

Security



Protection, as we discussed in Chapter 13, is strictly an *internal* problem: How do we provide controlled access to programs and data stored in a computer system? **Security**, on the other hand, requires not only an adequate protection system but also consideration of the *external* environment within which the system operates. A protection system is ineffective if user authentication is compromised or a program is run by an unauthorized user.

Computer resources must be guarded against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. These resources include information stored in the system (both data and code), as well as the CPU, memory, disks, tapes, and networking that are the computer. In this chapter, we start by examining ways in which resources may be accidentally or purposely misused. We then explore a key security enabler—cryptography. Finally, we look at mechanisms to guard against or detect attacks.

CHAPTER OBJECTIVES

- To discuss security threats and attacks.
- To explain the fundamentals of encryption, authentication, and hashing.
- To examine the uses of cryptography in computing.
- To describe various countermeasures to security attacks.

14.1 The Security Problem

In many applications, ensuring the security of the computer system is worth considerable effort. Large commercial systems containing payroll or other financial data are inviting targets to thieves. Systems that contain data pertaining to corporate operations may be of interest to unscrupulous competitors. Furthermore, loss of such data, whether by accident or fraud, can seriously impair the ability of the corporation to function.

In Chapter 13, we discussed mechanisms that the operating system can provide (with appropriate aid from the hardware) that allow users to protect

their resources, including programs and data. These mechanisms work well only as long as the users conform to the intended use of and access to these resources. We say that a system is **secure** if its resources are used and accessed as intended under all circumstances. Unfortunately, total security cannot be achieved. Nonetheless, we must have mechanisms to make security breaches a rare occurrence, rather than the norm.

Security violations (or misuse) of the system can be categorized as intentional (malicious) or accidental. It is easier to protect against accidental misuse than against malicious misuse. For the most part, protection mechanisms are the core of protection from accidents. The following list includes several forms of accidental and malicious security violations. We should note that in our discussion of security, we use the terms **intruder** and **cracker** for those attempting to breach security. In addition, a **threat** is the potential for a security violation, such as the discovery of a vulnerability, whereas an **attack** is the attempt to break security.

- **Breach of confidentiality.** This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.
- **Breach of integrity.** This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.
- **Breach of availability.** This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Website defacement is a common example of this type of security breach.
- **Theft of service.** This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.
- **Denial of service.** This violation involves preventing legitimate use of the system. **Denial-of-service (DOS)** attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread. We discuss DOS attacks further in Section 14.3.3.

Attackers use several standard methods in their attempts to breach security. The most common is **masquerading**, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach **authentication**, the correctness of identification; they can then gain access that they would not normally be allowed or escalate their privileges—obtain privileges to which they would not normally be entitled. Another common attack is to replay a captured exchange of data. A **replay attack** consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack—for example, in a repeat of a request to transfer money. But frequently it is done along with **message modification**, again to escalate privileges. Consider

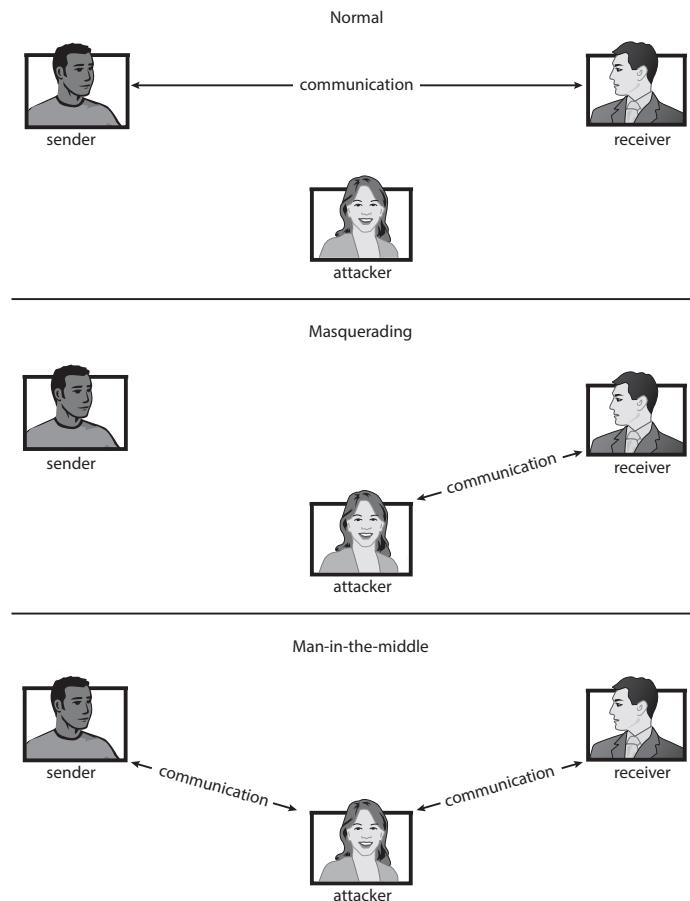


Figure 14.1 Standard security attacks.

the damage that could be done if a request for authentication had a legitimate user's information replaced with an unauthorized user's. Yet another kind of attack is the **man-in-the-middle attack**, in which an attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa. In a network communication, a man-in-the-middle attack may be preceded by a **session hijacking**, in which an active communication session is intercepted. Several attack methods are depicted in Figure 14.1.

As we have already suggested, absolute protection of the system from malicious abuse is not possible, but the cost to the perpetrator can be made sufficiently high to deter most intruders. In some cases, such as a denial-of-service attack, it is preferable to prevent the attack but sufficient to detect the attack so that countermeasures can be taken.

To protect a system, we must take security measures at four levels:

1. **Physical.** The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders. Both the machine rooms and the terminals or workstations that have access to the machines must be secured.

2. **Human.** Authorization must be done carefully to assure that only appropriate users have access to the system. Even authorized users, however, may be “encouraged” to let others use their access (in exchange for a bribe, for example). They may also be tricked into allowing access via **social engineering**. One type of social-engineering attack is **phishing**. Here, a legitimate-looking e-mail or web page misleads a user into entering confidential information. Another technique is **dumpster diving**, a general term for attempting to gather information in order to gain unauthorized access to the computer (by looking through trash, finding phone books, or finding notes containing passwords, for example). These security problems are management and personnel issues, not problems pertaining to operating systems.
3. **Operating system.** The system must protect itself from accidental or purposeful security breaches. A runaway process could constitute an accidental denial-of-service attack. A query to a service could reveal passwords. A stack overflow could allow the launching of an unauthorized process. The list of possible breaches is almost endless.
4. **Network.** Much computer data in modern systems travels over private leased lines, shared lines like the Internet, wireless connections, or dial-up lines. Intercepting these data could be just as harmful as breaking into a computer, and interruption of communications could constitute a remote denial-of-service attack, diminishing users’ use of and trust in the system.

Security at the first two levels must be maintained if operating-system security is to be ensured. A weakness at a high level of security (physical or human) allows circumvention of strict low-level (operating-system) security measures. Thus, the old adage that a chain is only as strong as its weakest link is especially true of system security. All of these aspects must be addressed for security to be maintained.

Furthermore, the system must provide protection (Chapter 13) to allow the implementation of security features. Without the ability to authorize users and processes, to control their access, and to log their activities, it would be impossible for an operating system to implement security measures or to run securely. Hardware protection features are needed to support an overall protection scheme. For example, a system without memory protection cannot be secure. New hardware features are allowing systems to be made more secure, as we shall discuss.

Unfortunately, little in security is straightforward. As intruders exploit security vulnerabilities, security countermeasures are created and deployed. This causes intruders to become more sophisticated in their attacks. For example, recent security incidents include the use of spyware to provide a conduit for spam through innocent systems (we discuss this practice in Section 14.2). This cat-and-mouse game is likely to continue, with more security tools needed to block the escalating intruder techniques and activities.

In the remainder of this chapter, we address security at the network and operating-system levels. Security at the physical and human levels, although important, is for the most part beyond the scope of this text. Security within the operating system and between operating systems is implemented in several

ways, ranging from passwords for authentication through guarding against viruses to detecting intrusions. We start with an exploration of security threats.

14.2 Program Threats

Processes, along with the kernel, are the only means of accomplishing work on a computer. Therefore, writing a program that creates a breach of security, or causing a normal process to change its behavior and create a breach, is a common goal of crackers. In fact, even most nonprogram security events have as their goal causing a program threat. For example, while it is useful to log in to a system without authorization, it is quite a lot more useful to leave behind a **back-door** daemon that provides information or allows easy access even if the original exploit is blocked. In this section, we describe common methods by which programs cause security breaches. Note that there is considerable variation in the naming conventions for security holes and that we use the most common or descriptive terms.

14.2.1 Trojan Horse

Many systems have mechanisms for allowing programs written by users to be executed by other users. If these programs are executed in a domain that provides the access rights of the executing user, the other users may misuse these rights. A text-editor program, for example, may include code to search the file to be edited for certain keywords. If any are found, the entire file may be copied to a special area accessible to the creator of the text editor. A code segment that misuses its environment is called a **Trojan horse**. Long search paths, such as are common on UNIX systems, exacerbate the Trojan-horse problem. The search path lists the set of directories to search when an ambiguous program name is given. The path is searched for a file of that name, and the file is executed. All the directories in such a search path must be secure, or a Trojan horse could be slipped into the user's path and executed accidentally.

For instance, consider the use of the “.” character in a search path. The “.” tells the shell to include the current directory in the search. Thus, if a user has “.” in her search path, has set her current directory to a friend's directory, and enters the name of a normal system command, the command may be executed from the friend's directory. The program will run within the user's domain, allowing the program to do anything that the user is allowed to do, including deleting the user's files, for instance.

A variation of the Trojan horse is a program that emulates a login program. An unsuspecting user starts to log in at a terminal and notices that he has apparently mistyped his password. He tries again and is successful. What has happened is that his authentication key and password have been stolen by the login emulator, which was left running on the terminal by the thief. The emulator stored away the password, printed out a login error message, and exited; the user was then provided with a genuine login prompt. This type of attack can be defeated by having the operating system print a usage message at the end of an interactive session or by a nontrappable key sequence,

such as the control-alt-delete combination used by all modern Windows operating systems.

Another variation on the Trojan horse is **spyware**. Spyware sometimes accompanies a program that the user has chosen to install. Most frequently, it comes along with freeware or shareware programs, but sometimes it is included with commercial software. The goal of spyware is to download ads to display on the user's system, create pop-up browser windows when certain sites are visited, or capture information from the user's system and return it to a central site. This latter practice is an example of a general category of attacks known as **covert channels**, in which surreptitious communication occurs. For example, the installation of an innocuous-seeming program on a Windows system could result in the loading of a spyware daemon. The spyware could contact a central site, be given a message and a list of recipient addresses, and deliver a spam message to those users from the Windows machine. This process continues until the user discovers the spyware. Frequently, the spyware is not discovered. In 2010, it was estimated that 90 percent of spam was being delivered by this method. This theft of service is not even considered a crime in most countries!

Spyware is a micro example of a macro problem: violation of the principle of least privilege. Under most circumstances, a user of an operating system does not need to install network daemons. Such daemons are installed via two mistakes. First, a user may run with more privileges than necessary (for example, as the administrator), allowing programs that she runs to have more access to the system than is necessary. This is a case of human error—a common security weakness. Second, an operating system may allow by default more privileges than a normal user needs. This is a case of poor operating-system design decisions. An operating system (and, indeed, software in general) should allow fine-grained control of access and security, but it must also be easy to manage and understand. Inconvenient or inadequate security measures are bound to be circumvented, causing an overall weakening of the security they were designed to implement.

14.2.2 Trap Door

The designer of a program or system might leave a hole in the software that only she is capable of using. This type of security breach (or **trap door**) was shown in the movie *War Games*. For instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures. Programmers have been arrested for embezzling from banks by including rounding errors in their code and having the occasional half-cent credited to their accounts. This account crediting can add up to a large amount of money, considering the number of transactions that a large bank executes.

A clever trap door could be included in a compiler. The compiler could generate standard object code as well as a trap door, regardless of the source code being compiled. This activity is particularly nefarious, since a search of the source code of the program will not reveal any problems. Only the source code of the compiler would contain the information.

Trap doors pose a difficult problem because, to detect them, we have to analyze all the source code for all components of a system. Given that software systems may consist of millions of lines of code, this analysis is not done frequently, and frequently it is not done at all!

14.2.3 Logic Bomb

Consider a program that initiates a security incident only under certain circumstances. It would be hard to detect because under normal operations, there would be no security hole. However, when a predefined set of parameters was met, the security hole would be created. This scenario is known as a **logic bomb**. A programmer, for example, might write code to detect whether he was still employed; if that check failed, a daemon could be spawned to allow remote access, or code could be launched to cause damage to the site.

14.2.4 Stack and Buffer Overflow

The stack- or buffer-overflow attack is the most common way for an attacker outside the system, on a network or dial-up connection, to gain unauthorized access to the target system. An authorized user of the system may also use this exploit for privilege escalation.

Essentially, the attack exploits a bug in a program. The bug can be a simple case of poor programming, in which the programmer neglected to code bounds checking on an input field. In this case, the attacker sends more data than the program was expecting. By using trial and error, or by examining the source code of the attacked program if it is available, the attacker determines the vulnerability and writes a program to do the following:

1. Overflow an input field, command-line argument, or input buffer—for example, on a network daemon—until it writes into the stack.
2. Overwrite the current return address on the stack with the address of the exploit code loaded in step 3.
3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute—for instance, spawn a shell.

The result of this attack program's execution will be a root shell or other privileged command execution.

For instance, if a web-page form expects a user name to be entered into a field, the attacker could send the user name, plus extra characters to overflow the buffer and reach the stack, plus a new return address to load onto the stack, plus the code the attacker wants to run. When the buffer-reading subroutine returns from execution, the return address is the exploit code, and the code is run.

