CONTROLLING FILES

CHAPTER 3

ABOUT THIS CHAPTER

In this chapter, we look at access control mechanisms used to protect larger-scale resources like files and folders. The chapter focuses on these topics:

- Overview of file system technology
- Structure of executable files and viruses
- Goals and basic policies for file protection and sharing
- File security controls based on "permission flags"

3.1 The File System

Data sharing poses a problem: If people can look at *some* information on a computer, how do we prevent them from seeing *all* the information? This is the same problem as RAM protection in Chapter 2. If any user can modify any information inside the computer, we can't prevent users from reaching any of its contents.

For most people, their most important information assets reside in their files. Modern file systems keep the hard drive organized and give us a way to reliably locate and save our data.

All modern systems use a ***hierarchical directory*** to organize files into groups. The directory forms an inverted tree growing from the topmost directory, called the ***root directory***. Many users think of files as living in ***file folders*** instead of directories. In practice, we use the terms interchangeably.

On Microsoft Windows, each drive has its own directory, starting at its own root. Figure 3.1 shows the root folder for a Windows system drive. We can tell it's the root because the "Address" field of the window only contains the drive letter ("C:\") and no other names.

On Unix-based systems, the system drive provides the root. The directories on all other drives appear beneath it in the hierarchy. In OS-X, there is a Volumes subdirectory that leads to the directories for individual drives.

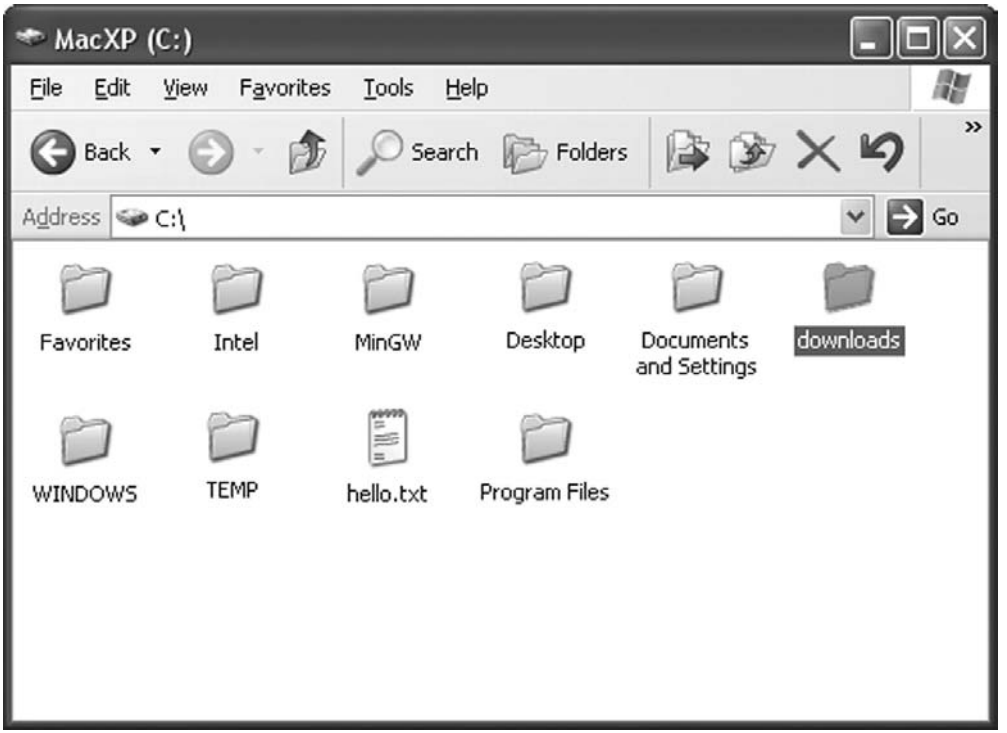

Figure 3.1

Root directory folder on Windows drive C:\.

When we click on a folder icon within a folder on our desktop, we go "down" a level in the directory hierarchy (Figure 3.2). On Windows, the "Address" field contains the directory's name following the "\" character.

Directory Path and File Names

Each file on a hard drive has a unique identifier made of the file's name and the list of directories leading to it. If a file is in the root directory, then we don't need a directory name to find it. All we need is the "\" character, which by itself indicates the root directory. In Figure 3.1, we see a file named "hello.txt" in the root directory. The file's entire name consists of the drive letter, a "\" character, and the file name:

```
C:\hello.txt
```

If the file is in a subdirectory, then we need both the name and the ***directory path***, which leads us to the file's directory starting from the root directory. We see from Figure 3.2 that there is also a file named "hello.txt" in the "downloads" directory. To choose this file, instead of the one in the root directory, we include the path to its directory:

```
C:\downloads\hello.txt
```

