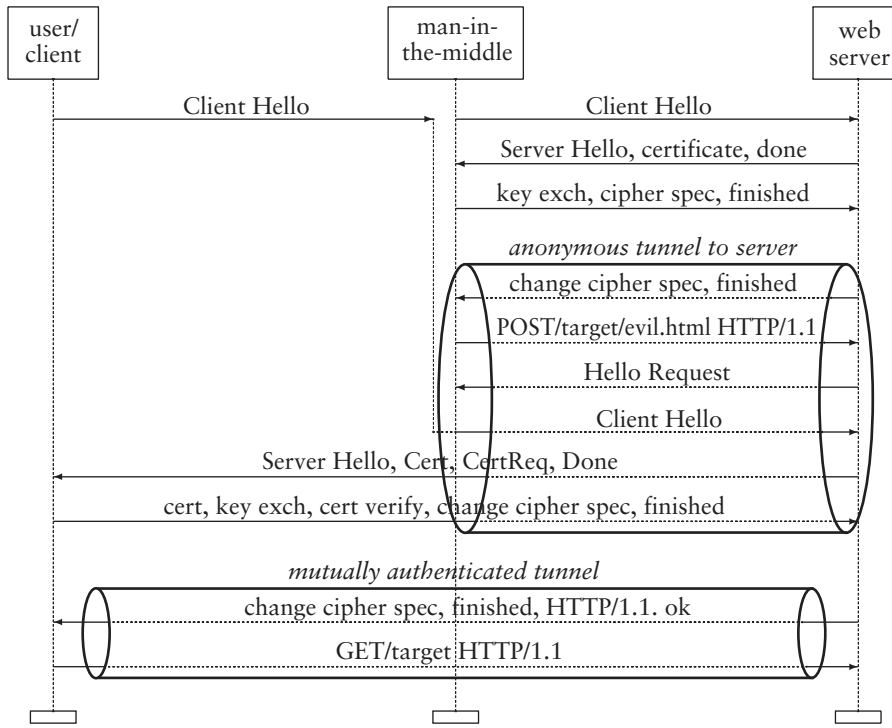


Figure 18.4: Man-in-the-Middle Attack Exploiting TLS Session Renegotiation

claims about any connection between the two sessions, but web servers assumed that a renegotiation starting in a TLS tunnel must extend this tunnel.

This assumption is consistent with the old threat model of communications security. A TLS tunnel gets traffic securely across an insecure network. The end point of the tunnel is honest. When you renegotiate a tunnel with an honest end point, the new tunnel will have the same end point. The assumption is incompatible with the web adversary. When you renegotiate a tunnel with a malicious end point, the new tunnel may end somewhere else.

The attack is launched by a malicious end point. When the victim starts a TLS session, the attacker blocks the initial Client Hello message and negotiates itself an anonymous TLS session with the server. The attacker then performs an action requiring user authentication, e.g. posting something to a target site. The server now sends a TLS Hello Request to the attacker, the attacker replies with Client Hello and then passes the server's certificate request to the victim. The victim submits the user's credentials and a new mutually authenticated tunnel is created. When the attacker's pending HTTP request is somehow attached to the first request in the new tunnel (there are ways in HTTP to influence how the next HTTP header received will be processed), it will be executed with the user's permissions.

In response to this attack, the TLS specification for session renegotiation was modified so that the new session can be cryptographically linked to the old session (RFC 5746). The implementation of a service was made to follow the abstraction users relied on.

18.3 CODE ORIGIN POLICIES

Same-origin policies enforced by web browsers aim to protect application payloads and session identifiers from outside attackers. A web application is identified by the *domain* of its hosting web server. The same-origin policy states that scripts may only connect back to the domain they came from, or that cookies are only included in requests to the domain that placed them. Two Uniform Resource Locators (URLs) have the same origin if they share the protocol, host name, and port number. Table 18.1 illustrates this policy. The policy does not restrict static HTML content. You can embed images from other domains in your web page.

The same-origin policy is too restrictive if interaction between hosts in the same domain is permissible. *Parent domain traversal* is a common exception to the same-origin policy. The domain name held in `document.domain` in the DOM is shortened to its *.domain.tld* portion. Thus, `wwww.my.org` can be shortened to `my.org` but not to `org`. This exception can have undesirable side effects when the Domain Name System is used creatively. For example, domain names of UK academic institutions end with `.ac.uk`, but `ac.uk` is not a proper top-level domain. If a browser restricts access to the *domain.tld* portion of the host name only, it potentially leaves all `ac.uk` domains open

URL	Result	Reason
`http://www.my.org/dir1/some.html`	success	
`http://www.my.org/dir2/sub/another.html`	success	
`https://www.my.org/dir2/some.html`	failure	different protocol
`http://www.my.org:81/dir2/some.html`	failure	different port
`http://host.my.org/dir2/some.html`	failure	different host

Table 18.1: Evaluating Same Origin for `http://www.my.org/dir1/hello.html`

to same-origin policy violations. Browsers therefore come with a list of exceptions to this exception, i.e. domains where parent domain traversal must not be performed.

Origin-based policies are in general relevant for Service Oriented Architectures (SOAs). SOA is an architectural paradigm for utilizing distributed *capabilities* that may be under the control of different owners. A capability realizes real world effects and is implemented by *services*. Security policies regulate how services may interact. Such policies refer to services, thus services become principals in SOA access control. We then must find a way of naming those principals.

For services provided on the web, it has become customary to use the DNS name of the server hosting the service. Note, though, that DNS was not designed for the purpose of providing evidence for access control decisions. Services communicate via messages. When services are principals and when principals are known by host names, security policies refer to host names. When enforcing such security policies, the origin of messages has to be authenticated.

18.3.1 HTTP Referer

Web pages may include requests from different domains. The *Referer* field in the HTTP request-header is intended to give the client the means of specifying the URI of the resource from which a request was obtained. However, the *Referer* field is not always included and might be forged. Access control thus cannot rely on it.

18.4 CROSS-SITE SCRIPTING

Cross-site scripting (XSS) is an *elevation of privilege* attack that exploits the client's 'trust' in a server. Web pages from the trusted server are processed in a context that has more permissions than a page from the attacker's server would get. For example, the trusted server could be in an intranet zone while the attacker's server is in the Internet zone. The attack passes a script to the client via the trusted server, evading the client's origin-based security policy. The user has to be lured into taking an action that triggers the attack, e.g. by clicking on a poisoned link. There are, however, more surreptitious ways of causing the user action.

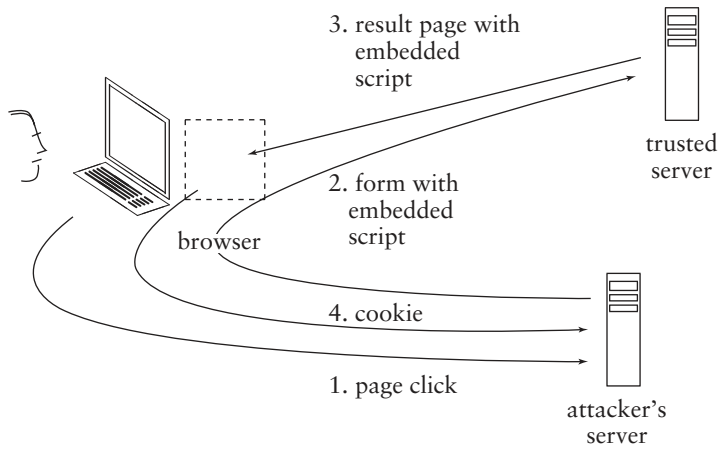


Figure 18.5: Reflected Cross-Site Scripting Vulnerability with Cookie Stealing

In *reflected* XSS (Figure 18.5) the script resides on a page on the attacker's server, and the victim has to be lured to this site first. The user action then sends the script to the trusted server, which has to echo back the client's input for the script to be executed at the client. In the following example, the attacker has prepared a page containing this element:

```
<A
HREF="http://trusted.com/comment.cgi?
mycomment=<SCRIPT alert('You have a XSS problem')></SCRIPT>">
Click here
</A>
```

When a victim clicks on the link to this element on the attacker's page, the URL sent to *trusted.com* includes the script in *mycomment*. If the trusted server echoes the value of *mycomment* in the result page, the script gets executed on the client within the page from the trusted server. In our example, the user will be alerted to the XSS vulnerability. Typical examples of applications echoing client input are search engines or custom 404 (not found) pages. There are many ways of embedding scripts. For example, the script may be fetched from the attacker's site via an image element:

```
mycomment=<IMG SRC='http://attacker.org/badfile'></IMG>">
```

In a *stored* XSS attack, the attacker places the script directly at the trusted server. Bulletin board applications are candidates for stored XSS. When the victim visits the attacker's bulletin board entry, the script embedded in the entry is executed at the client.

In a *DOM-based XSS* attack, the attacker injects the script via the `document` object in the DOM. The attack vector is split into two parts. The malicious script is embedded

in the URL of the attacker's web page; this page also contains a link to a page on the trusted server that references the URL in the DOM when being loaded by the browser. When the victim is lured into visiting the attacker's web page, the browser stores the bad URL in `document.URL` and requests the web page from the trusted server. When the browser loads the web page, `document.URL` will be referenced and the attacker's script will be executed.

18.4.1 Cookie Stealing

The browser stores cookies in `document.cookie`. Cookies are subject to the same-origin policy and are only included in requests to the domain that set the cookie. In a reflected XSS attack, the attacker's script executing on the client may read the client's cookie for the trusted server from `document.cookie` and send its value back to the attacker. This does not violate the same-origin policy as the script runs in the context of the attacker's web page.

A web page vulnerable to XSS can be exploited to capture data from another page in the same domain, which itself is not vulnerable to XSS. The attack script opens a window linked to the target page in the browser. To hide the attack from the user, the page could take over the entire browser window and open an inline frame to display the target page. The attack could use a *pop-under* window that sends itself to the background. In both cases, the rogue window is not visible to the user but has access to the DOM of the target page and can monitor the user's input.

18.4.2 Defending against XSS

The browser fails to enforce its code origin policy because it can just check the origin of the web page it downloads, but not the true origin of all the elements within it. For example, the browser authenticates the bulletin board service but not the user who placed a particular entry. *If the browser cannot authenticate the origin of all its inputs, it cannot enforce a code origin policy*. The defences against XSS fall into three general categories:

- Treat XSS as a code injection attack – encode server outputs, filter client inputs (see Section 10.7.4). A client may, for example, compare request and response to check whether a suspiciously large part of the request has been mirrored in the response.
- Change the modus operandi – disable execution of scripts.
- Improve authentication, or use only attributes for access control that can be authenticated.

You can protect the cookie by utilizing the browser's security policy, e.g. by putting the visited web page in a zone that is not given permission to access `document.cookie`. You could forgo the use of cookies for creating sessions and use other mechanisms for authenticating client requests (see Section 18.5). For example, unpredictable one-time URLs sent by the server to the client during user authentication can *authenticate* requests as coming directly from the client [131].

18.5 CROSS-SITE REQUEST FORGERY

A cross-site request forgery (XSRF, 'sea surf', also cross-site reference forgery, session riding) attack executes actions at a target website with the privileges of a 'trusted' user [50]. Here, trust means an authenticated session between client and web server. It does not matter how the session was established, whether by TLS, by password-based HTTP authentication, or by any other means. What matters is that requests within this session are executed with security permissions attributed to the client.

In a *reflected* XSRF attack the user has to visit the attacker's web page, which contains hidden actions at the target site, e.g. in an HTML form. Simultaneously, the client must have established an active session to the target. When the user visits the attacker's page, the browser automatically submits the form data to the target. The target interprets the request as coming from the client and the action in the form is accepted by the server as coming from an authenticated user. Thus, XSRF evades the target's origin-based security policy.

In a *stored* XSRF attack a malicious page is stored at the server (Figure 18.6). When a client visits this page, the client's browser will be directed back to the server and actions inserted by the attacker are executed as coming from the client. Stored XSRF attacks have a good chance of success as the client requesting the malicious content is likely to be authenticated and authorized to perform the actions.

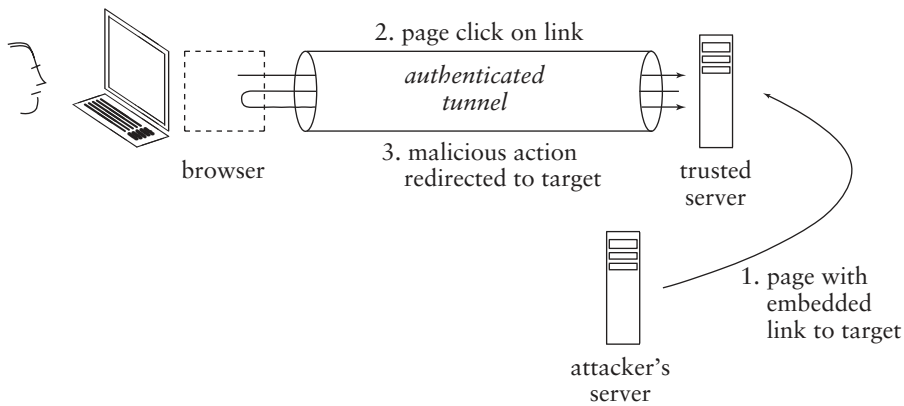


Figure 18.6: Stored Cross-Site Request Forgery Attack

With XSRF, the target fails to enforce its code origin policy because it can only authenticate the last stepping stone of a request, but not necessarily its true origin. To defend against XSRF you have to authenticate actions properly. For authentication you need a secret shared by client and server. This (temporary) secret can be sent (in the

clear!) from server to client when a session is being established. In the web attack model, a secret cannot be compromised in transit but only at vulnerable end systems. It is thus essential that the secret is stored in a location that is not accessible to scripts executing in the browser. If a page is vulnerable to XSS, authentication would be compromised.

To authenticate a request, the client constructs an authenticator derived from the secret. The authenticator could be an unpredictable session identifier used by all actions in the session, different actions could use individual authenticators, or the authenticator could be a message authentication code for the action as in

```
XSRFPreventionToken = HMAC(Action_Name+Secret, SessionID).
```

The browser sends the authenticator with the action. In a GET request, the authenticator is inserted as a token in the URI (also known as URI rewriting). In a POST request, the authenticator is sent in a hidden form field. The server authenticates any action request before execution. An attacker who does not know the secret is unable to form legitimate action requests. Action requests are now authenticated at the level of the web application, i.e. in a layer 'above' the browser. Cookies are not suitable for storing and transmitting the authenticator as they are sent automatically by the browser and may be stored beyond the duration of the session.

A proxy placed between browser and the network can implement a client-side defence for network application-layer sessions (but not for TLS sessions) by authenticating the origin of local requests [133]. The proxy marks all URLs in incoming web pages with an unpredictable token and keeps a database associating tokens with domains. The proxy also checks all outgoing requests for the presence of a token:

- If no token is found, the request is locally generated and can be sent in authenticated sessions.
- If a token is found and the origin of the request matches the domain it is being sent to, the request is permitted by the same-origin policy and can be sent in authenticated sessions.
- Otherwise, all authenticators (SIDs, cookies) added by the browser are stripped from the URI before sending the request.

18.5.1 Authentication for Credit

In the attacks against authentication protocols, the attacker has so far always attempted to impersonate someone else. These attacks wrongly assign responsibility (accountability). The victim may be held responsible for the attacker's actions. However, there are also attacks where the victim is made to impersonate the attacker. The actions of the victim are then credited to the attacker. For example, the attacker might become the owner of any files created by the victim and can later check what has been written.

The differences between authentication for responsibility and authentication for credit are discussed in [1].

Login XSRF [21] is an example of an attack of the second kind. The page prepared by the attacker contains a form that logs the attacker in at a target website. When the victim visits the attacker's page and triggers the action in the form, the target server will receive the attacker's credentials via the victim's browser and will associate with the attacker any input the victim enters in the page that has been opened (Figure 18.7).

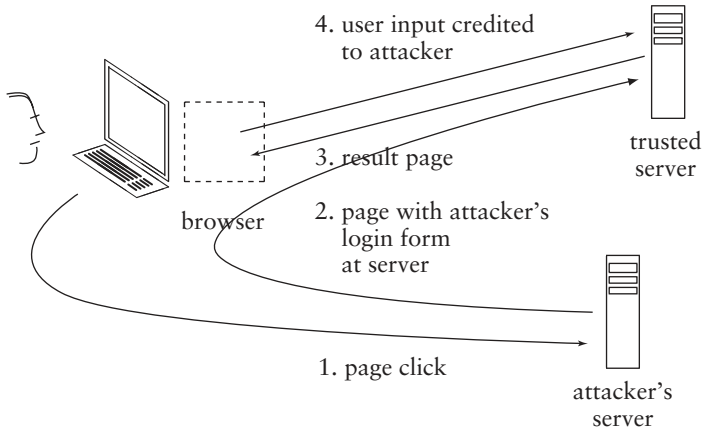


Figure 18.7: Login Cross-Site Request Forgery

18.6 JAVASCRIPT HIJACKING

Web 2.0 technologies combine features that facilitate the creation of new types of web applications. These features can also be the source of new vulnerabilities.

- AJAX (Asynchronous JavaScript and XML) supports asynchronous interactions between client and web server. The browser sends JavaScript requests to an AJAX engine, which handles the communication between client and web server (Figure 18.8).
- JSON (JavaScript Object Notation) is a format for data transport. A JSON string is a serialized JavaScript object, turned back into an object in the AJAX engine by calling *eval*() with the JSON string as the argument. The object is created using the JavaScript object *constructor*.
- A web server can dynamically update information held at the client. This is a useful feature for automatic news of software updates. A web server can manipulate the client's DOM via dynamic script tags and can override JavaScript constructors for its pages.

These features can be used to extend XSRF and disclose confidential data from the server to the attacker [62]. The first phase of a JavaScript hijacking attack follows the pattern

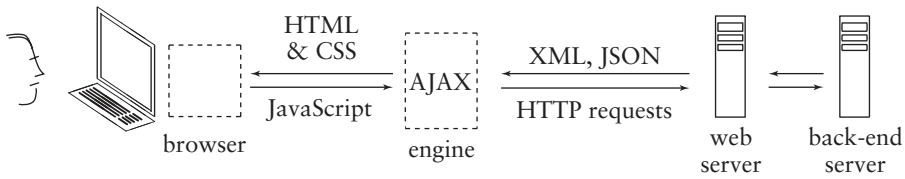


Figure 18.8: Web 2.0 Application

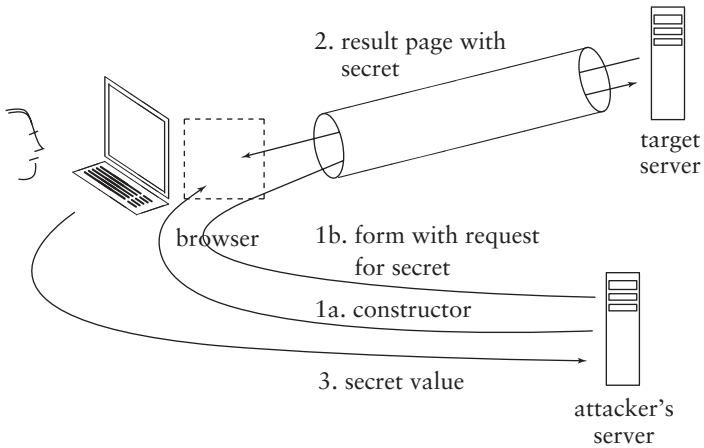


Figure 18.9: JavaScript Hijacking Attack

of XSRF. The user has to visit the attacker's web page and simultaneously have an authenticated session with the target server. The attacker's page includes a script that redefines a JavaScript constructor in the client's browser (step 1a in Figure 18.9) and a script within a request to the target server. The request asks for secret data the user is authorized to access. When the user clicks on the link on the attacker's page, the browser will send the request to the target site using the client's current session parameters, e.g. the client's cookie (step 1b). This request will be authenticated as coming from a legitimate user and the secret data is returned to the client (step 2).

The second phase of the attack fetches the secret from the client. The attacker's script overrides the native JavaScript object constructor in the browser. JSON arriving in the result page from the target site is processed by the AJAX engine. The modified constructor creates the JavaScript object, captures the secret data, and sends it back to the attacker (step 3). The execution is performed in the context of the attacker's web page so sending data to the attacker is legal.

Defences against the first attack phase are the same as for XSRF. Defences against the second phase change the modus operandi at the client. The server modifies its JSON

response so that it cannot be executed directly by the browser but has to be first processed by the requesting application. For example, each JSON response could be prefixed with a `while(1);` statement causing an infinite loop.[1] The application must remove this prefix before any JavaScript in the response can be run. Alternatively, the JSON could be sent as a comment. The application first has to uncomment the JSON received. In both cases, JavaScript in the response must be executed at the client in the context of the application. The malicious web page cannot remove the block.

