TOPICS IN THE ENGINEERING OF COMPUTER SYSTEMS

CHAPTER 11
PROTECTION OF INFORMATION

FEBRUARY 2005

TABLE OF CONTENTS

Glossary

access control list (ACL)—A list of principals authorized to have access to some object.

active attack—An attack in which the intruder can create, delete, and manipulate messages destined for a principal (including substituting one message for another and replaying a message that was copied earlier).

appropriateness—A property of a message in cryptographic protocol: if the message is appropriate, then it is a member of this instance of this protocol (i.e., it is not copied from another instance of this protocol or from another protocol).

authentication—Verifying the identity of a principal or the authenticity of a message (its origin and integrity).

authentication keys—Cryptographic keys used for signing and verifying messages.

authentication tag—A cryptographically-computed string, associated with a message, that allows a receiver to verify the authenticity of the message.

authorization—Granting a principal permission to perform some operation, such as reading certain information.

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.

certificate—A message that certifies the binding of a principal identifier to a key.

certificate authority (CA)—A principal that signs certificates.

certify (v.)—To check the accuracy, correctness, and completeness of a security or protection mechanism.

cipher—A cryptographic transformation that transforms its input in such a way that it is difficult to reconstruct the input from the output, unless one knows a secret.

ciphertext—A message that has been sealed. Compare with plaintext.

cleartext—Synonym for plaintext.

confidentiality—Limiting information access to authorized principals. Synonym for secrecy.

confidentiality keys—Cryptographic keys used to seal or unseal messages.

confinement—Allowing a (perhaps) untrusted program to have access to data, while ensuring that the program cannot release information.

covert channel—A hidden communication channel in flow-controlled access-control system.

cryptographic hash function—A cryptographic function that maps messages to short values in such a way that it is difficult to construct two messages having the same hash value. Thus, if an attacker has the output of a hash function it is difficult to compute the input message.

cryptographic protocol—A message protocol designed to achieve some security objective (e.g., authenticating a sender). In designing cryptographic protocols each party assumes that all the other parties may be malicious.

cryptography—A discipline of theoretical computer science that specializes in the study of mathematical techniques for the protection and authentication of information.

data integrity—Synonym for message integrity.

digital signature—An authentication tag computed with a public-key system.

discretionary—A property of an access control system. In a discretionary access control system, the owner of an object has at least some control on access to that object. Compare with nondiscretionary.

flow control—An access control system that provides both nondiscretionary and discretionary access.

forward secrecy—A property of a cryptographic protocol. A protocol has forward secrecy if information learned from previous transcripts (e.g., confidentiality keys) doesn't allow an attacker to unseal future ones.

freshness—A property of a message in a cryptographic protocol: if the message is fresh, it has been sent recently (e.g., it is not a replay).

fey—A cryptographic key used for either sealing or authentication. See also shared-secret and public-key system.

fey distribution center (KDC)—A principal that authenticates other principals to one another and also provides temporary key(s) for communication between other principals.

fist system—A protection system in which each protected object is associated with a list of authorized principals.

fediation—Before a service performs a requested operation, determining what real-world person is associated with the request and whether the person is authorized to request the operation.

fessage authentication—To verify the origin and integrity of a message. See message integrity and message origin.

message authentication code (MAC)—An authentication tag computed with a shared-secret system.

message integrity—The authenticity of message content: a message has not been changed since it was sent.

message origin—The alleged origin of the message, as inferred by the receiver from the message content or from other information.

name-to-key binding—A binding between a principal name and a cryptographic key.

nondiscretionary—A property of an access control system. In a nondiscretionary access control system some principal other than the owner limits the ways an owner of an object can control access to the object. Compare with discretionary.

passive attacks—An attack in which the intruder overhears a message destined to a principal and may make a copy for analysis.

password—A secret character string used to authenticate the claimed identity of an individual.

plaintext—A message that is not sealed (i.e., its content is directly readable). Compare with ciphertext.

principal—An agent (a person, a computer, a thread) that makes requests to the protection system. It is the entity in a computer system to which authorizations are granted; thus, the unit of accountability and responsibility in a computer system.

privacy—The ability of an individual (or organization) to decide if, when, and to whom personal (or organizational) information is released.

private key—The key in a public-key system that must be kept secret. Compare with public key.

protection—1) security; 2) used more narrowly to denote mechanisms and techniques that control the access of executing programs to information.

protection group—Principals that are shared by more than one user.

public key—The key in a public-key system that can be published (i.e., the one that doesn't have to be kept secret). Compare with private key.

public-key system—A key-based cryptographic system in which only one of the two keys has to be kept secret. The key that must be kept secret is called the private key and the key that can be made public is called the public key.

repudiate (v.)—To disown an apparently authenticated message.

revoke (v.)—To take away previously authorized access from some principal.

safety-net approach—An approach to designing systems that must achieve a negative goal.

seal (v.)—To cryptographically transform a message using a key, known as the *sealing* key with the objective of achieving confidentiality. Compare with the inverse operation, *unseal*, which can recover the original message.

secrecy—Synonym for confidentiality.

security—With respect to computer systems, used to denote mechanisms and techniques that control who may use or modify the computer system or the information stored in it.

shared-secret key—The key in a shared-secret system.

shared-secret system—A key-based cryptographic system in which the key for transforming can be easily determined from the key for the reverse transformation, and vice versa. In most shared-secret systems, the keys are identical.

sign (v.)—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 "to sign" is usually restricted to public-key authentication systems; the word "to MAC" is then used for shared-secret authentication systems.)

speak for (v.)—A phrase used to express trust relationships between principals. If A speaks for B, then B has delegated some authority to A.

ticket system—A protection system in which each principal maintains a list of capabilities, one for each object to which the principal is authorized to have access.

trusted computing base (TCB)—That part of a system that must work properly to make the overall system secure.

unseal (v.)—To apply a reverse cryptographic transformation to a sealed message using the unseal key to obtain the plaintext.

verify (v.)—To establish the authenticity of a message by verifying its authentication tag.

Overview

Secure computer systems ensure that users' privacy and possessions are protected against intentionally malicious 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 attacker is a *negative* goal. The designer of a computer system must assure that the attacker cannot breach the security of the system in *any* way. Furthermore, the designer must ensure that an attacker cannot side-step the security mechanism; one of the simplest ways for an attacker to steal confidential information is to bribe someone on the inside.

Because security is a negative goal, it requires a more sophisticated design approach from the one we have seen in the previous chapters. We will introduce the *safety net* approach, which has two guidelines for designers:

1. Consider *everything* that possibly could go wrong. An attacker need find only *one* hole in the security of the system to achieve his objective. The designer must therefore consider any attack that has security implications; these attacks include snooping on private conversations, exploiting programming mistakes in the software, breaking locks, guessing passwords, bribing a guard, reconstructing information from the radiation of electronic circuits, abusing the maintenance procedures to change the system, inspecting the trash for useful information, etc. These attacks might be launched either from the outside (e.g., by someone who doesn't have permission to use the system) or from the inside (e.g., by an employee who may even have permission to use the system).

2. Assume you will make errors. Because the designer must plan for everything that might go wrong, he must also assume he will make errors. Furthermore, because an attacker has no incentive to report errors, the designer must have a plan for detecting errors quickly, repairing them, and avoiding the same mistake in the future. Such a plan includes designing for feedback to find errors quickly, designing for iteration to repair errors quickly, designing multiple layers of defense so that one error won't break the security of the system, certifying the system to ensure that the design meets the specification and the implementation meets the design, and designing audits and checklists so that previous mistakes aren't repeated.

The conceptual model for protecting computer systems against attackers is based on the client/service model, in which a client module, on behalf of some user, sends a request to a service. To achieve security, the service must answer the following three questions before performing the requested operation:

1. What is the origin of the request? (Did the request from the user?)

2. Is this the request that the user made? (Did the attacker modify the request?)

3. Is the user permitted to make such a request?

The service should answer *all* of these questions for *every* request. To protect against inside attacks (users who have the appropriate permissions, but abuse them) or attackers who successfully break the security mechanisms, the service should also maintain audits of who used the system, what authorization decisions have been made, etc. This information can be used to determine who the attacker was after the attack and bring him to justice. In the end, a primary instrument to deter attackers is detection and punishment.

The next section provides a general introduction to security. It discusses possible attacks (section 1), why protection is a negative goal (section 2), presents the safety-net approach (section 3), lays out principles for designing secure computer systems (section 4), the basic protection model for structuring secure computer systems (section 5), an implementation strategy based on minimizing the trusted computing base (section 6), and concludes with a layered implementation of the protection model, which also provides a road map for the rest of this chapter (section 7).

A. 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 module. We saw how to enforce modularity between modules, how to ensure that modules can communicate despite failures in the network, and how modules can name other modules. 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 desire computer systems that can tolerate *malicious* users. By a "malicious" user we mean a user who breaks into systems *intentionally*, for example, to steal information from other users, to deny other users access to services, etc.

Almost all computers are connected to networks, which means that they can be attacked by a malicious user from any place in the world. Not only must the protection mechanism withstand attackers 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 attacker is particularly challenging. Any person who has access to the public network might be able to listen, modify, or even replace packets on the public network.

Although in most secure systems, keeping the bad guys from doing bad things is the primary objective, there is usually also a need to provide users with different 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.

---

* This section draws heavily from "Protection of computer information" by Saltzer and Schroeder, *Proceedings of the IEEE, Vol 63*, No 9, Sept. 1975, pages 1278-1308.

For example, the owner of an account should be allowed to withdraw money from his account, but he 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.

Another challenge in defending against attackers is that the attacker may be a legitimate user of the system. For example, in the banking system the attacker might be a user who has accounts and tries to get access to other accounts. Or more seriously, the attacker might be someone on the staff of the bank who has the privilege of moving money between accounts. Ensuring that bank personnel do not misbehave requires a carefully thought out security plan with checks and balances enforced in the appropriate places.

Of course, in some applications no special security plan for the computer system is necessary. For instance, an externally administered code of ethics or other mechanisms outside of the computer system may protect the system adequately. With the rising importance of computers and the Internet, however, many systems require some protection 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 authors use the terms "privacy," "security," and "protection" in the same way. This chapter adopts the definitions that are commonly encountered in the computer science literature. The term *privacy* denotes a socially defined ability of an individual (or organization) to determine if, when, and to whom personal (or organizational) information is to be released. The term *security* describes techniques that control who may use or modify the computer or the information contained therein. In this chapter the term *protection* is used as a synonym for security.

The goal in a secure system is to enforce a particular privacy policy. An example of a policy in the banking system is that only an owner and 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.

1. Attack classification

The design of any security system starts with identifying the attacks that the system should withstand. There are three broad categories of attacks:

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 intruder 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 attacker 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 intruder be able to see the information he has changed.

3. Unauthorized denial of use: an intruder can prevent an authorized user from reading or modifying information, even though the intruder himself 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.

The term "unauthorized" in the three categories listed above means that release, modification, or denial of use occurs contrary to the desire of the person who controls the information, possibly even contrary to the constraints supposedly enforced by the system. A major complication is that the "intruder" in these definitions may be a user who is legitimately authorized to do other things in the same system.

Defending against unauthorized use requires a broad set of security techniques. Example techniques include:

1. making credit card information sent over the Internet unreadable by attackers,

2. verifying the identity of a user over an untrusted network,

3. labeling files with lists of authorized users,

4. executing secure protocols for electronic voting or auctions,

5. installing a firewall between the public Internet and a company's private network to protect the company's network,

6. shielding the computer to prevent interception and subsequent interpretation of electromagnetic radiation,

7. locking the room containing the computer,

8. certifying that the hardware and software are actually implemented as intended,

9. keeping write-once logs to maintain audits.

A wide range of considerations are pertinent to the engineering of security applications. Historically, the literature of computer systems has more narrowly defined the term *protection* to be just those security techniques that control the access of executing programs to information stored either locally or remotely.

2. Protection is a negative goal

Having a narrow view of protection is dangerous, because the objective of a secure system is to prevent *all* unauthorized use of information. 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 protection and consider any method in which the protection 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, he receives direct feedback: Alice sends the designer a message "I can't read x!" In contrast, with a negative goal as "Eve cannot read file x", the designer must check that all the ways Eve might be able to read x are blocked, and it's likely that the designer won't receive any direct feedback if he slips up. Eve won't tell the designer because she has no reason to and it may not even be in her 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 don't know."

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 system is insecure. In contract, in order 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 think the system is secure, but we don't know for sure."

Another way of appreciating the difficulty of achieving a negative goal is to model a computer system as a state machine with states for the possible states in which the system can be and with links for transitions between states. As shown in Fig. 11-1, the possible states and links form a graph, with the states as nodes and possible transitions as edges. Assume that the system is in some current state, getting into some other state is desired, and there is also one undesirable state. To get to the desired state (a positive goal), we just need to find *one* path through the graph from the current state to the desired state. To avoid ending up in the undesired state (a negative goal), we must identify *all* possible paths in the graph that lead to the undesired state, and use that to determine the set of edges that need to be cut.

3. Safety-net approach

To successfully design systems that satisfy negative goals, a designer needs a more sophisticated design approach from the one introduced in previous chapters, which we call the

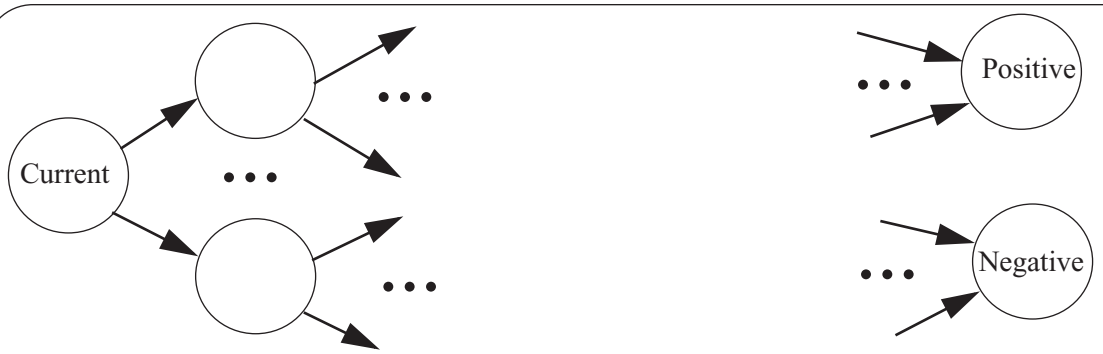


Figure 11-1: Modeling a computer systems as state machine with a current state and states for a positive and a negative goal. To reach the positive state *any* path through the graph will do, but to avoid the negative state *all* paths leading to it must be blocked off.

safety-net approach. This approach has two guidelines for designers:

1. Consider everything that could possibly go wrong; be paranoid. An attacker has to find only *one* hole in the security of the system to achieve his objective. The designer must therefore consider any attack that has security implications.

2. Assume you will make errors—the paranoid attitude. Because the designer must consider everything that possibly could go wrong, he must assume that he himself will make mistakes. Furthermore, since an attacker has no incentive to report errors, the designer must design the system in such a way that errors during the design, implementation, and operation of the system can be detected and easily repaired.

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

- Design the system for immediate feedback. Since the designer must assume he will make errors, he must design and operate the system in such a way that he will learn about those errors. A designer must design paths for feedback in the system; for example, he must embed security checks in the system itself to verify the design assumptions so that he will be alerted when these assumptions turn out to be false in practice. The designer should also create an environment in which staff and customers are not blamed for system failures, 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.

- Design the system for iteration. Because the designer must assume he will make errors, the designer must plan for a design that allows for iteration. In the specification, design, and implementation phase of a system, the designer wants to catch mistakes early. Therefore, during the design phase, a designer must state all assumptions explicitly so that he can check them and discover problems during the design phase rather than when the system is in operation. Further,

when on discovery of a security hole, the designer must check the assumptions and adjust them or repair the design. When a designer discovers an error in the system, the designer must reiterate the whole design and implementation process.

- Certify the security of the system. The designer must *certify* that the implementation indeed implements the design and that there are no loopholes. Certification allows a designer to argue that all possible paths leading to a security breach have been considered.

- Maintain audits of all authorization decisions. Since the designer must assume that legitimate users might abuse their permissions, the system should maintain an append-only log with all authorization decisions made. If, despite all security mechanisms, an attacker (either from the inside or from the outside) succeeds in breaking the security of the system, the log might help in identifying the attacker so that he can be prosecuted after the fact.

As part of the safety-net approach, a designer must consider the environment in which the system is running. He must secure all communication links (e.g., the modem lines that may bypass the router that filters traffic between a private network and a public network), prepare for malfunctioning equipment and trapdoors in software, and determine who is trustworthy enough to own a key to the room that isolates 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 access 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 Eve could access file x?" One should check not only whether Eve has access to file x, but also whether Eve may change the specification of file x's accessibility. The next step is to see if Eve 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 he is "finished" with the information may not be acceptable, 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. It should be apparent that provisions for the dynamics of use are at least as important as those for static specification of protection.

Finally, the safety-net approach suggests that a designer should never believe that a system is 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

attacker to break the security of the system, however, he has to find flaws in the firewall *and* in the end-to-end security mechanisms.

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 attacker has fewer paths available to get to and exploit the undesired state.

4. Design principles

In practice, because protection 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 in systems designed and implemented with security as an important objective, design and implementation flaws provide paths that circumvent the intended access constraints. Appendix 11–B 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 some useful principles to guide the design towards minimizing the number of security flaws in an implementation. The three main design principles for protection mechanisms are:

a) *Make designs open*. The design should not be secret. The mechanisms should not depend on the ignorance of potential attackers, but rather on the possession of specific, more easily protected, keys or passwords. This decoupling of protection mechanisms from protection 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 can convince himself that the system he is about to use is adequate for his purpose. Finally, it is simply not realistic to maintain the secrecy of any system that receives wide distribution. More specific examples of this principle include:

1. *Minimize secrets*. Since it is difficult to keep a secret, minimize what must be kept secret. If the secret is comprised, it must be replaced; if the secret is minimal, then replacing a secret is easier.

2. *Be explicit*: 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 attacker might be able to trick a receiver into interpreting the message in a different context and break the security of a protocol.

3. *Match mental models*: the design should be transparent enough to a user that he can easily map his mental image of protection goal to the provided protection

mechanism. It is essential that the human interface be designed for ease of use, so that users routinely and automatically apply the protection mechanisms correctly. If a user must 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 his protection goals matches the mechanisms, mistakes will be minimized. If a user must translate his mental model of protection needs into a radically different specification language, errors are inevitable. Ideally, protection mechanisms should make a user's computer experience better instead of worse.

b) *Apply economy of mechanism*: Keep the design as simple and small as possible. This widely applicable principle applies to any aspect of a system, but it particularly applies to secure systems. Verifying the security of a system is difficult, because security is a negative goal. Every access path must be considered, 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 protection mechanisms may be necessary. For such techniques to be successful, a small and simple design is essential. More specific examples of this principle include:

1. *Mediate all accesses*: *Every* access to *every* object must be checked for authority. This principle, when systematically applied, is the primary underpinning of the protection system. It forces every access to be explicitly authorized, including ones for initialization, recovery, shutdown, and maintenance. It implies that a foolproof method of verifying the origin and integrity of every request must be devised. This principle applies to services mediating requests, as well as to kernels mediating supervisor calls and virtual memory managers 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.

2. *Minimize common mechanism*: Minimize the amount of mechanism common to more than one user and depended on by all users. This principle reduces the amount of hardware and software that all users are dependent on, which makes it easier to verify if there are any undesirable security implications. 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. 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.

c) *Use fail-safe defaults*: Base access decisions on permission rather than exclusion. This principle means that lack of access should be the default, and the protection scheme identifies 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, since it will 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 which may go unnoticed in normal use.

1. *Perform operations with least privilege*: Every requested operation should be performed using the least set of privileges necessary to complete the operation. This principle limits the damage that can result from an accident, error, or an inside intruder. The military security rule of "need-to-know" is an example of this principle.

2. *Separate privileges*: It may be useful to require that two principals give their approval before authorizing a requested operation. This approach helps in protecting against inside intruders. Two inside persons must conspire to subvert the system, which is harder to do and easier to detect than if a single person behaves badly.

Much software on the Internet and on personal computers fails to follow these principles. The reasons why are different for the Internet and personal computers. When the Internet was started, it was determined that software implementations of cryptographic techniques (see appendix 11–A) would increase latency to unacceptable levels. Hardware implementations of cryptographic operations were considered too expensive. Finally, the US government enforced rules that limited the use of cryptography. Since, historically, the Internet was primarily used by academics—a cooperative community—the lack of security was initially not considered a serious defect.

In 1994 the Internet was opened to commercial activities, which resulted in Internet sites storing more valuable information and thus attracting many more attackers. 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 provided in an ad-hoc fashion instead of following an overall plan based on established security principles.

For various historical reasons, most personal computers come with no 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 with the arrival of the Internet, the desire to get online created a need to address the security problems. Furthermore, because of the 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 and are easily accessible over networks, personal computers have become a prime target for attackers.

The designers of the personal computer didn't originally foresee the developments of network access. When they later did respond to security concerns, the designers simply added security mechanisms that could be found in higher-end computers. 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 so that it can take advantage of the hardware protection mechanisms, while they are also being hit at the same time with requirements for good network security. 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.

5. Protection model

The client/service model provides a good starting point for constructing secure systems. Client and service modules interact only through arms-length transactions, defined by messages. Conceptually, to be secure, the service must mediate *every* request, including requests to configure and manage the service. In this model, the terms client and service should be interpreted broadly. The client module might be a Web browser, a user's workstation, the processor running a thread, etc. A service module might be a file service, a kernel, a virtual memory manager, etc. The client and service might be implemented using hardware, software, or both.

Client modules usually send messages on behalf of some entity that corresponds to a person outside the computer system; we call such an entity a *principal*. Examples of principals include the user's smart card, his client workstation, and a thread that is running the user's program. The principal is the unit of authorization in a computer system, and therefore also the unit of accountability and responsibility. Using these terms, mediating a request is asking the question, "Is the principal who originated the message authorized to request the operation specified?"

An intruder has two basic approaches to break the protection mechanisms of the service: he can circumvent the message interface or he can manipulate legitimate messages. To circumvent the message interface, the intruder must create or find another opening in the service. A simple opening for an intruder might be a private internal network (e.g., a modem line), which is not mediated. If the intruder finds the phone number (and perhaps the password to dial in), the intruder can gain control over the service. A more sophisticated way to create an opening is a *buffer-overrun attack*, common on services written in the C programming language. In these attacks, an intruder overruns a message buffer with a carefully constructed message; this message is long enough that it overwrites the values on the service's stack with new values (e.g., replacing the return address of a procedure) causing the service to execute a program under the control of the attacker, which then creates an interface for the attacker that is not checked by the service.