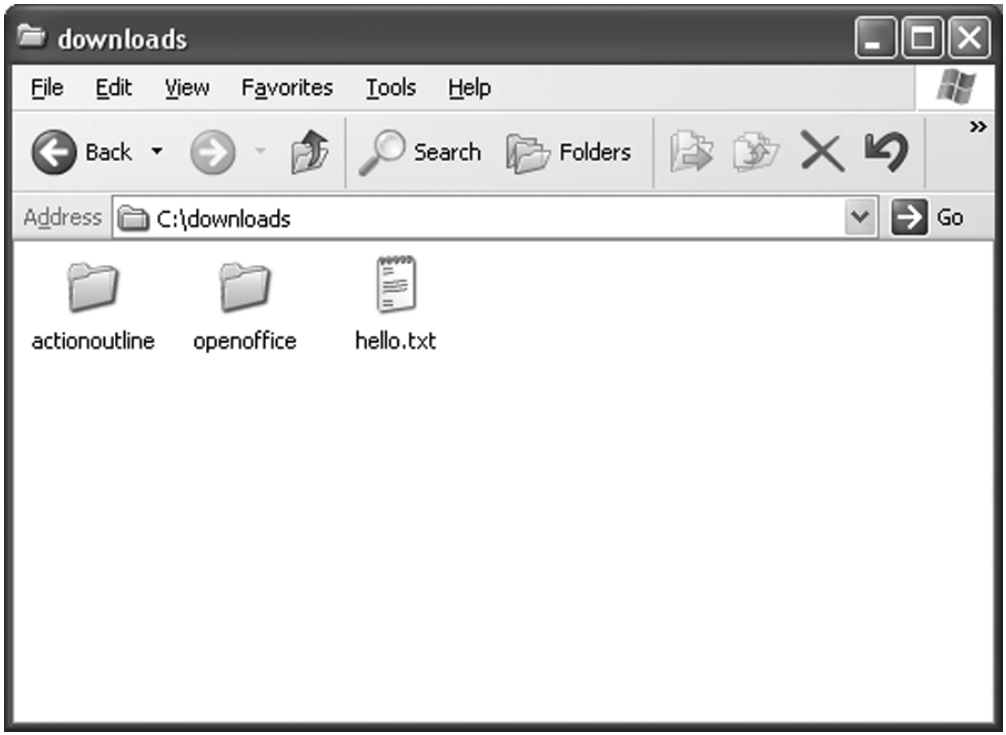

Figure 3.2
The "downloads" subdirectory.

If the file resides in a directory within the "downloads" directory, then we must include that directory in the path. Each directory name is separated by a "\" and they occur in order starting from the root directory. Not all systems use the same character to separate directory names; Table 3.1 lists the commonly used characters.

File systems, like operating systems, mostly fall into either of two categories: Windows systems and Unix-based systems. Apple's Macintosh OS-X is based on a version of Unix, but it uses a distinctive file system. Unless otherwise noted, like in Table 3.1, OS-X has the features of a Unix-based system.

TABLE 3.1 File and directory name formats

System	Separator	Character	Example File Name with Path
Macintosh	Colon	:	Mac:downloads:hello.txt
Other Unix-like systems	Slash	/	/downloads/hello.txt
Windows and MS-DOS	Back slash	\	C:\downloads\hello.txt

3.1.1 File Ownership and Access Rights

Bob has decided to share his tower computer with his suitemates. He has activated the login mechanism and constructed three logins: an administrator ("superbob"), one for his schoolwork ("bob"), and one for his suitemates ("suitemates"). He established a password for the first two accounts. The third account doesn't need a password. Any suite mate or guest may walk up to the computer and click on the "suitemates" name to log in.

Bob has just typed up an essay for his English class. He saved it on the hard drive. How can he protect it from injury by his suitemates?

Even though Bob shares the computer with his suitemates, he still needs to protect his files. Some desktop operating systems allow us to run without the apparent need for user identities or file protection, but most modern systems have those features built in. In older systems, we need to activate multiuser features before we can use them. Once Bob activated user identities, he needs to protect *his* files.

Modern operating systems protect files according to user identity. When Bob logs in as "bob" and presents his secret password, the system starts up a graphical interface process so Bob can run programs and use his files (Figure 3.3).

The system associates Bob's user identity with the processes he runs. When he edits an essay for English class, the word processor takes on his user identity. It then creates and modifies his files on his behalf. Since Bob created his essay file, the system naturally grants him access rights to the file.

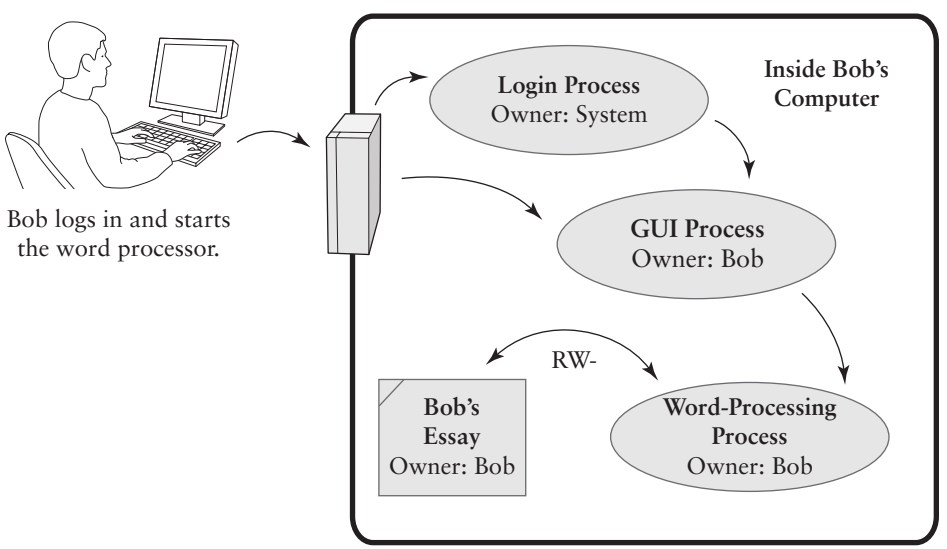

Figure 3.3

Bob's processes inherit his identity and access rights.