Lesson
Browser and server communicate via a transport protocol. When assessing a transport protocol, look not only at the data formats but also at the algorithms for packaging and unpacking application payloads.

18.6.1 Outlook

The web execution model is still evolving, and important security aspects may change rapidly. A Web 2.0 security issue to look out for are *mashups*, i.e. web applications combining content from multiple sites. Access control in this setting has to move beyond the same-origin policy. It must be possible to specify how applications may interact. To enforce those policies, it must be possible to authenticate services. Authentication might:

- verify the name of the service (the familiar interpretation).
- recognize that it is the same party as last time (*recognition*). This is a very useful property facilitating an inductive security argument: if you have been to the right site the first time, you will always return to the right site.
- pick out material planted by someone else on your system ('know thyself').
- check that the party at the other end is a human and not a bot. Visual puzzles (CAPTCHA – Completely Automated Public Turing test to tell Computers and Humans Apart) are being applied for this purpose.

Then there is the question of who sets the policy. For example, with the JavaScript *callback* function it is the web page that tells the browser what to do next. Section 17.2.6 has shown the problems that may arise when the client trusts the server on policy.

18.7 WEB SERVICES SECURITY

The web was originally designed to give users access to resources hosted on the Internet. Web services build on the communications infrastructure that was created and use the web to facilitate computer-to-computer interaction. Web services are an architectural

[1] This was the defence adopted by Gmail when hit by a JavaScript hijacking attack in 2006.

paradigm for implementing distributed applications. Loosely coupled applications need standards for encoding documents and for transferring messages, there has to be a specification of the service delivered, and users have to be able to find the services on offer. The standards constituting the foundations of Web services are:

- HTTP for message transfer;
- XML for encoding documents;
- SOAP for encoding messages and defining elementary message patterns;
- WSDL (Web Services Description Language) for describing services;
- UDDI (Universal Description, Discovery and Integration) for describing entries in service directories, and for publishing and retrieving service descriptions.

Distributed applications have to protect data in transit and in the end systems. For the former, standards for encrypting and signing documents are needed, and communicating parties have to agree on the cryptographic protection applied to message exchanges. WS-Security standardizes the cryptographic protection of SOAP messages. WS-Policy is the standard for describing policies on required cryptographic protection. Section 18.7.1 has a closer look at the issues arising when digitally signing business documents encoded in XML.

For protecting data in end systems, access control policies have to be specified and enforced. XACML is an XML-based standard for writing policies (Section 18.7.3). User authentication and the exchange of access credentials (Kerberos tickets, certificates, security tokens) are defined at an abstract level to be independent of specific authentication technologies. SAML specifies single sign-on patterns (Section 18.7.2). WS-Trust is a framework for requesting and issuing access credentials.

18.7.1 XML Digital Signatures

In Chapter 14 digital signature algorithms took bit strings as their input. At the application layer you are signing documents, not bit strings. Documents (e.g. XML documents) may have different but equivalent representations. Documents have internal structure. Documents in a workflow change as they are being processed. Some parts of a document may not yet have been completed when a document is signed. A signer may only sign parts of a document.

The administrative steps around a business trip may illustrate some of those issues. You apply to your manager. The manager approves the travel request. After the trip, you file an expenses claim. The manager has to approve the claim. The finance department has to authorize payment of the expenses. Several parties are involved in this process. It may not be meaningful for each party to sign the entire document passed to it.

If we simply hash documents and sign bit strings, and if signer and verifier use different but equivalent representations, or if parts of the document have changed for legitimate

reasons, signature verification fails. At the application level we have to do some additional work to prepare a document for signing. The XML-DSIG standard [20] gives a format for signed XML documents:

```
<Signature ID?>
      <SignedInfo>
            <CanonicalizationMethod/>
            <SignatureMethod/>
            (<Reference URI?>
                  (<Transforms>)?
                  <DigestMethod>
                  <DigestValue>
            </Reference>)+
      </SignedInfo>
      <SignatureValue>
      (<KeyInfo>)?
      (<Object ID?>)*
</Signature>
```

The *canonicalization method* converts a document to a given canonical form. Verifiers must exercise care when accepting unknown canonicalization methods. The method might modify the document to the extent that validation will always succeed.

Inclusive canonicalization copies all name space declarations currently in force, even if they are defined outside of the scope of the signature. This guarantees that all declarations that might be used are unambiguously specified. However, putting a signed document into an XML document that has other declarations will invalidate the signature.

Exclusive canonicalization includes only *visibly used* name spaces and does not look into attribute values or element content. Name space declarations required there are not covered. There is the option of explicitly specifying name spaces that must be declared and thus go beyond the visibly used name spaces. For this option, the signing software must know the relevant name spaces. Exclusive canonicalization is useful when signing documents that should be inserted into other XML documents. The signer is likely to be aware of the relevant name spaces to be included in the canonicalization.

The *SignatureMethod* gives the digital signature algorithm. The *Reference* fields are URIs identifying the data objects to be signed. *Transforms* describe how the signer modified the object before it was hashed (digested). For example, a digital signature might only apply to parts of the document. *DigestMethod* gives the hash function, *DigestValue* the hash of the transformed object. The verifier may obtain the hash value in some other way, e.g. by obtaining the object from a location other than that specified in the URI. *SignatureValue* contains the signature of the document. *KeyInfo* gives the verifier information about the

key to use. The *object* fields may include parts of the signed document in the signature (*enveloping* signature).

The *core validation* algorithm canonicalizes the *SignedInfo* element, recomputes the hash values for all (transformed) references, and compares the values obtained with the values in *SignedInfo*. If there is any mismatch, signature validation fails. Otherwise, processing continues to the verification of the digital signature proper.

There exist applications where some parts of a document matter less than others, so that processing of the document can proceed as long as the signatures on the important parts can be verified. Such an application cannot use the core validation algorithm, but can implement its own customized validation algorithm and collect *Reference* fields in a *Manifest*. Such references will not be checked by core validation.

18.7.2 Federated Identity Management

A digital identity is an electronic representation of a real-life entity. It can refer to persons, organizations, or machines. Digital identities can be names, social security numbers, user identities, host names, IP addresses, and so on. *Identity management* is the process of creating and removing digital identity accounts (provisioning), and of managing authenticating and authorization (access management).

Federated identity management refers to the management of identity information between organizations. A person may use the same username, password or other credential to sign on to the networks of more than one enterprise. Federated single sign-on implements a 'trust, but verify' strategy. A user is authenticated at one site, but gets access to resources at other sites. Partners in a federation rely on each other to authenticate their respective users and vouch for their access to services. A federation needs an agreement between partners specifying the rules governing the exchange of identity information, addressing e.g. legal liability and dispute resolution. Privacy aspects have to be reconsidered as identities are no longer used exclusively by the issuing organization. At a technical level there has to be agreement on the security protocols to be used.

Security Assertion Markup Language (SAML) is a meta-level single sign-on protocol [183]. It could be implemented by Kerberos or by PKI-based protocols. An *asserting party* (SAML authority) makes *assertions* about a subject that other applications within the federation may decide to rely on. An assertion could indicate the identity of the user authenticated, privileges granted to that user, or the strength of the authentication mechanism used. SAML defines a number of mechanisms that enable the *relying party* to *trust* the assertions provided to it. Here, trust means that the origin and integrity of assertions can be verified. The relying party decides whether to *trust* the assertions provided to it. Here, trust means that the relying party decides whether to give access to local resources based on its own local security policy.

SAML defines two profiles for conveying an assertion about a user from the asserting party to the relying party. It is assumed that the user interacts with both parties using a standard web browser. In the *Browser/Artefact Profile* the relying party pulls the assertion (Figure 18.10). The user is authenticated to the asserting party and requests an assertion. The asserting party returns a handle to the assertion as an HTTP query variable called an *artefact*. The browser passes the artefact to the relying party, which sends an SAML request containing the artefact to the asserting party. The assertions about the user are then transferred back in a SAML response.

In the *Browser/POST Profile* the assertion is pushed to the relying party (Figure 18.11). A user authenticated at the asserting party receives an HTML form with an assertion about the user. The form contains a trigger causing a POST of the assertion to the relying party.

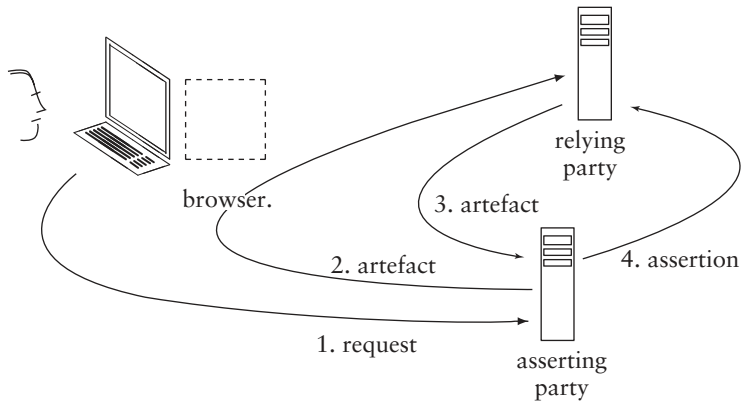


Figure 18.10: SAML Artefact Profile

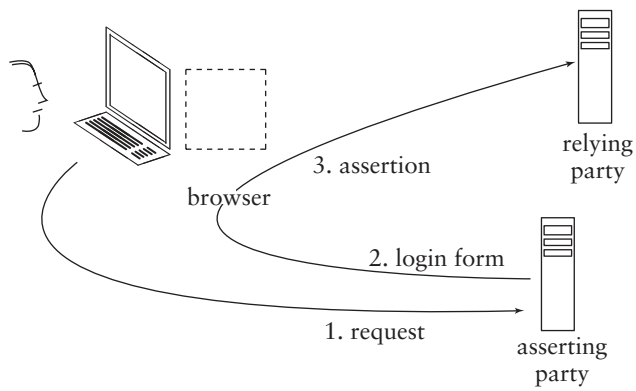


Figure 18.11: SAML Browser/POST Profile

There is a further facet of federated identity management. A person may have several identities used in different contexts, but may on occasion want to link these identities. For example, when an airline and a car rental company have a deal whereby frequent flyers can earn miles by renting from that company, passengers might be willing to disclose their frequent flyer number to the car rental company. Protocols where the user controls how different identities can be linked have been standardized by the Liberty Alliance.[2]

18.7.3 XACML

In heterogeneous access control environments, be it within a federation or within a single organization, it is desirable to describe policies for the different components in a common policy language. XACML is a descriptive policy language for such environments [184]. XACML policies provide an abstraction layer shielding policy-writers from the details of the application environment. Access requests are submitted to a policy enforcement point (PEP). The PEP asks the policy decision point (PDP) for a decision. The PDP may have to collect further evidence from policy information points (PIPs) to make a decision. In the end the decision is returned in a *response context* to the PEP, where it is enforced (Figure 18.12).

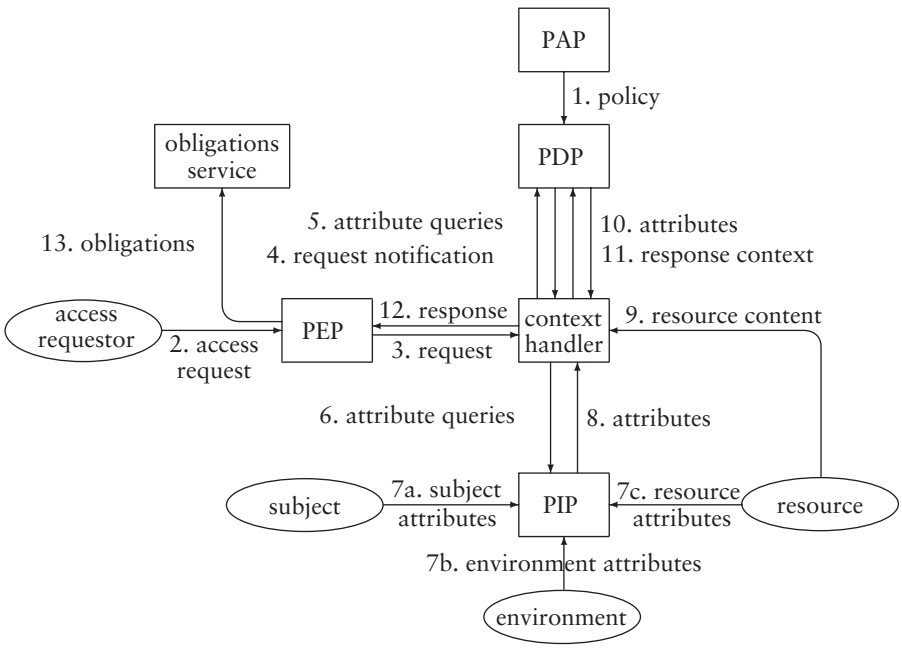


Figure 18.12: Message Flow during XACML Policy Evaluation

[2] http://www.projectliberty.org/

An XACML *policy* consists of a set of rules, a rule combining algorithm, and optionally includes *obligations*. An obligation is an operation the PEP must perform when executing the authorization decision obtained from the PDP. For example, the PEP could be told to log access operations. A *rule* consists of target, effect, and condition. The *target* is the set of decision requests that should be evaluated. Targets are identified by resource, subject, action, and environment. The *effect* can be *permit* or *deny*. The *condition* collects additional logic predicates required for the rule. Conditions can evaluate to *true*, *false* and *indeterminate*.

The *resource* in a target is the object of the access control rule. The *subject* is the entity making the request. Rules can refer to attributes of the subject and to the content of the resource. The *action* is the operation performed on the resource and can be application-specific. The *environment* gives attributes relevant to an authorization decision that are independent of subject, resource or action, e.g. the current date and time.

An XACML policy has to specify *matching functions* to match request attributes against policy targets. Once the rules in a policy have been evaluated, their results have to be combined. There exist several options for a *rule combining algorithm*. Typical examples are *deny-overrides* giving precedence to negative permissions and *first applicable*, often used when policies are specified as a list of rules. With *first applicable*, the decision may depend on the order in which rules are listed. We mention *only-one-applicable* as a further option. This algorithm returns 'NotApplicable' if it finds no applicable policy for a target, and 'Indeterminate' if it finds more than one policy.

18.8 FURTHER READING

For an early description of cross-site scripting, see [60]. For an update on web attacks and defences, consult the OWASP website (`www.owasp.org`).

OpenID is an HTTP-based implementation of a federated single sign-on protocol (see `http://openid.net/`). The identities used are URIs and can be issued by any OpenID identity provider. The message flow is similar to the SAML Browser/POST profile, with an additional association establishment between identity provider and relying party.

On XACML, look out for updates on performance measurements. Cost factors when processing XACML are the marshalling and unmarshalling of XML, the look-up of missing attributes, and the evaluation of the access control logic itself.

18.9 EXERCISES

Exercise 18.1 Document the current security settings of your web browser. Where is the security-relevant information stored on your system?

Exercise 18.2 Find the cookie file on your system and the options for setting cookie retention policies.

Exercise 18.3 Consider an application that uses predictable cookies to maintain session state. How could an attacker hijack a session of some other user? How could the application be secured?

Exercise 18.4 Validate the claim that the POST method is more secure than the GET method when requesting an action.

Exercise 18.5 List the dangerous characters that have to be removed from input to the client to prevent XSS attacks. What can the server do to avoid including these characters in the responses sent to the client?

Exercise 18.6 Investigate the methods for embedding malicious scripts in a web page.

Exercise 18.7 CRLF (carriage return, line feed) separates HTTP response headers; a double CRLF separates the headers from the body of the response (RFC 2616). Explain how CRLF can be used in an XSS attack if the server echoes CRLF in input received from the client.

Exercise 18.8 You are given an application where many XML documents are signed by many different keys. How can the provisions in XML-DSIG be used most efficiently to support this application?

Exercise 18.9 Would you prefer inclusive or exclusive canonicalization when signing SOAP messages?

Exercise 18.10 Examine the attacks that are possible if assertions do not identify the specific request and relying party they should be used with.

Chapter 19

Mobility

The first mobile IT service to find wide acceptance was the second-generation digital cell phone network. The number of mobile services has since grown, supporting a wider range of applications using different underlying technologies. Mobile services pose new security challenges. Some challenges derive from the technology. Messages transmitted over a radio link can be intercepted by third parties. Secure sessions should persist when a device changes its point of attachment. To access a wireless network you do not have to attach a cable to a socket, you only need to be within range of an access point. Physical access control to buildings or rooms is no longer an effective barrier keeping unauthorized users out. Other challenges derive from the applications. Frequently, the entity offering a service to its subscribers is different from the operator managing the local network providing access to the service. The security interests of all three parties – subscriber, network operator and service provider – have to be taken into account, and possibly also the requirements of law enforcement agencies.

OBJECTIVES

- **Examine new security challenges and attacks specific to mobile services.**
- **Give an overview of the security solutions adopted for different mobile services.**
- **Show some novel ways of using cryptographic mechanisms.**
- **Discuss the security aspects of location management in TCP/IP networks.**

19.1 INTRODUCTION

The first-generation analogue mobile cell phone network provided direct dialling, automatic handover between cells, and call forwarding. As a general comment on security, note that criminals tried to use the latter feature for creating alibis, and that telecommunications experts were then called upon by courts to explain why a person who had answered a call at a particular land line number was not necessarily at this place at the time of the call. A challenge-response protocol was used for authentication but the relevant secrets were transmitted in the clear, so that some networks suffered a high level of charge fraud. There was some obfuscation of voice traffic, but no strong protection against eavesdropping.

19.2 GSM

Against this backdrop of low security in the first-generation network, the Groupe Spéciale Mobile (GSM) study group of the European Conference Postal and Telecommunications Administrations (CEPT) was founded in 1982 to specify the second-generation mobile network. The design goals for this digital network were good subjective voice quality, cheap end systems, low running costs, international roaming, handheld mobile devices, ISDN compatibility, and support for new services such as SMS. In 1989 the responsibility for GSM was transferred to the European Telecommunication Standards Institute (ETSI). Phase I of the GSM specification was published in 1990 and GSM was renamed the Global System for Mobile Communications.

To understand some of the design decisions made, the reader has to be conscious of the political influences on the development of GSM. An international system has to take into account various national regulations and attitudes to the public use of cryptography. Restrictions on the use of strong cryptography were an issue until the mid 1990s. Moreover, law enforcement authorities requested support for conducting authorized 'wiretaps', in analogy to wiretaps in the fixed network. The partners in the GSM consortium were mainly national post, telephone and telegraph operators. This influenced the view on the trustworthiness of the parties running the GSM network.

The main security goals of GSM are protection against charge fraud (unauthorized use of a service) and the protection of voice traffic and signalling data on the radio channel. Once traffic is in the fixed network, no added cryptographic protection is provided. Thirdly, there is a contribution to physical security. It should be possible to track stolen end devices. This feature was not always implemented.[1]

[1]At a time when cell phone robberies among school children became a non-negligible item in the UK crime statistics, service providers were strongly encouraged to implement the mechanisms for tracking stolen phones.

19.2.1 Components

Each GSM user has a subscription in a *home network*. The network where a service is requested is called the *visited network* (or serving network). A mobile station (MS) or cell phone consists of the mobile equipment (ME) and the subscriber identity module (SIM). The SIM is a smart card chip that performs cryptographic operations in the MS and stores the relevant cryptographic keys. The SIM may also contain other personal data of the subscriber, such as a personal phone book, and gives personal mobility independent of the ME. On the network side, there is the base station (BS), the mobile switching centre (MSC), the home location register (HLR) of a subscriber, the authentication centre (AuC), and the visitor location register (VLR). The HLR and VLR manage call routing and roaming information. The AuC manages a subscriber's security-relevant information. The relationship between different network operators is managed through service level agreements (SLAs) and the GSM Memorandum of Understanding (GSM/MoU).

The identifier for a GSM subscriber is the international mobile subscriber identity (IMSI). Subscriber and HLR/AuC share a secret 128-bit individual subscriber authentication key *Ki*. The SIM stores *Ki*, the IMSI, the TMSI (Section 19.2.2), and a current 64-bit encryption key *Kc*. Algorithms A3 and A8 (Section 19.2.3) are implemented in the SIM. Access to the SIM is controlled by a personal identification number (PIN). The SIM is locked after three attempts with an incorrect PIN. A SIM can be unblocked using a personal unblocking key (PUK).