The basic approach to protection against these kinds of attacks is to make sure that there is really only *one* way into the service and that all messages are carefully checked to ensure that they adhere to their specified formats. Conceptually, we want to build a wall around the service with one small opening through which all messages pass. The small opening is protected by a guard who mediates requests. The guard is often independent of the service to simplify the implementation of the guard and the service.

If a service module is only accessible through a guard, then an intruder has only one option left to break the security of the service, namely by manipulating messages. There are two classes of message attacks: *active attacks* and *passive attacks*. In active attacks, the intruder opens the wire to delete or insert bits. The attacker may substitute one message for another, replay a message that was copied earlier, or modify a message that was copied earlier and replay the modified version. In passive attacks, the intruder taps the wire to make a copy of the message between client and service for analysis. Because an active attacker is more powerful than a passive attacker defending against active attacks is more challenging than defending against passive attacks.

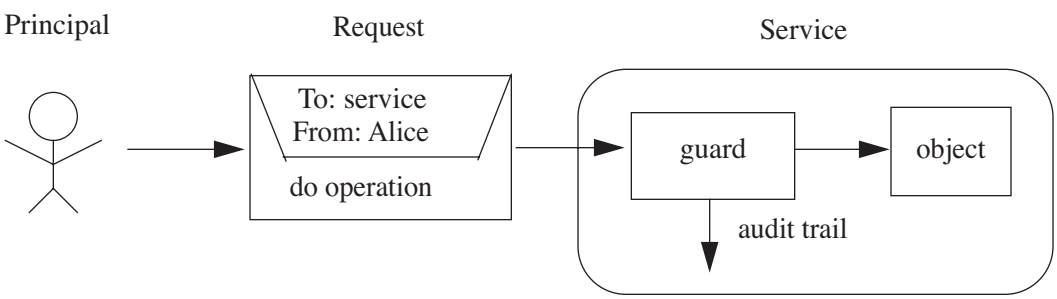


Figure 11-2: The protection model. A principal sends a request to a service. The guard verifies if the principal listed in the message is indeed the source of the message (authentication), and determines whether to grant the requested operation on the object, or to deny it (authorization). If confidentiality of the request is desired too, the message might be sealed to that an attacker cannot discover the request in the message.

Figure 11-2 depicts the basic model to protect services against active attacks. The guard must make the question, "Is the principal who originated the message authorized to request the operation specified?" more precise. The question has essentially three subparts to it:

1. Who is this principal making the request? The guard needs to establish the origin of the message.

2. Is this request actually the one that the user made? The guard needs to establish the integrity of the message.

3. Is the user permitted to make the request? The guard needs to establish if the principal is authorized.

Answering these questions is called *mediation*. Answering these questions on *all* requests is *complete mediation*. We desire complete mediation in any secure system: the guard must check every requested operation, including ones that deal with the management of the service. (In the literature, a guard that provides complete mediation is usually called a *reference monitor.*)

The first two of the three mediation questions fall in the province of *authentication* of the request. Authentication mechanisms allow a guard to verify the identity of the principal and the integrity of the message. After answering the authentication questions, the guard knows who the principal associated with the request is and that no attacker has modified the message.

The third, and final, question falls in the province of *authorization*. Authorization techniques allow the guard to determine if a principal is permitted to request the specified operation on an object. Thus, after answering the three questions, the guard knows it is performing an operation on behalf of the legitimate principal and that the requested operation is authentic.

Complete mediation applies broadly to computer systems. Whether the messages are Web requests for an Internet store, read-write 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, and kernel, respectively.

To protect against passive attacks, information stored at the service and transmitted in messages must be kept private. Keeping things private falls in the province of *confidentiality (*sometimes called *secrecy)*. Confidentiality techniques limit information to authorized principals.

These techniques by themselves are not enough. The designer of a secure service must assume that an attacker might be able to impersonate a legitimate principal and thereby breach the security provided by complete mediation—an example of the paranoid design attitude. There are two ways an intruder might achieve this goal, despite all the protection mechanisms. First, a design or implementation error might allow the intruder to break or circumvent the protection mechanisms. Second, the intruder might be a legitimate principal (e.g., a disgruntled employee) or may have been able to convince, perhaps through bribery, a legitimate principal to act on his behalf. Measures against badly behaving principals are the final line of defense against attackers who successfully break the security of the system, thus appearing as 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) maintaining a secure audit trail of the authorization 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 must be maintained securely; an intruder must not be able to modify the archived data or the audit. Measures to secure archives and audit trails include using a storage medium that is 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 audits may help in figuring out what happened (e.g., what data has been damaged) and which principal did it. The audit might also be useful as a proof in court to punish attackers, which, in the end, is a primary instrument to deter attackers from breaking into computer systems. These measures can be viewed as an example of defense in depth—if the first line of defense works well, these measures are unnecessary.

6. Trusted computing base

Implementing the protection model described above 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.

When designing a secure system, we split 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 when a designer develops a new module, he doesn't have to worry about getting the trusted modules right. The collection of trusted modules is usually called the *trusted computing base* (TCB).

Once this separation has been worked out, the challenge is designing and implementing a trustable 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 kind of attacks we are protected and which kinds we are vulnerable to.

- Design a minimal TCB. Use good tools (such as authentication logic, which we will discuss in section F) 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 if 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 the system running in the computer is the one that was actually implemented. The problem in computer security is typically *not* inventing clever mechanisms and architectures, but ensuring that the installed system actually meets the design and implementation. In order to perform an end-to-end check, one hires *a tiger team* (e.g., a bunch of high-school hackers) to attack the system to see if the security requirements are met.

When a bug is detected and repaired, one reviews the assumptions to see which ones were wrong or missing, repairs the assumptions, and repeats 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 his 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 repair, after the security failure

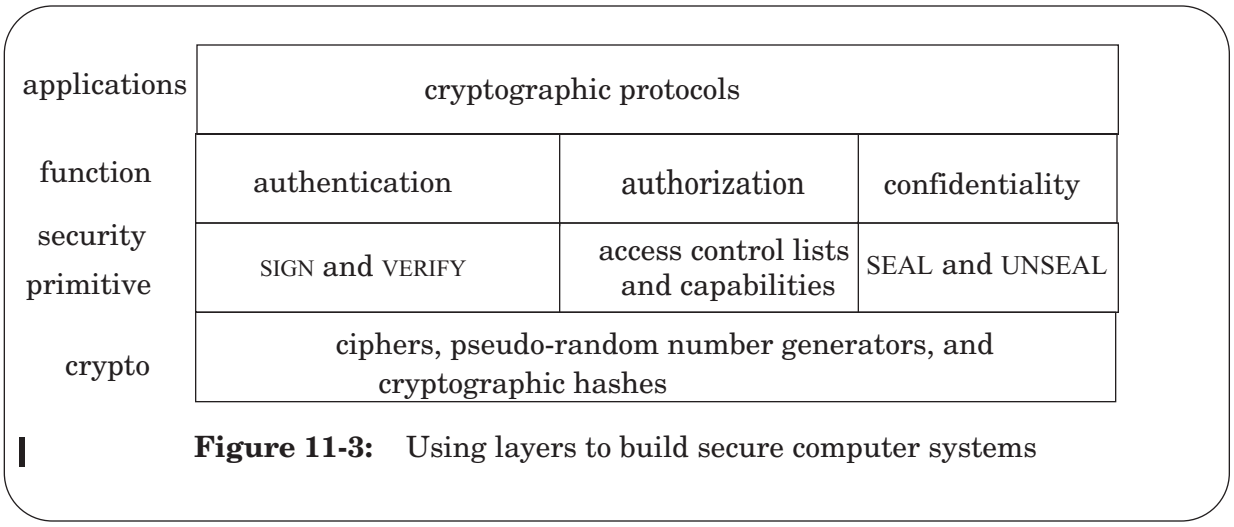


Figure 11-3: Using layers to build secure computer systems

happened, is small compared to designing a system that negates the risk.

7. Using layering to build secure computer systems

A typical implementation of the protection model is layered (see Fig. 11-3). The bottom layer contains the cryptographic transformations: ciphers, pseudo-random number generators, and cryptographic hashes. The transformations can be used protect against attacks on a message. They hide the content of a messages or allow the guard to establish if the message has been modified. These transformations are studied by a field of mathematics, called *cryptography*.

Our interest in cryptographic transformations is not the underlying mathematics (which is fascinating by itself), but that these transformations can be used to implement security primitives, which are in the next layer up. SIGN securely attaches a signature to a message and VERIFY checks using the signature if the message hasn't been modified. SEAL seals a message so that it cannot be read by attackers and UNSEAL unseals the message so that the recipient can read it. Access control lists and capabilities can be used to implement authorization.

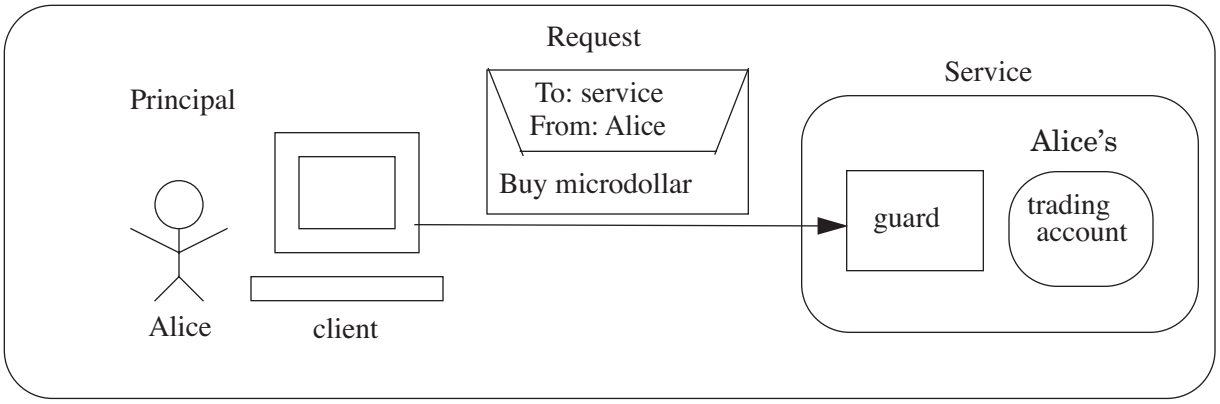
Cryptographic protocols, the top layer, combine the security primitives to implement secure client/server applications such as setting up a secure connection from a Web browser to a Web server.

The rest of this chapter follows the layering picture closely. The cryptography we outsource to the mathematicians, but a brief summary of how to implement the security primitives is provided in appendix 11–A. Section B presents SIGN and VERIFY, and how to use them for authentication. Section C discusses how to use capabilities and access control lists for authorization. Section D describes SEAL and UNSEAL and how to use them for confidentiality. Section E introduce cryptographic protocols and some usages. And, because authentication is the foundation of security, section F discusses cryptographic protocols for authentication and how to reason about them systematically.

B. Authentication

Most security policies involve real people. For example, a simple policy might say 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 principal, the internal computer representation of responsibility inside 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 protection model, the set up for authentication can be presented pictorially as follows:



A principal (Alice) asks her client computer to send a message "Buy 100 shares of Microdollar" to her favorite electronic trading service. An attacker may be able to copy the message, delete it, modify it, or replace it. As explained in section A, when Alice's trading service receives this message, the guard must establish two important facts related to authentication:

1. Who is this principal making the request? The guard must establish if the message indeed came from the real-world person named "Alice." More generally, the guard must establish what the origin of a message is.

2. Is this request actually the one that Alice made? Or, for example, has an attacker modified the message? The guard must establish the integrity of the message.

This section provides the techniques to answer these two questions. These techniques form the foundation of computer security. Only when the guard has identified the principal associated with a request, can the guard decide whether to authorize the requested operation. Also, if the guard can authenticate the principal associated with each operation reliably, it can construct strong audit trails. If there is a breach of security later, we can inspect the audit

trail, perhaps find the operation that led to the breach, and identify the principal associated with the breach. This information might then be used as a proof in court to punish the principal.

1. Separating trust from authenticating principals

Authentication divides into two distinct problems: authenticating the principal associated with a request and establishing trust in the principal. The first problem can be tackled by technical means such as signing messages and passwords, which we discuss later in this section B. 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.

The second problem is a psychological one and is addressed by techniques to develop trust in a principal. The technical means may be able to establish that the real-world identity for a principal is "Alice", but other techniques are required to decide whether and how much to trust Alice. The trading service may decide to honor 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 two 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 "amazing.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 publically that this store is where he buys books, and from all of this information you build some trust that perhaps this bookstore is for real and can be trusted. You order one book from "amazing.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 "amazing.com" is the psychological part. The name "amazing.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 email offer from "amazone.com", you toss it in the trash, knowing that it is not from the name you trust.

When you actually buy a book at "amazing.com", the authentication of principal comes

into play. The mechanical techniques allow you to establish a secure connection to a web site that claims to be "amazing.com", and verify that this web site indeed has the name "amazing.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 "amazing.com." You must decide yourself (the psychological component) who that is and how much to trust them.

In the reverse direction, "amazing.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 connection between your browser and the web site of "amazing.com"' assures "amazing.com" that the credit card number you supply is securely associated with the transaction, and a similar secure connection to the credit card company assures "amazing.com" that the credit card number is a valid principal identifier.

2. Authenticating principals

When the trading service receives the message, the guard knows that the message *claims* to come from the principal 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 principal tells guard its claimed identity, and the guard verifies that claim. This verification protocol has two stages:

> *1.* A name discovery step, in which a real-world principal presents himself to the guard, and agrees on a method by which the guard can later identify him. One must be particularly cautious in discovering the name of a principal. Several of the methods of name discovery listed in chapter 2 are too insecure for any purpose, because an attacker can fake them. At most these methods may be useful as hints, assuming they can be verified in some other way. In practice the name discovery step can be one of the weakest links in the security of a computer system.
>
> *2.* A verification of identity, which occurs at various later times. The principal presents a claimed identity and the guard uses the agreed-upon method to verify the claimed identity.

For example, when Alice created a trading account, the guard might have asked her for her name and a *password* (a secret character string), which the guard stores. This step is the name discovery step. Later when Alice sends a message to trade, she puts in the message her claimed identity ("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." 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 a lot of questions, provide evidence of our identity, and a photograph. Then, 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. Obtaining the passport is the name discovery step through physical rendezvous, presenting the passport and having it examined is the verification step.

To verify a claimed identity the guard and the user both know the unique thing, which they agreed upon during the name discovery step, and nobody else does. Then to allow the guard to verify a claimed identity, the user (or the user's device) communicates identification material along with the claimed identity to the guard. The guard verifies the claimed identity with material that it has stored locally and the identification material supplied by the user. If the guard is able to verify the claimed identity, then the user is authenticated. If not, the guard disallows access, and raises an alert.

The security of this approach depends on, among other things, how carefully the name discovery step is executed. The typical process is that before the user is allowed to use the computer system, he has to see an administrator who will create an entry for the user in a table, and who will store something secret in the table that allows the guard to later verify the claimed identity. The system administrator of course has to authenticate the user through some other means than the secret to be stored in the table. For example, an administrator might ask the prospective user for a passport or a driving license to identify himself. (The passport system also has a name discovery step, and relies on the fact that the agency that issues the passport did a good job in establishing the identity of a person.)

In other applications the name discovery step is a lightweight procedure and the guard cannot trust the claimed identity of the principal. For example, when creating an account at an Internet shop, the principal presents a name and a password. The guard just stores the name and passwords in its table, but it has no direct way of verifying the name; an Internet user is unlikely to be able to present himself physically to the system administrator of the Internet shop, because he might on the Internet at a computer on the other side of the world. Since the Internet shop cannot verify the name of the user, the Internet shop can put little trust in the name. The account exists for the convenience of the user to maintain, for example, a shopping basket; when the user actually buys something, the service doesn't verify the name of principal, but the credit card number. It trusts the credit card company to verify the principal associated with the credit card number, which may involve a physical rendezvous.

The unique thing that the user and guard agreed upon during the name discovery step falls in three broad categories:

- Some unique thing that the user *is*. Humans have unique physical properties. For example, faces, voice, DNA strings, etc. identify a human uniquely. For some of these properties it is possible to design an interface that is acceptable to users. A user speaks a sentence into a microphone and the system generates the voice print. For other properties it is difficult to design an acceptable user interface. For example, a computer system that asks "please urinate in this bucket to log in" is not likely to sell well. The uniqueness of the physical property and if it is easy to reproduce (e.g., replaying a recorded voice) determine the strength of this identification approach. Physical identification is typically a combination of a number of techniques (e.g., voice and face recognition) and is combined with other methods of verification.

- Some unique thing that 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 attacker 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 attacker might bribe the user or physically threaten the user to give the attacker the user's card. It is also difficult to make tamper-proof devices that will not reveal their secret.

- Some unique thing that the user *knows*. The user remembers a secret string, for example, a password or a personal identification number (PIN). 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 attacker to guess the user's secret. Your mother's maiden name is a *weak* secret.

The most commonly employed method for verifying identities is based on passwords, because it has a convenient user interface; users can just type in their name and password on a keyboard. The name discovery step in password authentication is that the real-world person sees an administrator, who checks the identity of the user through some other means (e.g., passport), and then asks the user to invent a password and type it in to the computer. The computer stores the password in some local table with the identity of the principal, corresponding to the user.

Typically, the computer will store a cryptographic hash of the password. A *cryptographic hash function* maps an array of bytes to short values (e.g., 128-bit or 160-bit). The mapping is done in such a clever way that it is difficult to construct two inputs having the same hash value. Thus, if the attacker has the output of a hash function, it is difficult to compute the corresponding input. One can think of the cryptographic hash as an identifier for an array of bytes that is unique but shorter than the message itself. The advantage of storing the cryptographic hash of the password instead of the password is that not even the system administrator can find out what the user's password is. (Design principle: *Minimize what needs to be kept secret*.)

The verification of identity happens when the user logs on the computer. He types in his password, the guard computes the cryptographic hash of the 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 user is denied access.

The most common attack on this method is a *dictionary attack*. In a dictionary attack, an intruder 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 attacker either computes the cryptographic hash of all these strings and compares the result to the value stored in the computer system (if he 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 attacker 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. Several studies have shown that these attacks are effective in practice.

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 in the clear over the network. That is easier said than done. For example, sending the cryptographic hash of the password is not good enough, because if the attacker can capture the hash by eavesdropping, the attacker might be able to determine the secret using a dictionary attack.

A related concern is that passwords tend to be inherently weak, since humans tend to choose passwords that they can easily remember. Because passwords are weak, dictionary attacks work well. Thus, we want to minimize the exposure of passwords, either to prying eyes watching it being typed or to active attackers who want to steal it. To achieve this goal, 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 desire is a protocol between the user and the service that has the following properties:

1. it establishes a confidential communication path between user and service;
2. it authenticates the principal to the guard;
3. it authenticates the service to the principal;
4. the password never travels over the network;
5. the password is only used once per session;
6. attackers cannot guess the password based on network traffic.

Protocols that can establish these kinds of properties are called cryptographic protocols. Before we can discuss such protocols, however, we need some other protection mechanisms first. For example, since each message in a cryptographic protocol might be forged by an attacker, we first need a method to check the authenticity of messages. We discuss message authentication next, the design of confidential communication paths in section D, and the design of cryptographic protocols in section E.

3. Message authentication

When receiving a message, the guard usually wants to know *who* sent the message and *what* was sent. Answering these two questions is the province of *message authentication*. Message authentication techniques prevent an intruder from forging messages that pretend to be from someone else, and allow the guard to determine if an intruder has modified a legitimate message while it was en route.

The problem of message authentication splits neatly into two subproblems: (1) establishing the origin of the message, and (2) establishing the integrity of message content. When origin is established the recipient is confident that it knows which principal sent the message. When integrity is established, the guard is confident that the message received is

the same as the message that the originator sent.

In practice, the ability to establish the origin of 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 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 name of a real-world person with the 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 1.

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

- *message integrity*: the message has not been changed since it was sent (this property is also called *data integrity*);

- *message origin*: the alleged origin of the message, as inferred by the receiver from the message content or from other information, is the same as the actual origin.

The issues of message integrity and origin cannot be separated. Messages that have been altered effectively have a new origin. If an origin cannot be determined, message integrity is a meaningless property (message unchanged with respect to what?). Thus, integrity of message content has to imply message origin, and vice versa.

In the context of authentication, we mostly talk about authenticating messages. However, the concept also applies to communication streams. A stream is authenticated by authenticating successive fragments of a stream. We can think of each fragment as a message from the point of view of authentication. In the rest of this section we will use the word message to mean either a datagram or a fragment of a stream.

4. Authentication is different from confidentiality

The goal of confidentiality (keeping information private) and the goal of authentication (establishing the origin) are related but different goals, and separate techniques are usually used for each objective, similar to the physical world. With surface letters, signatures authenticate the author and sealed envelops protect the letter from being read by others.

Authentication and confidentiality can be combined in 4 ways, 2 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 letter in a signed and sealed envelop, which is appropriate if the content of the message (e.g., it contains a PIN) must be protected and the origin of the

message must be established (e.g., the user who own the bank account).

- Authentication only. An application, a DNS announcement, might require just authentication. This case is like a signed letter in unsealed envelop. It is appropriate, for example, for a public announcement from a CEO to the employees of the company.

- Confidentiality only. Requiring confidentially without authentication is uncommon; who cares if the message is confidential, if you're not even sure that it is right. This case is like a letter in a sealed envelop, but without a signature. The guard has no idea from whom the letter is; what level of confidence can the guard have in the content of the letter? The guard cannot even be assured that the letter was not read by others, because the guard cannot tell if an attacker broke the seal and resealed the message, since the sealed letter is not signed. For these reasons confidentiality only is uncommon in practice.