File Access Rights

When we looked at RAM access in Chapter 2, we focused on "read" and "write" access permissions. An operating system provides four access rights for files and most other resources:

- **Create** a new instance of the resource (for example, a new file).
- **Read** the contents of a particular resource.
- **Update** or "write" or "modify" a particular resource.
- **Delete** or destroy an existing resource.

The names of these four rights form the acronym ***CRUD***. While the four rights also may apply to RAM and processes, it's more common for programs to apply them to files. When a process tries to create or use a file, it inherits whatever rights belong to the user who started the process.

Some systems also provide a right to **append** to a file; that is, add data to the end of the file without reading its contents. There also may be an **execute** right that we will examine later.

Initial File Protection

A modern personal computer easily may contain hundreds of thousands of files. This multitude poses a real problem: How do we establish protections for all of them? It's not practical to set permissions individually. The system has to apply the correct permissions automatically whenever a user creates a new file or directory. Systems tend to implement either of two mechanisms to apply initial access rights:

1. Default rights. The system applies a standard set of permissions to all files a particular user creates. This is common in Unix-based systems.
2. Inherited rights. The system applies permissions inherited from one or more enclosing directories. We will see specific examples of this in Chapter 4.

3.1.2 Directory Access Rights

Directories (folders) tie together files in different parts of the hard drive. A file exists because it has a directory entry, and we must change that directory entry to delete it. Thus, we don't have full control over a file unless we have the right to change its directory entry.

Figure 3.4 shows the directory structure in which Bob's essay file resides. The file resides in Bob's personal directory (named **bob**). His personal directory resides in the system's "root" directory. To find the file, Bob's process starts at the root directory and looks for Bob's personal directory. The root directory entry leads the process to Bob's personal directory, then the process searches Bob's personal directory. The entry in that directory leads it, finally, to the file itself.

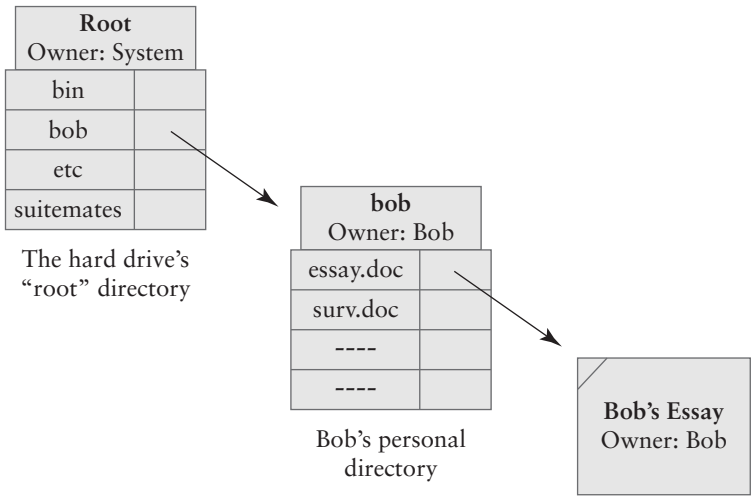

Figure 3.4
Directory and file ownership.

Operating systems grant or deny rights in terms of *containers*. We may have the right to change the file itself; we may even own the file, but we need rights to its directory in order to change that directory entry.

The basic access rights to a directory are **read**, **write**, and **seek**. Bob has the right to read his own directory, so he can list the files in his directory and look at file details stored in that directory. Bob also can write to his own directory, so he can add, modify, or delete files listed in it.

The ***seek right*** is separate from the read right; it allows the user's process to search a directory for a particular name in a file's path, but not to examine the directory as a whole. This allows other users to retrieve a file from a directory even if they don't have the right to read the entire directory.

For example, the system could give Bob the right to seek file names in the root directory but withhold the right to read that directory. Bob would be able to locate his own directory by name. He would not be allowed to list the names of other files or directories in the root.

To protect the directories belonging to other users, Bob does not have the right to write to the root directory. Because of this, Bob does not have the right to delete his own directory.

Some systems provide very specific rights for managing directories, while others provide minimal rights, like read, write, and seek. Here is a detailed listing of possible rights a system could grant for access to a directory:

- **Create** a new directory—like the generic creation right.