19.2.2 Temporary Mobile Subscriber Identity

When an MS connects to the network it has to identify itself by some means. If a fixed identity is used at each call, the movements of subscribers can be tracked, even if subsequent traffic is encrypted. As a step towards better subscriber privacy, the unencrypted IMSI is sent only when an MS makes initial contact with the GSM network. Thereafter a temporary mobile subscriber identity (TMSI) is assigned in the visited network and used in the entire range of the MSC. The IMSI is thus not normally used for addressing on the radio path. The VLR maintains a mapping $\langle$(TMSI, LAI), IMSI$\rangle$ from TMSI and location area identity (LAI) to IMSI. When an MS moves into the range of another MSC a new TMSI is assigned. When permitted by signalling procedures, signalling information elements that convey information about the mobile subscriber identity are encrypted for transmission on the radio path.

Lesson
Protection of location information is a security issue specific to mobile services. Fixed identifiers leak information about the movement of a mobile station.

19.2.3 Cryptographic Algorithms

GSM uses symmetric cryptography for encryption and subscriber authentication. The use of public-key cryptography was considered, but when decisions were held in the 1980s public-key cryptography was not yet a feasible option with the technology of the time. There are three cryptographic algorithms: the authentication algorithm, A3; the encryption algorithm, A5, used on signalling and user data; and the key generation algorithm, A8. These algorithms were not published outside the GSM/MoU, but were eventually leaked or reverse-engineered.

Algorithms A3 and A8 are shared between subscriber and home network. Hence, each network may choose its own algorithms A3 and A8. Only the formats of their inputs and outputs must be specified. A3 and A8 compute a response RES and ciphering key *Kc* from a random challenge RAND and the key *Ki*. Processing times should remain below a maximum value, e.g. 500 milliseconds for A8. Proposals for A3 and A8 are managed by the GSM/MoU.

Algorithm A5 has to be shared between all subscribers and all network operators. This algorithm has to be standardized. There are three versions: A5/1, a less secure 'export' version A5/2, and a stronger version A5/3 introduced later. Cryptanalytic attacks against all three versions have been published.

19.2.4 Subscriber Identity Authentication

Subscriber authentication is triggered by the network at the first network access after a restart of the MSC/VLR or when the subscriber applies for access to a service, e.g. set-up of a mobile originating or terminated call. Authentication is also performed when the subscriber applies for a change of subscriber-related information in the VLR or HLR, e.g. location updating involving change of VLR, or in the event of cipher key sequence number mismatch.

Figure 19.1 describes the message flow during authentication. The initial message from ME to VLR contains a subscriber identity, either TMSI or IMSI. The VLR maps TMSI to IMSI and forwards the IMSI to the HLR/AuC over the fixed subnet. The AuC generates a non-predictable 128-bit challenge RAND and computes the response RES = A3(*Ki*, RAND) and a 64-bit encryption key *Kc* = A8(*Ki*, RAND). The triple ⟨RAND, RES, *Kc*⟩ is sent to the VLR. The VLR stores RAND and *Kc* and passes the challenge RAND on to the MS. The key *Kc* is only valid within one location area.

In the MS, the response SRES = A3(*Ki*, RAND) is computed in the SIM ('signature' in GSM terminology, but not a digital signature) and transmitted by the MS back to the VLR. The VLR compares SRES and RES. Authentication succeeds if the two values match. To speed up subsequent authentications in a visited network, the AuC sends

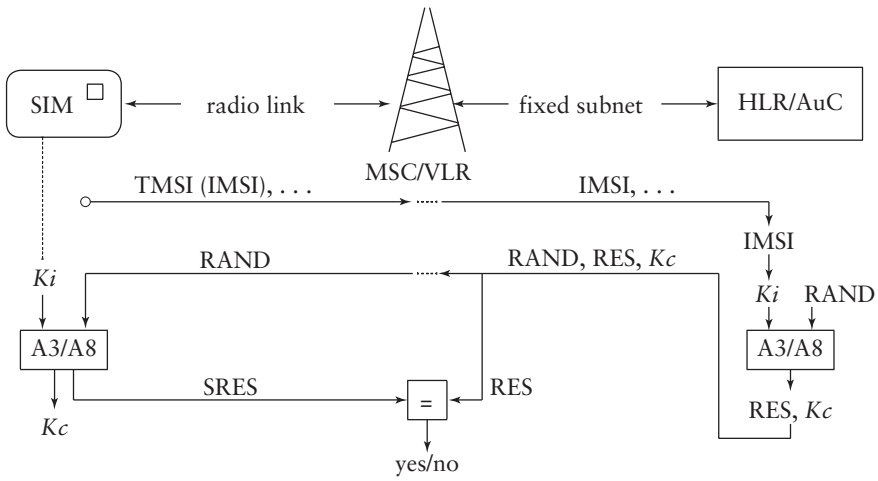


Figure 19.1: Subscriber Identity Authentication in GSM

several triplets ⟨RAND, SRES, *Kc*⟩ to the VLR, which are then used in turn for subscriber authentication.

19.2.5 Encryption

Normally, all voice and non-voice traffic on the radio link is encrypted. The infrastructure is responsible for deciding which algorithm to use, or whether to switch off encryption so that no confidentiality protection is afforded. If necessary, the MS signals to the network which encryption algorithms it supports. The serving network then selects one based on a priority order preset in the network, and signals the choice to the MS.

The *ciphering indicator* feature allows the ME to detect that ciphering is not switched on and to flag this to the user. This feature may be disabled by the home network operator by setting the administrative data field (EF_{AD}) in the SIM accordingly (GSM 11.11). If it is not disabled in the SIM, then whenever a connection is or becomes unenciphered, an indication shall be given to the user (GSM 02.07 version 7.0.0, Release 1998).

The encryption algorithm A5 is a stream cipher applied to 114-bit frames. The key for each frame is derived from the secret key *Kc* and the current 22-bit frame number (Figure 19.2). Radio links can be noisy, so a stream cipher is preferable to a block cipher. With a block cipher, a single bit error in the cipher text affects an entire plaintext frame. In a stream cipher, a single bit error in the cipher text affects a single plaintext bit. By today's standards a key length of 64 bits is short and cryptanalysis of A5 has further reduced the effective key length.

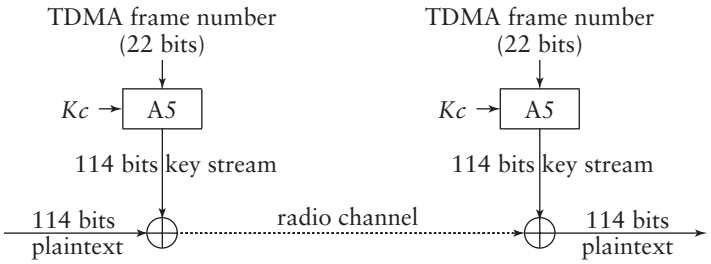


Figure 19.2: Encryption of GSM Frames from MS to MSC

Lesson
The characteristics of the physical network layer can be relevant for the choice of cryptographic algorithms.

19.2.6 Location-Based Services

The GSM network records the location of mobile equipment. This information can be used to offer location-based services such as traffic information for motorists. There are also emergency location services that give the location of an ME making a call to an emergency number. This service is obligatory in some countries – e.g. US Federal Communications Commission regulations demand mobile location to an accuracy of 50–300 metres, depending on the technology used. By 2012 more stringent requirements will be imposed.

19.2.7 Summary

Challenge-response authentication in GSM does not transmit secrets in the clear, so one major vulnerability of the first-generation network has been removed. Voice traffic is encrypted over the radio link but calls are transmitted in the clear after the base station. There is optional encryption of signalling data, but the ME can be asked to switch off encryption. There is some protection of location privacy through the TMSI, but attackers can use so-called IMSI catchers that ask the MS to revert to initial authentication using the IMSI. This attack is possible because the network is not authenticated to the MS. Law enforcement agencies have access to recorded movement data of subscribers. There is a separation of subscriber identity (IMSI) from equipment identity (IMEI), and there are provisions for tracking stolen devices.

The main criticisms of GSM security are directed against the decision not to publish the cryptographic algorithms so that they could be publicly scrutinized, and against the decision to provide only unilateral authentication of the subscriber to the network, but no authentication of the network to the subscriber.

An overall assessment of GSM security must look beyond the technical security features. Many cases of GSM fraud attack the *revenue flow* rather than the data flow and do not break the underlying technology. In *roaming fraud*, subscriptions are taken out with a home network. The SIM is shipped abroad and used in a visited network. The fraudster never pays for the calls (soft currency fraud), but the home network also has to pay the visited network for the services used by the fraudster (hard currency fraud). There is obvious scope for fraudsters and rogue network operators to collude. In premium rate fraud, unwitting customers are lured into calling premium rate numbers owned by the attacker, who uses the existing charging system to get the victim's money. Criminals may also start a premium rate service, make fraudulent calls to their own numbers to generate revenue, collect their share of the revenue from the network operator, and disappear at the time the network operator realizes the fraud.

Countermeasures are to be found at the human level (e.g. exercising caution before answering a call-back request), in the legal system (e.g. clarifying how user consent has to be sought for subscribers to be liable for charges to their account), and in the business models of network operators. GSM operators have taken the lead in using sophisticated fraud detection techniques to detect fraud early and limit their losses.

Lesson
Do not lose sight of application-level attacks when analyzing the security of a service provided to end users.

19.3 UMTS

Work on third-generation (3G) mobile communications systems started in the early 1990s. One of those systems is the Universal Mobile Telecommunications System (UMTS). The standardization organization for UMTS is the 3G Partnership Project (3GPP). The organizational partners of 3GPP (as of spring 2010) are ARIB (Japan), ATIS (North America), CCSA (China), ETSI (Europe), TTA (South Korea) and TTC (Japan). The first UMTS specifications were released in 1999. The UMTS security architecture is similar to that of GSM. Subscribers have a universal subscriber identity module (USIM) that is part of the user equipment (UE) and share a secret key with the AuC in the home network. The UE requests services from a visited network or a serving GPRS support node (SGSN).

19.3.1 False Base Station Attacks

In GSM, the network is not authenticated to the ME. Hence, the ME cannot tell whether requests to use the IMSI for authentication or to switch off encryption are genuine or come from a bogus base station. To address this problem, one might call for mutual

authentication between ME and network. At this point it matters how we interpret the 'network'. We might refer to the entire UMTS network so authentication proves that signalling comes from a genuine operator and not a bogus base station. This is the traditional viewpoint of the network operators and can be found in earlier UMTS documentation. The other viewpoint is that of a subscriber making a call who wants to be sure about the identity of the network (operator) handling the call.

There are several reasons that militate against authenticating the visited network. The ME has a pre-established relationship only with the home network, so direct authentication of a visited network is not feasible. Extending the challenge-response protocol to give mutual authentication would not prevent false base station attacks either. The attacker need only wait for authentication with the genuine base station to complete and then take over communications with the ME by sending with a stronger signal than the genuine BS. Furthermore, cryptographic authentication mechanisms are of limited use on noisy channels. Any bit error in a message will cause authentication to fail. So, the longer the authenticated message, the larger is the likelihood that it will be rejected because of a transmission error.

Section 19.3.3 covers the defences against false base station attacks adopted in UMTS. A deeper discussion of the design rationale for these mechanisms is given in [171].

19.3.2 Cryptographic Algorithms

The UMTS authentication functions $f1$ and $f2$ and the key generation functions $f3$, $f4$, and $f5$ can be specific to the service provider. The MILENAGE framework, a recommendation for authentication and key agreement (AKA) functions, has a block cipher with 128-bit blocks and 128-bit keys at its core. The encryption algorithm for the radio link and the integrity check algorithm for signalling data on the radio link have to be standardized. UMTS has adopted KASUMI, an eight-round Feistel cipher with 64-bit blocks and 128-bit keys. KASUMI is used in a variation of output feedback mode for encryption and in a variation of CBC-MAC mode for integrity protection. All algorithms proposed for UMTS are published and have been subject to thorough cryptanalysis.

19.3.3 UMTS Authentication and Key Agreement

Home network (AuC) and subscriber (USIM) share a secret 128-bit key K and maintain a synchronized sequence number SQN. In response to an authentication request, the AuC generates a random challenge RAND and an expected response XRES $= f2$(RAND,K). The AuC also computes a 128-bit cipher key CK $= f3$(RAND,K), a 128-bit integrity key IK $= f4$(RAND,K), a 48-bit anonymity key AK $= f5$(RAND,K), and a message authentication code of the challenge RAND, a sequence number SEQ, and an authentication management field AMF that may contain a key lifetime (Figure 19.3). The AuC

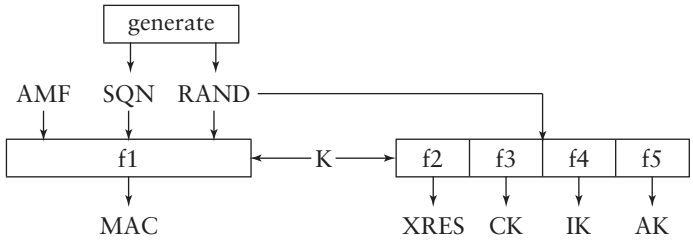


Figure 19.3: Generation of Authentication Vector at AuC

then constructs AUTN = ⟨SQN⊕AK, AMF, MAC⟩ and sends the authentication vector ⟨RAND, AUTN, XRES, CK, IK⟩ to the VLR/SGSN, which stores ⟨RAND, AUTN, XRES, CK, IK⟩ for the IMSI and passes RAND and AUTN to the UE.

Upon receipt of RAND and AUTN, the USIM first computes AK = $f5$(RAND, K) and SQN = (SQN ⊕ AK) ⊕ AK (Figure 19.4). Then the expected message authentication code XMAC is derived from RAND, SQN and AMF, and compared with the message authentication code value received. This verifies that RAND and AUTN had been generated by the home AuC. If there is a mismatch, the USIM aborts the protocol run, sending a reject message to the VLR. Otherwise, the USIM continues the protocol and checks that SQN is valid to detect replay attacks. When this check fails a synchronization error is signalled to the VLR. False base station attacks are thus prevented by a combination of key freshness and integrity protection of signalling data, not by authenticating the serving network.

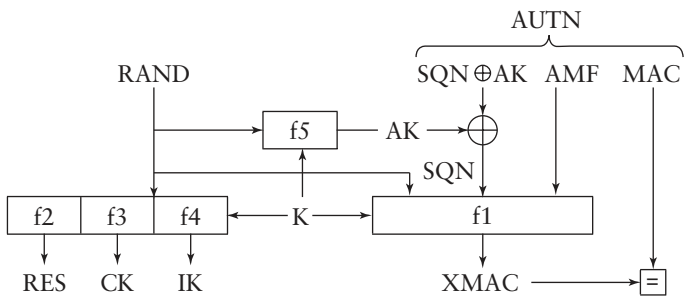


Figure 19.4: Authentication in USIM

Finally, the USIM computes the response RES and the keys CK and IK from the challenge RAND and its secret key K, and returns the response RES to the VLR. The VLR compares RES and XRES to authenticate the USIM (Figure 19.5).

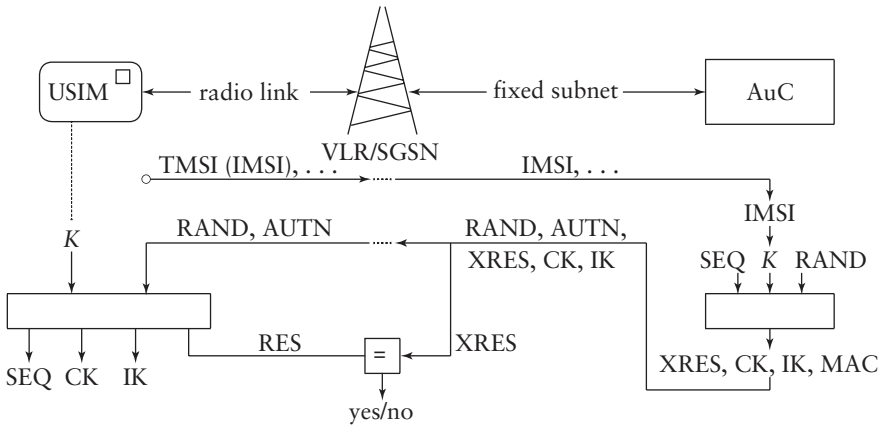


Figure 19.5: Authentication and Key Agreement in UMTS

19.4 MOBILE IPv6 SECURITY

The security problem in GSM and UMTS was access control to services when users are mobile and do not have a pre-established relationship with the operator of the network where the service is delivered. In this section we will use Mobile IPv6 (MIPv6) to discuss another mobility problem. When a device changes its location, how can other nodes verify that information about the new location of the device is correct? The standard remedy in such a situation is authentication, but we run into two problems. To authenticate a message, we have to know the identity of the sender. In IP, the identities are IP addresses. To verify cryptographically that a message comes from a claimed address, we have to reliably associate a cryptographic key with this address. The use of a PKI was suggested for this purpose, but currently there is no such PKI we could readily use and it is unlikely that this situation will change anytime soon.

Secondly, data origin authentication does not solve our problem. In the 'old' setting of the wired network, a node could lie about its identity. Alice could claim to be Bob to get messages intended for Bob. A mobile node can lie about its identity and about its location. Alice could claim that Bob is at her location so that traffic intended for Bob is sent to her. This is still a variation of the 'old' attack. Alice could also claim that Bob is at a non-existent location so that traffic intended for him is lost. Both attacks could be stopped by checking that Bob gave the information about his location.

There is yet another denial-of-service attack. Alice could claim that she is at Bob's location so that traffic intended for her is sent to Bob. In a *bombing attack*, Alice orders a lot of traffic and has it delivered to Bob's location. Verifying that the information about Alice's location came from Alice does not help in this case. The information came from her, but she lied about her location.

Bombing attacks are a flow control issue. Data is sent to a victim who did not ask for it. Authenticating the origin of location information does not prevent bombing. It would be more appropriate to check whether the receiver of a data stream is willing to accept the stream. Instead of origin authentication we require an authorization from the destination to send traffic to it. Flow control issues should in principle be handled at the transport layer. It was decided to address this issue at the IP layer because otherwise all transport protocols would have to be modified.

Lesson
Mobility changes the rules of the security game. In a fixed network, nodes may use different identities in different sessions (e.g. NAT in IPv4), but in each session the current identity is the location messages are sent to. With mobile nodes, we have to treat identity and location as separate concepts.

19.4.1 Mobile IPv6

An IPv6 address specifies a node and a location. A 128-bit MIPv6 address consists of a 64-bit subnet prefix (location) and a 64-bit interface ID (identity within the location indicated by the prefix). IPv6 addresses of mobile nodes and stationary nodes are indistinguishable. A mobile node is always addressable at its *home address* (HoA), whether it is currently attached to its home link or away from home. The home address is an IP address assigned to the mobile node within its home link. The mobile node and its home agent have a preconfigured IP security association that constitutes a secure tunnel. RFC 3776 specifies the use of Encapsulating Security Payload to protect MIPv6 signalling between mobile and home agent. While a mobile node is attached to some foreign link away from home, it is also addressable at a *care-of address* (CoA). This care-of address is an IP address with a subnet prefix from the visited foreign link.

19.4.2 Secure Binding Updates

The association between a mobile node's home address and care-of address is known as a binding for the mobile node. Away from home, a mobile node registers a *binding* with a router on its home link, requesting this router to function as its *home agent*. Any other nodes communicating with a mobile node are called *correspondent nodes*. Mobile nodes can inform correspondent nodes about their current location using binding updates. The correspondent node stores the location information in a binding cache. Binding updates refresh the binding cache entries. Packets between the mobile node and the correspondent node are either tunnelled via the home agent, or sent directly if the correspondent node has a binding for the current location of the mobile node.

Binding updates are in essence a network management task, but can be performed by any node. If binding updates cannot be verified, attackers could create havoc with the

Internet, including the wired Internet. Not surprisingly, it is a problem if any node can participate in – and interfere with – network management. These security concerns halted work in the IETF on mobile IP for a while. The first attempt at securing binding updates suggested using IPsec. This proposal applies an old solution to a new problem and has serious deficiencies. Security associations would have to be established, but IKE is a heavyweight protocol not suited for mobility. A chain of trust between mobile node and correspondent node would have to be constructed in the absence of a global PKI for the Internet.

Thus, dedicated protocols are needed for verifying that a node with a claimed identity is in its claimed location. A secure binding update protocol for MIPv6 is specified in RFC 3775 (Figure 19.6). The design considerations behind this protocol are explained in [18]. The primary security goal was that mobility must not weaken the security of IP, in particular with respect to nodes not involved in an exchange, e.g. nodes in the fixed Internet. The protocol also has several features providing resilience against denial-of-service attacks.