- 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 importance of authentication without 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 announcement 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
Data: At 10 a.m. Microdollar was trading at $10
```

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 he might decide to trade a particular stock based on that data. The users wants to be ensured that he is receiving the data 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 ensured that an attacker hasn't changed "Microdollar", the price, and the time.

We need a scheme that allows the user to verify the authenticity of the complete *plaintext*, a message whose content is directly readable (also called *cleartext*). As we will see, that can be done if the message is accompanied with an authentication tag that is based on the message content and a secret that is shared between "stock.com"'s Web site and John's browser.

5. Authentication model

In the authentication model there are two secure areas separated by an insecure path (as shown in figure 11-4). In the secure area, the sender (usually called Alice) creates an authentication tag for a message by signing the message. The tag and message are communicated through the insecure area to the receiver (usually called Bob). The insecure

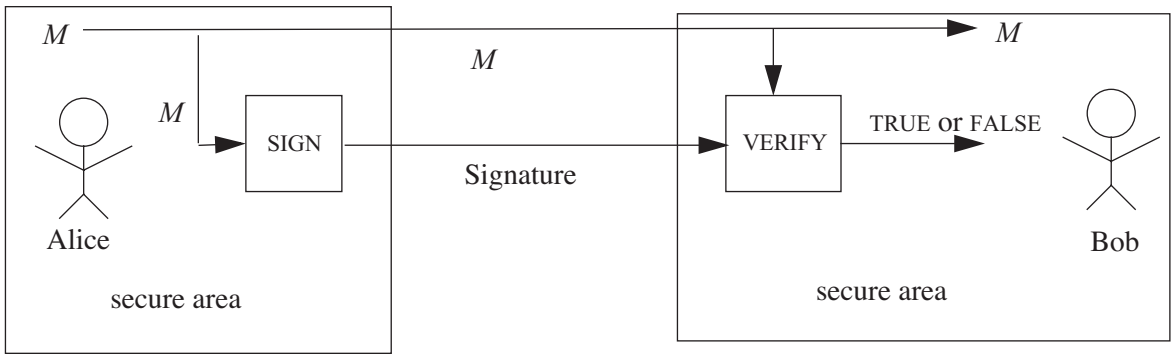


Figure 11-4: Closed design for authentication

path might be an insecure physical wire or a connection across the Internet. The receiver verifies the authenticity of the message by recomputing the tag from the message content. If the recomputed tag matches the received tag, then the received message is authentic; otherwise, it is not. Conceptually, the tag can be transmitted independently of the message, but typically the authentication tag is appended to the message, and the result is sent to the receiver.

In the insecure area, an intruder can attack the tag. The goal of the attack might be to forge a signature, re-use a signature from a previous message on a message fabricated by the attacker, to uncover how SIGN and VERIFY operate, etc. Depending on the sophistication of the intruder, two kinds of attacks are possible: tap the wire to make a copy of the tag for later analysis, or open the wire to delete or insert bits. An intruder who can launch only the first kind of attacks is a passive intruder, and is usually called Eve. An intruder who can launch both kinds of attacks is called an active intruder, and is usually called Lucifer. Lucifer is more powerful than Eve, because Lucifer can substitute one message for another, cut and paste parts of messages, and replay messages copied from earlier transmissions. When designing a cryptographic system, the designer must consider both passive and active attacks.

One approach to designing a crypto system, called a *closed* design, is to keep the construction of VERIFY and SIGN secret on the assumption that if the attacker doesn't understand how VERIFY and SIGN work, it will be difficult to break the tag. This 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 a closed design were noted by Auguste Kerchkoffs more than a century ago*. They led him to propose a design rule, now known as Kerchkoffs' criterion, which is a particular application of the open design principle and least privilege: minimize

* "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).

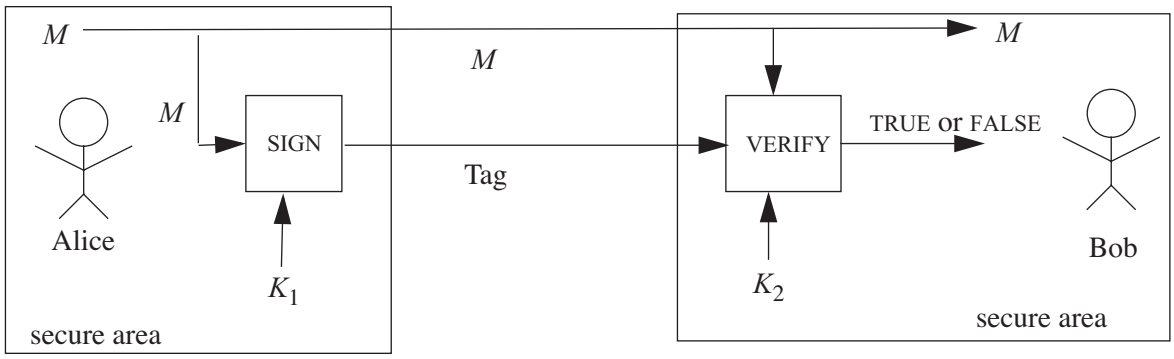


Figure 11-5: Open design for authentication

what needs to be secret.

In an open design (see figure 11-5), we concentrate the secret in a corner of SIGN and VERIFY, and make the secret removable and changeable. An effective way of doing this is to reduce the secret to a bit string; this secret bit string is called a *key*. Thus, the only secrets we keep are two keys, K_1 and K_2.

To authenticate a message, the sender *signs* the messages using a key K_1. Signing generates an *authentication tag*: a key-based cryptographic transformation (usually shorter than the message) that is appended to the message and that can be verified by the receiver. Key-based means that the tag depends both on the plaintext message and on the secret authentication key. The tag and the message are communicated to the receiver through an insecure area. The receiver *verifies* the authenticity of the message, by performing a computation using the message and key K_2. This computation returns TRUE if the message is authentic; otherwise, it returns FALSE.

There are two distinct reasons why open designs are better than closed designs. 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 cryptography uses open designs and reduces the secrets to keys, we might have keys for authentication, keys for confidentiality, etc. To keep them apart, we refer to the keys for authentication as *authentication keys*.

6. Public-key versus shared-secret keys

Depending on the relation between K_1 and K_2, there are two basic approaches to transforming a message: shared-secret systems and public-key systems. A *shared-secret system* is one in which K_1 is easily computed from K_2 and vice versa. Usually in a shared-secret system $K_1 = K_2$. Put another way, if both K_1 and K_2 must be kept secret to maintain

security, it's a shared-secret system.

If K_1 cannot be derived easily from K_2 (and visa versa), the system is *public-key system*. Thus, in a public-key system, only one of the two keys has to be secret; the other one can be made public. Before public-key systems were invented cryptographers worked under the assumption that Alice and Bob needed to have a shared secret for SIGN and VERIFY to work. Because sharing a secret is difficult, this assumption made certain applications of cryptography complicated. The invention of public-key cryptography removed the assumption, resulting in a sea change in the way cryptographers thought, and has led to interesting applications.

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

If Alice signs the message using a shared-secret key, then Bob verifies the signature using the *same* shared-secret key: Bob *recomputes* the authentication tag from the message and the shared-secret key, and then checks that it is equal to the received authentication tag. An authentication tag computed with a shared-secret key is called a *message authentication code (MAC)*. (The verb "to MAC" is used for computing authentication tags using a shared-secret system.)

In the literature, the word "sign" is usually reserved for generating authentication tags with a public-key system. If Alice signs the message using a public key, 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. Bob cannot verify by recomputing the authentication tag, since he doesn't know Alice's private key. However, Alice's public key enables Bob to verify that the authentication tag was computed correctly. An authentication tag computed with a public-key system is called a *digital signature*.

Alice's digital signatures can be checked by anyone who knows Alice's public key, while checking her MACs requires knowledge of the same secret key used by her 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 secret key) forged the message and 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 when 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 Word document 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 Word documents are actually identical in hex. What Alice did was insert an IF field that branched on an external input such as 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!

7. *Properties of* SIGN *and* VERIFY

To get a sense of the complexities, we outline some of the basic requirements and some attacks that a designer has to consider. Referring to figure 11-5, the requirements for the authentication functions for a shared-secret system are as follows (where the tag T = SIGN(M, K)):

1. VERIFY(M, SIGN(M, K), K) returns TRUE;
2. For every $M' \neq M$, VERIFY(M', SIGN(M, K), K) returns FALSE;
3. Knowing M and T doesn't allow an attacker to compute K;
4. Knowing M doesn't allow an attacker to compute T.

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

An equivalent set of properties must hold for public-key systems, where the tag T = SIGN(M, K_1):

1. VERIFY(M, SIGN(M, K_1), K_2) returns TRUE;
2. For every $M' \neq M$, VERIFY(M', SIGN(M, K_1), K_2) returns FALSE;
3. Knowing M and T doesn't allow an attacker to compute K_1 or K_2;
4. Knowing M and K_2 doesn't allow an attacker to compute T.

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 high probability. But, these probabilities are so high that in practice we can treat a good implementation as meeting the requirements

perfectly. The probability of *not* meeting property 2 is much lower than the likelihood of a human error that leads to a security breach.

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

1. Modifications to *M*. An attacker may attempt to change *M*. The VERIFY function should return FALSE even if the attacker deletes or flips only a single bit. Returning to our trading example from above, VERIFY should return FALSE if the attacker changes *M* from "At 10 a.m. Microdollar was trading at $10" to "At 10 a.m. Microdollar was trading at $19."

2. Reordering *M*. An attacker may not change any bits, but just reorder the content of *M*. For example, VERIFY should return FALSE if the attacker changes *M* to "At 1 a.m. Microdollar was trading at $100." (The "0" from "10 a.m." has been appended to "$10.")

3. Extending *M* by prepending or appending information to *M*. An attacker may not change the content of M, but just prepend or append some information to *M*. For example, an attacker may change *M* to "At 10 a.m. Microdollar was trading at $100."

4. Splicing several messages. An attacker may have recorded two messages and their tags, and try to combine them into a new message and tag. For example, an attacker might take "At 10 a.m. Microdollar" 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. Attacks on the underlying cryptographic primitives. Any of the attacks on the crypto primitives are also possible attacks on SIGN and VERIFY.

These requirements and the possible attacks make clear that the construction of SIGN and VERIFY functions is a difficult task. To protect messages against the attacks listed above requires a crypto expert who can design the appropriate 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 with 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 appendix 11–A to get a flavor of the complexities.)

8. 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, is wrong. What Bob really knows is that the message originated from a principal that knows key K_1. The assumption that the key K_1 speaks for or belongs to Alice may not be true. An intruder 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 and not 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 terminates, perhaps unsurprisingly, with trusted physical delivery. When Alice and Bob meet, Alice can give Bob a cryptographic key. This key is authenticated, since 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 active attacker might add, delete, or modify messages on the network. A good crypto 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 generate periodically new keys.

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 exactly depends on if 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 another cryptographic protocol, which we will study in section E.

In some applications it is difficult to arrange for 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 his 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 tell the owner apart from other principals (e.g., burglars)? One option is that the manufacturer or distributor of the device plays the role of Charles. When a person buys the device, he gives his public key to the manufacturer. The device has burned into it the public key of the manufacturer; when the owner turns on the device, the device establishes a secure connection 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 disconnected from a global network and cannot reach the

Sidebar 11-1: Authenticating personal devices: the resurrecting duckling policy

Inexpensive consumer devices have (or will have soon) embedded microprocessors in them and be able to communicate with other devices over inexpensive wireless networks. If household devices such as the home theatre, the heating system, the lights, and surveillance cameras are controlled by, say, a universal remote control, an owner must ensure that these devices (and new ones) obey to his commands, and not to the neighbor's, or worse a burglar's.This situation requires that a device and the remote control can 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"*, 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.

An important question is how the owner imprints the authentication key into a newborn device. Many devices are not powerful enough to perform public-key cryptography. Furthermore, if there are many devices, how can the remote control ensure that it is talking to the right device? Stajano and Anderson propose to solve this problem by arranging for a literal, physical rendezvous. An owner imprints a newborn device by simply touching it with an electronic contact that transfers a shared-secret authentication key into the device, in the clear. Once the owner and the device have each other's authentication keys, they can send authenticated messages over untrusted networks to each other.

Stajano and Anderson have augmented the resurrecting duckling solution to handle multiple owners and recovery from a remote control losing its key.

---

* Frank Stajano and Ross Anderson, *The resurrecting duckling: security issues for ad-hoc wireless networks*. Proceedings of 7th international workshop on security protocols. Springer Verlag Lecture Notes in computer science, 1999.

manufacturer? This solution might also privacy objections: should manufactures be able to track when consumers use devices? The sidebar on the resurrecting duckling provides a solution that allows key distribution to be performed locally, without a central principal involved.

Not all applications deploy a key-distribution protocol that ends in a physical rendezvous. For example, the secure shell (SSH), a popular Internet application to log on to a remote computer, has a simple key distribution protocol that is insecure on the first interaction. When a user logs on to a computer named "amsterdam.lcs.mit.edu", SSH will send a message in the clear to the machine "amsterdam.lcs.mit.edu" asking it for its public key. SSH uses that public key to setup an authenticated and confidential connection with the remote computer. SSH also caches this key and remembers that the key is associated with the name "amsterdam.lcs.mit.edu." The next time the user logs on to "amsterdam.lcs.mit.edu", SSH uses the cached key to set up the connection.

The security risk in SSH's approach is a man-in-the-middle attack: an attacker might be able to intercept the first message that SSH sends to "amsterdam.lcs.mit.edu", and reply with a key that the attacker has generated. When the user makes an SSH connection to that public key, he believes that he is connecting to "amsterdam.lcs.mit.edu", but instead it might

be a computer controlled by the attacker. Thus, when a user first logs on to a remote computer, he cannot assume SSH connected him to the right computer. A user might first carefully browse the files at the remote computer. Are the expected programs there? Are they expected files there? The user can use the answers to these questions to build some trust in the key that it used to establish the secure connection. Once it has built that trust in the key obtained, it can subsequently use the cached version of the key to make secure connections to "amsterdam.lcs.mit.edu."

In the SSH example, the user uses psychological means to bind "amsterdam.lcs.mit.edu" to its public key. The advantage of the approach is that SSH doesn't need a key distribution protocol, which has simplified the deployment of SSH (e.g., there is no need for physical rendezvous) and made it a big success. As we will see in section F, securely distributing keys is a challenging problem.

C. Authorization*

With the techniques from the previous section we can authenticate principals and requests. Once we know which principal is the source of a request, we can tackle the third question of the protection model: is the principal authorized to request the specified operation? Answering this question falls in the province of authorization, and is the topic of this section.

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 if a principal has permission to perform an operation on a particular object. To mediate the request, the service has to answer the three questions listed above.
- *revocation*. This operation removes a previously-granted permission from a principal.

Authorization and revocation increase and decrease respectively the set of principals that have access to a particular object.

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 simpler one, while flow control is the most complex model and is only used in heavy-duty security systems.

1. The simple guard model

In the simple guard model, the service conceptually surrounds each object with a wall that has only one door providing access to the object (see figure 11-6). The service posts a guard at the door who decides whether a principal has access or not. The principal must present a token to the guard, who also has a token, and then the guard compares them. If they match, the principal is granted access. If not, the principal is denied access. The guard may be implemented in hardware or software.

In the guard system, in order to authorize a principal to access an object, the access

---

* This section draws heavily from "Protection of computer information" by Saltzer and Schroeder, *Proceedings of the IEEE, Vol 63*, No 9, Sept. 1975, pages 1278-1308.

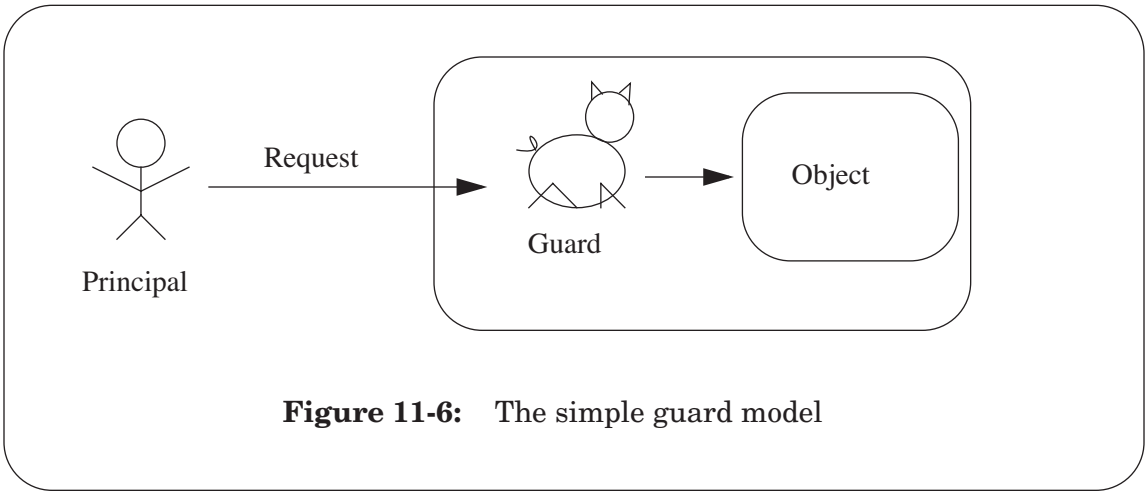


Figure 11-6: The simple guard model

control system gives the guard a token and the principal a matching token. To mediate, the guard matches the principal's token with the guard's token. If they match, access is allowed; otherwise, it is denied.

To revoke, the access control system removes a token, either from the principal or from the guard. The two options lead to different guard systems: the ticket system and the list system. The systems differ in who maintains tokens (i.e., the principal or the guard). We will discuss ticket and lists systems in turn.

1.1. The ticket system

In the *ticket system*, each guard has one token for each object it is guarding. Each principal has a token for each object she is authorized to use. One can compare the set of tokens that the principal owns to a ring with keys. The set of tokens that principal possesses determines exactly which objects the principal can access.

To authorize a principal access to an object, the access-control system gives the principal a matching token for the object. If the principal wishes, the principal can simply pass this token to other principals giving them access to the object.

Tokens, of course, have to be protected. The guard has to be able to check whether a token is valid. Principals shouldn't be able to cook up their own tokens. A token in ticket-oriented system is usually called a *capability*.

To revoke a principal's authorization, the access-control system has to either hunt down the principal and take the token back, or change the guard's token and reissue tokens to any principals who should still be authorized. The first choice may be hard to implement; the second may be disruptive.

1.2. The list system

In the *list system*, revocation is less disruptive. In the list system, there is only one token per principal and each guard has a list of tokens that correspond to the set of principals

that have access. To mediate, a guard must check *all* her tokens when a principal presents her with a token. If the search for a match succeeds, the principal is granted access; if not, the principal is denied access. To revoke access, the access-control system removes the principal's token from the relevant guard's list. The list of tokens is usually called an *access-control list (ACL)*.

1.3. Protection groups

Cases often arise where it would be inconvenient to list by name every principal who is to have access to a particular object, either because the list would be awkwardly long or because the list would change frequently. 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 are to be permitted access to that object.

An access control system can implement protection groups by giving a principal a token for each group of which the principal is a member. A simple way 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, he can specify the set of principal identifiers he proposes to use. His right to use his personal principal identifier is authenticated, for example, by a password. His right to use the group principal identifiers can then be authenticated by looking up the now-authenticated personal identifier on each named group's access control list. The result of this sequence of operations at the end of logging in is that the principal has tokens for each group it belongs to. As before, the guard can then mediate access based on the tokens, either using a ticket system or a list system.

1.4. Agencies

Ticket and list systems each have advantages over the other. A natural question to ask is if it is possible to get the best of both. Agencies 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, when one first shows up the organizers 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).

Table 11-1 compares the different guard systems. In the ticket system access checks amount to comparing two tokens. In the list system access checks require searching a list of tokens. As a result, access checks in the list system are more expensive in time.

In most ticket systems, principals can pass tokens to other principals without involving the guard. This property makes sharing easy (no interaction with the guard required), but makes it hard for a guard to revoke access and to perform audits. In the list system, a token identifies a principal, which makes audits possible; on the other hand, to grant another principal access to an object, it requires an interaction with the guard.

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

Table 11-1: Comparison of access control systems

2. Example: dynamics of use in a multi-user time-sharing system

The dynamics of use is one of the more difficult parts of a design to get right in authorization 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? It should be clear that the overall security of a computer system is for a large part based on how carefully the dynamics of use have been thought through.

The issues in dynamics of use are best illustrated by describing a complete authorization system. We consider the access control system for the UNIX operating system, which is similar to the one used by other multi-user operating systems such as Windows and MacOS X. UNIX was originally designed for time-sharing a computer among multiple users, but that is not the main use of computers anymore; most computers are not time-shared. The protection mechanisms provided for time-sharing, however, are also useful in a single-user system. For example, a user may want to run some programs with different authorities (e.g., one might want to run programs downloaded from the Web under the authority of a principal with few privileges).

To isolate programs (and users) from each other, UNIX creates for each program a thread. The UNIX kernel multiplexes these threads over one or more physical processors (as described in chapter 5).

All shared resources (disk blocks, devices, etc.) are protected by the kernel. The UNIX kernel represents every shared resource as a file. Thus, all authorization decisions can be viewed as if a particular principal should have access to a particular file. To associate running programs with principals, the kernel stores with each thread the user identifier (UID) of the principal on whose behalf the thread is running. When a thread wants to request an operation on a file, it contacts the kernel. The kernel makes the authorization decision. Thus, the kernel is an integral part of the trusted computing base. How does the kernel come into existence?

2.1. Initializing the authorization system

When a system administrator turns on a machine, the computer loads the kernel program from a protected area on an attached storage device. The processor loads the kernel

from the protected disk area into main memory at a predefined memory location and starts executing instructions at the predefined memory address.

During initialization, the kernel program constructs a thread (and an address space) for itself, configures any devices attached to the computer, and starts other programs (e.g., a file service to manage the storage device). As the final step of initialization, the kernel runs the login service to allow users to have access to the computer. The kernel runs the login service as owned by the system user (called *root*) and redirects input on the keyboard to the thread of the login program so that input goes directly to the login service.

2.2. Verifying identities

Users type in their user name and a password to a login program. The login program immediately hashes the password using a cryptographic hash. Then, the login program reads the password file. The password file has one line per user, which contains the user name, the hashed password, a user identifier (UID), a set of group identifiers (GID), and the name for the first program to run on behalf of this user. To restrict who can modify the password file, the access control list for the password file contains only the system user.

The login program searches the password file for the line that matches the user name that was input to the login program. Then, it compares the hashed password in the password file with the hashed password computed by the login program. If they match, the login program asks the kernel to create a thread and address space to run the first program with the UID and GIDs specified in the password file. If hashes don't match, the user is denied access.

When the first program starts other programs, for example to run commands, they inherit their UID and GIDs from the first program. In general, programs inherit by default their UID and GID from their parent. As we will see later, programs are allowed to change their UID and GID in certain restricted cases.

The login program can be viewed as an access control system following the pure guard model. The guard is the login program. The object is the UNIX system. The principal is the user. The token is the password, which is protected using a cryptographic hash function. If the tokens match, access is granted; otherwise, access is denied.

2.3. Authorization

Once a user is logged in, subsequent access control is performed by the kernel based on the UID and GID of the user, using a list system. The kernel stores with each file an ACL. When a thread request access to a file, the thread performs a system call to enter the kernel. The kernel looks up the UID in the data structure that is associated with each thread. Then, the kernel searches the ACL of the file for the UID of the user. If the UID is on the ACL, the user is allowed access; otherwise, she is denied access to the file.

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.

As a further sophistication, for each entry on the ACL, UNIX keeps 3 bits: 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 rights "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 other users are allowed only read access. Thus, users have fine grained control over how they want to share files with other users.

When a program (a thread) requests access to a file, the kernel performs the access check as follows:

1. The kernel checks if the UID of the thread matches the UID of the owner of the file. If so, the owner permission bits are checked.

2. If UIDs do not match, UNIX checks if *any* GID of the thread matches the GID of the file. If so, the group permission bits are checked.

3. If the UID and GIDs do not match, UNIX checks the permission bits for other users.

Let's fit the kernel in the guard model. The guard is the kernel. The principals are threads running programs on behalf of users. The objects are files, which have access control lists. The tokens are UIDs and GIDs. The tokens and access control lists are protected by the UNIX kernel: they are maintained in the kernel address space and user threads have no direct access to these values. (If a thread wants to modify an access control list, then it asks the kernel; the kernel checks if the thread is allowed to modify the list.) Thus, the kernel is an example of a list-oriented guard system. We can view the whole UNIX system as an agent system. It switches from a ticket-oriented system (the login program) to a list-oriented system (the kernel).

2.4. Dynamics of use

The owner of a file is allowed to change the protection bits on a file to authorize principals and revoke rights from principals.

New principals can be created by the system user by editing the password file. The system user also can change ownership of files, etc. The access check algorithm *always* grants access to the system user, independent of who owns the file.

UNIX allows running programs to change UIDs and GIDs in two protected ways. First, a program running with the root UID can make a supervisor call (SETUID) to change the UID of the current running thread. This scheme allows a thread running as the system user to drop privileges. Threads with UIDs other than the system user are not allowed to perform this operation.