Let's look at a buffer-overflow exploit in more detail. Consider the simple C program shown in Figure 14.2. This program creates a character array of size BUFFER_SIZE and copies the contents of the parameter provided on the command line—`argv[1]`. As long as the size of this parameter is less than BUFFER_SIZE (we need one byte to store the null terminator), this program works properly. But consider what happens if the parameter provided on the command line is longer than BUFFER_SIZE. In this scenario, the `strcpy()` function will begin copying from `argv[1]` until it encounters a null terminator (\0) or until the program crashes. Thus, this program suffers from a potential buffer-overflow problem in which copied data overflow the buffer array.

```

#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[])
{
    char buffer[BUFFER_SIZE];

    if (argc < 2)
        return -1;
    else {
        strcpy(buffer, argv[1]);
        return 0;
    }
}

```

Figure 14.2 C program with buffer-overflow condition.

Note that a careful programmer could have performed bounds checking on the size of `argv[1]` by using the `strncpy()` function rather than `strcpy()`, replacing the line “`strcpy(buffer, argv[1]);`” with “`strncpy(buffer, argv[1], sizeof(buffer)-1);`”. Unfortunately, good bounds checking is the exception rather than the norm.

Furthermore, lack of bounds checking is not the only possible cause of the behavior of the program in Figure 14.2. The program could instead have been carefully designed to compromise the integrity of the system. We now consider the possible security vulnerabilities of a buffer overflow.

When a function is invoked in a typical computer architecture, the variables defined locally to the function (sometimes known as **automatic variables**), the parameters passed to the function, and the address to which control returns once the function exits are stored in a **stack frame**. The layout for a typical stack frame is shown in Figure 14.3. Examining the stack frame from top to bottom, we first see the parameters passed to the function, followed by any automatic variables declared in the function. We next see the **frame pointer**, which is the address of the beginning of the stack frame. Finally, we have the return

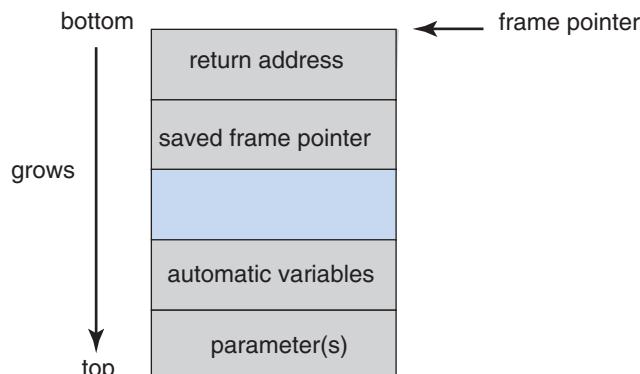


Figure 14.3 The layout for a typical stack frame.

address, which specifies where to return control once the function exits. The frame pointer must be saved on the stack, as the value of the stack pointer can vary during the function call. The saved frame pointer allows relative access to parameters and automatic variables.

Given this standard memory layout, a cracker could execute a buffer-overflow attack. Her goal is to replace the return address in the stack frame so that it now points to the code segment containing the attacking program.

The programmer first writes a short code segment such as the following:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    execvp('\\bin\\sh', '\\bin \\sh', NULL);
    return 0;
}
```

Using the `execvp()` system call, this code segment creates a shell process. If the program being attacked runs with system-wide permissions, this newly created shell will gain complete access to the system. Of course, the code segment could do anything allowed by the privileges of the attacked process. This code segment is then compiled so that the assembly language instructions can be modified. The primary modification is to remove unnecessary features in the code, thereby reducing the code size so that it can fit into a stack frame. This assembled code fragment is now a binary sequence that will be at the heart of the attack.

Refer again to the program shown in Figure 14.2. Let's assume that when the `main()` function is called in that program, the stack frame appears as shown in Figure 14.4(a). Using a debugger, the programmer then finds the

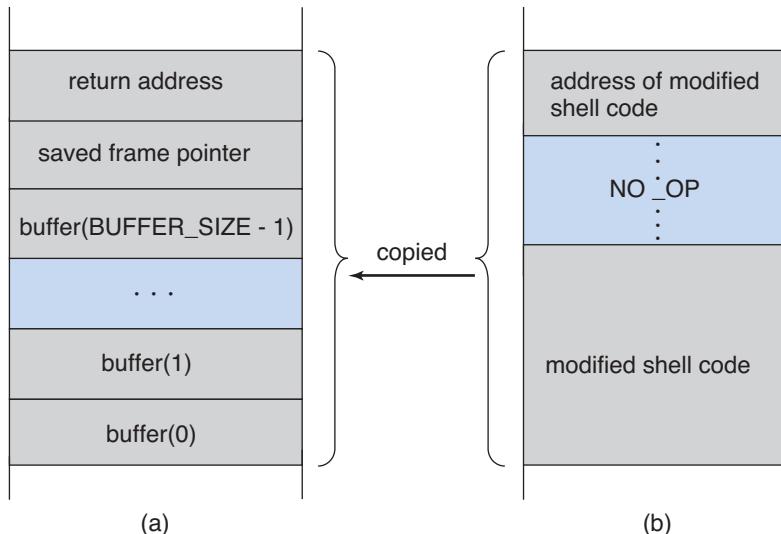


Figure 14.4 Hypothetical stack frame for Figure 14.2, (a) before and (b) after.

address of `buffer[0]` in the stack. That address is the location of the code the attacker wants executed. The binary sequence is appended with the necessary amount of NO-OP instructions (for NO-OPERation) to fill the stack frame up to the location of the return address, and the location of `buffer[0]`, the new return address, is added. The attack is complete when the attacker gives this constructed binary sequence as input to the process. The process then copies the binary sequence from `argv[1]` to position `buffer[0]` in the stack frame. Now, when control returns from `main()`, instead of returning to the location specified by the old value of the return address, we return to the modified shell code, which runs with the access rights of the attacked process! Figure 14.4(b) contains the modified shell code.

There are many ways to exploit potential buffer-overflow problems. In this example, we considered the possibility that the program being attacked—the code shown in Figure 14.2—ran with system-wide permissions. However, the code segment that runs once the value of the return address has been modified might perform any type of malicious act, such as deleting files, opening network ports for further exploitation, and so on.

This example buffer-overflow attack reveals that considerable knowledge and programming skill are needed to recognize exploitable code and then to exploit it. Unfortunately, it does not take great programmers to launch security attacks. Rather, one cracker can determine the bug and then write an exploit. Anyone with rudimentary computer skills and access to the exploit—a so-called **script kiddie**—can then try to launch the attack at target systems.

The buffer-overflow attack is especially pernicious because it can be run between systems and can travel over allowed communication channels. Such attacks can occur within protocols that are expected to be used to communicate with the target machine, and they can therefore be hard to detect and prevent. They can even bypass the security added by firewalls (Section 14.7).

One solution to this problem is for the CPU to have a feature that disallows execution of code in a stack section of memory. Recent versions of Sun's SPARC chip include this setting, and recent versions of Solaris enable it. The return address of the overflowed routine can still be modified; but when the return address is within the stack and the code there attempts to execute, an exception is generated, and the program is halted with an error.

Recent versions of AMD and Intel x86 chips include the NX feature to prevent this type of attack. The use of the feature is supported in several x86 operating systems, including Linux and Windows XP SP2. The hardware implementation involves the use of a new bit in the page tables of the CPUs. This bit marks the associated page as nonexecutable, so that instructions cannot be read from it and executed. As this feature becomes more prevalent, buffer-overflow attacks should greatly diminish.

14.2.5 Viruses

Another form of program threat is a **virus**. A virus is a fragment of code embedded in a legitimate program. Viruses are self-replicating and are designed to “infect” other programs. They can wreak havoc in a system by modifying or destroying files and causing system crashes and program malfunctions. As with most penetration attacks, viruses are very specific to architectures, operating systems, and applications. Viruses are a particular problem for users of

PCs. UNIX and other multiuser operating systems generally are not susceptible to viruses because the executable programs are protected from writing by the operating system. Even if a virus does infect such a program, its powers usually are limited because other aspects of the system are protected.

Viruses are usually borne via e-mail, with spam the most common vector. They can also spread when users download viral programs from Internet file-sharing services or exchange infected disks.

Another common form of virus transmission uses Microsoft Office files, such as Microsoft Word documents. These documents can contain *macros* (or Visual Basic programs) that programs in the Office suite (Word, PowerPoint, and Excel) will execute automatically. Because these programs run under the user's own account, the macros can run largely unconstrained (for example, deleting user files at will). Commonly, the virus will also e-mail itself to others in the user's contact list. Here is a code sample that shows how simple it is to write a Visual Basic macro that a virus could use to format the hard drive of a Windows computer as soon as the file containing the macro was opened:

```
Sub AutoOpen()
Dim oFS
    Set oFS = CreateObject("Scripting.FileSystemObject")
    vs = Shell("c:\ command.com /k format c:",vbHide)
End Sub
```

How do viruses work? Once a virus reaches a target machine, a program known as a **virus dropper** inserts the virus into the system. The virus dropper is usually a Trojan horse, executed for other reasons but installing the virus as its core activity. Once installed, the virus may do any one of a number of things. There are literally thousands of viruses, but they fall into several main categories. Note that many viruses belong to more than one category.

- **File.** A standard file virus infects a system by appending itself to a file. It changes the start of the program so that execution jumps to its code. After it executes, it returns control to the program so that its execution is not noticed. File viruses are sometimes known as parasitic viruses, as they leave no full files behind and leave the host program still functional.
- **Boot.** A boot virus infects the boot sector of the system, executing every time the system is booted and before the operating system is loaded. It watches for other bootable media and infects them. These viruses are also known as memory viruses, because they do not appear in the file system. Figure 14.5 shows how a boot virus works.
- **Macro.** Most viruses are written in a low-level language, such as assembly or C. Macro viruses are written in a high-level language, such as Visual Basic. These viruses are triggered when a program capable of executing the macro is run. For example, a macro virus could be contained in a spreadsheet file.
- **Source code.** A source code virus looks for source code and modifies it to include the virus and to help spread the virus.

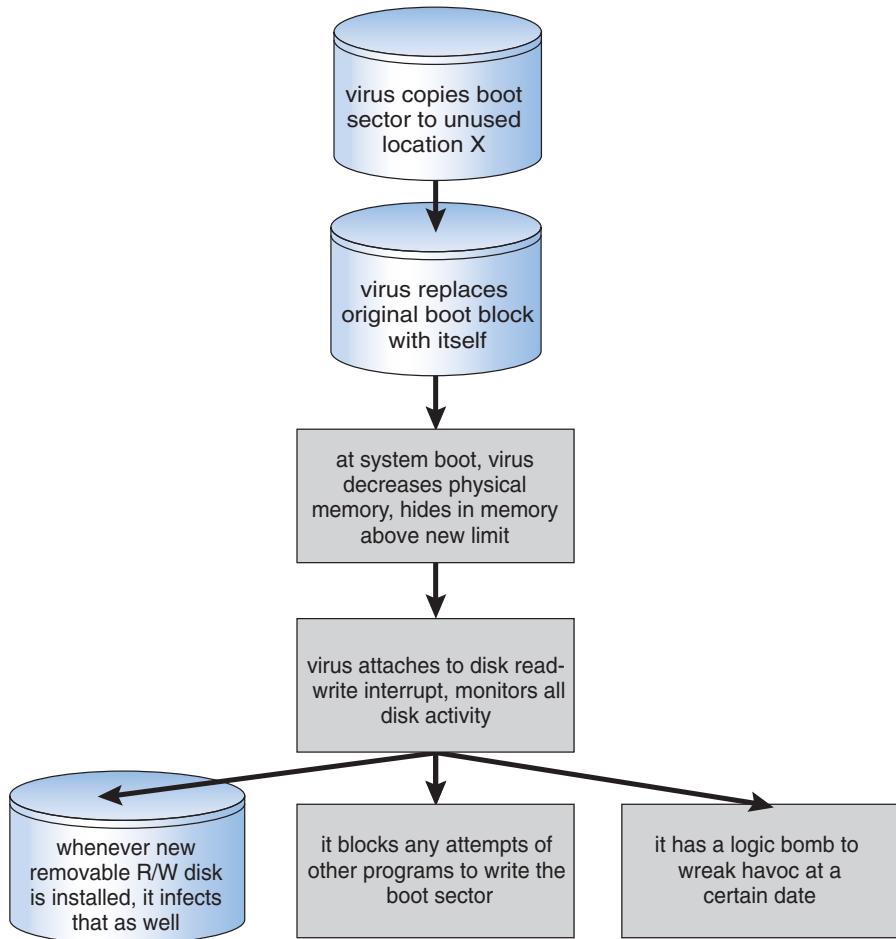


Figure 14.5 A boot-sector computer virus.

- **Polymorphic.** A polymorphic virus changes each time it is installed to avoid detection by antivirus software. The changes do not affect the virus's functionality but rather change the virus's signature. A **virus signature** is a pattern that can be used to identify a virus, typically a series of bytes that make up the virus code.
- **Encrypted.** An encrypted virus includes decryption code along with the encrypted virus, again to avoid detection. The virus first decrypts and then executes.
- **Stealth.** This tricky virus attempts to avoid detection by modifying parts of the system that could be used to detect it. For example, it could modify the read system call so that if the file it has modified is read, the original form of the code is returned rather than the infected code.
- **Tunneling.** This virus attempts to bypass detection by an antivirus scanner by installing itself in the interrupt-handler chain. Similar viruses install themselves in device drivers.

- **Multipartite.** A virus of this type is able to infect multiple parts of a system, including boot sectors, memory, and files. This makes it difficult to detect and contain.
- **Armored.** An armored virus is coded to make it hard for antivirus researchers to unravel and understand. It can also be compressed to avoid detection and disinfection. In addition, virus droppers and other full files that are part of a virus infestation are frequently hidden via file attributes or unviewable file names.

This vast variety of viruses has continued to grow. For example, in 2004 a new and widespread virus was detected. It exploited three separate bugs for its operation. This virus started by infecting hundreds of Windows servers (including many trusted sites) running Microsoft Internet Information Server (IIS). Any vulnerable Microsoft Explorer web browser visiting those sites received a browser virus with any download. The browser virus installed several back-door programs, including a **keystroke logger**, which records everything entered on the keyboard (including passwords and credit-card numbers). It also installed a daemon to allow unlimited remote access by an intruder and another that allowed an intruder to route spam through the infected desktop computer.

Generally, viruses are the most disruptive security attacks, and because they are effective, they will continue to be written and to spread. An active security-related debate within the computing community concerns the existence of a **monoculture**, in which many systems run the same hardware, operating system, and application software. This monoculture supposedly consists of Microsoft products. One question is whether such a monoculture even exists today. Another question is whether, if it does, it increases the threat of and damage caused by viruses and other security intrusions.

14.3 System and Network Threats

Program threats typically use a breakdown in the protection mechanisms of a system to attack programs. In contrast, system and network threats involve the abuse of services and network connections. System and network threats create a situation in which operating-system resources and user files are misused. Sometimes, a system and network attack is used to launch a program attack, and vice versa.

The more *open* an operating system is—the more services it has enabled and the more functions it allows—the more likely it is that a bug is available to exploit. Increasingly, operating systems strive to be **secure by default**. For example, Solaris 10 moved from a model in which many services (FTP, telnet, and others) were enabled by default when the system was installed to a model in which almost all services are disabled at installation time and must specifically be enabled by system administrators. Such changes reduce the system's **attack surface**—the set of ways in which an attacker can try to break into the system.

In the remainder of this section, we discuss some examples of system and network threats, including worms, port scanning, and denial-of-service

attacks. It is important to note that masquerading and replay attacks are also commonly launched over networks between systems. In fact, these attacks are more effective and harder to counter when multiple systems are involved. For example, within a computer, the operating system usually can determine the sender and receiver of a message. Even if the sender changes to the ID of someone else, there may be a record of that ID change. When multiple systems are involved, especially systems controlled by attackers, then such tracing is much more difficult.

In general, we can say that sharing secrets (to prove identity and as keys to encryption) is required for authentication and encryption, and sharing secrets is easier in environments (such as a single operating system) in which secure sharing methods exist. These methods include shared memory and interprocess communications. Creating secure communication and authentication is discussed in Section 14.4 and in Section 14.5.

14.3.1 Worms

A **worm** is a process that uses the **spawn** mechanism to duplicate itself. The worm spawns copies of itself, using up system resources and perhaps locking out all other processes. On computer networks, worms are particularly potent, since they may reproduce themselves among systems and thus shut down an entire network. Such an event occurred in 1988 to UNIX systems on the Internet, causing the loss of system and system-administrator time worth millions of dollars.

At the close of the workday on November 2, 1988, Robert Tappan Morris, Jr., a first-year Cornell graduate student, unleashed a worm program on one or more hosts connected to the Internet. Targeting Sun Microsystems' Sun 3 workstations and VAX computers running variants of Version 4 BSD UNIX, the worm quickly spread over great distances. Within a few hours of its release, it had consumed system resources to the point of bringing down the infected machines.

Although Morris designed the self-replicating program for rapid reproduction and distribution, some of the features of the UNIX networking environment provided the means to propagate the worm throughout the system. It is likely that Morris chose for initial infection an Internet host left open for and accessible to outside users. From there, the worm program exploited flaws in the UNIX operating system's security routines and took advantage of UNIX utilities that simplify resource sharing in local-area networks to gain unauthorized access to thousands of other connected sites. Morris's methods of attack are outlined next.

The worm was made up of two programs, a **grappling hook** (also called a **bootstrap** or **vector**) program and the main program. Named 11.c, the grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. Once established on the computer system under attack, the grappling hook connected to the machine where it originated and uploaded a copy of the main worm onto the **hooked** system (Figure 14.6). The main program proceeded to search for other machines to which the newly infected system could connect easily. In these actions, Morris exploited the UNIX networking utility **rsh** for easy remote task execution. By setting up special files that list host–login name pairs, users can omit entering a password each time

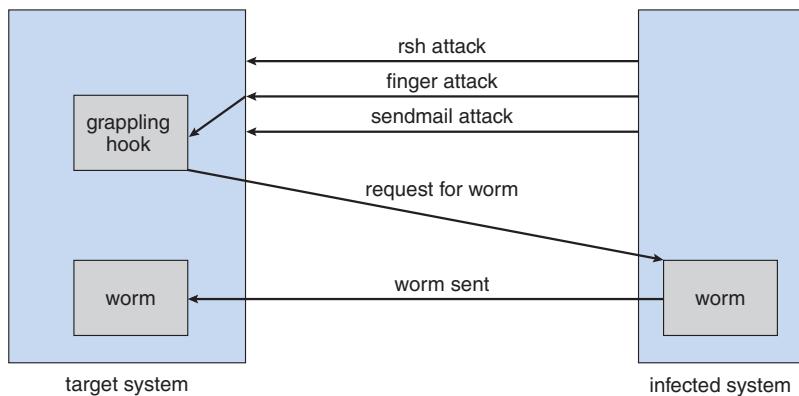


Figure 14.6 The Morris Internet worm.

they access a remote account on the paired list. The worm searched these special files for site names that would allow remote execution without a password. Where remote shells were established, the worm program was uploaded and began executing anew.

The attack via remote access was one of three infection methods built into the worm. The other two methods involved operating-system bugs in the UNIX `finger` and `sendmail` programs.

The `finger` utility functions as an electronic telephone directory. The command