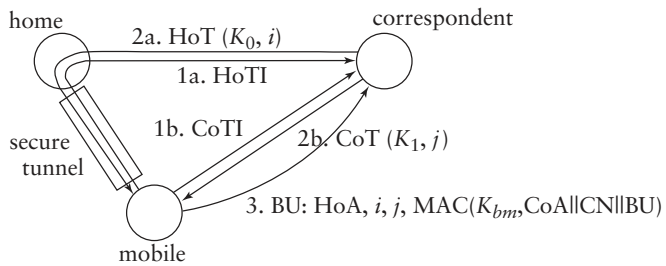


Figure 19.6: Binding Updates in MIPv6

In this protocol, the mobile node first sends two *Init* messages to the correspondent (CN), *Home Test Init* (HoTI) via the home network and *Care-of Test Init* (CoTI) directly (steps 1a and 1b in Figure 19.6). The correspondent replies to both requests independently. A *Home Test* (HoT) message containing a 64-bit home keygen token K_0 and a home nonce index i is sent to the mobile node via the mobile's home address (step 2a). A *Care-of Test* (CoT) message containing a 64-bit care-of keygen token K_1 and a care-of nonce index j is sent directly to the claimed current location (step 2b). The mobile node uses both keygen tokens to compute a binding key

$$K_{bm} := \text{SHA1(home keygen token||care-of keygen token)},$$

and the *Binding Update* (BU) authenticated by a 96-bit message authentication code (step 3)

$$\text{HoA}, i, j, \text{HMAC\_SHA1}(K_{bm}, \text{CoA||CN||BU})\_96.$$

This protocol does not rely on the secrecy of cryptographic keys but on *return routability*. The correspondent checks that it receives a confirmation from the advertised location. In

the threat model assumed, keys K_0 and K_1 may be sent in the clear on the channels from the correspondent. These keys could also be interpreted as challenges (nonces) that bind identity to location through the binding key K_{bm}. In communications security the term 'authentication' typically refers to the corroboration of a link between an identity of some kind and an aspect of the communications model, such as a message or a session [103]. In this interpretation, binding update protocols provide *location authentication*. The protocol is vulnerable to an attacker who can intercept both communications links, in particular the wired Internet. If you are concerned about the security of the wired Internet, you can use IPsec to protect traffic between the correspondent and the home agent.

To be resilient against denial-of-service attacks, the correspondent should be stateless in a protocol run. That is, it should not need to remember the keys K_0 and K_1. For this reason, each correspondent node has a secret node key K_{cn} used for producing the keys sent to the mobile node. This key must not be shared with any other entity. Each correspondent node generates nonces at regular intervals. Nonces are identified by an index (indices i and j in Figure 19.6). The keys are obtained as the first 64 bits of message authentication code values,

$$K_0 := \text{HMAC\_SHA1}(K_{cn}, \text{HoA}||\text{nonce}[i]||0)_{64},$$
$$K_1 := \text{HMAC\_SHA1}(K_{cn}, \text{CoA}||\text{nonce}[j]||1)_{64}.$$

After replying, the correspondent discards K_0 and K_1 because it is able to reconstruct the keys when it receives the final confirmation. The state kept by the correspondent is thus independent of the number of BU requests it receives. Balancing message flows is another provision against denial-of-service attacks. A protocol where more than one message is sent in reply to one message received can be used to amplify denial-of-service attacks. Therefore, the BU request is split in two. The mobile node could send the home address and care-of address in one message but then the correspondent would reply to one BU request with two BU acknowledgements.

Lesson

In communications security it is traditionally assumed that passive eavesdropping attacks are easier to perform than active attacks. In mobile systems, the reverse can be true. To intercept traffic from a specific mobile node, one has to be in its vicinity. Attempts to interfere with location management can be launched from anywhere.

19.4.3 Ownership of Addresses

Schemes that dynamically allocate addresses must check that a new address is still free. This can be done by broadcasting a query asking whether any node on the network is already using this address. In a *squatting attack*, an attacker falsely claims to have

the address that should be allocated, thereby preventing the victim from obtaining an address in the network.

We will now show how a node can prove that it 'owns' an IP address without relying on any third party, be it home agent or certification authority. The core idea is to have the address owner create a public/private key pair and use the hash of the public key as the interface ID in an IPv6 address. To prove ownership of an address, a claim is signed with the private key. The signed claim together with the public key is sent to the correspondent. The correspondent verifies the signature on the claim and checks the link between public key and IP address. The address is the 'certificate' for its public key. Public-key cryptography is used without using a PKI.

Cryptographically generated addresses (CGAs), as specified in RFC 3972 and [17], apply this idea to IPv6 addresses. In this case the length of the hash value is limited to 62 bits, as two bits of the interface ID are reserved. An attacker does not have to find the original key pair to forge claims for a given address, but only a collision, i.e. a key pair where the public key hashes to the interface ID. The feasibility of finding collisions for 62-bit hash values is too close for comfort so a method of extending the hash has to be found. A CGA has a security parameter *Sec* (3-bit unsigned integer) encoded in the three leftmost bits of the interface ID. The security parameter increases the length of the hash in increments of 16 bits. Hash values *Hash1* and *Hash2* for the public key are computed as

$$Hash1 = \text{SHA-1}(modifier||subnet\ prefix||collision\ count||public\ key),$$
$$Hash2 = \text{SHA-1}(modifier||0_{64}||0_{8}||public\ key).$$

The $16 \cdot Sec$ leftmost bits of *Hash2* must be zero. The 64 leftmost bits of *Hash1* become the interface ID, excluding the five fixed bits so that only 59 bits are used. Resistance against collision attacks is proportional to finding a collision for a $(59+16 \cdot Sec)$-bit hash. To get a *Hash2* value of the required format, the address owner does a brute force search varying a random 128-bit *modifier*. The effort for this search amounts to getting a hash with $16 \cdot Sec$ bits equal to a fixed value (all zeros). The collision count is initialized to 0 and incremented if a collision in the address space is reported. An error is reported after three failures.

The workload of the verifier is constant. To verify the link between address and public key two hashes have to be computed. To verify a claim, a signature has to be verified. This may, however, become an issue in denial-of-service attacks as signature verification is an expensive operation. CGAs do not stop an attacker from creating bogus addresses from its own public key and any subnet prefix. Thus, CGAs do not prevent bombing attacks against the subnet. To defend against this attack, a return routability test may check whether the receiver is willing to accept traffic.

19.5 WLAN

Wireless LAN (WLAN) is a technology for wireless local area networks specified in the IEEE 802.11 series of standards. The security challenges in this application are not primarily consequences of mobility but of wireless network access. Again, the integrity and confidentiality of data transmitted over the air should be protected. Suitable cryptographic algorithms have to be chosen and key management procedures have to be defined. In addition, access to the local network and to mobile devices has to be controlled. An unprotected WLAN would give an attacker the opportunity to use bandwidth services that will be charged to the network owner, or to use the victim's system as a staging post for attacks against third parties.

A WLAN can be operated in infrastructure mode or in ad hoc mode. In infrastructure mode, mobile terminals connect to a local network via *access points*. In ad hoc mode, mobile terminals communicate directly. The security discussions in this section will focus on infrastructure mode, and on the access control options at an access point. Each access point has a service set identifier (SSID). An *open* WLAN does not restrict who may connect to an access point. An open WLAN is not necessarily unprotected; security mechanisms could be provided in other protocol layers. Public access points are known as hotspots.

To control access to a WLAN, access points can be configured not to broadcast their SSIDs but to require clients to know the SSID to connect to. The SSID would be the secret necessary to gain access. This approach does not work too well. The SSID is included in many signalling messages where it could be intercepted by an attacker, so it should not be treated as a secret. Also, access points are delivered with a default SSID set by the manufacturer. An attacker could try these default values in the hope, often justified, that they have not been changed. It is further possible to configure access points so that only mobile terminals with known medium access control addresses are accepted. This attempt at address-based authentication is not very secure. An attacker could learn valid medium access control addresses by listening to connections from legitimate devices and then connect with a spoofed medium access control address.

It is in general problematic to base access control on information needed by the network to manage connections, such as SSIDs or medium access control addresses. Typically, this information has to be transmitted when setting up a connection before security mechanisms can be started. Hence, it is a more promising strategy to let the client establish a connection to the access point and then authenticate the client before giving access to protected services.

The universal access mechanism (UAM) shown in Figure 19.7 illustrates this approach. Clients are assumed to have a web browser installed. A client connecting to an access point gets a dynamic IP address from a DHCP server. When the client's web browser

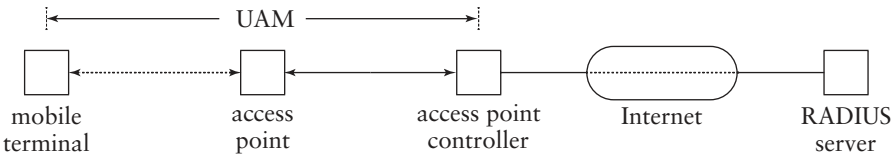


Figure 19.7: Hotspot Access Using UAM and a RADIUS Server

is started, the first DNS or HTTP request is intercepted and the client is directed via an HTTPS session to a start page asking for username and password. The web server at the access point controller refers verification of username and password entered to a RADIUS server. Once the client has been authenticated, the access point can apply familiar identity-based access control to the client's requests. Protection of subsequent traffic between client and access point is a separate issue. An analysis of the security provided by this scheme is left as an exercise.

19.5.1 WEP

The Wireless Equivalent Privacy (WEP) protocol was the first IEEE 802.11 proposal for protecting the confidentiality and integrity of data passed between mobile terminal and access point, and for authenticating mobile terminals to the network. Authentication is based on a shared secret. So-called pre-shared secrets are installed manually in all devices that may get access, and in all access points of the network. This solution is suitable for small installations such as home networks. WEP is seriously flawed and can only serve as a case study for demonstrating mistakes that must not be repeated.

WEP uses a stream cipher for encryption. In a stream cipher, keys must not be repeated; the exclusive-or of two plaintexts encrypted with the same key stream is equal to the exclusive-or of the two plaintexts. The statistical properties of the exclusive-or of two plaintexts can be used for cryptanalysis that reconstructs the two plaintexts and thereby also the key stream. Hence, a 24-bit initialization vector (IV) randomizes encryption. Sender and receiver share a secret 40-bit or 104-bit key K. To transmit a message m, the sender computes a 32-bit checksum CRC-32(m), takes the 24-bit IV, and generates a key stream with the 64-bit (128-bit) key $K' = \text{IV}||K$ using the stream cipher RC4. The ciphertext c is the bitwise exclusive-or of (m||CRC-32(m)) and the key stream,

$$c = (m||\text{CRC-32}(m)) \oplus \text{RC4}(K').$$

Ciphertext and IV are transmitted to the receiver, who computes $c \oplus \text{RC4}(K') = (m||\text{CRC-32}(m))$ and verifies the checksum. To authenticate a client, the access point sends a 1024-bit challenge in the clear to the client. The client uses the above algorithm to encrypt the challenge with the pre-shared key, and the access point verifies this response.

The cryptographic mechanisms used in WEP suffer from two major design flaws. First, CRC-32 is a cyclic redundancy check, i.e. a linear function useful for detecting random

errors but no defence against targeted modifications. When used in conjunction with the linear encryption operation (exclusive-or) of a stream cipher, it does not offer strong integrity protection. An attacker who has a WEP ciphertext, but neither key K' nor plaintext m, can modify the plaintext as follows. Let Δ be the intended alteration of the plaintext. The attacker computes $\delta = \text{CRC-32}(\Delta)$ and adds $(\Delta||\delta)$ to the ciphertext, obtaining a valid encryption of the plaintext $m \oplus \Delta$ as

$$\begin{aligned}(m||\text{CRC-32}(m)) \oplus \text{RC4}(K') \oplus (\Delta||\delta) &= (m \oplus \Delta||\text{CRC-32}(m) \oplus \delta) \oplus \text{RC4}(K') \\ &= (m \oplus \Delta||\text{CRC-32}(m \oplus \Delta)) \oplus \text{RC4}(K').\end{aligned}$$

A second problem is the size of the IV. As long as the secret key K remains unchanged, the IV is the only variable part of the key K'. An attacker could observe traffic for a longer period until IVs repeat and then try to reconstruct the key streams and build a table of IVs and corresponding key streams.

On top of these design flaws, there has been progress in the cryptanalysis of RC4. Attacks that find the key K are described in [94, 162]. These attacks retrieve the key byte by byte so a 104-bit key does not offer much more resistance to cracking than a 40-bit key. Some attacks replay encrypted control messages to create more traffic so that they can obtain the amount of data necessary for cryptanalysis.

19.5.2 WPA

WEP comprehensively failed to meet its security goals. WiFi Protected Access (WPA) was designed as a quick interim solution that removed the major flaws of WEP prior to a complete redesign of the WLAN security architecture. WPA was required to run on existing WLAN hardware. There are also improved procedures for authenticating the client to the network and for establishing temporary encryption keys dynamically. The Extensible Authentication Protocol (EAP; see Section 16.6) has its origins in work on WLAN security.

A message integrity code (MIC) called Michael replaces CRC-32 for better integrity protection. The length of the IV is doubled to 48 bits. The Temporal Key Integrity Protocol (TKIP) creates a key hierarchy where data encryption keys are changed for each packet (Figure 19.8). *Supplicant* (mobile station) and authenticator (e.g. an access point controller) need to have a long-term pairwise master key (PMK).

When WPA is deployed with pre-shared master keys (WPA-PSK), the PMK is computed with the key generation function PBKDF2 (RFC 2898) as

$$\text{PMK} = \text{PBKDF2}(\textit{passphrase}, \text{SSID}, \text{SSID } \textit{length}, 4096, 256)$$

with a secret passphrase (20 characters are recommended), the SSID of the access point, and the SSID length as input. This input is hashed 4096 times and a 256-bit key is returned.

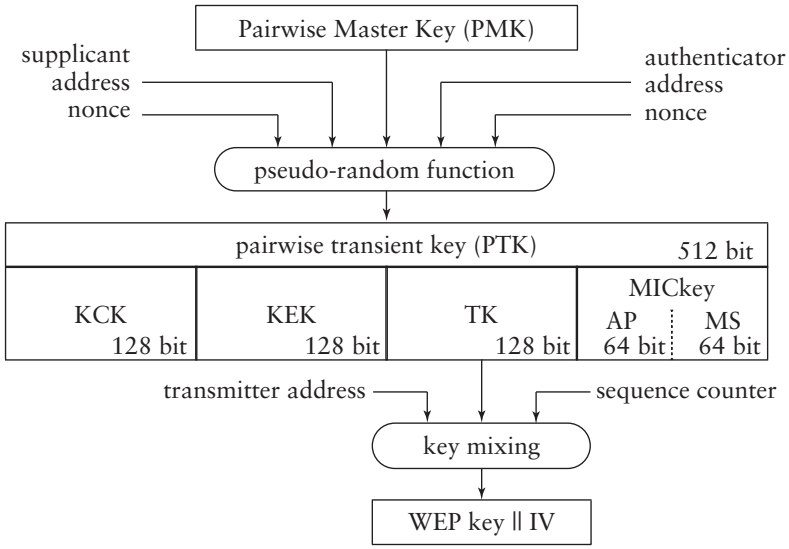


Figure 19.8: The TKIP Key Hierarchy

For each connection new pairwise transient keys (PTKs) are derived from the master key and used for protecting traffic between mobile station and access point. The algorithm computing the PTK takes the PMK, the medium access control address of both devices, and nonces generated by both devices as its inputs. Nonces are transmitted in the clear. The key hierarchy is then further extended. The PTK is split into a key confirmation key (KCK) for key authentication, a key encryption key (KEK) for distributing group keys, a temporal key (TK) for data encryption, and possibly unidirectional MIC keys for integrity protection. The temporal session key is derived from the TK and the medium access control address of the access point, so different access points managed by the same controller will use different keys. Finally the WEP key and IV are derived from the temporal session key and the packet sequence counter. Each packet is thus encrypted under its own key and IV.

Lesson

Patching a large-scale system that uses dedicated hardware is difficult. The new solution has to use whatever is present in the field.[2]

[2]There were several incidents at the start of 2010 where bank card transactions were rejected at ATMs because of incorrect processing of the year field when checking the validity of the card. Fixing such a problem might require a call-back for all cards affected.

19.5.3 IEEE 802.11i – WPA2

A complete redesign of WLAN security mechanisms is specified in the standard IEEE 802.11i, published in 2004. The WPA2 specification produced by the WiFi Alliance and IEEE 802.11i are sometimes used as synonyms. The two specifications overlap to a large extent, but are not completely identical.

IEEE 802.11i has two modes. *Robust Security Network* (RSN) needs new hardware and is not backward compatible with WEP. RSN supports dynamic negotiation of authentication and encryption algorithms. For authentication in large networks EAP is used. Smaller networks can use TKIP. Authentication establishes temporal keys per packet. *Transitional Security Network* (TSN) allows RSN and WEP to coexist on the same WLAN. Devices using WEP may pose a security risk in such a configuration.

RSN uses CCMP (Counter mode CBC-MAC Protocol) for encryption and integrity protection. CCMP is based on 128-bit AES (key and block size) in *Counter with CBC-MAC Mode* (CCM). CCM is defined in RFC 3610. For encryption, AES is used in counter mode. The counter is initialized to 1 when a new temporal key is established. Each 128-bit plaintext block is exclusive-ored with the counter value encrypted under the temporal key; then the counter is incremented. CBC-MAC mode is used for integrity. Header blocks put in front of the message contain the packet number and parts of the MAC (media access) address field. Headers and plaintext block are encrypted in CBC mode with IV 0 under the temporal key; the last ciphertext block serves as the 128-bit CBC-MAC (see Section 14.5.2). A 64-bit MIC is derived from the CBC-MAC and transmitted with the encrypted packet.

19.6 BLUETOOTH

Bluetooth is a technology for wireless ad hoc networks, initially envisaged for short-range communications (up to 10 metres) between personal devices – PC, keyboard, mouse, printer, headset, or other peripherals – forming a *piconet* (also personal area network).

A local ad hoc network of devices in close proximity does not require a sophisticated key management infrastructure. A security association between two devices can be established manually by *pairing*. When both devices have a keypad the user enters a common PIN on both devices. A random nonce is generated on the *master* device and sent via Bluetooth to the *slave*. A 128-bit *link key* is derived from the PIN and the nonce. The link key is used for encryption and later authentication between the devices.

Bluetooth in *secure mode* 4 has the *Simple Secure Pairing* (SSP) protocol for establishing link keys. SSP uses elliptic curve Diffie–Hellman (ECDH). It is at the user's discretion when or whether to change the public/private key pair of a device. Physical proximity is the main protection against man-in-the-middle attacks. There are four phases:

1. Public key exchange. The devices exchange their public keys and derive a shared Diffie–Hellman key.
2. Authentication phase 1. Checks the success of phase 1; nonces are exchanged.
   - Numeric comparison. Both devices have a display and a yes/no button. Master and slave create nonces N_a and N_b, respectively. The slave commits to N_b by computing a MAC over the public keys from phase 1 with N_b as the MAC key and sends the commitment to the master. Then the nonces are exchanged. The master verifies the commitment. Both devices display a numeric verification value derived from the two public keys and two nonces. If both devices show the same value the user may approve the association; if any of the checks fails the protocol is aborted.
   - Out-of-band. Communication via an out-of-band channel is supported. The devices create random values r_a and r_b and commit as above. Each device then sends its random value and commitment and verifies the commitment received. If verifications succeed nonces N_a and N_b are exchanged.
   - Passkey entry. Devices have a keypad. A passkey is entered on each device. For each bit of the passkey the following protocol is run: for the ith bit, the devices create nonces N_{ai} and N_{bi}, respectively. Each device commits to its nonce by computing a MAC over the public keys from phase 1 and the ith bit of the passkey with its nonce as the MAC key. Then commitments are exchanged. Thereafter the nonces are exchanged and the commitments are verified. If all checks succeed and the protocol proceeds, the last nonce pair is used as N_a and N_b.
3. Authentication phase 2. Each device computes under the Diffie–Hellman key from phase 1 a MAC over N_a, N_b, the passkey or the random values, the Bluetooth addresses, and other relevant fields from the device state. The MACs are exchanged and verified.
4. Link key calculation. The link key is derived from the MAC over N_a, N_b, a KeyID, and the Bluetooth addresses, computed under the Diffie–Hellman key.

Authentication in Bluetooth is performed in a challenge-response protocol, similar to GSM authentication. Temporary communication keys for encrypting messages are derived from the shared PIN, the PIN length, a random value generated by the sender, and the receiver's address. Authentication of devices is the basis for access control. The objects of access control are the services (dial-in, file transfer) offered on a device.

The Bluetooth security architecture was designed for piconets. As Bluetooth applications as well as Bluetooth technology are changing, e.g. extending the range of communications, new security architectures will have to be developed. For Bluetooth applications, application-level attacks that exploit weaknesses in the software configuration of the devices have to be considered. Attacks such as Bluesnarf retrieve personal data from devices with flawed implementation of access control. Roaming profiles of users can be established when Bluetooth devices are configured to broadcast their identities on request.