Second, UNIX allows threads to gain certain privileges in a constrained manner. For example, an email service that receives an email for a particular user must be able to append the mail to the user's mailbox. Making the target mailbox writeable would allow any user to destroy another user's mailbox. UNIX, therefore, allows the creation of programs that are

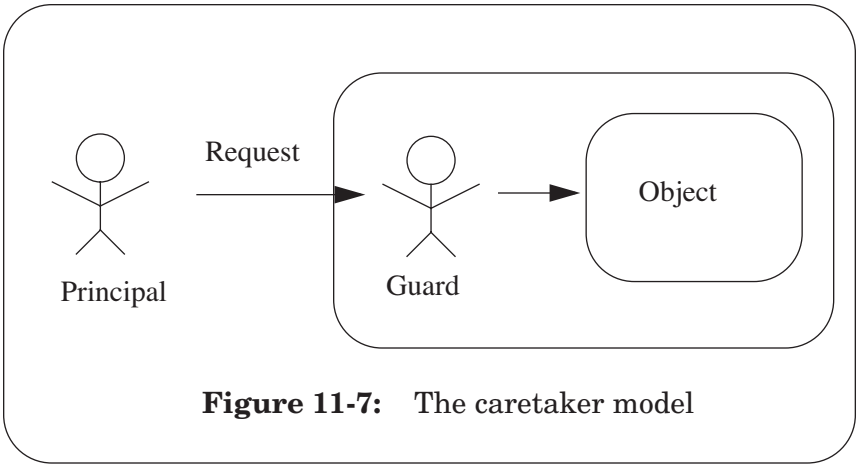


Figure 11-7: The caretaker model

granted additional privileges when they are run. These programs are called *setuid* programs. When such a program is executed, the permissions of the thread are augmented to include those of the UID associated with the program.

2.5. Summary

The trusted computing base of the UNIX system consists of the kernel (which includes the file service), the services that run under the UID of the system user (such as the login program), and every principal that is allowed login as the system user. Files that are owned by a particular user are under the discretion of that user.

3. The caretaker model

The caretaker model differs from the guard model in two crucial ways (see figure 11-7). First, objects are only accessible through the caretaker: the principal has no direct access. So here, there is no door in the wall surrounding the object and the caretaker, only a mail slot for communicating with the caretaker. Two, the caretaker is intelligent. It can enforce arbitrary constraints on access, and it may even 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 5:30pm, but not at any other time.
- A box that can only be opened when two principals agree.
- Releasing salary information only to principals who have a higher salary.

The guard model can be viewed as a specialized instance of the caretaker model—the guard is a very simple caretaker.

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 attackers.

4. Non-discretionary access control with the flow-control model

Our discussion of authorization and authority structures has so far rested on an unstated assumption: the principal that creates a file or other object in a computer system has unquestioned authority to authorize access to it by other principals. In the UNIX example, the owner of the file can give all permissions including the ability to change the ACL, to another user.

This control pattern is *discretionary*: the individual user may, at his own discretion, determine who is authorized to access the objects he creates. In a variety of situations, discretionary control may not be acceptable and must be limited or prohibited. For example, the manager of a department developing a new product line may want to "compartmentalize" his 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 principle of least privilege. 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 not acceptable that any individual employee within his department can abridge the compartmentalization decision merely by changing an access control list on an object he creates.

The manager has a need to limit the use of discretionary controls by his employees. Any limits he imposes on authorization are controls that are out of the hands of his employees, and are viewed by them as *nondiscretionary*. Similar constraints are imposed in military security applications, in which not only isolated compartments are required, but also nested sensitivity levels (e.g., confidential, secret, and unclassified) that must be modeled in the authorization mechanics of the computer system. Nondiscretionary controls may need to be imposed in addition to or instead of discretionary controls. For example, the department manager may be prepared to allow his employees to adjust their access control lists any way they wish, within the constraint that no one outside the department is ever given access. In that case, both nondiscretionary and discretionary controls apply.

The *flow-control* model captures systems that provide both nondiscretionary and discretionary access control. In the flow-control model, there is a wall around the room that contains the object and the principal (see figure 11-8). This wall has a door, which is guarded by a simple guard. But the door is not simple: when the principal enters the room with the object, she is sprayed with indelible dye. If the principal is dyed, the guard will not let her out of the wall.

As an example, money handed over by force to bank robbers is dyed so that the bank robbers can be traced. As another example, non-disclosure agreements may require you for the rest of your life not to disclose the information that the agreement gave you access to. This model is also called the contamination model.

The key reason for interest in nondiscretionary controls is not so much the threat of malicious insubordination as the need to safely use complex and sophisticated programs

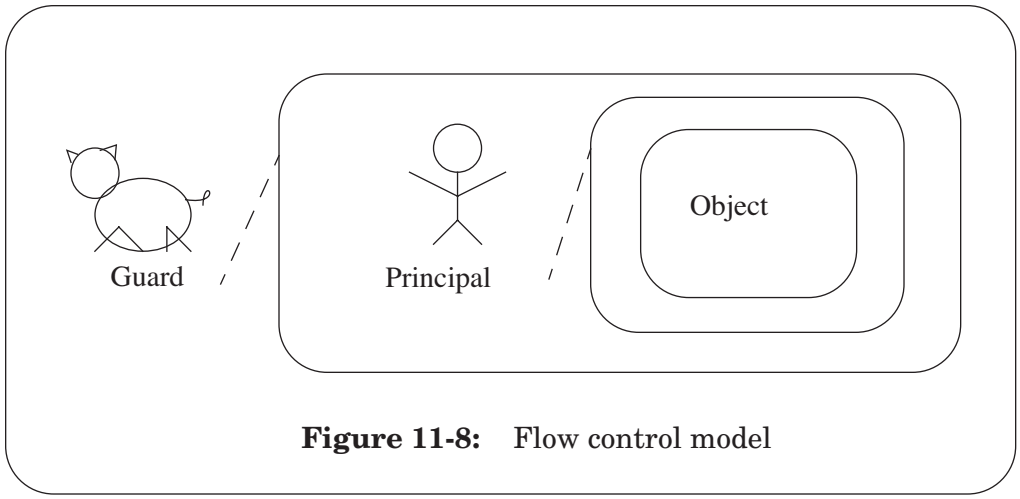


Figure 11-8: Flow control model

created by suppliers who are not under the manager's control. A user may obtain some code from a third party (e.g., a 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. 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 downloaded 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 downloaded program. That is, the downloaded program should run in a domain containing the necessary data, but should be constrained so that it cannot authorize sharing of anything found or created in that domain with other domains.

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 domain 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 and the user would have complete control.

An interesting requirement for a nondiscretionary control system that implements isolated compartments arises whenever a principal is authorized to access 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 thread be constrained to write only into objects that have a compartment set that is a subset of that of

the thread 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 a segment 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. To make the flow-control model more concrete, consider the following example.

4.1. Flow-control example

Consider a company that has information with 3 levels of sensitivity:

- Public (e.g., press releases)
- Internal use only (e.g., the company's phone book)
- Confidential (e.g., production schedule)

Each file in the computer system is labeled with a level in this classification. This classification is modeled after the military security classification (confidential, secret, unclassified).

Every user that has access to the computer system is given a clearance for a certain level. The CEO and upper-level management may have clearance level of 3 (i.e., they have permission to read all files), while the rest of the company has clearance level of 2 (i.e., they don't have permission to read confidential files). Guest accounts may have clearance level 1 (i.e., they have permission to read only public files).

To make this approach work, every thread is labeled with the clearance of the principal it is working for; this clearance is stored in C_{thread}. Furthermore, the system remembers the maximum clearance of data the thread has seen, $C_{maxseen}$. The system follows different access rules for reads and writes, both to enforce authorization and to ensure that secret data never makes it out to the public level.

The read rule is:

- Before reading an object with clearance C_{object}, check that $C_{object} <= C_{\text{thread}}$.
- If so, set $C_{maxseen} = \text{MAXIMUM}(C_{maxseen}, C_{object})$, and grant access.

This rule can be summarized by "no read up." The thread is not allowed to access information with higher class than its clearance.

The corresponding write rule is:

- Allow a write to an object with clearance C_{object} only if $C_{maxseen} <= C_{object}$.

This rule could be called "no write down." Everything written by a thread that read data with clearance L must be given a classification L or higher.

Downgrading of information can only be done after human inspection, since a program cannot exercise judgement. Only a human can establish that information to be written is not sensitive. Developing a systematic way to interpose such human judgements is an important issue in the implementation of the flow-control model.

The flow-control model can be extended to include integrity rules to avoid unreliable information from documents of questionable ancestry flowing into documents that are known to be reliable, and thereby polluting the information in the reliable document.

The flow-control model is very heavy-duty security, but computer systems have been built that support it. The CIA and FBI like these computer systems, as do companies that have very sensitive data to protect. The Department of Defense has a specification for what these computer systems should provide (it's part of the Orange Book, which classifies systems according to their security guarantees).

4.2. Covert channels

Complete confinement of a program in a shared system 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 segments or sending messages over the network. Computer systems with shared resources always contain *covert channels*, which are hidden communication channels. For example, two threads might conspire to send bits by the logical equivalent of "banging against the wall." Avoiding leaking information is called the confinement problem.

Consider two conspiring threads running on a preemptive timesharing system with a single processor. One thread (the sender) could yield its time slice quickly to signal a "0" bit and keep its time slice until preempted to signal a "1" bit. Another thread (the receiver) can measure how much CPU time it is getting, and deduce whether it is receiving a "0" or a "1". This covert channel is low bandwidth, but sufficient, for example, to leak your credit card number from one browser extension to another browser extension.

In practice, just finding covert channels is difficult. Blocking them is an even harder problem: there are no generic solutions.

D. Confidentiality

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 confidential, and not send them in the clear over an untrusted network. 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 data is to isolate 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 isolation makes this sharing impossible. A slight variation, however, of the strict isolation 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. 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 (and vice versa) is to manually transfer by security personnel who will deny or authorize the transfer after a careful inspection of the data. We can view this setup as a rudimentary access control system based on the flow-control model.

For many situations, however, this slightly more relaxed version of strict isolation is still inconvenient. Many users acquire programs created by third parties, run them on their computer, and don't have the capability to verify all acquired software; nevertheless, users want to be assured that their confidential data cannot be read by these untrusted programs. Many users want to access their bank accounts over untrusted networks without having an attacker being able to read that data while it is being transferred over the Internet. In this section, we introduce techniques to provide confidentiality of data stored in an address space and of data that is transferred between computers over untrusted networks.

1. Using virtual memory to provide confidentiality within a shared system

Keeping information private from untrusted programs running on a shared computer system is a good starting point for our study of mechanisms to achieve confidentiality, because most of the necessary machinery has been introduced in chapter 5. The basic approach is to borrow the idea of strict isolation: run each program in its own address space. The goal is to isolate the programs in such a way that they cannot read or write one another's address spaces.

To see how the virtual memory system can be used to achieve confidentiality, let's briefly review how a virtual memory system works. All memory references by a thread (an

executing program) are checked by the virtual memory manager that is interposed in the path to the memory (see figure 11-9). The page map address register points to the page map, which controls exactly which part of memory is accessible. The page map describes an object (in this case, an address space) stored in memory. Since the page map address register is part of the state of the thread, each thread can have a different value in that register, pointing to a different page map, and thus a different address space. A thread has full access to everything in its address space, and no access to any other region of physical memory. In the terminology of chapter 2, the page maps are contexts and the value in the page map address register is an explicit context reference.

To switch the physical processor from one thread to another, the thread manager changes the page map address register. To achieve confidentiality, we need to ensure that *only* the thread manager can change the page map address register; if any thread could load it with any arbitrary value, there would be no confidentiality. The instruction that loads the page map address register with a new value has special controls. Chapter 5 introduced an additional bit in the processor state, the kernel/user mode bit, to guard which thread can load the page map address register. The processor checks all attempts to load the page map address register against the value of the kernel/user mode bit; a thread can change the register only if "kernel" mode bit is set. Only threads in the kernel address space run with the kernel/user mode bit set to "kernel". All that is needed to make the scheme complete is to ensure that the privileged state bit cannot be changed by the user programs except, perhaps, by an instruction that simultaneously transfers control to the kernel program at a planned entry location. The kernel can then load the address of the appropriate page map in the page map address register.

This virtual memory implementation contains four protection mechanisms to achieve confidentiality: pages, address spaces, page map address register, and the kernel/user mode bit. The first mechanism, pages, allows the information to be stored in the distinct pages. All references are forced to go through the page map. The authority check on a request to access

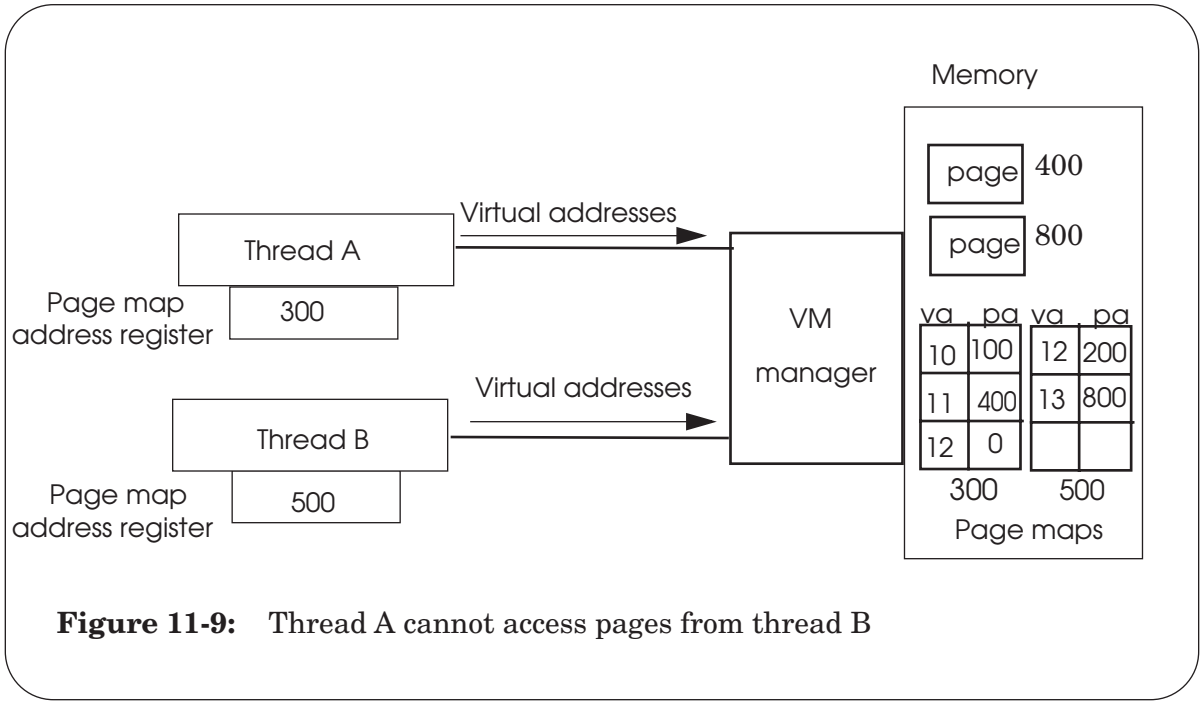


Figure 11-9: Thread A cannot access pages from thread B

memory is simple. The guard checks that the memory location to which access is requested appears in the page map.

The second mechanism is distinct address spaces. The guard is represented by the extra piece of hardware that enforces the restriction on the page map address register, which points to a particular page map. The impenetrable wall with a door is the hardware that forces all references to memory through the page map address register.

The third mechanism protects the contents of the page map address register. The wall, door, and guard are implemented typically in hardware. An executing program requesting to load the descriptor register is identified by the kernel/user mode bit. If this bit is set to "user", indicating that the requester is a user program, then the guard does not allow the register to be loaded. If this bit is set to "kernel", indicating that the requester is the kernel program, then the guard does allow it.

The fourth mechanism protects the kernel/user mode bit. It allows an executing user program (identified by kernel/user mode bit set to "user") to perform the single operation "turn privileged mode bit to kernel and transfer control to the kernel program at a designated entry point." The kernel can set the bit to "user" when switching back to the user program, since it has the privilege to do so. This fourth mechanism is an embryonic form of the sophisticated protection mechanisms required to implement a trusted third party.

The kernel program is a component of all four protection mechanisms, for it is responsible for setting up page maps, maintaining the integrity of the values in the page map address register and the kernel/user mode bit. If the kernel does not do its job correctly, threads could become associated with the wrong page-map values, or a user thread could become labeled with a kernel/user mode set to "kernel." The kernel protects itself from the user programs with the same isolation hardware that separates users, an example of the "economy of mechanism" design principle.

The mechanisms so far provide complete confidentiality of each address space. In some cases a program might want to control at a finer grain what is kept confidential and what is shared with other threads. Consider for a moment the problem of sharing a library program, say, a mathematical function subroutine. We could place a copy of the math routine in each thread that had a use for it. This scheme, although workable, has the obvious disadvantage that the multiple copies require multiple storage spaces. One would like to have some scheme to allow different threads to share a single master copy of the program.

We can share a single copy of the math routine by having entries in different page maps point to the same physical pages that store the math routine (see figure 11-9). When thread A is in control, it can have access to itself and the math routine but nothing else; similarly, when thread B is in control, it can have access to itself and the math routine but nothing else. Since neither thread has the power to change the page map address register, sharing of the math routine has been accomplished while maintaining isolation of thread A from thread B.

The basic mechanism for information sharing raises a remarkable variety of implications. These implications include the following.

1. If thread A can overwrite the shared math routine, then it could disrupt the

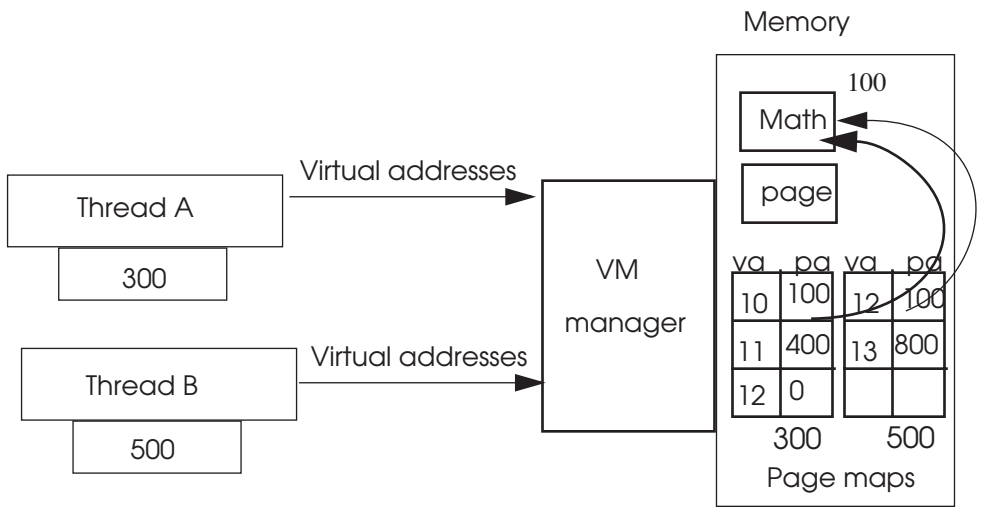


Figure 11-10: Threads A and B share a page with a math routine through their page maps

work of thread B.

2. The shared math routine must be careful about making modifications to itself and about where in memory it writes temporary results, since it is to be used by independent computations, perhaps simultaneously.

3. The scheme needs to be expanded and generalized to cover the possibility that more than one program or database is to be shared.

4. The kernel needs to be informed about which principals are authorized to use the shared math routine (unless it happens to be completely public with no restrictions).

Let us consider these four implications in order. If the shared area of memory is a procedure, then to avoid the possibility that thread A will maliciously overwrite it, we can restrict the methods of access. Thread A needs to retrieve instructions from the area of the shared procedure, and may need to read out the values of constants embedded in the program, but it has no need to write into any part of the shared procedure.

We may accomplish this restriction by extending the page map entries to include access permissions. For example, we may add two bits to each page map entry, one controlling permission to read and the other permission to write in the page defined by that page map entry, as shown in Fig. 11-11.

Thread A's page map entry would have read and write permissions granted, while the page map entry for thread B would permit only reading of data and execution of instructions. An alternative scheme would be to attach the permission bits directly to the shared page. Such a scheme would not work because permission bits attached to the data would provide

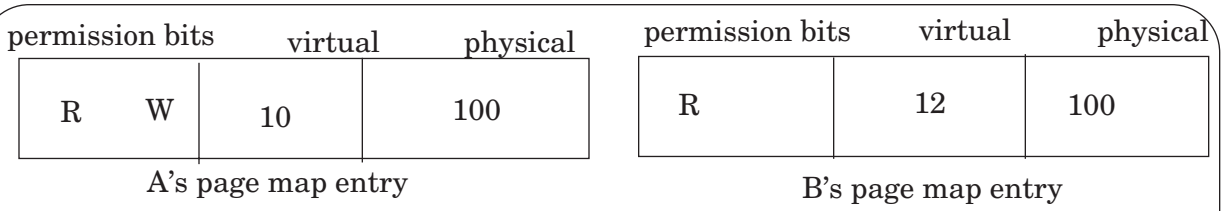


Figure 11-11: Extending page-table entries with permission bits

identical access to all programs that had a page map entry for page 100. Although identical access for all users of the shared math routine might be acceptable, a database could be set up with several users having permission to read but only a few also having permission to write.

The second implication of a shared procedure, mentioned before, is that the shared procedure must be careful about where it stores temporary results, since it may be used simultaneously by several threads. In particular, it should avoid modifying itself. The enforcement of access by the permission bits further constrains the situation. To prevent thread B from writing into the shared math routine, we have also prohibited the shared math routine from writing into itself, since the page map entries do not change when, for example, thread B transfers control to the math routine. The math routine will find that it can read but not write into itself, but that it can both read and write into the area of thread B. Thus thread B might allocate an area of its own address range for the math routine to use as temporary storage. That raises the problem how that area can be named. (The simplest answer is to use the stack, which is a private memory region that has the same name for every thread.)

As for the third implication, the need for expansion, we could generalize our example to permit several distinct shared items merely by increasing the number of shared pages and informing the kernel which shared pages should be addressable by each thread.

As for the fourth implication, informing the kernel which principals have access, that can be addressed by using an authorization system, as discussed in section C.

With an appropriately sophisticated and careful kernel program, we now have an example of a system that completely isolates its programs from one another. Similarly isolated permanent storage can be added to such a system by attaching some long-term storage device (e.g., magnetic disk) and developing a similar protection scheme for its use. The UNIX authorization system described in section C provides such a protection scheme. Since long-term storage is accessed less frequently than primary memory, it is common to implement its protection scheme with more kernel software and less reliance on hardware, but the principle is the same. Data streams to input or output devices can be controlled similarly.

2. Sealing primitives: communicating privately over untrusted networks

Two threads may want to communicate *privately* without unauthorized parties having access to the communicated information. If threads are running on a shared physical

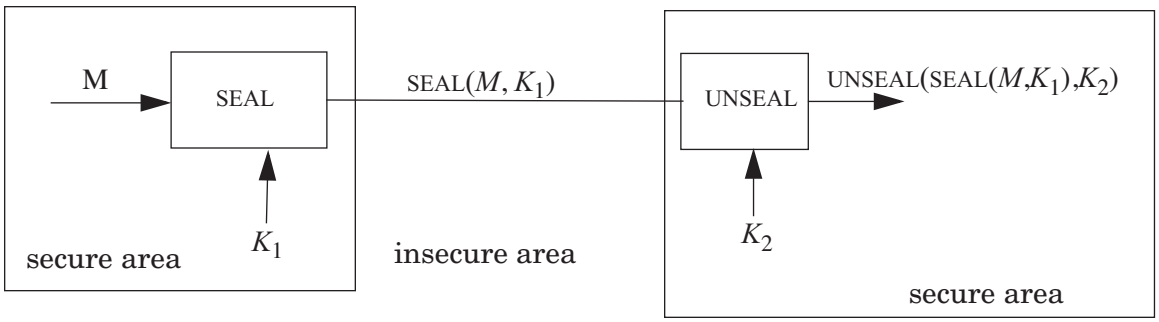


Figure 11-12: Providing confidentiality using SEAL and UNSEAL over untrusted networks

computer, this goal is easily accomplished using the kernel. For example, in LRPC, it is safe to ask the kernel to map a page with a message in both the send and recipient's address space, since the kernel is already trusted; the kernel can read A and B's address space anyway.

If the threads are running 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. We will introduce sealing and unsealing as techniques that allow two parties to communicate without anyone else being able to tell what is being communicated (at least, from just looking at the bits that are communicated between the two parties).

The setup for providing confidentiality over untrusted networks is shown in figure 11-12. Two secure areas are separated by an insecure path. The sender *seals* the message before sending it over an insecure path in such a way that an observer cannot construct the original message from the transformed version, yet the intended receiver can. To transfer the message back to its original, the receiver *unseals* the received sealed message. Thus, one challenge in the implementation of channels that provide confidentiality is to use a sealing scheme that is difficult to break. 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 break the sealed message and reconstruct the original message.

The SEAL and UNSEAL primitive can be implemented using cryptographic transformations. Seal and unseal systems can be either use shared-secret systems or public-key systems. We refer to the keys used for achieving confidentiality as *confidentiality keys*.

In a shared-secret system, Alice and Bob share a key that only they know. Alice computes SEAL(*M, K)* and sends the resulting *ciphertext* to Bob. If our sealing box is good, an attacker will not to be able to get any use out of the ciphertext. Bob computes UNSEAL(ciphertext, K_2), which will recover M in plaintext. Bob can send a reply to Alice using exactly the same system with the same key.

In a public-key system, Alice and Bob do *not* have to share a secret to achieve confidentiality for communication from Alice to Bob. Suppose Bob has a 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, Alice can compute SEAL(M, K_{Bpub}) and send the sealed message over an insecure network. Only Bob can read this message, since he is the only person who has the secret key to unseal her ciphertext message. Thus, using sealing, Alice can ensure that her communication with Bob stays confidential.

In this example, Bob's public key is used for sealing and his private key is used for unseal. So $K_{\mathrm{Bpub}} = \mathrm{K_E}$ and $K_{Bpriv} = K_D$. 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.

The properties that SEAL and UNSEAL should have using a shared-secret system are:

1. Given K, it easy to compute UNSEAL(SEAL(M, K), K);

2. Given *seal* and *unseal*, and given the output of SEAL(M, K), it should be difficult for an attacker who doesn't know K to compute M.

For a public-key signing system, the following properties must hold:

1. Knowing SEAL(M,K_1) and K_2 it is easy to unseal;

2. Knowing K_1, SEAL(M,K_1), and M it is difficult (or impossible) to compute K_2;

3. Knowing K_2, SEAL(M,K_1), and M it is difficult (or impossible) to compute K_1.

To understand the difficulties of implementing SEAL and UNSEAL, consider the attacks that they must withstand. The attacks fall in the following four categories, each category containing stronger attacks.

1. Ciphertext-only attack. The primary information available to the attacker is ciphertext. The problem is that redundancy or repeated patterns in the original message may show through even in the ciphertext, allowing a attacker to reconstruct the plaintext. An attacker may also try to mount a brute-force attack by trying all possible keys.

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

3. Chosen-plaintext attack. The attacker has access to ciphertext that corresponds to plaintext that the attacker has chosen. For instance, the attacker may convince you to transform some data chosen by her to give her information about your transforming system, which may allow her to more easily discover the key. As a special case, the attacker may be able in real time to choose the plaintext based on ciphertext just transmitted. This variant is known as an adaptive attack.

4. Chosen-ciphertext attack. The attacker might be able to select a ciphertext and then see the result of untransforming that ciphertext. One way to mount such an attack is for the attacker to steal a copy of the untransforming box. And again, an attacker may be able to mount an adaptive chosen-ciphertext attack.

Appendix 11–A provides cryptographic implementation of SEAL and UNSEAL that protect against these attacks.

3. Achieving both confidentiality and authentication

The keys used for authentication and confidentiality are typically different, because reusing a key can allow an attacker break the system. (Appendix 11–A shows how reusing a key in a one-time pad can allow an attacker to break the pad.) The sender authenticates with an authentication key, and seals with a confidentiality key. The receiver would use the appropriate corresponding keys to sign and then to verify the authentication tags.

Confidentiality and authentication are thus *separable* and *orthogonal* security objectives that can be achieved by using sealing and signing, respectively:

- For confidentiality only, Alice just seals the message.
- For authentication only, Alice just signs the message.
- For both confidentiality and authentication, Alice first seals and then appends the authentication tag for the sealed message.

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? (At one time it was thought that if messages, when unsealed, revealed recognizable data, they could not have been changed. This thought turns out to be wrong and eradicating it has required a major change in terminology. See sidebar 11-2 on cryptographic terminology.) Therefore, if confidentiality is required, one also provides authenticity.

The second option is common. Much data is public (e.g., routing updates, stock updates, etc.), but it is important to know whether it has been tampered with it or not. In fact, it is easy to argue the default should be that all messages are at least authenticated.

For the third option, one could, in principle, also first authenticate and then seal. It is important to realize that there are limits to the modularity of sealing and confidentiality. To achieve confidentiality and authentication one computes SEAL(M, K_{seal}) and appends SIGN(SEAL(M, K_{seal}), K_{sign}), that is one signs the sealed message. It is, for example, incorrect to compute SEAL(M, K_{seal}) and append SIGN(M, K_{sign}).

A recent paper* on the topic on the order of authentication and sealing suggests that first

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

Sidebar 11-2: An historical perspective on cryptographic terminology

This chapter uses terminology that may seem strange to readers who are familiar with the traditional ways of explaining cryptographic techniques. Specifically, we use the terms "seal" and "unseal" where earlier texts used the terms “encrypt” and “decrypt”. Here is why.