- **Delete** a directory—like the generic deletion right. This operation can be tricky because directories contain files, and the process might not have delete rights for all of its files.
- **Seek** a directory for an entry—the right doesn't necessarily allow the process to list the names of all directory entries, or to examine properties of individual entries.
- **Read** a directory—this right allows listing of the directory as well as seeking files in it.
- **Create new files** in a directory—may be part of a generic "write" permission or a separate right.
- **Delete files** in a directory—part of a generic "write" permission or a separate right.

We can come close to implementing these distinct rights even if the system restricts us to read, write, and seek rights. The arrangement would not be able to distinguish between creating or deleting files or directories.

If the system implements these six distinct rights, it can achieve a higher degree of Least Privilege. This may or may not be important, depending on the security policy we need to enforce. If the policy doesn't need to distinguish between files and directories, then the extra rights aren't necessary. In either case, we want the system to apply protections that enforce our policy objectives.

3.2 Executable Files

When we look in our directories at the files, we must distinguish between two distinct file types:

1. Data files, like a word-processing file containing an essay, a spreadsheet, or a configuration file used by a program or the operating system.
2. Executable files that contain application programs or other programs.

An executable file contains instructions to be executed by the CPU. When we load those instructions into a control section in RAM, the CPU runs that program. This was described in Section 2.1. The program's instructions must be organized and formatted in a special "executable" file format. Although the format varies with different operating systems, they are all similar to what we see in Figure 3.5.

Every executable file begins with a "file header" that describes the structure and format of the program. The header contains these fields:

- Magic number—a standard data value that appears in the first location of the executable file. Different operating systems and different CPUs typically use different magic numbers so that the system won't accidentally run a program that can only run correctly on a different type of system.
- Program size—indications of the size of the block of machine instructions that make up the program itself.

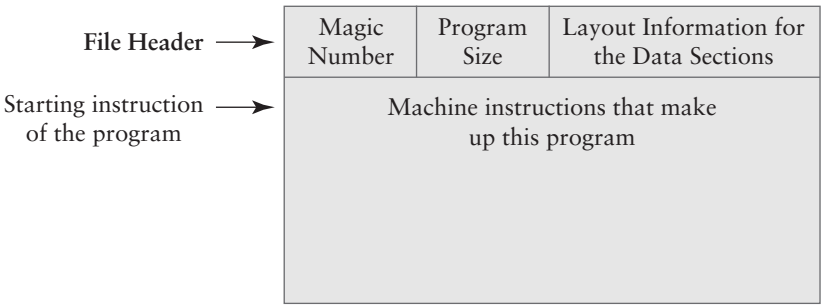

Figure 3.5
Executable file format.

- Layout information—addresses and offsets to be used to lay out variables and stack locations in the program's data section.

Following the header, the rest of the file consists primarily of machine instructions. When we execute the program in a file, the operating system loads those instructions directly into a control section. When starting the program, the operating system sets the program counter to the starting address of the control section.

3.2.1 Execution Access Rights

Figure 3.6 shows what happens when Bob runs the word-processor program to edit his essay. The square boxes on the top left and right represent files. The box with the tab in the left corner is a data file, owned by Bob, that contains his essay. The plain box on the right is an executable file owned by the system that contains the word-processor program.

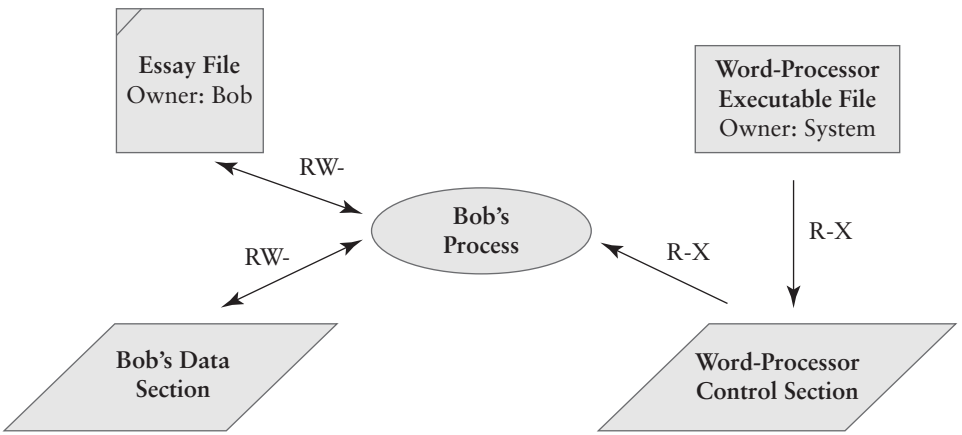

Figure 3.6
Bob executes the word processor to edit his essay.

When Bob starts the word processor, the system reads the executable file into the word-processor control section (lower right corner). As the process executes the program, it uses the data section (lower left corner) to contain working variables and to store the parts of the essay file being edited.

Bob has full ownership of his essay file and full read/write access to his data section. The file containing the word-processor program is owned by the operating system. Bob has read-only access to the file, and to its control section. When Bob starts the program, the operating system itself copies the file's instructions into the control section.