19.7 FURTHER READING

This chapter has described various mobile and wireless technologies with a focus on security issues of general interest. Further interesting security issues of great practical relevance not covered in this chapter relate to more specific aspects of individual technologies. A detailed discussion of GSM and UMTS security is given in [182]. GSM standards are managed by ETSI (`www.etsi.org`), the UMTS specifications by the 3GPP (`www.3gpp.org`). Mobility issues in TCP/IP networks are discussed in various IETF working groups. The current status of these discussions can be found on the respective IETF websites (`www.ietf.org`). WLAN security is specified in IEEE 802.11. The official Bluetooth membership site is (`www.bluetooth.com`).

19.8 EXERCISES

Exercise 19.1 Telephone service providers have to deal with complaints from customers who claim not to have made a call (to a premium rate number) that appears to have been made from their phone. What kind of protection measures could be offered to address such problems?

Exercise 19.2 Investigate the issues that arise when wiretaps in a mobile system with international roaming should be authorized.

Exercise 19.3 Let a 128-bit code be used for message authentication on a channel with an error rate of 1:1000. What is the probability of rejecting a message because of a transmission error when the message length is 1 Kbit, 2 Kbit, and 1 Kbyte? What are the probabilities when the bit error rate rises to 1:100? For both bit error rates, what are the probabilities when a 32-bit code is used?

Exercise 19.4 Nodes that have established a session at the transport layer change their IP address during the session. How could a session be protected at the IP layer in such a situation? Give an analysis of the general security problems that a solution to this problem has to address, and of proposals currently discussed in the IETF.

Exercise 19.5 What effort is required from the address owner to create a valid CGA in comparison to the effort for an attacker for values $Sec = 1, 2, 3$ when the computation of a hash value takes 1 microsecond?

Exercise 19.6 Consider a busy access point that sends 1500-byte packets at the IEEE 802.11b data rate of 11 Mbit/s. How long would an attacker have to

wait for IVs to start repeating? If IVs are generated randomly, how long does an attacker have to wait on average for the first collision (two packets encrypted with the same IV)?

Exercise 19.7 The WEP challenge-response protocol sends challenges in the clear and encrypts the responses. Would it make any difference if the challenges were encrypted and the responses sent in the clear?

Exercise 19.8 Describe an attack whereby WEP encrypted IP packets are rerouted to a destination chosen by the attacker.

Exercise 19.9 Give a comparison of the security services provided by UAM and 802.1X with EAP (Section 16.6).

Exercise 19.10 WPA-PSK uses a passphrase for authentication. Examine whether this protocol is vulnerable to password guessing attacks.

Exercise 19.11 Bluetooth users confirm a pairing of devices by typing in a common password on both. Explain how a social engineering attack can lure the user into accepting a pairing unintentionally.

Chapter 20

New Access Control Paradigms

The Internet and the World Wide Web have brought a large but overwhelmingly security-unaware user population into direct contact with new IT applications. Mobile code moves through the Internet and runs on the clients' machines. Electronic commerce promises new business opportunities on a global scale. We are faced with considerable change in the way IT systems are used. We thus have to question whether the old security paradigms still fit and where new policies and new enforcement mechanisms are needed.

This transition phase has not yet concluded. New concepts have emerged but more experience has to be gained before we can give clear advice on the best security strategies to follow in this new environment.

OBJECTIVES

- Explore new paradigms for access control.
- Explain the background and rationale for the move to code-based access control.
- Discuss the stack walk as the main security enforcement algorithm used in code-based access control.
- Outline options for digital rights management.

20.1 INTRODUCTION

The client–server architecture was the major paradigm in distributed computing before the World Wide Web appeared on the scene. A client wants to perform computations on a server. The server authenticates the client to protect itself. Kerberos is a prime example of an authentication service designed for such an environment (Section 15.4). This architecture has changed in several core aspects. Computation moves from server to client. When a client looks up a web page, the client's browser runs programs (scripts) embedded in the page. The nature of access operations has changed. Instead of simple read/write requests to an operating system or a database, programs are sent to be executed at the other side. Servers execute scripts coming from the clients. Security aspects of scripting have been discussed in Section 10.5.1. Clients receive scripts from servers and may store session states in cookies (Section 18.2). So, what has changed with the web?

- The separation of program and data becomes blurred. *Executable content* (applets, scripts) is embedded in interactive web pages that can process user input.
- Computation moves to the client. Documents contain executable code and clients run on quite powerful machines, so servers can free resources by passing computational tasks to their clients. The client needs protection from rogue content providers.
- As computation moves to the client, the client is asked to make decisions on security policy, and on enforcing security. Users are forced to become system administrators and policy-makers.
- The browser becomes part of the trusted computing base.

Whilst the World Wide Web has not created fundamentally new security problems, it has changed the context in which security has to be enforced to such an extent that a fundamental rethink of access control paradigms is necessary. This chapter will explore the changes that are taking place in access control.

20.1.1 Paradigm Shifts in Access Control

In the access control model of Section 5.2, principals make access requests that have to be authenticated. Security policies authorize principals to access an object. There is a strong underlying assumption that principals by and large correspond to known people. Access control in Unix and Windows builds on this model and implements *identity-based access control* (IBAC). This model originated in 'closed' organizations such as universities or research laboratories. This approach to access control is underpinned by some important organizational assumptions. The organization has authority over its members. The members (users) can be physically located. Audit logs point to users who can be held accountable for their actions. In such a setting, security policies refer naturally to user identities and access control seems to require by definition that identities of persons are verified. Biometrics (Section 4.7) may then appear as a logical step towards stronger identity-based access control.

Access control systems were characterized by the following features. Access rules are local. There is no need to search for the rule that applies for the current request or for the credentials a subject has to supply. The rules are stored as ACLs with the objects. The credentials are the UIDs (SIDs in Windows) subjects are running under. Enforcement of rules is centralized. The reference monitor does not consult other parties when making a decision. Permissions apply to simple and generic operations such as read, write, or execute. There is a degree of *homogeneity* when setting security policies. The same organization is in charge of defining organizational and automated security policies. Responsibility for system administration is clearly delineated. System managers are in charge. Users do not participate in systems administration.

20.1.2 Revised Terminology for Access Control

For historic reasons, the established terminology for access control is geared towards explaining identity-based access control. In code-based access control, a major topic of this chapter, we do not necessarily ask 'who made this statement?' so we either have access control without authentication, or we have to reinterpret authentication and widen its meaning so that it stands for the verification of any externally provided evidence associated with a request (Figure 20.1). In addition, we might associate local evidence with a request. For example, the request could be part of a current session, or the decision whether to grant access could depend on the current date and time. The term *authentication* may then also include the verification of such an association.

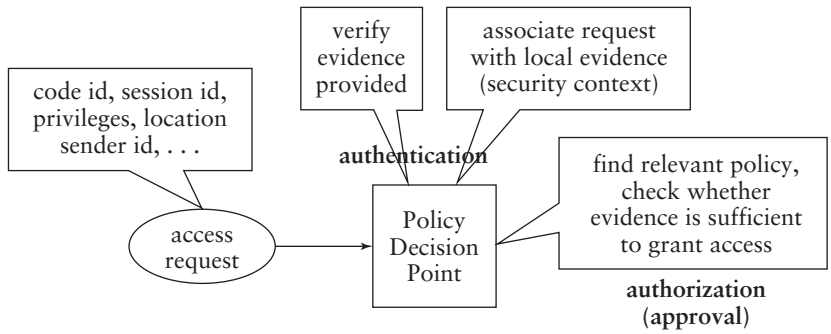


Figure 20.1: Access Control in an Open Environment

Secondly, we have to find the policy that applies to the request and then check whether the available authenticated evidence is sufficient to grant the request. In a system that uses permissions, we would check whether all *required permissions* are included in the *granted permissions*. This phase could be called *authorization*, but it no longer tells us who is entitled to access a resource but what evidence has to be provided to gain access. If we want to reserve *authorization* for the setting of policies, we might use *approval* for the phase that checks whether a request is permitted according to the given policy.

In Figure 5.1 the reference monitor was in charge of authentication and authorization. In a heterogeneous environment authentication and authorization may be taken care of by different entities (Figure 18.12).

- A policy administration point (PAP) creates the policy.
- A policy decision point (PDP) evaluates the applicable policy and makes an authorization decision (Figure 20.1).
- A policy enforcement point (PEP) performs access control. It sends decision requests to a PDP and enforces the authorization decisions received.
- A policy information point (PIP) is a source of evidence.

20.2 SPKI

In the old access control paradigm, access rules are stored locally in protected memory. In decentralized access control systems, you can protect the integrity of access rules by cryptographic means. Specifically, you can encode rules in digitally signed certificates. Identity-based access control can be implemented using X.509 identity certificates and attribute certificates as discussed in Section 15.5.3.

SPKI, the Simple Public Key Infrastructure (RFC 2692, 2693), is a PKI intended to support authorization schemes that work without user identities. The designers of SPKI postulate that names used in access control have essentially only a local meaning, and that access rights are likely to derive from attributes other than a person's name. For the purpose of access control, names are just pointers to access rights (authorizations). A security policy sets the rules in a given domain, e.g. a university department. The names used to state the policy must have a meaning locally within the domain, but need not be globally unique.

When there is interaction between domains, you may need to refer to names from other local name spaces. You need globally unique identifiers for name spaces to avoid confusion about names. Public-key cryptography offers an elegant solution. Public/private key pairs generated at random are unique with extremely high probability, as would be the hash of the public key. The public key of an issuer (or its hash) can be the unique identifier for the name space defined by that issuer. In SPKI terminology, such keys are called *principals*.[1] Name certificates signed with the private key define a name in the local name space.

Access rights are bound directly to public keys through *authorization certificates*. Authorization certificates contain at least an *issuer* and a *subject*, and may also include a delegation bit, authorizations, and validity conditions. The issuer signs the certificate

[1] The terms 'principal' and 'subject' are used here in a different sense than in Chapter 5.

and authorizes the subject, i.e. assigns access rights to the subject. The issuer sets the policy and is the source of authority ('empowerment' in SPKI terminology). The subject is typically a public key, the hash of a key, or the name for a key. The root key for verifying certificate chains is stored in an ACL. Identity certificates that bind a person's name to a key are used for accountability, and could be issued by a CA other than the one issuing authorization certificates (Figure 20.2).

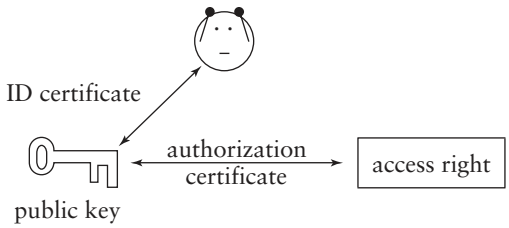


Figure 20.2: Certificates in SPKI

Access rights are assigned to public keys. The holders of the corresponding private keys may delegate (grant) some of their access rights to other subjects by issuing certificates for those rights. Delegation is restricted in two ways. Delegators can only delegate rights they hold themselves. This design decision is justified with the observation that a key-holder who may delegate a right without being able to exercise it could just generate a new key pair and delegate the right to that key. Secondly, a flag indicates whether the access rights in a certificate may be delegated further.

Authorization certificates and ACLs have the same function and differ only in the mode of protection. Certificates are signed by an issuer. ACLs are unsigned and have no issuer. They are stored locally and are never transmitted. Tuples are an abstract notation for SPKI certificates and ACL entries. The algorithm for processing authorizations is expressed in terms of 5-tuples *<issuer, subject, delegation, authorization, validity dates>*. The issuer is a public key or the reserved word 'Self' for ACLs. The subject is defined as above. The delegation bit is a Boolean value. The authorization field gives application-specific access rights. Validity is defined by a not-before date and a not-after date. The algorithm for evaluating certificate chains takes two 5-tuples as input and gives the result as a 5-tuple. Authorizations and validity periods can only be reduced by the respective intersection algorithms:

```
Input: <Issuer1,Subject1,D1,Auth1,Val1>
       <Issuer2,Subject2,D2,Auth2,Val2>

IF Subject1 = Issuer2 AND D1 = TRUE
THEN output <Issuer1, Subject2, D2,
      AIntersect(Auth1,Auth2), VIntersect(Val1,Val2)>
```

SPKI is a key-centric PKI and an interesting alternative to X.509, at least from the standpoint of certificate theory. SPKI standardizes certain automated policy decisions, e.g. on delegation. Hence, one has to check that these rules match the given organizational policy. Organizational policies are unlikely to be expressed by direct reference to cryptographic keys.

20.3 TRUST MANAGEMENT

A subject (as defined in Chapter 5) requesting access to a resource presents a set of *credentials*. Credentials are security evidence. The security policy states what credentials are required for the request to be granted. Three inputs enter a security decision: the set of credentials granted to the subject; the set of credentials required to gain access; and an algorithm that evaluates whether the required credentials are implied by the granted credentials.

In IBAC, typical credentials are usernames and passwords. Granted and required credentials are found in well-defined places such as the subject's access token and an ACL for the object. The algorithm checking whether the granted credentials are sufficient is relatively simple, going through an ACL, granting access once all required credentials have been found, and denying access otherwise. The reference monitor does not ask third parties for help when making its decisions.

In heterogeneous environments, it may become more complicated to find granted and required credentials, and third parties may get involved in access decisions. Consider the following hypothetical example. Two telecommunications providers *X* and *Y* have a service level agreement that gives customers from *X* access to the services offered by *Y*. Provider *Y* will not get a list of all subscribers from *X*, but *X* issues its subscribers with certificates and gives *Y* the required verification key. Subscribers from *X* request services from *Y* by presenting their certificates. Provider *Y* calls back *X* to perform an on-line status check on the certificates, 'deferring' this check to *X*. The reply from *X* is input to *Y*'s decision.

Trust management is the name adopted in [40] for a more general and flexible approach to access control. For this section, we have to redefine parts of our terminology. Principals can be public keys and we can directly authorize actions the key-holder can perform. There is no need to authenticate a user to perform access control. The *actions* a policy regulates are defined in an application-specific action description language. An *assertion* binds a public key to a *predicate* on actions. An assertion is a policy statement that authorizes actions if a digitally signed request to perform those actions can be verified with the public key given in the assertion and if the requested actions satisfy the predicate. *Credentials* are digitally signed assertions. They generalize authorization certificates. *Policies* are locally stored assertions. They are the roots of trust (authority).

A *trust relationship* is a policy rule to accept credentials issued by another party. The *policy language* predicates are written in can be any safe interpreted language. We can thus specify more sophisticated evaluation algorithms than with ACLs. Trust management provides a common language for policies, credentials, and trust relationships. The first trust management system was PolicyMaker [40]. KeyNote is a trust management system for the Internet (RFC 2704).

It is important to understand what trust means in such a trust management system. When we defer trust to some other party, we introduce a policy rule that assigns to this party the authority to make assertions we will consider when evaluating an action request. In real life, trust relationships may be based on contractual relationships between parties. They may also be based on 'trust' in the colloquial sense of the word, but it is much safer to separate the rationale for entering into a trust relationship (which may have nothing to do with trust) from the operational aspects of a trust management system.

Access requests are evaluated by a Policy Decision Point (also known as a *compliance checker* or *trust management engine*) that receives as its inputs a request r and a set of credentials C, and refers to the local policy P when answering the question: 'Does the set C of credentials prove that the request r complies with the local security policy P?' [40]. Possible answers to this question are 'yes', 'no', 'don't know', or an indication that further checks are required. There is a trade-off between the expressiveness of the policy language and the complexity of the compliance checker. Depending on the language, compliance checking may turn out to be undecidable. You also have to consider how users find out about the credentials they need to present so that their request complies with the given policy. Should the server provide this information? Should the server publish its policy at all? The policy itself may be sensitive information. Alternatively, there could be an algorithm for users to compute the set of credentials required for their request, or users could refer to a *credential chain discovery service*.

To add a further degree of freedom to your problem, consider a *federated environment* where several organizations collaborate, but each has its own security policy. There is now no longer a single entity setting the policy. Conflicts between policies are bound to occur. In such a setting you have to consider how conflicts could be resolved, and on the course of action that should be taken if this is not possible.

20.4 CODE-BASED ACCESS CONTROL

The Internet connects you to parties you never met before. In applications such as e-commerce it is often commercially necessary to permit unknown entities access to your systems. Naturally, this should not compromise your own security. You have moved out of closed organizations into an 'open' environment.

- You are dealing with people who are a priori unknown. Their 'identity' can hardly figure in your access rules.

- You may not be able to physically locate other parties.
- You may have no real authority over other parties. Even if you could identify the persons responsible for some action, you will not always be able to hold them accountable. They might live in different jurisdictions so that recourse to the legal process would be too slow, cumbersome and expensive to be considered as part of a security strategy. The relationship to customers is in many ways different from the relationship to employees.

In these circumstances user identities are not helpful when setting security policies. In consequence, the source of a request is no longer a useful access control parameter. At the same time, reference monitors do not receive read/write/execute requests but entire programs (scripts). Access control has to decide whether to execute a script, and what rights to give to the script if it is run. The Java sandbox showed the way to control access based on code instead of user identities. The evidence used in making access decisions can be:

- Code origin. Is the code stored locally or does it come from a remote site? If it is remote code, which URL does it come from? You can use the URL to determine a *zone* and define security policies on the basis of zones. As a simple example, you could have separate policies for code coming from our intranet and for code from the Internet.
- Code signature. Has the code been digitally signed by a trusted author? In this context, 'trust' simply means that you have made a decision to run code signed by a given author (code distributor) and assign certain privileges to it. This is usually an operational decision and not a judgement about the trustworthiness of the author.
- Code identity. You could decide to give certain approved (trusted) scripts specific privileges. A home banking application would be a typical example. You could compute the hash of the script received and check whether it is on a list of trusted scripts. Again, this is likely to be an operational decision and not a technical judgement about the trustworthiness of the script.
- Code proof. The decision whether to run a script could be based on specific properties of the script, such as the files it can write to or the sites it can connect to. Creating a proof that such a security property holds is a non-trivial task. Thus, the author could provide the proof and the reference monitor would only check the proof when making an access decision. *Proof-carrying code* [178] is still explored mainly in research but has found some real applications [89]. Deciding on useful security properties may actually be more difficult than proving that they hold.

We refer to this type of access control as code-based access control. *Evidence-based access control* is an alternative term denoting the same concept. To keep security policies manageable, permissions are usually assigned to code sets (assemblies in .NET terminology). Individual software components derive their permissions from the container they have been placed in.

20.4.1 Stack Inspection

Subjects act on behalf of principals. In identity-based access control, the principal is a user. When the user logs in, a process is created that runs under the UID of that user. The access rights (permissions) given to that process derive from this UID. Normally, the permissions do not change while the process is running. To effect a change, a new process is spawned that runs under a UID different from that of the parent process. SUID programs in Unix are a typical example.

In code-based access control, when one software component calls another component, the *effective* permissions of the process executing these components will change. The algorithm that determines the effective permissions has to take into account the permissions of the calling component and of the component being called. We have to decide how to assign permissions when a 'trusted' component calls an 'untrusted' component, and vice versa. Consider the following example. Function `g` has permission to access resource *R* but function `f` does not have this permission. Function `g` calls `f` and `f` requests access to *R* (Figure 20.3, left). Should access be granted? The conservative answer is 'no' because `f` does not have the required permission. However, `g` could explicitly *delegate* (grant) the permission to `f`. Conversely, let `f` call `g` and `g` requests access to *R* (Figure 20.3, right). Should access be granted? The conservative answer is again 'no' because `g` could be exploited in a *luring attack* (also known as a *confused deputy attack*; see [113] and Section 6.3.6) by `f`, but `g` could explicitly *assert* its permission. In both cases we need to know the entire call chain when making access decisions.

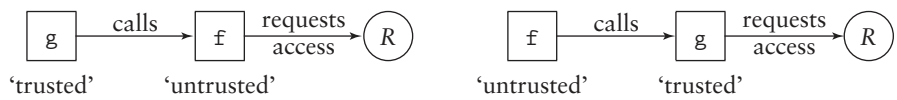


Figure 20.3: Call Chains in Code-Based Access Control

It is customary to refer to components that have been assigned powerful permissions as 'trusted'. Usually, this does not imply that there exist particularly good reasons to trust that such a component is free of security flaws, but that it needs those permissions to do its job. Similarly, an 'untrusted' component is not so much a component you cannot trust but a component that should run with fewer permissions.

