Principles of Computer System Design

An Introduction

Chapter 11
Information Security

Jerome H. Saltzer

M. Frans Kaashoek

Massachusetts Institute of Technology

Version 5.0

Suggestions, Comments, Corrections, and Requests to waive license restrictions: Please send correspondence by electronic mail to:

Saltzer@mit.edu

and

kaashoek@mit.edu

CHAPTER

Information Security 11

Information security. The protection of information and information systems against unauthorized access or modification of information, whether in storage, processing, or transit, and against denial of service to authorized users.

— *Information Operations*. Joint Chiefs of Staff of the United States Armed Forces, Joint Publication 3-13 (13 February 2006).

CHAPTER CONTENTS

Overview

Secure computer systems ensure that users' privacy and possessions are protected against malicious and inquisitive users. Security is a broad topic, ranging from issues such as not allowing your friend to read your files to protecting a nation's infrastructure against attacks. Defending against an adversary is a *negative* goal. The designer of a computer system must ensure that an adversary cannot breach the security of the system in *any* way. Furthermore, the designer must make it difficult for an adversary to side-step the security mechanism; one of the simplest ways for an adversary to steal confidential information is to bribe someone on the inside.

Because security is a negative goal, it requires designers to be careful and pay attention to the details. Each detail might provide an opportunity for an adversary to breach the system security. Fortunately, many of the previously-encountered design principles can also guide the designer of secure systems. For example, the principles of the *safety net* approach from Chapter 8[on-line], *be explicit* (state your assumptions so that they can be reviewed) and *design for iteration* (assume you will make errors), apply equally, or perhaps even with more force, to security.

The conceptual model for protecting computer systems against adversaries is that some agent presents to a computer system a claimed identity and requests the system to perform some specified action. To achieve security, the system must obtain trustworthy answers to the following three questions before performing the requested action:

1. Authenticity: Is the agent's claimed identity authentic? (Or, is someone masquerading as the agent?)
2. Integrity: Is this request actually the one the agent made? (Or, did someone tamper with it?)
3. Authorization: Has a proper authority granted permission to this agent to perform this action?

The primary underpinning of security of a system is the set of mechanisms that ensures that these questions are answered satisfactorily for every action that the system performs. This idea is known as the principle of

Complete mediation

For every requested action, check authenticity, integrity, and authorization.

To protect against inside attacks (adversaries who are actually users that have the appropriate permissions, but abuse them) or adversaries who successfully break the security mechanisms, the service must also maintain audit trails of who used the system, what authorization decisions have been made, etc. This information may help determine who the adversary was after the attack, how the adversary breached the security of the system, and bring the adversary to justice. In the end, a primary instrument to deter adversaries is to increase the likelihood of detection and punishment.

The next section provides a general introduction to security. It discusses possible threats (Section 11.1.1), why security is a negative goal (Section 11.1.2), presents the safety net approach (Section 11.1.3), lays out principles for designing secure computer systems (Section 11.1.4), the basic model for structuring secure computer systems (Section 11.1.6), an implementation strategy based on minimizing the trusted computing base (Section 11.1.7), and concludes with a road map for the rest of this chapter (Section 11.1.8). The rest of the chapter works the ideas introduced in the next section in more detail, but by no means provides a complete treatment of computer security. Computer security is an active area of research with many open problems and the interested reader is encouraged to explore the research literature to get deeper into the topic.

11.1 Introduction to Secure Systems

In Chapter 4 we saw how to divide a computer system into modules so that errors don't propagate from one module to another. In the presentation, we assumed that errors happen *unintentionally*: modules fail to adhere to their contracts because users make mistakes or hardware fails accidently. As computer systems become more and more deployed for mission-critical applications, however, we require computer systems that can tolerate adversaries. By an *adversary* we mean a entity that breaks into systems *intentionally*, for example, to steal information from other users, to blackmail a company, to deny other users access to services, to hack systems for fun or fame, to test the security of a system, etc. An adversary encompasses a wide range of bad guys as well as good guys (e.g., people hired by an organization to test the security of that organization's computers systems). An adversary can be a single person or a group collaborating to break the protection.

Almost all computers are connected to networks, which means that they can be attacked by an adversary from any place in the world. Not only must the security mechanism withstand adversaries who have physical access to the system, but the mechanism also must withstand a 16-year old wizard sitting behind a personal computer in some country one has never heard of. Since most computers are connected through *public* networks (e.g., the Internet), defending against a remote adversary is particularly challenging. Any person who has access to the public network might be able to compromise any computer or router in the network.

Although, in most secure systems, keeping adversaries from doing bad things is the primary objective, there is usually also a need to provide users with different levels of authority. Consider electronic banking. Certainly, a primary objective must be to ensure that no one can steal money from accounts, modify transactions performed over the public networks, or do anything else bad. But in addition, a banking system must enforce other security constraints. For example, the owner of an account should be allowed to withdraw money from the account, but the owner shouldn't be allowed to withdraw money from other accounts. Bank personnel, though, (under some conditions) should be allowed to transfer money between accounts of different users and view any account. Some scheme is needed to enforce the desired authority structure.

In some applications no enforcement mechanism internal to the computer system may be necessary. For instance, an externally administered code of ethics or other mechanisms outside of the computer system may protect the system adequately. On the other hand, with the rising importance of computers and the Internet many systems require some security plan. Examples include file services storing private information, Internet stores, law enforcement information systems, electronic distribution of proprietary software, on-line medical information systems, and government social service data processing systems. These examples span a wide range of needs for organizational and personal privacy.

Not all fields of study use the terms "privacy," "security," and "protection" in the same way. This chapter adopts definitions that are commonly encountered in the computer science literature. The traditional meaning of the term *privacy* is the ability of an individual to determine if, when, and to whom personal information is to be released (see Sidebar 11.1). The term *security* describes techniques that protect information and information systems against unauthorized access or modification of information, whether in storage, processing, or transit, and against denial of service to authorized users. In this chapter the term *protection* is used as a synonym for security.

Sidebar 11.1: Privacy The definition of privacy (the ability of an individual to determine if, when, and to whom personal information is to be released) comes from the 1967 book *Privacy and Freedom* by Alan Westin [Suggestions for Further Reading 1.1.6]. Some privacy advocates (see for example Suggestions for Further Reading 11.1.2) suggest that with the increased interconnectivity provided by changing technology, Westin's definition now covers only a subset of privacy, and is in need of update. They suggest this broader definition: the ability of an individual to decide how and to what extent personal information can be used by others.

This broader definition includes the original concept, but it also encompasses control over use of information that the individual has agreed to release, but that later can be systematically accumulated from various sources such as public records, grocery store frequent shopper cards, Web browsing logs, on-line bookseller records about what books that person seems interested in, etc.. The reasoning is that modern network and data mining technology add a new dimension to the activities that can constitute an invasion of privacy. The traditional definition implied that privacy can be protected by confidentiality and access control mechanisms; the broader definition implies adding accountability for use of information that the individual has agreed to release.

A common goal in a secure system is to enforce some privacy policy. An example of a policy in the banking system is that only an owner and selected bank personnel should have access to that owner's account. The nature of a privacy policy is not a technical question, but a social and political question. To make progress without having to solve the problem of what an acceptable policy is, we focus on the mechanisms to enforce policies. In particular, we are interested in mechanisms that can support a wide variety of policies. Thus, the principle *separate mechanism from policy* is especially important in design of secure systems.

11.1.1 Threat Classification

The design of any security system starts with identifying the threats that the system should withstand. *Threats* are potential security violations caused either by a planned attack by an adversary or unintended mistakes by legitimate users of the system. The designer of a secure computer system must be consider both.

There are three broad categories of threats:

1. Unauthorized information release: an unauthorized person can read and take advantage of information stored in the computer or being transmitted over networks. This category of concern sometimes extends to "traffic analysis," in which the adversary observes only the patterns of information use and from those patterns can infer some information content.
2. Unauthorized information modification: an unauthorized person can make changes in stored information or modify messages that cross a network—an

adversary might engage in this behavior to sabotage the system or to trick the receiver of a message to divulge useful information or take unintended action. This kind of violation does not necessarily require that the adversary be able to see the information it has changed.

3. Unauthorized denial of use: an adversary can prevent an authorized user from reading or modifying information, even though the adversary may not be able to read or modify the information. Causing a system "crash," flooding a service with messages, or firing a bullet into a computer are examples of denial of use. This attack is another form of sabotage.

In general, the term "unauthorized" means that release, modification, or denial of use occurs contrary to the intent of the person who controls the information, possibly even contrary to the constraints supposedly enforced by the system.

As mentioned in the overview, a complication in defending against these threats is that the adversary can exploit the behavior of users who are legitimately authorized to use the system but are lax about security. For example, many users aren't security experts and put their computers at risk through surfing the Internet and downloading untrusted, third-party programs voluntarily or even without realizing it. Some users bring their own personal devices and gadgets into their work place; these devices may contain malicious software. Yet other users allow friends and family members to use computers at institutions for personal ends (e.g., storing personal content or playing games). Some employees may be disgruntled with their company and may be willing to collaborate with an adversary.

A legitimate user acting as an adversary is difficult to defend against because the adversary's actions will appear to be legitimate. Because of this difficulty, this threat has its own label, the *insider threat*.

Because there are many possible threats, a broad set of security techniques exists. The following list just provides a few examples (see Suggestions for Further Reading 1.1.7 for a wider range of many more examples):

- making credit card information sent over the Internet unreadable by anyone other than the intended recipients,
- verifying the claimed identity of a user, whether local or across a network,
- labeling files with lists of authorized users,
- executing secure protocols for electronic voting or auctions,
- installing a router (in security jargon called a firewall) that filters traffic between a private network and a public network to make it more difficult for outsiders to attack the private network,
- shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation,
- locking the room containing the computer,
- certifying that the hardware and software are actually implemented as intended,

- providing users with configuration profiles to simplify configuration decisions with secure defaults,
- encouraging legitimate users to follow good security practices,
- monitoring the computer system, keeping logs to provide audit trails, and protecting the logs from tampering.

11.1.2 Security is a Negative Goal

Having a narrow view of security is dangerous because the objective of a secure system is to prevent *all* unauthorized actions. This requirement is a negative kind of requirement. It is hard to prove that this negative requirement has been achieved, for one must demonstrate that *every possible* threat has been anticipated. Therefore, a designer must take a broad view of security and consider any method in which the security scheme can be penetrated or circumvented.

To illustrate the difficulty, consider the positive goal, "Alice can read file x." It is easy to test if a designer has achieved the goal (we ask Alice to try to read the file). Furthermore, if the designer failed, Alice will probably provide direct feedback by sending the designer a message "I can't read x!" In contrast, with a negative goal, such as "Lucifer cannot read file x", the designer must check that all the ways that the adversary Lucifer might be able to read x are blocked, and it's likely that the designer won't receive any direct feedback if the designer slips up. Lucifer won't tell the designer because Lucifer has no reason to and it may not even be in Lucifer's interest.

An example from the field of biology illustrates nicely the difference between proving a positive and proving a negative. Consider the question "Is a species (for example, the Ivory-Billed Woodpecker) extinct?'' It is generally easy to prove that a species exists; just exhibit a live example. But to prove that it is extinct requires exhaustively searching the whole world. Since the latter is usually difficult, the most usual answer to proving a negative is "we aren't sure".*

The question "Is a system secure?" has these same three possible outcomes: insecure, secure, or don't know. In order to prove a system is insecure, one must find just one example of a security hole. Finding the hole is usually difficult and typically requires substantial expertise, but once one hole is found it is clear that the system is insecure. In contrast, to prove that a system is secure, one has to show that there is *no* security hole *at all*. Because the latter is so difficult, the typical outcome is "we don't know of any remaining security holes, but we are certain that there are some."

Another way of appreciating the difficulty of achieving a negative goal is to model a computer system as a state machine with states for all the possible configurations in which the system can be and with links between states for transitions between configurations. As shown in Figure 11.1, the possible states and links form a graph, with the

* The woodpecker was believed to be extinct, but in 2005 a few scientists claimed to have found the bird in Arkansas after a kayaker caught a glimpse in 2004; if true, it is the first confirmed sighting in 60 years.

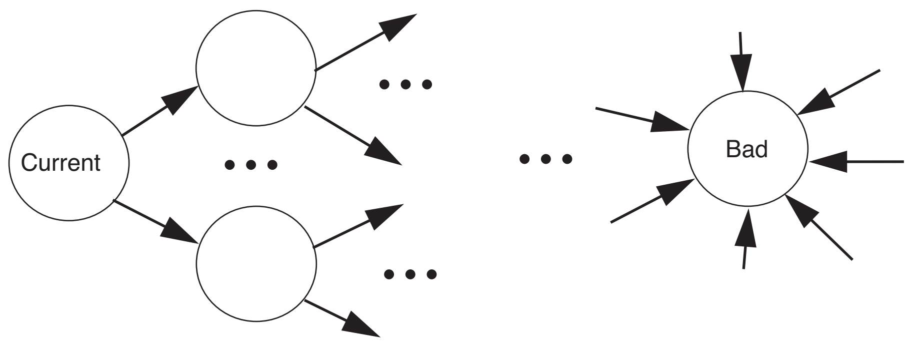


FIGURE 11.1

Modeling a computer systems as a state machine. An adversary's goal is to get the system into a state, labeled "Bad", that gives the adversary unauthorized access. To prevent the adversary from succeeding, *all* paths leading to the bad state must be blocked off because the adversary needs to find only *one* path to succeed.

states as nodes and possible transitions as edges. Assume that the system is in some current state. The goal of an adversary is to force the system from the current state to a state, labeled "Bad" in the figure, that gives the adversary unauthorized access. To defend against the adversary, the security designers must identify and block *every* path that leads to the bad state. But the adversary needs to find only *one* path from the current state to the bad state.

11.1.3 The Safety Net Approach

To successfully design systems that satisfy negative goals, this chapter adopts the safety net approach of Chapter 8[on-line], which in essence guides a designer to be paranoid—never assume the design is right. In the context of security, the two safety net principles *be explicit* and *design for iteration* reinforce this paranoid attitude:

1. Be explicit: Make all assumptions explicit so that they can be reviewed. It may require only *one* hole in the security of the system to penetrate it. The designer must therefore consider any threat that has security implications and make explicit the assumption on which the security design relies. Furthermore, make sure that all assumptions on which the security of the system is based are apparent at all times to all participants. For example, in the context of protocols, the meaning of each message should depend only on the content of the message itself, and should not be dependent on the context of the conversation. If the content of a message depends on its context, an adversary might be able to break the security of a protocol by tricking a receiver into interpreting the message in a different context.

2. Design for iteration: Assume you will make errors. Because the designer must assume that the design itself will contain flaws, the designer must be prepared to iterate the design. When a security hole is discovered, the designer must review the assumptions, if necessary adjust them, and repair the design. When a designer discovers an error in the system, the designer must reiterate the whole design and implementation process.

The safety net approach implies several requirements for the design of a secure system:

- Certify the security of the system. *Certification* involves verifying that the design matches the intended security policy, the implementation matches the design, and the running system matches the implementation, followed up by end-to-end tests by security specialists looking for errors that might compromise security. Certification provides a systematic approach to reviewing the security of a system against the assumptions. Ideally, certification is performed by independent reviewers, and, if possible, using formal tools. One way to make certification manageable is to identify those components that must be trusted to ensure security, minimize their number, and build a wall around them. Section 11.1.7 discusses this idea, known as the trusted computing base, in more detail.
- Maintain audit trails of all authorization decisions. Since the designer must assume that legitimate users might abuse their permissions or an adversary may be masquerading as a legitimate user, the system should maintain an tamper-proof log (so that an adversary cannot erase records) of all authorization decisions made. If, despite all security mechanisms, an adversary (either from the inside or from the outside) succeeds in breaking the security of the system, the log might help in forensics. A forensics expert may be able to use the log to collect evidence that stands in court and help establish the identity of the adversary so that the adversary can be prosecuted after the fact. The log also can be used as a source of feedback that reveals an incorrect assumption, design, or implementation.
- Design the system for feedback. An adversary is unlikely to provide feedback when compromising the system, so it is up to the designer to create ways to obtain feedback. Obtaining feedback starts with stating the assumptions explicitly, so the designer can check the designed, implemented, and operational system against the assumptions when a flaw is identified. This method by itself doesn't identify security weaknesses, and thus the designer must actively look for potential problems. Methods include reviewing audit logs and running programs that alert system administrators about unexpected behavior, such as unusual network traffic (e.g., many requests to a machine that normally doesn't receive many requests), repeated login failures, etc. The designer should also create an environment in which staff and customers are not blamed for system compromises, but instead are rewarded for reporting them, so that they are encouraged to report problems

instead of hiding them. Designing for feedback reduces the chance that security holes will slip by unnoticed. Anderson illustrates well through a number of real-world examples how important it is to design for feedback [Suggestions for Further Reading 11.5.3].

As part of the safety net approach, a designer must consider the environment in which the system runs. The designer must secure all communication links (e.g., dial-up modem lines that would otherwise bypass the firewall that filters traffic between a private network and a public network), prepare for malfunctioning equipment, find and remove back doors that create security problems, provide configuration settings for users that are secure by default, and determine who is trustworthy enough to own a key to the room that protects the most secure part of the system. Moreover, the designer must protect against bribes and worry about disgruntled employees. The security literature is filled with stories of failures because the designers didn't take one of these issues into account.

As another part of the safety net approach, the designer must consider the *dynamics of use*. This term refers to how one establishes and changes the specification of who may obtain access to what. For example, Alice might revoke Bob's permission to read file "x." To gain some insight into the complexity introduced by changes to access authorization, consider again the question, "Is there any way that Lucifer could obtain access to file x?" One should check not only whether Lucifer has access to file x, but also whether Lucifer may change the specification of file x's accessibility. The next step is to see if Lucifer can change the specification of who may change the specification of file x's accessibility, etc.

Another problem of dynamics arises when the owner revokes a user's access to a file while that file is being used. Letting the previously authorized user continue until the user is "finished" with the information may be unacceptable if the owner has suddenly realized that the file contains sensitive data. On the other hand, immediate withdrawal of authorization may severely disrupt the user or leave inconsistent data if the user was in the middle of an atomic action. Provisions for the dynamics of use are at least as important as those for static specification of security.

Finally, the safety net approach suggests that a designer should never believe that a system is completely secure. Instead, one must design systems that *defend in depth* by using redundant defenses, a strategy that the Russian army deployed successfully for centuries to defend Russia. For example, a designer might have designed a system that provides end-to-end security over untrusted networks. In addition, the designer might also include a firewall between the trusted and untrusted network for network-level security. The firewall is in principle completely redundant with the end-to-end security mechanisms; if the end-to-end security mechanism works correctly, there is no need for network-level security. For an adversary to break the security of the system, however, the adversary has to find flaws in both the firewall *and* in the end-to-end security mechanisms, and be lucky enough that the first flaw allows exploitation of the second.

The defense-in-depth design strategy offers no guarantees, but it seems to be effective in practice. The reason is that conceptually the defense-in-depth strategy cuts more edges in the graph of all possible paths from a current state to some undesired state. As a result, an adversary has fewer paths available to get to and exploit the undesired state.

11.1.4 Design Principles

In practice, because security is a negative goal, producing a system that actually does prevent all unauthorized acts has proved to be extremely difficult. Penetration exercises involving many different systems all have shown that users can obtain unauthorized access to these systems. Even if designers follow the safety net approach carefully, design and implementation flaws provide paths that circumvent the intended access constraints. In addition, because computer systems change rapidly or are deployed in new environments for which they were not designed originally, new opportunities for security compromises come about. Section 11.11 provides several war stories about security breaches.

Design and construction techniques that systematically exclude flaws are the topic of much research activity, but no complete method applicable to the design of computer systems exists yet. This difficulty is related to the negative quality of the requirement to prevent all unauthorized actions. In the absence of such methodical techniques, experience has provided several security principles to guide the design towards minimizing the number of security flaws in an implementation. We discuss these principles next.

The design should not be secret:

Open design principle

Let anyone comment on the design. You need all the help you can get.

Violation of the open design principle has historically proven to almost always lead to flawed designs. The mechanisms should not depend on the ignorance of potential adversaries, but rather on the possession of specific, more easily protected, secret keys or passwords. This decoupling of security mechanisms from security keys permits the mechanisms to be examined by many reviewers without concern that the review may itself compromise the safeguards. In addition, any skeptical user must be able to review that the system is adequate for the user's purpose. Finally, it is simply not realistic to maintain secrecy of any system that receives wide distribution. However, the open design principle can conflict with other goals, which has led to numerous debates; Sidebar 11.2 summarizes some of the arguments.

The right people must perform the review because spotting security holes is difficult. Even if the design and implementation are public, that is an insufficient condition for spotting security problems. For example, standard committees are usually open in principle but their openness sometimes has barriers that cause the proposed standard not to be reviewed by the right people. To participate in the design of the WiFi Wired Equivalent Privacy standard required committee members to pay a substantial fee, which apparently discouraged security researchers from participating. When the standard was

Sidebar 11.2: Should designs and vulnerabilities be public? The debate of closed versus open designs has been raging literally for ages, and is not unique to computer security. The advocates of closed designs argue that making designs public helps the adversaries, so why do it? The advocates of open designs argue that closed designs don't really provide security because in the long run it is impossible to keep a design secret. The practical result of attempted secrecy is usually that the bad guys know about the flaws but the good guys don't. Open design advocates disparage closed designs by describing them as "security through obscurity".

On the other hand, the open design principle can conflict with the desire to keep a design and its implementation proprietary for commercial or national security reasons. For example, software companies often do not want a competitor to review their software in fear that the competitor can easily learn or copy ideas. Many companies attempt to resolve this conflict by arranging reviews, but restricting who can participate in the reviews. This approach has the danger that not the right people are performing the reviews.

Closely related to the question whether designs should be public or not is the question whether vulnerabilities should be made public or not? Again, the debate about the right answer to this question has been raging for ages, and is perhaps best illustrated by the following quote from a 1853 book* about old-fashioned door locks:

> A commercial, and in some respects a social doubt has been started within the last year or two, whether or not it is right to discuss so openly the security or insecurity of locks. Many well-meaning persons suppose that the discussion respecting the means for baffling the supposed safety of locks offers a premium for dishonesty, by showing others how to be dishonest. This is a fallacy. Rogues are very keen in their profession, and know already much more than we can teach them respecting their several kinds of roguery.
>
> Rogues knew a good deal about lock-picking long before locksmiths discussed it among themselves, as they have lately done. If a lock, let it have been made in whatever country, or by whatever maker, is not so inviolable as it has hitherto been deemed to be, surely it is to the interest of honest persons to know this fact, because the dishonest are tolerably certain to apply the knowledge practically; and the spread of the knowledge is necessary to give fair play to those who might suffer by ignorance.
>
> It cannot be too earnestly urged that an acquaintance with real facts will, in the end, be better for all parties.

Computer security experts generally believe that one should publish vulnerabilities for the reasons stated by Hobbs and that users should know if the system they are using has a problem so they can decide whether or not they care. Companies, however, are typically reluctant to disclose vulnerabilities. For example, a bank has little incentive to advertise successful compromises because it may scare away customers.

(*sidebar continues*)

* A.C Hobbs (Charles Tomlinson, ed.), *Locks and Safes: The Construction of Locks*. Virtue & Co., London, 1853 (revised 1868).

To handle this tension, many governments have created laws and organizations that make vulnerabilities public. In California companies must inform their customers if an adversary might have succeeded in stealing customer priviate information (e.g., a social security number). The U.S federal government has created the Computer Emergency Response Team (CERT) to document vulnerabilities in software systems and help with the response to these vulnerabilities (see www.cert.org). When CERT learns about a new vunerability, it first notifies the vendor, then it waits for some time for the vendor to develop a patch, and then goes public with the vulnerability and the patch.

finalized and security researchers began to examine the standard, they immediately found several problems, one of which is described on page 11–51.

Since it is difficult to keep a secret:

Minimize secrets

Because they probably won't remain secret for long.

Following this principle has the following additional advantage. If the secret is comprised, it must be replaced; if the secret is minimal, then replacing the secret is easier.

An open design that minimizes secrets doesn't provide security itself. The primary underpinning of the security of a system is, as was mentioned on page 11–5, the principle of *complete mediation*.This principle forces every access to be explicitly authenticated and authorized, including ones for initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of verifying the authenticity of the origin and data of every request must be devised. This principle applies to a service mediating requests, as well as to a kernel mediating supervisor calls and a virtual memory manager mediating a read request for a byte in memory. This principle also implies that proposals for caching results of an authority check should be examined skeptically; if a change in authority occurs, cached results must be updated.

The human engineering *principle of least astonishment* applies especially to mediation. The mechanism for authorization should be transparent enough to a user that the user has a good intuitive understanding of how the security goals map to the provided security mechanism. It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the security mechanisms correctly. For example, a system should provide intuitive, default settings for security mechanisms so that only the appropriate operations are authorized. If a system administrator or user must first configure or jump through hoops to use a security mechanism, the user won't use it. Also, to the extent that the user's mental image of security goals matches the security mechanisms, mistakes will be minimized. If a user must translate intuitive security objectives into a radically different specification language, errors are inevitable. Ideally, security mechanisms should make a user's computer experience better instead of worse.

Another widely applicable principle, *adopt sweeping simplifications*, also applies to security. The fewer mechanisms that must be right to ensure protection, the more likely the design will be correct:

Economy of mechanism

The less there is, the more likely you will get it right.

Designing a secure system is difficult because every access path must be considered to ensure complete mediation, including ones that are not exercised during normal operation. As a result, techniques such as line-by-line inspection of software and physical examination of hardware implementing security mechanisms may be necessary. For such techniques to be successful, a small and simple design is essential.

Reducing the number of mechanisms necessary helps with verifying the security of a computer system. For the ones remaining, it would be ideal if only a few are common to more than one user and depended on by all users because every shared mechanism might provide unintended communication paths between users. Further, any mechanism serving all users must be certified to the satisfaction of every user, a job presumably harder than satisfying only one or a few users. These observations lead to the following security principle:

Minimize common mechanism

Shared mechanisms provide unwanted communication paths.

This principle helps reduce the number of unintended communication paths and reduces the amount of hardware and software on which all users depend, thus making it easier to verify if there are any undesirable security implications. For example, given the choice of implementing a new function as a kernel procedure shared by all users or as a library procedure that can be handled as though it were the user's own, choose the latter course. Then, if one or a few users are not satisfied with the level of certification of the function, they can provide a substitute or not use it at all. Either way, they can avoid being harmed by a mistake in it. This principle is an *end-to-end argument*.

Complete mediation requires that every request be checked for authorization and only authorized requests be approved. It is important that requests are not authorized accidently. The following security principle helps reduce such mistakes:

Fail-safe defaults

Most users won't change them, so make sure that defaults do something safe.

Access decisions should be based on permission rather than exclusion. This principle means that lack of access should be the default, and the security scheme lists conditions under which access is permitted. This approach exhibits a better failure mode than the alternative approach, where the default is to permit access. A design or implementation mistake in a mechanism that gives explicit permission tends to fail by refusing permission, a safe situation that can be quickly detected. On the other hand, a design or implementation mistake in a mechanism that explicitly excludes access tends to fail by allowing access, a failure that may long go unnoticed in normal use.

To ensure that complete mediation and fail-safe defaults work well in practice, it is important that programs and users have privileges only when necessary. For example, system programs or administrators who have special privileges should have those privileges only when necessary; when they are doing ordinary activities the privileges should be withdrawn. Leaving them in place just opens the door to accidents. These observations suggest the following security principle:

Least privilege principle

Don't store lunch in the safe with the jewels.

This principle limits the damage that can result from an accident or an error. Also, if fewer programs have special privileges, less code must be audited to verify the security of a system. The military security rule of "need-to-know" is an example of this principle.

Security experts sometimes use alternative formulations that combine aspects of several principles. For example, the formulation "minimize the attack surface" combines aspects of economy of mechanism (a narrow interface with a simple implementation provides fewer opportunities for designer mistakes and thus provides fewer attack possibilities), minimize secrets (few opportunies to crack secrets), least privilege (run most code with few privileges so that a successful attack does little harm), and minimize common mechanism (reduce the number of opportunities of unintended communication paths).

11.1.5 A High d(technology)/dt Poses Challenges For Security

Much software on the Internet and on personal computers fails to follow these principles, even though most of these principles were understood and articulated in the 1970s, before personal computers and the Internet came into existence. The reasons why they weren't followed are different for the Internet and personal computers, but they illustrate how difficult it is to achieve security when the rate of innovation is high.

When the Internet was first deployed, software implementations of the cryptographic techniques necessary to authenticate and protect messages (see Section 11.2 and Section 11.1) were considered but would have increased latency to unacceptable levels. Hardware implementations of cryptographic operations at that time were too expensive, and not exportable because the US government enforced rules to limit the use of cryptography. Since the Internet was originally used primarily by academics—a mostly cooperative community—the resulting lack of security was initially not a serious defect.

In 1994 the Internet was opened to commercial activities. Electronic stores came into existence, and many more computers storing valuable information came on-line. This development attracted many more adversaries. Suddenly, the designers of the Internet were forced to provide security. Because security was not part of the initial design plan, security mechanisms today have been designed as after-the-fact additions and have been provided in an *ad-hoc* fashion instead of following an overall plan based on established security principles.

For different historical reasons, most personal computers came with little internal security and only limited stabs at network security. Yet today personal computers are almost always attached to networks where they are vulnerable. Originally, personal computers were designed as stand-alone devices to be used by a single person (that's why they are called *personal* computers). To keep the cost low, they had essentially no security mechanisms, but because they were used stand-alone, the situation was acceptable. With the arrival of the Internet, the desire to get on-line exposed their previously benign security problems. Furthermore, because of rapid improvements in technology, personal computers are now the primary platform for all kinds of computing, including most business-related computing. Because personal computers now store valuable information, are attached to networks, and have minimal protection, personal computers have become a prime target for adversaries.

The designers of the personal computer didn't originally foresee that network access would quickly become a universal requirement. When they later did respond to security concerns, the designers tried to add security mechanism quickly. Just getting the hardware mechanisms right, however, took multiple iterations, both because of blunders and because they were after-the-fact add-ons. Today, designers are still trying to figure out how to retrofit the existing personal-computer software and to configure the default settings right for improved security, while they are also being hit with requirements for improved security to handle denial-of-service attacks, phishing attacks*, viruses, worms, malware, and adversaries who try to take over machines without being noticed to create botnets (see Sidebar 11.3). As a consequence, there are many *ad hoc* mechanisms found in the field that don't follow the models or principles suggested in this chapter.

11.1.6 Security Model

Although there are many ways to compromise the security of a system, the conceptual model to secure a system is surprisingly simple. To be secure, a system requires *complete mediation*: the system must mediate every action requested, including ones to configure and manage the system. The basic security plan then is that for each requested action the

* Jargon term for an attack in which an adversary lures a victim to Web site controlled by the adversary; for an example see Suggestions for Further Reading 11.6.6.

Sidebar 11.3: Malware: viruses, worms, trojan horses, logic bombs, bots, etc. There is a community of programmers that produces *malware*, software designed to run on a computer without the computer owner's intent. Some malware is created as a practical joke, other malware is designed to make money or to sabotage someone; Hafner and Markoff profile a few early high-profile cases of computer break-ins and the perpetrator's motivation [Suggestions for Further Reading 1.3.5]. More recently, there is an industry in creating malware that silently turns a user's computer into a *bot*, a computer controlled by an adversary, which is then used by the adversary to send unsolicited e-mail (SPAM) on behalf of paying customers, which generates a revenue stream for the adversary [Suggestions for Further Reading 11.6.5].*

Malware uses a combinations of techniques to take control of a user's computer. These techniques include ways to install malware on a user's computer, ways to arrange that the malware will run on the user's computer, ways to replicate the malware on other computers, and ways to do perfidious things. Some of the techniques rely on users naïvety while others rely on innovative ideas to exploit errors in the software running on the user's computer. As an example of both, in 2000 an adversary constructed the "ILOVEYOU" *virus*, an e-mail message with a malicious executable attachment. The adversary sent the e-mail to a few recipients. When a recipient opened the executable e-mail (attracted by "ILOVEYOU" in the e-mail's subject), the malicious attachment read the recipient's address book, and sent itself to the users in the address book. So many users opened the e-mail that it spread rapidly and overwhelmed e-mail servers at many institutions.

The Morris *worm* [Suggestions for Further Reading 11.6.1], created in 1984, is an example of malware that relies only on clever ways to exploit errors in software. The worm exploited various weaknesses in remote computers, among them a buffer overrun (see Sidebar 11.4) in an e-mail server (sendmail) running on the UNIX operating system, which allowed it to install and run itself on the compromised computer. There it looked for network addresses of computers in configuration files, and then penetrated those computers, and so on. According to its creator it was not intended to create damage but a design error caused it to effectively create a denial-of-service attack. The worm spread so rapidly, infecting some computers multiple times, that it effectively shut down parts of the Internet.

The popular jargon attaches colorful labels to describe different types of malware such as virus, worm, trojan horse, logic bomb, drive-by download, etc., and new ones appear as new types of malware show up. These labels don't correspond to precise, orthogonal technical concepts, but combine various malware features in different ways. All of them, however, exploit some weakness in the security of a computer, and the techniques described in this chapter are also relevant in containing malware.

* Problem set *47* explores a potential stamp-based solution.

agent requesting the operation proves its identity to the system and then the system decides if the agent is allowed to perform that operation.

This simple model covers a wide range of instances of systems. For example, the agent may be a client in a client/service application, in which case the request is in the form of a message to a service. For another example, the agent may be a thread referring to virtual memory, in which case the request is in the form of a LOAD or STORE to a named memory cell. In each of these cases, the system must establish the identity of the agent and decide whether to perform the request or not. If all requests are mediated correctly, then the job of the adversary becomes much harder. The adversary must compromise the mediation system, launch an insider attack, or is limited to denial-of-service attacks.

The rest of this section works out the mediation model in more detail, and illustrates it with various examples. Of course a simple conceptual model cannot cover all attacks and all details. And, unfortunately, in security, the devil is often in the details of the implementation: does the system to be secure implement the model for all its operations and is the implementation correct? Nevertheless, the model is helpful in framing many security problems and then addressing them.

Agents perform on behalf of some entity that corresponds to a person outside the computer system; we call the representation of such an entity inside the computer system a *principal*. The principal is the unit of authorization in a computer system, and therefore also the unit of accountability and responsibility. Using these terms, mediating an action is asking the question, "Is the principal who requested the action authorized to perform the action?"

The basic approach to mediating every requested action is to ensure that there is really only *one* way to request an action. Conceptually, we want to build a wall around the system with one small opening through which all requested actions pass. Then, for every requested action, the system must answer "Should I perform the action?". To do so a system is typically decomposed in two parts: one part, called a *guard*, that specializes in deciding the answer to the question and a second part that performs the action. (In the literature, a guard that provides complete mediation is usually called a *reference monitor*.)

The guard can clarify the question, "Is the principal who originated the requested action allowed to perform the action?" by obtaining answers to the three subquestions of complete mediation (see Figure 11.2). The guard verifies that the message containing the request is authentic (i.e., the request hasn't been modified and that the principal is indeed the source of the request), and that the principal is permitted to perform the requested action on the object (authorization). If so, the guard allows the action; otherwise, it denies the request. The guard also logs all decisions for later audits

The first two (has the request been modified and what is the source of the request) of the three mediation questions fall in the province of *authentication* of the request. Using an authentication service the guard verifies the identity of the principal. Using additional information, sometimes part of the request but sometimes communicated separately, the guard verifies the integrity of the request. After answering the authenticity questions, the guard knows who the principal associated with the request is and that no adversary has modified the request.

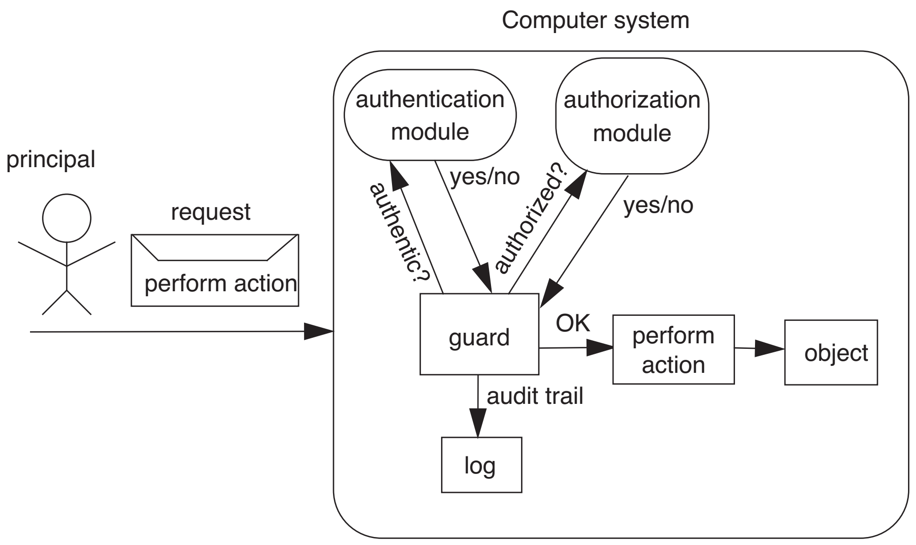


FIGURE 11.2

The security model based on complete mediation. The authenticity question includes both verifying the integrity and the source of the request.

The third, and final, question falls in the province of *authorization*. An authorization service allows principals to specify which objects they share with whom. Once the guard has securely established the identity of the principal associated with the request using the authentication service, the guard verifies with the authorization service that the principal has the appropriate authorization, and, if so, allows the requested service to perform the requested action.

The guard approach of complete mediation applies broadly to computer systems. Whether the messages are Web requests for an Internet store, LOAD and STORE operations to memory, or supervisor calls for the kernel, in all cases the same three questions must be answered by the Web service, virtual memory manager, or kernel, respectively. The implementation of the mechanisms for mediation, however, might be quite different for each case.

Consider an on-line newspaper. The newspaper service may restrict certain articles to paying subscribers and therefore must authenticate users and authorize requests, which often work as follows. The Web browser sends requests on behalf of an Internet user to the newspaper's Web server. The guard uses the principal's subscriber number and an authenticator (e.g., a password) included in the requests to authenticate the principal associated with the requests. If the principal is a legitimate subscriber and has authorization to read the requested article, the guard allows the request and the server replies with the article. Because the Internet is untrusted, the communications between the Web browser and the server must be protected; otherwise, an adversary can, for example, obtain the subscriber's password. Using *cryptography* one can create a *secure channel* that protects the communications over an untrusted network. Cryptography is a branch of computer science that designs primitives such as ciphers, pseudorandom number generators, and hashes, which can be used to protect messages against a wide range of attacks.

As another example, consider a virtual memory system with one domain per thread. In this case, the processor issues LOAD and STORE instructions on behalf of a thread to a virtual memory manager, which checks if the addresses in the instructions fall in the thread's domain. Conceptually, the processor sends a message across a bus, containing the operation (LOAD or STORE) and the requested address. This message is accompanied with a principal identifier naming the thread. If the bus is a trusted communication link, then the message doesn't have to be protected. If the bus isn't a secure channel (e.g., a digital rights management application may want to protect against an owner snooping on the bus to steal the copyrighted content), then the message between the processor and memory might be protected using cryptographic techniques. The virtual memory manager plays the role of a guard. It uses the thread identifier to verify if the address falls in the thread's domain and if the thread is authorized to perform the operation. If so, the guard allows the requested operation, and virtual memory manager replies by reading and writing the requested memory location.

Even if the mechanisms for complete mediation are implemented perfectly (i.e., there are no design and implementation errors in the cryptography, password checker, the virtual memory manager, the kernel, etc.), a system may still leave opportunities for an adversary to break the security of the system. The adversary may be able to circumvent the guard, or launch an insider attack, or overload the system with requests for actions, thus delaying or even denying legitimate principals access. A designer must be prepared for these cases—an example of the paranoid design attitude. We discusses these cases in more detail.

To circumvent the guard, the adversary might create or find another opening in the system. A simple opening for an adversary might be a dial-up modem line that is not mediated. If the adversary finds the phone number (and perhaps the password to dial in), the adversary can gain control over the service. A more sophisticated way to create an opening is a *buffer overrun attack* on services written in the C programming language (see Sidebar 11.4), which causes the service to execute a program under the control of the adversary, which then creates an interface for the adversary that is not checked by the system.

As examples of insider attacks, the adversary may be able to guess a principal's password, may be able to bribe a principal to act on the adversary's behalf, or may be able to trick the principal to run the adversary's program on the principal's computer with the principal's privileges (e.g., the principal opens an executable e-mail attachment sent by the adversary). Or, the adversary may be a legitimate principal who is disgruntled.

Measures against badly behaving principals are also the final line of defense against adversaries who successfully break the security of the system, thus appearing to be legitimate users. The measures include (1) running every requested operation with the least privilege because that minimizes damage that a legitimate principal can do, (2) maintain-

Sidebar 11.4: Why are buffer overrun bugs so common? It has become disappointingly common to hear a news report that a new Internet worm is rapidly spreading, and a little research on the World-Wide Web usually turns up as one detail that the worm exploits a *buffer overrun* bug. The reason that buffer overrun bugs are so common is that some widely used programming languages (in particular, C and C++) do not routinely check array bounds. When those languages are used, array bounds checking must be explicitly provided by the programmer. The reason that buffer overrun bugs are so easily exploited arises from an unintentional conspiracy of common system design and implementation practices that allow a buffer overrun to modify critical memory cells.

1. Compilers usually allocate space to store arrays as contiguous memory cells, with the first element at some starting address and successive elements at higher-numbered addresses.

2. Since there usually isn't any hardware support for doing anything different, most operating systems allocate a single, contiguous block of address space for a program and its data. The addresses may be either physical or virtual, but the important thing is that the programming environment is a single, contiguous block of memory addresses.

3. Faced with this single block of memory, programming support systems typically suballocate the address block into three regions: They place the program code in low-numbered addresses, they place static storage (the heap) just above those low-numbered addresses, and they start the stack at the highest-numbered address and grow it down, using lower addresses, toward the heap.

These three design practices, when combined with lack of automatic bounds checking, set the stage for exploitation. For example, historically it has been common for programs written in the C language to use library programs such as

```
GETS (character array reference string_buffer)
```

rather than a more elaborate version of the same program

```
FGETS (character array reference string_buffer, integer string_length, file stream)
```

to move character string data from an incoming stream to a local array, identified by the memory address of *string_buffer*. The important difference is that GETS reads characters until it encounters a new-line character or end of file, while FGETS adds an additional stop condition: it stops after reading *string_length* characters, thus providing an explicit array bound check. Using GETS rather than FGETS is an example of Gabriel’s *Worse is Better*: “it is slightly better to be simple than to be correct." [Suggestions for Further Reading 1.5.1]

A program that is listening on some Internet port for incoming messages allocates a *string_buffer* of size 30 characters, to hold a field from the message, knowing that that field should never be larger. It copies data of the message from the port into *string_buffer*, using GETS An adversary prepares and sends a message in which that field contains a string of, say, 250 characters. GETS overruns *string_buffer*.

(*Sidebar continues*)

Because of the compiler practice of placing successive array elements of *string_buffer* in higher-numbered addresses, if the program placed *string_buffer* in the stack the overrun overwrites cells in the stack that have higher-numbered addresses. But because the stack grows toward lower-numbered addresses, the cells overwritten by the buffer overrun are all *older* variables, allocated before *string_buffer*. Typically, an important older variable is the one that holds the return point of the currently running procedure. So the return point is vulnerable. A common exploit is thus to include runnable code in the 250-character string and, knowing stack offsets, smash the return point stack variable to contain the address of that code. Then, when the thread returns from the current procedure, it unwittingly transfers control to the adversary's code.

By now, many such simple vulnerabilities have been discovered and fixed. But exploiting buffer overruns is not limited to smashing return points in the stack. Any writable variable that contains a jump address and that is located adjacent to a buffer in the stack or the heap may be vulnerable to an overrun of that buffer. The next time that the running thread uses that jump address, the adversary gains control of that thread. The adversary may not even have to supply executable code if he or she can cause the jump to go to some existing code such as a library routine that, with a suitable argument value, can be made to do something bad [Suggestions for Further Reading 11.6.2]. Such attacks require detailed knowledge of the layout and code generation methods used by the compiler on the system being attacked, but adversaries can readily discover that information by examining their own systems at leisure. Problem set *49* explores some of these attacks.

From that discussion one can draw several lessons that invoke security design principles:

1. The root cause of buffer overruns is the use of programming languages that do not provide the *fail-safe default* of automatically checking all array references to verify that they do not exceed the space allocated for the array.

2. Be explicit. One can interpret the problem with GETS to be that it relies on its context, rather than the program, to tell it exactly what to do. When the context contains contradictions (a string of one size, a buffer of another size) or ambiguities, the library routine may resolve them in an unexpected way. There is a trade-off between convenience and explicitness in programming languages. When security is the goal, a programming language that requires that the programmer be explicit is probably safer.

3. Hardware architecture features can help minimize the impact of common programming errors, and thus make it harder for an adversary to exploit them. Consider, for example, an architecture that provides distinct, hardware-enforced memory segments as described in Section 5.4.5, using one segment for program code, a second segment for the heap, and a third segment for the stack. Since different segments can have different read, write, and execute permissions, the stack and heap segments might disallow executable instructions, while the program area disallows writing. The *principle of least privilege* suggests that no region of memory should be simultaneously writable and executable. If all buffers are in segments that

(*Sidebar continues*)

are not executable, an adversary would find it more difficult to deposit code in the execution environment. Instead, the adversary may have to resort to methods that exploit code already in that execution environment. Even better might be to place each buffer in a separate segment, thus using the hardware to check array bounds.

Hardware for Multics [Suggestions for Further Reading 3.1.4 and 5.4.1], a system implemented in the 1960s, provided segments. The Multics kernel followed the principle of least privilege in setting up permissions, and the observed result was that addressing errors were virtually always caught by the hardware at the instant they occurred, rather than leading to a later system meltdown. Designers of currently common hardware platforms have recently modified the memory management unit of these platforms to provide similar features, and today's popular operating systems are using the features to provide better protection.