Historically, cryptography was designed to achieve confidentiality. In the process it was noticed that when it is used in certain ways, cryptography allows the recipient of a sealed message to be confident not only of its confidentiality, but also of its authenticity. From this observation arose the misleading intuition that unsealing a message and finding something recognizable inside was an effective way of establishing the authenticity of that message. The intuition is based on the claim that if only the sender knows the key used to seal 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 key. As a result, it is non-trivial to establish that a system based on the claim is secure even in the cases where it is. Many protocols that have been published and later found to be defective were designed using that intuition. Those protocols using this approach that are secure require an outlandish amount of 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 way of using cipher chaining.

A second development occurred more recently: cryptographers noticed that there is usually a need to verify the authenticity even of messages that are not confidential. This realization, together with the realization that authenticity is not an automatic result of applying a cryptographic transformation, led to the conclusion that the design of cryptographic techniques for authentication is a distinct exercise from the design of cryptographic techniques for confidentiality. Although in principle any cryptographic transformation can be used for either purpose, it is unusual to find a single cryptographic transformation that does both operations well. For this reason, it is usually a mistake to use the same transformation for both. Instead, the modern perspective is to develop two distinct modular transformations, which we call “seal” and “sign”, that achieve confidentiality and authenticity, respectively. If an application requires both features, then it applies both transformations. The primary result of this modularity is that arguments about security are much simpler, more compelling, and more likely to be correct.

(Continued on next page)

sealing and then authenticating can cover up the weaknesses in some implementations of the sealing primitives. Also, recently (2001) new cryptographic transformations have been proposed that perform the transformation for sealing and authentication in a single pass over the message, saving time compared to first sealing and then authenticate. In general, the area of cryptography is a fast developing area, and the last word on this topic is not said; interested readers should check out the proceedings of the conferences on cryptography.

Sidebar 11-2, continued: An historical perspective on cryptographic terminology

The raw DES engine, for example, isn't very suitable for *either* signing or sealing; it is usually surrounded by various cipher-feedback mechanisms, and the mechanisms that are good for sealing are generally somewhat different from those that are good for creating a MAC. Similarly, RSA, when used for signing, is usually preceded by hashing the message to be signed, rather than applying RSA directly to the message; as clarified in section B, failure to do it this way can lead to a security blunder.

This distinction between the requirements for signing and sealing has been understood only relatively recently; most older papers and texts assumed that the same cryptographic transformations would be used for both signing and sealing, and they used the terms "encrypt" and "decrypt" for those transformations.

The problem with using this terminology shows up when you try to explain the use of the RSA algorithm for authentication. The simplest description is that you sign by encrypting the message with the signer's private key; you verify by decrypting the authentication tag with the signer's public key. Similarly, when one describes how DES is used for signing, the instructions are to encrypt the message with a particular chaining scheme. But in both cases, the only thing that is confidential is the key that was used.

So the fundamental problem is that the words encrypt and decrypt are in practice used in two quite different ways: in some contexts those words are used to describe a transformation used for confidentiality, and in other contexts those words are labels for a key-based cryptographic transformation and its reverse that may be used for both confidentiality and for authentication. Applying the same label to two fundamentally different but closely related things can be very confusing.

The fix for all of this is to use the terms seal and unseal only for cryptographic transformations intended to achieve confidentiality; for cryptographic transformations intended to achieve authenticity, we use the words sign and verify. We thus adopt the words "cryptographic transformation" to describe the application of key-based cryptographic engines such as DES and RSA. We will treat the words encrypt and decrypt as synonyms for applying a cryptographic transformation and its inverse, but mostly don't use them. The terms "encipher" and "decipher" are another widely-used set of synonyms for applying a cryptographic transformation and its inverse, but we make little use of them.

E. Cryptographic protocols

In the previous sections we have seen how to seal and authenticate messages, with both shared-secret and public-key techniques. These techniques presuppose that the parties have already created and appropriately distributed the cryptographic keys to be used. In particular, the only thing an authentication tag tells the guard about the origin of a message is that the message came from the holder of the key used to verify the tag. The identity of the key holder may be a mystery. In many applications, the guard wants to know if we can *trust* the key to speak for a given principal. These problems (key distribution and authentication of keys) can be solved using cryptographic protocols.

Cryptographic protocols are exchanges of messages designed to allow mutually-distrustful parties to achieve an objective securely. Example objectives include: key distribution, login over untrusted networks using passwords, electronic voting, anonymous, untraceable email, and electronic shopping. In a two-party protocol, the pattern is generally a back-and-forth pattern. With more than two parties, 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 and also to require four or more messages.

The difference between the network protocols discussed in chapter 7 and the cryptographic ones is that standard networking protocols assume that the communication parties cooperate and trust each other. In cryptographic protocols we must assume that the other party may be an attacker and that there may be an outside party attacking the protocol.

1. Example: key distribution

To illustrate the need for cryptographic protocols, let's study two protocols for key distribution. In section B, we have already seen that distributing keys is based on a name discovery protocol, which terminates 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 can exchange keys securely over an untrusted network?

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 sealed or signed), asking:

1. Alice to 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 doesn't know if the message came from Alice, since the source (and destination) field of chapter 7 are not authenticated.

For this message, Charles doesn't really care who sent it, so he replies:

2. Charles to Alice: "To communicate with Bob, use public key K_{Bpub}." (signed by Charles)

Here the message content (the text between quotes) is signed by Charles. When Alice receives this message, she can tell from the fact that this message was signed by Charles 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. Typically writing down the meaning of a message in English makes apparent obvious oversights, such as omitting the name of the intended recipient. This method is an example of the explicitness principle.

To illustrate what problems can be caused because of lack explicitness, suppose that the previous message two were:

Charles to Alice: "Use public key K_{Bpub}." (signed by Charles)

If Alice receives this message, Alice is not able to tell whose public key K_{Bpub} is. An attacker 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:

Charles to Lucifer: "Use public key K_{Lpub}." (signed by Charles)

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:

Someone to Alice: "Use public key K_{Lpub}." (signed by Charles)

From looking at the source address (Someone), she *cannot* tell where that this message came from. The source and destination fields of chapter 7 are not authenticated. Thus, Lucifer might be able to 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 this message says that the message 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 conclude erroneously 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 if message two explicitly contained the name of the person whose public key follows, because Alice would notice that it was Lucifer's, rather than Bob's key.

Returning to the correct protocol, after receiving Charles's reply, Alice can then sign (with her own private key, which she already knows) and seal (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 attacker. 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. Why should she 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 usually 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 that certificate from Charles. Charles may be called a *certificate authority (CA)*. Certificates were invented in Loren Kohnfelder's 1978 MIT 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 confidentiality key Ck_{AC}, and that Bob and Charles have similarly pre-established a shared-secret authentication key Ak_{BC} and a shared confidentiality key Ck_{BC}. Alice begins by sending a message to Charles (again, it does not need to be sealed or signed):

1. Alice to Charles: "Please, give me keys for Bob"

Charles would reply to Alice:

2. Charles to Alice: "Use authentication key Ak_{AB} = … and confidentiality key Ck_{AB} = … to talk to Bob." (sealed and MACed by Charles)

The keys in Charles' reply are freshly-generated random shared-secret keys. Since shared-secret keys must be kept confidential, Charles must both sign *and* seal the message, using the two shared-secret keys Ak_{AC} and Ck_{AC}.

The important part is that this message is both authenticated with Charles' and Alice's shared key Ak_{AC} and sealed with their shared Ck_{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:

1. Charles to Bob: "Use the keys Ak_{AB} =… and Ck_{AB} =… to talk to Alice." (sealed and MACed by Charles)

This message is both authenticated with key Ak_{BC} and sealed with key Ck_{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 Ak_{AB} to authenticate and Ck_{AB} to seal their messages. Charles should immediately erase any memory he has of the two 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 key distribution center was developed in classified military circles and first revealed to the public in a 1973 paper by Dennis Branstad.

A common variation is for Charles to include message (3) to Bob as an attachment to his reply (2); Alice can then forward this attachment to Bob along with her first real message to him. Since message (3) is both authenticated and sealed, 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.

A number of widely-used key distribution and authentication protocols (such as Needham-Schroeder, Otway/Reese, and Kerberos) do not separately authenticate and seal; they instead accomplish authentication by using carefully-crafted sealing, with just one shared key per participant. Although having fewer keys seems superficially simpler, it is very difficult 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 sealing is well worth the simplicity and ease of establishing correctness of the overall design.

For practical 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., computing an XOR). 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 seal subsequent messages in the session using the shared-secret keys distributed 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 for few messages, others are optimized for few cryptographic operations, or may be needed to avoid trusting a third party. Yet others require asynchronous checking (e.g., email), provide only one-way authentication, or require even 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 B).

2. Designing cryptographic protocols

Cryptographic protocols are vulnerable to a number of attacks beyond the ones on the underlying cryptographic primitives. The new attacks to protect against fall in the following categories:

- Known-key attacks. An attacker obtains some keys used previously and then uses this information to determine new keys.

- Replay attack. An attacker records parts of a session and replays them later, hoping that the guards treats the replayed messages as new messages. These replayed messages might trick the guard into divulging useful information to the attacker or taking an unintended action.

- Impersonation attacks. An attacker impersonates one of the other principals in the protocol. A common variant of this attack is the man-in-the-middle attack, where an attacker is relaying messages between two principals and impersonating the principals to each other.

- Reflection attacks. An attacker records parts of a session and replays it to the party that originally sent it. Protocols that use shared-secret keys for authentication are sometimes vulnerable to this special kind of replay attack.

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

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?

- *Appropriateness*. Is this message really a member of this instance of this protocol, or is it copied from an instance of another protocol with an entirely different function between the same parties?

- *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 appropriateness; 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 appropriateness include:

- Include with each message a nonce (a value, perhaps a counter value, serial number, or a timestamp, that will never ever 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 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. Including names provides appropriateness and helps foiling impersonation and reflection attacks.

- Ensure that each message specifies the cryptographic 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 replayed 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 appropriateness and foils replay attacks.

The last two techniques can be viewed as specific instances of explicitness: ensuring 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 attacker 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 first key-distribution protocol above might look more like:

> *1.* Alice to 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?" (signed with Alice's private key)
>
> *2.* Charles to 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 (…)." (signed with Charles' private key)

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 over-designed, but it typically hard to figure out what can be dropped from the protocol. It is surprisingly easy to underdesign a protocol and leave security loopholes. And, unfortunately, a cryptographic protocol may still seem to "work OK"

in the field, until the loophole is exploited by an intruder. Whether a protocol "seems to work OK" for the legitimate participants following the protocol is an altogether different question from whether an intruder 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 seem happy with it. Applying the safety-net approach to cryptographic protocols tells us to overdesign protocols instead of underdesign.

Some applications require more properties beyond freshness, appropriateness, and forward secrecy. For example, a service way want to make sure that a single client cannot flood the service with messages, rendering the service useless. 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 cryptographic protocols is difficult and should be left to experts. The rest of this section presents some common cryptographic protocol problems that appear in computer systems and shows how one can reason about them.

3. An incorrect key distribution protocol

To illustrate the points in the last section, let's consider two different protocols for key distribution. The first protocol is incorrect, the second is 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 sealing. 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 (the notation $\{M\}_k$ denotes signing a message with key k):

1. Alice to CA: {"Give me certificates for Alice and Bob"}

2. CA to Alice: {"Here are the certificates:"

$$\{\text{Alice}, A_{pub}, T\}_{CApriv},$$

$$\{\text{Bob}, B_{pub}, T\}_{CApriv}\}$$

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 a message for Bob (the notation $\{M\}^k$ denotes sealing with key k):

3. Alice to 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 a proposal for a shared-secret key (K_{AB}). The time-stamped shared-secret key is signed by Alice and the complete message is sealed with Bob's public key. Thus, only Bob should be able to read K_{AB}.

Now Alice sends a message to Bob sealed with K_{AB}:

4. Alice to Bob: {"Here is my message:" ..... $T\}^{KAB}$

Bob should be able to unseal 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.

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

Alice to 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 sends, for example, Charles a claim that he is Alice and a proposal to use K_{AB} for sealing messages.

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

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

2. CA to Bob: {"Here are the certificates:"

$\{Bob, B_{pub}, T\}_{CApriv}$,

$\{Charles, C_{pub}, T\}_{CApriv}\}$

3. Bob to Charles: {"Here is my certificate and a proposed key":

$\{Alice, A_{pub}, T\}_{CApriv,}$

$\{K_{AB}, T\}_{Apriv}\}^{Cpub}$

Message 3 is a carefully crafted message by Bob: 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}.

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:

> *4.* Bob to 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, and will respond. Designing cryptographic 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 attacker (Bob) is trusted for something, because Alice first has to have a conversation with Bob before Bob can spoof 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 appropriateness. An attacker can replace the string "Here is my certificate and a proposed key" with any other string (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., $\{Alice, Bob, K_{AB}, T\}_{Apriv}$).

Making Alice and Bob explicit in the proposal for the key addresses the lack of explicitness, but doesn't address the lack of appropriateness. 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 sealing and signing was widely understood. It used sealing 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 sealing is now considered to be a fundamentally bad practice.

4. 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*", 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; between 1930 and 1975 only three papers with significant technical contributions regarding cryptography were published; now there are multiple conferences on cryptography per 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 $n = g^k \pmod p$.

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*-2}. Alice sends to Bob: $g^a \pmod p$, and Bob sends to Alice: $g^b \pmod p$.

On receiving these messages, Alice computes $g^{ab} = (g^b)^a \pmod p$, and Bob computes $g^{ba} = (g^a)^b \pmod p$. Since $g^{ab} = g^{ba} = k$, Alice and Bob now have a shared-secret key *k*. An attacker hearing the messages exchanged between Alice and Bob cannot compute that value, because the attacker doesn't know *a* and *b*; he 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 $k = g^{ab} \pmod p$ given the two public values $g^a \pmod p$ and $g^b \pmod p$. 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 man-in-the-middle attack, similar to the one in the previous section. 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 original Diffie-Hellman protocol.

5. A key distribution protocol using a public-key system

The second protocol uses a Diffie-Hellman-like exchange to set up keys for sealing 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; a similar protocol is also used in the Taos distributed operating system.

A client connects to a service whose public key is S_{pub}, as follows:

1. Client to service: {"Here is a temporary public key T_{pub} and two key halves sealed with your public key:" $\{K_{C1}, K_{C2}\}^{Spub}$}

The client proposes a temporary public key and two key halves. The client picks a short-lived temporary key to ensure forward secrecy. The client discards this temporary key and generates a new public key at regular intervals (e.g., every hour).

Since the key halves are sealed with the service's public key, only the holder of S_{priv} can unseal this message. Therefore, if the service has kept S_{priv} private, then only the service will learn K_{C1} and K_{C2}.

The service responds as follows:

2. Service to client: {"Here are two key halves sealed with your temporary public key:" $\{K_{S1}, K_{S2}\}^{Tpub}$}

The service also proposes two key halves. The ones proposed by the service are sealed with the temporary public key proposed by the client in message 1. Anyone can know $T_{pub,}$ but only the client, who proposed T_{pub} can unseal these key halves, since only it knows T_{priv}.

Using the key halves the client and the service compute two keys that are to be used to seal and authenticate data sent between them for the next hour:

$$K_{CS} = SHA\text{-}1\ (\text{"KCS"}, S_{pub}, K_{S1}, T_{pub}, K_{C1})$$

$$K_{SC} = SHA\text{-}1\ (\text{"KSC"}, S_{pub}, K_{S2}, T_{pub}, K_{C2})$$

SHA-1 is a cryptographic hash function that outputs a 160-bit number; thus, only the parties who know the input should be able to compute K_{CS} and K_{SC}. *"KCS"* and *"KSC"* are strings that name the communication direction for which the keys should be used. Since K_{CS} is dependent on K_{C1} and K_{S1}, only the client and the service can compute K_{CS}; an attacker might be able to obtain S_{pub} and T_{pub}, but not K_{C1} and K_{S1}. Similarly, since K_{SC} is dependent on K_{S2} and K_{C2}, only the client and the service can compute K_{SC}.

K_{CS} and K_{SC} are used to derive keys for shared-secret sealing and authentication. The keys derived from K_{CS} are used for sealing and MACing data sent from the client to the service. The keys derived from K_{SC} are used for sealing and MACing data sent from the service to the client. For example, if the client sends the message to the service, it seals and MACs the content with K_{CS}:

3. Client to service: {"Please, give me the file x.c"}$^{KCS\text{-}seal}_{KCS\text{-}mac}$

This protocol assures that no one knows K_{CS} and K_{SC} without also possessing S_{priv}. Thus, it gives the client a secure channel to the service that holds the private key corresponding to S_{pub}. The service, in contrast, knows nothing about the client; this allows the client to anonymously visit the service and read, for example, public information.

K_{CS} and K_{SC} are constructed from a key half provided by the client and a key half provided by the service. This construction is a good one, because even if one of the client and service uses a bad method for generating its keys, the shared-secret key for the session will be still difficult to guess. By including T_{pub} the client assures forward secrecy; even if an attacker manages to obtain K_{CS} and $K_{SC,}$ he will be able to unseal the conversation for only a short period of time.

If the service desires to establish the principal associated with T_{pub} (e.g., to perform access control), an additional protocol is needed. We also assumed that the client knew that S_{pub} belongs to the right service; both Taos and SFS have additional machinery for distributing such public keys securely.

The chief advantage of this protocol is its simplicity. It requires only two messages and therefore can be easily understood. So far, no one has found a flaw with this protocol.

6. Example: the secure socket layer (SSL) protocol

The Secure Socket Layer is a cryptographic protocol to establish a confidential and authenticated communication channel over the Internet. It allows client/service applications to communicate in the face of eavesdroppers and attackers who tamper with and forge messages. In the handshake phase, the SSL protocol negotiates, using public-key cryptography, shared-secret keys for message authentication and confidentiality. After the handshake, messages are sealed and MACed using the shared-secret keys.

The SSL handshake protocol is awkward in at least one aspect: the initial messages of the negotiation protocol are not authenticated and not sealed, as can be seen in figure 11-13. When the client hasn't connected to the service before, the first 6 messages and message 8 are sent in unauthenticated plaintext.

This lack of authentication introduces a number of potential vulnerabilities. Attackers can change an initial message without the receiver being able to detect changes immediately. To address this concern, SSL computes at the end of the handshake protocol a MAC over all messages sent in the handshake. Both the client and service check that MAC before proceeding with sending sealed and authenticated data.

A better solution for SSL would have been to generate first a temporary key to authenticate all handshake messages. The designers of SSL didn't choose this solution, because they wanted to support a wide range of different algorithms for authentication instead of choosing one specific one for the duration of the exchange protocol. As we will see when we study the protocol in more detail, this choice makes it more difficult to reason about the correctness of the protocol. In fact, we will see that the designers made some mistakes in the earlier versions of SSL.

Let's dive into the details of the protocol. The CLIENTHELLO message announces to the service the version of the protocol that the client is running, a random sequence number, and a prioritized set of ciphers 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, a random number, a session identifier, and the ciphers selected from the ones offered by the client.

Some of the next messages in the protocol are optional. We show the protocol messages for the case for which SSL was designed, namely where an anonymous user is browsing a web

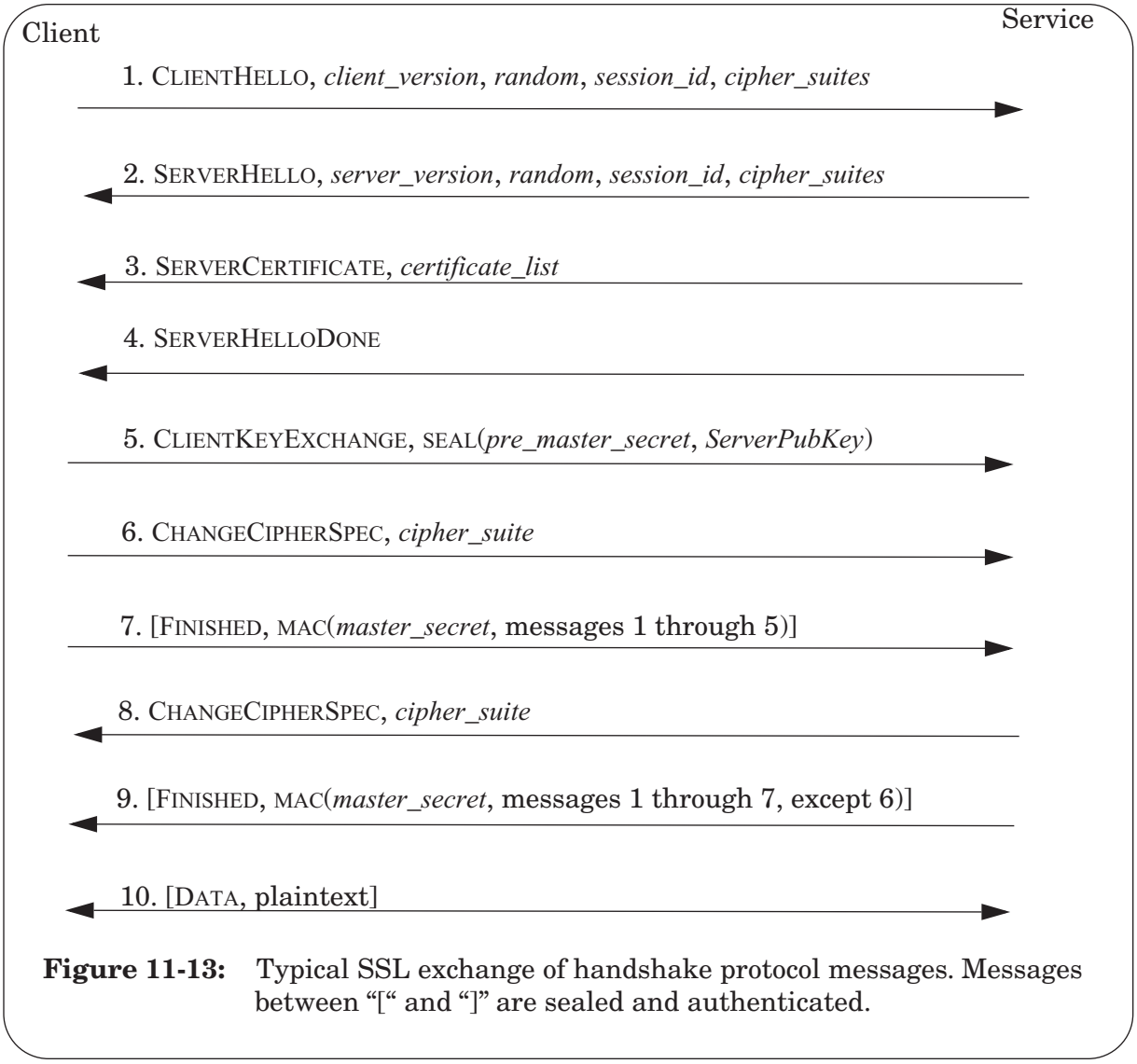


Figure 11-13: Typical SSL exchange of handshake protocol messages. Messages between "[" and "]" are sealed and authenticated.

site and desires service authentication. 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.

As we will discuss in section F in detail, service authentication in SSL relies on trusted certificate authorities. The owner of a service purchases a certificate from a certificate authority in advance. The certificate binds the identify and the key of the service owner together; the certificate contains in essence the public key of the service signed with the private key from certificate authority. The certificate authority hands out such certificates after certifying that the owner of the service is indeed the principal it claims to be. A client will have a list of authorities it trusts, and will use their public keys to verify that a certificate is valid. Thus, if a service presents certificates signed by an authority that the client doesn't trust, service authentication will fail and the protocol will terminate. However, if the client

trusts the authority that signed one of the certificates presented by the service, then the client will be believe the identity of the service and the protocol will proceed.

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 *pre_master_secret*. SSL 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 SSL typically uses a public-key system, in which the client seals the secret with the public key of the service found in the certificate, and sends the result to the service in the CLIENTKEYEXCHANGE message. Only the service with the correct private key will be able to unseal the *pre_master_secret*. To guarantee forward secrecy the service public key needs to be a temporary key that the service periodically updates.

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

master_secret =
MD5(pre_master_secret +
SHA('A' + *pre_master_secret* + *ClientHello.random* +
ServerHello.random)) +
MD5(*pre_master_secret* +
SHA('BB' + *pre_master_secret* + *ClientHello.random* +
ServerHello.random)) +
MD5(pre_master_secret +
SHA('CCC' + *pre_master_secret* + *ClientHello.random* +
ServerHello.random));

Both the MD5 and SHA cryptographic hashes, are used in the computation. The hope is that if one of them gets compromised, the other one won't and therefore an attacker cannot recover the *pre_master_secret* value.

It is important that the *master_secret* is 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 was omitted from the protocol, an attacker 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 the *pre_master_secret* is sufficient to derive all keys.

After sending the sealed *pre_master_secret*, the client sends a CHANGECIPHERSPEC message. This message specifies that from now on the client will use the ciphers specified as the sealing and authentication ciphers for the record layer protocol.

The keys for message sealing and authentication ciphers are computed using the *master_secret*, *ClientHello.random,* and *ServerHello.random* (which both the client and the service

have). Using this information a key block is computed:

```
key_block =
        MD5(master_secret +
                SHA('A' + master_secret + ServerHello.random +
                        ClientHello.random)) +
        MD5(master_secret +
                SHA('BB' + master_secret + ServerHello.random +
                        ClientHello.random)) +
        MD5(master_secret +
                SHA('CCC' + master_secret + ServerHello.random +
                        ClientHello.random))
        + [...];
```

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 appendix 11–A). 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 is sent over the record layer like all previous messages, but now a cipher suite has been specified so it will be sealed and authenticated.

The finish message contains two hashes computed with different hash algorithms: MD5 and SHA. Both algorithms compute the hash over the same data, namely:

```
HASH(master_secret + pad2 +
        HASH(handshake_messsages + Sender + master_secret + pad1))
```

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 *Sender* value is 0x434C4E54, if the sender is the client, and 0x53525652, if the sender is the service. If the service verifies the hash, the service and client agree on the protocol sequence and the *master_secret*. Since this message is crucial, two different hash algorithms are used.

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 sealed 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. After both finish messages have been received and checked out correctly, the client and service can communicate data over a sealed and authenticated channel.

The SSL handshake protocol is more complicated than some of the protocols that we described earlier. In a large part, this complexity is due to all the options SSL 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 SSL protocol needs a number of additional protocol messages. This makes reasoning about SSL difficult, since depending on the client and service constraints, the protocol has a different set of message exchanges, different ciphers, and different key sizes.

In fact, because of these features, an earlier version of the SSL protocol was vulnerable to new attacks, such as cipher suite substitution and version rollback attacks. In version of 2 of SSL, the attacker could edit the CLIENTHELLO message undetected, convincing the service to use a weak cipher, for example one that is vulnerable to brute-force attacks. Version 3 protects 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 attacker convinces the service to use version of 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 active attacker 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 sealed and authenticated, this can open a serious security hole. Wagner and Schneier describe an attractive attack that exploits this observation. Currently, widely-used implementations of SSL 3.0 protect against this attack by accepting a FINISHED message only after receiving a CHANGECIPHER message. The best solution, of course, would be to include the CHANGECIPHER message in the MAC of the FINISHED message, but that would require a change in the specification of the protocol.

F. Advanced authentication

The protection model has three key steps that are executed by the guard on each request: authenticating the user, verifying the integrity of a message, and determining if the user is authorized. Authenticating the user is typically the most difficult of the three steps, because typically 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 follow back through a chain of messages that typically terminates in a message that was communicated by physical rendezvous. That physical rendezvous securely binds the name 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 cryptographic protocols in the previous section. If this is the case, then typically one principal speaks for another principal, and the guard must follow a chain of principals to establish that a message came from particular real-world user.

Consider a simple cryptographic protocol, where a certificate authority signs certificates, associating authentication keys with names (e.g., key *K* belong to user "X"). If a service receives this certificate and later receives a message that it verifies with the key *K*, then we have the following chain of reasoning:

1. The guard knows that a message came from a principal who has a private authentication key *K*.

2. The guard has received a message from the certification authority telling the guard that the authentication key *K* 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 it has obtained the public authentication key of the authority through some chain of other messages that terminated in physical rendezvous.)

3. If the guard is willing to believe that the certification authority speaks for user "X", then the guard might believe that key *K* speaks for user "X," and that therefore the origin of the first message is user "X."

The guard might believe the assumption in step 3 (i.e., that certificate authority speaks for user "X"), because it believes that certificate authority has carefully authenticated user "X", when the real-world person "X" asked for a private authentication key.

In this section, we will formalize this type of reasoning using an 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,

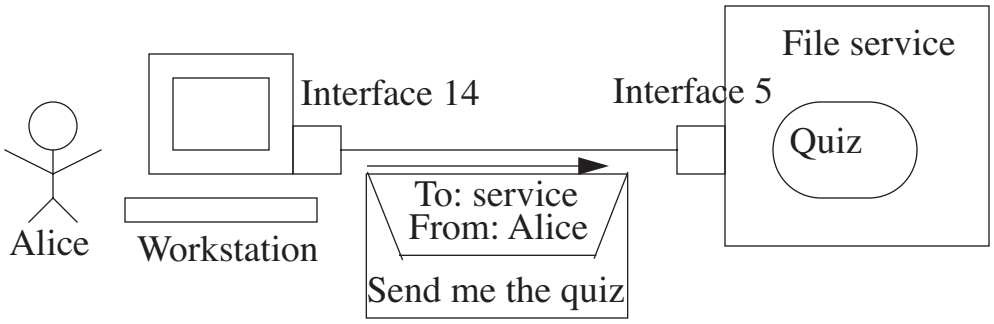


Figure 11-14: Authentication example

and, if the assumptions are acceptable, the guard can determine if the person is authorized.

1. Reasoning about authentication protocols

From the examples in the section about cryptographic protocols it is clear that reasoning about authentication protocols is difficult, partly because we are trying to achieve a negative goal. Burrows-Abadi-Needham (BAN) authentication logic is a theory that can help us to reason about authentication problems. 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" by Lampson et al., ACM Transactions on Computer Systems, Vol. 10, No 4, Nov. 1992, pages 265-310.

Consider the following example. Alice types at his workstation "Send me the quiz" (see figure 11-14). His workstation J sends a message over the wire from network interface 14 to network interface 5, which is connecting the file service machine, F, and 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. 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 other assumptions to prove 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 him. That is, the file system 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 so.

- 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-word situations better, it is possible to use more refined rules. For example, Alice could delegate authority to B on only specific topics. Similarly, the rules can be extended to capture freshness by introducing a new operator **said** instead of **says**. However, as we will see in the rest of this chapter, even these three simple rules help flush out fuzzy thinking.

1.1. Hard-wired approach

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

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

In short, we assume that the hardware and Alice's workstation are part of the TCB. If we assume this, then 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.

1.2. Internet approach

Now, suppose we attach the file service's network interface 5 to the Internet. Then, we get:

- network interface F.5 **speaks for** the Internet: we assume no one rewired it.
- the Internet **speaks for** network interface J.14: if we 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 ($\{M\}_k$ denotes a plaintext message signed with a key k):

Alice to 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 add to our list of assumptions:

- the file service is not trying to trick itself;
- the exchange of the shared-secret key was secure;
- Alice has not revealed his key.

Now we really do believe that "key T **speaks for** Alice", and we are home free. This

reasoning is not a proof, but it is a method that forces us to 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 the fact that Alice said it recently. Someone else might be replaying the message. Extensions to the basic logic deal with freshness by introducing new rules for freshness that relate **says** and **said**.

If the file service received key T in a signed message {Alice, file service, T}$_{KFS}$ from a central key distribution center (KDC), such as Kerberos, then we have:

- K_{FS} **says** (KDC **says** (T **speaks for** Alice)).

If we trust key K_{FS}, we can deduce that:

- KDC **says** (T **speaks for** Alice).

So if we trust KDC, we see that indeed, "T **speaks for** Alice", and we're home. Again, this line of reasoning is not a proof, but it forces us to state our assumptions clearly (e.g., trust in key K_{FS} and the KDC). If we wanted to analyze this deeper, we'd need to specify KDC's protocols more precisely; we stop here, though. The general picture should be clear.

2. Authentication in distributed systems

All of the authentication examples we have discussed so far have involved one service. Using the techniques from section C, it is easy to see how we can build a single-service 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 Web is an example of this approach. The browser sets up a secure channel using the secure socket layer protocol. Then, the browser asks the user for his password and sends this password over the secure channel to the service. If the service identifies the user successfully with the received password, it service returns a token (a cookie in Web terminology), which is stored by the browser. 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 in cleartext. 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 shortcoming are not a serious inconvenience. However, in an environment (say a large company, a university, or an Internet store with multiple departments) where there are many services and where information needs to be shared between services, a better plan is needed.

In such an environment we desire:

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.

Few system designs or implementations meet these requirements. The system that comes close is Kerberos, which has been adopted by Microsoft for doing distributed authentication.

In Kerberos, each user and service belongs to a *realm*, an organizational unit that is under the same administrative authority. For example, a university like MIT might be a single realm. Each each user's name is unique within a realm. Per realm, there is one authentication service. When a user intends to use any other service in the realm, the user first authenticates himself with a password to the authentication service, using the Kerberos cryptographic protocol. The authentication service returns a *ticket* identifying the user. The ticket is a kind of a certificate; it binds the user name to a key and some other information to protect the ticket from being falsified.

Now the user can access any other service in the realm without having to type his password again. A request to another service includes the ticket identifying the user. When the service receives a request, it authenticates the ticket using the information in the ticket. If the ticket is authentic, the service authorizes the request based on the user's identity in the ticket. Otherwise, it denies the access.

Services authenticate users from other realms in a similar way to users from their realm. Suppose a user Jones at MIT puts a user at Carnegie Mellon University (CMU) on his access control list: he adds the principal "smith@cmu.edu" to his ACL. Now Smith wants to access Jones' files. To make this possible in Kerberos, the system administrators of the MIT and CMU realms have to agree to set up a secure channel between their two authentication services.

When a user wants to access a service in another realm, he contacts his local authentication service to find out what the remote authentication service is and to get a ticket to identify himself to the remote authentication service. Then, the user asks the remote authentication service for a ticket for the remote service; the remote service verifies the ticket issued by the local authentication service, and, if valid, issues to the user a ticket for the remote service. Then, the user contacts the remote service and presents the ticket issued by the remote authentication service. The remote service verifies the ticket, and based on the information in the ticket decides whether to grant the service or not.

Thus, when a service S at MIT makes an authorization decision based on a ticket T issued to a remote user at CMU the following information contributes to the actual decision:

1. The password that user presented to authenticate himself to his local authentication service;

2. The authentication process that the remote authentication service performed to establish if the local authentication service is authentic;

3. The information that the remote authentication service used to generate ticket *T* for service *S*;

4. The user identity on which service *S* makes the authorization decision;

5. The information stored on the access control list at service *S*.

We can capture this chain of reasoning using the authentication logic. Informally, for service *S* at MIT to make the decision whether Smith from CMU should have access or not, he has to establish if the ticket *T* **speaks for** *smith@cmu.edu*. To make this decision, service *S* has to convince itself that the *T* **speaks for** MIT's authentication service, that MIT's authentication service **speaks for** CMU's authentication service, that CMU's authentication service **speaks for** the principal *Smith* at CMU, and that the principal *Smith* **speaks for** the actual user *Smith*. All the pieces of information listed above contribute to the decision that *T* **speaks for** the user Smith at CMU.

In this scheme, the two realms trust each other. MIT has delegated authority to CMU to authenticate users at CMU, and vice versa. Because they delegate authority, they should be careful how they assign tickets to their users. A Smith at CMU is likely to be someone different from Smith at MIT, but the scheme should still allow for a Smith that has accounts both at CMU and MIT. The owner of a file must learn through a name discovery protocol what principal identifier to use for a particular person.

3. Authentication across administrative realms

The realm approach as described above doesn't scale well to many realms that are managed differently by different administrative authorities. For example, a student might have a service in his dorm that is not under the administrative authority of the university; yet the student might want to access his service from computers on the main campus, administered by central campus authority. Furthermore, the student might want to provide access to his service to family and friends in yet other administrative realms. It is unlikely that the campus administration will delegate authority to the dorms, and will 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? In the example above, the system administrators authenticate CMU and MIT by setting up a secure channel between their authentication services. If a user accesses a service in a realm that he just learned about, he cannot ask the system administrator to authenticate that service manually. We need a more automatic solution.

2. How can we name users securely? We could use email addresses, such as smith@cmu.edu, to identify principals securely, but the Domain Name System doesn't provide security.

3. How do we manage many users? If MIT is willing to share course software with all students at CMU, MIT shouldn't have to list every CMU student individually on the access control list for the files. Clearly, protection groups are needed. But, how does a student at CMU prove to MIT's service that he is part of the group students@CMU?

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 (services, users, etc.) by *public keys* and (2) distribute public keys securely. 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 MIT might be named by a public key K_{Apub} and a user Bob at CMU is named by a public K_{Bpub}; from the public key we cannot tell whether the Alice is at MIT or CMU. 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 access Alice's files, Bob sends a request to Alice's service including his public key K_{Bpub}, and then Alice's service challenges Bob to prove that he has the private key corresponding to K_{Bpub}. If Bob can prove that he is Bob (e.g., for example by signing a challenge that Alice verifies with Bob's public key K_{Bpub}), then Alice's service authorizes access.

We can name protection groups also by a public key. Suppose that $K_{CMUStudentspub}$ is a public key representing CMU students. If Alice wanted to grant all students at CMU access to her files, she could add $K_{CMUStudentspub}$ to her access control list. Then, if Charles, a student at CMU, wanted to have access to Alice her file, he would have to present a proof that he is a member of that group, for example, by providing a statement signed by $K_{CMUStudentsPriv}$ to Alice saying "$K_{Charlespub}$ is a member of the group $K_{CMUStudentspub}$" (signed by $K_{CMUStudentspriv}$). Alice can verify this statement using $K_{CMUStudentspub}$, which is on her access control list. After Alice successfully verifies the statement, then she can challenge Charles to prove that he is the holder of the private key $K_{Charlespub}$. Once Charles can prove he is the holder of that private key, then Alice can grant access to Charles. For Alice to believe this chain of reasoning she must trust the holder of $K_{CMUStudentpriv}$ to be a responsible person who carefully verifies that Charles is a student at CMU. If she trusts the holder of that key to do so, then Alice doesn't have to maintain our own list of who is a student at CMU; in fact, she doesn't need to know at all which particular principals are students at CMU.

If services are 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, users don't want to remember or type public keys. When Alice adds K_{Bobpub} and $K_{CMUStudentpub}$ 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 authenticating public keys.

4. Authenticating public keys

How do we authenticate that K_{Bpub} is Bob's public key? As we have seen before, that is based on a key-distribution protocol, which terminates with a name discovery step. If Bob and Alice met face-to-face and Alice hands Bob a piece of paper with her public key, Bob can have reasonable confidence that the key is Alice's. However, if Bob receives a message electronically that says "Hi, I am Alice and here is my public key", Bob cannot be sure the message has not been tampered with. (Alice could authenticate the message using a digital signature, but Bob does not know her public key yet, so no luck there.)

To make clear what the assumptions are in a key-distribution protocol, we capture the whole process using the authentication logic. Consider the following example where Alice receives a message from Chris, 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 Chris.

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

Chris **says** ($K_{pubChris}$ **speaks for** Chris) (belief #1)

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

$K_{pubChris}$ **speaks for** Chris (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, Chris prepares a message, M_1:

Chris **says** M_1

signs it with her private key:

$\{M_1\}_{KprivChris}$

which, in authentication logic, can be described as

$K_{privChris}$ **says** (Chris **says** M_1)

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

Chris **says** M_1 (?)

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:

$$\text{VERIFY}(\ \{M_1\}_{\text{KprivChris}},\ K_{\text{pubChris}})$$

If VERIFY returns TRUE then one might think that Alice is entitled to believe:

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

but that belief actually requires a leap of faith: that the crypto 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

$$(K_{\text{privChris}}\ \textbf{speaks for}\ K_{\text{pubChris}}) \qquad \text{(belief \#4?)}$$

which sounds plausible, but for her to accept that belief requires another leap of faith: that Chris is the only person who knows $K_{\text{privChris}}$. Alice decides that Chris is probably careful enough to be trusted to keep her 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

$$K_{\text{privChris}}\ \textbf{speaks for}\ \text{Chris} \qquad \text{(belief \#5)}$$

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

$$\text{Chris}\ \textbf{says}\ M_1 \qquad \text{(belief \#6)}$$

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

- K_{pubChris} is a true copy of Chris's public key.
- The crypto system used for signing is computationally secure.
- Chris has kept $K_{\text{privChris}}$ secret.

5. Authenticating certificates

Suppose now that Mary, whom Alice does not know, sends Alice the message

$$\{M_2\}_{\text{KprivMary}}$$

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

$$\text{VERIFY}(\ M_2,\ \text{SIGN}(M_2,\ K_{\text{privMary}}),\ K_{\text{pubMary}})$$

returns TRUE, all she believes (again assuming that the crypto system is secure) is that

$$K_{privMary} \textbf{ says } (\text{Mary } \textbf{says } M_2)$$

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

$$K_{privMary} \textbf{ speaks for } \text{Mary} \qquad (?)$$

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

$$\{\text{“Mary's public key is } K_{pubMary}\text{”}\}_{KprivTrustUs}$$

and posted this certificate on her 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 Chris:

$$K_{privTrustUs} \textbf{ says } M_3$$

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

$$\text{TrustUs } \textbf{says } M_3$$

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

- $K_{pubTrustUs}$ is a true copy of the TrustUs.com public key.
- The crypto 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 “Mary's public key is $K_{pubMary}$”. Since TrustUs.com is taking Mary's word on this, that statement can be expressed in authentication logic as

$$\text{Mary } \textbf{says } (K_{pubMary} \textbf{ speaks for } \text{Mary})$$

Combining, we have:

$$\text{TrustUs } \textbf{says } (\text{Mary } \textbf{says } (K_{pubMary} \textbf{ speaks for } \text{Mary}))$$

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

$$\text{TrustUs } \textbf{speaks for } \text{Mary} \qquad (?)$$

then she could apply the use of delegated authority rule to conclude that

$$\text{Mary } \textbf{says } (K_{\text{pubMary}} \textbf{ speaks for } \text{Mary})$$

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

(1) TrustUs.com has been authorized by Mary to create certificates for her. Alice might think that finding the certificate on Mary's Web site gives her some assurance on this point, but Alice has no way to verify that Mary'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 Mary presented for identification. If TrustUs.com was not careful, it might, without realizing it, be speaking for Lucifer rather than Mary. (Unfortunately, certificate authorities have been known to make exactly that mistake.) And, of course, TrustUs.com is assuming that the credentials Mary presented were legitimate; it is possible that Mary 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 Mary's original message M_2 as genuine:

- $K_{\text{pubTrustUs}}$ is a true copy of the TrustUs.com public key.
- The crypto system used for signing is computationally secure.
- TrustUs.com has kept $K_{\text{privTrustUs}}$ secret.
- TrustUs.com has been authorized by Mary.
- TrustUs.com carefully checked Mary's credentials.
- Mary has kept K_{privMary} secret.

and she notices that in addition to relying heavily on the trustworthiness of TrustUs.com, she doesn't know Mary, 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 Mary's public key was not obtained by a physical rendezvous, she knows only that the message came from someone named "Mary"; 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. A more powerful authentication logic would, for example, have rules that distinguish between **says** and **said**, and raise an alert that neither the certificate nor the message was time stamped.

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 and authorize Mary's request. Given this observation, this immediately raises the question of how Mary revokes the certificate that she bought from TrustUs.com. If Mary, for example, accidently discloses her private key, the certificate from TrustUS.com becomes worthless and she should revoke it so that Alice cannot be tricked in believing that M_2 came from Mary. One

way to address this 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 a disadvantage that if Mary loses her 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.

6. Certificate chains

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. There are two ways of organizing such chains; we discuss them in turn.

6.1. Hierarchy of central certificate authorities

In the central-authority approach, key certificate authorities record public keys and are managed by central authorities. To make this approach scalable, the CAs are arranged in a hierarchy.

For example, Bob asks his local MIT CA for Alice's key. If the MIT CA does not know Alice's public key, it asks the USA CA, which might refer the request to the Berkeley CA, if the Berkeley CA knows about Alice. If the USA CA doesn't know about Alice, it asks the United Nations CA, and so forth. MIT will return Alice's public key to Bob with the chain of authorities that were involved in certifying it.

One problem with this approach is that you must trust a central authority. You may be perfectly happy if you must trust MIT's CA, but trusting Libya's CA (or the CIA's CA!) may be less attractive. The other problem is that you cannot get a certificate without the permission of a widely-trusted, credible authority. The certificate authority may ask an unreasonable price for the service (either in cost or in a requirement that you disclose unrelated personal information such as your salary or the number of guns you own).

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 Chris, Dawson, and Ella and posts these on her Web page: {Alice, $K_{Apub}\}_{KCpriv}$, {Alice, $K_{Apub}\}_{KDpriv}$, {Alice, $K_{Apub}\}_{KEpriv}$. If Bob knows the public key of any one of Chris, Dawson, 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 Chris, Dawson, or Ella, he might know someone (say Felipe) who knows one of them. 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. He can inspect the chain of trust by which he 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.

This scheme relies on the observation that it 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.) And with luck, there will be many chains going from Bob to 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) and a number of other systems use the web of trust approach.

7. Example: service authentication with SSL

In section E we studied the SSL protocol for setting up a confidential and authenticated communication channel. Here we study how an SSL client obtains the public key to authenticate the service.

SSL can be used for many client/service applications, but its main use is for secure Web transactions. In this case, a Web browser uses SSL to set up a message-authenticated, confidential connection with a Web service. HTTP requests and responses are sent over this secure connection. Since users typically visit Web sites anonymously 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 intruder who can now record private information (e.g., credit card numbers) and supply fake information. Therefore, a key problem SSL addresses is service authentication. As we will see, SSL relies on certificates and certificate authorities.

The main challenge for a client is to convince itself that the service's public key is authentic. If a user anonymously visits a Web site, say amazon.com (an on-line book retailer), then a user wants to make sure that the Web site he connects to is indeed owned by amazon.com. The basic idea is for Amazon to sign its host 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 attacker can convince the client that she has the public key of Amazon, but substitute her own public key and sign Amazon's name with the attacker's private key. This problem is an instance of the key-distribution problem, discussed in section E.

SSL relies on well-known certification authorities for key distribution. An owner of a Web site buys a certificate at one or more certification authorities. Each authority runs a certification check to validate that the Web site is the one it claims to be. For example, a certification authority might check if "amazon.com" belongs to the Amazon.com company and might ask Amazon for articles of incorporation to prove that it is a legal company. After the certification authority has verified the identity of company, it issues a certificate. The certificate contains the public key of the service and the DNS name of the service (along other distinguishing information), signed with the private key of the certificate authority. (The

service sends the certificates in Step 3 of the handshake protocol, described in section E.)

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 trusts the identify of the service.

SSL uses certificates that are standardized by the X.509 standard. Version 3 of X.509 certificates includes among other fields (the standard specifies them in a different order):