As before, we represent access rights with labeled arrows. Note that executable files and control sections may have an "X" permission. In the case of the file, this indicates the ***execute right***. When present, a user may execute the file. Most modern systems provide this right. This helps implement Least Privilege; it allows a user to run a program, but does not imply the right to copy it or visually examine the executable instructions. In practice, however, most systems provide both Read and Execute permissions, because parts of the system might not respond well if they can't actually read the program file. The programmer who actually creates a program has all three rights, read, write, and execute, which yields "RWX."

Types of Executable Files

Application programs aren't the only executable files on a typical operating system. The first and most important executable file is the "kernel" of the operating system itself. When we boot the system, the BIOS reads this file from the hard drive and loads it into RAM. From there, the rest of the operating system starts up. Any part of the operating system that may be added while the system is running must reside in its own executable file.

Here is a list of common types of executable files:

- **Application programs**—the files we execute to run useful applications. Some "system utility" programs, like those for managing and formatting hard drives, are application programs.
- **Operating system kernel**—the file the BIOS reads into RAM during the bootstrap process. This contains the machine instructions that make up the operating system's main procedures.
- **Device drivers**—custom procedures for using I/O devices.
- **Shared libraries**—useful functions that may be shared among multiple programs (dynamic link libraries in Windows).

These files, like typical executable files, contain machine instructions. To run the program, we load it into RAM and direct the CPU to execute the instructions. Some systems, however, notably those derived from Unix, also treat some text files as executable files. These are files written in a ***scripting language***. These include programming languages like Perl, Python, PHP, or Lisp. The CPU can't execute these programs directly, but instead must execute the language's ***interpreter*** program. This is also true of

programs written in versions of the Basic language, including Visual Basic. We discuss scripting languages further when we discuss computer viruses in Section 3.2.3.

A system's security depends on protecting executable files, especially operating system files. An attacker can break our Chain of Control by modifying an executable file, especially if it is a critical operating system file.

3.2.2 Computer Viruses

By 1981, the Apple II had become an incredibly popular desktop computer. Introduced at the West Coast Computer Faire in 1977, the Apple received a huge boost in sales with the introduction of "Visicalc" in 1979, the first spreadsheet program. Visicalc justified buying an Apple II for the office. Back then, hard drives were too expensive for small computers. Most computers used removable diskettes. The operating system and one or two applications could fit on a single diskette.

The computer software industry was in its infancy. People often borrowed software from one another and often copied each others' application programs. This behavior was especially rampant with computer games. Bill Gates, then the young co-founder of the tiny Microsoft Corporation, published articles urging computer users to pay for their software and shun unauthorized copies, but the sharing (called piracy today) persisted.

There was little to discourage computer users from sharing software until a few of them noticed this odd phenomenon: Their application programs grew larger and started more slowly. Closer investigation revealed that a small program attached itself to each of their application program files. When such a program was executed, it would attach a similar program to any other program files it found.

In 1983, researcher Fred Cohen performed a series of studies in which he constructed programs that replicated themselves. He used the term *computer virus* to describe them. At first, other researchers saw little relevance in this work, but soon viruses emerged as the first widespread form of computer malware.

Virus Infection

When a virus "infected" an application program, it added its own machine instructions to the end of the program's file (Figure 3.7).

By adjusting the file headers, the virus ensured that its instructions were loaded into RAM along with the application program. The virus also modified the application itself so that the virus program would run first.

When the user started up an infected application, the operating system loaded the program into RAM, including the virus. The system jumped to the start of the application, which had been modified to make the virus program run first. Here is what a typical virus did:

1. The virus searched the operating system's file directory to locate every file containing an application program. It would then inspect each application program file.

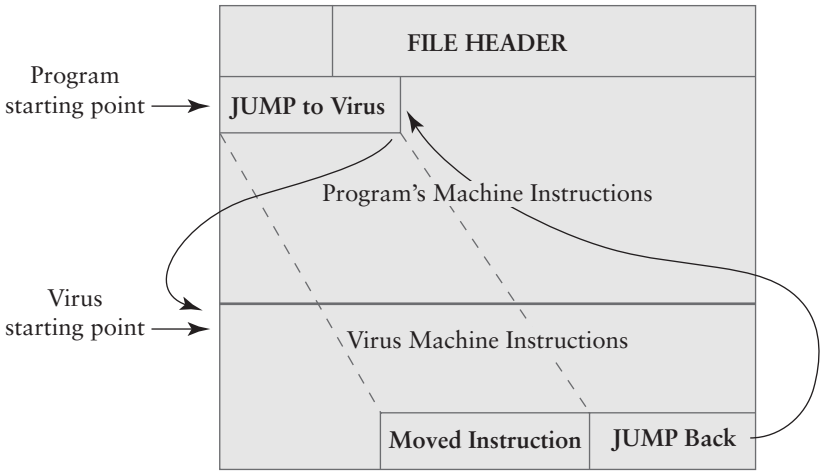

Figure 3.7

A virus infection in an executable program file.

2. If the program file had already been infected by the virus, the virus skipped to the next file.
3. If the file was not infected, the virus copied its executable instructions to the end of the file, "appending" to it. This reduced the risk of accidentally damaging the application.
4. Next, the virus modified the first instruction in the application to "Jump" to the start of the virus program. The virus also saved that first instruction, so that it would be the last instruction the virus executed itself before jumping back to resume the application. Now the file was fully "infected" and would run the virus the next time application started.
5. Once the virus had checked every application program on the system, it executed the instruction from the start of the application, and jumped back to the application program.