4. Storing a jump address in the midst of writable data is hazardous because it is hard to protect it against either programming errors or intentional attacks. If an adversary can control the value of a jump address, there is likely to be some way that the adversary can exploit it to gain control of the thread. *Complete mediation* suggests that all such jump values should be validated before being used. Designers have devised schemes to try to provide at least partial validation. An example of such a scheme is to store an unpredictable nonce value (a "canary") adjacent to the memory cell that holds the jump address and, before using the jump address, verify that the canary is intact by comparing it with a copy stored elsewhere. Many similar schemes have been devised, but it is hard to devise one that is foolproof. For the most part these schemes do not prevent exploits, they just make the adversary's job harder.

ing an audit trail, of the mediation decisions made for *every* operation, (3) making copies and archiving data in secure places, and (4) periodically manually reviewing which principals should continue to have access and with what privileges. Of course, the archived data and the audit trail must be maintained securely; an adversary must not be able to modify the archived data or the audit trail. Measures to secure archives and audit trails include designing them to be write once and append-only.

The archives and the audit trail can be used to recover from a security breach. If an inspection of the service reveals that something bad has happened, the archived copies can be used to restore the data. The audit trail may help in figuring out what happened (e.g., what data has been damaged) and which principal did it. As mentioned earlier, the audit trail might also be useful as a proof in court to punish adversaries. These measures can be viewed as an example of defense in depth—if the first line of defense fails, one hopes that the next measure will help.

An adversary's goal may be just to deny service to other users. To achieve this goal an adversary could flood a communication link with requests that take enough time of the service that it is unavailable for other users. The challenge in handling a denial-of-service attack is that the messages sent by the adversary may be legitimate requests and the adversary may use many computers to send these legitimate requests (see Suggestions for Further Reading 11.6.4 for an example). There is no single technique that can address denial-of-service attacks. Solutions typically involve several ideas: audit messages to be able to detect and filter bad traffic before it reaches the service, careful design of services to control the resources dedicated to a request and to push work back to the clients, and replicating services (see Section 10.3[on-line]) to keep the service available during an attack. By replicating the service, an adversary must flood multiple replicas to make the service unavailable. This attack may require so many messages that with careful analysis of audit trails it becomes possible to track down the adversary.

11.1.7 Trusted Computing Base

Implementing the security model of Section 11.1.6 is a negative goal, and therefore difficult. There are no methods to verify correctness of an implementation that is claimed to achieve a negative goal. So, how do we proceed? The basic idea is to minimize the number of mechanisms that need to be correct in order for the system to be secure—*the economy of mechanism principle*, and to follow the safety net approach (*be explicit* and *design for iteration)*.

When designing a secure system, we organize the system into two kinds of modules: *untrusted* modules and *trusted* modules. The correctness of the untrusted modules does not affect the security of the whole system. The trusted modules are the part that must work correctly to make the system secure. Ideally, we want the trusted modules to be usable by other untrusted modules, so that the designer of a new module doesn't have to worry about getting the trusted modules right. The collection of trusted modules is usually called the *trusted computing base* (TCB).

Establishing whether or not a module is part of the TCB can be difficult. Looking at an individual module, there isn't any simple procedure to decide whether or not the system's security depends on the correct operation of that module. For example, in UNIX if a module runs on behalf of the superuser principal (see page 11–77), it is likely to be part of the TCB because if the adversary compromises the module, the adversary has full privileges. If the same module runs on behalf of a regular principal, it is often not part of the trusted computing base because it cannot perform privileged operations. But even then the module could be part of the TCB; it may be part of a user-level service (e.g., a Web service) that makes decisions about which clients have access. An error in the module's code may allow an adversary to obtain unauthorized access.

Lacking a systematic decision procedure for deciding if a module is in the TCB, the decision is difficult to make and easy to get wrong, yet a good division is important. A bad division between trusted and untrusted modules may result in a large and complex TCB, making it difficult to reason about the security of the system. If the TCB is large, it also means that ordinary users can make only few changes because ordinary users should only change modules outside the TCB that don't impact security. If ordinary users can change the system in only limited ways, it may make it difficult for them to get their job done in an effective way and result in bad user experiences. A large TCB also means that much of the system can be modified by only trusted principals, limiting the rate at which the system can evolve. The design principles of Section 11.1.4 can guide this part of the design process, but typically the division must be worked out by security experts.

Once the split has been worked out, the challenge becomes one of designing and implementing a TCB. To be successful at this challenge, we want to work in a way that maximizes the chance that the design and implementation of the TCB are correct. To do so, we want to minimize the chance of errors and maximize the rate of discovery of errors. To achieve the first goal, we should minimize the size of the TCB. To achieve the second goal, the design process should include feedback so that we will find errors quickly.

The following method shows how to build such a TCB:

- Specify security requirements for the TCB (e.g., secure communication over untrusted networks). The main reason for this step is to *explicitly* specify assumptions so that we can decide if the assumptions are credible. As part of the requirements, one also specifies the attacks against which the TCB is protected so that the security risks are assessable. By specifying what the TCB does and does not do, we know against which kinds of attacks we are protected and to which kinds we are vulnerable.
- Design a minimal TCB. Use good tools (such as authentication logic, which we will discuss in Section 11.5) to express the design.
- Implement the TCB. It is again important to use good tools. For example, buffer-overrun attacks can be avoided by using a language that checks array bounds.
- Run the TCB and try to break the security.

The hard part in this multistep design method is verifying that the steps are consistent: verifying that the design meets the specification, verifying that the design is resistant to the specified attacks, verifying that the implementation matches the design, and verifying that the system running in the computer is the one that was actually implemented. For example, as Thompson has demonstrated, it is easy for an adversary with compiler expertise to insert a Trojan Horses into a system that is difficult to detect [Suggestions for Further Reading 11.3.3 and 11.3.4].

The problem in computer security is typically *not* one of inventing clever mechanisms and architectures, but rather one of ensuring that the installed system actually meets the design and implementation. Performing such an end-to-end check is difficult. For example, it is common to hire *a tiger team* whose mission is to find loopholes that could be exploited to break the security of the system. The tiger team may be able to find some loopholes, but, unfortunately, cannot provide a guarantee that all loopholes have been found.

The design method also implies that when a bug is detected and repaired, the designer must review the assumptions to see which ones were wrong or missing, repair the assumptions, and repeat this process until sufficient confidence in the security of the system has been obtained. This approach flushes out any fuzzy thinking, makes the system more reliable, and slowly builds confidence that the system is correct.

The method also clearly states what risks were considered acceptable when the system was designed, because the prospective user must be able to look at the specification to evaluate whether the system meets the requirements. Stating what risks are acceptable is important because much of the design of secure systems is driven by economic constraints. Users may consider a security risk acceptable if the cost of a security failure is small compared to designing a system that negates the risk.

11.1.8 The Road Map for this Chapter

The rest of this chapter follows the security model of Figure 11.2. Section 11.2 presents techniques for authenticating principals. Section 11.2 explains how to authenticate messages by using a pair of procedures named SIGN and VERIFY. Section 11.4 explains how to keep messages confidential using a pair of procedures named ENCRYPT and DECRYPT. Section 11.5 explains how to set up, for example, an authenticated and secure communication link using security protocols. Section 11.6 discusses different designs for an authorization service. Because authentication is the foundation of security, Section 11.5 discusses how to reason about authenticating principals systematically. The actual implementation of SIGN, VERIFY, ENCRYPT, and DECRYPT we outsource to theoreticians specialized in cryptography, but a brief summary of how to implement SIGN, VERIFY, ENCRYPT, and DECRYPT is provided in Section 11.8. The case study in Section 11.10 provides a complete example of the techniques discussed in this chapter by describing how authentication and authorization is done in the World-Wide Web. Finally, Section 11.11 concludes the chapter with war stories of security failures, despite the best intentions of the designers; these stories emphasize how difficult it is to achieve a negative goal.

11.2 Authenticating Principals

Most security policies involve people. For example, a simple policy might say that only the owner of the file “x” should be able to read it. In this statement the owner corresponds to a human. To be able to support such a policy the file service must have a way of establishing a secure binding between a user of the service and the origin of a request. Establishing and verifying the binding are topics that fall in the province of authentication.

Returning to our security model, the setup for authentication can be presented pictorially as in Figure 11.3. A person (Alice) asks her client computer to send a message “Buy 100 shares of Generic Moneymaking, Inc.” to her favorite electronic trading service. An adversary may be able to copy the message, delete it, modify it, or replace it. As explained in Section 11.1, when Alice’s trading service receives this message, the guard must establish two important facts related to authenticity:

1. Who is this principal making the request? The guard must establish if the message indeed came from the principal that represents the real-world person “Alice.” More generally, the guard must establish the origin of the message.
2. Is this request actually the one that Alice made? Or, for example, has an adversary modified the message? The guard must establish the integrity of the message.

This section provides the techniques to answer these two questions.

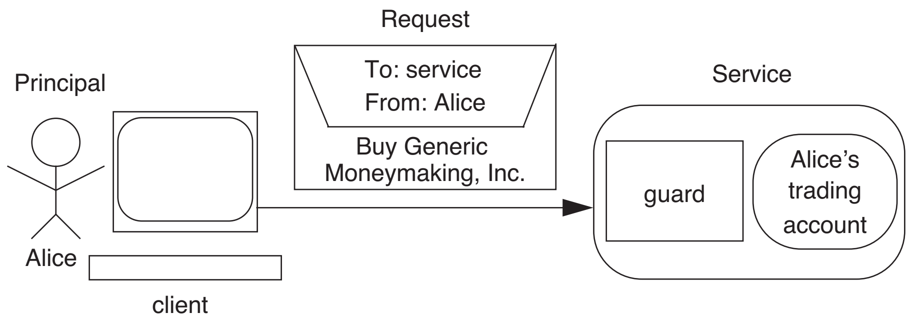


FIGURE 11.3

Authentication model.

11.2.1 Separating Trust from Authenticating Principals

Authentication consists of reliably identifying the principal associated with a request. Authentication can be provided by technical means such as passwords and signing messages. The technical means create a chain of evidence that securely connects an incoming request with a principal, perhaps by establishing that a message came from the same principal as a previous message. The technical means may even be able to establish the real-world identity of the principal.

Once the authentication mechanisms have identified the principal, there is a closely related but distinct problem: can the principal be trusted? The authentication means may be able to establish that the real-world identity for a principal is the person "Alice," but other techniques are required to decide whether and how much to trust Alice. The trading service may decide to consider Alice's request because the trading service can, by technical means, establish that Alice's credit card number is valid. To be more precise, the trading service trusts the credit card company to come through with the money and relies on the credit card company to establish the trust that Alice will pay her credit card bill.

The authenticity and trust problems are connected through the name of the principal. The technical means establish the name of the principal. Names for principals come in many flavors: for example, the name might be a symbolic one, like "Alice", a credit card number, a pseudonym, or a cryptographic key. The psychological techniques establish trust in the principal's name. For example, a reporter might trust information from an anonymous informer who has a pseudonym because previous content of the messages connected with the pseudonym has always been correct.

To make the separation of trust from authentication of principals more clear, consider the following example. You hear about an Internet bookstore named "ShopWithUs.com". Initially, you may not be sure what to think about this store. You look at their Web site, you talk to friends who have bought books from them, you hear a respectable person say publicly that this store is where the person buys books, and from all of this information you develop some trust that perhaps this bookstore is for real and is safe to order from. You order one book from ShopWithUs.com and the store delivers it faster than you expected. After a while you are ordering all your books from them because it saves the drive to the local bookstore and you have found that they take defective books back without a squabble.

Developing trust in ShopWithUs.com is the psychological part. The name ShopWithUs.com is the principal identifier that you have learned that you can trust. It is the name you heard from your friends, it is the name that you tell your Web browser, and it is the name that appears on your credit card bill. Your trust is based on that name; when you receive an e-mail offer from “ShopHere.com”, you toss it in the trash because, although the name is similar, it does not precisely match the name.

When you actually buy a book at ShopWithUs.com, the authentication of principal comes into play. The mechanical techniques allow you to establish a secure communication link to a Web site that claims to be ShopWithUs.com, and verify that this Web site indeed has the name ShopWithUs.com. The mechanical techniques do not themselves tell you who you are dealing with; they just assure you that whoever it is, it is named ShopWithUs.com. You must decide yourself (the psychological component) who that is and how much to trust them.

In the reverse direction, ShopWithUs.com would like to assure itself that it will be paid for the books it sends. It does so by asking you for a principal identifier—your credit card number—and subcontracting to the credit card company the psychological component of developing trust that you will pay your credit card bills. The secure communication link between your browser and the Web site of ShopWithUs.com assures ShopWithUs.com that the credit card number you supply is securely associated with the transaction, and a similar secure communication link to the credit card company assures ShopWithUs.com that the credit card number is a valid principal identifier.

11.2.2 Authenticating Principals

When the trading service receives the message, the guard knows that the message *claims* to come from the person named “Alice”, but it doesn’t know whether or not the claim is true. The guard must verify the claim that the identifier Alice corresponds to the principal who sent the message.

Most authentication systems follow this model: the sender tells the guard its principal identity, and the guard verifies that claim. This verification protocol has two stages:

1. A rendezvous step, in which a real-world person physically visits an authority that configures the guard. The authority checks the identity of the real-world person, creates a principal identifier for the person, and agrees on a method by which the guard can later identify the principal identifier for the person. One must be

particularly cautious in checking the real-world identity of a principal because an adversary may be able to fake it.

2. A verification of identity, which occurs at various later times. The sender presents a claimed principal identifier and the guard uses the agreed-upon method to verify the claimed principal identifier. If the guard is able to verify the claimed principal identifier, then the source is authenticated. If not, the guard disallows access and raises an alert.

The verification method the user and guard agree upon during the rendezvous step falls in three broad categories:

- The method uses a unique physical property of the user. For example, faces, voices, fingerprints, etc. are assumed to identify a human uniquely. For some of these properties it is possible to design a verification interface that is acceptable to users: for example, a user speaks a sentence into a microphone and the system compares the voice print with a previous voice print on file. For other properties it is difficult to design an acceptable user interface; for example, a computer system that asks "please, give a blood sample" is not likely to sell well. The uniqueness of the physical property and whether it is easy to reproduce (e.g., replaying a recorded voice) determine the strength of this identification approach. Physical identification is sometimes a combination of a number of techniques (e.g., voice and face or iris recognition) and is combined with other methods of verification.
- The method uses something unique the user *has*. The user might have an ID card with an identifier written on a magnetic strip that can be read by a computer. Or, the card might contain a small computer that stores a secret; such cards are called smart cards. The security of this method depends on (1) users not giving their card to someone else or losing it, and (2) an adversary being unable to reproduce a card that contains the secret (e.g., copying the content of the magnetic strip). These constraints are difficult to enforce, since an adversary might bribe the user or physically threaten the user to give the adversary the user's card. It is also difficult to make tamper-proof devices that will not reveal their secret.
- The method uses something that only the user *knows*. The user remembers a secret string, for example, a password, a personal identification number (PIN) or, as will be introduced in Section 11.3, a cryptographic key. The strength of this method depends on (1) the user not giving away (voluntarily or involuntarily) the password and (2) how difficult it is for an adversary to guess the user's secret. Your mother's maiden name and 4-digit PINs are *weak* secrets.

For example, when Alice created a trading account, the guard might have asked her for a principal identifier and a *password* (a secret character string), which the guard stores. This step is the rendezvous step. Later when Alice sends a message to trade, she includes in the message her claimed principal identifier ("Alice") and her password, which the guard verifies by comparing it with its stored copy. If the password in the message matches, the guard knows that this message came from the principal Alice, assuming that Alice didn't disclose her password to anyone else voluntarily or involuntarily. This step is the verification step.

In real-life authentication we typically use a similar process. For example, we first obtain a passport by presenting ourselves at the passport bureau, where we answer questions, provide evidence of our identity, and a photograph. This step is the rendezvous step. Later, we present the passport at a border station. The border guard examines the information in the passport (height, hair color, etc.) and looks carefully at the photograph. This step is the verification step.

The security of authenticating principals depends on, among other things, how carefully the rendezvous step is executed. As we saw above, a common process is that before a user is allowed to use a computer system, the user must see an administrator in person and prove to the administrator the user's identity. The administrator might ask the prospective user, for example, for a passport or a driving license. In that case, the administrator relies on the agency that issued the passport or driving license to do a good job in establishing the identity of the person.

In other applications the rendezvous step is a lightweight procedure and the guard cannot place much trust in the claimed identity of the principal. In the example with the trading service, Alice chooses her principal identifier and password. The service just stores the principal identifier and password in its table, but it has no direct way of verifying Alice's identity; Alice is unlikely to be able to see the system administrator of the trading service in person because she might be at a computer on the other side of the world. Since the trading service cannot verify Alice's identity, the service puts little trust in any claimed connection between the principal identifier and a real-world person. The account exists for the convenience of Alice to review, for example, her trades; when she actually buys something, the service doesn't verify Alice's identity, but instead verifies something else (e.g., Alice's credit card number). The service trusts the credit card company to verify the principal associated with the credit card number. Some credit card companies have weak verification schemes, which can be exploited by adversaries for identity theft.

11.2.3 Cryptographic Hash Functions, Computationally Secure, Window of Validity

The most commonly employed method for verifying identities in computer systems is based on passwords because it has a convenient user interface; users can just type in their name and password on a keyboard. However, there are several weaknesses in this approach. One weakness is that the stored copy of the password becomes an attractive target for adversaries. One way to remove this weakness is to store a cryptographic hash of the password in the password file of the system, rather than the password itself.

A *cryptographic hash function* maps an arbitrary-sized array of bytes M to a fixed-length value V, and has the following properties:

1. For a given input M, it is easy to compute $V \leftarrow H(M)$, where H is the hash function;
2. It is difficult to compute M knowing only V;
3. It is difficult to find another input M' such that $H(M') = H(M)$;
4. The computed value V is as short as possible, but long enough that H has a low probability of collision: the probability of two different inputs hashing to the same value V must be so low that one can neglect it in practice. A typical size for V is 160 to 256 bits.

The challenge in designing a cryptographic hash function is finding a function that has all these properties. In particular, providing property 3 is challenging. Section 11.8 describes an implementation of the Secure Hash Algorithm (SHA), which is a U.S. government and OECD standard family of hash algorithms.

Cryptographic hash functions, like most cryptographic functions, are *computationally secure.* They are designed in such a way that it is computationally infeasible to break them, rather than being impossible to break. The idea is that if it takes an unimaginable number of years of computation to break a particular function, then we can consider the function secure.

Computationally security is measured quantified using a *work factor*. For cryptographic hash functions, the work factor is the minimum amount of work required to compute a message M' such that for a given M, $H(M') = H(M)$. Work is measured in primitive operations (e.g., processor cycles). If the work factor is many years, then for all practical purposes, the function is just as secure as an unbreakable one because in both cases there is probably an easier attack approach based on exploiting human fallibility.

In practice, computationally security is measured by a *historical* work factor. The historical work factor is the work factor based on the current best-known algorithms and current state-of-the-art technology to break a cryptographic function. This method of evaluation runs the risk that an adversary might come up with a better algorithm to break a cryptographic function than the ones that are currently known, and furthermore technology changes may reduce the work factor. Given the complexities of designing and analyzing a cryptographic function, it is advisable to use only ones, such as SHA-256, that have been around long enough that they have been subjected to much careful, public review.

Theoreticians have developed models under which they can make absolute statements about the hardness of some cryptographic functions. Coming up with good models that match practice and the theoretical analysis of security primitives is an active area of research with a tremendous amount of progress in the last three decades, but also with many open problems.

Given that *d(technology)/dt* is so high in computer systems and cryptography is a fast developing field, it is good practice to consider the *window of validity* for a specific cryptographic function. The window of validity of a cryptographic function is the minimum of the time-to-compromise of all of its components. The window of validity for cryptographic hash functions is the minimum of the time to compromise the hash algorithm and the time to find a message M' such that for a given M, $H(M') = H(M)$. The window of validity of a password-based authentication system is the minimum of the window of validity of the hashing algorithm, the time to try all possible passwords, and the time to compromise a password.

A challenge in system design is that the window of validity of a cryptographic function may be shorter than the lifetime of the system. For example, SHA, now referrred to as "SHA-0" and which produces a 160-bit value for V was first published in 1993, and superseded just two years later by SHA-1 to repair a possible weakness. Indeed, in 2004, a cryptographic researcher found a way to systematically derive examples of messages M and M' that SHA-0 hashes to the same value. Research published in 2005 suggest weaknesses in SHA-1, but as of 2007 no one has yet found a systematic way to compromise that widely used hash algorithm (i.e., for a given M no one has yet found a M' that hashes to the same value of $H(M)$). As a precaution, however, the National Institute for Standards and Technology is recommending that by 2010 users switch to versions of SHA (for example, SHA-256) that produce longer values for V. A system designer should be prepared that during the lifetime of a computer system the cryptographic hash function may have to be replaced, perhaps more than once.

11.2.4 Using Cryptographic Hash Functions to Protect Passwords

There are many usages of cryptographic hash functions, and we will see them show up in this chapter frequently. One good use is to protect passwords. The advantage of storing the cryptographic hash of the password in the password file instead of the password itself is that the hash value does not need to be kept secret. For this purpose, the important property of the hash function is the second property in the list in Section 11.2.3, that if the adversary has only the output of a hash function (e.g., the adversary was able to steal the password file), it is difficult to compute a corresponding input. With this scheme, even the system administrator cannot figure out what the user's password is. (Design principle: *Minimize secrets*.)

The verification of identity happens when a user logs onto the computer. When the user types a password, the guard computes the cryptographic hash of the typed password and compares the result with the value stored in the table. If the values match, the verification of identity was successful; if the verification fails, the guard denies access.

The most common attack on this method is a brute-force attack, in which an adversary tries all possible passwords. A brute-force attack can take a long time, so adversaries often use a more sophisticated version of it: a *dictionary attack*, which works well for passwords because users prefer to select an easy-to-remember password. In a dictionary attack, an adversary compiles a list of likely passwords: first names*, last names, street names, city names, words from a dictionary, and short strings of random characters. Names of cartoon characters and rock bands have been shown to be effective guesses in universities.

The adversary either computes the cryptographic hash of these strings and compares the result to the value stored in the computer system (if the adversary has obtained the table), or writes a computer program that repeatedly attempts to log on with each of these strings. A variant of this attack is an attack on a specific person's password. Here the adversary mines all the information one can find (mother's maiden name, daughter's birth date, license plate number, etc.) about that person and tries passwords consisting of that information forwards and backwards. Another variant is of this attack is to try a likely password on each user of a popular Internet site; if passwords are 20 bits (e.g., a 6-digit PIN), then trying a given PIN as a password for 10,000,000 accounts is likely to yield success for 10 accounts ($10 \times 2^{20} = 10{,}000{,}000$).

Several studies have shown that brute-force and dictionary attacks are effective in practice because passwords are often inherently weak. Users prefer easy-to-remember passwords, which are often short and contain existing words, and thus dictionary attacks work well. System designers have countered this problem in several ways. Some systems force the user to chose a strong password, and require the user to change it frequently. Some systems disable an account after 3 failed login attempts. Some systems require users to use both a password and a secret generated by the user's portable cryptographic device (e.g., an authentication device with a cryptographic coprocessor). In addition, system designers often try to make it difficult for adversaries to compile a list of all users on a service and limit access to the file with cryptographic hashes of passwords.

Since the verification of identity depends solely on the password, it is prudent to make sure that the password is never disclosed in insecure areas. For example, when a user logs on to a remote computer, the system should avoid sending the password unprotected over an untrusted network. That is easier said than done. For example, sending the cryptographic hash of the password is not good enough because if the adversary can capture the hash by eavesdropping, the adversary might be able to replay the hash in a later message and impersonate a principal or determine the secret using a dictionary attack.

In general, it is advisable to minimize repeated use of a secret because each exposure increases the chance that the adversary may discover the secret. To minimize exposure, any security scheme based on passwords should use them only *once* per session with a particular service: to verify the identity of a person at the first access. After the first access, one should use a newly-generated, strong secret for further accesses. More generally, what we need is a protocol between the user and the service that has the following properties:

1. it authenticates the principal to the guard;
2. it authenticates the service to the principal;

* A classic study is by Frederick T. Grampp and Robert H. Morris. UNIX operating system security. *Bell System Technical Journal 63,* 8, Part 2 (October, 1984), pages 1649–1672. The authors made a list of 200 names by selecting 20 common female names and appending to each one a single digit (the system they tested required users to select a password containing at least 6 characters and one digit). At least one entry of this list was in use as a password on each of several dozen UNIX machines they examined.

3. the password never travels over the network so that adversaries cannot learn the password by eavesdropping on network traffic;
4. the password is used only once per session so that the protocol exposes this secret as few times as possible. This has the additional advantage that the user must type the password only once per session.

The challenge in designing such a protocol is that the designer must assume that one or more of the parties involved in the protocol may be under the control of an adversary. An adversary should not be able to impersonate a principal, for example, by recording all network messages between the principal and the service, and replaying it later. To withstand such attacks we need a *security protocol*, a protocol designed to achieve some security objective. Before we can discuss such protocols, however, we need some other security mechanisms. For example, since any message in a security protocol might be forged by an adversary, we first need a method to check the authenticity of messages. We discuss message authentication next, the design of confidential communication links in Section 11.4, and the design of security protocols in Section 11.5. With these mechanisms one can design among many other things a secure password protocol.

11.3 Authenticating Messages

Returning to Figure 11.3, when receiving a message, the guard needs an ensured way of determining *what* the sender said in the message and *who* sent the message. Answering these two questions is the province of *message authentication*. Message authentication techniques prevent an adversary from forging messages that pretend to be from someone else, and allow the guard to determine if an adversary has modified a legitimate message while it was en route.

In practice, the ability to establish who sent a message is limited; all that the guard can establish is that the message came from the same origin as some previous message. For this reason, what the guard really does is to establish that a message is a member of a chain of messages identified with some principal. The chain may begin in a message that was communicated by a physical rendezvous. That physical rendezvous securely binds the identity of a real-world person with the name of a principal, and both the real-world person and that principal can now be identified as the origin of the current message. For some applications it is unimportant to establish the real-world person that is associated with the origin of the message. It may be sufficient to know that the message originated from the same source as earlier messages and that the message is unaltered. Once the guard has identified the principal (and perhaps the real-world identity associated with the principal), then we may be able to use psychological means to establish trust in the principal, as explained in Section 11.2.

To establish that a message belongs to a chain of messages, a guard must be able to verify the authenticity of the message. Message authenticity requires *both*:

- *data integrity*: the message has not been changed since it was sent;

- *origin authenticity*: the claimed origin of the message, as learned by the receiver from the message content or from other information, is the actual origin.

The issues of data integrity and origin authenticity are closely related. Messages that have been altered effectively have a new origin. If an origin cannot be determined, the very concept of message integrity becomes questionable (the message is unchanged with respect to what?). Thus, integrity of message data has to include message origin, and vice versa. The reason for distinguishing them is that designers using different techniques to tackle the two.

In the context of authentication, we mostly talk about authenticating messages. However, the concept also applies to communication streams, files, and other objects containing data. A stream is authenticated by authenticating successive segments of the stream. We can think of each segment as a message from the point of view of authentication.

11.3.1 Message Authentication is Different from Confidentiality

The goal of message confidentiality (keeping the content of messages private) and the goal of message authentication are related but different, and separate techniques are usually used for each objective, similar to the physical world. With paper mail, signatures authenticate the author and sealed envelopes protect the letter from being read by others.

Authentication and confidentiality can be combined in four ways, three of which have practical value:

- Authentication and confidentiality. An application (e.g., electronic banking), might require both authentication and confidentiality of messages. This case is like a signed letter in a sealed envelope, which is appropriate if the content of the message (e.g., it contains personal financial information) must be protected and the origin of the message must be established (e.g., the user who owns the bank account).
- Authentication only. An application, like DNS, might require just authentication for its announcements. This case is like a signed letter in an unsealed envelope. It is appropriate, for example, for a public announcement from the president of a company to its employees.
- Confidentiality only. Requiring confidentially without authentication is uncommon. The value of a confidential message with an unverified origin is not great. This case is like a letter in a sealed envelope, but without a signature. If the guard has no idea who sent the letter, what level of confidence can the guard have in the content of the letter? Moreover, if the receiver doesn't know who the sender is, the receiver has no basis to trust the sender to keep the content of the message confidential; for all the receiver knows, the sender may have released the content of the letter to someone else too. For these reasons confidentiality only is uncommon in practice.

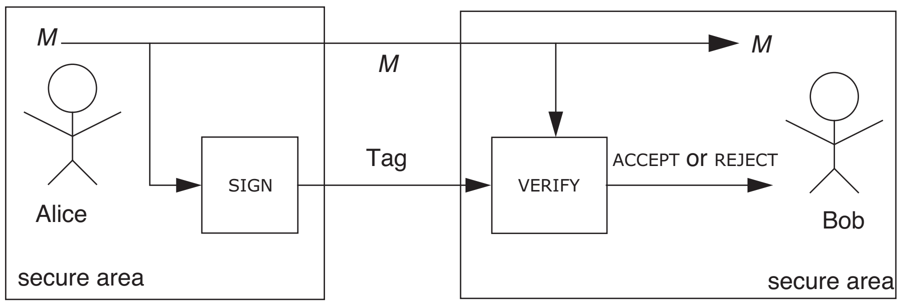


FIGURE 11.4

A closed design for authentication relies on the secrecy of an algorithm.

- Neither authentication or confidentiality. This combination is appropriate if there are no intentionally malicious users or there is a separate code of ethics.

To illustrate the difference between authentication and confidentiality, consider a user who browses a Web service that publishes data about company stocks (e.g., the company name, the current trading price, recent news announcements about the company, and background information about the company). This information travels from the Web service over the Internet, an untrusted network, to the user's Web browser. We can think of this action as a message that is being sent from the Web service to the user's browser:

From: stock.com
To: John's browser
Body: At 10 a.m. Generic Moneymaking, Inc. was trading at $1

The user is not interested in confidentiality of the data; the stock data is public anyway. The user, however, is interested in the authenticity of the stock data, since the user might decide to trade a particular stock based on that data. The user wants to be assured that the data is coming from "stock.com" (and not from a site that is pretending to be stock.com) and that the data was not altered when it crossed the Internet. For example, the user wants to be assured that an adversary hasn't changed "Generic Moneymaking, Inc.", the price, or the time. We need a scheme that allows the user to verify the authenticity of the publicly readable content of the message. The next section introduces cryptography for this purpose. When cryptography is used, content that is publicly readable is known as *plaintext* or *cleartext*.

11.3.2 Closed versus Open Designs and Cryptography

In the authentication model there are two *secure areas* (a physical space or a virtual address space in which information can be safely confined) separated by an insecure communication path (as shown in Figure 11.4) and two boxes: SIGN and VERIFY. Our goal is to set up a *secure channel* between the two secure areas that provides authenticity for messages sent between the two secure areas. (Section 11.4 shows how one can implement a secure channel that also provides confidentiality.)

Before diving in the details of how to implement SIGN and VERIFY, lets consider how we might use them. In a secure area, the sender Alice creates an authentication tag for a message by invoking SIGN with the message as an argument. The tag and message are communicated through the insecure area to the receiver Bob. The insecure communication path might be a physical wire running down the street or a connection across the Internet. In both cases, we must assume that a wire-tapper can easily and surreptitiously gain access to the message and authentication tag. Bob verifies the authenticity of the message by a computation based on the tag and the message. If the received message is authentic, VERIFY returns ACCEPT; otherwise it returns REJECT.

C*ryptographic transformations* can be used protect against a wide range of attacks on messages, including ones on the authenticity of messages. Our interest in cryptographic transformations is not the underlying mathematics (which is fascinating by itself, as can been seen in Section 11.8), but that these transformations can be used to implement security primitives such as SIGN and VERIFY.

One approach to implementing a cryptographic system, called a *closed* design, is to keep the construction of cryptographic primitives, such as VERIFY and SIGN, secret with that idea that if the adversary doesn't understand how SIGN and VERIFY work, it will be difficult to break the tag. Auguste Kerchkoffs more than a century ago* observed that this closed approach is typically bad, since it violates the basic design principles for secure systems in a number of ways. It doesn't minimize what needs to be secret. If the design is compromised, the whole system needs to be replaced. A review to certify the design must be limited, since it requires revealing the secret design to the reviewers. Finally, it is unrealistic to attempt to maintain secrecy of any system that receives wide distribution.

These problems with closed designs led Kerchkoffs to propose a design rule, now known as Kerchkoffs' criterion, which is a particular application of the principles of *open design* and *least privilege*: *minimize secrets*. For a cryptographic system, open design means that we concentrate the secret in a corner of a cryptographic transformation, and make the secret removable and easily changeable. An effective way of doing this is to reduce the secret to a string of bits; this secret bit string is known as a *cryptographic key*, or *key* for short. By choosing a longer key, one can generally increase the time for the adversary to compromise the transformation.

Figure 11.5 shows an open design for SIGN and VERIFY. In this design the algorithms for SIGN and VERIFY are public and the only secrets are two keys, K_1 and K_2. What distinguishes this open design from a closed design is (1) that public analysis of SIGN and VERIFY can provide verification of their strength without compromising their security; and (2)

* "Il faut un systeme remplissant certaines conditions exceptionelles ... il faut qu'il n'exige pas le secret, et qu'il puisse sans inconvenient tomber entre les mains de l'ennemi." (Compromise of the system should not disadvantage the participants.) Auguste Kerchkoffs, *La cryptographie Militaire*, Chapter II (1883).

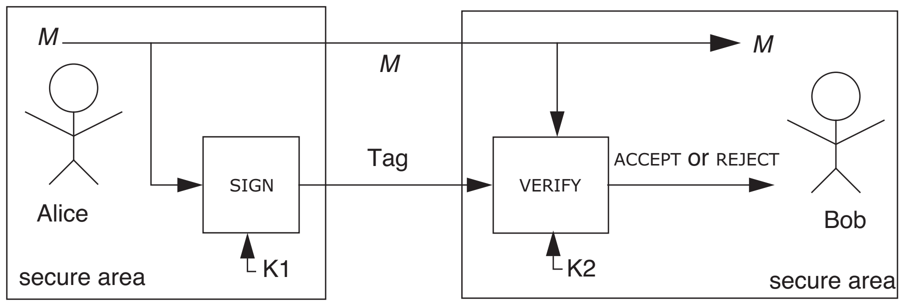


FIGURE 11.5

An open design for authentication relies on the secrecy of keys.

it is easy to change the secret parts (i.e., the two keys) without having to reanalyze the system's strength.

Depending on the relation between K_1 and K_2, there are two basic approaches to key-based transformations of a message: *shared-secret cryptography* and *public-key cryptography.* In shared-secret cryptography K_1 is easily computed from K_2 and vice versa. Usually in shared-secret cryptography $K_1 = K_2$, and we make that assumption in the text that follows.

In public-key cryptography K_1 cannot be derived easily from K_2 (and vice versa). In public-key cryptography, only one of the two keys must be kept secret; the other one can be made public. (A better label for public-key cryptography might be "cryptography without shared secrets", or even "non-secret encryption", which is the label adopted by the intelligence community. Either of those labels would better contrast it with shared-secret cryptography, but the label "public-key cryptography" has become too widely used to try to change it.)

Public-key cryptography allows Alice and Bob to perform cryptographic operations without having to share a secret. Before public-key systems were invented, cryptographers worked under the assumption that Alice and Bob needed to have a shared secret to create, for example, SIGN and VERIFY primitives. Because sharing a secret can be awkward and maintaining its secrecy can be problematic, this assumption made certain applications of cryptography complicated. Because public-key cryptography removes this assumption, it resulted in a change in the way cryptographers thought, and has led to interesting applications, as we will see in this chapter.

To distinguish the keys in shared-secret cryptography from the ones in public-key cryptography, we refer to the key in shared-secret cryptography as the *shared-secret key.* We refer to the key that can be made public in public-key cryptography as the *public key* and to the key that is kept secret in public-key cryptography as the *private key.* Since shared-secret keys must also be kept secret, the unqualified term "secret key," which is sometimes used in the literature, can be ambiguous, so we avoid using it.

We can now see more specifically the two ways in which SIGN and VERIFY can benefit if they are an open design. First, If K_1 or K_2 is compromised, we can select a new key for future communication, without having to replace SIGN and VERIFY. Second, we can now publish the overall design of the system, and how SIGN and VERIFY work. Anyone can review the design and offer opinions about its correctness.

Because most cryptographic techniques use open design and reduce any secrets to keys, a system may have several keys that are used for different purposes. To keep the keys apart, we refer to the keys for authentication as *authentication keys*.

11.3.3 Key-Based Authentication Model

Returning to Figure 11.5, to authenticate a message, the sender *signs* the messages using a key K_1. Signing produces as output an *authentication tag*: a *key-based cryptographic transformation* (usually shorter than the message). We can write the operation of signing as follows:

$$T \leftarrow \text{SIGN}\ (M, K_1)$$

where T is the authentication tag.

The tag may be sent to the receiver separately from the message or it may be appended to the message. The message and tag may be stored in separate files or attachments. The details don't matter.

Let's assume that the sender sends a message $\{M, T\}$. The receiver receives a message $\{M', T'\}$, which may be the same as $\{M, T\}$ or it may not. The purpose of message authentication is to decide which. The receiver unmarshals $\{M', T'\}$ into its components M' and T', and *verifies* the authenticity of the received message, by performing the computation:

$$result \leftarrow \text{VERIFY}\ (M', T', K_2)$$

This computation returns ACCEPT if M' and T' match; otherwise, it returns REJECT.

The design of SIGN and VERIFY should be such that if an adversary forges a tag, re-uses a tag from a previous message on a message fabricated by the adversary, etc. the adversary won't succeed. Of course, if the adversary replays a message $\{M, T\}$ without modifying it, then VERIFY will again return ACCEPT; we need a more elaborate security protocol, the topic of Section 11.5, to protect against replayed messages.

If M is a long message, a user might sign and verify the cryptographic hash of M, which is typically less expensive than signing M because the cryptographic hash is shorter than M. This approach complicates the protocol between sender and receiver a bit because the receiver must accurately match up M, its cryptographic hash, and its tag. Some implementations of SIGN and VERIFY implement this performance optimization themselves.

11.3.4 Properties of SIGN and VERIFY

To get a sense of the challenges of implementing SIGN and VERIFY, we outline some of the basic requirements for SIGN and VERIFY, and some attacks that a designer must consider.

The sender sends {M, T} and the receiver receives {M', T'}. The requirements for an authentication system with shared-secret key K are as follows:

1. VERIFY (M', T', K) returns ACCEPT if $M' = M$, $T' =$ SIGN (M, K)
2. Without knowing K, it is difficult for an adversary to compute an M' and T' such that VERIFY (M', T', K) returns ACCEPT
3. Knowing M, T, and the algorithms for SIGN and VERIFY doesn't allow an adversary to compute K

In short, T should be dependent on the message content M and the key K. For an adversary who doesn't know key K, it should be impossible to construct a message M' and a T' different from M and T that verifies correctly using key K.

A corresponding set of properties must hold for public-key authentication systems:

1. VERIFY (M', T', K_2) returns ACCEPT if $M' = M$, $T' =$ SIGN (M, K_1)
2. Without knowing K_1, it is difficult for an adversary to compute an M' and T' such that VERIFY (M', T', K_2) returns ACCEPT
3. Knowing M, T, K_2, and the algorithms for verify and sign doesn't allow an adversary to compute K_1

The requirements for SIGN and VERIFY are formulated in absolute terms. Many good implementations of VERIFY and SIGN, however, don't meet these requirements perfectly. Instead, they might guarantee property 2 with very high probability. If the probability is high enough, then as a practical matter we can treat such an implementation as being acceptable.What we require is that the probability of *not* meeting property 2 be much lower than the likelihood of a human error that leads to a security breach.

The work factor involved in compromising SIGN and VERIFY is dependent on the key length; a common way to increase the work factor for the adversary is use a longer key. A typical key length used in the field for the popular RSA public-key cipher (see Section 11.8) is 1,024 or 2,048 bits. SIGN and VERIFY implemented with shared-secret ciphers often use shorter keys (in the range of 128 to 256 bits) because existing shared-secret ciphers have higher work factors than existing public-key ciphers. It is also advisable to change keys periodically to limit the damage in case a key is compromised and cryptographic protocols often do so (see Section 11.5).

Broadly speaking, the attacks on authentication systems fall in five categories:

1. Modifications to M and T. An adversary may attempt to change M and the corresponding T. The VERIFY function should return REJECT even if the adversary deletes or flips only a single bit in M and tries to make corresponding change to T. Returning to our trading example, VERIFY should return REJECT if the adversary changes M from "At 10 a.m. Generic Moneymaking, Inc. was trading at $1" to "At 10 a.m. Generic Moneymaking, Inc. was trading at $200" and tries to make the corresponding changes to T.

2. Reordering *M*. An adversary may not change any bits, but just reorder the existing content of *M*. For example, VERIFY should return REJECT if the adversary changes *M* to "At 1 a.m. Generic Moneymaking, Inc. was trading at $10" (The adversary has moved "0" from "10 a.m." to "$10").
3. Extending *M* by prepending or appending information to *M*. An adversary may not change the content of *M*, but just prepend or append some information to the existing content of *M*. For example, an adversary may change *M* to "At 10 a.m. Generic Moneymaking, Inc. was trading at $10". (The adversary has appended "0" to the end of the message.)
4. Splicing several messages and tags. An adversary may have recorded two messages and their tags, and tried to combine them into a new message and tag. For example, an adversary might take "At 10 a.m. Generic Moneymaking, Inc." from one transmitted message and combine it with "was trading at $9" from another transmitted message, and splice the two tags that go along with those messages by taking the first several bytes from the first tag and the remainder from the second tag.
5. Since SIGN and VERIFY are based on cryptographic transformations, it may also be possible to directly attack those transformations. Some mathematicians, known as cryptanalysts, are specialists in devising such attacks.

These requirements and the possible attacks make clear that the construction of SIGN and VERIFY primitives is a difficult task. To protect messages against the attacks listed above requires a cryptographer who can design the appropriate cryptographic transformations on the messages. These transformations are based on sophisticated mathematics. Thus, we have the worst of two possible worlds: we must achieve a negative goal using complex tools. As a result, even experts have come up with transformations that failed spectacularly. Thus, a non-expert certainly should *not* attempt to implement SIGN and VERIFY, and their implementation falls outside the scope of this book. (The interested reader can consult Section 11.8 to get a flavor of the complexities.)

The window of validity for SIGN and VERIFY is the minimum of the time to compromise the signing algorithm, the time to compromise the hash algorithm used in the signature (if one is used), the time to try out all keys, and the time to compromise the signing key.

As an example of the importance of keeping track of the window of validity, a team of researchers in 2008 was able to create forged signatures that many Web browsers accepted as valid.* The team used a large array of processors found in game consoles to perform a collision attack on a hash function designed in 1994 called MD5. MD5 had been identified as potentially weak as early as 1996 and a collision attack was demonstrated in 2004. Continued research revealed ways of rapidly creating collisions, thus allowing a search for helpful collisions. The 2008 team was able to find a helpful collision

* A. Sotirov et al. MD5 considered harmful: creating a rogue CA certificate. *25th Annual Chaos Communication Congress*, Berlin, December 2008.

with which they could forge a trusted signature on an authentication message. Because some authentication systems that Web browsers trust had not yet abandoned their use of MD5, many browsers accepted the signature as valid and the team was able to trick these browsers into making what appeared to be authenticated connections to well-known Web sites. The connections actually led to impersonation Web sites that were under the control of the research team. (The forged signatures were on certificates for the transport layer security (TLS) protocol. Certificates are discussed in Sections 11.5.1 and 11.7.4, and Section 11.10 is a case study of TLS.)

11.3.5 Public-key versus Shared-Secret Authentication

If Alice signs the message using a shared-secret key, then Bob verifies the tag using the *same* shared-secret key. That is, VERIFY checks the received authentication tag from the message and the shared-secret key. An authentication tag computed with a shared-secret key is called a *message authentication code* (*MAC*). (The verb "to MAC" is the common jargon for "to compute an authentication tag using shared-secret cryptography".)

In the literature, the word "sign" is usually reserved for generating authentication tags with public-key cryptography. If Alice signs the message using public-key cryptography, then Bob verifies the message using a *different* key from the one that Alice used to compute the tag. Alice uses her private key to compute the authentication tag. Bob uses Alice's corresponding public key to verify the authentication tag. An authentication tag computed with a public-key system is called a *digital signature*. The digital signature is analogous to a conventional signature because only one person, the holder of the private key, could have applied it.

Alice's digital signatures can be checked by anyone who knows Alice's public key, while checking her MACs requires knowledge of the shared-secret key that she used to create the MAC. Thus, Alice might be able to successfully *repudiate* (disown) a message authenticated with a MAC by arguing that Bob (who also knows the shared-secret key) forged the message and the corresponding MAC.

In contrast, the only way to repudiate a digital signature is for Alice to claim that someone else has discovered her private key. Digital signatures are thus more appropriate for electronic checks and contracts. Bob can verify Alice's signature on an electronic check she gives him, and later when Bob deposits the check at the bank, the bank can also verify her signature. When Alice uses digital signatures, neither Bob nor the bank can forge a message purporting to be from Alice, in contrast to the situation in which Alice uses only MACs.

Of course, non-repudiation depends on not losing one's private key. If one loses one's private key, a reliable mechanism is needed for broadcasting the fact that the private key is no longer secret so that one can repudiate later forged signatures with the lost private key. Methods for revoking compromised private keys are the subject of considerable debate.

SIGN and VERIFY are two powerful primitives, but they must be used with care. Consider the following attack. Alice and Bob want to sign a contract saying that Alice will pay Bob $100. Alice types it up as a document using a word-processing application and both digitally sign it. In a few days Bob comes to Alice to collect his money. To his surprise, Alice presents him with a Word document that states he owes her $100. Alice also has a valid signature from Bob for the new document. In fact, it is the exact same signature as for the contract Bob remembers signing and, to Bob's great amazement, the two documents are actually bit-for-bit identical. What Alice did was create a document that included an **if** statement that changed the displayed content of the document by referring to an external input such as the current date or filename. Thus, even though the signed contents remained the same, the displayed contents changed because they were partially dependent on unsigned inputs. The problem here is that Bob's mental model doesn't correspond to what he has signed. As always with security, all aspects must be thought through! Bob is much better off signing only documents that he himself created.

11.3.6 Key Distribution

We assumed that if Bob successfully verified the authentication tag of a message, that Alice is the message's originator. This assumption, in fact, has a serious flaw. What Bob really knows is that the message originated from a principal that knows key K_1. The assumption that the key K_1belongs to Alice may not be true. An adversary may have stolen Alice's key or may have tricked Bob into believing that K_1 is Alice's key. Thus, the way in which keys are bound to principals is an important problem to address.