```
finger user-name@hostname
```

returns a person's real and login names along with other information that the user may have provided, such as office and home address and telephone number, research plan, or clever quotation. `Finger` runs as a background process (or daemon) at each BSD site and responds to queries throughout the Internet. The worm executed a buffer-overflow attack on `finger`. The program queried `finger` with a 536-byte string crafted to exceed the buffer allocated for input and to overwrite the stack frame. Instead of returning to the main routine where it resided before Morris's call, the `finger` daemon was routed to a procedure within the invading 536-byte string now residing on the stack. The new procedure executed `/bin/sh`, which, if successful, gave the worm a remote shell on the machine under attack.

The bug exploited in `sendmail` also involved using a daemon process for malicious entry. `sendmail` sends, receives, and routes electronic mail. Debugging code in the utility permits testers to verify and display the state of the mail system. The debugging option was useful to system administrators and was often left on. Morris included in his attack arsenal a call to debug that—instead of specifying a user address, as would be normal in testing—issued a set of commands that mailed and executed a copy of the grappling-hook program.

Once in place, the main worm systematically attempted to discover user passwords. It began by trying simple cases of no password or passwords constructed of account–user-name combinations, then used comparisons with an internal dictionary of 432 favorite password choices, and then went to the

final stage of trying each word in the standard UNIX on-line dictionary as a possible password. This elaborate and efficient three-stage password-cracking algorithm enabled the worm to gain access to other user accounts on the infected system. The worm then searched for `rsh` data files in these newly broken accounts and used them as described previously to gain access to user accounts on remote systems.

With each new access, the worm program searched for already active copies of itself. If it found one, the new copy exited, except in every seventh instance. Had the worm exited on all duplicate sightings, it might have remained undetected. Allowing every seventh duplicate to proceed (possibly to confound efforts to stop its spread by baiting with “fake” worms) created a wholesale infestation of Sun and VAX systems on the Internet.

The very features of the UNIX network environment that assisted in the worm’s propagation also helped to stop its advance. Ease of electronic communication, mechanisms to copy source and binary files to remote machines, and access to both source code and human expertise allowed cooperative efforts to develop solutions quickly. By the evening of the next day, November 3, methods of halting the invading program were circulated to system administrators via the Internet. Within days, specific software patches for the exploited security flaws were available.

Why did Morris unleash the worm? The action has been characterized as both a harmless prank gone awry and a serious criminal offense. Based on the complexity of the attack, it is unlikely that the worm’s release or the scope of its spread was unintentional. The worm program took elaborate steps to cover its tracks and to repel efforts to stop its spread. Yet the program contained no code aimed at damaging or destroying the systems on which it ran. The author clearly had the expertise to include such commands; in fact, data structures were present in the bootstrap code that could have been used to transfer Trojan-horse or virus programs. The behavior of the program may lead to interesting observations, but it does not provide a sound basis for inferring motive. What is not open to speculation, however, is the legal outcome: a federal court convicted Morris and handed down a sentence of three years’ probation, 400 hours of community service, and a \$10,000 fine. Morris’s legal costs probably exceeded \$100,000.

Security experts continue to evaluate methods to decrease or eliminate worms. A more recent event, though, shows that worms are still a fact of life on the Internet. It also shows that as the Internet grows, the damage that even “harmless” worms can do also grows and can be significant. This example occurred during August 2003. The fifth version of the “Sobig” worm, more properly known as “W32.Sobig.F@mm,” was released by persons at this time unknown. It was the fastest-spreading worm released to date, at its peak infecting hundreds of thousands of computers and one in seventeen e-mail messages on the Internet. It clogged e-mail inboxes, slowed networks, and took a huge number of hours to clean up.

Sobig.F was launched by being uploaded to a pornography newsgroup via an account created with a stolen credit card. It was disguised as a photo. The virus targeted Microsoft Windows systems and used its own SMTP engine to e-mail itself to all the addresses found on an infected system. It used a variety of subject lines to help avoid detection, including “Thank You!” “Your details,”

and “Re: Approved.” It also used a random address on the host as the “From:” address, making it difficult to determine from the message which machine was the infected source. Sobig.F included an attachment for the target e-mail reader to click on, again with a variety of names. If this payload was executed, it stored a program called WINPPR32.EXE in the default Windows directory, along with a text file. It also modified the Windows registry.

The code included in the attachment was also programmed to periodically attempt to connect to one of twenty servers and download and execute a program from them. Fortunately, the servers were disabled before the code could be downloaded. The content of the program from these servers has not yet been determined. If the code was malevolent, untold damage to a vast number of machines could have resulted.

14.3.2 Port Scanning

Port scanning is not an attack but rather a means for a cracker to detect a system’s vulnerabilities to attack. Port scanning typically is automated, involving a tool that attempts to create a TCP/IP connection to a specific port or a range of ports. For example, suppose there is a known vulnerability (or bug) in `sendmail`. A cracker could launch a port scanner to try to connect, say, to port 25 of a particular system or to a range of systems. If the connection was successful, the cracker (or tool) could attempt to communicate with the answering service to determine if the service was indeed `sendmail` and, if so, if it was the version with the bug.

Now imagine a tool in which each bug of every service of every operating system was encoded. The tool could attempt to connect to every port of one or more systems. For every service that answered, it could try to use each known bug. Frequently, the bugs are buffer overflows, allowing the creation of a privileged command shell on the system. From there, of course, the cracker could install Trojan horses, back-door programs, and so on.

There is no such tool, but there are tools that perform subsets of that functionality. For example, `nmap` (from <http://www.insecure.org/nmap/>) is a very versatile open-source utility for network exploration and security auditing. When pointed at a target, it will determine what services are running, including application names and versions. It can identify the host operating system. It can also provide information about defenses, such as what firewalls are defending the target. It does not exploit any known bugs.

Because port scans are detectable (Section 14.6.3), they frequently are launched from **zombie systems**. Such systems are previously compromised, independent systems that are serving their owners while being used for nefarious purposes, including denial-of-service attacks and spam relay. Zombies make crackers particularly difficult to prosecute because determining the source of the attack and the person that launched it is challenging. This is one of many reasons for securing “inconsequential” systems, not just systems containing “valuable” information or services.

14.3.3 Denial of Service

As mentioned earlier, denial-of-service attacks are aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. Most such attacks involve systems that the attacker has

not penetrated. Launching an attack that prevents legitimate use is frequently easier than breaking into a machine or facility.

Denial-of-service attacks are generally network based. They fall into two categories. Attacks in the first category use so many facility resources that, in essence, no useful work can be done. For example, a website click could download a Java applet that proceeds to use all available CPU time or to pop up windows infinitely. The second category involves disrupting the network of the facility. There have been several successful denial-of-service attacks of this kind against major websites. These attacks result from abuse of some of the fundamental functionality of TCP/IP. For instance, if the attacker sends the part of the protocol that says “I want to start a TCP connection,” but never follows with the standard “The connection is now complete,” the result can be partially started TCP sessions. If enough of these sessions are launched, they can eat up all the network resources of the system, disabling any further legitimate TCP connections. Such attacks, which can last hours or days, have caused partial or full failure of attempts to use the target facility. The attacks are usually stopped at the network level until the operating systems can be updated to reduce their vulnerability.

Generally, it is impossible to prevent denial-of-service attacks. The attacks use the same mechanisms as normal operation. Even more difficult to prevent and resolve are **distributed denial-of-service (DDOS)** attacks. These attacks are launched from multiple sites at once, toward a common target, typically by zombies. DDOS attacks have become more common and are sometimes associated with blackmail attempts. A site comes under attack, and the attackers offer to halt the attack in exchange for money.

Sometimes a site does not even know it is under attack. It can be difficult to determine whether a system slowdown is an attack or just a surge in system use. Consider that a successful advertising campaign that greatly increases traffic to a site could be considered a DDOS.

There are other interesting aspects of DOS attacks. For example, if an authentication algorithm locks an account for a period of time after several incorrect attempts to access the account, then an attacker could cause all authentication to be blocked by purposely making incorrect attempts to access all accounts. Similarly, a firewall that automatically blocks certain kinds of traffic could be induced to block that traffic when it should not. These examples suggest that programmers and systems managers need to fully understand the algorithms and technologies they are deploying. Finally, computer science classes are notorious sources of accidental system DOS attacks. Consider the first programming exercises in which students learn to create subprocesses or threads. A common bug involves spawning subprocesses infinitely. The system’s free memory and CPU resources don’t stand a chance.

14.4 Cryptography as a Security Tool

There are many defenses against computer attacks, running the gamut from methodology to technology. The broadest tool available to system designers and users is cryptography. In this section, we discuss cryptography and its use in computer security. Note that the cryptography discussed here has been simplified for educational purposes; readers are cautioned against using any

of the schemes described here in the real world. Good cryptography libraries are widely available and would make a good basis for production applications.

In an isolated computer, the operating system can reliably determine the sender and recipient of all interprocess communication, since it controls all communication channels in the computer. In a network of computers, the situation is quite different. A networked computer receives bits “from the wire” with no immediate and reliable way of determining what machine or application sent those bits. Similarly, the computer sends bits onto the network with no way of knowing who might eventually receive them. Additionally, when either sending or receiving, the system has no way of knowing if an eavesdropper listened to the communication.

Commonly, network addresses are used to infer the potential senders and receivers of network messages. Network packets arrive with a source address, such as an IP address. And when a computer sends a message, it names the intended receiver by specifying a destination address. However, for applications where security matters, we are asking for trouble if we assume that the source or destination address of a packet reliably determines who sent or received that packet. A rogue computer can send a message with a falsified source address, and numerous computers other than the one specified by the destination address can (and typically do) receive a packet. For example, all of the routers on the way to the destination will receive the packet, too. How, then, is an operating system to decide whether to grant a request when it cannot trust the named source of the request? And how is it supposed to provide protection for a request or data when it cannot determine who will receive the response or message contents it sends over the network?

It is generally considered infeasible to build a network of any scale in which the source and destination addresses of packets can be *trusted* in this sense. Therefore, the only alternative is somehow to eliminate the need to trust the network. This is the job of cryptography. Abstractly, **cryptography** is used to constrain the potential senders and/or receivers of a message. Modern cryptography is based on secrets called **keys** that are selectively distributed to computers in a network and used to process messages. Cryptography enables a recipient of a message to verify that the message was created by some computer possessing a certain key. Similarly, a sender can encode its message so that only a computer with a certain key can decode the message. Unlike network addresses, however, keys are designed so that it is not computationally feasible to derive them from the messages they were used to generate or from any other public information. Thus, they provide a much more trustworthy means of constraining senders and receivers of messages. Note that cryptography is a field of study unto itself, with large and small complexities and subtleties. Here, we explore the most important aspects of the parts of cryptography that pertain to operating systems.

14.4.1 Encryption

Because it solves a wide variety of communication security problems, **encryption** is used frequently in many aspects of modern computing. It is used to send messages securely across a network, as well as to protect database data, files, and even entire disks from having their contents read by unauthorized entities. An encryption algorithm enables the sender of a message to ensure that

only a computer possessing a certain key can read the message, or ensure that the writer of data is the only reader of that data. Encryption of messages is an ancient practice, of course, and there have been many encryption algorithms, dating back to ancient times. In this section, we describe important modern encryption principles and algorithms.

An encryption algorithm consists of the following components:

- A set K of keys.
- A set M of messages.
- A set C of ciphertexts.
- An encrypting function $E : K \rightarrow (M \rightarrow C)$. That is, for each $k \in K$, E_k is a function for generating ciphertexts from messages. Both E and E_k for any k should be efficiently computable functions. Generally, E_k is a randomized mapping from messages to ciphertexts.
- A decrypting function $D : K \rightarrow (C \rightarrow M)$. That is, for each $k \in K$, D_k is a function for generating messages from ciphertexts. Both D and D_k for any k should be efficiently computable functions.

An encryption algorithm must provide this essential property: given a ciphertext $c \in C$, a computer can compute m such that $E_k(m) = c$ only if it possesses k . Thus, a computer holding k can decrypt ciphertexts to the plaintexts used to produce them, but a computer not holding k cannot decrypt ciphertexts. Since ciphertexts are generally exposed (for example, sent on a network), it is important that it be infeasible to derive k from the ciphertexts.

There are two main types of encryption algorithms: symmetric and asymmetric. We discuss both types in the following sections.

14.4.1.1 Symmetric Encryption

In a **symmetric encryption algorithm**, the same key is used to encrypt and to decrypt. Therefore, the secrecy of k must be protected. Figure 14.7 shows an example of two users communicating securely via symmetric encryption over an insecure channel. Note that the key exchange can take place directly between the two parties or via a trusted third party (that is, a certificate authority), as discussed in Section 14.4.1.4.

For the past several decades, the most commonly used symmetric encryption algorithm in the United States for civilian applications has been the **data-encryption standard (DES)** cipher adopted by the National Institute of Standards and Technology (NIST). DES works by taking a 64-bit value and a 56-bit key and performing a series of transformations that are based on substitution and permutation operations. Because DES works on a block of bits at a time, is known as a **block cipher**, and its transformations are typical of block ciphers. With block ciphers, if the same key is used for encrypting an extended amount of data, it becomes vulnerable to attack.

DES is now considered insecure for many applications because its keys can be exhaustively searched with moderate computing resources. (Note, though, that it is still frequently used.) Rather than giving up on DES, NIST created a modification called **triple DES**, in which the DES algorithm is repeated three times (two encryptions and one decryption) on the same plaintext using two

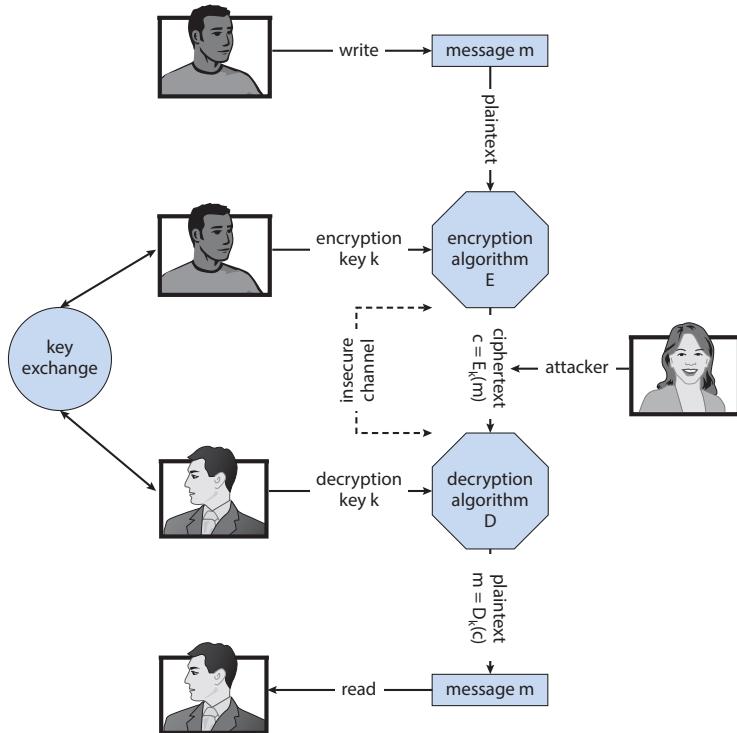


Figure 14.7 A secure communication over an insecure medium.

or three keys—for example, $c = E_{k3}(D_{k2}(E_{k1}(m)))$. When three keys are used, the effective key length is 168 bits. Triple DES is in widespread use today.

In 2001, NIST adopted a new block cipher, called the **advanced encryption standard (AES)**, to replace DES. AES is another block cipher. It can use key lengths of 128, 192, or 256 bits and works on 128-bit blocks. Generally, the algorithm is compact and efficient.