The computer user might see a delay before the application started, while these steps were performed—or not.

Biological viruses rely on particular techniques to infect other victims; some are airborne, some travel in food, and so on. In the case of computer viruses, infection only occurs if the victim executes the virus code. In other words, the virus subverts the Chain of Control to cause itself to be executed. There are often many ways this can happen. In the early days, viruses most often spread by running an infected application.

Originally, viruses seemed benign. Early virus authors sought persistence and replication. They found their viruses spread more effectively if they were innocuous and hard

to detect. Even so, viruses made assumptions about executable files, and could damage a file if the assumptions were incorrect.

Malicious Viruses

By the late 1980s, however, some virus writers were inclined toward destruction. The Jerusalem virus, which appeared in 1987 in the city of Jerusalem, contained a "destructive payload" that would delete all executable files on the system on Friday the 13th, starting in 1988. This type of trigger was typical of destructive viruses: They would spread for some period of time, and then a calendar date would trigger a destructive action. The Michelangelo virus (1991) was a famous example of this; it was programmed to reformat hard drives on March 6, Michelangelo's birthday.

Michelangelo, like several later viruses, did not directly infect applications. Instead, it infected the ***Master Boot Record*** (MBR) of diskettes and hard drives (Figure 3.8). Whenever a computer bootstrapped from a particular disk, the BIOS would first read in and execute the instructions stored on the MBR.

By that time, low-cost disk drives were common on desktop computers. The Michelangelo virus code ran whenever the system was booted from an infected disk, whether a hard drive or diskette. The virus would then modify the bootstrap data on every disk on the system so that it would execute the virus code before it bootstrapped the operating system.

Figure 3.8

A diskette infected with a virus, circa 1991.

Applications and bootstraps were not the only techniques used to spread viruses. Any time the virus writer can trick a victim into executing a program, the virus writer will use it to spread a virus. There are several ways to automatically execute a program when inserting a USB drive into a computer; several viruses use such techniques to spread between computers.

3.2.3 Macro Viruses

Whenever we write a computer program, we write it in a programming language. For us to use a particular language, we need a program that translates what we wrote into actions performed by the CPU. These "programming language programs" fall into two categories:

1. Compiler—a program that converts our program into machine language. The CPU executes the machine language. C, Java, Fortran, and Cobol are examples of compilers.
2. Interpreter—a program that "interprets" the text of our program a word at a time, and performs the actions specified in the text. Python, Visual Basic, Javascript, PHP, Lisp, and keyboard command processors are examples of interpreters.

We called interpreted languages *scripting languages* earlier. A program in a scripting language sometimes is called a *script*. For example, a file containing Unix shell commands or MSDOS commands is often called a ***shell script***. Today's languages may even straddle the line between compiling and interpreting. For example, Java usually compiles the program into "bytecodes," which then are run by a highly optimized interpreter that is customized to specific CPUs and operating systems.

Adobe's Acrobat program and most web browsers understand scripts written in Javascript. Typically, the script's author ties each script to a particular event within the document. In Acrobat, a file in ***Portable Document Format*** (PDF) may include a script that executes when we open the document, when we reach a particular page, or when we fill in fields in a form. On a web page, the script runs when we open the page or when the mouse moves or clicks on particular items on the page.

Sometimes we call scripts ***macros***, especially when we embed them in other documents. For example, many Microsoft Office applications will allow us to create a macro by recording a sequence of operations. The Office application typically translates the sequence into a Visual Basic program and saves it with the file. We can tell the application to perform the macro when particular events occur.

While these programming capabilities provide some interesting features, they also open users to attack. Most scripting languages can create files or modify other files and execute other programs. A script-oriented or macro virus uses these features to propagate the virus code.

In the early 1990s, a rumored macro virus infected Microsoft Word documents. When executed, the virus copied itself to other Word files it could find and also posted

the document to the Usenet News system. This published the document worldwide for viewing by everyone who used the Usenet News system.

While macro viruses posed a serious risk for some systems, vendors largely ignored the problem until such viruses were widely distributed by email (see Chapter 16). To reduce the risk of such viruses, Microsoft introduced "Macro Virus Protection." In 2010, Adobe announced plans to reduce the risks of Javascript-based viruses embedded in Acrobat documents.

3.2.4 Modern Malware: A Rogue's Gallery

Since the 1980s, antivirus researchers and vendors have identified thousands of viruses and other forms of malware. In 2008, vendor Symantec announced that the total number passed one million. Here is a selection of malware packages:

- Waledac
- Conficker, also called Downadup
- Pushdo/Cutwail
- ZeuS
- Stuxnet

Many of these packages propagate like viruses while others propagate across networks. Here is a summary of different propagation techniques used in modern malware:

- Infect USB drives—The malware copies itself to any plugged-in USB drives. The malware then configures the drive to automatically execute the malware when the drive is inserted in a computer.
- ***Drive-by downloads***—This is a network-based attack in which a user visits a web page or clicks on a pop-up window, and unintentionally downloads the malware.
- Worm propagation—The malware exploits a desktop or server vulnerability that it can reach via a network connection. The vulnerability allows the malware to install itself on the vulnerable computer (see Section 10.1.3).
- ***Trojan*** infection—A "Trojan" is a program that appears benign but in fact contains malware. In a Trojan infection, the malware arrives on the computer and the user is tricked into executing it. The term Trojan is derived from the story of the Trojan horse (Section 3.7).
- Email infection—Trojan malware arrives on the desktop via email, and the user executes it, causing the infection (see Section 15.3.3).