The problem of securely distributing keys is also sometimes called the *name-to-key binding* problem; in the real world, principals are named by descriptive names rather than keys. So, when we know the name of a principal, we need a method for securely finding the key that goes along with the named principal. The trust that we put in a key is directly related to how secure the key distribution system is.

Secure key distribution is based on a name discovery protocol, which starts, perhaps unsurprisingly, with trusted physical delivery. When Alice and Bob meet, Alice can give Bob a cryptographic key. This key is authenticated because Bob knows he received it exactly as Alice gave it to him. If necessary, Alice can give Bob this key secretly (in an envelope or on a portable storage card), so others don't see or overhear it. Alice could also use a mutually trusted courier to deliver a key to Bob in a secret and authenticated manner.

Cryptographic keys can also be delivered over a network. However, an adversary might add, delete, or modify messages on the network. A good cryptographic system is needed to ensure that the network communication is authenticated (and confidential, if necessary). In fact, in the early days of cryptography, the doctrine was never to send keys over a network; a compromised key will result in more damage than one compromised message. However, nowadays cryptographic systems are believed to be strong enough to take that risk. Furthermore, with a key-distribution protocol in place it is possible to periodically generate new keys, which is important to limit the damage in case a key is compromised.

The catch is that one needs cryptographic keys already in place in order to distribute new cryptographic keys over the network! This approach works if the recursion "bottoms out" with physical key delivery. Suppose two principals Alice and Bob wish to communicate, but they have no shared (shared-secret or public) key. How can they establish keys to use?

One common approach is to use a mutually-trusted third party (Charles) with whom Alice and Bob already each share key information. For example, Charles might be a mutual friend of Alice and Bob. Charles and Alice might have met physically at some point in time and exchanged keys and similarly Charles and Bob might have met and also exchanged keys. If Alice and Bob both trust Charles, then Alice and Bob can exchange keys through Charles.

How Charles can assist Alice and Bob depends on whether they are using shared-secret or public-key cryptography. Shared-secret keys need to be distributed in a way that is both confidential and authenticated. Public keys do not need to be kept secret, but need to be distributed in an authenticated manner. What we see developing here is a need for another security protocol, which we will study in Section 11.5.

In some applications it is difficult to arrange for a common third party. Consider a person who buys a personal electronic device that communicates over a wireless network. The owner installs the new gadget (e.g., digital surveillance camera) in the owner's house and would like to make sure that burglars cannot control the device over the wireless network. But, how does the device authenticate the owner, so that it can distinguish the owner from other principals (e.g., burglars)? One option is that the manufacturer or distributor of the device plays the role of Charles. When purchasing a device, the manufacturer records the buyer's public key. The device has burned into it the public key of the manufacturer; when the buyer turns on the device, the device establishes a secure communication link using the manufacturer's public key and asks the manufacturer for the public key of its owner. This solution is impractical, unfortunately: what if the device is not connected to a global network and thus cannot reach the manufacturer? This solution might also have privacy objections: should manufacturers be able to track when consumers use devices? Sidebar 11.5, about the *resurrecting duckling* provides a solution that allows key distribution to be performed locally, without a central principal involved.

Not all applications deploy a sophisticated key-distribution protocol. For example, the secure shell (SSH), a popular Internet protocol used to log onto a remote computer has a simple key distribution protocol. The first time that a user logs onto a server named "athena.Scholarly.edu", SSH sends a message in the clear to the machine with DNS name athena.Scholarly.edu asking it for its public key. SSH uses that public key to set up an authenticated and confidential communication link with the remote computer. SSH also caches this key and remembers that the key is associated with the DNS name "athena.Scholarly.edu". The next time the user logs onto athena.Scholarly.edu, SSH uses the cached key to set up the communication link.

Because the DNS protocol does not include message authentication, the security risk in SSH's approach is a masquerading attack: an adversary might be able to intercept the

Sidebar 11.5: Authenticating personal devices: the resurrecting duckling policy
Inexpensive consumer devices have (or will soon have) embedded microprocessors in them that are able to communicate with other devices over inexpensive wireless networks. If household devices such as the home theatre, the heating system, the lights, and the surveillance cameras are controlled by, say, a universal remote control, an owner must ensure that these devices (and new ones) obey the owner's commands and not the neighbor's or, worse, a burglar's.This situation requires that a device and the remote control be able to establish a secure relationship. The relationship may be transient, however; the owner may want to resell one of the devices, or replace the remote control.

In *The resurrecting duckling: security issues for ad-hoc wireless networks* [Suggestions for Further Reading 11.4.2], Stajano and Anderson provide a solution based on the vivid analogy of how ducklings authenticate their mother. When a duckling emerges from its egg, it will recognize as its mother the first moving object that makes a sound. In the Stajano and Anderson proposal, a device will recognize as its owner the first principal that sends it an authentication key. As soon as the device receives a key, its status changes from newborn to imprinted, and it stays faithful to that key until its death. Only an owner can force a device to die and thereby reverse its status to newborn. In this way, an owner can transfer ownership.

A widely used example of the resurrecting duckling is purchasing wireless routers. These routers often come with the default user name "Admin" and password "password". When the buyer plugs the router in for the first time, it is waiting to be imprinted with a better password; the first principal to change the password gets control of the router. The router has a resurrection button that restores the defaults, thus again making it imprintable (and allowing the buyer to recover if an adversary did grab control).

DNS lookup for "athena.Scholarly.edu" and return an IP address for a computer controlled by the adversary. When the user connects to that IP address, the adversary replies with a key that the adversary has generated. When the user makes an SSH connection using that public key, the adversary's computer masquerades as athena.Scholarly.edu. To counter this attack, the SSH client asks a question to the user on the first connection to a remote computer: "I don't recognize the key of this remote computer, should I trust it?" and a wary user should compare the displayed key with one that it received from the remote computer's system administrator over an out-of-band secure communication link (e.g., a piece of paper). Many users aren't wary and just answer "yes" to the question.

The advantage of the SSH approach is that no key distribution protocol is necessary (beyond obtaining the fingerprint). This has simplified the deployment of SSH and has made it a success. As we will see in Section 11.5, securely distributing keys such that a masquerading attack is impossible is a challenging problem.

11.3.7 Long-Term Data Integrity with Witnesses

Careful use of SIGN and VERIFY can provide both data integrity and authenticity guarantees. Some applications have requirements for which it is better to use different techniques for integrity and authenticity. Sidebar 7.1[on-line] mentions a digital archive, which requires protection against an adversary who tries to change the content of a file stored in the archive. To protect a file, a designer wants to make many separate replicas of the file, following the durability mantra, preferably in independently administered and thus separately protected domains. If the replicas are separately protected, it is more difficult for an adversary to change all of them.

Since maintaining widely-separated copies of large files consumes time, space, and communication bandwidth, one can reduce the resource expenditure by replacing some (but not all) copies of the file with a smaller witness, with which users can periodically check the validity of replicas (as explained in Section 10.3.4[on-line]). If the replica disagrees with the witness, then one repairs the replica by finding a replica that matches the witness. Because the witness is small, it is easy to protect it against tampering. For example, one can publish the witness in a widely-read newspaper, which is likely to be preserved either on microfilm or digitally in many public libraries.

This scheme requires that a witness be cryptographically secure. One way of constructing a secure witness is using SIGN and VERIFY. The digital archiver uses a cryptographic hash function to create a secure fingerprint of the file, signs the fingerprint with its private key, and then distributes copies of the file widely. Anyone can verify the integrity of a replica by computing the finger print of the replica, verifying the witness using the public key of the archiver, and then comparing the finger print of the witness against the finger print of the replica.

This scheme works well in general, but is less suitable for long-term data integrity. The window of validity of this scheme is determined by the minimum time to compromise the private key used for signing, the signing algorithm, the hashing algorithm, and the validity of the name-to-public key binding. If the goal of the archiver is to protect the data for many decades (or forever), it is likely that the digital signature will be invalid before the data.

In such cases, it is better to protect the witness by widely publishing just the cryptographic hash instead of using SIGN and VERIFY. In this approach, the validity of the witness is the time to compromise the cryptographic hash. This window can be made large. One can protect against a compromised cryptographic hash algorithm by occasionally computing and publishing a new witness with the latest, best hash algorithm. The new witness is a hash of the original data, the original witness, and a timestamp, thereby demonstrating the integrity of the original data at the time of the new witness calculation.

The confidence a user has in the authenticity of a witness is determined by how easily the user can verify that the witness was indeed produced by the archiver. If the newspaper or the library physically received the witnesses directly from the archiver, then this confidence may be high.

11.4 Message Confidentiality

Some applications may require message confidentiality in addition to message authentication. Two principals may want to communicate *privately* without adversaries having access to the communicated information. If the principals are running on a shared physical computer, this goal is easily accomplished using the kernel. For example, when sending a message to a port (see Section 5.3.5), it is safe to ask the kernel to copy the message to the recipient's address space, since the kernel is already trusted; the kernel can read the sender's and receiver's address space anyway.

If the principals are on different physical processors, and can communicate with each other only over an *untrusted* network, ensuring confidentiality of messages is more challenging. By definition, we cannot trust the untrusted network to not disclose the bits that are being communicated. The solution to this problem is to introduce encryption and decryption to allow two parties to communicate without anyone else being able to tell what is being communicated.

11.4.1 Message Confidentiality Using Encryption

The setup for providing confidentiality over untrusted networks is shown in Figure 11.6. Two secure areas are separated by an insecure communication path. Our goal is to provide a secure channel between the two secure areas that provides confidentiality.

Encryption transforms a *plaintext* message into *ciphertext* in such a way that an observer cannot construct the original message from the ciphertext version, yet the intended receiver can. *Decryption* transforms the received ciphertext into plaintext. Thus, one challenge in the implementation of channels that provide confidentiality is to use an encrypting scheme that is difficult to reverse for an adversary. That is, even if an observer could copy a message that is in transit and has an enormous amount of time and computing power available, the observer should not be able to transform the encrypted message into the plaintext message. (As with signing, we use the term messages conceptually; one can also encrypt and decrypt files, e-mail attachments, streams, or other data objects.)

The ENCRYPT and DECRYPT primitives can be implemented using cryptographic transformations. ENCRYPT and DECRYPT can use either shared-secret cryptography or public-key cryptography. We refer to the keys used for encryption as *encryption keys*.

With shared-secret cryptography, Alice and Bob share a key K that only they know. To keep a message M confidential, Alice computes ENCRYPT (M, K) and sends the resulting ciphertext C to Bob. If the encrypting box is good, an adversary will not to be able to get any use out of the ciphertext. Bob computes DECRYPT (C, K), which will recover the plaintext form of M. Bob can send a reply to Alice using exactly the same system with the same key. (Of course, Bob could also send the reply with a different key, as long as that different key is also shared with Alice.)

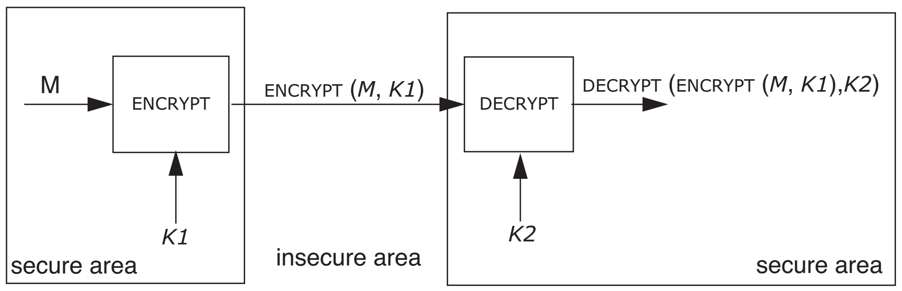


FIGURE 11.6

Providing confidentiality using ENCRYPT and DECRYPT over untrusted networks.

With public-key cryptography, Alice and Bob do *not* have to share a secret to achieve confidentiality for communication. Suppose Bob has a private and public key pair (K_{Bpriv}, K_{Bpub}), where K_{Bpriv} is Bob's private key and K_{Bpub} is Bob's public key. Bob gives his public key to Alice through an existing channel; this channel does not have to be secure, but it does have to provide authentication: Alice needs to know for sure that this key is really Bob's key.

Given Bob's public key (K_{Bpub}), Alice can compute ENCRYPT (M, K_{Bpub}) and send the encrypted message over an insecure network. Only Bob can read this message, since he is the only person who has the secret key that can decrypt her ciphertext message. Thus, using encryption, Alice can ensure that her communication with Bob stays confidential.

To achieve confidential communication in the opposite direction (from Bob to Alice), we need an additional set of keys, a K_{Apub} and K_{Apriv} for Alice, and Bob needs to learn Alice's public key.

11.4.2 Properties of ENCRYPT and DECRYPT

For both the shared-key and public-key encryption systems, the procedures ENCRYPT and DECRYPT should have the following properties. It should be easy to compute:

- $C \leftarrow$ ENCRYPT (M, K_1)
- $M' \leftarrow$ DECRYPT (C, K_2)

and the result should be that $M = M'$.

The implementation of ENCRYPT and DECRYPT should withstand the following attacks:

1. Ciphertext-only attack. In this attack, the primary information available to the adversary is examples of ciphertext and the algorithms for ENCRYPT and DECRYPT. Redundancy or repeated patterns in the original message may show through even in the ciphertext, allowing an adversary to reconstruct the plaintext. In an open

design the adversary knows the algorithms for ENCRYPT and DECRYPT, and thus the adversary may also be able to mount a brute-force attack by trying all possible keys.

More precisely, when using shared-secret cryptography, the following property must hold:

- Given ENCRYPT and DECRYPT, and some examples of C, it should be difficult for an adversary to reconstruct K or compute M.

When using public-key cryptography, the corresponding property holds:

- Given ENCRYPT and DECRYPT, some examples of C, and assuming an adversary knows K_1 (which is public), it should be difficult for the adversary to compute either the secret key K_2 or M.

2. Known-plaintext attack. The adversary has access to the ciphertext C and also to the plaintext M corresponding to at least some of the ciphertext C. For instance, a message may contain standard headers or a piece of predictable plaintext, which may help an adversary figure out the key and then recover the rest of the plaintext.

3. Chosen-plaintext attack. The adversary has access to ciphertext C that corresponds to plaintext M that the adversary has chosen. For instance, the adversary may convince you to send an encrypted message containing some data chosen by the adversary, with the goal of learning information about your transforming system, which may allow the adversary to more easily discover the key. As a special case, the adversary may be able in real time to choose the plaintext M based on ciphertext C just transmitted. This variant is known as an adaptive attack.

   A common design mistake is to unintentionally admit an adaptive attack by providing a service that happily encrypts any input it receives. This service is known as an *oracle* and it may greatly simplify the effort required by an adversary to crack the cryptographic transformation. For example, consider the following adaptive chosen-plaintext attack on the encryption of packets in WiFi wireless networks. The adversary sends a carefully-crafted packet from the Internet addressed to some node on the WiFi network. The network will encrypt and broadcast that packet over the air, where the adversary can intercept the ciphertext, study it, and immediately choose more plaintext to send in another packet. Researchers used this attack as one way of breaking the design of the security of WiFi Wired Equivalent Privacy (WEP)*.

4. Chosen-ciphertext attack. The adversary might be able to select a ciphertext C and then observe the M' that results when the recipient decrypts C. Again, an adversary may be able to mount an adaptive chosen-ciphertext attack.

* N. Borisov, I. Goldberg, and D. Wagner, *Intercepting mobile communications: the insecurity of 802.11*, MOBICOM '01, Rome, Italy, July 2001.

Section 11.8 describes cryptographic implementations of ENCRYPT and DECRYPT that provide protection against these attacks. A designer can increase the work factor for an adversary by increasing the key length. A typical key length used in the field is 1,024 bits.

The window of validity of ENCRYPT and DECRYPT is the minimum of the time to compromise of the underlying cryptographic transformation, the time to try all keys, and the time to compromise the key itself. When considering what implementation of ENCRYPT and DECRYPT to use, it is important to understand the required window of validity. It is likely that the window of validity required for encrypting protocol messages between a client and a server is smaller than the window of validity required for encrypting long-term file storage. A protocol message that must be private just for the duration of a conversation might be adequately protected by an cryptographic transformation that can be compromised with, say, one year of effort. On the other hand, if the period of time for which a file must be protected is greater than the window of validity of a particular cryptographic system, the designer may have to consider additional mechanisms, such as multiple encryptions with different keys.

11.4.3 Achieving both Confidentiality and Authentication

Confidentiality and message authentication can be combined in several ways:

- For confidentiality only, Alice just encrypts the message.
- For authentication only, Alice just signs the message.
- For both confidentiality and authentication, Alice first encrypts and then signs the encrypted message (i.e., SIGN (ENCRYPT (M, K_{encrypt}), K_{sign})), or, the other way around. (If good implementations of SIGN and VERIFY are used, it doesn't matter for correctness in which order the operations are applied.)

The first option, confidentiality without authentication, is unusual. After all, what is the purpose of keeping information confidential if the receiver cannot tell if the message has been changed? Therefore, if confidentiality is required, one also provides authentication.

The second option is common. Much data is public (e.g., routing updates, stock updates, etc.), but it is important to know its origin and integrity. In fact, it is easy to argue the default should be that all messages are at least authenticated.

For the third option, the keys used for authentication and confidentiality are typically different. The sender authenticates with an authentication key, and encrypts with a encryption key. The receiver would use the appropriate corresponding keys to decrypt and to verify the received message. The reason to use different keys is that the key is a bit pattern, and using the same bit pattern as input to two cryptographic operations on the same message is risky because a clever cryptanalyst may be able to discover a way of exploiting the repetition. Section 11.8 gives an example of exploitation of repetition in an otherwise unbreakable encryption system known as the one-time pad. Problem set *44* and *46* also explores one-time pads to setup a secure communicaiton channel.

In addition to using the appropriate keys, there are other security hazards. For example, *M* should have identified explicitly the communicating parties. When Alice sends a message to Bob, she should include in the message the names of Alice and Bob to avoid impersonation attacks. Failure to follow this *explicitness principle* can create security problems, as we will see in Section 11.5.

11.4.4 Can Encryption be Used for Authentication?

As specified, ENCRYPT and DECRYPT don't protect against an adversary modifying *M* and one must SIGN and VERIFY for integrity. With some implementations, however, a recipient of an encrypted message can be confident not only of its confidentiality, but also of its authenticity. From this observation arose the misleading intuition that decrypting a message and finding something recognizable inside is an effective way of establishing the authenticity of that message. The intuition is based on the claim that if only the sender is able to encrypt the message, and the message contains at least one component that the recipient expected the sender to include, then the sender must have been the source of the message.

The problem with this intuition is that as a general rule, the claim is wrong. It depends on using a cryptographic system that links all of the ciphertext of the message in such a way that it cannot be sliced apart and respliced, perhaps with components from other messages between the same two parties and using the same cryptographic key. As a result, it is non-trivial to establish that a system based on the claim is secure even in the cases in which it is. Many protocols that have been published and later found to be defective were designed using that incorrect intuition. Those protocols using this approach that are secure require much effort to establish the necessary conditions, and it is remarkably hard to make a compelling argument that they are secure; the argument typically depends on the exact order of fields in messages, combined with some particular properties of the underlying cryptographic operations.

Therefore, in this book we treat message confidentiality and authenticity as two separate goals that are implemented independently of each other. Although both confidentiality and authenticity rely in their implementation on cryptography, they use the cryptographic operations in different ways. As explained in Section 11.8, the shared-secret AES cryptographic transformation, for example, isn't by itself suitable for *either* signing or encrypting; it needs to be surrounded by various cipher-feedback mechanisms, and the mechanisms that are good for encrypting are generally somewhat different from those that are good for signing. Similarly, when RSA, a public-key cryptographic transformation, is used for signing, it is usually preceded by hashing the message to be signed, rather than applying RSA directly to the message; a failure to hash can lead to a security blunder.

A recent paper[*] on the topic on the order of authentication and encrypting suggests that first encrypting and then computing an authentication tag may cover up certain weaknesses in some implementations of the encrypting primitives. Also, cryptographic transformations have been proposed that perform the transformation for encrypting and computing an authentication tag in a single pass over the message, saving time compared to first encrypting and then computing an authentication tag. Cryptography is a developing area, and the last word on this topic has not been said; interested readers should check out the proceedings of the conferences on cryptography. For the rest of the book, however, the reader can think of message authentication and confidentiality as two separate, orthogonal concepts.

11.5 Security Protocols

In the previous sections we discovered a need for protecting a principal's password when authenticating to a remote service, a need for distributing keys securely, etc. Security protocols can achieve those objectives. A *security protocol* is an exchange of messages designed to allow mutually-distrustful parties to achieve an objective. Security protocols often use cryptographic techniques to achieve the objective. Other example objectives include: electronic voting, postage stamps for e-mail, anonymous e-mail, and electronic cash for micropayments.

In a security protocol with two parties, the pattern is generally a back-and-forth pattern. Some security protocols involve more than two parties in which case the pattern may be more complicated. For example, key distribution usually involves at least three parties (two principals and a trusted third party). A credit-purchase on the Internet is likely to involve many more principals than three (a client, an Internet shop, a credit card company, and one or more trusted third parties) and thus require four or more messages.

The difference between the network protocols discussed in Chapter 7[on-line] and the security ones is that standard networking protocols assume that the communicating parties cooperate and trust each other. In designing security protocols we instead assume that some parties in the protocol may be adversaries and also that there may be an outside party attacking the protocol.

11.5.1 Example: Key Distribution

To illustrate the need for security protocols, let's study two protocols for key distribution. In Section 11.3.6, we have already seen that distributing keys is based on a name discovery protocol, which starts with trusted physical delivery. So, let's assume that Alice has met Charles in person, and Charles has met Bob in person. The question then is: is there a protocol such that Alice and Bob, who have never met, can exchange keys securely

[*] Hugo Krawczyk, *The Order of Encryption and Authentication for Protecting Communications (or: How Secure is SSL?)*, Advances in Cryptology (Springer LNCS 2139), 2001, pages 310–331.

over an untrusted network? This section introduces the basic approach and subsequent sections work out the approach in detail.

The public-key case is simpler, so we treat it first. Alice and Bob already know Charles's public key (since they have met in person), and Charles knows each of Alice and Bob's public keys. If Alice and Bob both trust Charles, then Alice and Bob can exchange keys through Charles.

Alice sends a message to Charles (it does not need to be either encrypted or signed), asking:

1. Alice ⇒ Charles: {"Please give me keys for Bob"}

The message content is the string "Please, give me keys for Bob". The source address is "Alice" and the destination address is "Charles." When Charles receives this message from Alice, he cannot be certain that if the message came from Alice, since the source and destination fields of Chapter 7[on-line] are not authenticated.
For this message, Charles doesn't really care who sent it, so he replies:

1. Charles ⇒ Alice: {"To communicate with Bob, use public key K_{Bpub}."}$_{Cpriv}$

The notation $\{M\}_k$ denotes signing a message M with key k. In this example, the message is signed with Charles's private authentication key. This signed message to Alice includes the content of the message as well as the authentication tag. When Alice receives this message, she can tell from the fact that this message verifies with Charles's public key that the message actually came from Charles.

Of course, these messages would normally not be written in English, but in some machine-readable semantically equivalent format. For expository and design purposes, however, it is useful to write down the meaning of each message in English. Writing down the meaning of a message in English helps make apparent oversights, such as omitting the name of the intended recipient. This method is an example of the design principle *be explicit*.

To illustrate that problems can be caused by of lack of explicitness, suppose that the previous message 2 were:

2'. Charles ⇒ Alice: {"Use public key K_{Bpub}."}$_{Cpriv}$

If Alice receives this message, she can verify with Charles's public key that Charles sent the message, but Alice is unable to tell whose public key K_{Bpub} is. An adversary Lucifer, whom Charles has met, but doesn't know that he is bad, might use this lack of explicitness as follows. First, Lucifer asks Charles for Lucifer's public key, and Charles replies:

2'. Charles ⇒ Lucifer: {"Use public key K_{Lpub}."}$_{Cpriv}$

Lucifer saves the reply, which is signed by Charles. Later when Alice asks Charles for Bob's public key, Lucifer replaces Charles's response with the saved reply. Alice receives the message:

2'. Someone ⇒ Alice: {"Use public key K_{Lpub}."}$_{Cpriv}$

From looking at the source address (Someone), she *cannot* be certain where message 2' came from. The source and destination fields of Chapter 7[on-line] are not authenticated, so Lucifer can replace the source address with Charles's source address. This change won't affect the routing of the message, since the destination address is the only address needed to route the message to Alice. Since the source address cannot be trusted, the message itself has to tell her where it came from, and message 2' says that it came from Charles because it is signed by Charles.

Believing that this message came from Charles, Alice will think that this message is Charles's response to her request for Bob's key. Thus, Alice will incorrectly conclude that K_{Lpub} is Bob's public key. If Lucifer can intercept Alice's subsequent messages to Bob, Lucifer can pretend to be Bob, since Alice believes that Bob's public key is K_{Lpub} and Lucifer has K_{Lpriv}. This attack would be impossible with message 2 because Alice would notice that it was Lucifer's, rather than Bob's key.

Returning to the correct protocol using message 2 rather than message 2', after receiving Charles's reply, Alice can then sign (with her own private key, which she already knows) and encrypt (with Bob's public key, which she just learned from Charles) any message that she wishes to send to Bob. The reply can be handled symmetrically, after Bob obtains Alice's public key from Charles in a similar manner.

Alice and Bob are trusting Charles to correctly distribute their public keys for them. Charles's message (2) *must* be signed, so that Alice knows that it really came from Charles, instead of being forged by an adversary. Since we presumed that Alice already had Charles's public key, she can verify Charles' signature on message (2).

Bob cannot send Alice his public key over an insecure channel, even if he signs it. The reason is that she cannot believe a message signed by an unknown key asserting its own identity. But a message like (2) signed by Charles can be believed by Alice, if she trusts Charles to be careful about such things. Such a message is called a *certificate*: it contains Bob's name and public key, certifying the binding between Bob and his key. Bob himself could have sent Alice the certificate Charles signed, if he had the foresight to have already obtained a copy of that certificate from Charles. In this protocol Charles plays the role of a *certificate authority (CA)*. The idea of using the signature of a trusted authority to bind a public key to a principal identifier and calling the result a certificate was invented in Loren Kohnfelder's 1978 M.I.T. bachelor's thesis.

When shared-secret instead of public-key cryptography is being used, we assume that Alice and Charles have pre-established a shared-secret authentication key Ak_{AC} and a shared-secret encryption key Ek_{AC}, and that Bob and Charles have similarly pre-established a shared-secret authentication key Ak_{BC} and a shared-secret encryption key Ek_{BC}. Alice begins by sending a message to Charles (again, it does not need to be encrypted or signed):

1. Alice ⇒ Charles: {"Please, give me keys for Bob"}

Since shared-secret keys must be kept confidential, Charles must both sign *and* encrypt the response, using the two shared-secret keys Ak_{AC} and Ek_{AC}. Charles would reply to Alice:

2. Charles ⇒ Alice: $\{\text{“Use temporary authentication key } Ak_{AB} \text{ and temporary encryption key } Ek_{AB} \text{ to talk to Bob.”}\}^{Ek_{AC}}_{Ak_{AC}}$

The notation $\{M\}^{k}$ denotes encrypting message M with encryption key k. In this example, the message from Charles to Alice is signed by the shared-secret authentication key Ak_{AC} and encrypted with the shared-secret encryption key Ek_{AC}.

The keys Ak_{AB} and Ek_{AB} in Charles' reply are newly-generated random shared-secret keys. If Charles would have replied with Ak_{BC} and Ek_{BC} instead of newly-generated keys, then Alice would be able to impersonate Bob to Charles, or Charles to Bob.

It is also important is that message 2 is both authenticated with Charles' and Alice's shared key Ak_{AC} and encrypted with their shared Ek_{AC}. The k_{AC}'s are known only to Alice and Charles, so Alice can be confident that the message came from Charles and that only she and Charles know the k_{AB}'s. The next step is for Charles to tell Bob the keys:

3. Charles ⇒ Bob: $\{\text{“Use the temporary keys } Ak_{AB} \text{ and } Ek_{AB} \text{ to talk to Alice.”}\}^{Ek_{BC}}_{Ak_{BC}}$

This message is both authenticated with key Ak_{BC} and encrypted with key Ek_{BC}, which are known only to Charles and Bob, so Bob can be confident that the message came from Charles and that no one else but Alice and Charles know k_{AB}'s.

From then on, Alice and Bob can communicate using the temporary key Ak_{AB} to authenticate and the temporary key Ek_{AB} to encrypt their messages. Charles should immediately erase any memory he has of the two temporary keys k_{AB}'s. In such an arrangement, Charles is usually said to be acting as a *key distribution center* (or KDC). The idea of a shared-secret key distribution center was developed in classified military circles and first revealed to the public in a 1973 paper by Dennis Branstad*. In the academic community it first showed up in a paper by Needham and Schroeder†.

A common variation is for Charles to include message (3) to Bob as an attachment to his reply (2) to Alice; Alice can then forward this attachment to Bob along with her first real message to him. Since message (3) is both authenticated and encrypted, Alice is simply acting as an additional, more convenient forwarding point so that Bob does not have to match up messages arriving from different places.

Not all key distribution and authentication protocols separate authentication and encryption (e.g., see Sidebar 11.6[on-line] about Kerberos); they instead accomplish authentication by using carefully-crafted encrypting, with just one shared key per participant. Although having fewer keys seems superficially simpler, it is then harder to establish the correctness of the protocols. It is simpler to use the divide-and-conquer strategy: the additional overhead of having two separate keys for authentication and encrypting is well worth the simplicity and ease of establishing correctness of the overall design.

* Dennis K. Branstad. Security aspects of computer networks.American Institute of Aeronautics and Astronautics Computer Network Systems Conference, paper 73–427 (April, 1973).

† Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. Communications of the ACM 21, 12 (December, 1978), pages 993–999.

Sidebar 11.6: The Kerberos authentication system Kerberos[*] was developed in the late 1980's for project Athena, a network of engineering workstations and servers designed to support undergraduate education at M.I.T.[†] The first version in wide-spread use was Version 4, which is described here in simplified form; newer versions of Kerberos improve and extend Version 4 in various ways, but the general approach hasn't changed much.

A Kerberos service implements a unique identifier name space, called a *realm*, in which each name of the name space is the principal identifier of either a network service or an individual user. Kerberos also allows a confederation of Kerberos services belonging to different organizations to implement a name space of realms. Principal names are of the form "alice@Scholarly.edu", a principal identifier followed by the name of the realm to which that principal belongs. Kerberos principal identifiers are case-sensitive, some consequences of which were discussed in Section 3.3.4. Users and services are connected by an open, untrusted network. The goal of Kerberos is to provide two-way authentication between a user and a network service securely under the threat of adversaries.

A user authenticates the user's identity and logs on to a realm using a shared-secret protocol with the realm's Kerberos Key Distribution Service (KKDS). Kerberos derives the shared-secret key by cryptographically hashing a user-chosen password. During the name-discovery step (e.g., a physical rendezvous with its administrator), the Kerberos service learns the principal identifier for the user and the shared secret. When logging on, the user sends its principal identifier to KKDS and asks it for authentication information to talk to service *S*:

$$\text{Alice} \Rightarrow \text{KDDS: } \{\text{“alice@Scholarly.edu”}, S, T_{\text{current}}\}$$

and the service responds with a *ticket* identifying the user:

$$\text{KKDS} \Rightarrow \text{Alice: } \{K_{\text{tmp}}, S, \textit{Lifetime}, T_{\text{current}}, \textit{ticket}\}^{K\text{alice}}$$

The service encrypts this response with the user's shared secret. The verification step occurs when the user decrypts the encrypted response. If T_{current} and *S* in the response match with the values in the request, then Kerberos considers the response authentic, and uses the information in the decrypted response to authenticate the user to S. If the user does not posses the key (the hashed password) that decrypts the response, the information inside the response is worthless.

The ticket is a kind of certificate; it binds the user name to a temporary key for use during one session with service *S*. Kerberos includes the following information in the ticket:

$$\textit{ticket} = \{K_{\text{tmp}}, \text{“alice@Scholarly.edu”}, S, T_{\text{current}}, \textit{Lifetime}\}^{Ks}$$

(*Sidebar continues*)

[*] S[teven] P. Miller, B. C[lifford] Neuman, J[effrey] I. Schiller, and J[erome] H. Saltzer. Kerberos authentication and authorization system. Section E.2.1 of Athena Technical Plan, M.I.T. Project Athena, October 27, 1988.

[†] George A. Champine. M.I.T. Project Athena: A Model for Distributed Campus Computing. Digital Press, Bedford, Massachusetts, 1991. ISBN 1–55558–072–6. 282 pages.

The temporary key K_{tmp} is to allow a user to establish a continued chain of authentication without having to go back to KKDS for each message exchange. The ticket contains a time stamp, the principal identifier of the user, the principal identifier of the service, and a second copy of the temporary key, all encrypted in the key shared between the KDDS and the service S (e.g., a network file service).*

Kerberos includes in a request to a Kerberos-mediated network service the ticket identifying the user. When the service receives a request, it authenticates the ticket using the information in the ticket. It decrypts the ticket, checks that the timestamp inside is recent and that its own principal identifier is accurate. If the ticket passes these tests, the service believes that it has the authentic principal identifier of the requesting user and the Kerberos protocol is complete. Knowing the user's principal identifier, the service can then apply its own authorization system to establish that the user has permission to perform the requested operation.

A user can perform cross-realm authentication by applying the basic Kerberos protocol twice: first obtain a ticket from a local KDC for the other realm's KDC, and then using that ticket obtain a second ticket from the remote realm's KDC for a service in the remote realm. For cross-realm authentication to work, there are two prerequisites: (1) initialization: the two realms must have previously agreed upon a shared-secret key between the realms and (2) name discovery: the user and service must each know the other's principal identifier and realm name.

Versions 4 and 5 of Kerberos are in widespread use outside of M.I.T. (e.g., they were adopted by Microsoft). They are based on formerly classified key distribution principles first publicly described in a paper by Branstad and are strengthened versions of a protocol described by Needham and Schroeder (mentioned on page 11–57). These protocols don't separate authentication from confidentiality. They instead rely on clever use of cryptographic operations to achieve both goals. As explained in Section 11.4.4 on page 11–53, this property makes the protocols difficult to analyze.

* This description is a simplified version of the Kerberos protocol. One important omission is that the ticket a user receives as a result of successfully logging in is actually one for a ticket-granting service (TGS), from which the user can obtain tickets for other services. TGS provides what is sometimes called a *single login or single sign-on* system, meaning that a user needs to present a password only once to use several different network services.

For performance reasons, computer systems typically use public-key systems for distributing and authenticating keys and shared-secret systems for sending messages in an authenticated and confidential manner. The operations in public-key systems (e.g., raising to an exponent) are more expensive to compute than the operations in shared-secret cryptography (e.g., table lookups and computing several XORS). Thus, a session between two parties typically follows two steps:

1. At the start of the session use public-key cryptography to authenticate each party to the other and to exchange new, temporary, shared-secret keys;

2. Authenticate and encrypt subsequent messages in the session using the temporary shared-secret keys exchanged in step 1.

Using this approach, only the first few messages require computationally expensive operations, while all subsequent messages require only inexpensive operations.

One might wonder why it is not possible to the design the ultimate key distribution protocol once, get it right, and be done with it. In practice, there is no single protocol that will do. Some protocols are optimized to minimize the number of messages, others are optimized to minimize the cost of cryptographic operations, or to avoid the need to trust a third party. Yet others must work when the communicating parties are not both on-line at the same time (e.g., e-mail), provide only one-way authentication, or require client anonymity. Some protocols, such as protocols for authenticating principals using passwords, require other properties than basic confidentiality and authentication: for example, such a protocol must ensure that the password is sent only once per session (see Section 11.2).

11.5.2 Designing Security Protocols

Security protocols are vulnerable to several attacks in addition to the ones described in Section 11.3.4 (page 11–41) and 11.4.2 (page 11–50) on the underlying cryptographic transformations. The new attacks to protect against fall in the following categories:

- *Known-key attacks.* An adversary obtains some key used previously and then uses this information to determine new keys.
- *Replay attacks*. An adversary records parts of a session and replays them later, hoping that the recipient treats the replayed messages as new messages. These replayed messages might trick the recipient into taking an unintended action or divulging useful information to the adversary.
- *Impersonation attacks.* An adversary impersonates one of the other principals in the protocol. A common version of this attack is the person-in-the-middle attack, where an adversary relays messages between two principals, impersonating each principal to the other, reading the messages as they go by.
- *Reflection attacks*. An adversary records parts of a session and replays it to the party that originally sent it. Protocols that use shared-secret keys are sometimes vulnerable to this special kind of replay attack.

The security requirements for a security protocol go beyond simple confidentiality and authentication. Consider a replay attack. Even though the adversary may not know what the replayed messages say (because they are encrypted), and even though the adversary may not be able to forge new legitimate messages (because the adversary doesn't have the keys used to compute authentication tags), the adversary may be able to cause mischief or damage by replaying old messages. The (duplicate) replayed messages may well be accepted as genuine by the legitimate participants, since the authentication tag will verify correctly.

The participants are thus interested not only in confidentiality and authentication, but also in the three following properties:

- *Freshness*. Does this message belong to this instance of this protocol, or is it a replay from a previous run of this protocol?
- *Explicitness*. Is this message really a member of this run of the protocol, or is it copied from an run of another protocol with an entirely different function and different participants?
- *Forward secrecy*. Does this protocol guarantee that if a key is compromised that confidential information communicated in the past stays confidential? A protocol has forward secrecy if it doesn't reveal, even to its participants, any information from previous uses of that protocol.

We study techniques to ensure freshness and explicitness; forward secrecy can be accomplished by using different temporary keys in each protocol instance and changing keys periodically. A brief summary of standard approaches to ensure freshness and explicitness include:

- Ensure that each message contains a nonce (a value, perhaps a counter value, serial number, or a timestamp, that will never again be used for any other message in this protocol), and require that a reply to a message include the nonce of the message being replied to, as well as its own new nonce value. The receiver and sender of course have to remember previously used nonces to detect duplicates. The nonce technique provides freshness and helps foil replay attacks.

- Ensure that each message explicitly contain the name of the sender of the message and of the intended recipient of the message. Protocols that omit this information, and that use shared-secret keys for authentication, are sometimes vulnerable to reflection attacks, as we saw in the example protocol in Section 11.5.1. Including names provides explicitness and helps foil impersonation and reflection attacks.

- Ensure that each message specifies the security protocol being followed, the version number of that protocol, and the message number within this instance of that protocol. If such information is omitted, a message from one protocol may be replayed during another protocol and, if accepted as legitimate there, cause damage. Including all protocol context in the message provides explicitness and helps foil replay attacks.

The explicitness property is an example of the *be explicit* design principle: ensure that each message be totally explicit about what it means. If the content of a message is not completely explicit, but instead its interpretation depends on its context, an adversary might be able to trick a receiver into interpreting the message in a different context and break the protocol. Leaving the names of the participants out of the message is a violation of this principle.

When a protocol designer applies these techniques, the key-distribution protocol of Section 11.5.1 might look more like:

1 Alice ⇒ Charles: {"This is message number one of the "Get Public Key" protocol, version 1.0. This message is sent by Alice and intended for Charles. This message was sent at 11:03:04.114 on 3 March 1999. The nonce for this message is 1456255797824510. What is the public key of Bob?"}$_{\text{Apriv}}$

2 Charles ⇒ Alice: {"This is message number two of the "Get Public Key" protocol, version 1.0. This message is sent by Charles and intended for Alice. This message was sent at 11:03:33.004 on 3 March 1999. This is a reply to the message with nonce 1456255797824510. The nonce for this message is 5762334091147624. Bob's public key is (...)."}$_{\text{Cpriv}}$

In addition, the protocol would specify how to marshal and unmarshal the different fields of the messages so that an adversary cannot trick the receiver into unmarshaling the message incorrectly.

In contrast to the public-key protocol described above, the first message in this protocol is signed. Charles can now verify that the information included in the message came indeed from Alice and hasn't been tampered with. Now Charles can, for example, log who is asking for Bob's public key.

This protocol is almost certainly overdesigned, but it is hard to be confident about what can safely be dropped from a protocol. It is surprisingly easy to underdesign a protocol and leave security loopholes. The protocol may still seem to "work OK" in the field, until the loophole is exploited by an adversary. Whether a protocol "seems to work OK" for the legitimate participants following the protocol is an altogether different question from whether an adversary can successfully attack the protocol. Testing the security of a protocol involves trying to attack it or trying to prove it secure, not just implementing it and seeing if the legitimate participants can successfully communicate with it. Applying the safety net approach to security protocols tells us to overdesign protocols instead of underdesign.

Some applications require properties beyond freshness, explicitness, and forward secrecy. For example, a service way want to make sure that a single client cannot flood the service with messages, overloading the service and making it unresponsive to legitimate clients. One approach to provide this property is for the service to make it expensive for the client to generate legitimate protocol messages. A service could achieve this by challenging the client to perform an expensive computation (e.g., computing the inverse of a cryptographic function) before accepting any messages from the client. Yet other applications may require that more than one party be involved (e.g., a voting application). As in designing cryptographic primitives, designing security protocols is difficult and should be left to experts. The rest of this section presents some common security protocol problems that appear in computer systems and shows how one can reason about them. Problem set *43* explores how to use the signing and encryption primitives to achieve some simple security objectives.

11.5.3 Authentication Protocols

To illustrate the issues in designing security protocols, we will look at two simple authentication protocols. The second protocol uses a challenge and a response, which is an idea found in many security protocols. These protocols also provide the motivation for other protocols that we will discuss in subsequent sections.

A simple example of an authentication protocol is the one for opening a garage door remotely while driving up to the garage. This application doesn't require strong security properties (the adversary can always open the garage with a crowbar) but must be low cost. We want a protocol that can be implemented inexpensively so that the remote can be small, cheap, and battery-powered. For example, we want a protocol that involves only one-way communication, so that the remote control needs only a transmitter. In addition, the protocol should avoid complex operations so that the remote control can use an inexpensive processor.

The parties in the protocol are the remote control, a receiving device (the receiver), and an adversary. The remote control uses a wireless radio to transmit "open" messages to a receiver, which opens the garage door if an authorized remote control sends the message. The goal of the adversary is to open the garage without the permission of the owner of the garage.

The adversary is able to listen, replay, and modify the messages that the remote control sends to the receiver over the wireless medium. Of course, the adversary can also try to modify the remote control, but we assume that stealing the remote control is at least as hard as breaking into the garage physically, in which case there isn't much need to also subvert the remote control protocol.

The basic idea behind the protocol is for the receiver and the remote control to share a secret. The remote control sends the secret to the receiver and if it matches the receiver's secret, then the receiver opens the garage. If the adversary doesn't know the secret, then the adversary cannot open the garage. Of course, if the secret is transmitted over the air in clear text, the adversary can easily learn the secret, so we need to refine this basic idea.

A lightweight but correct protocol is as follows. At initialization, the remote control and receiver agree on some random number, which functions as a shared-secret key, and a random number, which is an initial counter value. When the remote control is pressed, it sends the following message:

remote ⇒ receiver: {*counter*, HASH(*key*, *counter*)},

and increments the counter.

When receiving the message, the receiver performs the following operations:

1. verify hash: compute HASH(*key*, *counter*) and compare result with the one in message
2. if hash verifies, then increment counter and open garage. If not, do nothing.

Because the holder of the remote control may have pressed the remote while out of radio range of the receiver, the receiver generally tries successive values of counter between its previous values N and, e.g., N+100 in step 1. If it finds that one of the values works, it resets the counter to that value and opens the garage.

This protocol meets our basic requirements. It doesn't involve two-way communication. It does involve computing a hash but strong, inexpensive-to-compute hashes are readily available in the literature. Most important, the protocol is likely to provide a good enough level of security for this application.

The adversary cannot easily construct a message with the appropriate hash because the adversary doesn't know the shared-secret key. The adversary could try all possible values for the hash output (or all possible keys, if the keys are shorter than the hash output). If the hash output and key are sufficiently long, then this brute-force attack would take a long time. In addition, if necessary, the protocol could periodically re-initialize the key and counter.

The protocol is not perfect. For example, it has a replay attack. Suppose an impatient user presses the button on the remote control twice in close succession, the receiver responds to the first signal and doesn't hear the second signal. An adversary who happens to be recording the signals at the time can notice the two signals and guess that replaying the recording of the second signal may open the garage door, at least until the next time that legitimate user again uses the remote control. This weakness is probably acceptable.

The adversary can also launch a denial-of-service attack on the protocol (e.g., by jamming the radio signal remotely). The adversary, however, could also wreck the garage's door physically, which is simpler. The owner can also always get out of the car, walk to the garage, and use a physical key, so there is little motivation to deny access to the remote control.

Protocols such as the one described above are used in practice. For example, the Chamberlain garage door opener[*] uses a similar protocol with an extremely simple hash function (multiplication by 3 in a finite field) and it computes the hash over the previous hash, instead of over the counter and key. The simple hash probably provides a little less security but it has the advantage that is cheap to implement. Other vendors seem to use similar protocols, but it is difficult to confirm because this industry has a practice of keeping its proprietary protocols secret, perhaps hoping to increase security through obscurity, which violates the *open design* principle and historically hasn't worked.

A version that is more secure than the garage-door protocol is used for authentication of users who want to download their e-mail from an e-mail service. Protocols for this application can assume two-way communication and exploit the idea of a challenge and a response. One widely used *challenge-response protocol* is the following[†]:

1 *Initialization*. M_1: Client ⇒ Server: (Opens a TCP connection)
2 *Challenge*. M_2: Server ⇒ Client: {"This is server *S* at 9:35:20.00165 EDT, 22

[*] Chamberlain Group, Inc. v. Skylink Techs., Inc., 292 F. Supp. 2d 1040 (N.D. Ill. 2003); aff'd 381 F.3d 1178 (U.S. App. 2004)

[†] Myers and M. Rose, *Post Office Protocol Version 3*, Internet Engineering Task Force Request For Comments (RFC) 1939, May 1996.

September 2006.”}
3 *Response*. M_3: Client ⇒ Server: {“This is user U and the hash of M_2 and U’s password is:” HASH{M_2, U’s password}”}

The server, which has its own copy of the secret password associated with user U, does its own calculation of HASH{M_2, U’s password}, and compares the result with the second field of M_3. If they match, it considers the authentication successful and it proceeds to download the e-mail messages.