Block ciphers are not in themselves secure encryption schemes. In particular, they do not directly handle messages longer than their required block sizes. However, there are many **modes of encryption** that are based on stream ciphers, which can be used to securely encrypt longer messages.

RC4 is perhaps the most common stream cipher. A **stream cipher** is designed to encrypt and decrypt a stream of bytes or bits rather than a block. This is useful when the length of a communication would make a block cipher too slow. The key is input into a pseudo-random-bit generator, which is an algorithm that attempts to produce random bits. The output of the generator when fed a key is a keystream. A **keystream** is an infinite set of bits that can be used to encrypt a plaintext stream by simply XORing it with the plaintext. (XOR, for “eXclusive OR” is an operation that compares two input bits and generates one output bit. If the bits are the same, the result is 0. If the bits are different, the result is 1.) RC4 is used in encrypting streams of data, such as in WEP, the wireless LAN protocol. Unfortunately, RC4 as used in WEP (IEEE standard 802.11) has been found to be breakable in a reasonable amount of computer time. In fact, RC4 itself has vulnerabilities.

14.4.1.2 Asymmetric Encryption

In an **asymmetric encryption algorithm**, there are different encryption and decryption keys. An entity preparing to receive encrypted communication creates two keys and makes one of them (called the public key) available to anyone who wants it. Any sender can use that key to encrypt a communication, but only the key creator can decrypt the communication. This scheme, known as **public-key encryption**, was a breakthrough in cryptography. No longer must a key be kept secret and delivered securely. Instead, anyone can encrypt a message to the receiving entity, and no matter who else is listening, only that entity can decrypt the message.

As an example of how public-key encryption works, we describe an algorithm known as **RSA**, after its inventors, Rivest, Shamir, and Adleman. RSA is the most widely used asymmetric encryption algorithm. (Asymmetric algorithms based on elliptic curves are gaining ground, however, because the key length of such an algorithm can be shorter for the same amount of cryptographic strength.)

In RSA, k_e is the **public key**, and k_d is the **private key**. N is the product of two large, randomly chosen prime numbers p and q (for example, p and q are 512 bits each). It must be computationally infeasible to derive $k_{d,N}$ from $k_{e,N}$, so that k_e need not be kept secret and can be widely disseminated. The encryption algorithm is $E_{k_e,N}(m) = m^{k_e} \bmod N$, where k_e satisfies $k_e k_d \bmod (p-1)(q-1) = 1$. The decryption algorithm is then $D_{k_d,N}(c) = c^{k_d} \bmod N$.

An example using small values is shown in Figure 14.8. In this example, we make $p = 7$ and $q = 13$. We then calculate $N = 7*13 = 91$ and $(p-1)(q-1) = 72$. We next select k_e relatively prime to 72 and < 72, yielding 5. Finally, we calculate k_d such that $k_e k_d \bmod 72 = 1$, yielding 29. We now have our keys: the public key, $k_{e,N} = 5, 91$, and the private key, $k_{d,N} = 29, 91$. Encrypting the message 69 with the public key results in the message 62, which is then decoded by the receiver via the private key.

The use of asymmetric encryption begins with the publication of the public key of the destination. For bidirectional communication, the source also must publish its public key. “Publication” can be as simple as handing over an electronic copy of the key, or it can be more complex. The private key (or “secret key”) must be zealously guarded, as anyone holding that key can decrypt any message created by the matching public key.

We should note that the seemingly small difference in key use between asymmetric and symmetric cryptography is quite large in practice. Asymmetric cryptography is much more computationally expensive to execute. It is much faster for a computer to encode and decode ciphertext by using the usual symmetric algorithms than by using asymmetric algorithms. Why, then, use an asymmetric algorithm? In truth, these algorithms are not used for general-purpose encryption of large amounts of data. However, they are used not only for encryption of small amounts of data but also for authentication, confidentiality, and key distribution, as we show in the following sections.

14.4.1.3 Authentication

We have seen that encryption offers a way of constraining the set of possible receivers of a message. Constraining the set of potential senders of a message is called **authentication**. Authentication is thus complementary to encryption.

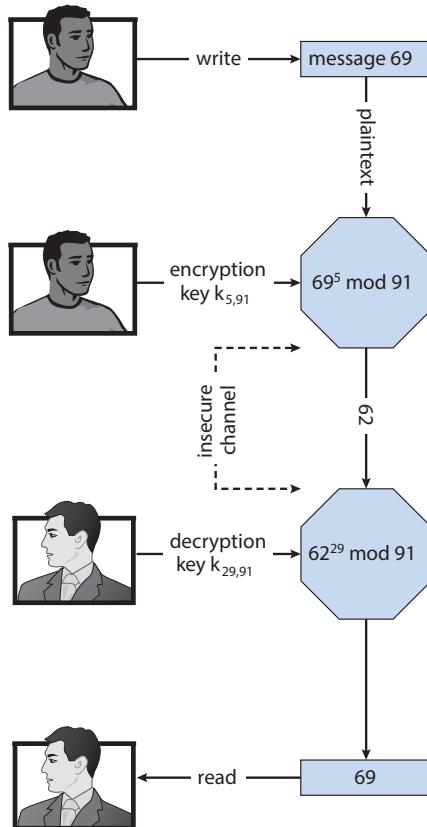


Figure 14.8 Encryption and decryption using RSA asymmetric cryptography.

Authentication is also useful for proving that a message has not been modified. In this section, we discuss authentication as a constraint on possible senders of a message. Note that this sort of authentication is similar to but distinct from user authentication, which we discuss in Section 14.5.

An authentication algorithm using symmetric keys consists of the following components:

- A set K of keys.
- A set M of messages.
- A set A of authenticators.
- A function $S : K \rightarrow (M \rightarrow A)$. That is, for each $k \in K$, S_k is a function for generating authenticators from messages. Both S and S_k for any k should be efficiently computable functions.
- A function $V : K \rightarrow (M \times A \rightarrow \{\text{true}, \text{false}\})$. That is, for each $k \in K$, V_k is a function for verifying authenticators on messages. Both V and V_k for any k should be efficiently computable functions.

The critical property that an authentication algorithm must possess is this: for a message m , a computer can generate an authenticator $a \in A$ such that $V_k(m, a) = \text{true}$ only if it possesses k . Thus, a computer holding k can

generate authenticators on messages so that any computer possessing k can verify them. However, a computer not holding k cannot generate authenticators on messages that can be verified using V_k . Since authenticators are generally exposed (for example, sent on a network with the messages themselves), it must not be feasible to derive k from the authenticators. Practically, if $V_k(m, a) = \text{true}$, then we know that m has not been modified, and that the sender of the message has k . If we share k with only one entity, then we know that the message originated from k .

Just as there are two types of encryption algorithms, there are two main varieties of authentication algorithms. The first step in understanding these algorithms is to explore hash functions. A **hash function** $H(m)$ creates a small, fixed-sized block of data, known as a **message digest** or **hash value**, from a message m . Hash functions work by taking a message, splitting it into blocks, and processing the blocks to produce an n -bit hash. H must be collision resistant—that is, it must be infeasible to find an $m' \neq m$ such that $H(m) = H(m')$. Now, if $H(m) = H(m')$, we know that $m = m'$ —that is, we know that the message has not been modified. Common message-digest functions include **MD5**, now considered insecure, which produces a 128-bit hash, and **SHA-1**, which outputs a 160-bit hash. Message digests are useful for detecting changed messages but are not useful as authenticators. For example, $H(m)$ can be sent along with a message; but if H is known, then someone could modify m to m' and recompute $H(m')$, and the message modification would not be detected. Therefore, we must authenticate $H(m)$.

The first main type of authentication algorithm uses symmetric encryption. In a **message-authentication code (MAC)**, a cryptographic checksum is generated from the message using a secret key. A MAC provides a way to securely authenticate short values. If we use it to authenticate $H(m)$ for an H that is collision resistant, then we obtain a way to securely authenticate long messages by hashing them first. Note that k is needed to compute both S_k and V_k , so anyone able to compute one can compute the other.

The second main type of authentication algorithm is a **digital-signature algorithm**, and the authenticators thus produced are called **digital signatures**. Digital signatures are very useful in that they enable *anyone* to verify the authenticity of the message. In a digital-signature algorithm, it is computationally infeasible to derive k_s from k_v . Thus, k_v is the public key, and k_s is the private key.

Consider as an example the RSA digital-signature algorithm. It is similar to the RSA encryption algorithm, but the key use is reversed. The digital signature of a message is derived by computing $S_{k_s}(m) = H(m)^{k_s} \bmod N$. The key k_s again is a pair $\langle d, N \rangle$, where N is the product of two large, randomly chosen prime numbers p and q . The verification algorithm is then $V_{k_v}(m, a) \stackrel{?}{=} a^{k_v} \bmod N = H(m)$, where k_v satisfies $k_v k_s \bmod (p-1)(q-1) = 1$.

Note that encryption and authentication may be used together or separately. Sometimes, for instance, we want authentication but not confidentiality. For example, a company could provide a software patch and could “sign” that patch to prove that it came from the company and that it hasn’t been modified.

Authentication is a component of many aspects of security. For example, digital signatures are the core of **nonrepudiation**, which supplies proof that an entity performed an action. A typical example of nonrepudiation involves

the filling out of electronic forms as an alternative to the signing of paper contracts. Nonrepudiation assures that a person filling out an electronic form cannot deny that he did so.

14.4.1.4 Key Distribution

Certainly, a good part of the battle between cryptographers (those inventing ciphers) and cryptanalysts (those trying to break them) involves keys. With symmetric algorithms, both parties need the key, and no one else should have it. The delivery of the symmetric key is a huge challenge. Sometimes it is performed **out-of-band**—say, via a paper document or a conversation. These methods do not scale well, however. Also consider the key-management challenge. Suppose a user wanted to communicate with N other users privately. That user would need N keys and, for more security, would need to change those keys frequently.

These are the very reasons for efforts to create asymmetric key algorithms. Not only can the keys be exchanged in public, but a given user needs only one private key, no matter how many other people she wants to communicate with. There is still the matter of managing a public key for each recipient of the communication, but since public keys need not be secured, simple storage can be used for that **key ring**.

Unfortunately, even the distribution of public keys requires some care. Consider the man-in-the-middle attack shown in Figure 14.9. Here, the person who wants to receive an encrypted message sends out his public key, but an attacker also sends her “bad” public key (which matches her private key). The person who wants to send the encrypted message knows no better and so uses the bad key to encrypt the message. The attacker then happily decrypts it.

The problem is one of authentication—what we need is proof of who (or what) owns a public key. One way to solve that problem involves the use of digital certificates. A **digital certificate** is a public key digitally signed by a trusted party. The trusted party receives proof of identification from some entity and certifies that the public key belongs to that entity. But how do we know we can trust the certifier? These **certificate authorities** have their public keys included within web browsers (and other consumers of certificates) before they are distributed. The certificate authorities can then vouch for other authorities (digitally signing the public keys of these other authorities), and so on, creating a web of trust. The certificates can be distributed in a standard X.509 digital certificate format that can be parsed by computer. This scheme is used for secure web communication, as we discuss in Section 14.4.3.

14.4.2 Implementation of Cryptography

Network protocols are typically organized in **layers**, like an onion or a parfait, with each layer acting as a client of the one below it. That is, when one protocol generates a message to send to its protocol peer on another machine, it hands its message to the protocol below it in the network-protocol stack for delivery to its peer on that machine. For example, in an IP network, TCP (a **transport-layer** protocol) acts as a client of IP (a **network-layer** protocol): TCP packets are passed down to IP for delivery to the IP peer at the other end of the connection. IP encapsulates the TCP packet in an IP packet, which it similarly passes down to the **data-link layer** to be transmitted across the network to its peer on the

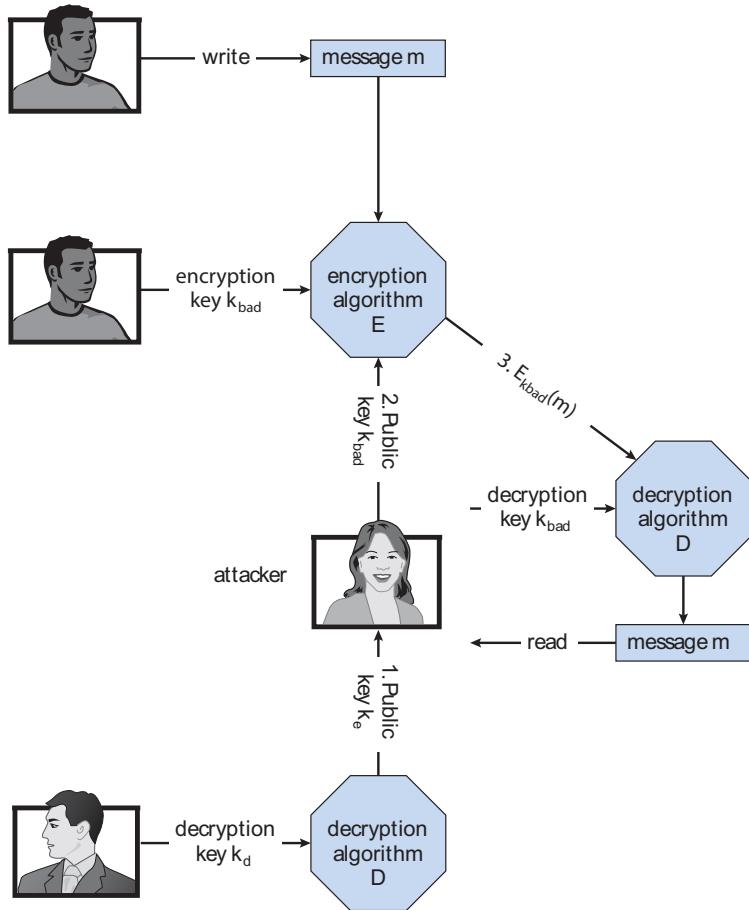


Figure 14.9 A man-in-the-middle attack on asymmetric cryptography.

destination computer. This IP peer then delivers the TCP packet up to the TCP peer on that machine.

Cryptography can be inserted at almost any layer in the OSI model. SSL (Section 14.4.3), for example, provides security at the transport layer. Network-layer security generally has been standardized on **IPSec**, which defines IP packet formats that allow the insertion of authenticators and the encryption of packet contents. IPSec uses symmetric encryption and uses the **Internet Key Exchange (IKE)** protocol for key exchange. IKE is based on public-key encryption. IPSec is becoming widely used as the basis for **virtual private networks (VPNs)**, in which all traffic between two IPSec endpoints is encrypted to make a private network out of one that may otherwise be public. Numerous protocols also have been developed for use by applications, such as PGP for encrypting e-mail, but then the applications themselves must be coded to implement security.

Where is cryptographic protection best placed in a protocol stack? In general, there is no definitive answer. On the one hand, more protocols benefit from protections placed lower in the stack. For example, since IP packets encapsulate TCP packets, encryption of IP packets (using IPSec, for example) also

hides the contents of the encapsulated TCP packets. Similarly, authenticators on IP packets detect the modification of contained TCP header information.

On the other hand, protection at lower layers in the protocol stack may give insufficient protection to higher-layer protocols. For example, an application server that accepts connections encrypted with IPSec might be able to authenticate the client computers from which requests are received. However, to authenticate a user at a client computer, the server may need to use an application-level protocol—the user may be required to type a password. Also consider the problem of e-mail. E-mail delivered via the industry-standard SMTP protocol is stored and forwarded, frequently multiple times, before it is delivered. Each of these transmissions could go over a secure or an insecure network. For e-mail to be secure, the e-mail message needs to be encrypted so that its security is independent of the transports that carry it.

14.4.3 An Example: SSL

SSL 3.0 is a cryptographic protocol that enables two computers to communicate securely—that is, so that each can limit the sender and receiver of messages to the other. It is perhaps the most commonly used cryptographic protocol on the Internet today, since it is the standard protocol by which web browsers communicate securely with web servers. For completeness, we should note that SSL was designed by Netscape and that it evolved into the industry-standard TLS protocol. In this discussion, we use SSL to mean both SSL and TLS.

SSL is a complex protocol with many options. Here, we present only a single variation of it. Even then, we describe it in a very simplified and abstract form, so as to maintain focus on its use of cryptographic primitives. What we are about to see is a complex dance in which asymmetric cryptography is used so that a client and a server can establish a secure **session key** that can be used for symmetric encryption of the session between the two—all of this while avoiding man-in-the-middle and replay attacks. For added cryptographic strength, the session keys are forgotten once a session is completed. Another communication between the two will require generation of new session keys.

The SSL protocol is initiated by a client c to communicate securely with a server. Prior to the protocol's use, the server s is assumed to have obtained a certificate, denoted cert_s , from certification authority CA. This certificate is a structure containing the following:

- Various attributes (attrs) of the server, such as its unique *distinguished* name and its *common* (DNS) name
- The identity of a asymmetric encryption algorithm $E()$ for the server
- The public key k_e of this server
- A validity interval (interval) during which the certificate should be considered valid
- A digital signature \mathbf{a} on the above information made by the CA—that is, $\mathbf{a} = \mathbf{S}_{kCA}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle)$

In addition, prior to the protocol's use, the client is presumed to have obtained the public verification algorithm V_{kCA} for CA. In the case of the Web, the user's browser is shipped from its vendor containing the verification

algorithms and public keys of certain certification authorities. The user can add or delete these as she chooses.

When c connects to s , it sends a 28-byte random value n_c to the server, which responds with a random value n_s of its own, plus its certificate cert_s . The client verifies that $V_{KCA}(\langle \text{attrs}, E_{ke}, \text{interval} \rangle, a) = \text{true}$ and that the current time is in the validity interval interval . If both of these tests are satisfied, the server has proved its identity. Then the client generates a random 46-byte **premaster secret** pms and sends $\text{cpms} = E_{ke}(\text{pms})$ to the server. The server recovers $\text{pms} = D_{kd}(\text{cpms})$. Now both the client and the server are in possession of n_c , n_s , and pms , and each can compute a shared 48-byte **master secret** $\text{ms} = H(n_c, n_s, \text{pms})$. Only the server and client can compute ms , since only they know pms . Moreover, the dependence of ms on n_c and n_s ensures that ms is a *fresh* value—that is, a session key that has not been used in a previous communication. At this point, the client and the server both compute the following keys from the ms :

- A symmetric encryption key k_{cs}^{crypt} for encrypting messages from the client to the server