We examine the malware packages later and examine how they propagate and what they do. Many packages construct botnets (see Section 10.1.3). The botnet controller uses the network of subverted computers for malicious and often illegal purposes. We find several examples of this.

Waledac

Waledac constructs and manages a sophisticated and hard-to-trace botnet. The malware then provides several ways to make money from the network. Waledac is a Trojan that originally propagated primarily through email and drive-by downloads. An existing botnet operator may also download Waledac into the network of bots.

Once the network is in place, the controller may use it to do any of the following:

- Deliver spam email (see Section 15.3.1).
- Create pop-up windows that harass the computer's owner. Waledac's pop-ups usually warn of a virus infection and offers the sale of special antivirus software. The software is sold by the botnet operator, and it contains additional malware.
- Harvest authentication credentials from the PC (see Section 6.2.2).
- Perform "distributed" denial of service attacks (see Section 10.1.3).

Conficker, Also Called Downadup

The name "Conficker" is the popular name given to a worm often called "Downadup" in the security community. Conficker propagated through millions of Windows computers in 2009; infections subsided after that. Conficker constructed a botnet, but did not itself contain software to exploit the network. Instead, botnet operators would install other malware, like Waledac, and use it to make money from the botnet.

Pushdo/Cutwail

Starting in 2007, a malware organization constructed a botnet using Pushdo and Cutwail malware. Pushdo implements a botnet and supports downloading of additional malware. Cutwail implements email spam campaigns. In early 2009, Symantec's MessageLabs estimated that this botnet contained at least 1 million bots and averaged over 7 million spam messages per day.

Over the next 18 months, however, work by regulators and security researchers closed down most of the network computers used by the botnet operators to operate the botnet. This yielded a dramatic decrease in worldwide spam traffic.

ZeuS

ZeuS is a malware "product" in that its developers offer it for sale on the black market. Criminal groups purchase the software as an easy-to-use package to create a botnet focused on financial fraud.

ZeuS generally propagates via email or drive-by downloads. Email may often use social engineering techniques that direct victims to spoofed versions of popular social networking sites, like MySpace, Facebook, or LinkedIn.

When ZeuS infects a computer, it installs its "Zbot" malware, which connects the infected machine to the botnet. Zbot then steals financial credentials from the infected machine and transmits them to the botnet operator. Zbot also can use the victim's computer to perform financial transactions using the victim's credentials.

To complete the fraud, the botnet operator recruits individuals to visit banks and withdraw cash from looted accounts. These individuals, called "money mules," then forward the money to the controllers. In late 2010, the U.S. FBI announced over 100 arrests related to a ZeuS botnet, including dozens of money mules.

Stuxnet

Stuxnet is a worm that targets programmable logic controllers or PLCs. Factories, refineries, oil pipelines, and the electrical power grid use PLCs to automate industrial and utility systems. These devices are often connected to Windows computers on a network, and Stuxnet propagates by attacking the Windows computers.

Stuxnet is the most sophisticated piece of malware ever found. It propagates via USB drives so that it may infect a factory network even if the network isn't connected to the Internet. The worm uses several previously unknown Windows vulnerabilities to propagate on the network. Stuxnet seeks out specific PLC models that perform specific tasks, and ignores all others. In particular, it seems to target PLCs often used to process uranium and other nuclear materials. Stuxnet modifies the PLC software and then hides itself from detection. The changes affect motor speeds in a hard-to-detect way, which would ruin whatever industrial process the motor performed.

As of late 2010, experts could identify particular steps performed by Stuxnet, but could not conclusively identify its purpose nor its creators. Stuxnet's features suggest that it is a "cyber weapon" created by a government to attack another country's nuclear development facilities. It contains no mechanisms to make money, so it is unlikely to be the product of a criminal gang. It required a lot of specific knowledge about factory control systems and knowledge of previously unreported Windows vulnerabilities. It required a team of very sophisticated software developers.

3.3 Sharing and Protecting Files

Computer sharing happens all the time. Personal computers may be cheap enough to be treated as "personal property," but computers running server software share their resources continuously with countless users. Some desktops require controlled sharing, like those in libraries or computer labs in schools. Physicians, attorneys, financial planners, and many consultants may use a home computer to process sensitive information about patients or clients; these professionals are obligated to keep this information away from family members.

Sharing always has its limits. In a shared kitchen, the household's human residents may be perfectly happy to share boxes of cereal and bins of sugar with one another, but not with their pets or with ants and mice. We prevent "undesirable" sharing by enforcing controls. In a kitchen, we control sharing by putting food in pet-proof and vermin-proof containers (Figure 3.9).