The protocol isn't vulnerable to the person-in-the-middle attack of the garage protocol because the date and time in M_2 functions as a nonce, which is included in the hash of M_3. But addressing the person-in-the-middle attack requires two-way communication, which couldn't be used by the garage door opener.

Although this protocol is a step up over the garage door protocol, it has weaknesses too. It is vulnerable to brute-force attacks. The adversary can learn the user name *U* from M_3. Then, later the adversary can connect to the mail server, receive M_2, guess a password for *U*, and see if the attempt is successful. Although each guess takes one round of the protocol and leaves an audit trail on the server, this might not stop a determined adversary.

A related weakness is that the protocol doesn't authenticate the server *S*, so the adversary can impersonate the server. The adversary tricks the client in connecting to a machine that the adversary controls (e.g., by spoofing a DNS response for the name *S*). When the client connects, the adversary sends M_2, and receives a correct M_3. Now the adversary can do an off-line brute-force attack on the user's password, without leaving an audit trail. The adversary can also provide the client with bogus e-mail.

These weaknesses can be addressed. For example, instead of sending messages in the clear over a TCP connection, the protocol could set up a confidential, authenticated connection to the server using SSL/TLS (see Section 11.10). Then, the client and server can run the challenge-response protocol over this connection. The server can also send the e-mail messages over the connection so that they are protected too. SSL/TLS authenticates all messages between a client and server and sends them encrypted. In addition, the client can require that the server provides a certificate with which the client can verify that the server is authentic. This approach could be further improved by using a client certificate instead of using U's password, which is a weak secret and vulnerable to dictionary attacks. Using SSL/TLS (either with or without client certificate) is common practice today.

A challenge-response protocol is a valuable tool only if it is implemented correctly. For example, a version of the UW IMAP server (a mail server that speaks the IMAP protocol and developed by the University of Washington) contained an implementation error that incorrectly specifies the conditions of successful authentication when using the challenge-response protocol described above*. After authenticating three times unsuccessfully using the challenge-response protocol, the server allowed the fourth attempt to

* United States Computer Emergency Readiness Team (US-CERT), *UW-imapd fails to properly authenticate users when using CRAM-MD5,* Vulnerability Note VU #702777*,* January 2005.

succeed; the intention was to fail the fourth attempt immediately, but the implementers got the condition wrong. This error allowed an adversary to successfully authenticate as any user on the server after three attempts. Such programming errors are all too often the reason why the security of a system can be broken.

11.5.4 An Incorrect Key Exchange Protocol

The challenge-response protocol over SSL/TLS assumes SSL/TLS can set up a confidential and authenticated channel, which requires that the sender and receiver exchange keys securely over an untrusted network. It is possible to do such an exchange, but it must be done with care. We consider two different protocols for key exchange. The first protocol is incorrect, the second is (as far as anyone knows) correct. Both protocols attempt to achieve the same goal, namely for two parties to use a public-key system to negotiate a shared-secret key that can be used for encrypting. Both protocols have been published in the computer science literature and systems incorporating them have been built.

In the first protocol, there are three parties: Alice, Bob, and a certificate authority (CA). The protocol is as follows:

1 Alice ⇒ CA: {"Give me certificates for Alice and Bob"}
2 CA ⇒ Alice: {"Here are the certificates:",
$\{\text{Alice}, A_{pub}, T\}_{CApriv}, \{\text{Bob}, B_{pub}, T\}_{CApriv}\}$

In the protocol, the CA returns certificates for Alice and Bob. The certificates bind the names to public keys. Each certificate contains a timestamp *T* for determining if the certificate is fresh. The certificates are signed by the CA.
Equipped with the certificates from the CA, Alice constructs an encrypted message for Bob:

3 Alice ⇒ Bob: {"Here is my certificate and a proposed key:",
$\{\text{Alice}, A_{pub}, T\}_{CApriv}, \{K_{AB}, T\}_{Apriv}\}^{Bpub}$

The message contains Alice's certificate and her proposal for a shared-secret key (K_{AB}). Bob can verify that A_{pub} belongs to Alice by checking the validity of the certificate using the CA's public key. The time-stamped shared-secret key proposed by Alice is signed by Alice, which Bob can verify using A_{pub}. The complete message is encrypted with Bob's public key. Thus, only Bob should be able to read K_{AB}.
Now Alice sends a message to Bob encrypted with K_{AB}:

4 Alice ⇒ Bob: {"Here is my message:", ... T}^{KAB}

Bob should be able to decrypt this message, once he has read message 3. So, what is the problem with this protocol? We suggest the reader pause for some time and try to discover the problem before continuing to read further. As a hint, note that Alice has signed only part of message 3 instead of the complete message. Recall that we should assume that some of the parties to the protocol may be adversaries.

The fact that there is a potential problem should be clear because the protocol fails the *be explicit* design principle. The essence of the protocol is part of message 3, which contains her proposal for a shared-secret key:

Alice ⇒ Bob: $\{K_{AB}, T\}_{Apriv}$

Alice tells Bob that K_{AB} is a good key for Alice and Bob at time T, but the names of Alice and Bob are missing from this part of message 3. The interpretation of this segment of the message is dependent on the context of the conversation. As a result, Bob can use this part of message 3 to masquerade as Alice. Bob can, for example, send Charles a claim that he is Alice and a proposal to use K_{AB} for encrypting messages.

Suppose Bob wants to impersonate Alice to Charles. Here is what Bob does:

1 Bob ⇒ CA: {"Give me the certificates for Bob and Charles"}

2 CA ⇒ Bob: {"Here are the certificates:", $\{Bob, B_{pub}, T'\}_{CApriv}$, $\{Charles, C_{pub}, T'\}_{CApriv}$}

3 Bob ⇒ Charles: {"Here is my certificate and a proposed key":, $\{Alice, A_{pub}, T\}_{CApriv}$, $\{K_{AB}, T\}_{Apriv}$ $\}^{Cpub}$

Bob's message 3 is carefully crafted: he has placed Alice's certificate in the message (which he has from the conversation with Alice), and rather than proposing a new key, he has inserted the proposal, signed by Alice, to use K_{AB}, in the third component of the message.

Charles has no way of telling that Bob's message 3 didn't come from Alice. In fact, he thinks this message comes from Alice, since $\{K_{AB}, T\}$ is signed with Alice's private key. So he (erroneously) believes he has key that is shared with only Alice, but Bob has it too. Now Bob can send a message to Charles:

1 Bob ⇒ Charles: {"Please send me the secret business plan. Yours truly, Alice."$\}^{KAB}$

Charles believes that Alice sent this message because he thinks he received K_{AB} from Alice, so he will respond. Designing security protocols is tricky! It is not surprising that Denning and Sacco*, the designers of this protocol, overlooked this problem when they originally proposed this protocol.

An essential assumption of this attack is that the adversary (Bob) is trusted for something because Alice first has to have a conversation with Bob before Bob can masquerade as Alice. Once Alice has this conversation, Bob can use this trust as a toehold to obtain information he isn't supposed to know.

The problem arose because of lack of explicitness. In this protocol, the recipient can determine the intended use of K_{AB} (for communication between Alice and Bob) only by examining the context in which it appears, and Bob was able to undetectably change that context in a message to Charles.

Another problem with the protocol is its lack of integrity verification. An adversary can replace the string "Here is my certificate and a proposed key" with any other string

* D. Denning and G. Sacco. Timestamps in key distribution protocols. *Communication of the ACM 24,* 8, pages 533–535, 1981.

(e.g., "Here are the President's certificates") and the recipient would have no way of determining that this message is not part of the conversation. Although Bob didn't exploit this problem in his attack on Charles, it is a weakness in the protocol.

One way of repairing the protocol is to make sure that the recipient can always detect a change in context; that is, can always determine that the context is authentic. If Alice had signed the entire message 3, and Charles had verified that message 3 was properly signed, that would ensure that the context is authentic, and Bob would not have been able to masquerade as Alice. If we follow the *explicitness principle*, we should also change the protocol to make the key proposal itself explicit, by including the name of Alice and Bob with the key and timestamp and signing that entire block of data (i.e., $\{\text{Alice, Bob, } K_{AB}, T\}_{\text{Apriv}}$).

Making Alice and Bob explicit in the proposal for the key addresses the lack of explicitness, but doesn't address the lack of verifying the integrity of the explicit information. Only signing the entire message 3 addresses that problem.

You might wonder how it is possible that many people missed these seemingly obvious problems. The original protocol was designed in an era before the modular distinction between encrypting and signing was widely understood. It used encrypting of the entire message as an inexpensive way of authenticating the content; there are some cases where that trick works, but this is one where the trick failed. This example is another one of why the idea of obtaining authentication by encrypting is now considered to be a fundamentally bad practice.

11.5.5 Diffie-Hellman Key Exchange Protocol

The second protocol uses public-key cryptography to negotiate a shared-secret key. Before describing that protocol, it is important to understand the Diffie-Hellman key agreement protocol first. In 1976 Diffie and Hellman published the ground-breaking paper *New Directions in cryptography* [Suggestions for Further Reading 1.8.5], which proposed the first protocol that allows two users to exchange a shared-secret key over an untrusted network without any prior secrets. This paper opened the floodgates for new papers in cryptography. Although there was much work behind closed doors, between 1930 and 1975 few papers with significant technical contributions regarding cryptography were published in the open literature. Now there are several conferences on cryptography every year.

The Diffie-Hellman protocol has two public system parameters: p, a prime number, and g, the generator. The generator g is an integer less than p, with the property that for every number n between 1 and $p - 1$ inclusive, there is a power k of g such that $n = g^k$ (modulo p).

If Alice and Bob want to agree on a shared-secret key, they use p and g as follows. First, Alice generates a random value a and Bob generates a random value b. Both a and b are drawn from the set of integers $\{1, ..., p\text{-}2\}$. Alice sends to Bob: g^a (modulo p), and Bob sends to Alice: g^b (modulo p).

On receiving these messages, Alice computes $g^{ab} = (g^b)^a$ (modulo p), and Bob computes $g^{ba} = (g^a)^b$ (modulo p). Since $g^{ab} = g^{ba} = k$, Alice and Bob now have a shared-secret key k. An adversary hearing the messages exchanged between Alice and Bob cannot compute that value because the adversary doesn't know a and b; the adversary hears only p, g, g^a and g^b.

The protocol depends on the difficulty of calculating discrete logarithms in a finite field. It assumes that if p is sufficiently large, it is computationally infeasible to calculate the shared-secret key $k = g^{ab}$ (modulo p) given the two public values g^a (modulo p) and g^b (modulo p). It has been shown that breaking the Diffie-Hellman protocol is equivalent to computing discrete logarithms under certain assumptions.

Because the participants are not authenticated, the Diffie-Hellman protocol is vulnerable to a person-in-the-middle attack, similar to the one in Section 11.5.4. The importance of the Diffie-Hellman protocol is that it is the first example of a much more general cryptographic approach, namely the derivation of a shared-secret key from one party's public key and another party's private key. The second protocol is a specific instance of this approach, and addresses the weaknesses of the Denning-Sacco protocol.

11.5.6 A Key Exchange Protocol Using a Public-Key System

The second protocol uses a Diffie-Hellman-like exchange to set up keys for encrypting and authentication. The protocol is designed to set up a secure channel from a client to a service in the SFS self-certifying file system [Suggestions for Further Reading 11.4.3]; a similar protocol is also used in the Taos distributed operating system [Suggestions for Further Reading 11.3.2]. Web clients and servers use the more complex SSL/TLS protocol, which is described in Section 11.10.

The goal of the SFS protocol is to create a secure (authenticated and encrypted) connection between a client and a server that has a well-known public key. The client wants to be certain that it can authenticate the server and that all communication is confidential, but at the end of this protocol, the client will still be unauthenticated; an additional protocol will be required to identify and authenticate the client.

The general plan is to create two shared-secret nonce keys for each connection between a client and a server. One nonce key (K_{cs}) will be used for authentication and encryption of messages from client to server, the other (K_{sc}) for authentication and encryption of messages from server to client. Each of these nonce keys will be constructed using a Diffie-Hellman-like exchange in which the client and the server each contribute half of the key.

To start, the client fabricates two nonce half-keys, named $K_{c\text{-}cs}$ and $K_{c\text{-}sc}$, and also a nonce private and public key pair: T_{priv} and T_{pub}. T_{pub} is, in effect, a temporary name for this connection with this anonymous client.

The client sends to the service a request message to open a connection, containing T_{pub}, $K_{c\text{-}cs}$, and $K_{c\text{-}sc}$. The client encrypts the latter two with S_{pub}, the public key of the service:

Client ⇒ service: {"Here is a temporary public key T_{pub} and two key halves encrypted with your public key:", $\{K_{c\text{-}cs}, K_{c\text{-}sc}\}^{Spub}$}

The protocol encrypts $K_{c\text{-}cs}$, and $K_{c\text{-}sc}$ to protect against eavesdroppers. Since T_{pub} is a public key, there is no need to encrypt it.

The service can decrypt the keys proposed by the client with its private key, thus obtaining the three keys. At this point, the service has no idea who the client may be, and because the message may have been modified by an adversary, all it knows is that it has received three keys, which it calls T_{pub}', $K_{c\text{-}cs}'$ and $K_{c\text{-}sc}'$, and which may or may not be the same as the corresponding keys fabricated by the client. If they are the same, then $K_{c\text{-}cs}'$ and $K_{c\text{-}sc}'$ are shared secrets known only to the client and the server.

The service now fabricates two more nonce half-keys, named $K_{s\text{-}cs}$ and $K_{s\text{-}sc}$. It sends a response to the client, consisting of these two half-keys encrypted with T_{pub}':

Service ⇒ client: {"Here are two key halves encrypted with your temporary public key:", $\{K_{s\text{-}cs}, K_{s\text{-}sc}\}^{Tpub}$}

Unfortunately, even if $T_{pub}' = T_{pub}$, T_{pub} is public, so the client has no assurance that the response message came from the service; an adversary could have sent it or modified it. The client decrypts the message using T_{priv}, to obtain $K_{s\text{-}cs}'$ and $K_{s\text{-}sc}'$.

At this point in the protocol, the two parties have the following components in hand:

- Client: S_{pub}, T_{pub}, $K_{c\text{-}cs}$, $K_{c\text{-}sc}$, $K_{s\text{-}cs}'$, $K_{s\text{-}sc}'$
- Server: S_{pub}, T_{pub}', $K_{c\text{-}cs}'$, $K_{c\text{-}sc}'$, $K_{s\text{-}cs}$, $K_{s\text{-}sc}$

Now the client calculates

- $K_{cs} \leftarrow$ HASH ("client to server", S_{pub}, T_{pub}, $K_{s\text{-}cs}'$, $K_{c\text{-}cs}$)
- $K_{sc} \leftarrow$ HASH ("server to client", S_{pub}, T_{pub}, $K_{s\text{-}sc}'$, $K_{c\text{-}sc}$)

and the server calculates

- $K_{cs}' \leftarrow$ HASH ("client to server", S_{pub}, T_{pub}', $K_{s\text{-}cs}$, $K_{c\text{-}cs}'$)
- $K_{sc}' \leftarrow$ HASH ("server to client", S_{pub}, T_{pub}', $K_{s\text{-}sc}$, $K_{c\text{-}sc}'$)

If all has gone well (that is, there have been no attacks), $K_{cs} = K_{cs}'$ and $K_{sc} = K_{sc}'$.

At this point there are three concerns:

1. An adversary may have replaced one or more components in such a way that the two parties do not have matching sets. If so, and assuming that the hash function is cryptographically secure, about half the bits of K_{cs} will not match K_{cs}'; the same will be true for K_{sc} and K_{sc}'. K_{sc} and K_{cs} are about to be used as keys, so the parties will quickly discover any such mismatch.

2. An adversary may have replaced a component in such a way that both parties still have matching sets. But if we compare the components of K_{cs} and K_{cs}', we notice that at least one of the parties uses a personally chosen (unprimed) version of every component, and the adversary could not have changed that version, so there is no way for an adversary to make a matching change for both parties.

3. An adversary may have been able to discover all of the components and thus be able to calculate K_{sc}, K_{cs}, or both. But the values of $K_{c\text{-}cs}$ and $K_{c\text{-}sc}$ were created by the client and encrypted under S_{pub} before sending them to the service, so only the client and the service know those two components.

If $K_{cs} = K_{cs}'$ and $K_{sc} = K_{sc}'$, the two parties have two keys that only they know, and only the service and this client could have calculated them. In addition, because they are calculated using $K_{s\text{-}sc}$, $K_{c\text{-}sc}$, $K_{s\text{-}cs}$, and $K_{c\text{-}cs}$, which are nonces created just for this exchange, both parties are ensured that K_{cs} and K_{sc} are fresh. In summary, K_{cs} and K_{sc} are newly generated shared secrets.

The protocol proceeds with the client generating a shared-secret authentication key $K_{ssa\text{-}cs}$ and a shared-secret encryption key $K_{sse\text{-}cs}$ from K_{cs}, perhaps by simply using the first half of K_{cs} as $K_{ssa\text{-}cs}$ and the second half as $K_{sse\text{-}cs}$. The client can now prepare and send an encrypted and authenticated request:

$$\{M\}^{K_{\text{sse-cs}}}_{K_{\text{ssa-cs}}}$$

to the server. The server generates the same shared-secret authentication key $K_{ssa\text{-}cs}$ and a shared-secret encryption key $K_{sse\text{-}cs}$ from K_{cs}' and it can now try to decrypt and authenticate M. If the authentication succeeds, the server knows that $K_{cs} = K_{cs}'$.

The server performs a similar procedure based on K_{sc} for its response. If the client successfully authenticates the response the client knows $K_{sc} = K_{sc}'$. The fact that it received a response tells it that the server successfully verified that $K_{cs} = K_{cs}'$.

From now on, the client knows that it is talking to the server associated with S_{pub}, and the connection is confidential. The server knows that the connection is confidential and that all messages are coming from the same source, but it does not know what that source is. If the server wants to know the source, it can ask and, for example, demand a password to authenticate the identity that the source claims.

To ensure forward secrecy, the client periodically repeats the whole protocol periodically. At regular intervals (e.g., every hour), the client discards the temporary keys T_{pub} and T_{priv}, generates a new public key T_{pub} and private key T_{priv}, and runs the protocol again.

11.5.7 Summary

This section described several security protocols to obtain different objectives. We studied a challenge-response protocol to open garage doors. We studied an incorrect protocol to set up a secure communication channel between two parties. Then, we studied a correct protocol for that same purpose that provides confidentiality but doesn't authenticate the participants. Finally, we studied a protocol for setting up a secure communication channel that provides both confidentiality and authenticity. Protocols for setting up secure channels become imporant whenever the participants are separated by a network. Section 11.10 describes a protocol for setting up secure channels in the World-Wide Web.

Many systems have additional security requirements, and therefore may need protocols with different features. For example, a system that provides anonymous e-mail must provide an authenticated and confidential communication channel between two parties with the property that the receiver knows that a message came from the same source as previous messages and that nobody else has read the message, but must also hide the identity of the sender from the receiver. Such a system requires a more sophisticated design and protocols because hiding the identity of the sender is a difficult problem. The receiver may be able to learn the Internet address from which some of the messages were sent or may be able to observe traffic on certain communication links; to make anonymous e-mail resist such analysis requires elaborate protocols that are beyond the scope of this text, but see, for example, Chaum's paper for a solution [Suggestions for Further Reading 11.5.6]. Security protocols are also an active area of research and researchers continuously develop novel systems and protocols for new scenarios or for particular challenging problems such as electronic voting, which may require keeping the identity of the voter secret, preventing a voter from voting more than once, allowing the voter to verify that the vote was correctly recorded, and permitting recounts. The interested reader is encouraged to consult the professional literature for developments.

11.6 Authorization: Controlled Sharing

Some data must stay confidential. For example, users require that their private authentication key stay confidential. Users wish to keep their password and credit card numbers confidential. Companies wish to keep the specifics of their upcoming products confidential. Military organizations wish to keep attack plans confidential.

The simplest way of providing confidentiality of digital data is to separate the programs that manipulate the data. One way of achieving that is to run each program and its associated data on a separate computer and require that the computers cannot communicate with each other.

The latter requirement is usually too stringent: different programs typically need to share data and strict separation makes this sharing impossible. A slight variation, however, of the strict separation approach is used by military organizations and some businesses. In this variation, there is a trusted network and an untrusted network. The trusted network connects trusted computers with sensitive data, and perhaps uses encryption to protect data as it travels over the network. By policy, the computers on the untrusted network don't store sensitive data, but might be connected to public networks such as the Internet. The only way to move data between the trusted and untrusted network is manual transfer by security personnel who can deny or authorize the transfer after a careful inspection of the data.

For many services, however, this slightly more relaxed version of strict isolation is still inconvenient because users need to have the ability to share more easily but keep control over what is shared and with whom. For example, users may want share files on a file server, but have control over whom they authorize to have access to what files. As another example, many users acquire programs created by third parties, run them on their computer, but want to be assured that their confidential data cannot be read by these untrusted programs. This section introduces authorization systems that can support these requirements.

11.6.1 Authorization Operations

We can distinguish three primary operations in authorization systems:

- *authorization*. This operation grants a principal permission to perform an operation on an object.
- *mediation*. This operation checks whether or not a principal has permission to perform an operation on a particular object.
- *revocation*. This decision removes a previously-granted permission from a principal.

The agent that makes authorization and revocation decisions is known as an authority. The authority is the principal that can increase or decrease the set of principals that have access to a particular object by granting or revoking respectively their permissions. In this chapter we will see different ways how a principal can become an authority.

The guard is distinct from, but operates on behalf of the authority, making mediation decisions by checking the permissions, and denying or allowing a request based on the permissions.

We discuss three models that differ in the way the service keeps track of who is authorized and who isn't: (1) the simple guard model, (2) the caretaker model, and (3) the flow-control model. The simple guard model is the simplest one, while flow control is the most complex model and is used primarily in heavy-duty security systems.

11.6.2 The Simple Guard Model

The simple guard model is based on an *authorization matrix*, in which principals are the rows and objects are the columns. Each entry in the matrix contains the permissions that a principal has for the given object. Typical permissions are read access and write access. When the service receives a request for an object, the guard verifies that the requesting principal has the appropriate permissions in the authorization matrix to perform the requested operation on the object, and if so, allows the request.

The authority of an object is the principal who can set the permissions for each principal, which raises the question how a principal can become an authority. One common design is that the principal who creates an object is automatically the authority for that object. Another option is to have an additional permission in each entry of the authorization matrix that grants a principal permission to change the permissions. That is, the permissions of an object may also include a permission that grants a principal authority to change the permissions for the object.

When a principal creates a new object, the access-control system must determine which is the appropriate authority for the new object and also what initial permissions it should set. *Discretionary access-control* systems make the creator of the object the authority and allow the creator to change the permission entries at the creator's discretion. The creator can specify the initial permission entries as an argument to the create operation or, more commonly, use the system's default values. *Non-discretionary access-control* systems don't make the creator the authority but chose an authority and set the permission entries in some other way, which the creator cannot change at the creator's discretion. In the simple guard model, access control is usually discretionary. We will return to non-discretionary access control in Section 11.6.5.

There are two primary instances of the simple guard model: list systems, which are organized by column, and ticket systems, which are organized by row. The primary way these two systems differ is who stores the authorization matrix: the list system stores columns in a place that the guard can refer to, while the ticket system stores rows in a place that principals have access to. This difference has implications on the ease of revocation. We will discuss ticket systems, list systems, and systems that combine them, in turn.

11.6.2.1 The Ticket System

In the *ticket system*, each guard holds a ticket for each object it is guarding. A principal holds a separate ticket for each different object the principal is authorized to use. One can compare the set of tickets that the principal holds to a ring with keys. The set of tickets that principal holds determines exactly which objects the principal can obtain access to. A ticket in a ticket-oriented system is usually called a *capability*.

To authorize a principal to have access to an object, the authority gives the principal a matching ticket for the object. If the principal wishes, the principal can simply pass this ticket to other principals, giving them access to the object.

To revoke a principal's permissions, the authority has to either hunt down the principal and take the ticket back, or change the guard's ticket and reissue tickets to any other principals who should still be authorized. The first choice may be hard to implement; the second may be disruptive.

11.6.2.2 The List System

In the *list system*, revocation is less disruptive. In the list system, each principal has a token identifying the principal (e.g., the principal's name) and the guard holds a list of tokens that correspond to the set of principals that the authority has authorized. To mediate, a guard must search its list of tokens to see if the principal's token is present. If the search for a match succeeds, the guard allows the principal access; if not, the guard denies that principal access. To revoke access, the authority removes the principal's token from the guard's list. In the list system, it is also easy to perform audits of which principals have permission for a particular object because the guard has access to the list of tokens for each object. The list of tokens is usually called an *access-control list (ACL)*.

Table 11.1: Comparison of access control systems

System	Advantage	Disadvantage
Ticket	Quick access check	Revocation is difficult
	Tickets can be passed around	Tickets can be passed around
List	Revocation is easy	Access check requires searching a list
	Audit possible	
Agency	List available	Revocation might be hard

11.6.2.3 Tickets Versus Lists, and Agencies

Ticket and list systems each have advantages over the other. Table 11.1 summarizes the advantages and disadvantages. The differences in the ticket and list system stem primarily from who gathers, stores, and searches the authorization information. In the ticket system, the responsibility for gathering, storing, and searching the tickets rests with the principal. In the list system, responsibility for gathering, storing, and searching the tokens on a list rests with the guard. In most ticket systems, the principals store the tickets and they can pass tickets to other principals without involving the guard. This property makes sharing easy (no interaction with the authority required), but makes it hard for an authority to revoke access and for the guard to prepare audit trails. In the list system, the guard stores the tokens and they identify principals, which makes audit trails possible; on the other hand, to grant another principal access to an object requires an interaction between the authority and the guard.

The tokens in the ticket and list systems must be protected against forgery. In the ticket system, tickets must be protected against forgery. If an adversary can cook up valid tickets, then the adversary can obtain access to any object. In the list system, the token identifying the principal and the access control list must be protected. If an adversary can cook up valid principal identifiers and change the access control list at will, then the adversary can have access to any object. Since the principal identifier tokens and access control lists are in the storage of the system, protecting them isn't too hard. Ticket storage, on the other hand, may be managed by the user, and in that case protecting the tickets requires extra machinery.

A natural question to ask is if it is possible to get the best of both ticket and list systems. An *agency* can combine list and ticket systems by allowing one to switch from a ticket system to a list system, or vice versa. For example, at a by-invitation-only conference, upon your arrival, the organizers may check your name against the list of invited people (a list system) and then hand you a batch of coupons for lunches, dinners, etc. (a ticket system).

11.6.2.4 Protection Groups

Cases often arise where it would be inconvenient to list by name every principal who is to have access to each of a large number of objects that have identical permissions, either because the list would be awkwardly long, or because the list would change frequently, or to ensure that several objects have the same list. To handle this situation, most access control list systems implement *protection groups*, which are principals that may be used by more than one user. If the name of a protection group appears in an access control list for an object, all principals who are members of that protection group share the permissions for that object.

A simple way to implement protection groups is to create an access control list for each group, consisting of a list of tokens representing the individual principals who are authorized to use the protection group's principal identifier. When a user logs in, the system authenticates the user, for example, by a password, and identifies the user's token. Then, the system looks up the user's token on each group's access control list and gives the user the group token for each protection group the user belongs to. The guard can then mediate access based on the user and group tokens.

11.6.3 Example: Access Control in UNIX

The previous section described access control based on a simple guard model in the abstract. This section describes a concrete access control system, namely the one used by UNIX (see Section 2.5). UNIX was originally designed for a computer shared among multiple users, and therefore had to support access control. As described in Section 4.4, the Network File System (NFS) extends the UNIX file system to shared file servers, reinforcing the importance of access control, since without access control any user has access to all files. The version of the UNIX system described in Section 2.5 didn't provide networking and didn't support servers well; modern UNIX systems, however, do, which further reenforces the need of security. For this reason, this section mostly describes the core access control features that one can find in a modern UNIX system, which are based on the features found in early UNIX systems. For the more advanced and latest features the reader is encouraged to consult the professional literature.

One of the benefits of studying a concrete example is that it makes the clear the importance of the dynamics of use in an access control system. How are running programs associated with principals? How are access control lists changed? Who can create new principals? How does a system get initialized? How is revocation done? From these questions it should be clear that the overall security of a computer system is to a large part based on how carefully the dynamics of use have been thought through.

11.6.3.1 Principals in UNIX

The principals in UNIX are users and groups. Users are named by a string of characters. A user name with some auxiliary information is stored in a file that is historically called the password file. Because it is inconvenient for the kernel to use character strings for user names, it uses fixed-length integer names (called UIDs). The UID of each user is stored along with the user name in a file called colloquially the password file (/etc/passwd). The password file usually contains other information for each user too; for example, it contains the name of the program that a users wants the system to run when the user logs in.

A group is a protection group of users. Like users, groups are named by a string of characters. The group file ("/etc/group") stores all groups. For each group it stores the group name, a fixed-length integer name for the group (called the GID), and the user names (or UIDs depending on which version of UNIX) of the users who are a member of the group. A user can be in multiple groups; one of these group is the user's default group. The name of the default group is stored in the user's entry in the password file.

The principal *superuser* is the one used by system administrators and has full authority; the kernel allows the superuser to change any permissions. The superuser is also called *root*, and has the UID 0.

A system administrator usually creates several service principals to run services instead of for running them with superuser authority. For example, the principal named "www" runs the Web server in a typical UNIX configuration. The reason to do so is that if the server is compromised (e.g., through a buffer overrun attack), then the adversary acquires only the privileges of the principal www, and not those of the superuser.

11.6.3.2 ACLs in UNIX

UNIX represents all shared objects (files, devices, etc.) as files, which are protected by the UNIX kernel (the guard). All files are manipulated by programs, which act on behalf of some principal. To isolate programs from one another, UNIX runs each program in its own address space with one or more threads (called a process in UNIX). All mediation decisions can be viewed as whether or not a particular process (and thus principal) should be allowed to have access to a particular file. UNIX implements this mediation using ACLs.

Each file has an owner, a principal that is the authority for the file. The UID of the owner of a file is stored in a file's inode (see page 2.5.11). Each file also has an owning group, designated by a GID stored in the file's inode. When a file is created its UID is the UID of the principal who created the file and its GID is the GID of principal's default group. The owner of a file can change the owner and group of the file.

The inode for each file also stores an ACL. To avoid long ACLs, UNIX ACLs contain only 3 entries: the UID of the owner of the file, a group identifier (GID), and other. "Other" designates all users with UIDs and GIDs different from the ones on the ACL.

This design is sufficient for a time-sharing system for a small community, where all one needs is some privacy between groups. But when such a system is attached to the Internet, it may run services such as a Web service that provide access to certain files to any user on the Internet. The Web server runs under some principal (e.g., "www"). The UID associated with that principal is included in the "other" category, which means that "other" can mean anyone in the entire Internet. Because allowing access to the entire world may be problematic, Web servers running under UNIX usually implement their own access restrictions in addition to those enforced by the ACL. (But recall the discussion of the TCB on page 11–26. This design drags the Web server inside the TCB.) For reasons such as these, file servers that are designed for a larger community or to be attached to the Internet, such as the Andrew File System [Suggestions for Further Reading 4.2.3], support full-blown ACLs.

Per ACL entry, UNIX keeps several permissions: READ (if set, read operations are allowed), WRITE (if set, write operations are allowed), and EXECUTE (if set, the file is allowed to be executed as a program). So, for example, the file "y" might have an ACL with UID 18, GID 20, and permissions "rwxr-xr--". This information says the owner (UID 18) is allowed to read, write, and execute file "y", users belonging to group 20 are allowed to read and execute file "y", and all other users are allowed only read access. The owner of a file has the authority to change the permission on the file.

The initial owner and permission entries of a new file are set to the corresponding values of the process that created the file. What the default principal and permissions are of a process is explained next.

11.6.3.3 The Default Principal and Permissions of a Process

The kernel stores for a process the UID and the GIDs of the principal on whose behalf the process is running. The kernel also stores for a process the default permissions for files that that process may create. A common default permission is write permission for the owner, and read permission for the owner, group, and other. A process can change its default permissions with a special command (called UMASK).

By default, a process inherits the UID, GIDs, and default permissions of the process that created it. However, if the SETUID permission of a file is set on—a bit in a file's inode—the process that runs the program acquires the UID of the principal that owns the file storing the program. Once a process is running, a process can invoke the SETUID supervisor call to change its UID to one with fewer permissions.

The SETUID permission of a file is useful for programs that need to increase their privileges to perform privileged operations. For example, an e-mail delivery program that receives an e-mail for a particular user must be able to append the mail to the user's mailbox. Making the target mailbox writable for anyone would allow any user to destroy another user's mailbox. If a system administrator sets the SETUID permission on the mail delivery program and makes the program owned by the superuser, then the mail program will run with superuser privileges. When the program receives an e-mail for a user, the program changes its UID to the target user's, and can append the mail to the user's mailbox. (In principle the delivery program doesn't have to change to the target's UID, but changing the UID is better practice than running the complete program with superuser privileges. It is another example of the *principle of least privilege*.)

Another design option would be for UNIX to set the ACL on the mailbox to include the principal of the e-mail deliver program. Unfortunately, because UNIX ACLs are limited to the user, group, and other entries, they are not flexible enough to have an entry for a specific principal, and thus the SETUID plan is necessary. The SETUID plan is not ideal either, however, because there is a temptation for application designers to run applications with superuser privileges and never drop them, violating the *principle of least* *privilege*. In retrospect, UNIX's plan for security is weak, and the combination of buffer-overrun attacks and applications running with too much privilege has led to many security breaches. To design an application to run securely on UNIX requires much careful thought and sophisticated use of UNIX.

With the exception of the superuser, only the principal on whose behalf a process is running can control a process (e.g., stop it). This design makes it difficult for an adversary who successfully compromised one principal to damage other processes that act on behalf of a different principal.

11.6.3.4 Authenticating Users

When a UNIX computer starts, it boots the kernel (see Sidebar 5.3). The kernel starts the first user program (called init in UNIX) and runs it with the superuser authority. The init program starts among other things a login program, which also executes with the superuser authority. Users type in their user name and a password to a login program. When a person types in a name and password, the login program hashes the password using a cryptographic hash (as was explained on page 11–32) and compares it with the hash of the password that it has on file that corresponds to the user name the person has claimed. If they match, the login program looks up the UID, GIDs, and the starting program for that user, uses SETUID to change the UID of the login program to the user's UID, and runs the user's starting program. If hashes don't match, the login program denies access.

As mentioned earlier, the user name, UID, default GID, and other information are stored in the password file (named "/etc/passwd"). At one time, hashed passwords were also stored in the password file. But, because the other information is needed by many programs, including programs run by other users, most systems now store the hashed password in a separate file called the "shadow file" that is accessible only to the superuser. Storing the passwords in a limited access file makes it harder for an adversary to mount a dictionary attack against the passwords. Users can change their password by invoking a SETUID program that can write the shadow file. Storing public user information in the password file and sensitive hashed passwords in the shadow file with more restrictive permissions is another example of applying the *principle of least privilege*.

11.6.3.5 Access Control Check

Once a user is logged in, subsequent access control is performed by the kernel based on UIDs and GIDs of processes, using a list system. When a process invokes OPEN to use a file, the process performs a system call to enter the kernel. The kernel looks up the UID and GIDs for the process in its tables. Then, the kernel performs the access check as follows:

1. If the UID of the process is 0 (superuser), the process has the necessary permissions by default.
2. If the UID of the process matches the UID of the owner of the file, the kernel checks the permissions in the ACL entry for owner.

3. If UIDs do not match, but if one of the process's GIDs match the GID of the file, the kernel checks the permissions in the ACL entry for group.
4. If the UID and GIDs do not match, the kernel checks the permissions in the ACL entry for "other" users.

If the process has the appropriate permission, the kernel performs the operation; otherwise, it returns a permission error.

11.6.3.6 Running Services

In addition to starting the login program, the init program usually starts several services (e.g., a Web server, an e-mail server, a X Windows System server, etc.). The services often start run with the privileges of the superuser principal, but switch to a service principal using SETUID. For example, a well-designed Web server changes its UID from the superuser principal to the www principal after it did the few operations that require superuser privileges. To ensure that these services have limited access if an adversary compromises one of them, the system administrator sets file permissions so that, for example, the principal named www has permission to access only the files it needs. In addition, a Web server designed with security in mind will also use the CHROOT call (see Section 2.5.1) so that it can name only the files in its corner of file system. These measures ensure that an adversary can do only restricted harm when compromising a service. These measures are examples of both the paranoid design attitude and of *the principle of least privilege*.

11.6.3.7 Summary of UNIX Access Control

The UNIX login program can be viewed as an access control system following the pure guard model that combines authentication of users with mediating access to the computer to which the user logs in. The guard is the login program. The object is the UNIX system. The principal is the user. The ticket is the password, which is protected using a cryptographic hash function. If the tickets match, access is allowed; otherwise, access is denied. We can view the whole UNIX system as an agent system. It switches from a simple ticket-based guard system (the login program) to a list-oriented system (the kernel and file system). UNIX thus provides a comprehensive example of the simple guard model. In the next two sections we investigate two other models for access control.

11.6.4 The Caretaker Model

The caretaker model generalizes the simple guard model. It is the object-oriented version of the simple guard model. The simple guard model checks permissions for simple methods such as read, write, and execute. The caretaker model verifies permissions for arbitrary methods. The caretaker can enforce arbitrary constraints on access to an object, and it may interpret the data stored in the object to decide what to do with a given request.

Example access-control systems that follow the caretaker model are:

- A bank vault that can be opened at 5:30 pm, but not at any other time.
- A box that can be opened only when two principals agree.
- Releasing salary information only to principals who have a higher salary.
- Allowing the purchase of a book with a credit card only after the bank approves the credit card transaction.

The hazard in the caretaker model is that the program for the caretaker is more complex than the program for the guard, which makes it easy to make mistakes and leave loopholes to be exploited by adversaries. Furthermore, the specification of what the caretaker's methods do and how they interact with respect to security may be difficult to understand, which may lead to configuration errors. Despite these challenges, database systems typically support the caretaker model to control access to rows and columns in tables.

11.6.5 Non-Discretionary Access and Information Flow Control

The description of authorization has so far rested on the assumption that the principal that creates an object is the authority. In the UNIX example, the owner of a file is the authority for that file; the owner can give all permissions including the ability to change the ACL, to another user.

This authority model is *discretionary*: an individual user may, at the user's own discretion, authorize other principals to obtain access to the objects the user creates. In certain situations, discretionary control may not be acceptable and must be limited or prohibited. In this case, the authority is not the principal who created the object, but some other principal. For example, the manager of a department developing a new product line may want to *compartmentalize* the department's use of the company computer system to ensure that only those employees with a need to know have access to information about the new product. The manager thus desires to apply the *least privilege principle*. Similarly, the marketing manager may wish to compartmentalize all use of the company computer for calculating product prices, since pricing policy may be sensitive.

Either manager may consider it unacceptable that any individual employee within the department can abridge the compartments merely by changing an access control list on an object that the employee creates. The manager has a need to limit the use of discretionary controls by the employees. Any limits the manager imposes on authorization are controls that are out of the hands of the employees, and are viewed by them as *non-discretionary*.

Similar constraints are imposed in military security applications, in which not only isolated compartments are required, but also nested sensitivity levels (e.g., unclassified, confidential, secret, and top secret) that must be modeled in the authorization mechanics of the computer system. Commercial enterprises also use non-discretionary controls. For example, a non-disclosure agreement may require a person for the rest of the person's life not to disclose the information that the agreement gave the person access to.

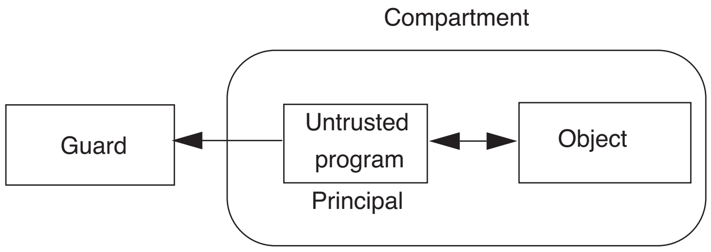


FIGURE 11.7

Confining a program within a compartment.

Non-discretionary controls may need to be imposed in addition to or instead of discretionary controls. For example, the department manager may be prepared to allow the employees to adjust their access control lists any way they wish, within the constraint that no one outside the compartment is ever given access. In that case, both non-discretionary and discretionary controls apply.

The reason for interest in non-discretionary controls is not so much the threat of malicious insubordination as the need to safely use complex and sophisticated programs created by programmers who are not under the authority's control. A user may obtain some code from a third party (e.g., a Web browser extension, a software upgrade, a new application) and if the supplied program is to be useful, it must be given access to the data it is to manipulate or interpret (see Figure 11.7). But unless the downloaded program has been completely audited, there is no way to be sure that it does not misuse the data (for example, by making an illicit copy and sending it somewhere) or expose the data either accidentally or intentionally. One way to prevent this kind of security violation would be to forbid the use of untrusted third-party programs, but for most organizations the requirement that all programs be locally written (or even thoroughly audited) would be an unbearable economic burden. The alternative is *confinement* of the untrusted program. That is, the untrusted program should run on behalf of some principal in a compartment containing the necessary data, but should be constrained so that it cannot authorize sharing of anything found or created in that compartment with other compartments.

Complete elimination of discretionary controls is easy to accomplish. For example, one could arrange that the initial value for the access control list of all newly created objects not give "ACL-modification" permission to the creating principal (under which the downloaded program is running). Then the downloaded program could not release information by copying it into an object that it creates and then adjusting the access control list on that object. If, in addition, all previously existing objects in the compartment of the downloaded program do not permit that principal to modify the access control list, the downloaded program would have no discretionary control at all.

An interesting requirement for a non-discretionary control system that implements isolated compartments arises whenever a principal is authorized to have access to two or more compartments simultaneously, and some data objects may be labeled as being simultaneously in two or more compartments (e.g., pricing data for a new product may be labeled as requiring access to the "pricing policy" compartment as well as the "new product line" compartment). In such a case it would seem reasonable that, before permitting reading of data from an object, the control mechanics should require that the set of compartments of the object being referenced be a subset of the compartments to which the accessor is authorized.

A more stringent interpretation, however, is required for permission to write, if downloaded programs are to be confined. Confinement requires that the program be constrained to write only into objects that have a compartment set that is a subset of that of the program itself. If such a restriction were not enforced, a malicious downloaded program could, upon reading data labeled for both the "pricing policy" and the "new product line" compartments, make a copy of part of it in an object labeled only "pricing policy," thereby compromising the "new product line'' compartment boundary. A similar set of restrictions on writing can be expressed for sensitivity levels. A set of such restrictions is known as rules for *information flow control*.

11.6.5.1 Information Flow Control Example

To make information flow control more concrete, consider a company that has information divided in two compartment:

1. financial (e.g., product pricing)
2. product (e.g., product designs)

Each file in the computer system is labeled to belong to one of these compartments. Every principal is given a clearance for one or both compartments. For example, the company's policy might be as follows: the company's accounts have clearance for reading and writing files in the financial compartment, the company's engineers have clearance for reading and writing files in the product compartment, and the company's product managers have clearance for reading and writing files in both compartments.

The principals of the system interact with the files through programs, which are untrusted. We want ensure that information flows only to the company's policy. To achieve this goal, every thread records the labels of the compartments for which the principal is cleared; this clearance is stored in $T_{\text{labelsseen}}$. Furthermore, the system remembers the maximum compartment label of data the thread has seen, $T_{maxlabels}$. Now the information flow control rules can be implemented as follows. The read rule is:

- Before reading an object with labels O_{labels}, check that $O_{labels} \subseteq T_{\text{maxlabels}}$.
- If so, set $T_{labelsseen} \leftarrow T_{labelsseen} \cup C_{labels}$, and allow access.

This rule can be summarized by "no read up." The thread is not allowed to have access to information in compartments for which it has no clearance.

The corresponding write rule is:

- Allow a write to an object with clearance O_{labels} only if $T_{labelsseen} \subseteq O_{labels}$

This rule could be called "no write down." Every object written by a thread that read data in compartments L must be labeled with L's labels. This rule ensures that if a thread T has read information in a compartment other than the ones listed in L than that information doesn't leak into the object O.

These information rules can be used to implement a wide range of policies. For example, the company can create more compartments, more principals, or modify the list of compartments a principal has clearance for. These changes in policy don't require changes in the information flow rules. This design is another example of the principle *separate mechanism from policy*.

Sometimes there is a need to move an object from one compartment to another because, for example, the information in the object isn't confidential anymore. Typically downgrading of information (*declassification* in the security jargon) must be done by a person who inspects the information in the object, since a program cannot exercise judgement. Only a human can establish that information to be declassified is not sensitive.

This example sketches a set of simple information flow control rules. In real system systems more complex information flow rules are needed, but they have a similar flavor. The United States National Security Agency has a strong interest in computer systems with information flow control, as do companies that have sensitive data to protect. The Department of Defense has a specification for what these computer systems should provide (this specification is part of a publication known as the Orange Book*, which classifies systems according to their security guarantees). It is possible that information flow control will find other usages than in high-security systems, as the problems with untrusted programs become more prevalent in the Internet, and sophisticated confinement is required.

11.6.5.2 Covert Channels

Complete confinement of a program in a system with shared resources is difficult, or perhaps impossible, to accomplish, since the program may be able to signal to other users by strategies more subtle than writing into shared objects. Computer systems with shared resources always contain *covert channels*, which are hidden communication channels through which information can flow unchecked. For example, two threads might conspire to send bits by the logical equivalent of "banging on the wall." See Section 11.11.10.1 for a concrete example and see problem set *43* for an example that literally involves banging. In practice, just finding covert channels is difficult. Blocking covert channels is an even harder problem: there are no generic solutions.

* U.S.A. Department of Defense, *Department of Defense trusted computer system evaluation criteria*, Department of Defense standard 5200, December 1985.

11.7 Advanced Topic: Reasoning about Authentication

The security model has three key steps that are executed by the guard on each request: authenticating the user, verifying the integrity of the request, and determining if the user is authorized. Authenticating the user is typically the most difficult of the three steps because the guard can establish only that the message came from the same origin as some previous message. To determine the principal that is associated with a message, the guard must establish that it is part of a chain of messages that often originated in a message that was communicated by physical rendezvous. That physical rendezvous securely binds the identity of a real-world person with a principal.