- A symmetric encryption key k_{sc}^{crypt} for encrypting messages from the server to the client
- A MAC generation key k_{cs}^{mac} for generating authenticators on messages from the client to the server
- A MAC generation key k_{sc}^{mac} for generating authenticators on messages from the server to the client

To send a message m to the server, the client sends

$$c = E_{k_{cs}^{\text{crypt}}}(\langle m, S_{k_{cs}^{\text{mac}}}(m) \rangle).$$

Upon receiving c , the server recovers

$$\langle m, a \rangle = D_{k_{cs}^{\text{crypt}}}(c)$$

and accepts m if $V_{k_{cs}^{\text{mac}}}(m, a) = \text{true}$. Similarly, to send a message m to the client, the server sends

$$c = E_{k_{sc}^{\text{crypt}}}(\langle m, S_{k_{sc}^{\text{mac}}}(m) \rangle)$$

and the client recovers

$$\langle m, a \rangle = D_{k_{sc}^{\text{crypt}}}(c)$$

and accepts m if $V_{k_{sc}^{\text{mac}}}(m, a) = \text{true}$.

This protocol enables the server to limit the recipients of its messages to the client that generated pms and to limit the senders of the messages it accepts to that same client. Similarly, the client can limit the recipients of the messages it sends and the senders of the messages it accepts to the party that knows k_d (that is, the party that can decrypt cpms). In many applications, such as web transactions, the client needs to verify the identity of the party that knows k_d . This is one purpose of the certificate cert_s . In particular, the attrs field contains information that the client can use to determine the identity—for example, the

domain name—of the server with which it is communicating. For applications in which the server also needs information about the client, SSL supports an option by which a client can send a certificate to the server.

In addition to its use on the Internet, SSL is being used for a wide variety of tasks. For example, IPSec VPNs now have a competitor in SSL VPNs. IPSec is good for point-to-point encryption of traffic—say, between two company offices. SSL VPNs are more flexible but not as efficient, so they might be used between an individual employee working remotely and the corporate office.

14.5 User Authentication

Our earlier discussion of authentication involves messages and sessions. But what about users? If a system cannot authenticate a user, then authenticating that a message came from that user is pointless. Thus, a major security problem for operating systems is **user authentication**. The protection system depends on the ability to identify the programs and processes currently executing, which in turn depends on the ability to identify each user of the system. Users normally identify themselves. How do we determine whether a user's identity is authentic? Generally, user authentication is based on one or more of three things: the user's possession of something (a key or card), the user's knowledge of something (a user identifier and password), or an attribute of the user (fingerprint, retina pattern, or signature).

14.5.1 Passwords

The most common approach to authenticating a user identity is the use of **passwords**. When the user identifies herself by user ID or account name, she is asked for a password. If the user-supplied password matches the password stored in the system, the system assumes that the account is being accessed by the owner of that account.

Passwords are often used to protect objects in the computer system, in the absence of more complete protection schemes. They can be considered a special case of either keys or capabilities. For instance, a password may be associated with each resource (such as a file). Whenever a request is made to use the resource, the password must be given. If the password is correct, access is granted. Different passwords may be associated with different access rights. For example, different passwords may be used for reading files, appending files, and updating files.

In practice, most systems require only one password for a user to gain full rights. Although more passwords theoretically would be more secure, such systems tend not to be implemented due to the classic trade-off between security and convenience. If security makes something inconvenient, then the security is frequently bypassed or otherwise circumvented.

14.5.2 Password Vulnerabilities

Passwords are extremely common because they are easy to understand and use. Unfortunately, passwords can often be guessed, accidentally exposed, sniffed (read by an eavesdropper), or illegally transferred from an authorized user to an unauthorized one, as we show next.

There are two common ways to guess a password. One way is for the intruder (either human or program) to know the user or to have information about the user. All too frequently, people use obvious information (such as the names of their cats or spouses) as their passwords. The other way is to use brute force, trying enumeration—or all possible combinations of valid password characters (letters, numbers, and punctuation on some systems)—until the password is found. Short passwords are especially vulnerable to this method. For example, a four-character password provides only 10,000 variations. On average, guessing 5,000 times would produce a correct hit. A program that could try a password every millisecond would take only about 5 seconds to guess a four-character password. Enumeration is less successful where systems allow longer passwords that include both uppercase and lowercase letters, along with numbers and all punctuation characters. Of course, users must take advantage of the large password space and must not, for example, use only lowercase letters.

In addition to being guessed, passwords can be exposed as a result of visual or electronic monitoring. An intruder can look over the shoulder of a user (**shoulder surfing**) when the user is logging in and can learn the password easily by watching the keyboard. Alternatively, anyone with access to the network on which a computer resides can seamlessly add a network monitor, allowing him to **sniff**, or watch, all data being transferred on the network, including user IDs and passwords. Encrypting the data stream containing the password solves this problem. Even such a system could have passwords stolen, however. For example, if a file is used to contain the passwords, it could be copied for off-system analysis. Or consider a Trojan-horse program installed on the system that captures every keystroke before sending it on to the application.

Exposure is a particularly severe problem if the password is written down where it can be read or lost. Some systems force users to select hard-to-remember or long passwords, or to change their password frequently, which may cause a user to record the password or to reuse it. As a result, such systems provide much less security than systems that allow users to select easy passwords!

The final type of password compromise, illegal transfer, is the result of human nature. Most computer installations have a rule that forbids users to share accounts. This rule is sometimes implemented for accounting reasons but is often aimed at improving security. For instance, suppose one user ID is shared by several users, and a security breach occurs from that user ID. It is impossible to know who was using the ID at the time the break occurred or even whether the user was an authorized one. With one user per user ID, any user can be questioned directly about use of the account; in addition, the user might notice something different about the account and detect the break-in. Sometimes, users break account-sharing rules to help friends or to circumvent accounting, and this behavior can result in a system's being accessed by unauthorized users—possibly harmful ones.

Passwords can be either generated by the system or selected by a user. System-generated passwords may be difficult to remember, and thus users may write them down. As mentioned, however, user-selected passwords are often easy to guess (the user's name or favorite car, for example). Some systems will check a proposed password for ease of guessing or cracking before accepting

it. Some systems also *age* passwords, forcing users to change their passwords at regular intervals (every three months, for instance). This method is not foolproof either, because users can easily toggle between two passwords. The solution, as implemented on some systems, is to record a password history for each user. For instance, the system could record the last N passwords and not allow their reuse.

Several variants on these simple password schemes can be used. For example, the password can be changed more frequently. At the extreme, the password is changed from session to session. A new password is selected (either by the system or by the user) at the end of each session, and that password must be used for the next session. In such a case, even if a password is used by an unauthorized person, that person can use it only once. When the legitimate user tries to use a now-invalid password at the next session, he discovers the security violation. Steps can then be taken to repair the breached security.

14.5.3 Securing Passwords

One problem with all these approaches is the difficulty of keeping the password secret within the computer. How can the system store a password securely yet allow its use for authentication when the user presents her password? The UNIX system uses secure hashing to avoid the necessity of keeping its password list secret. Because the list is hashed rather than encrypted, it is impossible for the system to decrypt the stored value and determine the original password.

Here's how this system works. Each user has a password. The system contains a function that is extremely difficult—the designers hope impossible—to invert but is simple to compute. That is, given a value x , it is easy to compute the hash function value $f(x)$. Given a function value $f(x)$, however, it is impossible to compute x . This function is used to encode all passwords. Only encoded passwords are stored. When a user presents a password, it is hashed and compared against the stored encoded password. Even if the stored encoded password is seen, it cannot be decoded, so the password cannot be determined. Thus, the password file does not need to be kept secret.

The flaw in this method is that the system no longer has control over the passwords. Although the passwords are hashed, anyone with a copy of the password file can run fast hash routines against it—hashing each word in a dictionary, for instance, and comparing the results against the passwords. If the user has selected a password that is also a word in the dictionary, the password is cracked. On sufficiently fast computers, or even on clusters of slow computers, such a comparison may take only a few hours. Furthermore, because UNIX systems use a well-known hashing algorithm, a cracker might keep a cache of passwords that have been cracked previously. For these reasons, systems include a “salt,” or recorded random number, in the hashing algorithm. The salt value is added to the password to ensure that if two plaintext passwords are the same, they result in different hash values. In addition, the salt value makes hashing a dictionary ineffective, because each dictionary term would need to be combined with each salt value for comparison to the stored passwords. Newer versions of UNIX also store the hashed password entries in a file readable only by the superuser. The programs that compare the hash to

the stored value are run setuid to root, so they can read this file, but other users cannot.

Another weakness in the UNIX password methods is that many UNIX systems treat only the first eight characters as significant. It is therefore extremely important for users to take advantage of the available password space. Complicating the issue further is the fact that some systems do not allow the use of dictionary words as passwords. A good technique is to generate your password by using the first letter of each word of an easily remembered phrase using both upper and lower characters with a number or punctuation mark thrown in for good measure. For example, the phrase “My mother’s name is Katherine” might yield the password “Mmn.isK!”. The password is hard to crack but easy for the user to remember. A more secure system would allow more characters in its passwords. Indeed, a system might also allow passwords to include the space character, so that a user could create a **passphrase**.

14.5.4 One-Time Passwords

To avoid the problems of password sniffing and shoulder surfing, a system can use a set of **paired passwords**. When a session begins, the system randomly selects and presents one part of a password pair; the user must supply the other part. In this system, the user is **challenged** and must **respond** with the correct answer to that challenge.

This approach can be generalized to the use of an algorithm as a password. Such algorithmic passwords are not susceptible to reuse. That is, a user can type in a password, and no entity intercepting that password will be able to reuse it. In this scheme, the system and the user share a symmetric password. The password pw is never transmitted over a medium that allows exposure. Rather, the password is used as input to the function, along with a **challenge** ch presented by the system. The user then computes the function $H(pw, ch)$. The result of this function is transmitted as the authenticator to the computer. Because the computer also knows pw and ch , it can perform the same computation. If the results match, the user is authenticated. The next time the user needs to be authenticated, another ch is generated, and the same steps ensue. This time, the authenticator is different. This **one-time password** system is one of only a few ways to prevent improper authentication due to password exposure.

One-time password systems are implemented in various ways. Commercial implementations use hardware calculators with a display or a numeric keypad. These calculators generally take the shape of a credit card, a key-chain dongle, or a USB device. Software running on computers or smartphones provides the user with $H(pw, ch)$; pw can be input by the user or generated by the calculator in synchronization with the computer. Sometimes, pw is just a **personal identification number (PIN)**. The output of any of these systems shows the one-time password. A one-time password generator that requires input by the user involves **two-factor authentication**. Two different types of components are needed in this case—for example, a one-time password generator that generates the correct response only if the PIN is valid. Two-factor authentication offers far better authentication protection than single-factor authentication because it requires “something you have” as well as “something you know.”

Another variation on one-time passwords uses a **code book**, or **one-time pad**, which is a list of single-use passwords. Each password on the list is used once and then is crossed out or erased. The commonly used S/Key system uses either a software calculator or a code book based on these calculations as a source of one-time passwords. Of course, the user must protect his code book, and it is helpful if the code book does not identify the system to which the codes are authenticators.

14.5.5 Biometrics

Yet another variation on the use of passwords for authentication involves the use of biometric measures. Palm- or hand-readers are commonly used to secure physical access—for example, access to a data center. These readers match stored parameters against what is being read from hand-reader pads. The parameters can include a temperature map, as well as finger length, finger width, and line patterns. These devices are currently too large and expensive to be used for normal computer authentication.

Fingerprint readers have become accurate and cost-effective and should become more common in the future. These devices read finger ridge patterns and convert them into a sequence of numbers. Over time, they can store a set of sequences to adjust for the location of the finger on the reading pad and other factors. Software can then scan a finger on the pad and compare its features with these stored sequences to determine if they match. Of course, multiple users can have profiles stored, and the scanner can differentiate among them. A very accurate two-factor authentication scheme can result from requiring a password as well as a user name and fingerprint scan. If this information is encrypted in transit, the system can be very resistant to spoofing or replay attack.

Multifactor authentication is better still. Consider how strong authentication can be with a USB device that must be plugged into the system, a PIN, and a fingerprint scan. Except for having to place ones finger on a pad and plug the USB into the system, this authentication method is no less convenient than that using normal passwords. Recall, though, that strong authentication by itself is not sufficient to guarantee the ID of the user. An authenticated session can still be hijacked if it is not encrypted.

14.6 Implementing Security Defenses

Just as there are myriad threats to system and network security, there are many security solutions. The solutions range from improved user education, through technology, to writing bug-free software. Most security professionals subscribe to the theory of **defense in depth**, which states that more layers of defense are better than fewer layers. Of course, this theory applies to any kind of security. Consider the security of a house without a door lock, with a door lock, and with a lock and an alarm. In this section, we look at the major methods, tools, and techniques that can be used to improve resistance to threats.

14.6.1 Security Policy

The first step toward improving the security of any aspect of computing is to have a **security policy**. Policies vary widely but generally include a statement

of what is being secured. For example, a policy might state that all outside-accessible applications must have a code review before being deployed, or that users should not share their passwords, or that all connection points between a company and the outside must have port scans run every six months. Without a policy in place, it is impossible for users and administrators to know what is permissible, what is required, and what is not allowed. The policy is a road map to security, and if a site is trying to move from less secure to more secure, it needs a map to know how to get there.

Once the security policy is in place, the people it affects should know it well. It should be their guide. The policy should also be a **living document** that is reviewed and updated periodically to ensure that it is still pertinent and still followed.

14.6.2 Vulnerability Assessment