The Java Virtual Machine and the .NET Common Language Runtime both use a call stack to manage executions. At every function call a new frame is created on the stack. Each frame contains the local state of the function, including the permissions directly granted to it. For the computation of the effective permissions, *lazy evaluation* is so far the preferred design choice. If a function requests access to a protected resource, a *stack walk* is performed to establish whether the caller has the required permissions. The

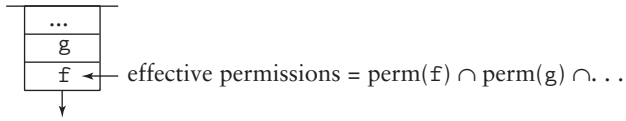


Figure 20.4: Computing Effective Permissions in a Stack Walk

effective permissions of the final caller are computed as the intersection of the permissions of all functions on the call stack (Figure 20.4).

The options for controlled invocation would be severely curtailed if this strategy were strictly adhered to. Therefore, it is usual to provide the option for code to *assert* a permission. Asserted permissions are attached to the current stack frame and removed when returning from that component. The stack walk for a permission terminates and grants the permission when it reaches a frame that asserts the permission. (All frames examined so far also have the permission.) The stack walk for other required permissions continues. Assertion allows 'untrusted' callers to call a 'trusted' method.

Programmers writing components and assigning permissions to components have to consider that untrusted components may take advantage of the functions they implement and build in the appropriate defences. On the other hand, they also have to consider that permissions may be missing when running their code because the caller has insufficient permissions.

20.4.2 History-Based Access Control

Stack inspection reuses the call stack for a new purpose. This can get in the way of its original use. In terms of performance, common optimizations might have to be disabled for security reasons. We use *tail call elimination* to illustrate this point. A function call that is the last instruction in a component is called a tail call. Consider this code example:

```
void g()                    void f()
{                           {
  ...;                        ...;
  f(); // tail call         }
  return;
}
```

Once `f()` is called, `g`'s frame is no longer needed and could be overwritten with `f`'s new frame. Eventually, `f` returns directly to `g`'s caller. The gains of this optimization are particularly significant for recursive functions. With tail call elimination, however, an 'untrusted' caller may leave no tracks on the stack. In our example, let `f` request an access that requires a permission p that has not been granted to `g`. Without tail call

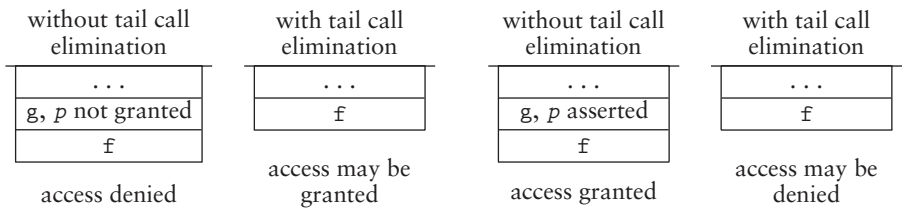


Figure 20.5: Effects of Tail Call Elimination

elimination, the stack walk would raise an exception when examining `g`'s stack frame. With tail call elimination, the stack walk proceeds and a situation may arise where access is granted that should be denied (Figure 20.5, left).

On the other hand, a trusted caller may have an asserted permission cancelled. In our example, let `f` have the permission p and `g` assert the permission p, but assume that some previous caller does not have this permission. Without tail call elimination, the stack walk for p would terminate at `g`'s stack frame and the permission would be granted. With tail call elimination, the stack walk is unaware of the *assert* and may eventually deny access. Here, access is denied although it should have been granted (Figure 20.5, right).

To avoid the shortcomings of stack walking as a security mechanism, do not be lazy but perform an *eager evaluation* that keeps track of the callers' rights proactively. In history-based access control [96], static permissions S are associated with each piece of code at load time. The current rights D associated with each execution unit are updated automatically at execute time with the rule $D := D \cap S$.

It is in general difficult to establish the precise guarantees provided by the stack walk. Security-relevant parameters may not be stored on the stack (see Section 10.4.3 on heap overruns). It is thus hard to relate the security policy enforced by this mechanism to high-level security policies.

20.5 JAVA SECURITY

To surf freely on the web, users have to be prepared to accept executable content from any website that catches their attention. To be prepared, they have to be able to control the actions of applets. Applets are Java programs interpreted by a virtual machine in a web browser. It is thus natural to let this virtual machine perform access control, constraining applets within a *sandbox*. In addition, the language the applets are written in can make it more difficult to create damage. To make proper use of the access control mechanisms in the execution environment, security policies have to be set correctly.

20.5.1 The Execution Model

Java is a type-safe object-oriented language. Java source code is compiled to machine-independent byte code and stored as class files. A platform specific virtual machine interprets the byte code translating it into machine specific instructions. When running a program, a class loader loads any additional classes required. Figure 20.6 shows the typical scenario for using Java applets. The applet is written in Java and compiled to byte code. The byte code is put on a web page, and gets executed by the browser on the client machine. A Java enabled browser has its own Java Virtual Machine (JVM). A JVM has three security components: byte code verifier, class loader, and security manager. Browsers tend to enforce similar policies, but this is by choice, not by necessity. Typical security policies for applets are as follows:

- Applets do not get access to the user's filesystem.
- Applets cannot obtain information about the user's name, email address, machine configuration, etc.
- Applets may make outward connections only back to the server they came from.
- Applets can only pop up windows that are marked 'untrusted'.
- Applets cannot reconfigure the JVM, e.g. by substituting their own classes for systems classes.

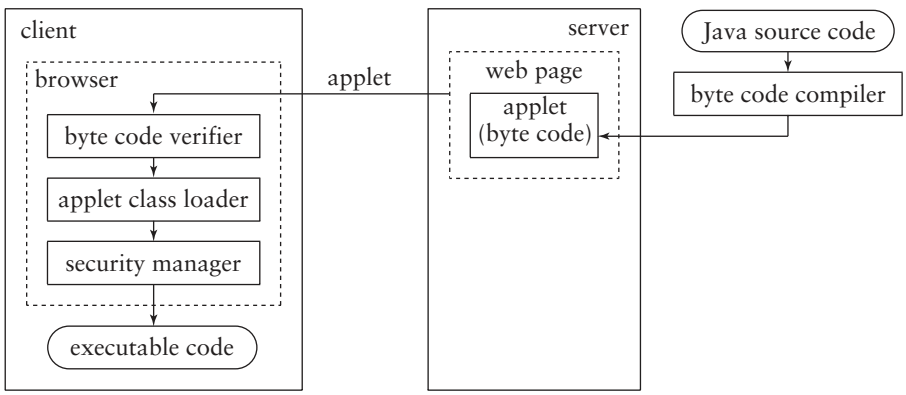


Figure 20.6: The Java Execution Model

20.5.2 The Java 1 Security Model

The initial Java security model implemented a very simple policy (Figure 20.7). Unsigned applets are restricted to a sandbox. Local code is unrestricted. Since Java version 1.1, signed code has also been unrestricted. This basic policy does not provide intermediate

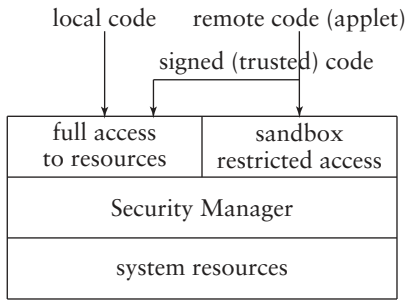


Figure 20.7: The Java 1 Security Model

levels of control. It lacks the flexibility to give some additional privileges to 'semi-trusted' applets, e.g. to an applet that is part of a home banking application.

Moreover, the location of code, local or remote, is not a precise security indicator. Parts of the local filesystem could reside on other machines and would then be constrained in the sandbox. Conversely, downloaded software becomes 'trusted' once it is cached or installed on the local system. For more flexible security policies you would have to implement a customized security manager, a non-trivial task that requires both security and programming skills.

20.5.3 The Java 2 Security Model

The Java 2 security model separates policy specification from policy enforcement. It provides a flexible framework for code-based access control. At the same time it simplifies some aspects of policy enforcement. A *security policy* grants resource access permissions to code by mapping code attributes to granted permissions. The *security manager* is the policy enforcement point called when access to a protected resource is requested. It collects the relevant evidence and invokes the *access controller*, the policy decision point making the actual decision.

We only give a broad overview of the Java 2 security model, concentrating on features of general interest. The four areas covered are language security (byte code verification), securing extensible systems (class loading), policy specification, and policy enforcement. For detailed guidance on using this model the reader is referred to [105].

20.5.4 Byte Code Verifier

The first line of defence is the programming language applets are written in. The language can limit the options for writing malicious code. Java applets are written in byte code and are expected to satisfy certain safety properties. The *byte code verifier* analyzes Java class files, performing syntactic checks and using theorem provers and data flow analysis

for static type checking to ensure that these expectations are met before the applet is executed. Verification intends to guarantee properties such as the following:

- The class file is in the proper format.
- Stacks will not overflow.
- Methods are called with arguments of the appropriate type and return results of the appropriate type.
- There is no illegal data conversion between types.
- All references to other classes are legal.
- There is no violation of access restrictions; e.g. private fields are not accessible from outside the object.

The byte code verifier reduces the workload for the interpreter, as the properties of code guaranteed by the verification do not have to be checked again at runtime. Byte code verification cannot take care of all security issues, so security also depends on the security mechanisms in the runtime environment.

20.5.5 Class Loaders

The Java platform is extensible. Classes are loaded on demand as they are needed to resolve links (lazy loading). Class loading is delayed as much as possible to reduce memory usage and improve system response time. There are three types of class loaders. The *bootstrap class loader* loads system classes that form the core of the virtual machine. Classes loaded in this way have unrestricted access to system resources. The bootstrap class loader is platform-dependent, usually written in a native language such as C, and often loads from the local filesystem. The *extension class loader* loads classes from the installed optional packages (formerly known as extensions). The *application class loader* (once known as the system class loader) loads user-defined classes.

When a class loader is being instantiated it reads in the byte code, defines the class, and assigns it a protection domain (Section 20.5.6). Link-time checks are performed by the JVM to maintain type safety [154]. Compared to runtime checks, link-time checks have the advantage of being performed only once. Every JVM has a class file verifier. The class file verifier makes sure that 'untrusted' classes are safe. Class file verification checks if the constraints of the Java language are respected and that class files have a proper internal structure. Class file verification invokes the byte code verifier.

Each class is uniquely defined by its class type and by its defining class loader. Class loaders provide separate name spaces for different classes. A browser loads applets from different sources into separate class loaders. These applets may contain classes of the same name, but the classes are treated as distinct types by the JVM. *User-definable class loading policies* are supported. A user-defined class loader can, for example, specify how

classes are discovered, or assign appropriate security attributes to classes loaded from a given source.

20.5.6 Policies

Security policies map code attributes to *permissions*. A permission refers to a resource and the operations permitted on the resource. The format of a permission is (*name, actions*), e.g. `(/tmp/abc, read)` stands for a read right on file `/tmp/abc`. Permissions are granted to code (granted permissions), and are required to access a resource (required permissions). There are only positive permissions. Developers are not restricted to predefined permissions but can define application-specific permissions.

There is a permission class hierarchy. Relationships between permissions are defined by `implies` and `equals` methods. `AllPermission` is a wildcard for all permissions, designed to ease policy evaluation for entities such as system administrators who need many permissions to do their job. `UnresolvedPermission` is a placeholder for permissions that have yet to be resolved, e.g. if a permission class has not yet been loaded when a `Policy` object is being constructed. To support policy management, permissions can be collected in *permission sets*.

The security-relevant attributes associated with code are *code source*, consisting of a URL (code origin) and possibly digital certificates (code signers), and *principals* representing users or services. *Protection domains* are a layer of indirection between classes and security attributes. Each class is associated at load time with a protection domain. A protection domain contains code source, principal, a reference to the defining class loader, static permissions granted to code, and permissions assigned at load time. Protection domains are *immutable* objects, i.e. they cannot be modified once created.

Policies are stored in policy files. A policy file can contain at most one *keystore* entry and an arbitrary number of *grant* entries. The keystore entry tells where to find the public keys of the signers given in the grant entries. A grant entry assigns permissions to a code base, a list of signers, and a list of principals. All three fields are optional. It is possible to specify a signer for individual permissions. In this case, the permission is only valid if it appears in a policy file signed by the nominated signer.

20.5.7 Security Manager

The *security manager* is invoked at runtime to check whether a process requesting access to a protected resource has the required permissions. To request this check, the program calls the `checkPermission` method of the security manager. This method can be called with the required permission as the only argument or with the required permission and an *execution context*. The execution context captures the content of the execution stack. The `checkPermission` method of the security manager in turn calls the `checkPermission` method of the *access controller*.

The access controller implements a uniform access decision algorithm for all permissions. To compute the granted permissions, the access controller examines the given execution context, i.e. the execution stack, and performs a stack walk. Each method on the stack has a class and each class belongs to a protection domain. The protection domain defines the permissions granted to this method. The basic access control algorithm computes the intersection of the granted permissions for all methods in the execution context. It grants access if all methods on the stack have been granted the required permission. This algorithm is extended so that methods can assert permissions by calling the `doPrivileged` method that stops the stack walk at this method ignoring callers still on the stack. Further modifications deal with inherited methods and inherited execution contexts.

20.5.8 Summary

The approach chosen in the Java 2 security model is called *declarative security*. The writers of application programs declare what permissions are required to access protected resources by including the relevant checks in the code. They have full flexibility in defining application-specific permissions. The administrators deploying the application assign permissions by specifying security policies. There is a single shared access control algorithm provided by the access controller, ensuring consistency in the enforcement of security policies. Application writers need not implement their own application-specific access control logic.

The JVM in a web browser enforces security in a layer above the operating system. Once a user has access to the layer below the security mechanisms, e.g. by running applications other than the browser, all bets on the integrity of the protection system are off. Furthermore, the Java platform is not secure simply because Java is designed to be a type-safe language. Over time, security problems have been found and had to be fixed. Section 10.4.5 mentioned some incidents related to type safety. Not all software security issues are addressed by the Java type system. For example, it is up to developers to deal with race conditions. A type system for Java that handles race conditions is described in [43].

20.6 .NET SECURITY FRAMEWORK

This section gives a general overview of .NET security features and introduces the terminology used in .NET. A detailed introduction to .NET security is provided in [141].

20.6.1 Common Language Runtime

The Common Language Runtime (CLR) supports a number of programming languages such as C#, Visual Basic, managed C++, Visual J#, and more. The Common Language Specification (CLS) specifies general requirements on all .NET languages. Code written

in any of the .NET languages is compiled into Microsoft Intermediate Language code (MSIL). MSIL is conceptually similar to Java byte code. MSIL code is sent to a just-in-time (JIT) compiler just before it is executed. The CLR loads and executes code, and performs security checks and automatic garbage collection. Architecturally, the CLR corresponds to the JVM.

Managed code is code compiled to run in the .NET framework and is controlled by the CLR. *Native code*, also called unmanaged code, is code compiled to machine language for a specific hardware platform. Native code is not controlled by the CLR and works at a layer below the CLR. Calls from managed code to native code are security-critical, as any guarantees provided by the CLR no longer apply. Particular care has to be taken to check that such a call does not lead to security violations.

An *assembly* is a logical unit of MSIL code. The *metadata* for an assembly provide information including the full assembly name, referenced assemblies, visibility, inherited class, implemented interfaces of each defined class, and information about class members such as methods, fields, and properties.

20.6.2 Code-Identity-Based Security

The .NET platform implements code-based access control, called code-identity-based security in [141]. The basic idea is again to grant different levels of 'trust', i.e. different permissions, to code. Applying the principle of least privilege is a good reason to do so. Calling code with all permissions 'fully trusted', code with very limited permissions 'untrusted', and code with permissions somewhere in between 'semi-trusted' mixes operational aspects of access control with the reasons why permissions have been assigned in a certain way.

Code-identity-based security refers to code identities. Security policies refer to *evidence* about assemblies, *authorizing* code rather than users to access resources. Evidence is dynamically calculated when code is running. Some pieces of evidence, such as the URL of origin of an assembly, are usually not known in advance. *Authentication* of code identity is the process of collecting and verifying evidence about an assembly.

20.6.3 Evidence

The evidence about assemblies may include objects from default classes such as the site of origin where the assembly has just been loaded from, the URL of origin, the hash of the assembly, an authenticode signature, a strong name signature, and a security zone. It is also possible to use other objects as evidence to define application-specific access control. Custom code usually has to be added to process such evidence. Permission Request Evidence states the permissions an assembly must have to run, the permissions it may be granted, and the permissions it must never get.

Evidence can be given by the entity that launches the assembly. This entity is called the *host* and can either be an unmanaged entity that initiates the CLR, e.g. Internet Explorer, or managed code launching other managed code. This evidence is known as *host-provided* evidence and uses the default classes mentioned above. *Assembly-provided* evidence is provided by an assembly itself using application-specific classes. It cannot override host-provided evidence.

Evidence is associated with assemblies and with so-called *app domains* (short for application domains). App domains contain loaded assemblies and provide in-process isolation of executing code. App domains provide a second container layer that can be used to adjust the permissions granted to an assembly based on the application it is being used for. All .NET code has to run in an app domain.

20.6.4 Strong Names

A strong name is a 'bare' public/private key pair. There is no certificate for the public key. Strong names create protected name spaces. The public key is the identifier of the name space. It is listed in the metadata of the assemblies that belong to the name space. An assembly is digitally signed during compilation and the signature is written into the assembly. This signature can later be verified using the public key in the assembly, thereby protecting the assemblies in the name space from modification and spoofing.

We have refrained from denoting the public key as the 'publisher' of the name space. This may create confusion with authenticode signatures. The public keys used for verifying authenticode signatures come with certificates, binding the public key to the entity publishing the software. Security policies may refer to name of this entity. Authenticode signatures match the traditional view of a software publisher as a person or company.

Lesson
Globally unique names can be generated by a central authority in a top-down manner. Alternatively, globally unique names can be generated locally with the help of public-key cryptography.

In the first approach, an authority is in charge of the name space and assigns names to applicants. The authority has to guarantee that names are only assigned once. In a hierarchical name space, subauthorities can be put in charge of their section of the name space. The X.500 directory exemplifies this approach (Section 15.5.3). In the second approach, local name spaces are defined by a public/private key pair. Uniqueness in a name space has to be guaranteed by the respective key-holder. Uniqueness of the public keys relies on randomness in key generation. Collisions are theoretically possible.

20.6.5 Permissions

Code access permissions represent rights to access resources or to perform protected operations, such as accessing unmanaged code. Code access permissions provided by the .NET framework are, for example, `FileIOPermission` giving the right to read, append, or write files or directories, `EventLogPermission` giving the right to read or write access to event log services, `PrintingPermission` giving the right to access printers, and `SecurityPermission` giving among others the right to assert permissions, to call into unmanaged code, and to skip verification. There are numerous built-in permissions of this kind.

Identity permissions represent specific attributes (evidence) of an assembly, e.g. its (authenticode) publisher, the strong name of the name space it belongs to, its hash value, or the URL or zone from which it has been loaded. The `PrincipalPermission` is different from code access permissions and represents a user identity. It is used when defining traditional identity-based security policies. For easier policy management, permissions can be collected in *permission sets*.

Permissions are granted to assemblies by the security policy (see below). Required permissions are specified by placing *security demands* in the code. During execution, a security demand will trigger a stack walk to check whether the permission has been granted. *Declarative* demands are stored in the assembly's metadata. They can be reviewed by looking at the metadata. Checks will occur at the beginning of a method. JIT security actions can only be expressed in declarative form. *Imperative* demands are placed in the code, facilitating more complex security logic that can also handle dynamic parameters in access requests (not yet known at the time the declarative demands are checked).

20.6.6 Security Policies

Security policies translate evidence into permissions. This process uses *code groups* and *permission sets* as its main levels of indirection. A code group has a *membership condition* that determines which code groups an assembly belongs to. For example, one might compare the zone evidence in the assembly with a zone given in a membership condition, or compare the strong name of the assembly with a strong name given in a membership condition. Membership conditions are checked at load time. The code group is associated with permission sets that may be granted to assemblies in the code group.

Code groups are arranged in hierarchies to support policy management. Permissions are resolved from the code group root looking for matching children. For an *exclusive code group*, assemblies matching such a code group are not evaluated against other code groups and get only permissions from this code group.

Policy management is further structured along four *policy levels*: enterprise, machine, user, and application domain. A policy level consists of a named permission set, a code

group hierarchy, and a list of 'full trust' assemblies. Evidence is evaluated against each policy level individually and the results are intersected.

Full trust assemblies are needed to avoid recursive security checks when loading assemblies. When a policy level is loaded, all referenced assemblies must be loaded. For permissions referred to in the permission sets in this policy level, the assemblies that contain these permissions have to be loaded. For example, assume the `System.Net.DnsPermission` permission is given in the assembly `System.dll`. To check whether this assembly may be loaded, the policy levels are required for making the decision and have to be loaded first. This creates deadlock; to load the permission, we must first check whether we have permission to load; to check the permission, we first have to load it.