The authentication step is further complicated because the messages in the chain might even come from different principals, as we have seen in some of the security protocols in Section 11.5. If a message in the chain comes from a different principal and makes a statement about another principal, we can view the message as one principal speaking for another principal. To establish that the chain of messages originated from a particular real-world user, the guard must follow a chain of principals.

Consider a simple security protocol, in which a certificate authority signs certificates, associating authentication keys with names (e.g., "key K_{pub} belongs to the user named X"). If a service receives this certificate together with a message M for which VERIFY (M, K_{pub}) returns ACCEPT, then the question is if the guard should believe this message originated with "X". The answer is no until the guard can establish the following facts:

1. The guard knows that a message originated from a principal who knows a private authentication key K_{priv} because the message verified with K_{pub}.
2. The certificate is a message from the certification authority telling the guard that the authentication key K_{pub} is associated with user "X." (The guard can tell that the certificate came from the certificate authority because the certificate was signed with the private authentication key of the authority and the guard has obtained the public authentication key of the authority through some other chain of messages that originated in physical rendezvous.)
3. The certification authority *speaks for* user "X". The guard may believe this assumption, if the guard can establish two facts:
   - User "X" says the certificate authority speaks for "X". That is, user "X" delegated authority to the certificate authority to speak on behalf of "X". If the guard bel ficate authority carefully minted a key for "X" that speaks for only "X" and verified the identity of "X", then the guard may consider this belief a fact.
   - The certificate authority says Kpub speaks for user "X". If the guard believes that the certificate authority carefully minted a key for "X" that speaks for

only "X" and verified the identity of "X", then the guard may consider this belief a fact.

With these facts, the guard can deduce that the origin of the first message is user "X" as follows:

1. If user "X" says that the certificate authority speaks on behalf of "X", then the guard can conclude that the certificate authority speaks for "X" because "X" said it.
2. If we combine the first conclusion with the statement that the certificate authority says that "X" says that K_{pub} speaks for X, then the guard can conclude that "X" says that K_{pub} speaks for "X".
3. If "X" says that K_{pub} speaks for X, then the guard can conclude that K_{pub} speaks for "X" because "X" said it.
4. Because the first message verified with K_{pub}, the guard can conclude that the message must have originated with user "X".

In this section, we will formalize this type of reasoning using a simple form of what is called *authentication logic*, which defines more precisely what "speaks for" means. Using that logic we can establish the assumptions under which a guard is willing to believe that a message came from a particular person. Once the assumptions are identified, we can decide if the assumptions are acceptable, and, if the assumptions are acceptable, the guard can accept the authentication as valid and go on to determine if the principal is authorized.

11.7.1 Authentication Logic

Burrows-Abadi-Needham (BAN) authentication logic is a particular logic to reason about authentication systems. We give an informal and simplified description of the logic and its usage. If you want to use it to reason about a complete protocol, read *Authentication in Distributed Systems: Theory and Practice* [Suggestions for Further Reading 11.3.1].

Consider the following example. Alice types at her workstation "Send me the quiz" (see Figure 11.8). Her workstation A sends a message over the wire from network interface 14 to network interface 5, which is attached to the file service machine F, which runs the file service. The file service stores the object "quiz."

What the file service needs to know is that "Alice **says** send quiz". This phrase is a statement in the BAN authentication logic. This statement "A **says** B" means that agent A originated the request B. Informally, "A **says** B" means we have determined somehow that A actually said B. If we were within earshot, "A **says** B" is an axiom (we saw A say it!); but if we only know that "A **says** B" indirectly ("through hearsay"), we need to use additional reasoning, and perhaps make some other assumptions before we believe it.

Unfortunately, the file system knows only that network interface F.5 (that is, network interface 5 on machine F) said Alice wants the quiz sent to her. That is, the file system

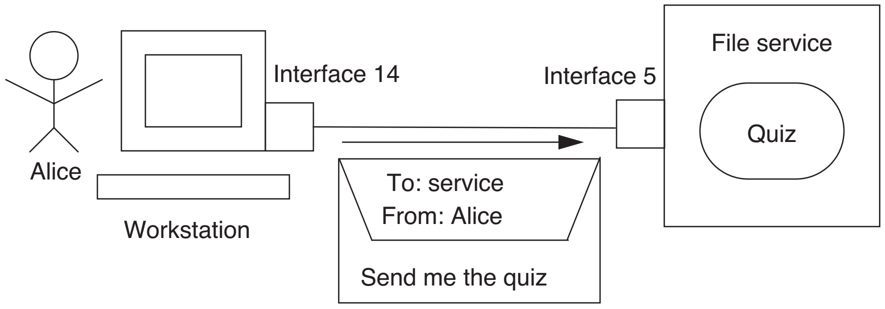


FIGURE 11.8

Authentication example.

knows "network interface F.5 **says** (Alice **says** send the quiz)". So "Alice **says** send the quiz" is only hearsay at the moment. The question is, can we trust network interface F.5 to tell the truth about what Alice did or did not say? If we do trust F.5 to speak for Alice, we write "network interface F.5 **speaks for** Alice" in BAN authentication logic. In this example, then, if we believe that "network interface F.5 **speaks for** Alice, we can deduce that "Alice **says** send the quiz."

To make reasoning with this logic work, we need three rules:

- Rule 1: Delegating authority:

If	A **says** (B **speaks for** A)
then	B **speaks for** A

This rule allows Alice to delegate authority to Bob, which allows Bob to speak for Alice.

- Rule 2: Use of delegated authority.

If	A **speaks for** B
and	A **says** (B **says** X)
then	B **says** X

This rule says that if Bob delegated authority to Alice, and Alice says that Bob said something then we can believe that Bob actually said it.

- Rule 3: Chaining of delegation.

If	A **speaks fo**r B
and	B **speaks for** C
then	A **speaks for** C

This rule says that delegation of authority is transitive: if Bob has delegated authority to Alice and Charles has delegated authority to Bob, then Charles also delegated authority to Alice.

To capture real-world situations better, the full-bore BAN logic uses more refined rules then these. However, as we will see in the rest of this chapter, even these three simple rules are useful enough to help flush out fuzzy thinking.

11.7.1.1 Hard-wired Approach

How can the file service decide that "network interface F.5 **speaks for** Alice"? The first approach would be to hard-wire our installation. If we hard-wire Alice to her workstation, her workstation to network interface A.14, and network interface A.14 through the wire to network interface F.5, then we have:

- network interface F.5 **speaks for** the wire: we must assume no one rewired it.
- the wire **speaks for** network interface A.14: we must assume no one tampered with the channel.
- network interface A.14 **speaks for** workstation A: we must assume the workstation was wired correctly.
- workstation A **speaks for** Alice: we assume the operating system on Alice's workstation can be trusted.

In short, we assume that the network interface, the wiring, and Alice's workstation are part of the trusted computing base. With this assumption we can apply the chaining of delegation rule repeatedly to obtain "network interface F.5 **speaks for** Alice". Then, we can apply the use of delegated authority rule and obtain "Alice **says** send the quiz". Authentication of message origin is now complete, and the file system can look for Alice's token on its access control list.

The logic forced us to state our assumptions explicitly. Having made the list of assumptions, we can inspect them and see if we believe each is reasonable. We might even hire an outside auditor to offer an independent opinion.

11.7.1.2 Internet Approach

Now, suppose we instead connect the workstation's interface 14 to the file service's interface 5 using the Internet. Then, following the previous pattern, we get:

- network interface F.5 **speaks for** the Internet: we must assume no one rewired it.
- the Internet **speaks for** network interface A.14: we must assume the Internet is trusted!

The latter assumption is clearly problematic; we are dead in the water.

What can we do? Suppose the message is sent with some authentication tag—Alice actually sends the message with a MAC (reminder: $\{M\}_k$ denotes a plaintext message signed with a key k):

Alice ⇒ file service: {From: Alice; To: file service; "send the quiz"}$_T$

Then, we have:

- key T **says** (Alice **says** send the quiz).

If we know that Alice was the only person in the world who knows the key *T*, then we would be able to say:

- key *T* **speaks for** Alice.

With the use of delegated authority rule we could conclude "Alice **says** send the quiz". But is Alice really the only person in the world who knows key *T*? We are using a shared-secret key system, so the file service must also know the key, and somehow the key must have been securely exchanged between Alice and the file service. So we must add to our list of assumptions:

- the file service is not trying to trick itself;
- the exchange of the shared-secret key was secure;
- Neither Alice nor the file service have revealed the key.

With these assumptions we really can believe that "key *T* **speaks for** Alice", and we are home free. This reasoning is not a proof, but it is a method that helps us to discover and state our assumptions clearly.

The logic as presented doesn't deal with freshness. In fact, in the example, we can conclude only that "Alice **said** send the quiz", but not that Alice said it recently. Someone else might be replaying the message. Extensions to the basic logic can deal with freshness by introducing additional rules for freshness that relate **says** and **said**.

11.7.2 Authentication in Distributed Systems

All of the authentication examples we have discussed so far have involved one service. Using the techniques from Section 11.6, it is easy to see how we can build a single-service authentication and authorization system. A user sets up a confidential and authenticated communication channel to a particular service. The user authenticates itself over the secure channel and receives from the service a token to be used for access control. The user sends requests over the secure channel. The service then makes its access control decisions based on the token that accompanies the request.

Authentication in the World-Wide Web is an example of this approach. The browser sets up a secure channel using the SSL/TLS protocol described in Section 11.10. Then, the browser asks the user for a password and sends this password over the secure channel to the service. If the service identifies the user successfully with the received password, the service returns a token (a cookie in Web terminology), which the browser stores. The browser sends subsequent Web requests over the secure channel and includes the cookie with each request so that the user doesn't have to retype the password for each request. The service authenticates the principal and authorizes the request based on the cookie. (In practice, many Web applications don't set up a secure channel, but just communicate the password and cookie without any protection. These applications are vulnerable to most of the attacks discussed in previous sections.)

The disadvantage of this approach to authentication is that services cannot share information about clients. The user has to log in to each service separately and each service has to implement its own authentication scheme. If the user uses only a few services, these shortcomings are not a serious inconvenience. However, in a realm (say a large company or a university) where there are many services and where information needs to be shared between services, a better plan is needed.

In such an environment we would like to have the following properties:

1. the user logs in once;
2. the tokens the user obtains after login in should be usable by all services for authentication and to make authorization decisions;
3. users are named in a uniform way so that their names can be put on and removed from access control lists;
4. users and services don't have to trust the network.

These goals are sometimes summarized as *single login* or *single sign-on*. Few system designs or implementations meet these requirements. One system that comes close is Kerberos (see Sidebar 11.6). Another system that is gaining momentum for single sign-on to Web sites is openID; its goal is to allow users to have one ID for different Internet stores. The openID protocols are driven by a public benefit organization called the OpenID Foundation. Many major companies have joined the openID Foundation and providing support in their services for openID.

11.7.3 Authentication across Administrative Realms

Extending authentication across realms that are administrated by independent authorities is a challenge. Consider a student who is running a service on a personal computer in his dorm room. The personal computer is not under the administrative authority of the university; yet the student might want to obtain access to his service from a computer in a laboratory, which is administered by central campus authority. Furthermore, the student might want to provide access to his service to family and friends who are in yet other administrative realms. It is unlikely that the campus administration will delegate authority to the personal computer, and set up secure channels from the campus authentication service to each student's authentication service.

Sharing information with many users across many different administrative realms raises a number of questions:

1. How can we authenticate services securely? The Domain Name System (DNS) doesn't provide authenticated bindings of name to IP addresses (see Section 4.4) and so we cannot use DNS names to authenticate services.
2. How can we name users securely? We could use e-mail addresses, such as bob@Scholarly.edu, to identify principals but e-mail addresses can be spoofed.
3. How do we manage many users? If Pedantic University is willing to share course software with all students at The Institute of Scholar Studies, Pedantic University

shouldn't have to list individually every student of The Institute of Scholar Studies on the access control list for the files. Clearly, protection groups are needed. But, how does a student at The Institute of Scholar Studies prove to Pedantic University's service that the student is part of the group students@Scholarly.edu?

These three problems are naming problems: how do we name a service, a user, a group, and a member of a protection group *securely*? A promising approach is to split the problem into two parts: (1) name all principals (e.g., services, users, and groups) by *public keys* and (2) securely distribute symbolic names for the public keys separately. We discuss this approach in more detail.

By naming principals by a public key we eliminate the distinction of realms. For example, a user Alice at Pedantic University might be named by a public key K_{Apub} and a user Bob at The Institute of Scholar Studies is named by a public K_{Bpub}; from the public key we cannot tell whether the Alice is at Pedantic University or The Institute of Scholar Studies. From the public key alone we cannot tell if the public key is Alice's, but we will solve the binding from public key to symbolic name separately in the next Sections 11.7.4 through 11.7.6.

If the Alice wants to authorize Bob to have access to her files, Alice adds K_{Bpub} to her access control list. If Bob wants to use Alice's files, Bob sends a request to Alice's service including his public key K_{Bpub}. Alice checks if K_{Bpub} appears on her access control list. If not, she denies the request. Otherwise, Alice's service challenges Bob to prove that he has the private key corresponding to K_{Bpub}. If Bob can prove that he has K_{Bpriv} (e.g., for example by signing a challenge that Alice's service verifies with Bob's public key K_{Bpub}), then Alice's service allows access.

When Alice approves the request, she doesn't know for sure if the request came from the principal named "Bob"; she just knows the request came from a principal holding the private key K_{Bpriv}. The symbolic name "Bob" doesn't play a role in the mediation decision. Instead, the crucial step was the authorization decision when Alice added K_{Bpub} to her access control; as part of that authorization decision Alice must assure herself that K_{Bpub} **speaks for** Bob before adding K_{Bpub} to her access control list. That assurance relies on securely distributing bindings from name to public key, which we separated out as an independent problem and will discuss in the next Sections 11.7.4 through 11.7.6.

We can name protection groups also by a public key. Suppose that Alice knew for sure that $K_{ISSstudentspub}$ is a public key representing students of The Institute of Scholarly Studies. If Alice wanted to grant all students at The Institute of Scholarly Studies access to her files, she could add $K_{ISSstudentspub}$ to her access control list. Then, if Charles, a student at The Institute of Scholar Studies, wanted to have access to one of Alice's files, he would have to present a proof that he is a member of that group, for example, by providing a statement to Alice signed by $K_{ISSstudentspriv}$ to Alice saying:

$\{K_{Charlespub}$ is a member of the group $K_{ISSstudentspub}\}_{K_{ISSstudentspriv}}$,

which in the BAN logic translates to:

$K_{Charlespub}$ **speaks for** $K_{ISSstudentspub}$, that is, Alice delegated authority to the member Charles to speak on behalf of the group of students at The Institute of Scholarly Studies.

Alice's service can verify this statement using $K_{ISSstudentspub}$, which is on Alice's access control list. After Alice's service successfully verifies the statement, then the service can challenge Charles to prove that he is the holder of the private key $K_{Charlespriv}$. Once Charles can prove he is the holder of that private key, then Alice's service can grant access to Charles.

In this setup, Alice must trust the holder of $K_{ISSstudentspriv}$ to be a responsible person who carefully verifies that Charles is a student at The Institute of Scholarly Studies. If she trusts the holder of that key to do so, then Alice doesn't have to maintain her own list of who is a student at The Institute of Scholar Studies; in fact, she doesn't need to know at all which particular principals are students at The Institute of Scholarly Studies.

If services are also named by public keys, then Bob and Charles can easily authenticate Alice's service. When Bob wants to connect to Alice's service, he specifies the public key of the service. If the service can prove that it possesses the corresponding private key, then Bob can have confidence that he is talking to the right service.

By naming all principals with public keys we can construct distributed authentication systems. Unfortunately, public keys are long, unintelligible bit strings, which are awkward and unfriendly for users to remember or type. When Alice adds K_{Bobpub} and $K_{ISSstudentspub}$ to her access control list, she shouldn't be required to type in a 1,024-bit number. Similarly when Bob and Charles refer to Alice's service, they shouldn't be required to know the bit representation of the public key of Alice's service. What is necessary is a way of naming public keys with symbolic names and authenticating the binding between name and key, which we will discuss next.

11.7.4 Authenticating Public Keys

How do we authenticate that K_{Bpub} is Bob's public key? As we have seen before, that authentication can be based on a key-distribution protocol, which start with a rendezvous step. For example, Bob and Alice meet face-to-face and Alice hands Bob a signed piece of paper with her public key and name. This piece of paper constitutes a *self-signed certificate*. Bob can have reasonable confidence in this certificate because Bob can verify that the certificate is valid and is Alice's. (Bob can ask Alice to sign again and compare it with the signature on the certificate and ask Alice for her driver license to prove her identity.)

If Bob receives a self-signed certificate over an untrusted network, however, we are out of luck. The certificate says "Hi, I am Alice and here is my public key" and it is signed with Alice's digital signature, but Bob does not know Alice's public key yet. In this case, anybody could impersonate Alice to Bob because Bob cannot verify whether or not Alice produced this certificate. An adversary can generate a public/private key pair, create a certificate for Alice listing the public key as Alice's public key, and sign it with the private key, and send this self-signed certificate to Bob.

Bob needs a way to find out securely what Alice's public key is. Most systems rely on a separate infrastructure for naming and distributing public keys securely. Such an infrastructure is called a *public key infrastructure*, PKI for short. There is a wide range of designs for such infrastructures, but their basic functions can be described well with the authentication logic. We start with a simple example using physical rendezvous and then later use certificate authorities to introduce principals to each other who haven't met through physical rendezvous.

Consider the following example where Alice receives a message from Bob, asking Alice to send a private file, and Alice wants to decide whether or not to send it. The first step in this decision is for Alice to establish if the message really came from Bob.

Suppose that Bob previously handed Alice a piece of paper on which Bob has written her public key, K_{pubBob}. We can describe Alice's take on this event in authentication logic as

$$\text{Bob } \textbf{says } (K_{pubBob} \textbf{ speaks for } \text{Bob}) \qquad \text{(belief \#1)}$$

and by applying the delegation of authority rule, Alice can immediately conclude that she is safe in believing

$$K_{pubBob} \textbf{ speaks for } \text{Bob} \qquad \text{(belief \#2)}$$

assuming that the information on the piece of paper is accurate. Alice realizes that she should should start making a list of assumptions for review later. (She ignores freshness for now because our stripped-down authentication logic has no **said** operation for capturing that.)

Next, Bob prepares a message, M_1:

$$\text{Bob } \textbf{says } M_1$$

signs it with his private key:

$$\{M_1\}_{KprivBob}$$

which, in authentication logic, can be described as

$$K_{privBob} \textbf{ says } (\text{Bob } \textbf{says } M_1)$$

and sends it to Alice. Since the message arrived via the Internet, Alice now wonders if she should believe

$$\text{Bob } \textbf{says } M_1 \qquad (?)$$

Fortunately, M_1 is signed, so Alice doesn't need to invoke any beliefs about the Internet. But the only beliefs she has established so far are (#1) and (#2), and those are not sufficient to draw any conclusions. So the first thing Alice does is check the signature:

$$\textit{result} \leftarrow \text{VERIFY } (\{M_1\}_{KprivBob}, K_{pubBob})$$

If *result* is ACCEPT then one might think that Alice is entitled to believe:

$$K_{privBob} \textbf{ says } (\text{Bob } \textbf{says } M_1) \qquad \text{(belief \#3?)}$$

but that belief actually requires a leap of faith: that the cryptographic system is secure. Alice decides that it probably is, adds that assumption to her list, and removes the question mark on belief #3. But she still hasn't collected enough beliefs to answer the question. In order to apply the chaining and use of authority rules, Alice needs to believe that

$(K_{privBob}$ **speaks for** $K_{pubBob})$ (belief #4?)

which sounds plausible, but for her to accept that belief requires another leap of faith: that Bob is the only person who knows $K_{privBob}$. Alice decides that Bob is probably careful enough to be trusted to keep his private key private, so she adds that assumption to her list and removes the question mark from belief #4.

Now, Alice can apply chaining of delegation rule to beliefs #4 and #2 to conclude

$K_{privBob}$ **speaks for** Bob (belief #5)

and she can now use the use of delegated authority rule to beliefs #5 and #3 to conclude that

Bob **says** M_1 (belief #6)

Alice decides to accepts the message as a genuine utterance of Bob. The assumptions that emerged during this reasoning were:

- K_{pubBob} is a true copy of Bob's public key.
- The cryptographic system used for signing is computationally secure.
- Bob has kept $K_{privBob}$ secret.

11.7.5 Authenticating Certificates

One of the prime usages of a public key infrastructure is to introduce principals that haven't met through a physical rendezvous. To do so a public key infrastructure provides certificates and one or more certificate authorities.

Continuing our example, suppose that Charles, whom Alice does not know, sends Alice the message

$\{M_2\}_{KprivCharles}$

This situation resembles the previous one, except that several things are missing: Alice does not know $K_{pubCharles}$, so she can't verify the signature, and in addition, Alice does not know who Charles is. Even if Alice finds a scrap of paper that has written on it Charles's name and what purports to be Charles's public key, $K_{pubCharles}$, and

result ← VERIFY $(M_2$, SIGN $(M_2, K_{privCharles}), K_{pubCharles})$

is ACCEPT, all she believes (again assuming that the cryptographic system is secure) is that

$K_{privCharles}$ **says** (Charles **says** M_2)

Without something corresponding to the previous beliefs #2 and #4, Alice still does not know what to make of this message. Specifically, Alice doesn't yet know whether or not to believe

$K_{privCharles}$ **speaks for** Charles (?)

Knowing that this might be a problem, Charles went to a well-known certificate authority, TrustUs.com, purchased the digital certificate:

{"Charles's public key is $K_{pubCharles}$"}$_{KprivTrustUs}$

and posted this certificate on his Web site. Alice discovers the certificate and wonders if it is any more useful than the scrap of paper she previously found. She knows that where she found the certificate has little bearing on its trustworthiness; a copy of the same certificate found on Lucifer's Web site would be equally trustworthy (or worthless, as the case may be).

Expressing this certificate in authentication logic requires two steps. The first thing we note is that the certificate is just another signed message, M_3, so Alice can interpret it in the same way that she interpreted the message from Bob:

$K_{privTrustUs}$ **says** M_3

Following the same reasoning that she used for the message from Bob, if Alice believes that she has a true copy of $K_{pubTrustUs}$ she can conclude that

TrustUs **says** M_3

subject to the assumptions (exactly parallel to the assumptions she used for the message from Bob)

- $K_{pubTrustUs}$ is a true copy of the TrustUs.com public key.
- The cryptographic system used for signing is computationally secure.
- TrustUs.com has kept $K_{privTrustUs}$ secret.

Alice decides that she is willing to accept those assumptions, so she turns her attention to M_3, which was the statement "Charles's public key is $K_{pubCharles}$". Since TrustUs.com is taking Charles's word on this, that statement can be expressed in authentication logic as

Charles **says** ($K_{pubCharles}$ **speaks for** Charles)

Combining, we have:

TrustUs **says** (Charles **says** ($K_{pubCharles}$ **speaks for** Charles))

To make progress, Alice needs to a further leap of faith. If Alice knew that

TrustUs **speaks for** Charles (?)

then she could apply the delegated authority rule to conclude that

Charles **says** ($K_{pubCharles}$ **speaks for** Charles) and she could then follow an analysis just like the one she used for the earlier message from Bob. Since Alice doesn't know Charles, she has no way of knowing the truth of the questioned belief (TrustUs **speaks for** Charles), so she ponders what it really means:

1. TrustUs.com has been authorized by Charles to create certificates for her. Alice might think that finding the certificate on Charles's Web site gives her some assurance on this point, but Alice has no way to verify that Charles's Web site is secure, so she has to depend on TrustUs.com being a reputable outfit.
2. TrustUs.com was careful in checking the credentials—perhaps, a driver's license—that Charles presented for identification. If TrustUs.com was not careful, it might, without realizing it, be speaking for Lucifer rather than Charles. (Unfortunately, certificate authorities have been known to make exactly that mistake.) Of course, TrustUs.com is assuming that the credentials Charles presented were legitimate; it is possible that Charles has stolen someone else's identity. As usual, authentication of origin is never absolute; at best it can provide no more than a secure tie to some previous authentication of origin.

Alice decides to review the complete list of the assumptions she needs to make in order to accept Charles's original message M_2 as genuine:

- $K_{pubTrustUs}$ is a true copy of the TrustUs.com public key.
- The cryptographic system used for signing is computationally secure.
- TrustUs.com has kept $K_{privTrustUs}$ secret.
- TrustUs.com has been authorized by Charles.
- TrustUs.com carefully checked Charles's credentials.
- TrustUs.com has signed the right public key (that is $K_{pubCharles}$).
- Charles has kept $K_{privCharles}$ secret.

and she notices that in addition to relying heavily on the trustworthiness of TrustUs.com, she doesn't know Charles, so the last assumption may be a weakness. For this reason, she would be well-advised to accept message M_2 with a certain amount of caution. In addition, Alice should keep in mind that since Charles's public key was not obtained by a physical rendezvous, she knows only that the message came from someone named "Charles"; she as yet has no way to connect that name with a real person.

As in the previous examples, the stripped-down authentication logic we have been using for illustration has no provision for checking freshness, so it hasn't alerted Alice that she is also assuming that the two public keys are fresh and that the message itself is recent.

The above example is a distributed authorization system that is ticket-oriented. Trust.com has generated a ticket (the certificate) that Alice uses to authenticate Charles's request. Given this observation, this immediately raises the question of how Charles revokes the certificate that he bought from TrustUs.com. If Charles, for example, accidently discloses his private key, the certificate from TrustUS.com becomes worthless and he should revoke it so that Alice cannot be tricked into believing that M_2 came from

Charles. One way to address this problem is to make a certificate valid for only a limited length of time. Another approach is for TrustUs.com to maintain a list of revoked certificates and for Alice to first check with TrustUS.com before accepting an certificate as valid.

Neither solution is quite satisfactory. The first solution has the disadvantage that if Charles loses his private key, the certificate will remain valid until it expires. The second solution has the disadvantage that TrustUs.com has to be available at the instant that Alice tries to check the validity of the certificate.

11.7.6 Certificate Chains

The public key infrastructure developed so far has one certificate authority, TrustUS.com. How do we certify the public key of TrustUs.com? There might be many certificate authorities, some of which Alice doesn't know about. However, Alice might possess a certificate for another certificate authority that certifies TrustUs.com, creating a chain of certification. Public key infrastructures organize such chains in two primary ways; we discuss them in turn.

11.7.6.1 Hierarchy of Central Certificate Authorities

In the central-authority approach, key certificate authorities record public keys and are managed by central authorities. For example, in the Word Wide Web, certificates authenticating Web sites are usually signed by one of several well-known root certificate authorities. Commercial Web sites, such as amazon.com, for instance, present a certificate signed by Versign to a client when it connects. All Web browsers embed the public key of the root certificates in their programs. When the browser receives a certificate from amazon.com, it uses the embedded public key for Verisign to verify the certificate.

Some Web sites, for example a company's internal Web site, generate a self-signed certificate and send that to a client when it connects. To be able to verify a self-signed certificate, the client must have obtained the key of the Web site securely in advance.

The Web approach to certifying keys has a shallow hierarchy. In DNSSEC*, a secure version of DNS, CAs can be arranged in a deeper hierarchy. If Alice types in the name "athena.Scholarly.edu", her resolver will contact one of the root servers and obtain an address and certificate for "edu". In authentication logic, the meaning of this certificate is "$K_{privroot}$ says that K_{pubedu} speaks for edu". To be able to verify this certificate she must have obtained the public key of the root servers in some earlier rendezvous step. If the certificate for "edu" verifies, she contacts the server for the "edu" domain, and asks for the server's address and certificate for "Scholarly", and so on.

One problem with the hierarchical approach is that one must trust a central authority, such as the DNS root service. The central authority may ask an unreasonable price for the service, enforce policies that you don't like, or considered untrustworthy by some.

* D. Eastlake, *Domain Name System Security Extensions*, Internet Engineering Task Force Request For Comments (RFC 2535), Mach 1999.

For example, in DNS and DNSSEC, there is a lot of politics around which institution should run the root servers and the policies of that institution. Since the Internet and DNS originated in the U.S.A., it is currently run by an U.S.A. organization. Unhappiness with this organization has led the Chinese to start their own root service.

Another problem with the hierarchical approach is that certificate authorities determine to whom they delegate authority for a particular domain name. You might be happy with the Institute of Schlarly Studies managing the "Scholarly" domain, but have less trust in a rogue government managing the top-level domain for all DNS names in that country.

Because of problems like these, it is difficult in practice to agree and manage a single PKI that allows for strong authentication world wide. Currently, no global PKI exist.

11.7.6.2 Web of Trust

The web-of-trust approach avoids using a chain of central authorities. Instead, Bob can decide himself whom he trusts. In this approach, Alice obtains certificates from her friends Charles, Dawn, and Ella and posts these on her Web page: $\{\text{Alice}, K_{Apub}\}_{K_{Cpriv}}$, $\{\text{Alice}, K_{Apub}\}_{K_{Dpriv}}$, $\{\text{Alice}, K_{Apub}\}_{K_{Epriv}}$. If Bob knows the public key of any one of Charles, Dawn, or Ella, he can verify one of the certificates by verifying the certificate that person signed. To the extent that he trusts that person to be careful in what he or she signs, he has confidence that he now has Alice's true public key.

On the other hand, if Bob doesn't know Charles, Dawn, or Ella, he might know someone (say Felipe) who knows one of them. Bob may learn that Felipe knows Ella because he checks Ella's Web site and finds a certificate signed by Felipe. If he trusts Felipe, he can get a certificate from Felipe, certifying one of the public keys K_{Cpub}, K_{Dpub}, or K_{Epub}, which he can then use to certify Alice's public key. Another possibility is that Alice offers a few certificate chains in the hope that Bob trusts one of the of the signers in one of the chains, and has the signer's public key in his set of keys. Independent of how Bob learned Alice's public key, he can inspect the chain of trust by which he learned and verified Alice's public key and see whether he likes it or not. The important point here is that Bob must trust *every* link in the chain. If any link untrustworthy, he will have no guarantees.

The web of trust scheme relies on the observation that it usually takes only a few acquaintance steps to connect anyone in the world to anyone else. For example, it has been claimed that everyone is separated by no more than 6 steps from the President of the United States. (There may be some hermits in Tibet that require more steps.) With luck, there will be many chains connecting Bob with Alice, and one of them may consist entirely of links that Bob trusts.

The central idea in the web-of-trust approach is that Bob can decide whom he trusts instead of having to trust a central authority. PGP (Pretty Good Privacy) [Suggestions for Further Reading 1.3.16] and a number of other systems use the web of trust approach.

11.8 Cryptography as a Building Block (Advanced Topic)

This section sketches how primitives such as ENCRYPT, DECRYPT, pseudorandom number generators, SIGN, VERIFY, and cryptographic hashes can be implemented using *cryptographic transformations* (also called *ciphers*). Readers who wish to understand the implementations in detail should consult books such as *Applied Cryptography* by Bruce Schneier [Suggestions for Further Reading 1.2.4], or *Handbook of Applied Cryptography* by Menezes, van Oorschot, and Vanstore [Suggestions for Further Reading 1.3.13]. *Introduction to cryptography* by Buchmann provides a concise description of the number theory that underlies cryptography [Suggestions for Further Reading 1.3.14]. There are many subtle issues in designing secure implementations of the primitives, which are beyond the scope of this text.

11.8.1 Unbreakable Cipher for Confidentiality (One-Time Pad)

Making an unbreakable cipher for *only* confidentiality is easy, but there's a catch. The recipe is as follows. First, find a process that can generate a truly random unlimited string of bits, which we call the *key string,* and transmit this key string through *secure* (i.e., providing confidentiality and authentication) channels to both the sender and receiver before they transmit any data through an insecure network.

Once the key string is securely in the hands of the sender, the sender converts the plaintext into a bit string and computes bit-for-bit the exclusive OR (XOR) of the plaintext and the key string. The sender can send the resulting ciphertext over an insecure network to a receiver. Using the previously communicated key string, the receiver can recover the plaintext by computing the XOR of the ciphertext and key string.

To be more precise, this transforming scheme is a *stream cipher*. In a stream cipher, the conversion from plaintext to ciphertext is performed one bit or one byte at a time, and the input can be of any length. In our example, a sequence of message (plaintext) bits m_1, m_2,…, m_n is transformed using an equal-length sequence of secret key bits k_1, k_2, …, k_n that is known to both the sender and the receiver. The i-th bit c_i of the ciphertext is defined to be the XOR (modulo-2 sum) of m_i and k_i, for $i = 1,…,n$:

$$c_i = m_i \oplus k_i$$

Untransforming is just as simple, because:

$$m_i = c_i \oplus k_i = m_i \oplus k_i \oplus k_i = m_i$$

This scheme, under the name "one-time pad" was patented by Vernam in 1919 (U.S. patent number 1,310,719). In his version of the scheme, the ''pad'' (that is, the one-time key) was stored on paper tape.

The key string is generated by a *random number generator*, which produces as output a "random" bit string. That is, from the bits generated so far, it is impossible to predict the next bit. True random-number generators are difficult to construct; in fact, true sources of random sequences come only from physical processes, not from deterministic computer programs.

Assuming that the key string is truly random, a one-time pad cannot be broken by the attacks discussed in Section 11.4, since the ciphertext does not give the adversary any information about the plaintext (other than the length of the message). Each bit in the ciphertext has an equal probability of being one or zero, assuming the key string consists of truly random bits. Patterns in the plaintext won't show up as patterns in the ciphertext. Knowing the value of any number of bits in the ciphertext doesn't allow the adversary to guess the bits of the plaintext or other bits in the ciphertext. To the adversary the ciphertext is essentially just a random string of the same length as the message, no matter what the message is.

If we flip a single message bit, the corresponding ciphertext bit flips. Similarly, if a single ciphertext bit is flipped by a network error (or an adversary), the receiver will untransform the ciphertext to obtain a message with a single bit error in the corresponding position. Thus, the one-time pad (both transforming and untransforming) has *limited change propagation*: changing a single bit in the input causes only a single bit in the output to change.

Unless additional measures are taken, an adversary can add, flip, or replace bits in the stream without the recipient realizing it. The adversary may have no way to know exactly how these changes will be interpreted at the receiving end, but the adversary can probably create quite a bit of confusion. This cipher provides another example of the fact that message confidentiality and integrity are separate goals.

The catch with a one-time pad is the key string. We must have a secure channel for sending the key string and the key string must be at least as long as the message. One approach to sending the key string is for the sender to generate a large key string in advance. For example, the sender can generate 10 CDs full of random bits and truck them over to the receiver by armored car. Although this scheme may have high bandwidth (6.4 Gigabytes per truckload), it probably has latency too large to be satisfactory.

The key string must be at least as long as the message. It is not hard to see that if the sender re-uses the one-time pad, an adversary can determine quickly a bit (if not everything) about the plaintext by examining the XOR of the corresponding ciphertext (if the bits are aligned properly, the pads cancel). The National Security Agency (NSA) once caught the Russians in such a mistake[*] in Project VENONA[†].

[*] R. L. Benson, The Venona Story, *National Security Agency, Center for logic History,* 2001. http://www.nsa.gov/publications/publi00039.cfm

[†] D. P. Moynihan (chair), Secrecy: Report of the commision on protecting and reducing government secrecy, *Senate document 105-2, 103rd congress*, United States government printing office,1997.

11.8.2 Pseudorandom Number Generators

One shortcut to avoid having to send a long key string over a secure channel is to use a *pseudorandom number generator*. A pseudorandom number generator produces deterministically a random-appearing bit stream from a short bit string, called the *seed*. Starting from the same seed, the pseudorandom generator will always produce the same bit stream. Thus, if both the sender and the receiver have the secret short key, using the key as a seed for the pseudorandom generator they can generate the same, long key string from the short key and use the long key string for the transformation.

Unlike the one-time pad, this scheme can in principle be broken by someone who knows enough about the pseudorandom generator. The design requirement on a pseudorandom number generator is that it is difficult for an opponent to predict the next bit in the sequence, even with full knowledge of the generating algorithm and the sequence so far. More precisely:

1. Given the seed and algorithm, it is easy to compute the next bit of the output of the pseudorandom generator.
2. Given the algorithm and some output, it is difficult (or impossible) to predict the next bit.
3. Given the algorithm and some output, it is difficult (or impossible) to compute what the seed is.

Analogous to ciphers, the design is usually open: the algorithm for the pseudorandom generator is open. Only the seed is secret, and it must be produced from a truly random source.

11.8.2.1 Rc4: A Pseudorandom Generator and its Use

RC4 was designed by Ron Rivest for RSA Data Security, Inc. RC4 stands for Ron's Code number 4. RSA tried to keep this cipher secret, but someone published a description anonymously on the Internet. (This incident illustrates how difficult it is to keep something secret, even for a security company!) Because RSA never confirmed whether the description is indeed RC4, people usually refer to the published version as ARC4, or alleged RC4.

The core of the RC4 cipher is a pseudorandom generator, which is surprisingly simple. It maintains a fixed array `S` of 256 entries, which contains a permutation of the numbers 0 through 255 (each array entry is 8 bits). It has two counters *i* and *j*, which are used as follows to generate a pseudorandom byte *k*:

```
1  procedure RC4_GENERATE ()
2      $i \leftarrow (i + 1)$ modulo 256
3      $j \leftarrow (j + S[i])$ modulo 256
4      SWAP ($S[i]$, $S[j]$)
5      $t \leftarrow (S[i] + S[j])$ modulo 256
6      $k \leftarrow S[t]$
7      return $k$
```

The initialization procedure takes as input a seed, typically a truly-random number, which is used as follows:

```
1  procedure RC4_INIT (seed)
2      for $i$ from 0 to 255 do
3          $S[i] \leftarrow i$
4          $K[i] \leftarrow seed[i]$
5      $j \leftarrow 0$
6      for $i$ from 0 to 255 do
7          $j \leftarrow (j + S[i] + K[i])$ modulo 256
8          SWAP($S[i]$, $S[j]$)
9      $i \leftarrow j \leftarrow 0$
```

The procedure RC4_INIT fills each entry of *S* with its index: $S[0] \leftarrow 0$, $S[1] \leftarrow 1$, etc. (see lines 2 through 4). It also allocates another 256-entry array (*K*) with each 8-bit entries. It fills *K* with the seed, repeating the seed as necessary to fill the array. Thus, $K[0]$ contains the first 8 bits of the key string, $K[1]$ the second 8 bits, etc. Then, it runs a loop (lines 6 through 8) that puts *S* in a pseudorandom state based on *K* (and thus the seed).

11.8.2.2 Confidentiality using RC4

Given the RC4 pseudorandom generator, ENCRYPT and DECRYPT can be implemented as in the one-time pad, except instead of using a truly-random key string, we use the output of the pseudorandom generator. To initialize, the sender and receiver invoke on their respective computers RC4_INIT, supplying the shared-secret key for the stream as the seed. Because the sender and receiver supply the same key to the initialization procedure, RC4_GENERATE on the sender and receiver computer will produce identical streams of key bytes, which ENCRYPT and DECRYPT use as a one-time pad.

In more detail, to send a byte *b*, the sender invokes RC4_GENERATE to generate a pseudorandom byte *k* and encrypts byte *b* by computing $c = b \oplus k$. When the receiver receives byte *c*, it invokes RC4_GENERATE on its computer to generate a pseudorandom byte k_1 and decrypts the byte *c* by computing $b \oplus k_1$. Because the sender and receiver initialized the generator with the same seed, *k* and k_1 are identical, and $c \oplus k_1$ gives *b*.

RC4 is simple enough that it can be coded from memory, yet it appears it is computationally secure and a moderately strong stream cipher for confidentiality, though it has been noticed that the first few bytes of its output leak information about the shared-secret key, so it is important to discard them. Like any stream cipher, it cannot be used for authentication without additional mechanism. When using it to encrypt a long stream, it doesn't seem to have any small cycles and its output values vary highly (RC4 can be in about $256! \times 256^2$ possible states). The key space contains 2^{256} values so it is also difficult to attack RC4 by brute force. RC4 must be used with care to achieve a system's overall security goal. For example, the Wired Equivalent Privacy scheme for WiFi wireless networks (see page 11–50) uses the RC4 output stream without discarding the beginning of the stream. As a result, using the leaked key information mentioned above it is relatively easy to crack WEP wireless encryption*.

The story of flawed confidentiality in WiFi's use of RC4 illustrates that it is difficult to create a really good pseudorandom number generator. Here is another example of that difficulty: during World War II, the Lorenz SZ 40 and SZ 42 cipher machines, used by the German Army, were similarly based on a (mechanical) pseudorandom number generator, but a British code-breaking team was able, by analyzing intercepted messages, to reconstruct the internal structure of the generator, build a special-purpose computer to search for the seed, and thereby decipher many of the intercepted messages of the German Army.†

11.8.3 Block Ciphers

Depending on the constraints on their inputs, ciphers are either stream ciphers or block ciphers. In a *block cipher*, the cipher performs the transformation from plaintext to ciphertext on fixed-size blocks. If the input is shorter than a block, ENCRYPT must pad the input to make it a full block in length. If the input is longer than a block, ENCRYPT breaks the input into several blocks, padding the last block is padded, if necessary, and then transforms the individual blocks. Because a given plaintext block always produces the same output with a block cipher, ENCRYPT must use a block cipher with care. We outline one widely used block cipher and how it can be used to implement ENCRYPT and DECRYPT.

11.8.3.1 Advanced Encryption Standard (AES)

Advanced Encryption Standard (AES)‡ has 128-bit (or longer) keys and 128-bit plaintext and ciphertext blocks. AES replaces *Data Encryption Standard (DES)***††, which is now regarded as too insecure for many applications, as distributed Internet computations or

* A. Stubblefield, J. Ioannidis, and A. Rubin, Using the Fluhrer, Mantin, and Shamir attack to break WEP, *Symposium on Network and Distributed System Security*, 2002.

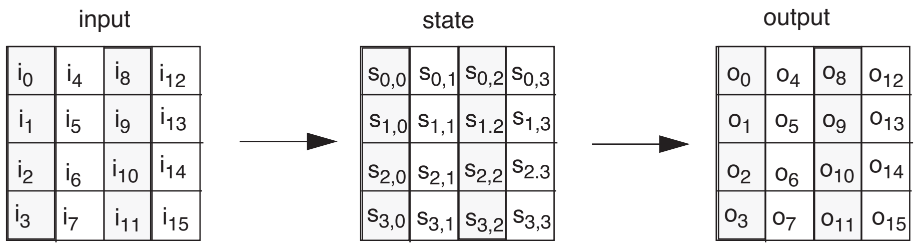
† F. H. Hinsley and Alan Stripp, *Code Breakers: The Inside Story of Bletchley Park* (Oxford University Press, 1993) page 161.

‡ Advanced Encryption Standard, *Federal Information Processing Standards Publications (FIPS PUBS) 197*, National Institute of Standards and Technology (NIST), Nov. 2001.

** Data Encryption Standard. U.S. Department of Standards, National Bureau of Standards, Federal Information Processing Standard (FIPS) Publication #46, January, 1977 (#46–1 updated 1988; #46–2 updated 1994).

dedicated special-purpose machines can use a brute-force exhaustive search to quickly find a 56-bit DES key given corresponding plaintext and ciphertext [Suggestions for Further Reading 11.5.2].

AES takes a 128-bit input and produces a 128-bit output. If you don't know the 128-bit key, it is hard to reconstruct the input given the output. The algorithm works on a 4×4 array of bytes, called *state*. At the beginning of the cipher the input array *in* is copied to the *state* array as follows:


input
i_0 i_4 i_8 i_{12}
i_1 i_5 i_9 i_{13}
i_2 i_6 i_{10} i_{14}
i_3 i_7 i_{11} i_{15}
state
$s_{0,0}$ $s_{0,1}$ $s_{0,2}$ $s_{0,3}$
$s_{1,0}$ $s_{1,1}$ $s_{1.2}$ $s_{1,3}$
$s_{2,0}$ $s_{2,1}$ $s_{2,2}$ $s_{2.3}$
$s_{3,0}$ $s_{3,1}$ $s_{3,2}$ $s_{3,3}$
output
o_0 o_4 o_8 o_{12}
o_1 o_5 o_9 o_{13}
o_2 o_6 o_{10} o_{14}
o_3 o_7 o_{11} o_{15}

At the end of the cipher the *state* array is copied into the output array *out* as depicted. The four bytes in a column form 32-bit words.

The cipher transforms *state* as follows:

```
1  procedure AES (in, out, key)
2      state ← in                          // copy in into state as described above
3      ADDROUNDKEY (state, key)            // mix key into state
4      for r from 1 to 9 do
5          SUBBYTES (state)                // substitute some bytes in state
6          SHIFTROWS (state)               // shift rows of state cyclically
7          MIXCOLUMNS (state)              // mix the columns up
8          ADDROUNDKEY (state, key[r×4, (r+1)×4 – 1])      // expand key, mix in
9      SUBBYTES (state)
10     SHIFTROWS (state)
11     ADDROUNDKEY (state, key[10×4, 11×4 – 1])
12     out ← state                         // copy state into out as described above
```

The cipher performs 10 rounds (denoted by the variable *r*), but the last round doesn't invoke MIXCOLUMNS. Each ADDROUNDKEY takes the 4 words from *key* and adds them into the columns of *state* as follows:

$$[s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}] \leftarrow [s_{0,c}, s_{1,c}, s_{2,c}, s_{3,c}, s_{4,c}] \oplus key_{r\times4+c}, \text{ for } 0 \le c < 4.$$

That is, each word of *key* is added to the corresponding column in *state*.

†† Horst Feistel, William A. Notz, and J. Lynn Smith. Some cryptographic techniques for machine-to-machine data communications. Proceedings of the IEEE 63, 11 (November, 1975), pages 1545–1554. An older paper by the designers of the DES providing background on why it works the way it does. One should be aware that the design principles described in this paper are incomplete; the really significant design principles are classified as military secrets.

For the first invocation (on line 3) of ADDROUNDKEY *r* is 0, and in that round ADDROUNDKEY uses the 128-bit key completely. For subsequent rounds, AES generates additional key words using a carefully-designed algorithm. The details and justification are outside of the scope of this textbook, but the flavor of the algorithm is as follows. It takes earlier-generated words of the key and produces a new word, by substituting well-chosen bits, rotating words, and computing the XOR of certain words.

The procedure SUBBYTES applies a substitution to the bytes of *state* according to a well-chosen substitution table. In essence, this mixes the bytes of state up.