How can we determine whether a security policy has been correctly implemented? The best way is to execute a vulnerability assessment. Such assessments can cover broad ground, from social engineering through risk assessment to port scans. **Risk assessment**, for example, attempts to value the assets of the entity in question (a program, a management team, a system, or a facility) and determine the odds that a security incident will affect the entity and decrease its value. When the odds of suffering a loss and the amount of the potential loss are known, a value can be placed on trying to secure the entity.

The core activity of most vulnerability assessments is a **penetration test**, in which the entity is scanned for known vulnerabilities. Because this book is concerned with operating systems and the software that runs on them, we concentrate on those aspects of vulnerability assessment.

Vulnerability scans typically are done at times when computer use is relatively low, to minimize their impact. When appropriate, they are done on test systems rather than production systems, because they can induce unhappy behavior from the target systems or network devices.

A scan within an individual system can check a variety of aspects of the system:

- Short or easy-to-guess passwords
- Unauthorized privileged programs, such as setuid programs
- Unauthorized programs in system directories
- Unexpectedly long-running processes
- Improper directory protections on user and system directories
- Improper protections on system data files, such as the password file, device drivers, or the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 14.2.1)
- Changes to system programs detected with checksum values
- Unexpected or hidden network daemons

Any problems found by a security scan can be either fixed automatically or reported to the managers of the system.

Networked computers are much more susceptible to security attacks than are standalone systems. Rather than attacks from a known set of access points, such as directly connected terminals, we face attacks from an unknown and large set of access points—a potentially severe security problem. To a lesser extent, systems connected to telephone lines via modems are also more exposed.

In fact, the U.S. government considers a system to be only as secure as its most far-reaching connection. For instance, a top-secret system may be accessed only from within a building also considered top-secret. The system loses its top-secret rating if any form of communication can occur outside that environment. Some government facilities take extreme security precautions. The connectors that plug a terminal into the secure computer are locked in a safe in the office when the terminal is not in use. A person must have proper ID to gain access to the building and her office, must know a physical lock combination, and must know authentication information for the computer itself to gain access to the computer—an example of multifactor authentication.

Unfortunately for system administrators and computer-security professionals, it is frequently impossible to lock a machine in a room and disallow all remote access. For instance, the Internet currently connects millions of computers and has become a mission-critical, indispensable resource for many companies and individuals. If you consider the Internet a club, then, as in any club with millions of members, there are many good members and some bad members. The bad members have many tools they can use to attempt to gain access to the interconnected computers, just as Morris did with his worm.

Vulnerability scans can be applied to networks to address some of the problems with network security. The scans search a network for ports that respond to a request. If services are enabled that should not be, access to them can be blocked, or they can be disabled. The scans then determine the details of the application listening on that port and try to determine if it has any known vulnerabilities. Testing those vulnerabilities can determine if the system is misconfigured or lacks needed patches.

Finally, though, consider the use of port scanners in the hands of a cracker rather than someone trying to improve security. These tools could help crackers find vulnerabilities to attack. (Fortunately, it is possible to detect port scans through anomaly detection, as we discuss next.) It is a general challenge to security that the same tools can be used for good and for harm. In fact, some people advocate **security through obscurity**, stating that no tools should be written to test security, because such tools can be used to find (and exploit) security holes. Others believe that this approach to security is not a valid one, pointing out, for example, that crackers could write their own tools. It seems reasonable that security through obscurity be considered one of the layers of security only so long as it is not the only layer. For example, a company could publish its entire network configuration, but keeping that information secret makes it harder for intruders to know what to attack or to determine what might be detected. Even here, though, a company assuming that such information will remain a secret has a false sense of security.

14.6.3 Intrusion Detection

Securing systems and facilities is intimately linked to intrusion detection. **Intrusion detection**, as its name suggests, strives to detect attempted or successful

intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion detection encompasses a wide array of techniques that vary on a number of axes, including the following:

- The time at which detection occurs. Detection can occur in real time (while the intrusion is occurring) or after the fact.
- The types of inputs examined to detect intrusive activity. These may include user-shell commands, process system calls, and network packet headers or contents. Some forms of intrusion might be detected only by correlating information from several such sources.
- The range of response capabilities. Simple forms of response include alerting an administrator to the potential intrusion or somehow halting the potentially intrusive activity—for example, killing a process engaged in such activity. In a sophisticated form of response, a system might transparently divert an intruder’s activity to a **honeypot**—a false resource exposed to the attacker. The resource appears real to the attacker and enables the system to monitor and gain information about the attack.

These degrees of freedom in the design space for detecting intrusions have yielded a wide range of solutions, known as **intrusion-detection systems (IDSs)** and **intrusion-prevention systems (IDPs)**. IDS systems raise an alarm when an intrusion is detected, while IDP systems act as routers, passing traffic unless an intrusion is detected (at which point that traffic is blocked).

But just what constitutes an intrusion? Defining a suitable specification of intrusion turns out to be quite difficult, and thus automatic IDSs and IDPs today typically settle for one of two less ambitious approaches. In the first, called **signature-based detection**, system input or network traffic is examined for specific behavior patterns (or **signatures**) known to indicate attacks. A simple example of signature-based detection is scanning network packets for the string /etc/passwd/ targeted for a UNIX system. Another example is virus-detection software, which scans binaries or network packets for known viruses.

The second approach, typically called **anomaly detection**, attempts through various techniques to detect anomalous behavior within computer systems. Of course, not all anomalous system activity indicates an intrusion, but the presumption is that intrusions often induce anomalous behavior. An example of anomaly detection is monitoring system calls of a daemon process to detect whether the system-call behavior deviates from normal patterns, possibly indicating that a buffer overflow has been exploited in the daemon to corrupt its behavior. Another example is monitoring shell commands to detect anomalous commands for a given user or detecting an anomalous login time for a user, either of which may indicate that an attacker has succeeded in gaining access to that user’s account.

Signature-based detection and anomaly detection can be viewed as two sides of the same coin. Signature-based detection attempts to characterize dangerous behaviors and to detect when one of these behaviors occurs, whereas anomaly detection attempts to characterize normal (or nondangerous) behaviors and to detect when something other than these behaviors occurs.

These different approaches yield IDSs and IDPs with very different properties, however. In particular, anomaly detection can find previously unknown

methods of intrusion (so-called **zero-day attacks**). Signature-based detection, in contrast, will identify only known attacks that can be codified in a recognizable pattern. Thus, new attacks that were not contemplated when the signatures were generated will evade signature-based detection. This problem is well known to vendors of virus-detection software, who must release new signatures with great frequency as new viruses are detected manually.

Anomaly detection is not necessarily superior to signature-based detection, however. Indeed, a significant challenge for systems that attempt anomaly detection is to benchmark “normal” system behavior accurately. If the system has already been penetrated when it is benchmarked, then the intrusive activity may be included in the “normal” benchmark. Even if the system is benchmarked cleanly, without influence from intrusive behavior, the benchmark must give a fairly complete picture of normal behavior. Otherwise, the number of **false positives** (false alarms) or, worse, **false negatives** (missed intrusions) will be excessive.

To illustrate the impact of even a marginally high rate of false alarms, consider an installation consisting of a hundred UNIX workstations from which security-relevant events are recorded for purposes of intrusion detection. A small installation such as this could easily generate a million audit records per day. Only one or two might be worthy of an administrator’s investigation. If we suppose, optimistically, that each actual attack is reflected in ten audit records, we can roughly compute the rate of occurrence of audit records reflecting truly intrusive activity as follows:

$$\frac{2 \frac{\text{intrusions}}{\text{day}} \cdot 10 \frac{\text{records}}{\text{intrusion}}}{10^6 \frac{\text{records}}{\text{day}}} = 0.00002.$$

Interpreting this as a “probability of occurrence of intrusive records,” we denote it as $P(I)$; that is, event I is the occurrence of a record reflecting truly intrusive behavior. Since $P(I) = 0.00002$, we also know that $P(\neg I) = 1 - P(I) = 0.99998$. Now we let A denote the raising of an alarm by an IDS. An accurate IDS should maximize both $P(I|A)$ and $P(\neg I|\neg A)$ —that is, the probabilities that an alarm indicates an intrusion and that no alarm indicates no intrusion. Focusing on $P(I|A)$ for the moment, we can compute it using **Bayes’ theorem**:

$$\begin{aligned} P(I|A) &= \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)} \\ &= \frac{0.00002 \cdot P(A|I)}{0.00002 \cdot P(A|I) + 0.99998 \cdot P(A|\neg I)} \end{aligned}$$

Now consider the impact of the false-alarm rate $P(A|\neg I)$ on $P(I|A)$. Even with a very good true-alarm rate of $P(A|I) = 0.8$, a seemingly good false-alarm rate of $P(A|\neg I) = 0.0001$ yields $P(I|A) \approx 0.14$. That is, fewer than one in every seven alarms indicates a real intrusion! In systems where a security administrator investigates each alarm, a high rate of false alarms—called a “Christmas tree effect”—is exceedingly wasteful and will quickly teach the administrator to ignore alarms.

This example illustrates a general principle for IDSs and IDPs: for usability, they must offer an extremely low false-alarm rate. Achieving a sufficiently low false-alarm rate is an especially serious challenge for anomaly-detection systems, as mentioned, because of the difficulties of adequately benchmarking normal system behavior. However, research continues to improve anomaly-detection techniques. Intrusion detection software is evolving to implement signatures, anomaly algorithms, and other algorithms and to combine the results to arrive at a more accurate anomaly-detection rate.

14.6.4 Virus Protection

As we have seen, viruses can and do wreak havoc on systems. Protection from viruses thus is an important security concern. Antivirus programs are often used to provide this protection. Some of these programs are effective against only particular known viruses. They work by searching all the programs on a system for the specific pattern of instructions known to make up the virus. When they find a known pattern, they remove the instructions, **disinfecting** the program. Antivirus programs may have catalogs of thousands of viruses for which they search.

Both viruses and antivirus software continue to become more sophisticated. Some viruses modify themselves as they infect other software to avoid the basic pattern-match approach of antivirus programs. Antivirus programs in turn now look for families of patterns rather than a single pattern to identify a virus. In fact, some antivirus programs implement a variety of detection algorithms. They can decompress compressed viruses before checking for a signature. Some also look for process anomalies. A process opening an executable file for writing is suspicious, for example, unless it is a compiler. Another popular technique is to run a program in a **sandbox**, which is a controlled or emulated section of the system. The antivirus software analyzes the behavior of the code in the sandbox before letting it run unmonitored. Some antivirus programs also put up a complete shield rather than just scanning files within a file system. They search boot sectors, memory, inbound and outbound e-mail, files as they are downloaded, files on removable devices or media, and so on.

The best protection against computer viruses is prevention, or the practice of **safe computing**. Purchasing unopened software from vendors and avoiding free or pirated copies from public sources or disk exchange offer the safest route to preventing infection. However, even new copies of legitimate software applications are not immune to virus infection: in a few cases, disgruntled employees of a software company have infected the master copies of software programs to do economic harm to the company. For macro viruses, one defense is to exchange Microsoft Word documents in an alternative file format called **rich text format (RTF)**. Unlike the native Word format, RTF does not include the capability to attach macros.

Another defense is to avoid opening any e-mail attachments from unknown users. Unfortunately, history has shown that e-mail vulnerabilities appear as fast as they are fixed. For example, in 2000, the *love bug* virus became very widespread by traveling in e-mail messages that pretended to be love notes sent by friends of the receivers. Once a receiver opened the attached Visual Basic script, the virus propagated by sending itself to the first addresses in the receiver's e-mail contact list. Fortunately, except for clogging e-mail systems

THE TRIPWIRE FILE SYSTEM

An example of an anomaly-detection tool is the [Tripwire file system](#) integrity-checking tool for UNIX, developed at Purdue University. Tripwire operates on the premise that many intrusions result in modification of system directories and files. For example, an attacker might modify the system programs, perhaps inserting copies with Trojan horses, or might insert new programs into directories commonly found in user-shell search paths. Or an intruder might remove system log files to cover his tracks. Tripwire is a tool to monitor file systems for added, deleted, or changed files and to alert system administrators to these modifications.

The operation of Tripwire is controlled by a configuration file `tw.config` that enumerates the directories and files to be monitored for changes, deletions, or additions. Each entry in this configuration file includes a selection mask to specify the file attributes (inode attributes) that will be monitored for changes. For example, the selection mask might specify that a file's permissions be monitored but its access time be ignored. In addition, the selection mask can instruct that the file be monitored for changes. Monitoring the hash of a file for changes is as good as monitoring the file itself, and storing hashes of files requires far less room than copying the files themselves.

When run initially, Tripwire takes as input the `tw.config` file and computes a signature for each file or directory consisting of its monitored attributes (inode attributes and hash values). These signatures are stored in a database. When run subsequently, Tripwire inputs both `tw.config` and the previously stored database, recomputes the signature for each file or directory named in `tw.config`, and compares this signature with the signature (if any) in the previously computed database. Events reported to an administrator include any monitored file or directory whose signature differs from that in the database (a changed file), any file or directory in a monitored directory for which a signature does not exist in the database (an added file), and any signature in the database for which the corresponding file or directory no longer exists (a deleted file).

Although effective for a wide class of attacks, Tripwire does have limitations. Perhaps the most obvious is the need to protect the Tripwire program and its associated files, especially the database file, from unauthorized modification. For this reason, Tripwire and its associated files should be stored on some tamper-proof medium, such as a write-protected disk or a secure server where logins can be tightly controlled. Unfortunately, this makes it less convenient to update the database after authorized updates to monitored directories and files. A second limitation is that some security-relevant files—for example, system log files—are *supposed* to change over time, and Tripwire does not provide a way to distinguish between an authorized and an unauthorized change. So, for example, an attack that modifies (without deleting) a system log that would normally change anyway would escape Tripwire's detection capabilities. The best Tripwire can do in this case is to detect certain obvious inconsistencies (for example, a shrinking log file). Free and commercial versions of Tripwire are available from <http://tripwire.org> and <http://tripwire.com>.

and users' inboxes, it was relatively harmless. It did, however, effectively negate the defensive strategy of opening attachments only from people known to the receiver. A more effective defense method is to avoid opening any e-mail attachment that contains executable code. Some companies now enforce this as policy by removing all incoming attachments to e-mail messages.