20.6.7 Stack Walk

In the basic mode, all assemblies on the call stack are checked, but not the current method making the demand, and need to have the required permission for access to be granted. This algorithm can be modified with *Assert* actions that assert a permission and stop the stack walk for that permission. There is also a *Deny* action that terminates the stack walk, denying access (raising a security exception). The *PermitOnly* action is equivalent to a *Deny* on all permissions other than the one specified. These two actions are mainly useful for testing.

A stack walk for a code access permission, e.g. `FileIOPermission`, succeeds only when all callers considered have the permission. This effect is intended. The stack walk is a defence against luring attacks launched by components that do not have the permission in question. A stack walk for an identity permission succeeds only when all callers considered have the permission, i.e. the specific attribute stipulated in the permission. Consider a method that demands an identity permission that refers to the hash value of code. The caller that has this hash value is able to call this method. This effect is intended. However, when the stack walk reaches the frame for the next caller the hash values will not match and the stack walk will fail. For situations like this the stack walk has to be further adjusted.

The solution in .NET is the *Link Demand* action that performs a shallow check for the permissions a caller must have. Only the immediate caller is checked and no stack walk is performed. So, if a method *A* has a *Link Demand* for permission *p* and method *B* calls *A*, the assembly containing *B* is checked to see whether it has permission *p*.

In-lining is a compiler optimization that copies a method into the caller instead of making the call. A JIT compiler may collapse several levels of call into a single piece of code. An in-lined method has no frames on the stack and is therefore not seen by the stack walk. Only in-lining that traverses assembly boundaries may pose a problem. Hence, in-lining

is not possible for methods that use stack walk modifiers. Cross boundary in-lines are permitted only for *trivial* callees, i.e. callees that do not contain further method calls.

20.6.8 Summary

.NET security follows the same strategy as Java security. Managed code is written in a type-safe language such as C#. The CLR verifies Intermediate Language code to ensure type safety. Code (not users) is authorized to access resources and has to be authenticated when being executed. The security enforcement algorithm performs a stack walk. There are numerous means for structuring security policies and two different styles for expressing required permissions. This is the framework provided. For practical use, you would then need strategies for setting policies and assigning permissions to assemblies. These are still very much open questions.

20.7 DIGITAL RIGHTS MANAGEMENT

At the technical level, digital rights management (DRM) enforces policies of content providers on a customer machine. This is another departure from the traditional access control paradigm. The policies enforced on a system are no longer set by the owner but by an external party. The adversary is no longer an external party trying to subvert the system but an owner trying to bypass the policy. The security goal is the integrity of the access control system, as interpreted by the external party.

To achieve this goal, you could try to make it difficult for the owner to change the policy settings. If the adversary is assumed to be a technically sophisticated owner, protection mechanisms may have to go down to the hardware level. The content provider could ask for a truthful report about the hardware and software configuration of a target machine before agreeing to a download. This solution can be implemented on a trusted platform module (TPM) running an attestation protocol (Section 15.6). The TPM is the root of trust for the content owner.

PC vendors have no control over the operating systems and software loaded by the customer. With mobile phones there is a tighter integration between hardware and software. This facilitates the following approach to DRM.

- There are *domains* managed by an *authority*. In some cases there is only one domain for a given mobile phone operating systems (e.g. Symbian). In other cases multiple domains (Android uses the term *markets*) are feasible.
- User devices in the domain have the public verification key of the authority installed.
- An application that requires privileges from the mobile phone operating system has to be submitted to the authority. The authority validates that the application does not abuse the privileges requested. Approved applications are signed by the authority.

- Applications are distributed to users with their signatures and installed on user devices.
- When the application is run, the mobile phone operating system checks the signature. If the application has a valid signature it will be granted the privileges it requests.

Several variations of this general pattern have been implemented. The phone vendor and the hardware platform are the root of trust for the content owner. The Open Mobile Alliance is specifying industry standards for DRM.

Content owners may embed information about owner or customer in digital *watermarks* in the content delivered. The main issues here are the difficulty of removing or modifying watermarks, and their impact on the quality of audio and video content. When analyzing the security of such schemes, you have to be clear about your threat model. The adversary may be an unsophisticated user who will shy away from manipulating his device and should just be 'kept honest'. The adversary could be a technically skilled user who wants free access to content for private use. The adversary could be an organization wanting to distribute pirated content.

Whether DRM is a useful basis for business models in content distribution is still open to debate. Content providers not only rely on technology but also frequently have recourse to the legal system. EMI announced in April 2007 that they would offer content DRM-free. Content providers brought civil actions against unauthorized distributors of their content. They have also lobbied successfully for legislation prohibiting the reverse engineering of copy protection mechanisms. As noted by Lessig [153], legal code and software code are two alternative options for enforcing desired behaviour.

20.8 FURTHER READING

For detailed expositions of Java security and .NET security, the reader is referred to [163] and [141] respectively. A comparison of Java and .NET is given in [190]. Readers with an interest in the history of the field may consult [163] to learn about problems discovered in the early stages of Java security. By way of background on DRM, an entertaining review of the arms race between copy protection and copy programs can be found in [110]. Current research on code-based access control looks at alternatives to stack inspections [96], and at policy management [34].

20.9 EXERCISES

Exercise 20.1 Consider a policy that, for reasons of separation of duties, does not allow an entity to exercise the rights it may grant (delegate) to others. How could SPKI be augmented to support such a policy?

Exercise 20.2 Examine the provisions in the Java 2 security model for performing a *doPrivileged* against a context other than that of the current execution thread.

Exercise 20.3 Examine the provisions in the Java 2 security model for inheriting access control contexts.

Exercise 20.4 Analyze the relative advantages and disadvantages of declarative and imperative security actions.

Exercise 20.5 You are given a data-dependent policy rule. Would you implement this rule with a declarative or an imperative security action?

Exercise 20.6 You could use the hash of an assembly or a digital signature of the assembly as its identity. Examine the relative advantages and disadvantages of those two options.

Exercise 20.7 Consider a recursive function that calls itself *n* times (as a tail call). Compare the performance of the stack walk when there is no tail call elimination and when there is tail call elimination.

Exercise 20.8 Give an example of an application where access decisions are deferred to a third party for reasons other than trust.

Exercise 20.9 Specify an extension to the DRM access control model where applications can define their own name spaces that are protected from other applications.

Exercise 20.10 Give a brief description of the main features of OMA DRM 2.0.

Bibliography

[1] Martín Abadi. Two facets of authentication. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, pp. 27–32, 1998.

[2] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pp. 122–136, 1994.

[3] Ahmad Abu El-Asa, Martin Aeberhard, Frank J. Furrer, Ian Gardiner-Smith, and David Kohn. *Our PKI-Experience*. SYSLOGIC Press, Birmensdorf, Switzerland, 2002.

[4] Anne Adams and M. Angela Sasse. Users are not the enemy. *Communications of the ACM*, 42(12): 40–46, 1999.

[5] C. Alberts and A. Dorofee. *Managing Information Security Risks*. Pearson Education, Boston, 2003.

[6] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49, 1996.

[7] B. Alpern and S. Schneider. Defining liveness. *Information Processing Letters*, 21(4): 181–185, 1985.

[8] E. Amoroso. *Fundamentals of Computer Security Technology*. Prentice Hall International, Englewood Cliffs, NJ, 1994.

[9] J. Anderson. Computer security technology planning study. Technical Report 73-51, U.S. Air Force Electronic Systems Technical Report, October 1972.

[10] Ross Anderson. *Security Engineering*, 2nd edition. Wiley Publishing, Indianapolis, IN, 2008.

[11] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12): 52–59, 2000.

[12] J. Arceneaux. Experiences in the art of security. *Security Audit & Control – Review*, 14(4): 12–16, October 1996.

[13] Alfred W. Aresenault and Sean Turner. Internet X.509 Public Key Infrastructure – Roadmap, November 2000. Internet Draft draft-ietf-pkix-roadmap-06.txt.

[14] Ken Ashcraft and Dawson Engler. Using programmer-written compiler extensions to catch security holes. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*, pp. 143–159, 2002.

[15] Paul M. Ashley and Mark Vandenwauver. *Practical Intranet Security*. Kluwer Academic Publishers, Boston, 1999.

[16] Dmitri Asonov and Rakesh Agrawal. Keyboard acoustic emanations. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, pp. 3–11, 2003.

[17] Tuomas Aura. Cryptographically generated addresses (CGA). In Colin Boyd and Wenbo Mao (eds), *Information Security*, LNCS 2851, pp. 29–43. Springer-Verlag, Berlin, 2003.

[18] Tuomas Aura, Michael Roe, and Jari Arkko. Security of Internet location management. In *Proceedings of the 18th Annual Computer Security Applications Conference*, pp. 78–87, December 2002.

[19] D. B. Baker. Fortresses built upon sand. In *Proceedings of the New Security Paradigms Workshop*, pp. 148–153, September 1996.

[20] Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia, and Ed Simon. XML signature syntax and processing. Technical report, W3C, June 2008.

[21] Adam Barth, Collin Jackson, and John C. Mitchell. Robust defenses for cross-site request forgery. In Paul Syverson, Somesh Jha, and Xiaolan Zhang (eds), *CCS '08: Proceedings of the 15th ACM Conference on Computer and Communications Security*, pp. 75–88. ACM, New York.

[22] Richard Bejtlich. Interpreting network traffic: A network intrusion detector's look at suspicious events. In *Proceedings of the 12th Annual Computer Security Incidence Handling Conference*, Chicago, 2000.

[23] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations and model. Technical Report M74-244, The MITRE Corporation, Bedford, MA, May 1973.

[24] D. E. Bell and L. J. LaPadula. MITRE technical report 2547 (secure computer system): Volume II. *Journal of Computer Security*, 4(2/3): 239–263, 1996.

[25] David Bell and Leonard LaPadula. Secure computer system: Unified exposition and Multics implementation. Technical Report ESD-TR-75-306, The MITRE Corporation, Bedford, MA, July 1975.

[26] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keyed hash functions and message authentication. In N. Koblitz (ed.), *Advances in Cryptology – Crypto '96*, LNCS 1109, pp. 1–15. Springer-Verlag, Berlin, 1996.

[27] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In D. R. Stinson (ed.), *Advances in Cryptology – Crypto '93*, LNCS 773, pp. 232–249. Springer-Verlag, Berlin, 1994.

[28] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption padding. In A. De Santis (ed.), *Advances in Cryptology – Eurocrypt '94*, LNCS 950, pp. 92–111. Springer-Verlag, Berlin, 1995.

[29] S. Bellovin. Security problems in the TCP/IP protocol suite. *ACM Computer Communications Review*, 19(2): 32–48, April 1989.

[30] Steve M. Bellovin and Michael Merritt. Limitations of the Kerberos Authentication System. *ACM Computer Communications Review*, 20(5): 119–132, 1990.

[31] Steve M. Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, pp. 72–84, 1992.

[32] D. Berstis *et al*. IBM System/38 addressing and authorization. Technical Report GS80-0237, IBM System/38 Technical Development, 1978.

[33] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gross, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines

of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2): 66–75, February 2010.

[34] Frédéric Besson, Tomasz Blanc, Cédric Fournet, and Andrew D. Gordon. From stack inspection to access control: A security analysis for libraries. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop*, pp. 61–75. IEEE Computer Society, 2004.

[35] K. J. Biba. Integrity consideration for secure computer systems. Technical Report ESD-TR-76-372, MTR-3153, The MITRE Corporation, Bedford, MA, April 1977.

[36] P. Bieber, J. Cazin, P. Girard, J.-L. Lanet, V. Wiels, and G. Zanon. Checking secure interactions of smart card applets. In F. Cuppens *et al.* (eds), *Computer Security – ESORICS 2000*, LNCS 1895, pp. 1–16. Springer-Verlag, Berlin, 2000.

[37] M. Bishop and M. M. Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2): 131–152, 1996.

[38] B. Blakley. The emperor's old armour. In *Proceedings of the New Security Paradigms Workshop*, pp. 2–16, September 1996.

[39] Bob Blakley. *CORBA Security: An Introduction to Safe Computing with Objects*. Addison-Wesley, Reading, MA, 2000.

[40] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 164–173, 1996.

[41] Daniel Bleichenbacher. A chosen ciphertext attack against protocols based on RSA encryption standard PKCS #1. In H. Krawczyk (ed.), *Advances in Cryptology – Crypto '98*, LNCS 1462, pp. 1–12. Springer-Verlag, Berlin, 1998.

[42] Joseph L. Bower and Clayton M. Christensen. Disruptive technologies: Catching the wave. *Harvard Business Review*, January–February 1995.

[43] Chandrasekhar Boyapati and Martin Rinard. A parameterized type system for race-free Java programs. In *OOPSLA '01: Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp. 56–69, ACM, New York, 2001.

[44] D. F. C. Brewer and M. J. Nash. The Chinese Wall security policy. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pp. 206–214, 1989.

[45] Ernie Brickell, Jan Camenisch, and Liqun Chen. Direct anonymous authentication. In B. Pfitzmann *et al.* (eds), *Proceedings of the 11th ACM Conference on Computer and Communications Security*, pp. 132–145. ACM Press, New York, 2004.

[46] P. Brinch Hansen. *Operating Systems Principles*. Prentice Hall, Englewood Cliffs, NJ, 1973.

[47] Keith Brown. *Programming Windows Security*. Addison-Wesley, Boston, 2000.

[48] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5): 701–716, August 2005.

[49] BSI. IT-Grundschutz-Catalogues. Technical report, Bundesamt für Sicherheit in der Informationstechnik, Bonn, Germany, November 2005.

[50] Jesse Burns. Cross site reference forgery. Technical report, Information Security Partners, LLC, 2005. Version 1.1.

[51] Jan Camenisch. Better privacy for Trusted Computing Platforms. In P. Samarati *et al*. (eds), *Computer Security – ESORICS 2004*, LNCS 3193, pp. 73–88. Springer-Verlag, Berlin, 2004.

[52] J. Camenisch and A. Lysyanskaya. A signature scheme with efficient protocols. In S. Cimato, C. Galdi, and G. Persiano (eds), *Security in Communication Networks, SCN 2002*, LNCS 2576, pp. 268–289. Springer-Verlag, Berlin, 2003.

[53] Canadian System Security Centre. *The Canadian Trusted Computer Product Evaluation Criteria*, 1993. Version 3.0e.

[54] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. Password interception in a SSL/TLS channel. In D. Boneh (ed.), *Advances in Cryptology – Crypto '03*, LNCS 1462, pp. 583–599. Springer-Verlag, Berlin, 2003.

[55] Luca Cardelli. Type systems. In Allen B. Tucker, Jr (ed.), *The Computer Science and Engineering Handbook*, Chapter 103. CRC Press, Boca Raton, FL, 1997.

[56] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, Reading, MA, 1994.

[57] CCIB. *Common Methodolgy for Information Technology Security Evaluation – Part 2: Evaluation Methodolgy*, January 2004. Version 2.2.

[58] CCIB. *Common Criteria for Information Technology Security Evaluation – Part 1: Introduction and General Model*, July 2009. Version 3.1, release 3.

[59] CCITT. *The Directory – Overview of Concepts, Models and Services*, 1988. CCITT Rec X.500.

[60] CERT Coordination Center. Malicious HTML tags embedded in client web requests, 2000. `http://www.cert.org/advisories/CA-2000-02.html`.

[61] David Chaum. Blind signatures for untraceable payments. In *Advances in Cryptology – Proceedings of Crypto '82*, pp. 199–203. Plenum Press, New York, 1983.

[62] Brian Chess, Yekaterina Tsipenyuk O'Neil, and Jacob West. JavaScript hijacking. Technical report, Fortify Software, 2007.

[63] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security: Repelling the Wily Hacker*, 2nd edition. Addison-Wesley, Reading, MA, 2003.

[64] Elizabeth Chew, Marianne Swanson, Kevin Stine, Nadya Bartol, Anthony Brown, and Will Robinson. *Performance Measurement Guide for Information Security*. National Institute of Standards and Technology, July 2008. NIST Special Publication 800-55 Revision 1.

[65] S. Chokhani. Trusted product evaluations. *Communications of the ACM*, 35(7): 64–76, July 1992.

[66] D. R. Clark and D. R. Wilson. A comparison of commercial and military computer security policies. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 184–194, 1987.

[67] Club de la Sécurité de l'Information Français. *Mehari Risk Analysis Guide*, April 2007.

[68] CNSS. *National Policy on the Use of the Advanced Encryption Standard (AES) to Protect National Security Systems and National Security Information*, June 2003. CNSS Policy No. 15, Fact Sheet No. 1.

[69] Fred Cohen. Computer viruses. PhD thesis, University of Southern California, 1985.

[70] Commission of the European Communities. *Information Technology Security Evaluation Criteria (ITSEC)*, 1991. Version 1.2.

[71] Commission of the European Communities. *Information Technology Security Evaluation Manual (ITSEM)*, 1993.

[72] F. J. Corbato. On building systems that will fail. *Communications of the ACM*, 34(9): 72–81, September 1991.

[73] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pp. 63–78, 1998.

[74] Lorrie Cranor, Brooks Dobbs, Serge Egelman, Giles Hogben, Jack Humphrey, Marc Langheinrich, Massimo Marchiori, Martin Presler-Marshall, Joseph Reagle, Matthias Schunter, David A. Sampley, and Rigo Wenning. The Platform for Privacy Preferences 1.1 (P3P1.1) specification. Technical report, W3C, November 2006. W3C Working Group Draft.

[75] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael. Technical report, September 1999. AES Algorithm Submission.

[76] C. J. Date. *An Introduction to Database Systems – Volume I*, 8th edition. Addison-Wesley, Reading, MA, 2003.

[77] John Daugman. High confidence visual recognition of persons by a test of statistical independence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15(11): 1148–1161, 1993.

[78] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from HotJava to Netscape and beyond. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pp. 190–200, 1996.

[79] D. E. Denning, T. F. Lunt, R. R. Schell, W. R. Shockley, and M. Heckman. The SeaView Security Model. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, pp. 218–233, 1988.

[80] Dorothy E. Denning. *Cryptography and Security*. Addison-Wesley, Reading, MA, 1982.

[81] Dorothy E. Denning and Giovanni M. Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8): 533–536, 1981.

[82] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22: 644–654, 1976.

[83] Hans Dobbertin. Cryptanalysis of MD4. In D. Gollmann (ed.), *Fast Software Encryption*, LNCS 1039, pp. 53–69. Springer-Verlag, Berlin, 1996.

[84] S. Dreyfuss. *Underground*. Reed Books, 1997.

[85] M. W. Eichin and J. A. Rochlis. With microscope and tweezers: An analysis of the Internet virus of November 1988. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pp. 326–343, 1989.

[86] Tahir ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4): 469–472, 1985.

[87] J. H. Ellis. The possibility of non-secret encryption. Technical report, CESG, January 1970. `http://cryptome.org/jya/ellisdoc.htm`.

[88] Carl M. Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M. Thomas, and Tatu Ylonen. *SPKI Certificate Theory*, September 1999. RFC 2693.

[89] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, pp. 75–88, 2006.

[90] Úlfar Erlingsson and Fred B. Schneider. IRM enforcement of Java stack inspection. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy*, pp. 246–255, 2000.

[91] European Computer Manufacturers Association. Secure information processing versus the concept of product evaluation. Technical Report ECMA TR/64, December 1993.

[92] R. S. Fabry. Capability-based addressing. *Communications of the ACM*, 17(7): 403–412, July 1974.

[93] D. C. Feldmeier and P. R. Karn. UNIX password security – ten years later. In G. Brassard (ed.), *Advances in Cryptology – Crypto '89*, LNCS 435, pp. 44–63. Springer-Verlag, Berlin, 1990.

[94] Scott R. Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of RC4. In S. Vaudenay and A. M. Youssef (eds), *Selected Areas in Cryptography*, LNCS 2259, pp. 1–24. Springer-Verlag, Berlin, 2001.

[95] James C. Foster. *Buffer Overflow Attacks*. Syngress Publishing, Rockland, MA, 2005.

[96] Cédric Fournet and Andrew D. Gordon. Stack inspection: Theory and variants. *ACM Transactions on Programming Languages and Systems*, 25(3): 360–399, May 2003.

[97] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, New York, 1988.

[98] M. Gasser. The role of naming in secure distributed systems. In *Proceedings of the CS'90 Symposium on Computer Security*, pp. 97–109, Rome, November 1990.

[99] M. Gasser, A. Goldstein, C. Kaufman, and B. Lampson. The digital distributed system security architecture. In *Proceedings of the 1989 National Computer Security Conference*, 1989.