The procedure SHIFTROWS shifts the last three rows of *state* cyclically as follows:

$s_{r,c} \leftarrow s_{r,(c+\text{shift}(r,\,4))\ \textbf{modulo}\ 4}$, for $0 \le c < 4$

The value of SHIFT is dependent on the row number as follows:

$$\text{SHIFT}(1,4) = 1,\ \text{SHIFT}(2,4) = 2, \text{ and } \text{SHIFT}(3,4) = 3$$

The procedure MIXCOLUMNS operates column by column, applying a well-chosen matrix multiplication.

In essence, AES is a complicated transformation of *state* based on *key*. Why this transformation is thought to be computationally secure is beyond the scope of this text. We just note that it has been studied by many cryptographers and it is believed to secure.

11.8.3.2 Cipher-Block Chaining

With block ciphers, the same input with the same key generates the same output. Thus, one must be careful in using a block cipher for encryption. For example, if the adversary knows that the plaintext is formatted for a printer and each line starts with 16 blanks, then the line breaks will be apparent in the ciphertext because there will always be an 8-byte block of blanks, enciphered the same way. Knowing the number of lines in the text and the length of each line may be usable for frequency analysis to search for the shared-secret key.

A good approach to constructing ENCRYPT using a block cipher is cipher-block chaining. *Cipher-block chaining (CBC)* randomizes each plaintext block by XOR-ing it with the previous ciphertext block before transforming it (see Figure 11.9). A dummy, random, ciphertext block, called the initialization vector (or IV) is inserted at the beginning.

More precisely, if the message has blocks $M_1, M_2, \ldots, M_n$, ENCRYPT produces the ciphertext consisting of blocks $C_0, C_1, \ldots, C_n$ as follows:

$C_0 = IV$ and $C_i \leftarrow \text{BC}\ (M_i \oplus C_{i-1}, key)$ for $i = 1, 2, \ldots, n$

where BC is some block cipher (e.g., AES).

To implement DECRYPT, one computes:

$M_i \leftarrow C_{i-1} \oplus \text{BC}\ (C_i, key)$

CBC has *cascading change propagation* for the plaintext: changing a single message bit (say in M_i), causes a change in C_i, which causes a change in C_{i+1}, and so on. CBC's cascading change property, together with the use of a random IV as the first ciphertext

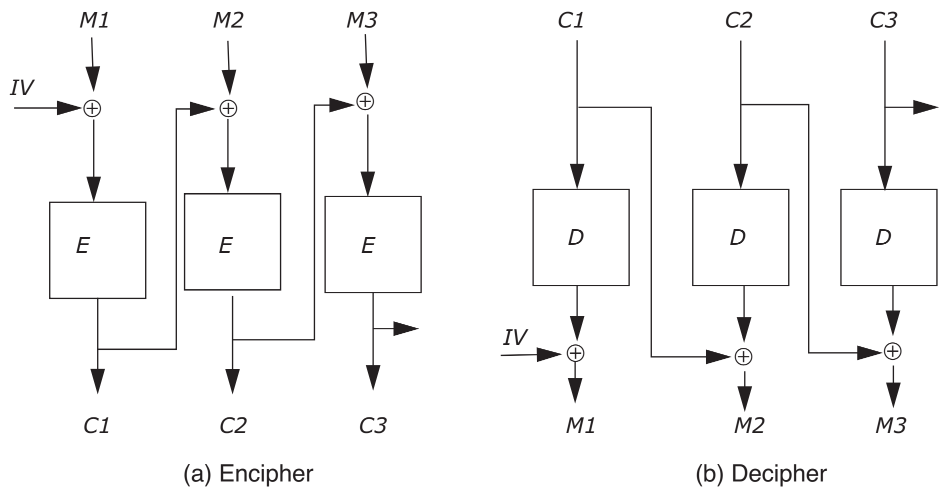


FIGURE 11.9

Cipher-block chaining.

block, implies that two encryptions of the same message with the same key will result in entirely different-looking ciphertexts. The last ciphertext block C_n is a complicated key-dependent function of the IV and of all the message blocks. We will use this property later.

On the other hand, CBC has limited change propagation for the ciphertext: changing a bit in ciphertext block C_i causes the receiver to compute M_i and M_{i+1} incorrectly, but all later message blocks are still computed correctly. Careful study of Figure 11.9 should convince you that this property holds.

Ciphers with limited change propagation have important applications, particularly in situations where ciphertext bits may sometimes be changed by random network errors and where, in addition, the receiving application can tolerate a moderate amount of consequently modified plaintext.

11.8.4 Computing a Message Authentication Code

So far we used ciphers for only confidentiality, but we can use ciphers also to compute authentication tags so that the receiver can detect if an adversary has changed any of the bits in the ciphertext. That is, we can use ciphers to implement the SIGN and VERIFY interface, discussed in Section 11.2. Using shared-secret cryptography, there are two different approaches to implementing the interface: 1) using a block or stream cipher or 2) using a cryptographic hash function. We discuss both.

11.8.4.1 MACs Using Block Cipher or Stream Cipher

CBC-MAC is a simple message authentication code scheme based on a block cipher in CBC mode. To produce an authentication tag for a message *M* with a key *k*, SIGN pads the message out to an integral number of blocks with zero bits, if necessary, and transforms the message *M* with cipher-block chaining, using the key *k* as the initialization vector (IV). (The key *k* is an authentication key, different from the encryption key that the sender and receiver may also use.) All ciphertext blocks except the last are discarded, and the *last* ciphertext block is returned as the value of the authentication tag (the MAC). As noted earlier, because of cascading change propagation, the last ciphertext block is a complicated function of the secret key and the entire message.

VERIFY recomputes the MAC from *M* and key *k* using the same procedure that SIGN used, and compares the result with the received authentication tag. An adversary cannot produce a message *M* that the receiver will believe is authentic because the adversary doesn't know key *k*.

One can also build SIGN and VERIFY using stream ciphers by, for example, using the cipher in a mode called *cipher-feedback (CFB)*. CFB works like CBC in the sense that it links the plaintext bytes together so that the ciphertext depends on all the preceding plaintext. For the details consult the literature.

11.8.4.2 MACs Using a Cryptographic Hash Function

The basic idea for computing a MAC with a cryptographic hash function is as follows. If the sender and receiver share an authentication key *k*, then the sender constructs a MAC for a message *M* by computing the cryptographic hash of the concatenated message *k* + *M*: HASH (*k* + *M*). Since the receiver knows *k*, the receiver can recompute HASH (*k* + *M*) and compare the result with the received MAC. Because an adversary doesn't know *k*, the adversary cannot forge the MAC for the message *M*.

This basic idea must be refined to make the MAC secure because without modifications it has problems. For example, Lucifer can add bytes to the end of the message without the receiver noticing. This attack can perhaps be countered with adding the length of the message to the beginning of the message. Cryptographers have given this problem a lot of attention and have come up with a construction, called *HMAC* [Suggestions for Further Reading 11.5.5], which is said to be as secure as the underlying cryptographic hash function. HMAC uses two strings:

- *innerpad*, which is the byte 36_{hex} repeated 64 times
- *outerpad*, which is the byte $5C_{hex}$ repeated 64 times

Using these strings, HMAC computes the MAC for a message *M* and an authentication key *k* as follows:

HASH ((*k* ⊕ *outerpad*) + HASH ((*k* ⊕ *innerpad*) + *M*))

To compute the XOR, HMAC pads *k* with enough zero bytes to make it of length 64. If *k* is longer than 64 bytes, HMAC uses HASH (*k*), padded with enough zero bytes to make the result of length 64 bytes.

Sidebar 11.7: Secure Hash Algorithm (**SHA**) SHA[*] is a family of cryptographic hash algorithms. SHA-1 takes as input a message of any length smaller than 2^{64} bits and produces a 160-bit hash. It is cryptographic in the sense that given a hash value, it is computationally infeasible to recover the corresponding message or to find two different messages that produce the same hash.

SHA-1 computes the hash as follows. First, the message being hashed is padded to make it a multiple of 512 bits long. To pad, one appends a 1, then as many 0's as necessary to make it 64 bits short of a multiple of 512 bits, and then a 64-bit big-endian representation of the length (in bits) of the unpadded message. The padded string of bits is turned into a 160-bit value as follows.

The message is split into 512-bit blocks. Each block is expanded from 512 bits (16 32-bit words M) to 80 32-bit words as follows (W(t) is the t-th word):

$$W(t) = \begin{cases} M_t, & \text{for } t = 0 \text{ to } 15 \\ (W(t-3) \oplus (W(t-8) \oplus (W(t-14) \oplus (W(t-16) <<< 1 & \text{for } t = 16 \text{ to } 79 \end{cases}$$

where <<< is a left circular shift.

SHA uses four nonlinear functions and four 32-bit constants. The four functions are

$$F(t, x, y, z) = \begin{cases} (X \,\&\, Y) \mid ((\sim X) \,\&\, Z), & \text{for } t = 0 \text{ to } 19 \\ (X \oplus Y \oplus Z), & \text{for } t = 20 \text{ to } 39 \\ (X \,\&\, Y) \mid (X \,\&\, Z) \mid (Y \,\&\, Z), & \text{for } t = 40 \text{ to } 59 \\ X \oplus Y \oplus Z, & \text{for } t = 60 \text{ to } 79 \end{cases}$$

The constants are

$$K(t) = \begin{cases} 5A827999_{hex}, \text{ for } t = 0 \text{ to } 19 & \text{// 2.5/4 in hex} \\ 6ED9EBA1_{hex}, \text{ for } t = 20 \text{ to } 39 & \text{// 3.5/4 in hex} \\ 8F1BBCDC_{hex}, \text{ for } t = 40 \text{ to } 59 & \text{// 5.5/5 in hex} \\ CA62C1D6_{hex}, \text{ for } t = 60 \text{ to } 79 & \text{// 10.5/4 in hex} \end{cases}$$

(*Sidebar continues*)

[*] Secure hash standard, *Federal Information Processing Standards Publications (FIPS PUBS) 180-1*, National Institute of Standards and Technology (NIST), April 1995.

HMAC can be used with any good cryptographic hash function. Sidebar 11.7 describes SHA-1, a widely used cryptographic hash function. Even though SHA-1 must have collisions, no one has uncovered an example of one so far. Recent findings (February 2005) suggest weaknesses in SHA-1 and National Institute for Standards and Technology is recommending switching to longer versions named SHA-256 and SHA-512. Some cryptographers are recommending that research on designing cryptographic hash functions should start over.

SHA uses five 32-bit variables (160 bits) to compute the hash. They are initialized and copied into 5 temporary variables:

$a \leftarrow A \leftarrow 67452301_{hex}$
$b \leftarrow B \leftarrow \mathrm{EFCDAB89}_{hex}$
$c \leftarrow C \leftarrow \mathrm{98BADCFE}_{hex}$
$d \leftarrow D \leftarrow 10325476_{hex}$
$e \leftarrow E \leftarrow \mathrm{C3D2E1F0}_{hex}$

The 160-bit hash value for a message is now computed as follows:

```
1    for each 512-bit block of M do
2        for t from 0 to 79 do
3            x ← (a <<< 5) + F(t, b, c, d) + e + W(t) + K(t)
4            e ← d
5            d ← c
6            c ← b <<< 30
7            b ← a
8            a ← x
9        A ← A + a; B ← B + b; C ← C + c; D ← D + d; E ← E + e
10   hash = A + B + C + D + E                    // concatenate A, B, C, D, and E
```

Other hashes in the SHA family are similar in spirit, but have different constants, word sizes, and produce hash values with more bits. For example, SHA-256 has a different W, F, and produces a 256-bit value.The justification for the SHA family of hashes is outside the scope of this text.

11.8.5 A Public-Key Cipher

The ciphers described so far are shared-secret ciphers. Both the sender and receiver must know the shared secret key. Public-key ciphers remove this requirement, which opens up new kinds of applications, as the main body of the chapter described. The literature contains several public-key ciphers. We explain the first invented one because it is easy to explain, yet is still believed to be secure.

11.8.5.1 Rivest-Shamir-Adleman (RSA) Cipher

The security of the RSA cipher relies on a simple-to-state (but hard to solve) well-known problem in number theory [Suggestions for Further Reading 11.5.1]. RSA was developed at M.I.T. in 1977 (patent number 4,405,829), and is named after its inventors: *Rivest, Shamir, and Adleman (RSA)*. It is based on properties of prime numbers; in particular, it is computationally expensive to factor large numbers (for ages mathematicians have been trying to come up with efficient algorithms with little success), but much cheaper to find large primes.

The basic idea behind RSA is as follows. Initially you choose two large prime numbers (p and q, each larger than 10^{100}). Then compute $n = p \times q$ and $z = (p - 1) \times (q - 1)$, and find a large number d that is relatively prime to z. Finally, find an e such that $e \times d = 1$ (*modulo* z). After finding these numbers once, you have two keys, (e, n) and (d, n), which are hard to derive from each other, even though n is public.

For now assume that the message to be transformed using RSA has a value P that is greater than or equal to zero and smaller than n. (Sections 11.8.5.2 and 11.8.5.3 discuss how to use RSA for signatures and encryption of any message in more detail.) The cipher C is computed by raising P to the power e: P^e (modulo n). To decipher, we compute C to the power d: C^d (modulo n).

The reason this works is as follows. $C^d = P^{ed} = P^{k(p-1)(q-1)+1}$, since $e \times d = 1$ (*modulo* z). Now, $P^{k(p-1)(q-1)+1} = P \times P^{k(p-1)(q-1)} = P \times P^0 = P \times 1 = P$. The theorem that the exponent $k(p - 1)(q - 1) = 0$ (*modulo* n) is a result by Euler and Fermat (see I. Niven and H.S. Zuckerman, *An introduction to the Theory of Numbers*, Wiley, New York, 1980).

An example with concrete numbers may illuminate the abstract mathematics. If one chooses $p = 47$ and $q = 59$, then e is 17 and $d = 157$ because $e \times d = 1$ (modulo 2668). This gives us two keys: (17, 2773) and (157, 2773). Now we can transform any P with a value between 0 and 2773. For example, if P is 31, C is $587 = 31^{17}$ (modulo 2773). To reverse the transform, we compute $587^{157} = 31$ (modulo 2773).

One way to break this scheme is to factor the modulus (n). In 1977 Ron Rivest (the R in RSA) estimated that factoring a 125-digit decimal number would take 40 quadrillion years, using the best known algorithms and state-of-the-art hardware running at 1 million instructions per second*. To test this claim and to encourage research into computational number theory and factoring, RSA Security, the company commercializing RSA, has posted several products of two primes, also called RSA numbers, as factoring challenges. Understanding the speed at which factoring can be done helps in choosing a suitable key length for a desired level of security.

In 1994, a group of researchers under the guidance of A.J. Lenstra factored a 129-digit decimal RSA number in 8 months using the Internet as a parallel computer, without paying for the cycles†. It required 5,000 MIPS years (i.e., 5,000 one-million-instructions-per-second computers each running for one year). Rivest's calculation is an example of the hazards involved in estimating an historic work factor. Better algorithms have been developed, allowing the computation to be performed in only 5,000 MIPS years instead of 40 quadrillion MIPS years, and communication technology has improved substantially, allowing a 5,000 or more computers to be harnessed to perform that much computation in only one year.

In November 2005, the RSA challenge number of 193 decimal digits was factored in 3 months using even better algorithms and faster computers (80 2.2 Gigahertz Opteron

* Martin Gardner, *Mathematical games: A new kind of cipher that would take million of years to break*, Scientific American 237, pages 120–124, August 1977.

† K. Leutwyler, *Superhack: forty quadrillion years early, 129-digit code is broken*, Scientific American, 271, 17–20, 1994.

processors). A 193 decimal digit number is 640 binary bits. Currently it is considered secure to use 1024-bit RSA numbers as keys. The RSA challenge numbers of 704, 768, 896, 1024, 1536, and 2048 bits are still open.

The security of RSA is based on its historical work factor. At this point, there are no known algorithms for factoring large numbers quickly. Although several other public-key ciphers exist, some of which are not covered by patents, to date no public-key system has been found for which one can *prove* a sufficiently large lower bound on the work factor. The best statement one can make now is the work factor based on the best known algorithms. It might be possible that some day a technique is discovered that may lead to fast factoring (e.g., using quantum computation), and thereby undermine the security of RSA.

RSA needs prime numbers; fortunately, there are many of them and generating them is much easier than factoring a product of two primes: "is n prime?" is a much easier question than "what are the factors of n?" There are approximately $n/\ln(n)$ prime number less than or equal to n. Thus, for numbers that can be expressed with 1024 bits or fewer, there are approximately 2^{1021} prime numbers. Therefore, we won't run out of prime numbers, if everyone needs two prime numbers different from everyone else's primes. In addition, an adversary won't have a lot of success creating a database that contains all prime numbers because there are so many.

11.8.5.2 Computing a Digital Signature

An important use of public-key ciphers is to implement the SIGN and VERIFY interface. If this interface is implemented using public-key cryptography, the authentication tag is called a digital signature. The basic idea—which needs refinement to be secure—for computing an RSA digital signature is as follows. SIGN produces an authentication tag by raising M to the private exponent. VERIFY raises the authentication tag to the public exponent, compares the result to the received message, and returns ACCEPT if they match and REJECT if don't.

The implementation doesn't always guarantee authenticity, however. For example, if Lucifer succeeds in having Alice sign messages M_1 and M_2, then he can claim that Alice also signed M_3, where M_3 is the product of M_1 and M_2: $(M_3)^d = (M_1 \times M_2)^d = M_1{}^d \times M_2{}^d$ *(modulo n)*. Thus, if Lucifer sends M_3 to Bob, when Bob uses Alice's public key to verify message M_3 that message will appear to have been signed by Alice.

To avoid this problem (and some others) SIGN usually computes a cryptographic hash of the message, and creates an authentication tag by raising this hash to the private exponent. This also has the pleasant side effect that it simplifies signing large messages because n only has to be larger than the value of the hash output, and we don't have to worry about splitting the message into blocks and signing each block. Upon receipt, VERIFY recomputes the hash from the received version of the message, raises the hash to the public exponent, and compares the result with the received authentication tag.

Using a cryptographic hash helps in constructing a secure SIGN and VERIFY but isn't sufficient either. There is a substantial literature that presents even better schemes that also address other subtle issues that come up in the design of a good digital signature scheme.

11.8.5.3 A Public-Key Encrypting System

ENCRYPT and DECRYPT can also be implemented using public-key cryptography, but because operations in public-key systems are expensive (e.g., exponentiation in RSA instead of XOR in RC4), public-key implementations of ENCRYPT and DECRYPT are used sparingly. As described in Section 11.5, public-key encryption is used only to encrypt a newly-minted shared-secret key during the set up of a connection between a sender and a receiver, and then that secret-secret key is used for shared-secret encryption of further communication between the sender and the receiver. For example, SSL/TLS, which is described in the next section, uses this approach.

The basic idea, which needs refinement to be secure, for implementing ENCRYPT and DECRYPT using RSA is as follows. Split the message M into fixed size blocks P so that the value of P is smaller than n, then ENCRYPT raises P to the *public* exponent (d). DECRYPT raises the encrypted block to the *private* exponent (e). This order is exactly the opposite of the one for SIGN; SIGN raises to the private exponent and VERIFY raises to the public exponent.

That the order is the opposite doesn't matter because RSA is reversible. Since $(M^d)^e = (M^e)^d = M^{ed}$ (modulo n), one can raise to the public exponent (e) first, and raise to the private exponent (d) second, or vice versa, and either way obtain M back. It is claimed that the security of RSA is equally good both ways.

This basic implementation is relatively weak; there are a number of well-known attacks if the RSA cipher is used by itself for encrypting. To counter these attacks, ENCRYPT should pad short blocks with independent randomized variables so that the value of P is close to n, and then raise the padded P to the public exponent. In addition, ENCRYPT should run the message through what is called an *all or nothing transform (AONT)*. An AONT is a non-secret, reversible transformation of a message that ensures that the receiver must have *all* of the bits of the transformed message in order to recover *any* of the bits of the original message. Thus, an adversary cannot launch an attack by just concentrating on individual blocks of the message. Readers should consult the literature to learn what other measures are necessary to obtain a good implementation of ENCRYPT and DECRYPT using RSA

11.9 .Summary

Section 11.1 of this chapter provided a general perspective on how to think about building secure systems, including a set of design principles, and was then followed by 7 sections of details. One might expect, after reading all this text, that one should now know how to build secure computer systems.

Unfortunately, this expectation is incorrect. Section 11.11 relates several war stories of security system failures that have occurred over a 40-year time span. Failures from decades past might be explained as mistakes while learning that have helped lead to the better understanding now provided in this chapter. But most of the design principles presented in this chapter were formulated and published back in 1975. The section includes several examples of recent failures, which are reinforced by regular reports in the media about yet another virus, worm, distributed denial-of-service attack, identity theft, stolen credit card, or defaced Web site. If we know how to build secure systems, why does the real world of the Internet, corporate services, desktop computers, and personal computers seem to be so vulnerable?

The question does not have a single, simple answer. A lot of different things are tangled together. There are honest and dishonest opinions that the security problem isn't that important, and thus it is unnecessary to get it right. Since organizations prefer not to disclose security problems, it is even difficult to establish what the cost of a security compromise is. Some problems are due to designers just building systems that are too complex. Some problems come from lack of awareness. Some problems are due to designers attempting to build secure systems on Internet time, and not taking the time to do it properly. Some problems arise from ignorance. To get a handle on this general question it is helpful to split the question into several more specific questions:

- The Internet protocols do not provide a default of authentication of message source and privacy of message contents. Why? As discussed in Section 11.1, when the Internet was designed processors weren't fast enough to apply cryptographic transformations in software, the deployment of cryptographic-transformation hardware was hindered by government export regulations, and good key distribution protocols hadn't been designed yet. Since the Internet was originally primarily used by a cooperative set of academics, this lack of security was also not a serious omission. By the time it became economically feasible to do ciphers in software, key distribution was understood, and government export regulations were relaxed, the insecure protocols were so widespread that it was too hard to do a retrofit. Section 11.10 describes one of the now most widely-used secure protocols for Web transactions on the Internet.

- Personal computer systems do not come with enforced modularity that creates strong internal firewalls between applications. Why? The main reasons are keeping the cost low and naivité. Initially PCs were designed to be inexpensive computers for personal use. Few people, or perhaps nobody, anticipated that the rapid improvements in technology would lead to the current situation where PCs are the dominant platform for all computing. Furthermore, as explained in Section 5.7, it took the PC designers and operating system vendors for PCs several iterations to get the designs for enforced modularity correct. Currently vendors are struggling to make PCs easier to configure and manage so that they aren't as vulnerable to attacks.

- Inadequately secured computers are attached to the Internet. Why? Most computers on the Internet are personal computers. When originally conceived personal computers were for *personal* computing, which at the time was editing documents and playing games. Network attacks were impossible, and thus network security was just not a requirement. But the value of being attached to the Internet grew rapidly as the number of available services increased. The result was that most users pursued that evident value, without much concern about the risks, which at first, despite warnings, seemed mostly hypothetical.

- UNIX systems, commonly used as services, have enforced modularity, but many UNIX services were originally (and some still seem to be) vulnerable to buffer-overrun attacks (see Sidebar 11.4), which subvert modular boundaries. Why are these buffer overruns so difficult to eradicate? As explained in the sidebar, the main reason is the success of the C programming language, which was not designed to check array bounds. Much system software is written in C and has been deployed successfully for decades. A drastic change to the C programming language (or its library) is now difficult because change would break most existing C programs. As a result, each service program must be fixed individually.
- Why isn't software verified for security? Recent progress has been made in analyzing cryptographic algorithms, checking software for common security problems, and verifying security protocols within an adversary model. All these techniques are useful for verifying properties of a system, but they don't prove that a system is secure. In general, we don't know what properties to verify to proof security.
- Why don't basic economic principles reward the company that produces secure systems? For example, why don't customers buy the more secure products, why don't firms that insure companies against security attacks cause software to be better, etc.? Economics is indeed a factor in information security, but the economic factors interact in surprising ways, and these questions don't have simple answers. Sidebar 11.8 summarizes some of the interactions, and their consequences.
- Why doesn't security certification help more? There are no adequate standards for what kind of attacks a minimal secure system should protect against. Standards that do exist for security requirements are out of date because they don't cover network security. Standardization organizations have a difficult time keeping up with the rate of change in technology.
- Many secure systems require a public key infrastructure, but no universal PKI exists. Why? PKIs exist only in isolated islands, limited to a single institution or application. For example, there is a specialized PKI that supports only the use of SSL/TLS in the World-Wide Web. Why doesn't a universal one exist? A reason is that realistically it is difficult to develop a single one that is satisfactory to everyone. Anyone trying to propose one has run into political and economic problems.
- Many organizations have installed network firewalls between their internal network and the Internet. Do they really help? Yes, but in a limited way, and they have the danger of creating a false sense of security. Because desktop and service operating systems have so many security problems (for the reasons mentioned

Sidebar 11.8: Economics of computer security Why is the company that produces software with fewest security vulnerabilities not the most successful one? Ross Anderson has studied some of the many economic factors in play and analyzed their impact on information security[*]. First, there are misaligned incentives. For example, under U.S. law it is the bank's burden to prove that a fraudulent withdrawal at an automated teller machine (ATM) is the customer's fault, but under U.K. law, it is the customer's burden to prove that a fraudulent ATM withdrawal is the bank's fault. One might think that U.K. banks spend less money on security, but Anderson reports that the opposite is true: U.K. banks spend more money on security and experience more fraud. It appears that U.K. banks became lazy and careless, knowing that customers complaints of fraud did not require a careful response on their part.

Second, there are network externalities: the larger the network of developers and users the more valuable that network is to each of its members. Selecting a new operating system partly depends on the number of other people who made the same choice (i.e., because it simplifies exchanging files in closed formats). While an operating system vendor is building market dominance, it must appeal to vendors that complement the operating system as well as the customers. Since security could get in the way of vendors complementing the operating system, operating system vendors have a strong incentive to ignore security in the beginning in favor of features that might help obtain market leadership, and address security later. Unfortunately, adding on security later is never as good as security that is part of the original design.

Third, there are security externalities. For example, if a PC owner considers spending $40 to buy a good firewall, that owner is not the primary beneficiary; what the firewall really protects is targets like Google and Microsoft because because by avoiding becoming a bot the firewall installer is helping prevent distributed denial-of-service attacks on *other* sites. Thus the incentive to purchase and install the firewall is low. Bot herders understand this phenomenon well, so they are careful not to attack the files stored on the bots themselves or otherwise give the owner of the bot any incentive to install the firewall.

Finally, security risks are interdependent. A firm's computer infrastructure is often connected to infrastructure under control of others (e.g., the Internet) or uses software written by others, and so the firm's efforts may be undermined by security failures elsewhere. In addition, attacks often exploit a weakness in a system used by many firms. This interdependence makes security risks unattractive to insurers, and as a result there are no market pressures from them.

The impact of economics on computer security is an emerging field of study, and as it develops the explanations might change, the actions of companies may change, but for now it is clear simple economic analysis may miss important interactions.

[*] Ross Anderson and Tyler Moore, *The Economics of Information Security*, Science, 314 (5799), Oct. 2006, pp. 610–613.

above), end-to-end security is difficult to achieve. If firewalls are properly deployed they can keep the external, low-budget adversaries away from the vulnerable internal computers. But firewalls don't help against inside adversaries, nor against adversaries that find ways around the firewall to reach the inside network from the outside (e.g., by using the internal wireless network from outside, dialing into a desktop computer that is connected both to the internal network and the telephone system, by hitching rides on data or program files that inside users download through the firewall or load from detachable media, etc.)

- One hears reports that wireless network (WiFi or 802.11b/g) security is weak. This is a relatively new design. Why is it so vulnerable? As mentioned in Section 11.1, one reason appears to be that the security design was done by a committee that was expensive to join, and that only committee members were allowed to review the design. As a result, although the design was nominally open, it was effectively closed, and few security experts actually reviewed the design until after it was deployed, at which point several security weaknesses (for an example see page 11–51) were identified.
- Cable TV scrambling systems, DSS (Satellite TV) security, the CSS system for protecting DVD movie content, and a proposed music watermarking system, were all compromised almost immediately following their deployment. Why were these systems so easy to break? Many of these systems used a closed design and the right people didn't review it. When the system was deployed, experts investigated the design and immediately found problems.

In addition to these more specific reasons, there are two general problems that contribute to the large number of security vulnerability. First, the rate of innovation is high in computer systems. New technologies emerge and are deployed must faster than their designers anticipated and the lack of a security plan in the initial versions becomes a problem suddenly. Furthermore, successful technologies become deployed for applications that the designer didn't anticipate and often turn out to have additional security requirements. Second, no one has a recipe for building secure systems because these systems try to achieve a negative goal. Designing and implementing secure systems requires experts that are extremely careful, have an eye for detail, and exhibit a paranoid attitude. As long as the rate of innovation is high and there is no recipe for engineering secure systems, it is likely that security exploits will be with us. The TLS example in Section 11.10 describes a successful secure protocol (with some growing pains to get it right) and the examples in Section 11.11 illustrate many ways to get things wrong.

11.10 Case Study: Transport Layer Security (TLS) for the Web

The Transport Layer Security (TLS) protocol* is a widely used security protocol to establish a secure channel (confidential and authenticated) over the Internet. The TLS protocol is at the time of this writing a proposed international standard. TLS is a version of the Socket Security Layer (SSL) protocol, defined by Netscape in 1999, so current literature frequently uses the name "SSL/TLS" protocol. The TLS protocol has some improvements over the last version (3) of the SSL protocol, and this case study describes the TLS protocol, version 1.2.

The TLS protocol allows client/service applications to communicate in the face of eavesdroppers and adversaries who would tamper with and forge messages. In the handshake phase, the TLS protocol negotiates, using public-key cryptography, shared-secret keys for message authentication and confidentiality. After the handshake, messages are encrypted and authenticated using the shared-secret keys. This case study describes how TLS sets up a secure channel, its evolution from SSL, and how it authenticates principals.

11.10.1 The TLS Handshake

The TSL protocol consists of several protocols, including the record protocol which specifies the format of messages between clients and services, the alert protocol to communicate errors, the change cipher protocol to apply a cipher suite to messages sent using the record layer protocol, and several handshaking protocols. We describe the handshake protocol for the case where an anonymous user is browsing a Web site and requires service authentication and a secure channel to that service.

Figure 11.10 shows the handshake protocol for establishing a connection from a client to a server. The CLIENTHELLO message announces to the service the version of the protocol that the client is running (SSL 2.0, SSL 3.0, TLS 1.0, etc.), a random sequence number, and a prioritized set of ciphers and compression methods that the client is willing to use. The *session_id* in the CLIENTHELLO message is null if the client hasn't connected to the service before.

The service responds to the CLIENTHELLO message with 3 messages. It first replies with a SERVERHELLO message, announcing the version of the protocol that will be used (the lower of the one suggested by the client and the highest one supported by the service), a random number, a session identifier, and the cipher suite and compression method selected from the ones offered by the client.

To authenticate the service to the client, the service sends a SERVERCERTIFICATE message. This message contains a chain of certificates, ordered with the service's certificate first followed by any certificate authority certificates proceeding sequentially upward. Usually the list contains just two certificates: a certificate for the public key of the service and a certificate for the public key of the certification authority. (We will discuss certificates in more detail in Section 11.10.3.)

After the service sends its certificates, it sends a SERVERHELLODONE message to indicate that it is done with the first part of the handshake. After receiving this message and after satisfactorily verifying the authenticity of the service, the client generates a 48-byte

* Tim Dierks and Eric Rescorla. The Transport Layer Security (TLS) protocol Version 1.2. *RFC 4346*. November 2007.

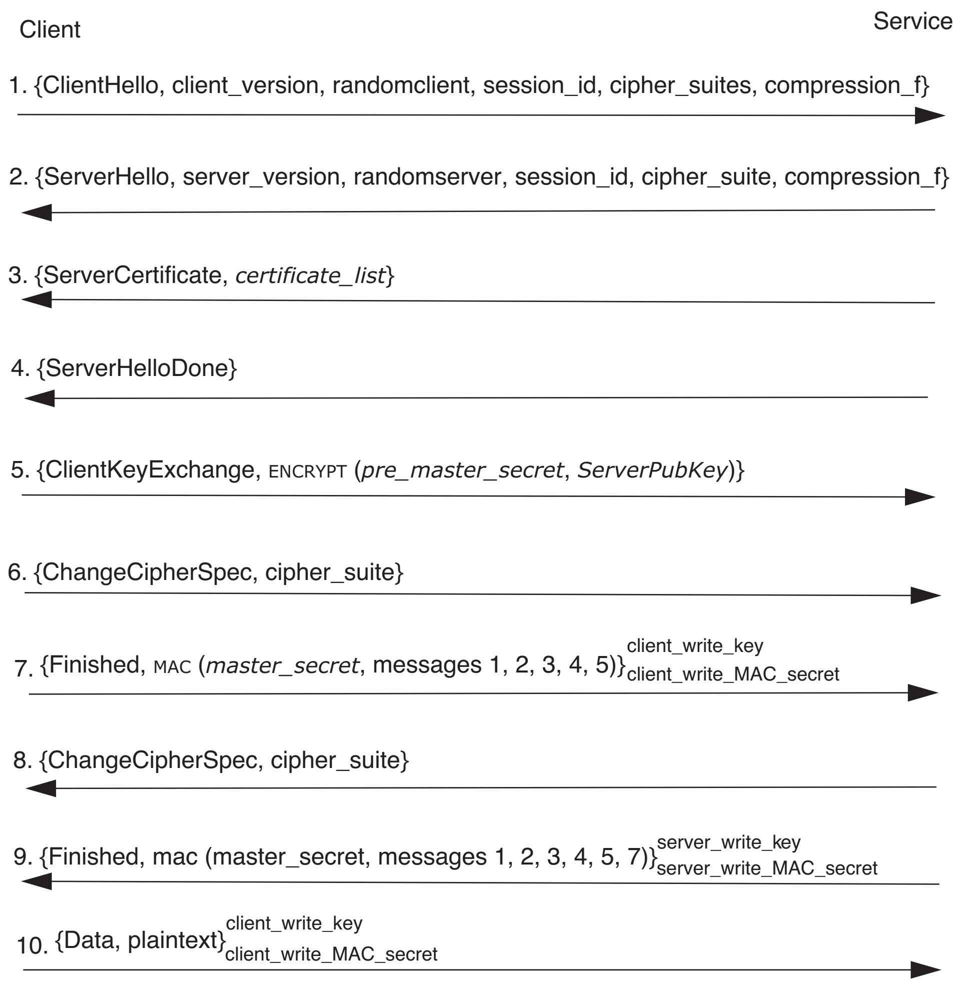

Client
Service
1. {ClientHello, client_version, randomclient, session_id, cipher_suites, compression_f}
2. {ServerHello, server_version, randomserver, session_id, cipher_suite, compression_f}
3. {ServerCertificate, certificate_list}
4. {ServerHelloDone}
5. {ClientKeyExchange, ENCRYPT (pre_master_secret, ServerPubKey)}
6. {ChangeCipherSpec, cipher_suite}
7. {Finished, MAC (master_secret, messages 1, 2, 3, 4, 5)}client_write_key client_write_MAC_secret
8. {ChangeCipherSpec, cipher_suite}
9. {Finished, mac (master_secret, messages 1, 2, 3, 4, 5, 7)}server_write_key server_write_MAC_secret
10. {Data, plaintext}client_write_key client_write_MAC_secret

FIGURE 11.10

Typical TLS exchange of handshake protocol messages.

pre_master_secret. TLS supports multiple public-key systems and depending on the choice of the client and service, the *pre_master_secret* is communicated to the service in slightly different ways.

In practice, TLS typically uses a public-key system, in which the client encrypts the *pre_master_secret* with the public key of the service found in the certificate, and sends the result to the service in the CLIENTKEYEXCHANGE message. The *pre_master_secret* thus can be decrypted by any entity that knows the private key that corresponds to the public key in the certificate that the service presented. The security of this scheme therefore depends on the client carefully verifying that the certificate is valid and that it corresponds to the desired service. This point is explored in more detail in Section 11.10.3, below.

The *pre_master_secret* is used to compute the *master_secret* using the service and client nonce ("+" denotes concatenation):

master_secret ← PRF (*pre_master_secret*, "master secret", $random_{client}$+ $random_{server}$)

PRF is a pseudorandom function, which takes as input a secret, a label, and a seed. As output it generates pseudorandom bytes. TLS assigns the first 48 bytes of the PRF output to the *master_secret*. The TLS version 1.2 uses a PRF function that is based on the HMAC construction and the SHA-256 hash function (see Section 11.8 for the HMAC construction and the SHA family of hash functions).

It is important that the *master_secret* be dependent both on the *pre_master_secret* and the random values supplied by the service and client. For example, if the random number of the service were omitted from the protocol, an adversary could replay a recorded conversation without the service being able to tell that the conversation was old.

After the *master_secret* is computed, the *pre_master_secret* should be deleted from memory, since it is no longer needed and continuing to store it would just create an unnecessary security risk.

After sending the encrypted *pre_master_secret*, the client sends a CHANGECIPHERSPEC message. This message[*] specifies that all future message from the client will use the ciphers specified as the encrypting and authentication ciphers.

The keys for message encrypting and authentication ciphers are computed using the *master_secret*, $random_{client}$, and $random_{server}$ (which both the client and the service now have). Using this information a key block is computed:

key_block ← PRF (*master_secret*, "key expansion", $random_{server}$ + $random_{client}$)

until enough output has been produced to provide the following keys:

client_write_MAC_secret[CipherSpec.hash_size]
server_write_MAC_secret[CipherSpec.hash_size]
client_write_key[CipherSpec.key_material]
server_write_key[CipherSpec.key_material]
client_write_IV[CipherSpec.IV_size]
server_write_IV[CipherSpec.IV_size]

The first 4 variables are the keys for authentication and confidentiality, one for each direction. The last 2 variables are the initialization vectors, one for each direction, for ciphers using CBC mode (see Section 11.8). These variables together are the state necessary for the client and the service to communicate securely.

Now the client sends a FINISHED message to announce that it is done with the handshake. The FINISHED message contains at least 12[†] bytes of the following output:

[*] The TLS standard considers ChangeCipherSpec not part of the handshake protocol, but part of the Change Cipher Spec protocol, even though the handshake protocol uses it.

[†] Clients may specify in the HELLO message that they prefer more bytes.

PRF (*master_secret*, *finish_label*, HASH (*handshake_messsages*))

The FINISHED message is a verifier of the protocol sequence so far (the value of all messages starting at the CLIENTHELLO message, but not including the FINISHED message). The client use the value "client finished" for *finish_label*. HASH is the same hash function used for the PRF, SHA-256. If the service verifies the hash, the service and client agree on the protocol sequence and the *master_secret*. TLS encrypts and authenticated the FINISHED message using the cipher suite that the client and service agreed on in the HELLO messages.

After the service receives the client's FINISHED message, it sends a CHANGECIPHERSPEC message, informing the client that all subsequent messages from service to client will be encrypted and authenticated with the specified ciphers. (The client and service can use different ciphers for their traffic.) Like the client, the service concludes the handshake with a FINISHED message, but uses the value "server finished" for *finish_label*. After both finish messages have been received and checked out correctly, the client and service have a secure (that is, encrypted and authenticated) channel over which they can carry on the remainder of their conversation.

11.10.2 Evolution of TLS

The TLS handshake protocol is more complicated than some of the protocols that we described in this chapter. In a large part, this complexity is due to all the options TLS supports. It allows a wide range of ciphers and key sizes. Service and client authentication are optional. Also, it supports different versions of the protocol. To support all these options, the TLS protocol needs a number of additional protocol messages. This makes reasoning about TLS difficult, since depending on the client and service constraints, the protocol has a different set of message exchanges, different ciphers, and different key sizes. Partly because of these features the predecessors of TLS 1.2, the earlier SSL protocols, were vulnerable to new attacks, such as cipher suite substitution and version rollback attacks.

In version 2 of SSL, the adversary could edit the CLIENTHELLO message undetected, convincing the service to use a weak cipher, for example one that is vulnerable to brute-force attacks. SSL Version 3 and TLS protect against this attack because the FINISHED message computes a MAC over all message values.

Version 3 of SSL accepts connection requests from version 2 of SSL. This opens a version-rollback attack, in which an adversary convinces the service to use version 2 of the protocol, which has a number of well-documented vulnerabilities, such as the cipher substitution attack. Version 3 appears to be carefully designed to withstand such attacks, but the specification doesn't forbid implementations of version 2 to resume connections that were started with version 3 of the protocol. The security implications of this design are unclear.

One curious aspect of version 3 of the SSL protocol is that the computation for the MAC of the FINISHED messages does not include the CHANGECIPHER messages. As pointed out by Wagner and Schneier, an adversary can intercept the CHANGECIPHER message and delete it, so that the service and client don't update their current cipher suite. Since messages during the handshake are not encrypted and authenticated, this can open a security hole. Wagner and Schneier describe an attack that exploits this observation [Suggestions for Further Reading 11.5.4]. Currently, widely used implementations of SSL 3.0 protect against this attack by accepting a FINISHED message only after receiving a CHANGECIPHER message.

TLS is the international standard version of SSL 3.0, but also improves over SSL 3.0. For example, it mandates that a FINISHED message must follow immediately after a CHANGECIPHER message. It also replaces ad-hoc ways of computing hash functions in various parts of the SSL protocol (e.g., in the FINISHED message and *master_secret*) with a single way, using the PRF function. TLS 1.1 has a number of small security improvements over 1.0. TLS 1.2 improves over TLS 1.1 by replacing an MD5/SHA-1 implementation of PRF with one specified in the cipher suite in the HELLO messages, preferable based on SHA-256. This allows TLS to evolve more easily when ciphers are becoming suspect (e.g., SHA-1).

11.10.3 Authenticating Services with TLS

TLS can be used for many client/service applications, but its main use is for secure Web transactions. In this case, a Web browser uses TLS to set up a message-authenticated, confidential communication connection with a Web service. HTTP requests and responses are sent over this secure connection. Since users typically visit Web sites and perform monetary transactions at these sites, it is important for users to authenticate the service. If users don't authenticate the service, the service might be one run by an adversary who can now record private information (e.g., credit card numbers) and supply fake information. Therefore, a key problem TLS addresses is service authentication.

The main challenge for a client is to convince itself that the service's public key is authentic. If a user visits a Web site, say amazon.com (an on-line book retailer), then a user wants to make sure that the Web site the user connects to is indeed owned by Amazon.com Inc. The basic idea is for Amazon to sign its name with its private key. Then, the client can verify the signed name using Amazon's public key. This approach reduces the problem to securely distributing the public key for Amazon. If it is done insecurely, an adversary can convince the client that the adversary has the public key of Amazon, but substitute the adversary's own public key and sign Amazon's name with the adversary's private key. This problem is an instance of the key-distribution problem, discussed in Section 11.5.

TLS relies on well-known certification authorities for key distribution. An organization owning a Web site buys a certificate from one or more certification authorities. Each authority runs a certification check to validate that the organization is the one it claims to be. For example, a certification authority might ask Amazon Inc. for articles of incorporation to prove that it is the entity it claims to be. After the certification authority has verified the identity of the organization, it issues a certificate. The certificate contains the public key of the organization and the name of the organization, signed with the private

```
structure certificate
    version
    serial_number
    signature_cipher_identifier
    issuer_signature
    issuer_name
    subject_name
    subject_public_key_cipher_identifier
    subject_public_key
    validity_period
```

FIGURE 11.11

Some fields in version 3 of the X.509 certificate

key of the certificate authority. (The service sends the certificates in step 3 of the handshake protocol, described in Section 11.10.1.)

The client verifies the certificate as follows. First, it obtains in a secure way the public key of certification authorities that it is willing to trust. Typically a number of public keys come along with the distribution of a Web browser. Second, after receiving the service certificates, it uses the public keys of the authorities to verify one of the certificates. If one of the certificates verifies correctly, the client can be confident about the name of the organization owning the service. Whether a user can trust the organization that goes by that name is a different question and one that the user must resolve using psychological means.

TLS uses certificates that are standardized by the ISO X.509 standard. Figure 11.11 shows some of the fields in Version 3 of X.509 certificates (the standard specifies them in a different order). The *version* field specifies the version of the certificate (it would be 3 in this example). The *serial_number* field contains a nonce assigned by the issuing certification authority and different for every certificate. The *signature_cipher_identifier* field identifies the algorithm used by the authority to sign this certificate. This information allows a client of the certification authority to know which of several standard algorithms to use to verify the *issuer_signature* field, which contains the value of the certificate's signature. If the signature checks out, the recipient can believe that the information in the certificate is authentic. The *issuer_name* field specifies the real-world name of the certificate authority. The *subject_name* field specifies the real-world name for the principal. The two other subject fields specify the public-key cipher the principal wants to use (say RSA), and the principal's public key.

The *validity_period* field specifies the time for which this signature is valid (the start and expiry dates and times). The *validity_period* field provides a weak method for key revocation. If Amazon obtains a certificate and the certificate is valid for 12 months (a typical number) and if the next day an adversary compromises the private key of amazon.com, then the adversary can impersonate amazon for the next 12 months. To counter this problem a certification authority maintains a certification revocation list, which contains compromised certificates (identified by the certificate's serial number). Anyone can download the certificate revocation list to check if a certificate is on this blacklist. Unfortunately, revocation lists are not in widespread use today. Good certificate revocation procedures are an open research problem.

The crucial security step for establishing a principal's identity is the certification process executed by the certification authority. If the authority issues certificates without checking out the identity of the organization owning the service, the certificate doesn't improve security. In that case, Lucifer could ask the certification authority to create a certificate for Amazon.com Inc. If the authority doesn't check Lucifer's identity, Lucifer will obtain a certificate for Amazon Inc. that binds the name Amazon Inc. to Lucifer's public key, allowing Lucifer to impersonate Amazon Inc. Thus, it is important that the certification authority do a careful job of certifying the principal's identity. A typical certification procedure includes paying money to the authority, sending by surface mail the articles of incorporation (or equivalent) of the organization. The authority will run a partly manual check to validate the provided information before issuing the certificate.

Certification authorities face an inherent conflict between good security and convenience. The procedure must be thorough enough that the certificate means something. On the other hand, the certification procedure must be convenient enough that organizations are able or willing to obtain a certificate. If it is expensive in time and money to obtain a certificate, organizations might opt to go for an insecure solution (i.e., not authenticating their identity with TLS). In practice, certification authorities have a hard time striking the appropriate balance and therefore specialize for a particular market. For example, Verisign, a well-known certification authority, is mostly used by commercial organizations. Private parties who want to obtain a certificate from Verisign for their personal Web sites are likely to find Verisign's certification procedure impractical.