Another safeguard, although it does not prevent infection, does permit early detection. A user must begin by completely reformatting the hard disk, especially the boot sector, which is often targeted for viral attack. Only secure software is uploaded, and a signature of each program is taken via a secure message-digest computation. The resulting file name and associated message-digest list must then be kept free from unauthorized access. Periodically, or each time a program is run, the operating system recomputes the signature and compares it with the signature on the original list; any differences serve as a warning of possible infection. This technique can be combined with others. For example, a high-overhead antivirus scan, such as a sandbox, can be used; and if a program passes the test, a signature can be created for it. If the signatures match the next time the program is run, it does not need to be virus-scanned again.

14.6.5 Auditing, Accounting, and Logging

Auditing, accounting, and logging can decrease system performance, but they are useful in several areas, including security. Logging can be general or specific. All system-call executions can be logged for analysis of program behavior (or misbehavior). More typically, suspicious events are logged. Authentication failures and authorization failures can tell us quite a lot about break-in attempts.

Accounting is another potential tool in a security administrator's kit. It can be used to find performance changes, which in turn can reveal security problems. One of the early UNIX computer break-ins was detected by Cliff Stoll when he was examining accounting logs and spotted an anomaly.

14.7 Firewalling to Protect Systems and Networks

We turn next to the question of how a trusted computer can be connected safely to an untrustworthy network. One solution is the use of a firewall to separate trusted and untrusted systems. A **firewall** is a computer, appliance, or router that sits between the trusted and the untrusted. A network firewall limits network access between the two **security domains** and monitors and logs all connections. It can also limit connections based on source or destination address, source or destination port, or direction of the connection. For instance, web servers use HTTP to communicate with web browsers. A firewall therefore may allow only HTTP to pass from all hosts outside the firewall to the web server within the firewall. The Morris Internet worm used the finger protocol to break into computers, so finger would not be allowed to pass, for example.

In fact, a network firewall can separate a network into multiple domains. A common implementation has the Internet as the untrusted domain; a semitrusted and semisecure network, called the **demilitarized zone (DMZ)**, as another domain; and a company's computers as a third domain (Figure

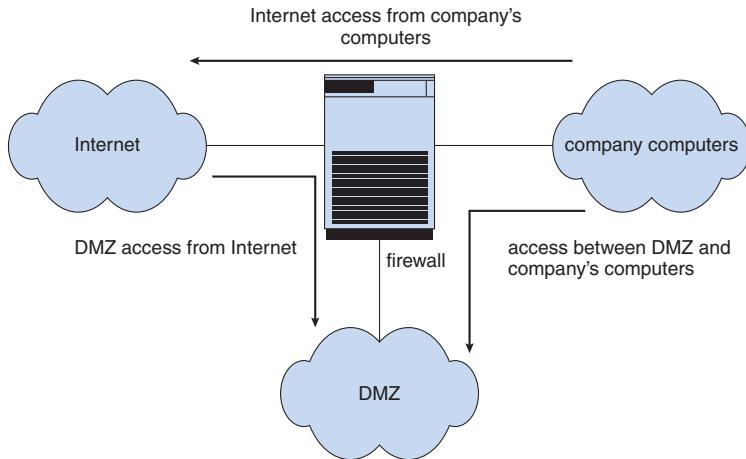


Figure 14.10 Domain separation via firewall.

14.10). Connections are allowed from the Internet to the DMZ computers and from the company computers to the Internet but are not allowed from the Internet or DMZ computers to the company computers. Optionally, controlled communications may be allowed between the DMZ and one company computer or more. For instance, a web server on the DMZ may need to query a database server on the corporate network. With a firewall, however, access is contained, and any DMZ systems that are broken into still are unable to access the company computers.

Of course, a firewall itself must be secure and attack-proof. Otherwise, its ability to secure connections can be compromised. Furthermore, firewalls do not prevent attacks that **tunnel**, or travel within protocols or connections that the firewall allows. A buffer-overflow attack to a web server will not be stopped by the firewall, for example, because the HTTP connection is allowed; it is the contents of the HTTP connection that house the attack. Likewise, denial-of-service attacks can affect firewalls as much as any other machines. Another vulnerability of firewalls is **spoofing**, in which an unauthorized host pretends to be an authorized host by meeting some authorization criterion. For example, if a firewall rule allows a connection from a host and identifies that host by its IP address, then another host could send packets using that same address and be allowed through the firewall.

In addition to the most common network firewalls, there are other, newer kinds of firewalls, each with its pros and cons. A **personal firewall** is a software layer either included with the operating system or added as an application. Rather than limiting communication between security domains, it limits communication to (and possibly from) a given host. A user could add a personal firewall to her PC so that a Trojan horse would be denied access to the network to which the PC is connected, for example. An **application proxy firewall** understands the protocols that applications speak across the network. For example, SMTP is used for mail transfer. An application proxy accepts a connection just as an SMTP server would and then initiates a connection to the original destination SMTP server. It can monitor the traffic as it forwards the message, watching for and disabling illegal commands, attempts to exploit

bugs, and so on. Some firewalls are designed for one specific protocol. An [XML firewall](#), for example, has the specific purpose of analyzing XML traffic and blocking disallowed or malformed XML. [System-call firewalls](#) sit between applications and the kernel, monitoring system-call execution. For example, in Solaris 10, the “least privilege” feature implements a list of more than fifty system calls that processes may or may not be allowed to make. A process that does not need to spawn other processes can have that ability taken away, for instance.

14.8 Computer-Security Classifications

The U.S. Department of Defense Trusted Computer System Evaluation Criteria specify four security classifications in systems: A, B, C, and D. This specification is widely used to determine the security of a facility and to model security solutions, so we explore it here. The lowest-level classification is division D, or minimal protection. Division D includes only one class and is used for systems that have failed to meet the requirements of any of the other security classes. For instance, MS-DOS and Windows 3.1 are in division D.

Division C, the next level of security, provides discretionary protection and accountability of users and their actions through the use of audit capabilities. Division C has two levels: C1 and C2. A C1-class system incorporates some form of controls that allow users to protect private information and to keep other users from accidentally reading or destroying their data. A C1 environment is one in which cooperating users access data at the same levels of sensitivity. Most versions of UNIX are C1 class.

The total of all protection systems within a computer system (hardware, software, firmware) that correctly enforce a security policy is known as a [trusted computer base \(TCB\)](#). The TCB of a C1 system controls access between users and files by allowing the user to specify and control sharing of objects by named individuals or defined groups. In addition, the TCB requires that the users identify themselves before they start any activities that the TCB is expected to mediate. This identification is accomplished via a protected mechanism or password. The TCB protects the authentication data so that they are inaccessible to unauthorized users.

A C2-class system adds an individual-level access control to the requirements of a C1 system. For example, access rights of a file can be specified to the level of a single individual. In addition, the system administrator can selectively audit the actions of any one or more users based on individual identity. The TCB also protects itself from modification of its code or data structures. In addition, no information produced by a prior user is available to another user who accesses a storage object that has been released back to the system. Some special, secure versions of UNIX have been certified at the C2 level.

Division-B mandatory-protection systems have all the properties of a class-C2 system. In addition, they attach a sensitivity label to each object in the system. The B1-class TCB maintains these labels, which are used for decisions pertaining to mandatory access control. For example, a user at the confidential level could not access a file at the more sensitive secret level. The TCB also denotes the sensitivity level at the top and bottom of each

page of any human-readable output. In addition to the normal user-name–password authentication information, the TCB also maintains the clearance and authorizations of individual users and will support at least two levels of security. These levels are hierarchical, so that a user may access any objects that carry sensitivity labels equal to or lower than his security clearance. For example, a secret-level user could access a file at the confidential level in the absence of other access controls. Processes are also isolated through the use of distinct address spaces.

A B2-class system extends the sensitivity labels to each system resource, such as storage objects. Physical devices are assigned minimum and maximum security levels that the system uses to enforce constraints imposed by the physical environments in which the devices are located. In addition, a B2 system supports covert channels and the auditing of events that could lead to the exploitation of a covert channel.

A B3-class system allows the creation of access-control lists that denote users or groups not granted access to a given named object. The TCB also contains a mechanism to monitor events that may indicate a violation of security policy. The mechanism notifies the security administrator and, if necessary, terminates the event in the least disruptive manner.

The highest-level classification is division A. Architecturally, a class-A1 system is functionally equivalent to a B3 system, but it uses formal design specifications and verification techniques, granting a high degree of assurance that the TCB has been implemented correctly. A system beyond class A1 might be designed and developed in a trusted facility by trusted personnel.

The use of a TCB merely ensures that the system can enforce aspects of a security policy; the TCB does not specify what the policy should be. Typically, a given computing environment develops a security policy for **certification** and has the plan **accredited** by a security agency, such as the National Computer Security Center. Certain computing environments may require other certification, such as that supplied by TEMPEST, which guards against electronic eavesdropping. For example, a TEMPEST-certified system has terminals that are shielded to prevent electromagnetic fields from escaping. This shielding ensures that equipment outside the room or building where the terminal is housed cannot detect what information is being displayed by the terminal.

14.9 An Example: Windows 7

Microsoft Windows 7 is a general-purpose operating system designed to support a variety of security features and methods. In this section, we examine features that Windows 7 uses to perform security functions.

The Windows 7 security model is based on the notion of **user accounts**. Windows 7 allows the creation of any number of user accounts, which can be grouped in any manner. Access to system objects can then be permitted or denied as desired. Users are identified to the system by a *unique* security ID. When a user logs on, Windows 7 creates a **security access token** that includes the security ID for the user, security IDs for any groups of which the user is a member, and a list of any special privileges that the user has. Examples of special privileges include backing up files and directories, shutting down the computer, logging on interactively, and changing the system clock. Every

process that Windows 7 runs on behalf of a user will receive a copy of the access token. The system uses the security IDs in the access token to permit or deny access to system objects whenever the user, or a process on behalf of the user, attempts to access the object. Authentication of a user account is typically accomplished via a user name and password, although the modular design of Windows 7 allows the development of custom authentication packages. For example, a retinal (or eye) scanner might be used to verify that the user is who she says she is.

Windows 7 uses the idea of a subject to ensure that programs run by a user do not get greater access to the system than the user is authorized to have. A **subject** is used to track and manage permissions for each program that a user runs. It is composed of the user's access token and the program acting on behalf of the user. Since Windows 7 operates with a client–server model, two classes of subjects are used to control access: simple subjects and server subjects. An example of a **simple subject** is the typical application program that a user executes after she logs on. The simple subject is assigned a **security context** based on the security access token of the user. A **server subject** is a process implemented as a protected server that uses the security context of the client when acting on the client's behalf.

As mentioned in Section 14.7, auditing is a useful security technique. Windows 7 has built-in auditing that allows many common security threats to be monitored. Examples include failure auditing for login and logoff events to detect random password break-ins, success auditing for login and logoff events to detect login activity at strange hours, success and failure write-access auditing for executable files to track a virus outbreak, and success and failure auditing for file access to detect access to sensitive files.

Windows added mandatory integrity control, which works by assigning an **integrity label** to each securable object and subject. In order for a given subject to have access to an object, it must have the access requested in the discretionary access-control list, and its integrity label must be equal to or higher than that of the secured object (for the given operation). The integrity labels in Windows 7 are (in ascending order): untrusted, low, medium, high, and system. In addition, three access mask bits are permitted for integrity labels: NoReadUp, NoWriteUp, and NoExecuteUp. NoWriteUp is automatically enforced, so a lower-integrity subject cannot perform a write operation on a higher-integrity object. However, unless explicitly blocked by the security descriptor, it can perform read or execute operations.

For securable objects without an explicit integrity label, a default label of medium is assigned. The label for a given subject is assigned during logon. For instance, a nonadministrative user will have an integrity label of medium. In addition to integrity labels, Windows Vista also added User Account Control (UAC), which represents an administrative account (not the built-in Administrators account) with two separate tokens. One, for normal usage, has the built-in Administrators group disabled and has an integrity label of medium. The other, for elevated usage, has the built-in Administrators group enabled and an integrity label of high.

Security attributes of an object in Windows 7 are described by a **security descriptor**. The security descriptor contains the security ID of the owner of the object (who can change the access permissions), a group security ID used only by the POSIX subsystem, a discretionary access-control list that identifies

which users or groups are allowed (and which are explicitly denied) access, and a system access-control list that controls which auditing messages the system will generate. Optionally, the system access-control list can set the integrity of the object and identify which operations to block from lower-integrity subjects: read, write (always enforced), or execute. For example, the security descriptor of the file `foo.bar` might have owner `avi` and this discretionary access-control list:

- `avi`—all access
- group `cs`—read–write access
- user `cliff`—no access

In addition, it might have a system access-control list that tells the system to audit writes by everyone, along with an integrity label of medium that denies read, write, and execute to lower-integrity subjects.

An access-control list is composed of access-control entries that contain the security ID of the individual and an access mask that defines all possible actions on the object, with a value of `AccessAllowed` or `AccessDenied` for each action. Files in Windows 7 may have the following access types: `ReadData`, `WriteData`, `AppendData`, `Execute`, `ReadExtendedAttribute`, `WriteExtendedAttribute`, `ReadAttributes`, and `WriteAttributes`. We can see how this allows a fine degree of control over access to objects.

Windows 7 classifies objects as either container objects or noncontainer objects. **Container objects**, such as directories, can logically contain other objects. By default, when an object is created within a container object, the new object inherits permissions from the parent object. Similarly, if the user copies a file from one directory to a new directory, the file will inherit the permissions of the destination directory. **Noncontainer objects** inherit no other permissions. Furthermore, if a permission is changed on a directory, the new permissions do not automatically apply to existing files and subdirectories; the user may explicitly apply them if he so desires.

The system administrator can prohibit printing to a printer on the system for all or part of a day and can use the Windows 7 Performance Monitor to help her spot approaching problems. In general, Windows 7 does a good job of providing features to help ensure a secure computing environment. Many of these features are not enabled by default, however, which may be one reason for the myriad security breaches on Windows 7 systems. Another reason is the vast number of services Windows 7 starts at system boot time and the number of applications that typically are installed on a Windows 7 system. For a real multiuser environment, the system administrator should formulate a security plan and implement it, using the features that Windows 7 provides and other security tools.

14.10 Summary

Protection is an internal problem. Security, in contrast, must consider both the computer system and the environment—people, buildings, businesses, valuable objects, and threats—with which the system is used.

The data stored in the computer system must be protected from unauthorized access, malicious destruction or alteration, and accidental introduction of