```
struct {
        version;
        serial_number;
        signature_cipher_identifier;
        issuer_signature;
        issuer_name;
        subject_name;
        subject_public_key_cipher_identifier;
        subject_public_key;
        validity_period;
}
```

The *version* field specifies the version of the certificate (3 in our example). The *serial_number* field contains a unique number for a certificate, which is assigned by the issuing certification authority. The *signature_cipher_identifier* field identifies the algorithm used by the authority to sign this certificate. This allows a client of the certification authority to know which algorithm to use to verify the *issuer_signature* field. The *issuer_name* field specifies the name of the issuer; for instance, the DNS name for Verisign. The next three fields specify the name for the subject (e.g., the DNS name of amazon.com), the public-key cipher it wants to use (say RSA), and its 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 attacker compromises the private key of amazon.com, then the attacker can impersonate amazon for the next 12 months. To counter this problem, issuers keep their private keys off-line, stored in a safe location. (The issuer needs to keep its private key corresponding to its service certificate on-line to participate in the SSL protocol.) In addition, a certification authority maintains a certification revocation list, which contains compromised certificates. Anyone can download the certificate revocation list to check if a certificate is on this blacklist. Unfortunately, revocation lists are not in wide spread use today. Good certificate revocation procedures are an open research problem.

The crucial security step for establishing a subject's identity is the certification procedure executed by the certification authority. If the authority issues certificates without checking out the identity of the Web site carefully, the certificate doesn't improve security. In that case, an attacker could ask the certification authority to create a certificate for amazon.com. If the authority doesn't check the identity of the attacker, the attacker will obtain a certificate for amazon.com, allowing her to impersonate Amazon. Thus, it is

important that the certification authority do a careful job of certifying the subject's identity. A typical certification procedure includes paying money to the authority, sending by surface mail the articles of incorporation (or equivalent) of the company. 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 Web sites are able or willing to obtain a certificate. If it is expensive in time and money to obtain a certificate, Web sites might opt to go for an insecure solution (i.e., not authenticating their identity with SSL). 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 Web sites. Students who want to obtain a certificate from Verisign for their personal Web sites will find Verisign's certification procedure impractical.

G. Summary

This chapter provided first a general perspective on how to think about building secure systems, followed by a set of design principles, and then a hundred or more pages giving 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. Appendix 11–B relates several war stories of protection 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 appendix 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, and 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 experts just building too complex systems. Some problems are due to 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 are due to 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 A, when the Internet was designed processors weren't efficient enough for applying cryptographic transformations in software, the deployment of cryptographic-transformation processors 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, and key distribution was understood, the insecure protocols were so widespread that it was too hard to do a retrofit.

- Personal computer systems do not come with enforced modularity that creates strong internal firewalls. The main reasons are keeping the cost low and stupidity. 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, the early designs to add enforced modularity were incorrect. The designers just got it wrong, a couple of times.

- 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 security was just not a requirement.

- Unix systems, commonly used as services, have enforced modularity, but many UNIX services are vulnerable to buffer-overrun attacks, which subvert modular boundaries. Why are these buffer overruns so difficult to eradicate? A reason is the design of the C programming language, which doesn't check array bounds, and its success. Most 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 it breaks most existing C programs. As a result, each service program must be fixed individually. Conceptually simple changes to the hardware could also address the problem, but for similar compatibility reasons that is also difficult to pull off.

- Why isn't software verified for security? First, we don't know how to specify programs precise enough so that automatic tools can be applied successfully. As a result, automatic verification doesn't seem to work for any programs larger than a few hundred lines of code. In the research community, some recent progress has been made in analyzing cryptographic algorithms formally, checking software for common security problems, and verifying the correctness of certain network protocols, but at this time no checker is able to verify the absence of security problems in a complete system.

- Why doesn't security certification help more? There is no adequate regulation for what kind of attacks a minimal secure system should protect against. And, what standards exist for security requirements are out of date, because they don't cover network security. Governments have a difficult time keeping up with the changes in technology.

- Many secure systems depend on a public key infrastructure. Why doesn't 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 economical 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 above), end-to-end security is difficult to achieve. If firewalls are properly deployed they can keep the external, low-budget attackers away from the vulnerable internal computers. Of course, firewalls don't help against inside attackers, and attackers that find ways around the firewall to reach the inside network from the outside (e.g., by accessing the internal wireless network from outside, breaking into the modem bank, etc.).

- We are hearing reports that wireless network (WiFi or 802.11b) security is awful. This is a brand-new design. Why is it so vulnerable? One reason appears to be that the design was done by a committee, which was expensive to join, and that only committee members were allowed to review the design. As a result, the design was closed effectively, and few experts were able to review the design until it was deployed, at which point a number of security holes were identified.

- Cable TV scrambling systems, DSS (Satellite TV) security, the CSS system for protecting DVD movie content, the 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 look at the design. When the system was deployed, experts investigated the design and found the problems immediately.

A final reason is that no one has a recipe for building secure systems, partially 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. The examples in the appendix illustrate these points further.

Acknowledgements

Ron Rivest, David Mazières, and Robert Morris made crucial contributions to material presented in this chapter. Brad Chen, Michael Ernst, Kevin Fu, Charles Leiserson, and Seth Teller made numerous suggestions for improving the text.

Appendix 11–A. Cryptography as a building block

In the protection model, a client module sends a message to a service module. An attacker can copy, delete, modify, and insert messages. To protect against these attacks, we want to transform the message in such a way that an attacker cannot figure out what the content of the message is and cannot modify the message undetected. This appendix sketches how primitives such as SIGN, VERIFY, SEAL, UNSEAL, and cryptographic hashes can be implemented using cryptographic transformations (also called *ciphers*). Readers who wish to understand the underlying cryptography in depth should consult books such as "Applied cryptography" by Bruce Schneier, or "Handbook of applied cryptography" by Menezes, van Oorschot, and Vanstore.

A model for understanding cryptographic transformations involves two secure areas separated by an insecure path (see figure 11–15). With an open design approach, we can write the transformation process as:

$$C = T(M, K_1),$$

$$M' = U(C, K_2),$$

and the end-to-end process is $M' = U(T(M, K_1), K_2)$.

Depending on the constraints on their inputs, transformation schemes are either stream ciphers or block ciphers. In a *stream cipher*, the conversion from plaintext to ciphertext is performed one bit at a time, and the input can be of any length. In a *block cipher*, the transformation from plaintext to ciphertext is performed on fixed-size blocks. If the input is shorter than a block, it must be padded to make it a full block in length. If the input is longer than a block, it is broken into blocks, if necessary, the last block is padded, and then the individual blocks are transformed.

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 transmitting 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

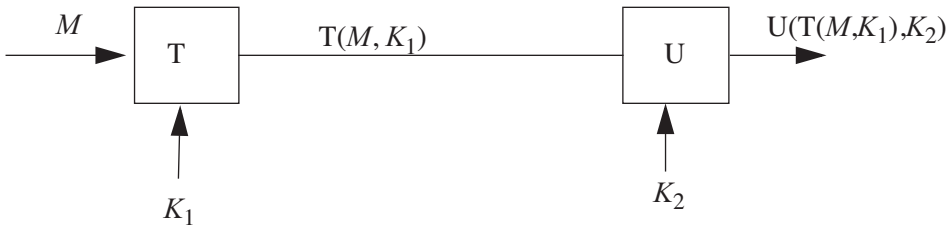


Figure 11–15: Cryptographic transformations

receiver. Using the previously communicated key string, the receiver can untransform the ciphertext by computing the XOR of the ciphertext and key string.

To be more precise, this transforming scheme is a stream cipher, in which a sequence of message (plaintext) bits $m_1, m_2,\ldots, m_n$ is transformed using an equal-length sequence of secret key bits $k_1, k_2, \ldots, k_n$ 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 (mod-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 above, since the ciphertext does not give the attacker 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. Thus, patterns in the plaintext won't show up as patterns in the ciphertext. Thus, knowing the value of any number of bits in the ciphertext doesn't allow the attacker to guess the bits of the plaintext or other bits in the ciphertext. To the attacker 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 attacker), 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 attacker can add, flip, or replace bits in the stream without the recipient realizing it. The attacker may have no way to know exactly how these changes will be interpreted at the receiving end, but he can probably create quite a bit of confusion. This cipher provides another example of the fact that the confidentiality and authentication 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 that the key string must be 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 (10×640 Megabytes), it probably has high latency.

The key string must be as long as the message. It is not hard to see that if the sender re-uses the one-time pad, an attacker 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.

2. Pseudo-random number generators

One shortcut to avoid having to send a long key string over a secure channel is to communicate a short key securely and use a pseudo-random number generator to generate a long key string from the short key. A *pseudo-random number generator* generates deterministically a random-appearing bit stream from a short *seed*. From the same seed, the pseudo-random generator will produce the same bit stream. Thus, if both the sender and the receiver have the secret short key, then using the pseudo-random generator they can generate the same, long key string from the short key and use that key for the transformation.

Unlike the one-time pad, this scheme can in principle be broken by someone who knows enough about the pseudo-random generator. The design requirement on a pseudo-random 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.

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 pseudo-random

generator is open and, as with the keys for transforming and untransforming, the seed is kept secret. High-security applications usually require that keys and seed originate from truly random processes. In lower-security applications, keys are typically created by pseudo-random number generators (starting with a truly random seed) on the basis that the resulting risk is tolerable.

Using a pseudo-random generator we need to send a relatively small seed, perhaps only 128 bits, over the secure channel. Unfortunately, we also have to rely on some cleverness: designing a good pseudo-random generator is tricky; one approach is to base it on a block-cipher such as DES, which we discuss below.

3. Shared-secret cryptography

Rather than designing ciphers that are impossible to break; most ciphers are designed in such a way that it is computationally infeasible to break them. The idea is that if it takes an unimaginable number of years of computation to break a particular cipher, then we can consider the cipher secure. Thus, ciphers in this class are *computationally secure* as opposed to be unbreakable.

In principle, computationally secure ciphers are evaluated by their work factor. The work factor is the minimum amount of work required to compute the shared-secret key or private key (given the public key) using ciphertext-only attacks. Work is measured in primitive operations (e.g., processor cycles). If the work factor is many years, then for all practical purposes, the cipher 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 secure ciphers are evaluated on their 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 cipher. This method of evaluation runs the risk that an attacker might come up with a better algorithm to break a cipher than the ones that are currently known. Theoreticians have developed models under which they can make absolute statements about the hardness of some ciphers. 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 3 decades, but also with many open problems. Given the complexities of designing a good computationally secure cipher, it is advisable to use only ciphers that have been around long enough that they have been subjected to much careful, public review.

We discuss two well-known computationally-based shared-secret ciphers: DES and RC4. *Data Encryption Standord (DES)* is an encryption system adopted by the U.S. government in 1976 (patent number 3,962,539) based on a proposal by International Business Machines (IBM) corporation. IBM's proposal used a 128-bit key, but that was changed after a little discussion with the National Security Agency (NSA). Instead, DES used a 56-bit key. DES is now widely regarded as too insecure for many applications, as distributed Internet computations or dedicated special-purpose machines can quickly find using a brute-force exhaustive search a 56-bit DES key given corresponding plaintext and ciphertext. Now there are multiple better algorithms available, including a new one released by the U.S. government: Advanced Encryption Standard (AES), which has 128-bit (or longer) keys and 128-bit plaintext and ciphertext blocks. A non-specialist should consult a specialist to

evaluate which is the best one for the task at hand.

3.1. A short-key shared-secret cipher: Data Encryption Standard (DES)

DES takes a 64-bit input and produces a 64-bit output. If you don't know the 56-bit key, it is hard to reconstruct the input given the output. The algorithm goes through 19 distinct stages: an initial transposition, followed by 16 iterations that are parameterized by the key, followed by a swap, and an inverse transposition. At each iteration i, DES splits the bits into 32 left bits (L) and 32 right bits (R) and then computes:

$$L_{i-1} \oplus f(R_{i-1}, K_i)$$

where f is a published complicated function (actually set of substitution tables) that mingles the bits based on the key. More precisely:

$$R_i = L_{i-1} \oplus f(R_{i-1}, K_i)$$

$$L_i = R_{i-1}$$

Since the same input always generates the same output, one has to be a bit careful in using DES. For example: if the attacker 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.

There are two good approaches to constructing a better transforming function using DES: cipher-block chaining, and using DES as a pseudo-random number generator. Both expand the length of a message slightly. *Cipher-block chaining (CBC)* randomizes each eight-byte plaintext block by XOR-ing it with the previous ciphertext block before transforming it (see figure 11-16). A dummy, random, ciphertext block, called the initialization vector (or IV) is inserted at the beginning to get things going. More formally, if the message has blocks M_1, M_2, …, M_n, the ciphertext consists of blocks C_0, C_1, …, C_n where:

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

To untransform, one computes:

$M_i = C_{i-1} \oplus D(C_i, key)$.

CBC encryption has *cascading change propagation*: 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 block, implies that two transformings 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

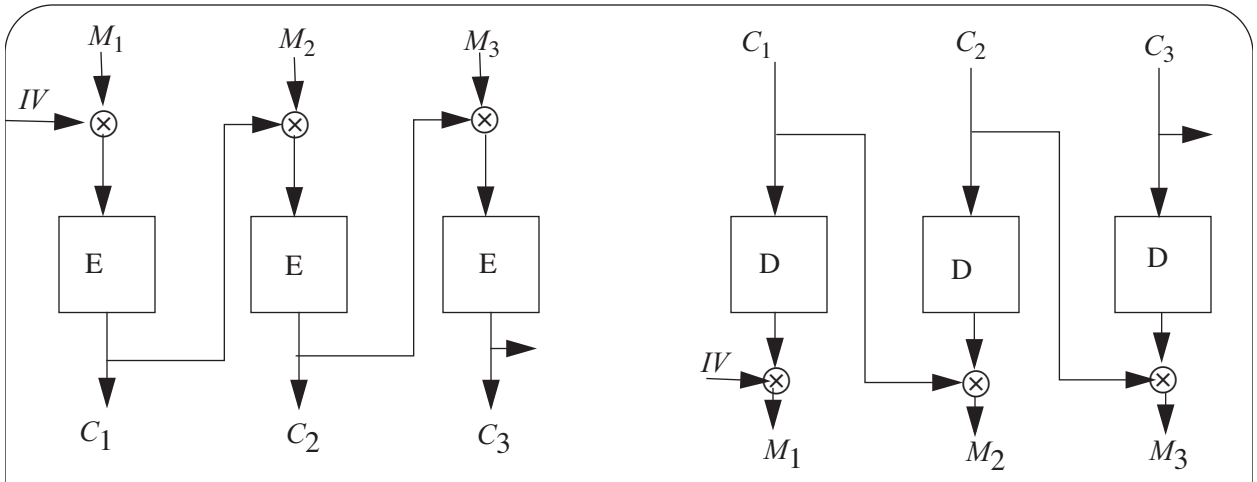


Figure 11-16: Cipher-block chaining

function of the IV and of all the message blocks. We will use this property later.

On the other hand, *CBC decryption* has bounded change propagation: 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 11-16 should convince you that this property holds.

Ciphers with bounded change propagation have important implications. In 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.

The second approach is to turn DES into a pseudo-random generator by picking a starting counter value *T*, sending it as the first block of ciphertext, and defining the *i*-th block of ciphertext as the XOR of the *i*-th message block with the transforming of the quantity *T*+*i*. More formally, if the message has blocks $M_1, M_2, \ldots, M_n$, then the ciphertext has blocks $C_0, C_1, \ldots, C_n$, where $C_0 = T$ and $C_i = M_i \oplus E(T+i, key)$. This is usually called "counter mode." *T* does not need to be random, but reusing counter values for the same key can cause a security breach. The sender should also be careful not to re-use the pseudo-random DES output (for the same reason that one should not re-use a one-time pad).

Ciphers with higher work factors than DES have been developed. For example, the next section introduces RC4, a cipher specifically designed to achieve confidentiality that has a much higher work factor than DES.

3.2. A shared-secret sealing system

RC4 is a shared-secret cipher, with variable size key, that is designed for sealing; like any stream cipher it cannot be used for authentication without additional mechanism. It 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 RC4 cipher 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:

```
i ⇐ (i + 1) mod 256;
j ⇐ (j + S[i]) mod 256;
SWAP(S[i], S[j]);
t ⇐ (S[i] + S[j]) mod 256;
K ⇐ S[t];
```

The sender seals a stream of bytes by sealing each byte individually. One seals byte B by computing $C = B$ *XOR* K. Then, one generates a new K for sealing the next byte.

The receiver unseals the byte C by computing C *XOR* K, which gives B.

The initialization procedure is also simple:

1. Fill each entry of S with its index. Thus, $S[0] \Leftarrow 0$, $S[1] \Leftarrow 1$, etc.

2. Allocate another 256-entry array (K) with each 8-bit entries. Fill K with the key, repeating the key 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.

3. Run the following initialization loop:

```
j ⇐ 0;
for (i ⇐ 0; i < 256; i ⇐ i + 1) {
        j ⇐ (j + S[i] + K[i]) mod 256;
        SWAP(S[i], S[j]);
}
i ⇐ j ⇐ 0;
```

RC4 is simple enough that it can be coded from memory, yet it appears to be a strong stream cipher for confidentiality. When using it to seal a long stream, it doesn't seem to have any small cycles and its output values vary highly. Finally, RC4 can be in about 256! x 256^2 possible states, which appears to create a barrier with a large work factor against brute-force attacks.

3.3. Computing a message authentication code

CBC-MAC is a simple message authentication code scheme based on DES in CBC mode. It uses cipher-block chaining. To sign a message M with a key k, Alice 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). All ciphertext blocks except the last are discarded, and the *last* ciphertext block is returned as the value of

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. Bob can verify the MAC by recomputing the MAC from M and key k using the same procedure that Alice used.

4. A public-key cipher

The security of DES relies on a transformation that is too complex to analyze, while the security of RSA relies on a simple-to-state (but hard to solve) well-known problem in number theory. RSA was developed at MIT 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, that 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.

4.1. Rivest-Shamir-Adleman (RSA)

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 $e \times d = 1 \pmod z$. 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.

To transform a text, you divide the text or a bit stream into blocks, not necessarily all of the same size, but all whose numerical value P is greater than or equal to zero and smaller than n. 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 $e \times d = 1 \pmod z$. 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 $k(p-1)(q-1) = 0 \pmod n$ is a result by Euler and Fermat (see I. Niven and H.S. Zuckerman, "An introduction to the Theory of Numbers", Wily, New York, 1980).

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 number would take 40 quadrillion years, using the best known algorithms and state-of-the-art hardware at that time. In 1994, Lenstra and Manasse factored a 129-digit 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, and because technology has improved substantially, allowing a set of computers to perform that much computation in a short time.

The security of RSA is based on historical work factor. At this point, there are no known algorithms for factoring large numbers very quickly. A number of other public-key ciphers exist, some of which are not covered by patents. But, to date no public-key system has been found where 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.

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?" Just for numbers that can be expressed with 512 bits or less, there are approximately 10^{151} primes. Therefore, we won't run out of prime numbers, if everyone needs two prime numbers different from everyone else's primes. Similarly, the probability that two persons will accidently pick the same prime number can be made very low. In addition, an attacker won't have a lot of success creating a database with all prime numbers; the number of primes is infinite.

4.2. Computing a digital signature

To produce an RSA digital signature private keys are used to sign, and the corresponding public keys are used to verify. RSA is reversible, since $(M^d)^e = (M^e)^d = M^{ed} \pmod n$; one can raise to the public exponent (e) first, and raise to the private exponent (d) second, or vice versa, and either way obtain the original message back. It is claimed that the security of RSA is equally good both ways.

When using RSA for digital signatures, *signing* of a message M, in principle, is just *transforming* the message M with the signer's private RSA key. This signature is appended to the message itself, like any authentication tag. Verification of the signature is done by applying the reverse transform to the signature with Alice's public key, and checking that the result is equal to the received message. The signed message can be passed around, and the signature can be re-verified as needed by anyone possessing Alice's public key.

For various technical reasons, this simple scheme is modified in practice. For example, if Eve succeeds in having Alice sign messages M_1 and M_2, then she can claim that Alice also signed M_3, where M_3 is the product of M_1 and M_2: $(M_3)^d = (M_1 \times M_2)^d = {M_1}^d \times {M_2}^d \pmod n$. Thus, if Eve sends M_3 to Bob, then Bob erroneously believes that Alice signed message M_3.

To avoid this problem (and some others) Alice usually creates a cryptographic hash of the message, and creates an authentication tag by signing this hash. (This also has the pleasant side effect that her signature doesn't have to be as long as the original message.) Bob verifies the signature by first recomputing the hash from the message, then applying the reverse transform to the received authentication tag with Alice's public key to obtain the received hash, and, finally, checking if the recomputed hash and the received hash are the same. In fact, 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.

4.3. A public-key sealing system

Sealing systems can also be constructed using public-key cryptography. To seal a message using RSA, the public key is used to seal, and the corresponding private key is used to unseal. To seal, one raises the message to the *public* exponent (d). To unseal one raises to the *private* exponent (e) second. This order is exactly the opposite of how a digital signature is computed; for signing the messages is raised to the private exponent for verifying it is raised to the public exponent.

RSA by itself doesn't guarantee confidentiality; there are a number of well-known

attacks if RSA is used by itself for sealing. For example, the length of blocks *P* should be about the same length of *n*, so one should pad short messages with independent randomized variables before sealing.

Seal and unseal use the same cryptographic operations (raising to the power of prime value) as for sign and verify. The explanation is that the RSA cipher can be used as a core to construct seal and unseal functions as well as sign and verify functions. As explained, RSA by itself is not sufficient to construct sign and verify functions. To construct good authentication functions, RSA is augmented, for example, with cryptographic hashing. To construct a good sealing system RSA is augmented, for example, with padding of randomized numbers.

5. Cryptographic hash functions

A final useful tool for a system designer is a cryptographic hash function. Despite the name, a cryptographic hash function does not have a key; it is a completely open design. A cryptographic hash function *H* has the following properties:

1. For a given message *M*, it is easy to compute *H(M) = V,* where *V* is the computed hash value;

2. It is difficult to compute *M* knowing only *H(M)*;

3. It is difficult to find another message *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 messages hashing to the same value *V* must be so low that one can neglect it in practice.

Because *V* is shorter than the original message *M,* function *H* must have a collision. Thus, the challenge in designing a cryptographic hash function is finding a function that has a negligible probability of generating collisions. The sidebar describes SHA-1, a widely-used cryptographic hash function. Even though SHA-1 must have collisions, no one has uncovered an example of one.

Sidebar 11-1: Secure Hash Algorithm (SHA-1)

SHA is a cryptographic hash algorithm. It takes as input a message of any length smaller than 2^{64} 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. We describe version 1 of SHA.

The hash is computed 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.

The message is split into 512-bit blocks. Each block is expanded from 512 bits (16 32-bit words) to 80 32-bit words:

```
     W[t] = M[t]for t = 0 to 15
     W[t] = (W[t]-3) XOR (W[t]-8) XOR (W[t]-14) XOR (W[t]-16)<<<1,
for t = 16 to 79,
```

where <<< is a left circular shift.

SHA uses 4 nonlinear functions and 4 32-bit constants. The 4 functions are

```
                (X & Y) | ((~X) & Z) for t = 0 to 19
f(t,x,y,x) =    (X XOR Y XOR Z, for t = 20 to 39
                (X & Y) | (X & Z) | (Y & Z), for t 40 to 59
                X XOR Y XOR Z, for t = 60 to 79
```

The constants are

```
           0x5A827999, for t = 0 to 19 /* 2.5/4 in hex */
k(t)=      0x6ED9EBA1, for t = 20 to 39 /* 3.5/4 in hex */
           0x8F1BBCDC, for t = 40 to 59 /* 5.5/5 in hex */
           0xCA62C1D6, for t = 60 to 79 /* 10.5/4 in hex */
```

(continued on next page)

Sidebar 11-1, continued: SHA-1

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

```
a = A = 0x67452301
b = B = 0xEFCDAB89
c = C = 0x98BADCFE
d = D = 0x10325476
e = E = 0xc3d2e1f0
```

The hash for a message is now computed as follows:

```
for (each 512-bit block of message) {
     for (t = 0; i < 80; t++) {
          x = (a <<< 5) + f(t, b,c,d) + e + W[t] + K[t];
          e = d;
          d = c;
          c = b <<< 30;
          b = a;
          a = x;
     }
     A += a; B += b; C += c; D += d; E += e;
}
hash = A, B, C, D, E;/* concatenate A, B, C, D, and E */
}
```

The justification for SHA is outside the scope of this book. Suffice to say it was designed by the NIST and the NSA, and no known cryptographic attacks exist.

Appendix 11–B. War Stories: Protection System Failures

A designer responsible for system security can bring to the job three different, related assets. The first is an understanding of the fundamental information protection concepts discussed in the main body of this chapter. The second is knowledge of several different real protection system designs; some examples have been discussed elsewhere in this chapter and more can be found in the Suggestions for Further Reading. This appendix concentrates on the third asset: familiarity with examples of real-world failures of information protection systems. In addition to encouraging a certain amount of humility, one can develop from failure 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 protection system must consider.

The case studies selected for description in this appendix 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 the fantasy that was added as part of the disguise. Consequently, this appendix cites original sources wherever possible.

In reviewing each case study, keep in mind several questions:

- What design principles were violated by this design? (Often the evidence indicates more than one was ignored.)
- Is this failure a particular example of a more general class of failure that the same system may exhibit in other forms, too? What other systems are likely to fail in the same (or analogous) ways?
- Given the situation in the field that surrounded this system, what should have been the most appropriate short-term response? Given a longer-term opportunity to redesign things, what would you suggest?

1. Residues: profitable garbage

Protection 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; attackers have discovered secrets by reading the contents of newly allocated primary memory, disk space, and recycled magnetic tapes as well as by pawing through piles of physical trash (popularly known as "dumpster diving").

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. But 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.*

1.2. 1997: Residues in network packets

Kerberos is a key distribution system developed at M. I. T. and used for authentication there and elsewhere. If one sends a badly formed request to a Kerberos Version 4 server, the service responds with a packet containing an error message. Since the error message is a terminated string, one might not think of looking into the contents of the packet after the string. Unfortunately, following the error string was the residue of the previous packet sent out by that Kerberos service. That previous packet was probably a response to a correctly formed request, and it 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 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†

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 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

---

* 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.

the service return just the data up to the end of the file and alert the client about the error.

The Apple Macintosh AppleShare IP 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.*

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 pornographic images to transfer.†

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

---

* 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/>.

† 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.

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 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.[*, †]

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. But 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. At one time, DoD regulation 5200.28–M called for overwriting of certain magnetic media 1000 or more times before a medium formerly containing classified information could be considered "declassified" and used for other purposes or returned to the vendor for repair.[‡]

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.[**] And 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).

‡ *Automated Information System Security Manual: Techniques and Procedures for Implementing, Deactivating, Testing, and Evaluating Secure Resource-Sharing ADP Systems*. United States Department of Defense Directive Manual 5200.28-M, Jan 1973, revised June 1979. A slightly more up-to-date version of these guidelines can be found in *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).

2. Plaintext passwords cause two failures

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 a text file 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).*

At another time, the same system administrator was reviewing and updating the master user list with the standard text editor. The editor program, to assure 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 the intermediate file, 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 temporary file; 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.†

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.)

But when the inverse transformation was actually tried 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

---

* 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 3, which follows.

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. But 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.*

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 are often the easiest thing 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 inappropriate or unauthorized action by an authorized person. And in many systems the number of people authorized to tinker with the configuration is alarmingly large.

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.†

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 protection checks. They then mailed to the DoD installation a copy

---

* 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).

† Reported by Richard G. Mills, 1965.

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.*

5. The kernel trusts the user

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 data base, 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.‡

5.2. Nonobvious trust (tocttou)

As a subtle variation of the previous problem, consider the following user-callable

---

* 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 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.

† Noticed by the author, exploit developed by Maxim G. Smith, 1963.

‡ Allocation strategy reported by Fred Brooks in *The Mythical Man-Month*.

kernel entry point (in this example, the language uses call-by-reference for arguments):

```
procedure DELETE_FILE (file_name, code)
{
        CHECK_AUTH(file_name, user_id, code);
        if code = 0
                then DESTROY(file_name);
        return;
}
```

This program is apparently correctly checking to see that the current user, identified in the global variable *user_id*, has correct authority before actually destroying the file. Program CHECK_AUTH sets *code* to some nonzero value if it finds that the user named by *user_id* does not have the necessary authority for file *file_name*. Unfortunately, at least two ways exist to defeat the test.

The variables *file_name* and *code* are user-supplied arguments, allocated and stored in user-chosen and user-accessible areas. The user can, in a parallel thread, observe the value of code being set by CHECK_AUTH and quickly reset it before DELETE_FILE gets a chance to test it. Alternatively, by careful timing, the user may be able to change the name stored in variable *file_name* between the time CHECK_AUTH examines it and DESTROY uses it, thereby causing deletion of some other file that CHECK_AUTH would not have permitted. This class of error is so common in kernel implementations that it is called the "Time Of Check To Time Of Use" race, written "tocttou" and pronounced "tock-two".*

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 protection constraints, and even more difficult for an operating system to create virtual DMA 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. In the case of CTSS, this problem led to a prohibition on timing-dependent and self-modifying DMA channel programs. The problem with this approach was that there was no methodical way to establish by inspection that a program conformed with the restriction. A battle of wits ensued. For every ingenious technique developed to discover that a DMA channel program contained an obscure self-modification feature, some clever user discovered a still more obscure way to conceal self-modification. Similar problems have been noted with virtualization of I/O channels in the IBM System/360.†

---

* 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).

6. Technology defeats economic barriers

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 then-novel 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 the exacting standards for availability, reliability, durability, maintainability, compatibility, and noise immunity that apply to a commercial ground station design, and (2) improvements in technology may rapidly reduce an eavesdropper's cost.

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 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.†

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 sealed message, and unseal and authenticate the response, within 90 minutes. They were

† 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.

† 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.

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 remainder 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.* One can draw at least two relevant lessons from the report:

1. The mental model that one needs to make effective 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.

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. But launching an application program to act on a file whose contents are specified by someone else can be dangerous.

Cognizant of this problem, 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.

Unfortunately, it is apparently 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

* 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.

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. But, 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.

The fact that these security problems were not discovered before product release suggests that competitive pressure can easily dominate security concern. One would expect that even a moderately superficial security inspection would have quickly revealed each of these problems.*

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

* Chris Rioux provided details on this collection of browser problems, and discovered the .ISP exploitation in 1998.

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.*

10. Signaling with clandestine channels

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 nondiscretionary controls may be in place, the program 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".

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. They also discuss how to design fonts to minimize unwanted radiation.

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

---

* This anecdote was reported in the 1970's, but its source has been lost.

† 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 website, at <www.multicians.org/thvv/timing-chn.html>.

‡ 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).

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.*

11. It seems to be working just fine

A hazard with systems that provide security is that there often is no obvious indication that they aren't actually doing their job.

11.1. But 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

* 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.

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 nonzero status. Assuming that 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 seal 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, seal it with the permanent key, and send the result to the service. The service unsealed the first block using the permanent key to obtain the temporary session key, and then both ends used the session key to seal and unseal 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, sealed it, and sent it to the service. The service unsealed 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.*

11.2. How large is the key space…really?

When a client presents a Kerberos ticket to a service, 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 seal the communication channel or to authenticate later messages.

---

* Reported by Theodore T'so in 1997.

Generating an unpredictable session key involves choosing a number at random from the 56-bit Data Encryption Standard key space. Since computers aren't very 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, 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. Nevertheless, 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.*

11.3. 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 sealing 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…"†

---

* 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.

Some web browsers display an indication, such as a padlock icon, that cryptographic transformations are 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.

12. Incomplete checking of parameters

A common method of penetrating systems is to examine the source code at the kernel entry points looking for places where unexpected, out-of-range parameter values might lead to a vulnerability. An interesting example occurred in an operating system that allowed the user to request a different memory allocation. In that system a penetrator discovered that if the user requested a negative memory size, the kernel would blindly assign additional contiguous storage at the wrong end of the user's program. That particular area happened to contain critical kernel information regarding that user, and the user could, by modifying it, obtain control of the kernel and thence of the entire system.*

13. Spoofing the operator

Many operating systems include a feature allowing a logged-in user to send a message to the system operator, for example, to ask a question. 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 it source. In CTSS, 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, displayed enough blank lines to push the warning banner out of sight, and then displayed a line that looked like a system-provided message, such as a warning that overheating had been detected, leading the operator to immediately shut down the system.†

14. 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.‡

---

† Jeremy Epstein. *Risks-Forum Digest 20,* 64 (Thursday 4 November 1999).

* This anecdote was reported in the 1970's, but its source has been lost.

† This vulnerability was noticed by Multics staff programmers in the late 1960's. As far as is known, it was never actually exploited.

15. A thorough system penetration job

One particularly thorough system penetration operation went as follows. First, the team of attackers obtained computer time at an installation different from the one to be penetrated, but which 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 compromising a critical kernel protection 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.†

16. Framing Enigma

Enigma is a family of encipherment machines designed in Poland and Germany in the 1920's and 1930's. An Enigma machine consists of a series of rotors, each with contacts on both sides, as in figure 1. 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 total number of key possibilities was thus about $3x10^{14}$. This number can be compared with the DES key space of $2x10^{17}$, about 600 times larger. Since DES was designed 50 years later and intended to cope with modern computers, the Enigma key space was, in terms of the computational abilities available during WWII, fairly formidable. (Or perhaps the converse is true: if Enigma was breakable by brute force then, DES must be

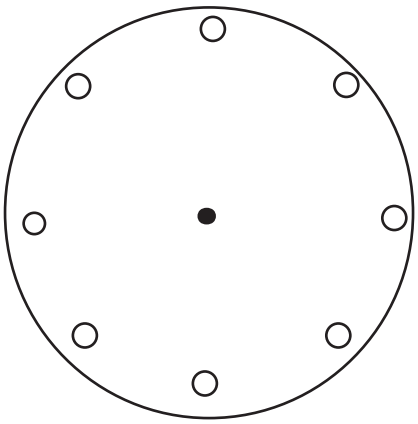
---

‡ Karger and Schell, *op. cit.*, section 3.2.2.

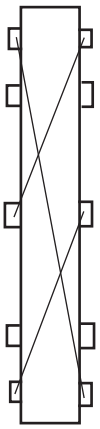
* Schell, 1979 *op. cit.*, page 22.

† Karger and Schell, 1974 *op. cit.,* sections 3.4.5 and 3.4.6.

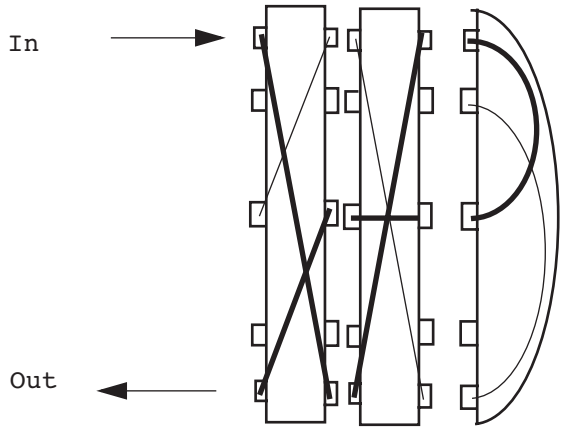
Figure 1: Enigma Rotor with eight contacts



Side view, showing contacts.



Edge view, showing some connections.



Two Enigma Rotors with a reflector, showing an input-output path.

equally breakable now.) 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. But this change created a restriction: since the reflector had to 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.*

* 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].