Ford and Baum provide a nice discussion of the current practice for secure electronic commerce using certificate authories, certificates, etc., and the legal status of certificates [Suggestions for Further Reading 1.3.17].

11.10.4 User Authentication

User authentication can in principle be handled in the same way as server authentication. The user could obtain a certificate from an authority testifying to the user's identity. When the server asks for it, the user could provide the certificate and the server could verify the certificate (and thus the user's identity according to a certification authority) by using the public key of the authority that issued the certificate. Extensions of the TLS handshake protocol support this form of user authentication.

In practice, and in particular in the Web, user authentication doesn't rely on user certificates. Some organizations run a certificate authority and use it to authenticate members of their organization. However, often it is too much trouble for a user to obtain a certificate, so few Web users are willing to obtain a certificate. Instead, many servers authenticate users based on the IP address of the client machine or based on shared passphrase. Both methods are currently implemented insecurely.

Using the IP address for authentication is insecure because it is easy for an adversary to spoof an IP address. Thus, when the server checks whether a user on a machine with a particular IP address has access, the server has no guarantees. Typically, this method is used inside an organization that puts all it's machines behind a firewall. The firewall attempts to keep adversaries out of the organization's network by monitoring all network traffic that is coming from the Internet and blocking bad traffic (e.g., a packet that is coming from outside the firewall but an internal IP address).

Passphrase authentication is better. In this case, the user sets up an account on the service and protects it with a passphrase that only the user and the service know. Later when the user visits the service again, the server puts up a login page and asks the user to provide the passphrase. If the passphrase is valid, the server assumes that the user is the principal who created the account.

To avoid having the user to type the password on each request, services can exploit a Web mechanism called *cookies*. A service sends a cookie, a service-specific piece of information, to the user's Web browser, which stores it for us in later requests to the service. The service sends the cookie by including in a response a SET_COOKIE directive containing data to be stored in the cookie. The browser stores the cookie in memory. (In practice, there may be many cookies, so they are named, but for this description, assume that there is only one and no name is needed.) On subsequent calls (i.e., GET or POST) to the service that installed the cookie, the browser sends the installed cookie along with the other arguments to GET or POST.

Web services can use cookies for user authentication as follows. When the user logs in, the service creates a cookie that contains information to authenticate the user later and sends it to the user's browser, which stores it for use in future requests to this service. Every subsequent request from that browser will include a copy of the cookie, and the service can use the information stored in the cookie to learn which user issued this request. If the cookie is missing (for example, the user is using a different browser), the service will return an error to the browser and ask the user to login again. The security of this scheme depends on how careful the service is in constructing the authenticating cookie. One possibility is to create a nonce for a session and sign the nonce with a MAC. Kevin Fu et al. describe some ways to get it wrong and recommend a secure approach*. Problem set *45* explores some of the issues in protecting and authenticating cookies.

Web sites use cookies in many ways. For example, many Web sites uses cookies to track the browsing patterns of returning visitors. Users who want to protect their privacy must disable cookie tracking in their browser.

* K. Fu, E. Sit, K. Smith, and N. Feamster, Dos and don'ts of client authentication on the Web, *Proceedings of the tenth USENIX Security Symposium*, Washington, August 2001.

11.11 War Stories: Security System Breaches

A designer responsible for system security can bring to the job three different, related assets. The first is an understanding of the fundamental security concepts discussed in the main body of this chapter. The second is knowledge of several different real security system designs; some examples have been discussed elsewhere in this chapter and more can be found in the Suggestions for Further Reading. This section concentrates on a third asset: familiarity with examples of real-world breaches of security systems. In addition to encouraging a certain amount of humility, one can develop from these case studies some intuition about approaches that are inherently fragile or difficult to implement correctly. They also provide evidence of the impressive range of considerations that a designer of a security system must consider.

The case studies selected for description all really happened, although inhibitions have probably colored some of the stories. Failures can be embarrassing, have legal consequences, or, if publicized, jeopardize production systems that have not yet been repaired or redesigned. For this reason, many of the cases described here were, when they first appeared in public, sanitized by omitting certain identifying details or adding misleading "facts". Years later, reconstructing the missing information is difficult, as is distinguishing the reality from any fantasy that was added as part of the disguise. To help separate fact from fiction, this section cites original sources wherever they are available.

The case studies start in the early 1960s, when the combination of shared computers and durable storage first brought the need for computer security into focus. In several examples, an anecdote describing a vulnerability discovered and a countermeasure devised decades ago is juxtaposed with a much more recent example of essentially the same vulnerability being again found in the field. The purpose is not to show that there is nothing new under the sun, but rather to emphasize Santayana's warning that "Those who cannot remember the past are condemned to repeat it."*

At the same time it is important to recognize that the rapid improvement of computer hardware technology over the last 40 years has created new vulnerabilities. Technology improvement has provided us with new case studies of security breaches in several ways:

- Adversaries can bring to bear new tools. For example, performance improvements have enabled previously infeasible attacks on security such as brute force key space searches.
- Cheap computers have increased the number of programmers much faster than the number of security-aware programmers.
- The attachment of computer systems to data communication networks has, from the point of view of a potential adversary, vastly increased the number of potential points of attack.

* George Santayana, *The Life of Reason, Volume 1, Introduction and Reason in Common Sense* (Scribner's: 1905)

- Rapid technology change has encouraged giving high priority to rolling out new features and applications, so the priority of careful attention to security suffers.
- Technology improvement has enabled the creation of far more complex systems. Complexity is a progenitor of error, and error is a frequent cause of security vulnerabilities.

Although it is common to identify a single mistake that was the proximate cause of a security breach, if one *keeps digging* it is usually possible to establish that several violations of security principles contributed to making the breach possible, and thus to failure of defense in depth.

11.11.1 Residues: Profitable Garbage

Security systems sometimes fail because they do not protect *residues*, the analyzable remains of a program or data after the program has finished. This general attack has been reported in many forms; adversaries have discovered secrets by reading the contents of newly allocated primary memory, second-hand hard disks, and recycled magnetic tapes as well as by pawing through piles of physical trash (popularly known as "dumpster diving").

11.11.1.1 1963: Residues in CTSS

In the M.I.T. Compatible Time-Sharing System (CTSS), a user program ran in a memory region of an allocated size, and the program could request a change in allocation by calling the operating system. If the user requested a larger allocation, the system assigned an appropriate block of memory. Early versions of the system failed to clear the contents of the newly allocated block, so the residue of some previous program would be accessible to any other program that extended its memory size.

At first glance, this oversight seems to provide an attacker with the ability to read only an uncontrollable collection of garbage, which appears hard to exploit systematically. An industrious penetrator noticed that the system administrator ran a self-rescheduling job every midnight that updated the primary accounting and password files. On the assumption that the program processed the password file by first reading it into primary memory, the penetrator wrote a program that extended its own memory size from the minimum to the maximum, then it searched the residue in the newly assigned area for the penetrator's own password. If the program found that password, it copied the entire memory residue to a file for later analysis, expecting that it might also contain passwords of other users. The penetrator scheduled the program to go into operation just before midnight, and then reschedule itself every few seconds. It worked well. The penetrator soon found in the residue a section of the file relating user names and passwords.*

Lesson: A design principle applies: use *fail-safe defaults*. In this case, the fail-safe default is for the operating system memory allocator to clear the contents of newly-allocated memory.

11.11.1.2 1997: Residues in Network Packets

If one sends a badly formed request to a Kerberos Version 4 server (Sidebar 11.6) describes the Kerberos authentication system), the service responds with a packet containing an error message. Since the error packet was shorter than the minimum frame size, it had to be padded out to reach the minimum frame size. The problem was that the padding region wasn't being cleared, so it contained the residue of the previous packet sent out by that Kerberos service. That previous packet was probably a response to a correctly formed request, which typically includes both the Kerberos realm name and the plaintext principal identifier of some authorized user. Although exposing the principal identifier of an authorized user to an adversary is not directly a security breach, the first step in mounting a dictionary attack (to which Kerberos is susceptible) is to obtain a principal identifier of an active user and the exact syntax of the realm name used by this Kerberos service[*]

Lesson: As in example 11.11.1.1, above, use *fail-safe defaults.* The packet buffer should have been cleared between uses.

11.11.1.3 2000: Residues in HTTP

To avoid retransmitting an entire file following a transmission failure, the HyperText Transfer Protocol (HTTP), the primary transport mechanism of the World Wide Web, allows a client to ask a service for just a portion of a file, describing that part by a starting address and a data length. If the requested region lies beyond the end of the file, the protocol specifies that the service return just the data up to the end of the file and alert the client about the error.

The Apple Macintosh AppleShare Internet Web service was discovered to return exactly as much data as the client requested. When the client asked for more data than was actually in the file, the service returned as much of the file as actually existed, followed by whatever data happened to be in the service's primary memory following the file. This implementation error allowed any client to mine data from the service.[†]

Lesson: Apparently unimportant specifications, such as "return only as much data as is actually in the file" can sometimes be quite important.

---

[*] Reported on CTSS by Maxim G. Smith in 1963. The identical problem was found in the General Electric GCOS system when its security was being reviewed by the U.S. Defense Department in the 1970's, as reported by Roger R. Schell. Computer Security: the Achilles' heel of the electronic Air Force? *Air University Review XXX,* 2 (January-February 1979) page 21.

[*] Reported by L0pht Heavy Industries in 1997, after the system had been in production use for ten years.

[†] Reported Monday 17April 2000 to an (unidentified) Apple Computer technical support mailing list by Clint Ragsdale, followed up by analysis by Andy Griffin in *Macintouch* (Tuesday 18 April 2000) `<http://www.macintouch.com/>`.

11.11.1.4 Residues on Removed Disks

The potential for analysis of residues turns up in a slightly different form when a technician is asked to repair or replace a storage device such as a magnetic disk. Unless the device is cleared of data first, the technician may be able to read it. Clearing a disk is generally done by overwriting it with random data, but sometimes the reason for repair is that the write operation isn't working. Worse, if the hardware failure is data-dependent, it may be essential that the technician be allowed to read the residue to reproduce and diagnose the failure.

In November 1998, the dean of the Harvard Divinity School was sacked after he asked a University technician to upgrade his personal computer to use a new, larger hard disk and transfer the contents of the old disk to the new one. When the technician's supervisor asked why the job was taking so long, the technician, after some prodding, reluctantly replied that there seemed to be a large number of image files to transfer. That reply led to further questions, upon which it was discovered that the image files were pornographic.*

Lesson: Physical possession of storage media usually allows bypass of security measures that are intended to control access within a system. The technician who removes a disk doesn't need a password to read it. Encryption of stored files can help minimize this problem.

11.11.1.5 Residues in Backup Copies

It is common practice for a data-storing system to make periodic backup copies of all files onto magnetic tape, often in several different formats. One format might allow quick reloading of all files, while another might allow efficient searching for a single file. Several backup copies, perhaps representing files at one-week intervals for a month, and at one-month intervals for a year, might be kept.

The administrator of a Cambridge University time-sharing system was served with an official government request to destroy all copies of a specific file belonging to a certain user. The user had compiled a list of secret telephone access codes, which could be used to place free long-distance calls. Removing the on-line file was straightforward, but the potential cost of locating and expunging the backup copies of that file—while maintaining backup copies of all other files—was enormous. (A compromise was reached, in which the backup tapes received special protection until they were due to be recycled.)†

A similar, more highly publicized backup residue incident occurred in November 1986 when Navy Vice-Admiral John M. Poindexter and Lieutenant Colonel Oliver North deleted 5,748 e-mail messages in connection with the Iran-Contra affair. They apparently did not realize that the PROFS e-mail system used by the National Security Council maintained backup copies. The messages found on the backup tapes became

* James Bandler. Harvard ouster linked to porn; Divinity School dean questioned. *Boston Globe* (Wednesday 19 May 1999) City Edition, page B1, Metro/Region section.

† Incident ca. 1970, reported by Roger G. Needham.

important evidence in subsequent trials of both individuals. An interesting aspect of this case was that the later investigation focused not just on the content of specific messages, but on their context in relation to other messages, which the backup system also preserved.[*] [†]

Lesson: there is a tension between reliability, which calls for maintaining multiple copies of data, and security, which is enhanced by minimizing extra copies.

11.11.1.6 Magnetic Residues: High-Tech Garbage Analysis

A more sophisticated version of the residue problem is encountered when recording on continuous media such as magnetic tape or disk. If the residue is erased by overwriting, an ordinary read to the disk will no longer return the previous data. However, analysis of the recording medium in the laboratory may disclose residual magnetic traces of previously recorded data. In addition, many disk controllers automatically redirect a write to a spare sector when the originally addressed sector fails, leaving on the original sector a residue that a laboratory can retrieve. For these reasons, certain U.S. Department of Defense agencies routinely burn magnetic tapes and destroy magnetic disk surfaces in an acid bath before discarding them. [‡]

11.11.1.7 2001 and 2002: More Low-tech Garbage Analysis

The lessons about residues apparently have not yet been completely absorbed by system designers. In July 2001, a user of the latest version of the Microsoft Visual C++ compiler who regularly clears the unused part of his hard disk by overwriting it with a characteristic data pattern discovered copies of that pattern in binary executables created by the compiler. Apparently the compiler allocated space on the disk as temporary storage but did not clear that space before using it.[**] In January 2002, people who used the Macintosh operating system to create CD's for distribution were annoyed to find that most disk-burning software, in order to provide icons for the files on the CD, simply copied the current desktop database, which contains those icons, onto the CD. But this database file contains icons for *every* application program of the user as well as incidental other information about many of the files on the user's personal hard disks—such as the World-Wide Web address from which they were downloaded. Thus users who received such CD's found that in addition to the intended files, there was a remarkable, and occasionally embarrassing, collection of personal information there, too.

[*] Lawrence E. Walsh. *Final report of the independent counsel for Iran/Contra matters Volume 1*, Chapter 3 (4 August 1993) U.S. Court of Appeals for the District of Columbia Circuit, Washington, D.C.

[†] The context issue is highlighted in *Armstrong v. Bush*, 721 F. Supp. 343, 345 n.1 (D.D.C. 1989).

[‡] *Remanence Security Guidebook.* Naval Staff Office Publication NAVSO P-5239-26 (September 1993:United States Naval Information Systems Management Center: Washington D.C.)

[**] David Winfrey. "Uncleared disk space and MSVC". *Risks Forum Digest 21*, 50 (12 July 2001).

Lesson: "Visit with your predecessors… They know the ropes and can help you see around some corners. Try to make original mistakes, rather than needlessly repeating theirs."[*]

11.11.2 Plaintext Passwords Lead to Two Breaches

Some design choices, while not directly affecting the internal security strength of a system, can affect operational aspects enough to weaken system security.

In CTSS, as already mentioned, passwords were stored in the file system together with user names. Since this file was effectively a master user list, the system administrator, whenever he changed the file, printed a copy for quick reference. His purpose was not to keep track of passwords. Rather, he needed the list of user names to avoid duplication when adding new users. This printed copy, including the passwords, was processed by printer controller software, handled by the printer operator, placed in output bins, moved to the system administrator's office, and eventually discarded by his secretary when the next version arrived. At least one penetration of CTSS was accomplished by a student who discovered an old copy of this printed report in a wastebasket (another example of a residue problem).[†]

Lesson: Pay attention to the *least privilege principle*: don't store your lunch (in this case, the names of users) in the safe with the jewels (the passwords).

At a later time, another system administrator was reviewing and updating the master user list, using the standard text editor. The editor program, to ensure atomic update of the file, operated by creating a copy of the original file under a temporary name, making all changes to that copy, and at the end renaming the copy to make it the new original. Another system operator was working at the same time as the system administrator, using the same editor to update a different file in the same directory. The different file was the "message of the day," which the system automatically displayed whenever a user logged in. The two instances of the editor used the same name for their intermediate copies, with the result that the master user list, complete with passwords, was posted as the message of the day. Analysis revealed that the designer of the editor had, as a simplification, chosen to use a fixed name for the editor's intermediate copy. That simplification seemed reasonable because the system had a restriction that prevented two different users from working in the same directory at the same time. But in an unrelated action, someone else on the system programming staff had decided that the restriction was inconvenient and unnecessary, and had removed the interlock.[‡]

[*] Donald Rumsfeld, "Rumsfeld's Rules: Advice on Government, Business, and Life", 1974. A later version appeared as an op-ed submission in *The Wall Street Journal*, 29 January 2001.

[†] Reported by Richard G. Mills, 1963.

[‡] Fernando J. Corbató. On building systems that will fail. *Communications of the ACM 34*, 9 (September, 1991) page 77. This 1966 incident led to the use of one-way transformations for stored password records in Multics, the successor system to CTSS. But see item 11.11.3, which follows.

Lesson (not restricted to security): Removing interlocks can be risky because it is hard to track down every part of the system that depended on the interlock being there.

11.11.3 The Multiply Buggy Password Transformation

Having been burned by residues and weak designs on CTSS, the architects of the Multics system specified and implemented a (supposedly) one-way cryptographic transformation on passwords before storing them, using the same one-way transformation on typed passwords before comparing them with the stored version. A penetration team mathematically examined the one-way transformation algorithm and discovered that it wasn't one-way after all: an inverse transformation existed.

Lesson: Amateurs should not dabble in crypto-mathematics.

To their surprise, when they tried the inverse transformation it did not work. After much analysis, the penetration team figured out that the system procedure implementing the supposedly one-way transformation used a mathematical library subroutine that contained an error, and the passwords were being transformed incorrectly. Since the error was consistent, it did not interfere with later password comparisons, so the system performed password authentication correctly. Further, the erroneous algorithm turned out to be reversible too, so the system penetration was successful.

An interesting sidelight arose when penetration team reported the error in the mathematical subroutine and its implementers released a corrected update. Had the updated routine simply been installed in the library, the password-transforming algorithm would have begun working correctly. But then, correct user-supplied passwords would transform to values that did not match the stored values previously created using the incorrect algorithm. Thus, no one would be able to log in. A creative solution (which the reader may attempt to reinvent) was found for the dilemma.*

11.11.4 Controlling the Configuration

Even if one has applied a consistent set of security techniques to the hardware and software of an installation, it can be hard to be sure that they are actually effective. Many aspects of security depend on the exact configuration of the hardware and software—that is, the versions being used and the controlling parameter settings. Mistakes in setting up or controlling the configuration can create an opportunity for an attacker to exploit. Before Internet-related security attacks dominated the news, security consultants usually advised their clients that their biggest security problem was likely to be unthinking or unauthorized action by an authorized person. In many systems the number of people authorized to tinker with the configuration is alarmingly large.

* Peter J. Downey. *Multics Security Evaluation: Password and File Encryption Techniques.* United States Air Force Electronics Systems Division Technical Report ESD–TR–74–193, Vol. III (June 1977).

11.11.4.1 Authorized People Sometimes do Unauthorized Things

A programmer was temporarily given the privilege of modifying the kernel of a university operating system as the most expeditious way of solving a problem. Although he properly made the changes appropriate to solve the problem, he also added a feature to a rarely-used metering entry of the kernel. If called with a certain argument value, the metering entry would reset the status of the current user's account to show no usage. This new "feature" was used by the programmer and his friends for months afterwards to obtain unlimited quantities of service time.*

11.11.4.2 The System Release Trick

A Department of Defense operating system was claimed to be secured well enough that it could safely handle military classified information. A (fortunately) friendly penetration team looked over the system and its environment and came up with a straightforward attack. They constructed, on another similar computer, a modified version of the operating system that omitted certain key security checks. They then mailed to the DoD installation a copy of a tape containing this modified system, together with a copy of the most recent system update letter from the operating system vendor. The staff at the site received the letter and tape, and duly installed its contents as the standard operating system. A few days later one of the team members invited the management of the installation to watch as he took over the operating system without the benefit of either a user id or a password.†

Lesson: *Complete mediation* includes checking the authenticity, integrity, and permission to install of software releases, whether they arrive in the mail or are downloaded over the Internet.

11.11.4.3 The Slammer Worm‡

A malware program that copies itself from one computer to another over a network is known as a "worm". In January 2003 an unusually virulent worm named Slammer struck, demonstrating the remarkable ease with which an attacker might paralyze the otherwise robust Internet. Slammer did not quite succeed because it happened to pick on an occasionally used interface that is not essential to the core operation of the Internet. If Slammer had found a target in a really popular interface, the Internet would have

* Reported by Richard G. Mills, 1965.

† This story has been in the folklore of security for at least 25 years, but it may be apocryphal. A similar tale is told of mailing a a bogus field change order, which would typically apply to the hardware, rather than the software, of a system. The folklore is probably based on a 1974 analysis of operating practices of United States Defense contractors and Defense Department sites that outlined this attack possibility in detail and suggested strongly that mailing a bogus software update would almost certainly result in its being installed at the target site. The authors never actually tried the attack. Paul A. Karger and Roger R. Schell. *MULTICS Security Evaluation: Vulnerability Analysis*. United States Air Force Electronics Systems Division Technical Report ESD–TR–74–193 Vol. II (June 1974), Section 3.4.5.1.

locked up before anyone could do anything about it, and getting things back to even a semblance of normal operation would probably have taken a long time.

The basic principle of operation of Slammer was stunningly simple:

1. Discover an Internet port that is enabled in many network-attached computers, and for which a popular listener implementation has a buffer overrun bug that a single, short packet can trigger. Internet Protocol UDP ports are thus a target of choice. Slammer exploited a bug in Microsoft SQL Server 2000 and Microsoft Server Desktop Engine 2000, both of which enable the SQL UDP port. This port is used for database queries, and it is vulnerable only on computers that run one of these database packages, so it is by no means universal.
2. Send to that port a packet that overruns a buffer, captures the execution point of the processor, and runs a program contained in the packet.
3. Write that program to go into a tight loop, generating an Internet address at random and sending a copy of the same packet to that address, as fast as possible. The smaller the packet, the more packets per second the program can launch. Slammer used packets that were, with headers, 404 bytes long, so a broadband-connected (1 megabit/second) machine could launch packets at a rate of 300/second, a machine with a 10 megabits/second path to the Internet could launch packets at a rate of 3,000/second and a high-powered server with a 155 megabits/second connection might be able to launch as many as 45,000 packets/second.

Forensics: Receipt of this single Slammer worm packet is enough to instantly recruit the target to help propagate the attack to other vulnerable systems. An interesting forensic problem is that recruitment modifies no files and leaves few traces because the worm exists only in volatile memory. If a suspicious analyst stops a recruited machine, disconnects it from the Internet, and reboots it, the analyst will find nothing. There may be some counters indicating that there was a lot of outbound network traffic, but no clue why. So one remarkable feature of this kind of worm is the potential difficulty of tracing its source. The only forensic information available is likely to be the payload of the intentionally tiny worm packet.

Exponential attack rate: A second interesting observation about the Slammer worm is how rapidly it increased its aggregate rate of attack. It recruited every vulnerable computer on the Internet as both a prolific propagator and also as an intense source of Internet traffic. The original launcher needed merely to find one vulnerable machine anywhere in the Internet and send it a single worm packet. This newly-recruited target immediately began sending copies of the worm packet to other addresses chosen at random. Internet version 4, with its 32-bit address fields, provided about 4 billion addresses,

‡ This account is based on one originally published under the title "Slammer: an urgent wake-up call", pages 243–248 in *Computer Systems: theory, technology and applications/A tribute to Roger Needham*, Andrew Herbert & Karen Spärck Jones, editors. (Springer: New York: 2004)

and even though many of them were unassigned, sooner or later one of these worm packets was likely to hit another machine with the same vulnerability. The worm packet immediately recruited this second machine to help with the attack. The expected time until a worm packet hit yet another vulnerable machine dropped in half and the volume of attack traffic doubled. Soon third and fourth machines were recruited to join the attack; thus the expected time to find new recruits halved again and the malevolent traffic rate doubled again. This epidemic process proceeded with exponential growth until either a shortage of new, vulnerable targets or bottlenecked network links slowed it down; the worm quickly recruited every vulnerable machine attached to the Internet.

The exponent of growth depends on the average time it takes to recruit the next target machine, which in turn depends on two things: the number of vulnerable targets and the rate of packet generation. From the observed rate of packet arrivals at the peak, a rough estimate is that there were 50 thousand or more recruits, launching at least 50 million packets per second into the Internet. The aggregate extra load on the Internet of these 3200-bit packets probably amounted to something over 150 Gigabits/second, but that is well below the aggregate capacity of the Internet, so reported disruptions were localized rather than universal.

With 50 thousand vulnerable ports scattered through a space of 4 billion addresses, the chance that any single packet hits a vulnerable port is one in 120 thousand. If the first recruit sends one thousand packets per second, the expected time to hit a vulnerable port would be about two minutes. In four minutes there would be four recruits. In six minutes, eight recruits. In half an hour, nearly all of the 50 thousand vulnerable machines would probably be participating.

Extrapolation: The real problem appears if we redo that analysis for a port to which five million vulnerable computers listen: the time scale drops by two orders of magnitude. With that many listeners, a second recruit would receive the worm and join the attack within one second, two more one second later, etc. In less than 30 seconds, most of the 5 million machines would be participating, each launching traffic onto the Internet at the fastest rate they (or their Internet connection) can sustain. This level of attack, about two orders of magnitude greater than the intensity of Slammer, would almost certainly paralyze every corner of the Internet. It could take quite a while to untangle because the overload of every router and link would hamper communication among people who are trying to resolve the problem. In particular, it could be difficult for owners of vulnerable machines to learn about and download any necessary patches.

Prior art: Slammer used a port that is not widely enabled, yet its recruitment rate, which determines its exponential growth rate, was at least one and perhaps two orders of magnitude faster than that reported for previous generations of fast-propagating worms. Those worms attacked much more widely-enabled ports, but they took longer to propagate because they used complex multipacket protocols that took much longer to set up. The Slammer attack demonstrates the power of brute force. By choosing a UDP port, infection can be accomplished by a single packet, so there is no need for a time-consuming protocol interchange. The smaller the packet size, the faster a recruit can then launch packets to discover other vulnerable ports.

Another risk: The worm also revealed a risk of networks that advertise a large number of addresses. At the time that individual computers that advertise a single address were receiving one Slammer worm packet every 80 seconds, a network that advertises 16 million addresses would have been receiving 200,000 packets/second, with a data rate of about 640 megabits/second. In confirmation, incoming traffic to the M.I.T. network border routers, which actually do advertise 16 million addresses, peaked at a measured rate of around 500 megabits/second with some of its links to the public Internet saturated. Being the home of 16 million Internet addresses has its hazards.

Lessons: From this incident we can draw different lessons for different network participants: For users, the perennial but often-ignored advice to disable unused network ports does more than help a single computer resist attack, it helps protect the entire network. For vendors, shipping an operating system that by default activates a listener for a feature that the user does not explicitly request is hazardous to the health of the network (*use fail-safe defaults*). For implementers, it emphasizes the importance of diligent care (and paranoid design) in network listener implementations, especially on widely activated UDP ports.*

11.11.5 The Kernel Trusts the User

11.11.5.1 Obvious Trust

In the first version of CTSS, a shortcut was taken in the design of the kernel entry that permitted a user to read a large directory as a series of small reads. Rather than remembering the current read cursor in a system-protected region, as part of each read call the kernel returned the cursor value to the caller. The caller was to provide that cursor as an argument when calling for the next record. A curious user printed out the cursor, concluded that it looked like a disk sector address, and wrote a program that specified sector zero, a starting block that contained the sector address of key system files. From there he was able to find his way to the master user table containing (as already mentioned, plaintext) passwords.†

Although this vulnerability seems obvious, many operating systems have been discovered to leave some critical piece of data in an unprotected user area, and later rely on its integrity. In OS/360, the operating system for the IBM System/360, each system module was allocated a limited quota of system-protected storage, as a strategy to keep the system small. Since the quota was unrealistically small in many cases, system programmers were effectively forced to place system data in unprotected user areas. Despite many later efforts to repair the situation, an acceptable level of security was never achieved in that system.‡

Lesson: A bit more attention to paranoid design would have avoided these problems.

* A detailed analysis of the Slammer worm and its effects on the Internet can be found in David Moore, *et al.*, "Inside the Slammer Worm", *IEEE Security and Privacy 1*, 4 (July 2003) pages 33 - 39.

† Noticed by the author, exploit developed by Maxim G. Smith, 1963.

11.11.5.2 Nonobvious Trust (Tocttou)

As a subtle variation of the previous problem, consider the following user-callable kernel entry point:

```
1   procedure DELETE_FILE (file_name)
2       auth ← CHECK_DELETE_PERMISSION (file_name, this_user_id)
3       if auth = PERMITTED
4           then DESTROY (file_name)
5           else signal ("You do not have permission to delete file_name")
```

This program seems to be correctly checking to verify that the current user (whose identity is found in the global variable *this_user_id*) has permission to delete file *file_name*. But, because the code depends on the meaning of *file_name* not changing between the call to CHECK_DELETE_PERMISSION on line 2 and the call to DESTROY on line 4, in some systems there is a way to defeat the check.

Suppose that the system design uses indirection to decouple the name of a file from its permissions (as for example, in the UNIX file system, which stores its permissions in the inode, as described in Section 2.5.7). With such a design, the user can, in a concurrent thread, unlink and then relink the name *file_name* to a different file, thereby causing deletion of some other file that CHECK_DELETE_PERMISSION would not have permitted. There is, of course a race—the user's concurrent thread must perform the unlinking and relinking in the brief interval between when CHECK_DELETE_PERMISSION looks up *filename* in the file system and DESTROY looks up that same name again. Nevertheless, a window of opportunity does exist, and a clever adversary may also be able to find a way to stretch out the window.

This class of error is so common in kernel implementations that it has a name: "Time Of Check To Time Of Use" error, written "tocttou" and pronounced "tock-two".*

Lesson: For <u>*complete mediation*</u> to be effective, one must also consider the dynamics of the system. If the user can change something after the guard checks for authenticity, integrity, and permission, all bets are off.

11.11.5.3 Tocttou 2: Virtualizing the DMA Channel.

A common architecture for Direct Memory Access (DMA) input/output channel processors is the following: DMA channel programs refer to absolute memory addresses without any hardware protection. In addition, these channel programs may be able to modify themselves by reading data in over themselves. If the operating system permits the user to create and run DMA channel programs, it becomes difficult to enforce security constraints, and even more difficult for an operating system to create virtual DMA

‡ Allocation strategy reported by Fred Brooks in *The Mythical Man-Month.*[Suggestions for Further Reading 1.1.3

* Richard Bisbey II, Gerald Popek, and Jim Carlstedt. *Protection errors in operating systems: inconsistency of a single data value over time.* USC/Information Sciences Institute Technical Report SR–75–4 (January 1976).

channels as part of a virtual machine implementation. Even if the channel programs are reviewed by the operating system to make sure that all memory addresses refer to areas assigned to the user who supplied the channel program, if the channel program is self-modifying, the checks of its original content are meaningless. Some system designers try to deal with this problem by enforcing a prohibition on timing-dependent and self-modifying DMA channel programs. The problem with this approach was that it is difficult to methodically establish by inspection that a program conforms with the prohibition. The result is a battle of wits: for every ingenious technique developed to discover that a DMA channel program contains an obscure self-modification feature, some clever adversary may discover a still more obscure way to conceal self-modification. Precisely such a problem was noted with virtualization of I/O channels in the IBM System/360 architecture and its successors.*

Lesson: It can be a major challenge to apply *complete mediation* to a legacy hardware architecture.

11.11.6 Technology Defeats Economic Barriers

11.11.6.1 An Attack on Our System Would be Too Expensive

A Western Union vice-president, when asked if the company was using encryption to protect the privacy of messages sent via geostationary satellites, dismissed the question by saying, "Our satellite ground stations cost millions of dollars apiece. Eavesdroppers don't have that kind of money."† This response seems oblivious of two things: (1) an eavesdropper may be able to accomplish the job with relatively inexpensive equipment that does not have to meet commercial standards of availability, reliability, durability, maintainability, compatibility, and noise immunity, and (2) improvements in technology can rapidly reduce an eavesdropper's cost. The next anecdote provides an example of the second concern.

Lesson: Never underestimate the effect of technology improvement, and the effectiveness of the resources that a clever adversary may bring to bear.

11.11.6.2 Well, it Used to be Too Expensive

In 2003, the University of Texas and Georgia Tech were victims of an attack made possible by advancing computer and network technology. The setup went as follows: The database of student, staff, and alumni records included in each record a field containing that person's Social Security number. Furthermore, the Social Security number field was

* This battle of wits is well known to people who have found themselves trying to "virtualize" existing computer architectures, but apparently the only specific example that has been documented is in C[lement]. R[ichard]. Attanasio, P[eter] W. Markstein and R[ay]. J. Philips, "Penetrating an operating system: a study of VM/370 integrity," *IBM System Journal 15*, 1 (1976), pages 102–117.

† Reported by F. J. Corbató, ca. 1975.

a key field, which means that it could be used to retrieve records. The assumption was that this feature was useful only to a client who knew a Social Security number.

The attackers realized that the universities had a high-performance database service attached to a high-bandwidth network, and it was therefore possible to systematically try all of the 999 million possible Social Security numbers in a reasonably short time—in other words, a dictionary attack. Most trials resulted in a "no such record" response, but each time an offered Social Security number happened to match a record in the database, the service returned the entire record for that person, thereby allowing the Social Security number to be matched with a name, address, and other personal information.

The attacks were detected only when it was noticed that the service seemed to be experiencing an unusually heavy load.*

Lesson: As technology improves, so do the tools available for adversaries.

11.11.7 Mere Mortals Must be Able to Figure Out How to Use it

In an experiment at Carnegie-Mellon University, Alma Whitten and Doug Tygar engaged twelve subjects who were experienced users of e-mail, but who had not previously tried to send secure e-mail. The task for these subjects was to figure out how to send a signed and encrypted message, and decrypt and authenticate the response, within 90 minutes. They were to use the cryptographic package Pretty Good Privacy (PGP) together with the Eudora e-mail system, both of which were already installed and configured to work together.

Of the twelve participants, four succeeded in sending the message correctly secured; three others sent the message in plaintext thinking that it was secure, and the remaining five never figured out how to complete the task. The report on this project provides a step-by-step analysis of the mistakes and misconceptions encountered by each of the twelve test subjects. It also includes a cognitive walkthrough analysis (that is, an *a priori* review) of the user interface of PGP.†

Lessons:

1. The mental model that a person needs to make correct use of public-key cryptography is hard for a non-expert to grasp; a simpler description is needed.
2. Any undetected mistake can compromise even the best security. Yet it is well known that it requires much subtlety to design a user interface that minimizes mistakes. The *principle of least astonishment* applies.

* Robert Lemos. "Data thieves nab 55,000 student records" *CNET News.com,* March 6, 2003. Robert Lemos. "Data thieves strike Georgia Tech" *CNET News.com,* March 31, 2003.

† Alma Whitten and J. D. Tygar. *Usability of Security: A Case Study.* Carnegie-Mellon University School of Computer Science Technical Report CMU–CS–98–155, December 1998. A less detailed version appeared in Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0. *Proceedings of the eighth USENIX security symposium*, August 1999.

11.11.8 The Web can be a Dangerous Place

In the race to create the World Wide Web browser with the most useful features, security sometimes gets overlooked. One potentially useful feature is to launch the appropriate application program (called a helper) after downloading a file that is in a format not handled directly by the browser. However, launching an application program to act on a file whose contents are specified by someone else can be dangerous.

Cognizant of this problem, the Microsoft browser, named Internet Explorer, maintained a list of file types, the corresponding applications, and a flag for each that indicates whether or not launching should be automatic or the user should be asked first. When initially installed, Internet Explorer came with a pre-configured list, containing popular file types and popular application programs. Some flags were preset to allow automatic launch, indicating that the designer believed certain applications could not possibly cause any harm.

Apparently, it is harder than it looks to make such decisions. So far, three different file types whose default flags allow automatic launch have been identified as exploitable security holes on at least some client systems:

- Files of type ".LNK", which in Windows terminology are called "shortcuts" and are known elsewhere as symbolic links. Downloading one of these files causes the browser to install a symbolic link in the client's file system. If the internals of the link indicate a program at the other end of the link, the browser then attempts to launch that program, giving it arguments found in the link.
- Files of type ".URL", known as "Internet shortcuts", which contain a URL. The browser simply loads this URL, which would seem to be a relatively harmless thing to do. But a URL can be a pointer to a local file, in which case the browser does not apply security restrictions (for example, in running scripts in that file) that it would normally apply to files that came from elsewhere.
- Files of type ".ISP", which are intended to contain scripts used to set up an account with an Information Service Provider. Since the script interpreter was an undocumented Microsoft-provided application, deciding that a script cannot cause any harm was not particularly easy. Searching the binary representation of the program for character strings revealed a list of script keywords, one of which was "RUN". A little experimenting revealed that the application that interprets this keyword invokes the operating system to run whatever command line follows the RUN key word.

The first two of these file types are relatively hard to exploit because they operate by running a program already stored somewhere on the client's computer. A prospective attacker would have to either guess the location of an existing, exploitable application program or surreptitiously install a file in a known location. Both of these courses are, however, easier than they sound. Most system installations follow a standard pattern, which means that vendor-supplied command programs are stored in standard places with standard names, and many of those command programs can be exploited by passing them appropriate arguments. By judicious use of comments and other syntactic tricks one can create a file that can be interpreted either as a harmless HTML Web page or as a command script. If the client reads such an HTML Web page, the browser places a copy in its Web cache, where it can then be exploited as a command script, using either the .LNK or .URL type.

Lesson: The fact that these security problems were not discovered before product release suggests that competitive pressures can easily dominate concern for security. One would expect that even a somewhat superficial security inspection would have quickly revealed each of these problems. Failure to adhere to the principle of *open design* is also probably implicated in this incident. Finally, the *principle of least privilege* suggests that automatically launched programs that could be under control of an adversary should be run in a distinct virtual machine, the computer equivalent of a padded cell, where they can't do much damage.*

11.11.9 The Reused Password

A large corporation arranged to obtain network-accessible computing services from two competing outside suppliers. Employees of the corporation had individual accounts with each supplier.

Supplier A was quite careful about security. Among other things, it did not permit users to choose their own passwords. Instead, it assigned a randomly-chosen password to each new user. Supplier B was much more relaxed—users could choose their own passwords for that system. The corporation that had contracted for the two services recognized the difference in security standards and instructed its employees not to store any company confidential or proprietary information on supplier B's more loosely managed system.

In keeping with their more relaxed approach to security, a system programmer for supplier B had the privilege of reading the file of passwords of users of that system. Knowing that this customer's staff also used services of supplier A, he guessed that some of them were probably lazy and had chosen as their password on system B the same password that they had been assigned by supplier A. He proceeded to log in to system A successfully, where he found a proprietary program of some interest and copied it back to his own system. He was discovered when he tried to sell a modified version of the program, and employees of the large corporation became suspicious.†

Lesson: People aren't good at keeping secrets.

* Chris Rioux provided details on this collection of browser problems, and discovered the .ISP exploitation, in 1998.

† This anecdote was reported in the 1970's, but its source has been lost.

11.11.10 Signaling with Clandestine Channels

11.11.10.1 Intentionally I: Banging on the Walls

Once information has been released to a program, it is difficult to be sure that the program does not pass the information along to someone else. Even though non-discretionary controls may be in place, a program written by an adversary may still be able to signal to a conspirator outside the controlled region by using a clandestine channel. In an experiment with a virtual memory system that provides shared library procedures, an otherwise confined program used the following signalling technique: For the first bit of the message to be transmitted, it touched (if the bit value was ONE) or failed to touch (if the bit value was ZERO) a previously agreed-upon page of a large, infrequently used, shared library program. It then waited a while, and repeated the procedure for the second bit of the message. A receiving thread observed the presence of the agreed-upon page in memory by measuring the time required to read from a location in that page. A short (microsecond) time meant that the page was already in memory and a ONE value was recorded for that bit. Using an array of pages to send multiple bits, interspersed with pauses long enough to allow the kernel to page out the entire array, a data rate of about one bit per second was attained.* This technique of transmitting data by an otherwise confined program is known as "banging on the walls".

In 2005, Colin Percival noticed that when two processors share a cache, as do certain chips that contain multiple processors, this same technique can be used to transmit information at much higher rate. Percival estimates that the L1 cache of a 2.8 gigahertz Pentium 4 could be used to transmit data upwards of 400 kilobytes per second†.

Lesson: *Minimize common mechanisms*. A common mechanism such as a shared virtual memory or a shared cache can provide an unintended communication path.

11.11.10.2 Intentionally II

In an interesting 1998 paper,‡ Marcus Kuhn and Ross Anderson describe how easy it is to write programs that surreptitiously transmit data to a nearby, cheap, radio receiver by careful choice of the patterns of pixels appearing on the computer's display screen. A display screen radiates energy in the form of radio waves whose shape depends on the particular pattern on the screen. They also discuss how to design fonts to minimize the ability for an adversary to interpret this unwanted radiation.

Lesson: Paranoid design requires considering *all* access paths.

---

* Demonstrated by Robert E. Mullen ca. 1976, described by Tom Van Vleck in a poster session at the *IEEE Symposium on Research in Security and Privacy*, Oakland, California, May 1990. The description is posted on the Multics Web site, at <www.multicians.org/thvv/timing-chn.html>.

† C. Percival, Cache missing for fun and profit. *Proceedings of BSDCAN 2005*, Ottawa. http://www.deamonology.net/papers/htt.pdf (May 2005).

‡ Markus G. Kuhn and Ross J. Anderson. Soft Tempest: Hidden Data Transmission Using Electromagnetic Emanations. In David Aucsmith (Ed.): *Information Hiding 1998, Lecture Notes in Computer Science 1525*, pages 124–142 (1998: Springer-Verlag: Berlin and Heidelberg).

11.11.10.3 Unintentionally

If an operating system is trying to avoid releasing a piece of information, it may still be possible to infer its value from externally observed behavior, such as the time it takes for the kernel to execute a system call or the pattern of pages in virtual memory after the kernel returns. An example of this attack was discovered in the Tenex time-sharing system, which provided virtual memory. Tenex allowed a program to acquire the privileges of another user if the program could supply that user's secret password. The kernel routine that examined the user-supplied password did so by comparing it, one character at a time, with the corresponding entry in the password table. As soon as a mismatch was detected, the password-checking routine terminated and returned, reporting a mismatch error.

This immediate termination turned out to be easily detectable by using two features of Tenex. The first feature was that the system reacted to an attempt to touch a nonexistent page by helpfully creating an empty page. The second feature was that the user can ask the kernel if a given page exists. In addition, the user-supplied password can be placed anywhere in user memory.

An attacker can place the first character of a password guess in the last byte of the last existing page, and then call the kernel asking for another user's privileges. When the kernel reports a password mismatch error, the attacker then can check to see whether or not the next page now exists. If so, the attacker concludes that the kernel touched the next page to look for the next byte of the password, which in turn implies that the first character of the password was guessed correctly. By cycling through the letters of the alphabet, watching for one that causes the system to create the next page, the attacker could systematically search for the first character of the password. Then, the attacker could move the password down in memory one character position and start a similar search for the second character. Continuing in this fashion, the entire password could be quickly exposed with an effort proportional to the length of the password rather than to the number of possible passwords.*

Lesson: We have here another example of a common mechanism, the virtual memory shared between the user and the password checker inside the supervisor. Common mechanisms can provide unintended communication paths.

11.11.11 It Seems to be Working Just Fine

A hazard with systems that are supposed to provide security is that there often is no obvious indication that they aren't actually doing their job. This hazard is especially acute in cryptographic systems.

* This attack (apparently never actually exploited in the field before it was blocked) has been confirmed by Ray Tomlinson and Dan Murphy, the designers of Tenex. A slightly different description of the attack appears in Butler Lampson, "Hints for computer system design," *Operating Systems Review 17*, 5 (October 1983) pages 35–36.

11.11.11.1 I Thought it was Secure

The Data Encryption Standard (DES) is a block cryptographic system that transforms each 64-bit plaintext input block into a 64-bit output ciphertext block under what appears to be a 64-bit key. Actually, the eighth bit of each key byte is a parity check on the other seven bits, so there are only 56 distinct key bits.

One of the many software implementations of DES works as follows. One first loads a key, say *my_key*, by invoking the entry

```
status ← LOAD_KEY (my_key)
```

The LOAD_KEY procedure first resets all the temporary variables of the cryptographic software, to prevent any interaction between successive uses. Then, it checks its argument value to verify that the parity bits of the key to be loaded are correct. If the parity does not check, LOAD_KEY returns a non-zero status. If the *status* argument indicates that the key loaded properly, the application program can go on to perform other operations. For example, a cryptographic transformation can be performed by invoking

```
ciphertext ← ENCRYPT (plaintext)
```

for each 64-bit block to be transformed. To apply the inverse transformation, the application invokes LOAD_KEY with the same key value that was used for encryption and then executes

```
plaintext ← DECRYPT (ciphertext)
```

A network application used this DES implementation to encrypt messages. The client and the service agreed in advance on a key (the "permanent key"). To avoid exposing the permanent key by overuse, the first step in each session of the client/service protocol was for the client to randomly choose a temporary key to be used in this session, encipher it with the permanent key, and send the result to the service. The service decrypted the first block using the permanent key to obtain the temporary session key, and then both ends used the session key to encrypt and decrypt the streams of data exchanged for rest of that session.

The same programmer implemented the key exchange and loading program for both the client and the service. Not realizing that the DES key was structured as 56 bits of key with 8 parity bits, he wrote the program to simply use a random number generator to produce a 64-bit session key. In addition, not understanding the full implications of the status code returned by LOAD_KEY, he wrote the call to that program as follows (in the C language):

```
LOAD_KEY (tempkey)
```

thereby ignoring the returned status value.

Everything seemed to work properly. The client generated a random session key, enciphered it, and sent it to the service. The service deciphered it, and then both the client and the service loaded the session key. But in 255 times out of 256, the parity bits of the session key did not check, and the cryptographic software did not load the key. With this particular implementation, failing to load a key after state initialization caused the program to perform the identity transformation. Consequently, in most sessions all the data of the session was actually transmitted across the network in the clear.*

Lesson: The programmer who ignored the returned status value was not sufficiently paranoid in the implementation. Also, the designer of LOAD_KEY, in implementing an encryption engine that performs the identity transformation when it is in the reset state did not apply the principle of *fail-safe defaults*,. That designer also did not apply the principle to *be explicit*; the documentation of the package could have included a warning printed in large type of the importance of checking the returned status values.