[100] V. D. Gligor. A note on denial of service in operating systems. *IEEE Transactions on Software Engineering*, 10(3): 320–324, May 1984.

[101] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, pp. 11–20, 1982.

[102] Ian Goldberg and David Wagner. Randomness and the Netscape browser. *Dr. Dobb's Journal*, pp. 3–38, January 1996.

[103] Dieter Gollmann. Authentication by correspondence. *IEEE Journal on Selected Areas in Communications*, 21(1): 88–95, January 2003.

[104] Li Gong. *Inside Java 2 Platform Security*. Addison-Wesley, Reading, MA, 1999.

[105] Li Gong, Mary Dageforde, and Gary W. Ellison. *Inside Java 2 Platform Security*, 2nd edition. Addison-Wesley, Reading, MA, 2003.

[106] Sudhakar Govindavajhala and Andrew W. Appel. Using memory errors to attack a virtual machine. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pp. 154–165, 2003.

[107] Mark G. Graff and Kenneth R. van Wyk. *Secure Coding*. O'Reilly & Associates, Cambridge, 2003.

[108] Robert M. Graham. Protection in an information processing utility. *Communications of the ACM*, 11(5): 365–369, 1968.

[109] Andrew Granville. It is easy to determine whether a given integer is prime. *Bulletin (New Series) of the American Mathematical Society*, 42(1): 3–38, 2005.

[110] D. Grover (ed.). *The Protection of Computer Software – its Technology and Applications*, 2nd edition. Cambridge University Press, Cambridge, 1992.

[111] Shai Halevi and Hugo Krawczyk. Public-key cryptography and password protocols. *ACM Transactions on Information and System Security*, 2(3): 230–268, August 1999.

[112] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pp. 6982. ACM Press, New York, 2002.

[113] Norm Hardy. The confused deputy. *Operating Systems Reviews*, 22(4): 36–38, 1988.

[114] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8): 461–471, August 1976.

[115] M. E. Hellman. A cryptanalytic time–memory trade-off. *IEEE Transactions on Information Theory*, 26(4): 401–406, 1980.

[116] J. Hennessy and D. Patterson. *Computer Architecture – A Quantitative Approach*, 3rd edition. Morgan Kaufmann, San Mateo, CA, 2002.

[117] F. H. Hinsley and Alan Stripp. *Codebreakers*. Oxford University Press, Oxford, 1993.

[118] Giles Hogben, Tom Jackson, and Marc Wilikens. A fully compliant research implementation of the P3P standard. In D. Gollmann *et al.* (eds), *Computer Security – ESORICS 2002*, LNCS 2502, pp. 104–120. Springer-Verlag, Berlin, 2002.

[119] Greg Hoglund and Gary McGraw. *Exploiting Software: How to Break Code*. Addison-Wesley, Boston, 2004.

[120] Russell Housley, Tim Polk, Warwick Ford, and David Solo. Internet X.509 Public Key Infrastructure – Certificate and Certificate Revocation List (CRL) Profile, April 2002. RFC 3280.

[121] Michael Howard and David LeBlanc. *Writing Secure Code*, 2nd edition. Microsoft Press, Redmond, WA, 2002.

[122] Michael Howard, David LeBlanc, and John Viega. *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*. McGraw-Hill, New York, 2010.

[123] Michael Howard and Steve Lipner. *The Security Development Life Cycle*. Microsoft Press, Redmond, WA, 2006.

[124] Michael Howard, Jon Pincus, and Jeanette M. Wing. Measuring relative attack surfaces. Technical Report CMU-CS-03-169, Carnegie Mellon University, Pittsburgh, PA, 2003.

[125] International Organization for Standardization. *Basic Reference Model for Open Systems Interconnection (OSI) Part 2: Security Architecture*. Geneva, Switzerland, 1989.

[126] International Organization for Standardization. *Information Technology – Security Techniques – Entity Authentication Mechanisms; Part 1: General Model*. Geneva, Switzerland, September 1991. ISO/IEC 9798-1, second edition.

[127] International Organization for Standardization. *ISO/IEC 9075: Information Technology – Database Languages – SQL*. Geneva, Switzerland, 1992.

[128] International Organization for Standardization. *Information Technology – Open Systems Interconnection – The Directory-Authentication Framework*. Geneva, Switzerland, June 1997. ISO/IEC 9594-8—ITU-T Rec X.509 (1997 E).

[129] International Organization for Standardization. *Information Technology – Security Techniques – Code of Practice for Information Security Management*. Geneva, Switzerland, 2005.

[130] Collin Jackson, Adam Barth, Andrew Bortz, Weidong Shao, and Dan Boneh. Protecting browsers from DNS rebinding attacks. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, pp. 421–431, 2007.

[131] Martin Johns. SessionSafe: Implementing XSS immune session handling. In D. Gollmann *et al.* (eds), *Computer Security – ESORICS 2006*, LNCS 4189, pp. 444–460. Springer-Verlag, Berlin, 2006.

[132] Martin Johns. (Somewhat) breaking the same-origin policy by undermining DNS pinning. Posting to the Bug Traq Mailinglist, August 2006. `http://www.securityfocus.com/archive/107/443429/30/180/threaded`.

[133] Martin Johns and Justus Winter. RequestRodeo: Client side protection against session riding. In F. Piessens (ed.), *Proceedings of the OWASP Europe 2006 Conference*, pp. 5–17. Departement Computerwetenschappen, Katholieke Universiteit Leuven, Report CW448, May 2006.

[134] D. Kahn. *The Codebreakers*. Macmillan, New York, 1967.

[135] M. H. Kang, A. P. Moore, and I. S. Moskowitz. Design and assurance strategy for the NRL pump. *IEEE Computer*, 31(4): 56–64, November 1998.

[136] P. A. Karger. Implementing commercial data integrity with secure capabilities. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pp. 130–139, 1991.

[137] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn. A VMM security kernel for the VAX architecture. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pp. 1–19, 1990.

[138] Paul A. Karger and Roger R. Schell. Thirty years later: Lessons from the Multics security evaluation. In *Proceedings of ACSAC 2002*, pp. 119–148, 2002.

[139] Hugo Krawczyk. HMVQ: A high performance secure Diffie-Hellman protocol. In V. Shoup (ed.), *Advances in Cryptology – Crypto 2005*, LNCS 3621. Springer-Verlag, Berlin, 2005.

[140] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. HMAC: Keyed-hashing for message authentication. Technical report, February 1997. RFC 2104.

[141] Brian A. La Macchia, Sebastian Lange, Matthew Lyons, Rudi Martin, and Kevin T. Price. `.NET` Framework Security. Addison-Wesley Professional, Boston, 2002.

[142] L. Lamport. Constructing digital signatures from a one way function. Technical Report CSL-98, SRI International Computer Science Laboratory, October 1979.

[143] L. Lamport. Checking secure interactions of smart card applets. In M. W. Alford *et al.* (eds), *Distributed Systems: Methods and Tools for Specification: An Advanced Course*, LNCS 190, pp. 119–130. Springer-Verlag, Berlin, 1985.

[144] B. Lampson. Protection. *ACM Operating Systems Reviews*, 8(1): 18–24, January 1974.

[145] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4): 265–310, November 1992.

[146] C. Landwehr. The best available technologies for computer security. *IEEE Computer*, 16(7): 86–100, 1983.

[147] Ulrich Lang and Rudolf Schreiner. *Developing Secure Distributed Systems with CORBA*. Artech House, Norwood, MA, 2002.

[148] J.-C. Laprie. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1992.

[149] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone. An efficient protocol for authenticated key agreement. *Designs, Codes and Cryptography*, 28: 119–134, 2003.

[150] P. Le Hégaret, R. Whitmer, and L. Wood. Document object model (DOM). W3C Recommendation, January 2005. `http://www.w3.org/DOM/`.

[151] Ruby B. Lee, David K. Karig, John P. McGregor, and Zhijie Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the International Conference on Security in Pervasive Computing (SPC-2003)*, LNCS 2802, pp. 237–252. Springer-Verlag, Berlin, 2003.

[152] T. M. P. Lee. Using mandatory integrity to enforce 'commercial' security. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pp. 140–146, 1991.

[153] Lawrence Lessig. *Code and Other Laws of Cyberspace*. Basic Books, New York, 1999.

[154] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java Virtual Machine. In *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pp. 3644. ACM, New York, 1998.

[155] R. J. Lipton and L. Snyder. On synchronization and security. In R. D. Demillo *et al*. (eds), *Foundations of Secure Computation*, pp. 367–385. Academic Press, New York, 1978.

[156] T. F. Lunt. Aggregation and inference: Facts and fallacies. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pp. 102–109, 1989.

[157] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6): 593–607, 1990.

[158] D. MacKenzie and G. Pottinger. Mathematics, technology, and trust: Formal verification, computer security, and the U.S. Military. *IEEE Annals of the History of Computing*, 19(3): 41–59, 1997.

[159] James Manger. A chosen ciphertext attack on RSA Optimal Asymmetric Encryption Padding (OAEP) as standardized in PKCS # 1 v2.0. In J. Kilian (ed.), *Advances in Cryptology – Crypto 2001*, LNCS 2139, pp. 230–238. Springer-Verlag, Berlin, 2001.

[160] Ray Marsh and Steve Dispensa. Renegotiating TLS. Technical report, PhoneFactor Inc., Malvern, November 2009.

[161] Tsutomu Matsumoto. Gummy and conductive silicon rubber fingers – importance of vulnerability analysis. In Y. Zheng (ed.), *Advances in Cryptology – Asiacrypt 2002, Queenstown, New Zealand*, LNCS 2501, pp. 574–575. Springer-Verlag, Berlin, 2002.

[162] Alexander Maximov and Dmitry Khovratovich. New state recovery attack on RC4. In D. Wagner (ed.), *Advances in Cryptology – Crypto 2008*, LNCS 5157, pp. 297–316. Springer-Verlag, Berlin, 2008.

[163] G. McGraw and E. W. Felten. *Java Security*. John Wiley & Sons, New York, 1997.

[164] J. McLean. Reasoning about security models. In *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, pp. 123–131, 1987.

[165] J. McLean. The specification and modeling of computer security. *IEEE Computer*, 23(1): 9–16, January 1990.

[166] J. McLean. Security models. In J. Marciniak (ed.), *Encyclopedia of Software Engineering*. John Wiley & Sons, New York, 1994.

[167] Peter Mell, Karen Scarfone, and Sasha Romanovsky. A complete guide to the common vulnerability scoring system. Technical report, June 2007. Version 2.0.

[168] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL, 1997.

[169] B. F. Miller, L. Frederiksen, and B. So. An empirical study of the reliability of Unix utilities. *Communications of the ACM*, 33(12): 32–44, December 1990.

[170] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer. Section E.2.1: Kerberos authentication and authorization system. Technical report, MIT Project Athena, Cambridge, MA, 1987.

[171] Christopher J. Mitchell and Paolo S. Pagliusi. Is entity authentication necessary? In B. Christiansen *et al.* (eds), *Security Protocols, 10th International Workshop, Cambridge*, LNCS 2845, pp. 20–33. Springer-Verlag, 2003.

[172] Kevin D. Mitnick and William L. Simon. *The Art of Deception*. Wiley Publishing, Indianapolis, IN, 2002.

[173] R. Morris and K. Thompson. Password security: A case history. *Communications of the ACM*, 22(11): 594–597, November 1979.

[174] R. T. Morris. A weakness in the 4.2BSD Unix TCP/IP software. Bell Labs Computer Science Technical Report, February 1985.

[175] National Computer Security Center. *Trusted Network Interpretation*, 1987. NCSC-TG-005, Version 1.0.

[176] National Computer Security Center. *Trusted Database Management Ssystem Interpretation*, April 1991. NCSC-TG-021.

[177] National Institute of Standards and Technology & National Security Agency. *Federal Criteria for Information Technology Security*, 1992. Version 1.0.

[178] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pp. 106–119, Paris, January 1997.

[179] R. M. Needham. Later developments at Cambridge: Titan, CAP, and the Cambridge Ring. *Annals of the History of Computing*, 14(4): 57, 1992.

[180] R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12): 993–999, 1978.

[181] R. P. Nelson. *The 80386 Book*. Microsoft Press, Redmond, WA, 1988.

[182] Valtteri Niemi and Kaisa Nyberg. *UMTS Security*. John Wiley & Sons, Chichester, 2003.

[183] OASIS. Assertions and protocols for the OASIS Security Assertion Markup Language (SAML) V2.0. Technical report, OASIS Standard, March 2005.

[184] OASIS. eXtensible Access Control Markup Language (XACML) Version V2.0. Technical report, OASIS Standard, February 2005.

[185] Machigar Ongtang, Steven McLaughlin, William Enck, and Patrick McDaniel. Semantically rich application-centric security in Android. In *Proceedings ACSAC 2009*, pp. 207–216, 1987.

[186] Rolf Oppliger, Ralf Hauser, and David A. Basin. SSL/TLS session-aware user authentication. *IEEE Computer*, 41(3): 59–65, 2008.

[187] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, 1972.

[188] Organisation for Economic Co-operation and Development. *OECD Guidelines on the Protection of Privacy and Transborder Flows of Personal Data*, December 1980. Republished February 2002.

[189] J. S. Park. *AS/400 Security in a Client/Server Environment*. John Wiley & Sons, New York, 1995.

[190] Nathanael Paul and David Evans. `.NET` security: Lessons learned and missed from Java. In *Proceedings of ACSAC 2004*, pp. 272–281, 2004.

[191] C. P. Pfleeger and S. Lawrence Pfleeger. *Security in Computing*, 4th edition. Prentice Hall, Englewood Cliffs, NJ, 2007.

[192] B. Preneel, B. B. Van Rompay, S. B. Örs, A. Biryukov, L. Granboulan, E. Dottax, M. Dichtl, M. Schafheutle, P. Serf, S. Pyka, E. Biham, E. Barkan, O. Dunkelman, J. Stolin, M. Ciet, J.-J. Quisquater, F. Sica, H. Raddum, and M. Parker. Performance of optimized implementations of the NESSIE primitives. Technical report, February 2003. Version 2.0.

[193] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998.

[194] Ron Rivest, Adi Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2): 120–126, 1978.

[195] D. Russel and G. T. Gangemi Sr. *Computer Security Basics*. O'Reilly & Associates, Sebastopol, CA, 1991.

[196] J. H. Saltzer. Protection and the control of information sharing in Multics. *Communications of the ACM*, 17: 388–402, 1974.

[197] S. Samalin. *Secure Unix*. McGraw-Hill, New York, 1997.

[198] Ravi S. Sandhu, David Ferraiolo, and Richard Kuhn. The NIST model for role based access control: Toward a unified standard. In *Proceedings of the 5th ACM Workshop on Role Based Access Control*, pp. 47–63, July 2000.

[199] R. S. Sandhu. Lattice-based access control models. *IEEE Computer*, 26(11): 9–19, November 1993.

[200] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2): 38–47, February 1996.

[201] Augustin P. Sarr, Philippe Elbaz-Vincent, and Jean-Claude Bajard. A secure and efficient authenticated Diffie–Hellman protocol. In F. Martinelli and B. Preneel (eds) *Proceedings of EuroPKI 2009*, LNCS 6391, pp. 83–98, Springer-Verlag, Berlin, 2010.

[202] M. Angela Sasse. Privacy and security – not seeing the crime for the cameras? *Communications of the ACM*, 53(2): 22–25, February 2010.

[203] Marvin Schaefer. If A1 is the answer, what was the question? An edgy naïf's retrospective on promulgating the Trusted Computer Systems Evaluation Criteria. In *Proceedings of ACSAC 2004*, pp. 204–228, 2004.

[204] Gerhard Schellhorn, Wolfgang Reif, Axel Schairer, Paul Karger, Vernon Austel, and David Toll. Verification of a formal security model for multiapplicative smart

cards. In F. Cuppens *et al.* (eds), *Computer Security – ESORICS 2000*, LNCS 1895, pp. 17–36. Springer-Verlag, Berlin, 2000.

[205] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1): 30–50, 2000.

[206] B. Schneier. *Applied Cryptography*, 2nd edition. John Wiley & Sons, New York, 1996.

[207] M. Schroeder and J. H. Saltzer. An hardware architecture for implementing protection rings. *Communications of the ACM*, 15(3): 157–170, 1972.

[208] T. Shimomura. *Takedown*. Secker & Warburg, London, 1995.

[209] John F. Shoch and Jon A. Hupp. The 'worm' programs – early experience with a distributed computation. *Communications of the ACM*, 25(3): 172–180, September 1982.

[210] Victor Shoup. OAEP reconsidered. In J. Kilian (ed.), *Advances in Cryptology – Crypto 2001*, LNCS 2139, pp. 239–259. Springer-Verlag, Berlin, 2001.

[211] Stephen Smalley. Configuring the SELinux policy. Technical Report NAI Labs Report #02-007, National Security Agency, 2002, revised 2005.

[212] H. J. Smith. Privacy policies and practices: Inside the organizational maze. *Communications of the ACM*, 36(12): 104–122, December 1993.

[213] E. H. Spafford. Crisis and aftermath. *Communications of the ACM*, 32(6): 678–687, June 1989.

[214] Lance Spitzner. Honeypots. `http//www.tracking-hackers.com`, May 2003.

[215] William Stallings. *Cryptography and Network Security*, 5th edition. Prentice Hall, 2011.

[216] Daniel F. Sterne. On the buzzword 'Security Policy'. In *Proceedings of the 1991 IEEE Symposium on Research in Security and Privacy*, pp. 219–230, 1991.

[217] C. Stoll. *The Cuckoo's Egg*. Simon & Schuster, New York, 1989.

[218] Michael M. Swift, Anne Hopkins, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control for Windows 2000. *ACM Transactions on Information and System Security*, 5(4): 398–437, 2002.

[219] UK ITSEC Scheme. *Description of the Scheme*, March 1991. UKSP 01.

[220] United Nations. *International Review of Criminal Policy – United Nations Manual on the Prevention and Control of Computer-Related Crime*. New York, 1999.

[221] U.S. Department of Commerce, National Bureau of Standards. *Data Encryption Standard*, January 1977. NBS FIPS PUB 46.

[222] U.S. Department of Commerce, National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, January 2000. FIPS PUB 186-2.

[223] U.S. Department of Commerce, National Institute of Standards and Technology. *Advanced Encryption Standard (AES)*, November 2001. FIPS 197.

[224] U.S. Department of Defense. *DoD Trusted Computer System Evaluation Criteria*, 1985. DOD 5200.28-STD.

[225] U.S. Department of Defense. *Industrial Security Manual for Safeguarding Classified Information*, June 1987. DOD 5220.22-M.

[226] Ton Van der Putte and Jeroen Keuning. Biometrical fingerprinting recognition: Don't get your fingers burned. In Josep Domingo-Ferrer, David Chan, and Anthony Watson (eds), *Smart Card Research and Applications*, pp. 289–303. Kluwer Academic Publishers, 2000.

[227] John Viega and Matt Messier. *Secure Programming Cookbook for C and C++*. O'Reilly, Cambridge, 2003.

[228] J. von Neumann. First draft of a report on the EDVAC (M. D. Godfrey (ed.)). *Annals of the History of Computing*, 15(4): 27–75, 1993.

[229] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *Proceedings of the Symposium on Operating System Principles*, 1993.

[230] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. In V. Shoup (ed.), *Advances in Cryptology – Crypto 2005*, LNCS 3621, pp. 17–36. Springer-Verlag, Berlin.

[231] Willis H. Ware. Security controls for computer systems. Technical Report R-609, The RAND Corporation, Santa Monica, CA, January 1970.

[232] C. Weissman. BLACKER: Security for the DDN, examples of A1 security engineering trades. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, pp. 286–292, 1992.

[233] Gordon Welchman. *The Hut Six Story*. Allan Lane, London, 1982.

[234] M. V. Wilkes. *Time-Sharing Computer Systems*. American Elsevier, New York, 1968.

[235] Thomas Wu. A real-world analysis of Kerberos password security. In *Proceedings of the 1999 Network and Distributed System Security Symposium*. Internet Society, February 1999.

[236] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In Ninghui Li Peng Ning, Sihan Qing (eds), *ICICS 2006*, LNCS 4307, pp. 379–398. Springer-Verlag, Berlin, 2006.

[237] Yves Younan, Wouter Joosen, Frank Piessens, and Hans Van den Eynden. Security of memory allocators for C and C++. Technical Report CW419, KU Leuven, July 2005.

[238] Elizabeth D. Zwicky, Simon Cooper, and D. Brent Chapman. *Building Internet Firewalls*, 2nd edition. O'Reilly & Associates, Sebastopol, CA, 2000.

Index

Note: Page references in *italics* refer to Figures

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

X

Z