11.11.11.2 How Large is the Key Space...Really?

When a client presents a Kerberos ticket to a service (see Sidebar 11.6 for a brief description of the Kerberos authentication system), the service obtains a relatively reliable certification that the client is who it claims to be. Kerberos includes in the ticket a newly-minted session key known only to it, the service, and the client. This new key is for use in continued interactions between this service and client, for example to encrypt the communication channel or to authenticate later messages.

Generating an unpredictable session key involves choosing a number at random from the 56-bit Data Encryption Standard key space. Since computers aren't good at doing things at random, generating a genuinely unpredictable key is quite difficult. This problem has been the downfall of many cryptographic systems. Recognizing the difficulty, the designers of Kerberos in 1986 chose to defer the design of a high-quality key generator until after they had worked out the design of the rest of the authentication system. As a placeholder, they implemented a temporary key generator which simply used the time of day as the initial seed for a pseudorandom-number generator. Since the time of day was measured in units of microseconds, using it as a starting point introduced enough unpredictability in the resulting key for testing.

When the public release of Kerberos was scheduled three years later, the project to design a good key generator bubbled to the top of the project list. A fairly good, hard-to-predict key generator was designed, implemented, and installed in the library. But, because Kerberos was already in trial use and the new key generator was not yet field-tested, modification of Kerberos to use the new key generator was deferred until experience with it and confidence in it could be accumulated.

In February of 1996, some 7 years later, two graduate students at Purdue University learned of a security problem attributed to a predictable key generator in a different network authentication system. They decided to see if they could attack the key generator in Kerberos. When they examined the code they discovered that the temporary, time-of-day key generator had never been replaced, and that it was possible to exhaustively search its rather limited key space with a contemporary computer in just a few seconds. Upon hearing this report, the maintainers of Kerberos were able to resecure Kerberos quickly

* Reported by Theodore T'so in 1997.

because the more sophisticated key-generator program was already in its library and only the key distribution center had to be modified to use the library program.

Lesson: This incident illustrates how difficult it is to verify proper operation of a function with negative specifications. From all appearances, the system with the predictable key generator was operating properly.[*]

11.11.11.3 How Long are the Keys?

A World Wide Web service can be configured, using the Secure Socket Layer, to apply either weak (40-bit key) or strong (128-bit key) cryptographic transformations in authenticating and encrypting communication with its clients. The Wells Fargo Bank sent the following letter to on-line customers in October, 1999:

"We have, from our initial introduction of Internet access to retirement account information nearly two years ago, recognized the value of requiring users to utilize browsers that support the strong, 128-bit encryption available in the United States and Canada. Following recent testing of an upgrade to our Internet service, we discovered that the site had been put into general use allowing access with standard 40-bit encryption. We fixed the problem as soon as it was discovered, and now, access is again only available using 128-bit encryption…We have carefully checked our Internet service and computer files and determined that at no time was the site accessed without proper authorization…"[†]

Some Web browsers display an indication, such as a padlock icon, that encryption is in use, but they give no clue about the size of the keys actually being used. As a result, a mistake such as this one will likely go unnoticed.

Lesson: The same as for the preceding anecdote 11.11.11.2.

11.11.12 Injection For Fun and Profit

A common way of attacking a system that is not well defended is to place control information in a typed input field, a method known as "injection". The programmer of the system provides an empty space, for example on a Web form, in which the user is supposed to type something such as a user name or an e-mail address. The adversary types in that space a string of characters that, in addition to providing the requested information, invokes some control feature. The typical mistake is that the program that reads the input field simply passes the typed string along to some potentially powerful interpreter without first checking the string to make sure that it doesn't contain escape characters, control characters, or even entire program fragments. The interpreter may be anything from a human operator to a database management system, and the result can be that the adversary gains unauthorized control of some aspect of the system.

* Jared Sandberg, with contribution by Don Clark. Major flaw in Internet security system is discovered by two Purdue students. *Wall Street Journal CCXXVII,* 35 (Tuesday 20 February 1996), Eastern Edition page B–7A.

† Jeremy Epstein. *Risks-Forum Digest 20,* 64 (Thursday 4 November 1999).

The countermeasure for injection is known as "sanitizing the input". In principle, santizing is simple: scan all input strings and delete inappropriate syntactical structures before passing them along. In practice, it it is sometimes quite challenging to distinguish acceptable strings from dangerous ones.

11.11.12.1 Injecting a Bogus Alert Message to the Operator

Some early time-sharing systems had a feature that allowed a logged-in user to send a message to the system operator, for example, to ask for a tape to be mounted. This message is displayed at the operator's terminal, intermixed with other messages from the operating system. The operating system normally displays a warning banner ahead of each user message so that the operator knows its source. In the Compatible Time Sharing System at M.I.T., the operating system placed no constraint on either the length or content of messages from users. A user could therefore send a single message that, first, cleared the display screen to eliminate the warning banner, and then displayed what looked like a standard system alert message, such as a warning that the system was overheating, which would lead the operator to immediately shut down the system.[*]

11.11.12.2 CardSystems Exposes 40,000,000 Credit Card Records to SQL Injection

A currently popular injection attack is known as "SQL injection". Structured Query Language (SQL) is a widely-implemented language for making queries of a database system. A typical use is that a Web form asks for a user name, and the program that receives the form inserts the typed string in place of *typedname* in an SQL statement such as this one:

select * **from** USERS **where** NAME = '*typedname*';

This SQL statement finds the record in the USERS table that has a NAME field equal to the value of the string that replaced *typedname*. Thus, if the user types "John Doe" in the space on the Web form, the SQL statement will look for and return the record for user John Doe.
Now, suppose that an adversary types the following string in the blank provided for the name field:

John Doe' ; **drop** USERS;

When that string replaces *typedname*, the result is to pass this input to the SQL interpreter:

select * **from** USERS **where** NAME = 'John Doe' ; **drop** USERS;';

The SQL interpreter considers that input to be three statements, separated by semicolons. The first statement returns the record corresponding to the name "John Doe". The second statement deletes the USERS table. The third statement consists of a single quote,

* This vulnerability was noticed, and corrected, by staff programmers in the late 1960's. As far as is known, it was never actually exploited.

which the interpreter probably treats as a syntax error, but the damage intended by the adversary has been done. The same scheme can be used to inject much more elaborate SQL code, as in the following incident, described by excerpts from published accounts.

Excerpt from `wired.com`, June 22, 2005: "MasterCard International announced last Friday that intruders had accessed the data from CardSystems Solutions, a payment processing company based in Arizona, after placing a malicious script on the company's network."* The New York Times reported that "…more than 40 million credit card accounts were exposed; data from about 200,000 accounts from MasterCard, Visa and other card issuers are known to have been stolen…"†

Excerpt from the testimony of the Chief Executive Officer of CardSystems Solutions before a Congressional committee: "An unauthorized script extracted data from 239,000 unique account numbers and exported it by FTP…"‡

Excerpt from the FTC complaint, filed a year later: "6. Respondent has engaged in a number of practices that, taken together, failed to provide reasonable and appropriate security for personal information stored on its computer network. Among other things, respondent: (1) created unnecessary risks to the information by storing it in a vulnerable format for up to 30 days; (2) did not adequately assess the vulnerability of its Web application and computer network to commonly known or reasonably foreseeable attacks, including but not limited to "Structured Query Language" (or "SQL") injection attacks; (3) did not implement simple, low-cost, and readily available defenses to such attacks; (4) failed to use strong passwords to prevent a hacker from gaining control over computers on its computer network and access to personal information stored on the network; (5) did not use readily available security measures to limit access between computers on its network and between such computers and the Internet; and (6) failed to employ sufficient measures to detect unauthorized access to personal information or to conduct security investigations.

"7. In September 2004, a hacker exploited the failures set forth in Paragraph 6 by using an SQL injection attack on respondent's Web application and Web site to install common hacking programs on computers on respondent's computer network. The programs were set up to collect and transmit magnetic stripe data stored on the network to computers located outside the network every four days, beginning in November 2004. As a result, the hacker obtained unauthorized access to magnetic stripe data for tens of millions of credit and debit cards.

"8. In early 2005, issuing banks began discovering several million dollars in fraudulent credit and debit card purchases that had been made with counterfeit cards. The counterfeit cards contained complete and accurate magnetic stripe data, including the security code used to verify that a card is genuine, and thus appeared genuine in the

* `http://www.wired.com/news/technology/0,67980-0.html`

† *The New York Times*, Tuesday, June 21, 2005.

‡ Statement of John M. Perry, President and CEO CardSystems Solutions, Inc., before the United States House of Representatives Subcommittee on Oversight and Investigations of the Committee on Financial Services, July 21, 2005.

authorization process. The magnetic stripe data matched the information respondent had stored on its computer network. In response, issuing banks cancelled and re-issued thousands of credit and debit cards. Consumers holding these cards were unable to use them to access their credit and bank accounts until they received replacement cards."[*]

Visa and American Express cancelled their contracts with CardSystems, and the company is no longer in business.

Lesson: Injection attacks, and the countermeasure of sanitizing the input, have been recognized and understood for at least 40 years, yet another example is reported nearly every day. The lesson following anecdote 11.11.1.7 seems to apply here, also.

11.11.13 Hazards of Rarely-Used Components

In the General Electric 645 processor, the circuitry to check read and write permission was invoked as early in the instruction cycle as possible. When the instruction turned out to be a request to execute an instruction in another location, the execution of the second instruction was carried out with timing later in the cycle. Consequently, instead of the standard circuitry to check read and write permission, a special-case version of the circuit was used. Although originally designed correctly, a later field change to the processor accidentally disabled one part of the special-case protection-checking circuitry. Since instructions to execute other instructions are rarely encountered, the accidental disablement was not discovered until a penetration team began a systematic study and found the problem. The disablement was dependent on the address of both the executed instruction and its operand, and was therefore unlikely to have ever been noticed by anyone not intentionally looking for security holes.[†]

Lesson: Most reliability design principles also apply to security: *avoid rarely-used components*.

11.11.14 A Thorough System Penetration Job

One particularly thorough system penetration operation went as follows. First, the team of attackers legitimately obtained computer time at a different site that ran the same hardware and same operating system. On that system they performed several experiments, eventually finding an obscure error in protecting a kernel routine. The error, which permitted general changing of any kernel-accessible variable, could be used to modify the current thread's principal identifier. After perfecting the technique, the team of attackers shifted their activities to the site where the operating system was being used for development of the operating system itself. They used the privilege of the new principal identifier to modify one source program of the operating system. The change was a one-byte revision—replacing a "less than" test with a "greater than" test, thereby com-

[*] United States Federal Trade Commission Complaint, Case 0523148, Docket C-4168, September 5, 2006.

[†] Karger and Schell, *op. cit.*, Section 3.2.2.

promising a critical kernel security check. Having installed this change in the program, they covered their trail by changing the directory record of date-last-modified on that file, thereby leaving behind no traces except for one changed line of code in the source files of the operating system. The next version of the system to be distributed to customers contained the attacker's revision, which could then be exploited at the real target site.*

This exploit was carried out by a tiger team that was engaged to discover security slip-ups. To avoid compromising the security of innocent customer sites, after verifying that the change did allow compromise, the tiger team further modified the change to one that was not exploitable, but was detectable by someone who knew where to look. They then waited until the next system release. As expected, the change did appear in that release.†

Lesson: *Complete mediation* includes verifying the authenticity, integrity, and authorization of the software development process, too.

11.11.15 Framing Enigma

Enigma is a family of encipherment machines designed in Poland and Germany in the 1920s and 1930s. An Enigma machine consists of a series of rotors, each with contacts on both sides, as in Figure 11.12. One can imagine a light bulb attached to each contact on one side of the rotor. If one touches a battery to a contact on the other side, one of the light bulbs will turn on, but which one depends on the internal wiring of that rotor. An Enigma rotor had 26 contacts on each side, thus providing a permutation of 26 letters, and the operator had a basket of up to eight such rotors, each wired to produce a different permutation.

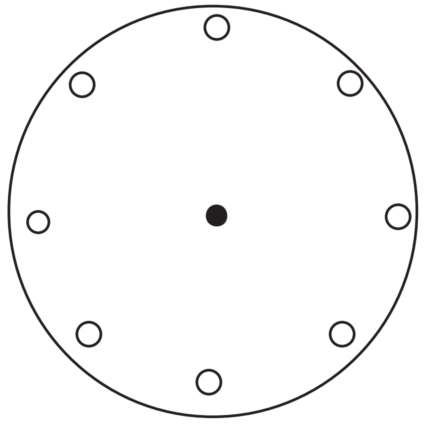
The first step in enciphering was to choose four rotors from the basket [j, k, l and m] and place them on an axle in that order. This choice was the first component of the encoding key. The next step was to set each rotor into one of 26 initial rotational positions [a, b, c, d], which constituted the second component of the encoding key. The third step was to choose one of 26 offsets [e, f, g, h] for a tab on the edge of each rotor. The offsets were the final component of the encoding key. The Enigma key space was, in terms of the computational abilities available during World War II, fairly formidable against brute force attack. After transforming one stream element of the message, the first rotor would turn clockwise one position, producing a different transformation for the next stream element. Each time the offset tab of the first rotor completed one revolution, it would strike a pawl on the second rotor, causing the second rotor to rotate clockwise by one position, and so on. The four rotors taken together act as a continually changing substitution cipher in which any letter may transform into any letter, including itself.

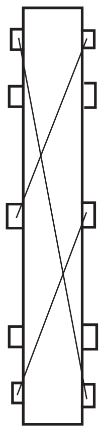
The chink in the armor came about with an apparently helpful change, in which a reflecting rotor was added at one end—in the hope of increasing the difficulty of cryptanalysis. With this change, input to and output from the substitution were both done at the same end of the rotors. This change created a restriction: since the reflector had to

* Schell, 1979 *op. cit.*, page 22.

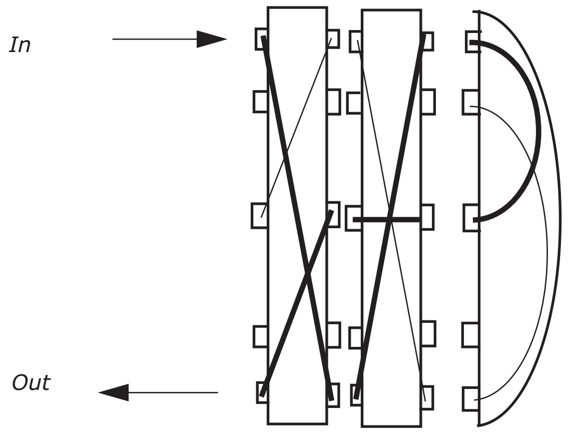
† Karger and Schell, 1974 *op. cit.,* Sections 3.4.5 and 3.4.6.

Enigma Rotor with eight contacts





Side view, showing contacts. Edge view, showing some connections.



Two Enigma Rotors with a reflector, showing an input-output path.

FIGURE 11.12

Enigma design concept (simplified for illustration).

connect some incoming character position into some *other* outgoing character position, no character could ever transform into itself. Thus the letter "E" never encodes into the letter "E".

This chink could be exploited as follows. Suppose that the cryptanalyst knew that every enciphered message began with the plaintext string of characters "The German High Command sends greetings to its field operations". Further, suppose that one has intercepted a long string of enciphered material, not knowing where messages begin and end. If one placed the known string (of length 60 characters) adjacent to a randomly selected adjacent set of 60 characters of intercepted ciphertext, there will probably be some positions where the ciphertext character is the same as the known string character. If so, the reflecting Enigma restriction guaranteed that this place could not be where that particular known plaintext was encoded. Thus, the cryptanalyst could simply slide the known plaintext along the ciphertext until he or she came to a place where no character matches and be reasonably certain that this ciphertext does correspond to the plaintext. (For a known or chosen plaintext string of 60 characters, there is a 9/10 probability that this framing is not a chance occurrence. For 120 characters, the probability rises to 99/100.)

Being able systematically to frame most messages is a significant step toward breaking a code because it greatly reduces the number of trials required to discover the key.*

Exercises

11.1 Louis Reasoner has been using a simple RPC protocol that works as follows†:

```
client ⇒ service:   {nonce, procedure, arguments}
service ⇒ client:   {nonce, response}
```

The client sets a timer, and if it does not receive a response before the timer expires, it restarts the protocol from the beginning, repeating this sequence as many times as necessary until a response returns. The service maintains a table of nonces and responses, and when it receives a request containing a duplicate nonce it repeats the response, rather than repeating execution of the procedure. The client similarly maintains a list of nonces for which no response has yet been received, and it

* A thorough explanation of the mechanism of Enigma appeared in Alan M. Turing, "A description of the machine," (Chapter 1 of an undated typescript, sometimes identified as the *Treatise on Enigma* or "the prof's book", c. 1942) [United States National Archives and Records Administration, record group 457, National Security Agency Historical Collection, box 204, Nr. 964, as reported by Frode Weierude]. A nontechnical account of the flaws in Enigma and the ways they could be exploited can be found in Stephen Budianski, *Battle of Wits* [New York: Simon & Schuster: 2000].

† Throughout the Problems and Solutions, the notation {*a*, *b*, *c*} denotes a message constructed of the named items, marshaled in some unspecified way that is unimportant for the purposes of the problem so long as the recipient knows how to unmarshal the individual arguments.

discards any responses for nonces not in that list, assuming that they are duplicates. One possible response is "unknown procedure", meaning that the service received a request it didn't know how to handle. The link layer checksums all frames and discards any that are damaged in transmission. All messages fit in one frame.

Louis wants to make this protocol secure against eavesdroppers. He has discovered that the client and the service already share a key, K_{cs}, for a shared-secret-key cryptographic system. So the first thing he tries is to encrypt the requests and responses of the simple RPC protocol:

client ⇒ service: ENCRYPT ({nonce, procedure, arguments}, K_{cs})
service ⇒ client: ENCRYPT ({nonce, response}, K_{cs})

This seems to work, but Louis has heard that if you use the same key to repeatedly transform predictable fields such as procedure names, someone may eventually discover the key by cryptanalysis. So he wants to use a different key for each RPC call. To minimize the coding effort, he changes the protocol to work as follows:

client ⇒ service: ENCRYPT ({K_{tn}}, K_{cs})
client ⇒ service: ENCRYPT ({nonce, procedure, arguments}, K_{tn})
service ⇒ client: ENCRYPT ({nonce, response}, K_{tn})

in which K_{tn} is a one-time key chosen by the client to be used only for the n'th RPC call. When the service receives a key, it decrypts it and uses it until the service gets another key message. Louis figures that since K_{cs} is now being used only to temporary keys, which look like random numbers, it should be safer from cryptanalysis.

At first, this protocol, too, seems to work. Then Louis notices that the client is receiving the response "unknown procedure" much more often than it used to. Explain why, using a timing diagram to demonstrate an example of the failure. And offer a suggestion to fix the problem.

1983-3-5b

11.2 Lucifer is determined to figure out Alice's password by a brute-force attack. From watching her log in he knows that her password is eight characters long and all lower-case letters, of which there are 26. He sets out to try all possible combinations of eight lower-case letters.

11.2a. Assuming he has to try about half the possibilities before he runs across the right one, one trial can be done in one machine cycle, and he has a 600 mHz computer available, about how long will the project take?

1994–2–1a

11.2b. How long will it take if Alice chooses an eight-character password that includes upper- and lower-case letters, numbers, and 16 special characters, 78 characters in all?

1994–2–1b

11.2c. Suppose processors continue to get faster, improving by a factor of three every two years. How long will it be until Alice's new password can be cracked as easily as her old one?

1994–2–1c

11.3 Tracy Swallow has a bright idea for avoiding the need to store passwords securely. She suggests transforming the user's name with a key-driven cryptographic transformation using a systemwide "password key" and giving the result back to the user to present as a password. A user who wishes to log in simply presents his or her name and this password; the system can authenticate the user by again transforming the user's name with the password key to see if the result is the same as the presented password. Thus no central file of passwords is needed. What is wrong with Tracy's idea?

[1983–2–4b]

11.4 Louis Reasoner is fascinated with the discovery that some cryptographic transformations are *commutative*. A commutative transformation has the interesting property that for every message and every pair of keys k_1 and k_2,

TRANSFORM (TRANSFORM (M, K_a), K_b) = TRANSFORM (TRANSFORM (M, K_b), K_a)

That is, you get the same result no matter in which order you do two transformations with different keys.

Louis did some further research, identified a high-quality commutative transformation, and used it to devise a commutative implementation of two confidentiality primitives he calls ENCRYPT_C and DECRYPT_C. He has proposed that Alice, in San Francisco, and Bob, in Boston, use the following scheme for secure private delivery of messages between their computers, which are connected via the Internet:

- Alice chooses a random key, K_a, encrypts her message M with that key, and sends the result, ENCRYPT_C (M, K_a), to Bob.

- Bob chooses another random key, K_b, encrypts the already-encrypted message to produce ENCRYPT_C (ENCRYPT_C (M, K_a), K_b) and sends the doubly-encrypted result back to Alice.
- By commutativity, this message is identical to ENCRYPT_C (ENCRYPT_C (M, K_b), K_a), which is a message that Alice can decrypt with her key K_a. She does so, revealing ENCRYPT_C (M, K_b).
- She sends this result back to Bob, who can now decrypt it with his key K_b to reveal M.

The appealing thing about this scheme is that Alice and Bob did not have to agree on a secret key in advance. Louis calls this the "No-Prior-Agreement" protocol.

11.4a. Is it possible for a passive intruder (that is, one who just listens to the encrypted messages) to discover M? If so, describe how. If not, explain why not.

1994–2–2a

11.4b. Is it possible for an active intruder (that is, one who can also insert, delete, or replay messages) to discover M? If so, describe how. If not, explain why not.

1994–2–2b

11.5 Secure Inc. is developing a remote file system, Secure RFS (SRFS), which automatically encrypts files to guarantee better privacy of information. When a request to store a file arrives, SRFS encrypts the file using the client's key. On arrival of a request to read a file, SRFS looks up the client key, decrypts the file, and sends the file back to the client. SRFS keeps for each client a separate key.

11.5a. The designers of Secure Inc. are wondering how long it would take to crack a file that is encrypted using RSA with a 512-bit key. To crack an RSA-encrypted file one has to factor the key. The designers found a 1993 paper that reports that factoring a 100 decimal digit number takes about 1 month using idle cycles from 300 3-MIPS workstations. It is estimated that factoring an additional 3 decimal digits roughly doubles the computation time needed. How many 3-MIPS computers would be needed to factor a 155 decimal digit number (which corresponds to about 512 bits) in one month?

1995–2–3a

11.5b. If processors are doubling computation performance per year, how many workstations would it take to factor a 512-bit key in one month in the year 2001?

1995–2–3b

11.5c. Assume that the cryptographic transformations can be done at 250 kilobytes per second. How much would the throughput be reduced for reading files stored by SRFS, if the current maximum throughput without cryptographic transformations is 800 kilobytes per second? (Assume that the cryptographic transformations cannot be pipelined with sending and receiving.)

1995–2–3c

11.5d. Secure Inc. is also considering adding automatic compression of files to SRFS. Compression reduces redundancy of information in a file so that the file takes less disk space. Should they first compress files, then encrypt them, or should they first encrypt files and then compress them? Explain.

1995–2–3d

11.6 Alice wants to communicate with Bob over an insecure network. She learned about one-time pads in Section 11.8, and decides to use a one-time pad to secure her communications. Since Alice wants to send a *k*-bit message to Bob in the future, she generates a random *k*-bit key *r* and hands it to Bob in person.

When Alice comes to send Bob her message, she XORS the message *m* with the key *r* to produce a ciphertext *c*, and sends this on the network. Bob XORS *c* with *r* to retrieve *m*.

11.6a. Assume that Alice's message *m* is a concatenation of a header followed by some data. Consider an eavesdropper Eve who snoops on Alice's conversation. If Eve can correctly guess the value of the header in Alice's message, which of the following are correct?

A. Eve's ability to decrypt the data bits in *m* is not improved by her knowledge of the header bits.
B. The data bits in Alice's message are confidential.
C. The data bits in Alice's message are securely authenticated.

Alice rapidly grows tired of the effort in exchanging one-time pads with Bob, and has an idea to simplify the key distribution process. Alice's idea works as follows:

To send a *k*-bit message *m1* to Bob, Alice picks a *k*-bit random number *r1*, computes ciphertext $c1 = m1 \oplus r1$, and sends *c1* to Bob. Bob then picks his own *k*-bit random number *r2*, computes $c2 = c1 \oplus r2$, and sends *c2* to Alice. Alice finally computes $c3 = c2 \oplus r1$ and sends *c3* to Bob.

11.6b. Which of the following statements are correct of Alice's new scheme?

A. Bob can correctly decrypt Alice's message *m1*, without receiving *r1* ahead of time, assuming all messages between Alice and Bob are correctly delivered.
B. An active attacker Lucifer (who can intercept, drop, and replay messages) can decrypt the message.
C. A passive eavesdropper Eve can decrypt the message.

2008-3-12-13

11.7 Bank of America is struggling to convince itself of the authenticity of a message it just received, and has asked your help in what to do next. So far, they know the following two facts to be true:

- Louis **says** (Ben **says** (Transfer $1,000,000 to Alyssa))
- Jim **speaks for** Ben

Ben's account has enough money for such a transaction, so if they can convince themselves that Ben really authorized the transaction, they will do the transfer. Which of the following things should they attempt to establish the truth of, and why?]

A. Louis **speaks for** Jim
B. Ben **speaks for** Louis
C. Ben **says** (Jim **speaks for** Louis)

1995–2–4a

11.8 Ben Bitdiddle has hit on a bright idea for fixing the problem that capabilities are hard to revoke. His plan is to invent something called *timed capabilities*. One of the fields of a timed capability is its expiration time, which is the time of creation plus E. A timed capability can be used like any other capability until the system clock reaches the expiration time; after that time, it becomes worthless. Analyze this proposal with respect to:

A. Performance.
B. Propagation.
C. Revocation.
D. Auditing.
E. Ease of use.

1984–2–4

11.9 Two banks are developing an inter-bank funds transfer system. They are connected by a telephone line which runs in a duct along Main street, and Alyssa P. Hacker is concerned that there might be foul play. The banks' expert, Ben Bitdiddle, says that the banks will use a shared-secret key K_1 to encrypt their communications and a second shared-secret key K_2 to authenticate their communications, using the following protocol:

Bank 1 ⇒ Bank 2 $\{\{\text{"transfer from our Account Y"}\}_{K2}\}^{K1}$
Bank 1 ⇒ Bank 2 $\{\{\text{"to your Account X"}\}_{K2}\}^{K1}$
Bank 1 ⇒ Bank 2 $\{\{\text{"Amount Z"}\}_{K2}\}^{K1}$
Bank 2 ⇒ Bank 1 $\{\{\text{"OK"}\}_{K2}\}^{K1}$

Alyssa immediately realizes that without knowing either $K1$ or $K2$ an intruder could subvert the banks.

11.9a. With an Apple II in the manhole in middle of Main street describe how Alyssa could

A. Increase or decrease the amount of a transfer.
B. Cause a transfer to occur more than once.
C. Cause a transfer not to occur at all without arousing suspicion at the requesting bank.

1984–2–3a

11.9b. Design a new protocol that eliminates these problems and uses only two messages.

1984–2–3b

11.10 To attract attention to their Web site, OutofMoney.com has added a feature that broadcasts a stream of messages containing free stock market quotations. They intend the information to be public, so there is no need for confidentiality, but they are concerned about their reputation, so they want the stream of data to be authenticated.

Their current implementation signs every message with the company's private key, and clients authenticate the data by verifying it with the company's widely publicized public key. This technique works, but is proving problematic because the public-key algorithm uses too much computation time and the typical client, running a four-year-old pentium processor, can't keep up with the stream of messages on days when the stock market is crashing.
From reading this chapter, they learned that authentication using a shared-secret-key MAC is much faster. They have hired Ben Bitdiddle and Louis Reasoner as a consulting team to put this idea into practice. (Unfortunately, they didn't do any of the problem sets, so they don't know about the reputations of these two characters.)

Louis's first proposal is as follows: any client who wishes to use the authenticated service starts by contacting the service and requesting a start message. The service signs this start message with the company's public key. The start message contains the shared-secret key that is currently being used to authenticate the stream of messages containing the stock market quotations.

11.10a. Ben's intuition is that this can't possibly work, but he isn't sure why. Give Ben some help by explaining why.

2002–0–1

Undaunted, Louis has been reading about *delayed authentication* and decides it is the ideal way to tackle this problem. The idea is the following: since the service is sending a stream of messages, for each message use a *different* shared-secret key to create its authentication tag, and then publicly disclose that shared-secret key *after* all clients have received that message.

In Louis's design, each message P_i is constructed as follows:

```
raw_message_i ← {i, D_i, K_{i-2}}
authtag_i ← SIGN (raw_message_i, K_i)
P_i ← {raw_message_i, authtag_i}
```

Thus P_i contains

- its own sequence number, i
- some data, D_i
- the key K_{i-2}, which can be used to verify the data in message P_{i-2}
- an authentication tag created by signing the rest of the message with K_i

The key that authenticates this message will appear in message P_{i+2}. Louis argues that even though the key K_i is sent in plaintext, if the client receives D_i before the service sends K_i, by the time the attacker knows K_i, it is too late for the attacker to modify D_i. As with Louis's previous system, a client begins by requesting a start message. This time, the start message contains the same data as the next message in the broadcast stream, but it is signed with the company's private key.

11.10b. Again, Ben is (rightly) suspicious of this system, but he can't figure out what is wrong with it. Help him out by explaining the flaw and how to fix it.

2002–0–2

11.11 This chapter discusses both capabilities and access control lists as mechanisms for authorization. Which of the following statements are true?

A. A capability system associates a list of object references with each principal, indicating which objects the principal is allowed to use.
B. An access control list system associates a list of principals with each object, indicating which principals are allowed to use the object.
C. Revocation of a particular access permission of a principal is more difficult in an access control list system than in a capability system.
D. Protection in the UNIX file system is based on capabilities only.

2002–2–04

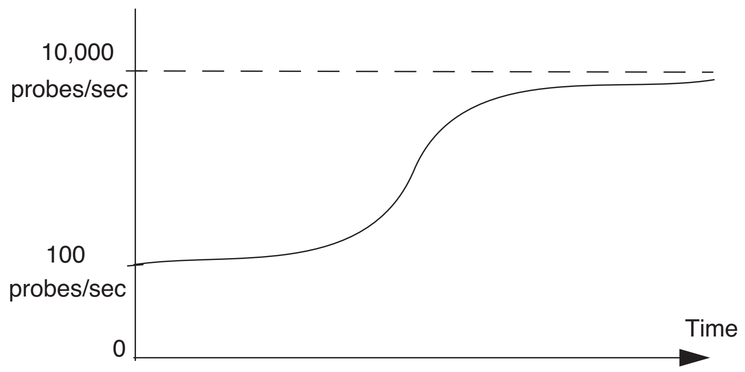
11.12 Alice decided to try out a new RFID Student Tracking System, so she created an access control list that allows a few close friends to track her. One of those friends, Bob, wants to ask Alice to join his design project team, so this morning he requested that the tracking system give him a callback if Alice walks by the Administration building. Alice, working in a nearby laboratory, belatedly realizes that Bob is probably going to pop that question, so she logs in to the tracking system and removes Bob from her access control list. She then logs out and leaves for lunch. As she walks by the Administration building, Bob comes running out of the library to greet her, saying that he just received a callback from the tracking system.

The designer of the tracking system made a security blunder. Which of the following is the most likely explanation?

A. The tracking system didn't properly erase residues.
B. In her rush to leave for lunch, Alice removed Lucy, rather than Bob, from her ACL.
C. The tracking system has a time-of-check to time-of-use bug.
D. The system used a version of SSL that is subject to cipher substitution attacks.
E. The system did not require a face-to-face rendezvous between users and system administrator.

2003–3–5

11.13 Ben decides to start an Internet Service Provider. He buys an address space that contains 2^{24} addresses (out of the total of 2^{32} in the Internet) that have never been used before. A few days after he buys this address space, someone launches a new worm similar in design to the Slammer worm described in Section 11.11.4.3. The new worm targets a buffer overflow in the FOO server, which listens on UDP port 5044. Ben monitors all traffic sent to his part of the Internet address space on port 5044 and plots the number of worm probes versus time below:



Assume the worm spreads by probing IP addresses chosen at random, and that its pseudorandom number generator is bug-free and generates a complete permutation of the integers before revisiting any integer. Ben learns from a security analyst that each infected machine sends 100 probes/second.

11.13a. Give an estimate of the total number of machines that run the FOO server.

A. 100 machines
B. 7.2×10^{18} machines
C. 25,600 machines
D. 8,000 machines

11.13b. Ben thinks that the worm used a hit list of vulnerable addresses (i.e., addresses of FOO servers). Do you agree? If you do, what is the best estimate for the number of machines contained in the hit list?

A. no hit list
B. 100 machines
C. 256 machines
D. 25600 machines
E. 80 machines

2007-3-3-4

11.14 Ben Bitdiddle, the new head of Cyber Security for the Department of Homeland Security, studied the war story about the Slammer worm in Section 11.11.4.3 and he wants to build a system that will detect and stop future worm attacks before they can reach 50% of the vulnerable hosts. Ben makes the following assumptions about the worms to be defended against:

- Each worm instance sends 512 (2^9) probes per second.
- The worm's software probes all IP addresses at random.
- Of the 2^{32} possible addresses on the Internet, there are 32,768 (2^{15}) that are attached to active hosts that are vulnerable to the worm.
- The worm begins by infecting a single vulnerable host.

11.14a. Given the assumptions above, roughly how many seconds will it take for the size of the infected population to double, during the early stages of a worm outbreak?\

A. 16 seconds
B. 256 seconds
C. 1024 seconds

Ben convinces a consortium of router vendors to develop a new, remotely-configurable packet-filtering feature, and develops a system that can propagate filter updates to all routers in the Internet within 15 minutes (900 seconds) of a detected outbreak. Once all routers have the filter, the filters will prevent all further worm infections. Ben's detection mechanism is a network monitor that can observe 1/256-th of the Internet address space. His system automatically sends a filter update whenever worm traffic directed to the set of addresses he monitors reaches a predefined threshold.

11.14b. What traffic threshold should Ben choose to stop the worm before it reaches 50% of the vulnerable hosts?

A. 10 worm probes/second
B. 100 worm probes/second
C. 1000 worm probes/second
D. 10000 worm probes/second
E. 100000 worm probes/second

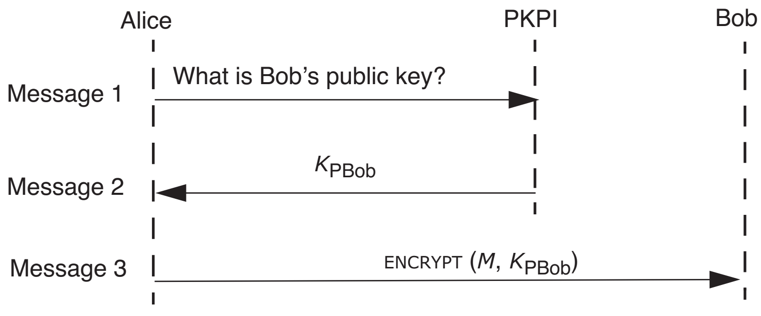
2008-3-6-7

11.15 Ben Bitdiddle visits the Web site *amazing.com* and obtains a fresh page signed with a private key. Which of these methods of obtaining the certificate for the server's public key can assure Ben that the private key used for the page's signature indeed belongs to the organization that owns the domain *amazing.com*? (Assume that the certificate is signed by a trusted certificate authority and is valid.)

A. Using HTTP Ben downloads the certificate from http://amazing6033.com.
B. Using HTTP Ben downloads the certificate from the certificate authority.
C. Ben finds the certificate by doing a Web search on Google.
D. Ben gets the certificate in e-mail from a spammer.

11.16 Ben Bitdiddle and Louis Reasoner have founded a startup company, named Public Key Publication, Inc. (PKPI), whose business is distributing public keys. Their idea is that people who have a key pair for use with a public-key system need a way of letting other people know the public key of their key pair. Ben and Louis are not interested in creating keys, but just in acting as a public key distributor.

Ben and Louis have designed the following protocol, in which Alice sends a private message to Bob. They need your help in debugging the protocol. $K_{P_{xyz}}$ is the public key of principal xyz.



Messages 1 and 2 constitute the PKPI protocol; message 3 is the beginning of Alice's protocol with Bob and is not under the control of PKPI; message 3 is shown here only to place the PKPI protocol in context.

11.16a. Louis believes that Eve, the passive eavesdropper, will find that she cannot learn anything by overhearing the PKPI protocol in use. Give an argument that supports Louis' position, or an example demonstrating that Louis is mistaken.

11.16b. Louis originally hoped that Lucifer, the active attacker, wouldn't be able to cause any problems, either, but since reading this chapter he is not sure. Give an example of an active attack that demonstrates that Louis needs to revise the PKPI protocol to protect against Lucifer.

11.16c. Ben suggests that the protocol could be improved by changing Message 2. What changes should be made so that Alice can be confident that no one but Bob can decrypt message 3?

1995–2–5a…c

11.17 Louis Reasoner's cousin Norris has discovered the following interesting fact, and would like to put it to use:

- Interesting fact: 2^{150} proton-sized objects will compactly fill the known universe.

Since nonces are used in so many different applications, Norris proposes to create the Norris Nonce Service for use by everyone. If you send a request to Norris's service it will return the next 200-bit integer, in increasing order, for use as a nonce. (Norris chose 200 in case the size of the universe turns out to have been underestimated.) What are some of the things that make this proposal harder to do than Norris probably suspects?

1983–3–3

Additional exercises relating to Chapter 11 can be found in problem sets *43–49*.

Glossary for Chapter 11

access control list (ACL)—A list of principals authorized to have access to some object. [Ch. 11]

adversary—An entity that intentionally tries to defeat the security measures of a computer system. The entity may be malicious, out for profit, or just a hacker. A friendly adversary is one that tests the security of a computer system. [Ch. 11]

authentication—Verifying the identity of a principal or the authenticity of a message. [Ch. 11]

authentication tag—A cryptographically computed string, associated with a message, that allows a receiver to verify the authenticity of the message. [Ch. 11]

authorization—A decision made by an authority to grant a principal permission to perform some operation, such as reading certain information. [Ch. 11]

capability—In a computer system, an unforgeable ticket, which when presented is taken as incontestable proof that the presenter is authorized to have access to the object named in the ticket. [Ch. 11]

certificate—A message that attests the binding of a principal identifier to a cryptographic key. [Ch. 11]

certificate authority (CA)—A principal that issues and signs certificates. [Ch. 11]

certify—To check the accuracy, correctness, and completeness of a security mechanism. [Ch. 11]

cipher—Synonym for a *cryptographic transformation*. [Ch. 11]

ciphertext—The result of encryption. Compare with *plaintext*. [Ch. 11]

cleartext—Synonym for *plaintext*. [Ch. 11]

close-to-open consistency—A consistency model for file operations. When a thread opens a file and performs several write operations, all of the modifications weill be visible to concurrent threads only after the first thread closes the file. [Ch. 4]

coheerence—See *read/write coherence* or *cache coherence*.

confidentiality—Limiting information access to authorized principals. *Secrecy* is a synonym. [Ch. 11]

confinement—Allowing a potentially untrusted program to have access to data, while ensuring that the program cannot release information. [Ch. 11]

covert channel—In a flow-control security system, a way of leaking information into or out of a secure area. For example, a program with access to a secret might touch several shared but normally unused virtual memory pages in a pattern to bring them into real memory; a conspirator outside the secure area may be able to detect the pattern by measuring the time required to read those same shared pages. [Ch. 11]

cryptographic hash function—A cryptographic function that maps messages to short values in such a way that it is difficult to (1) reconstruct a message from its hash value; and (2) construct two different messages having the same value. [Ch. 11]

cryptographic key—The easily changeable component of a key-driven cryptographic transformation. A cryptographic key is a string of bits. The bits may be generated randomly, or they may be a transformed version of a password. The cryptographic key, or at least part of it, usually must be kept secret, while all other components of the transformation can be made public. [Ch. 11]

cryptographic transformation—Mathematical transformation used as a building block for implementing security primitives. Such building blocks include functions for implementing encryption and decryption, creating and verifying authentication tags, cryptographic hashes, and pseudorandom number generators. [Ch. 11]

cryptography—A discipline of theoretical computer science that specializes in the study of cryptographic transformations and protocols. [Ch. 11]

data integrity—Authenticity of the apparent content of a message or file. [Ch. 11]

decrypt—To perform a reverse cryptographic transformation on a previously encrypted message to obtain the plaintext. Compare with *encrypt*. [Ch. 11]

digital signature—An authentication tag computed with public-key cryptography. [Ch. 11]

discretionary access control—A property of an access control system. In a discretionary access control system, the owner of an object has the authority to decide which principals have access to that object. Compare with *non-discretionary access control*. [Ch. 11]

encrypt—To perform a cryptographic transformation on a message with the objective of achieving confidentiality. The cryptographic transformation is usually key-driven. Compare with the inverse operation, **decrypt**, which can recover the original message. [Ch. 11]

explicitness—A property of a message in a security protocol: if a message is explicit, then the message contains all the information necessary for a receiver to reliably determine that the message is part of a particular run of the protocol with a specific function and set of participants. [Ch. 11]

flow control—In security, a system that allows untrusted programs to work with sensitive data but confines all program outputs to prevent unauthorized disclosure. [Ch. 11]

forward secrecy—A property of a security protocol. A protocol has forward secrecy if information, such as an encryption key, deduced from a previous transcript doesn't allow an adversary to decrypt future messages. [Ch. 11]

freshness—A property of a message in a security protocol: if the message is fresh, it is assured not to be a replay. [Ch. 11]

key-based cryptographic transformation—A cryptographic transformation for which successfully meeting the cryptographic goals depends on the secrecy of some component of the transformation. That component is called a cryptographic key, and a usual design is to make that key a small, modular, separable, and easily changeable component. [Ch. 11]

key distribution center (**KDC**)—A principal that authenticates other principals to one another and also provides one or more temporary cryptographic keys for communication between other principals. [Ch. 11]

list system—A design for an access control mechanism in which each protected object is associated with a list of authorized principals. [Ch. 11]

mediation—Before a service performs a requested operation, determining which principal is associated with the request and whether the principal is authorized to request the operation. [Ch. 11]

message authentication—The verification of the integrity of the origin and the data of a message. [Ch. 11]

message authentication code (**MAC**)—An authentication tag computed with shared-secret cryptography. MAC is sometimes used as a verb in security jargon, as in "Just to be careful, let's MAC the address field of that message." [Ch. 11]

name-to-key binding—A binding between a principal identifier and a cryptographic key. [Ch. 11]

non-discretionary access control—A property of an access control system. In a non-discretionary access control system, some principal other than the owner has the authority to decide which principals have to access the object. Compare with *discretionary access control*. [Ch. 11]

origin authenticity—Authenticity of the claimed origin of a message. Compare with *data integrity*. [Ch. 11]

page fault—See *missing-page exception*.

pair-and-spare—See *pair-and-compare*.

password—A secret character string used to authenticate the claimed identity of an individual. [Ch. 11]

plaintext—The result of decryption. Also sometimes used to describe data that has not been encrypted, as in "The mistake was sending that message as plaintext." Compare with *ciphertext*. [Ch. 11]

prepaging—An optimization for a multilevel memory manager in which the manager predicts which pages might be needed and brings them into the primary memory before the application demands them. Compare with *demand algorithm*.

presented load—See *offered load*.

principal—The representation inside a computer system of an agent (a person, a computer, a thread) that makes requests to the security system. A principal is the entity in a computer system to which authorizations are granted; thus, it is the unit of accountability and responsibility in a computer system. [Ch. 11]

privacy—A socially defined ability of an individual (or organization) to determine if, when, and to whom personal (or organizational) information is to be released and also what limitations should apply to use of released information. [Ch. 11]

private key—In public-key cryptography, the cryptographic key that must be kept secret. Compare with *public key.* [Ch. 11]

protection—1. Synonym for *security.* 2. Sometimes used in a narrower sense to denote mechanisms and techniques that control the access of executing programs to information. [Ch. 11]

protection group—A principal that is shared by more than one user. [Ch. 11]

public key—In public-key cryptography, the key that can be published (i.e., the one that doesn't have to be kept secret). Compare with *private key.* [Ch. 11]

public-key cryptography—A key-based cryptographic transformation that can provide both confidentiality and authenticity of messages without the need to share a secret between sender and recipient. Public-key systems use two cryptographic keys, one of which must be kept secret but does not need to be shared. [Ch. 11]

repudiate—To disown an apparently authenticated message. [Ch. 11]

secrecy—Synonym for *confidentiality.* [Ch. 11]

secure area—A physical space or a virtual address space in which confidential information can be safely confined. [Ch. 11]

secure channel—A communication channel that can safely send information from one secure area to another. The channel may provide confidentiality or authenticity or, more commonly, both. [Ch. 11]

security—The protection of information and information systems against unauthorized access or modification of information, whether in storage, processing, or transit, and against denial of service to authorized users. [Ch. 11]

security protocol—A message protocol designed to achieve some security objective (e.g., authenticating a sender). Designers of security protocols must assume that some of the communicating parties are adversaries. [Ch. 11]

shared-secret cryptography—A key-based cryptographic transformation in which the cryptographic key for transforming can be easily determined from the key for the reverse transformation, and vice versa. In most shared-secret systems, the keys for a transformation and its reverse transformation are identical. [Ch. 11]

shared-secret key—The key used by a shared-secret cryptography system. [Ch. 11]

sign—To generate an authentication tag by transforming a message so that a receiver can use the tag to verify that the message is authentic. The word "sign" is usually restricted to public-key authentication systems. The corresponding description for shared-secret authentication systems is "generate a MAC". [Ch. 11]

speaks for—A phrase used to express delegation relationships between principals. "A speaks for B" means that B has delegated some authority to A. [Ch. 11]

threat—A potential security violation from either a planned attack by an adversary or an unintended mistake by a legitimate user. [Ch. 11]

ticket system—A security system in which each principal maintains a list of capabilities, one for each object to which the principal is authorized to have access. [Ch. 11]

trusted computing base (**TCB**)—That part of a system that must work properly to make the overall system secure. [Ch. 11]

Index of Chapter 11

Design principles and hints appear in *underlined italics*. Procedure names appear in SMALL CAPS. Page numbers in **bold face** are in the chapter Glossary.