In a computer, the operating system controls the sharing. Files and folders provide the software equivalent of cabinets, boxes, and jars. The operating system controls

Figure 3.9
Controlled sharing in a kitchen.

program execution, controls RAM access, and controls access to files on the hard disk. This helps ensure the orderly and predictable behavior of the overall system.

Controls on sharing also help prevent accidental and some malicious acts from causing damage. If a person spills something in a cupboard filled with sealed containers, the mess won't contaminate other food. If a mouse can't chew into a container of raisins, then the raisins remain safe for people to eat. If a user runs a virus-infested application but lacks the rights to modify application programs, then the virus can't infect other applications.

As with all security, the challenge is to find a balance between safety and meeting the users' needs. It's not reasonable to block a user from fixing broken software simply because the same restriction might prevent a virus infection. On the other hand, there's no sense in granting access to modify applications if the user has no business modifying them.

Objectives

To create policies for sharing files, we start by identifying potential risks. To identify risks, we start with a list of objectives:

a. Provide computing facilities for authorized users

b. Preserve the Chain of Control

c. Either permit or prevent general sharing of information among the users

The third objective reflects two alternatives for file sharing: either users share their files by default or they are kept isolated by default. We develop two different policies to reflect this choice.

Threats

To develop our list of risks, we consult the list of threats in Section 1.4 and focus on those that we can address through file protection. This eliminates people who won't be logged into the computer and people who simply pose a physical threat, like theft. This leaves the following threats:

- Script kiddy—sharing a computer at a school, or possibly at work, with people who have too much time and curiosity.
- Embezzler—sharing with a coworker who is motivated to steal from the company.
- Suite/room/family/housemate—sharing with someone in our living space.
- Malicious acquaintances—sharing with someone who visits our living space, getting physical access to the computer.
- Administrators—sharing a computer at school or work where an administrator might be inclined to eavesdrop on our work or perhaps even meddle with our files.

Risks

Now we make a list of attacks based on Figure 1.6. There are six attacks in the complete list; we can omit the first attack: physical theft. We can generalize the remaining ones to capture the essence of what the threats might do on an isolated computer. This yields the following list of risks:

1. Denial of service—someone deletes some of our files or damages software, making all or part of the computer unusable.
2. Subversion—a program gets a virus infection or suffers some other malware damage.
3. Masquerade—one user logs in, trying to pretend to be another.
4. Disclosure—some of our personal data is disclosed.
5. Forgery—someone modifies one of our files without our knowledge, so their statements are presented as our own.

This list of risks is comprehensive enough to use when developing our file protection policies. It is short enough to make it unnecessary to prioritize the risks.

3.3.1 Policies for Sharing and Protection

When we look at policies for sharing on a computing system, we see two general types of policies: ***global policies*** and ***tailored policies***. A global policy establishes file permissions to apply by default and, in particular, determines the permissions to apply when we create a new file. Typically, there are two global policies:

1. ***Isolation policy***—no access granted to other users' files
2. ***File-sharing policy***—read access granted to other users' files

User files aren't the only ones in the system. There are system files that might not even be visible to user programs. Most important for users, the system contains application programs and those are stored in files. To use these programs, the system must grant users the necessary rights. However, the system also must maintain the Chain of Control, so it should not grant the right to modify those programs.

Systems rarely grant both read and write access to other users' files, at least by default. Even if people are working together on the same project, it is much safer to limit the files they share completely. We need *tailored* policies to handle special cases like those. Here are examples:

- **Privacy**—block all access to certain files in a sharing environment.
- **Shared reading**—grant read-only access to certain files in an isolation environment.
- **Shared updating**—grant full access to certain files in either environment.

For now, however, we focus on global policies.

Underlying System Policy

Regardless of the policies we establish, we will use certain policy statements in all of them. These statements form an underlying system policy that address Objectives **a** and **b** introduced in Section 3.3, and the risks associated with them. We won't make policy statements for user file protection, since that depends on the choice posed by Objective **c**.

The policy statements appear in Table 3.2. These statements apply to both user isolation and user sharing situations.

This fundamental policy addresses the first three risks. The first statement addresses denial of service. The second statement addresses masquerade. That statement also addresses disclosure and forgery indirectly, but we will address those more directly with additional policy statements. The third policy statement directly addresses denial of service and subversion.

User Isolation Policy

Let's return to Bob's predicament. He has a computer that he shares with his suitemates. He will let them use the computer but he needs to protect his own information assets.

TABLE 3.2 Underlying system policy for a shared computer

#	Policy Statement	Risks
1	All users shall be able to execute customary application programs and operating system service programs.	1
2	Each user shall have a separate login and, optionally, a password.	3
3	Programs shall be protected from damage or other modifications by regular users.	1, 2

TABLE 3.3 Policy for user isolation

#	Policy Statement	Risks
1	All users shall be able to execute customary application programs and operating system service programs.	1
2	Each user shall have a separate login and, optionally, a password.	4
3	Programs shall be protected from damage or other modifications by regular users.	1, 3
4	Files belonging to one user shall be protected from any access (read or write) by other users.	1, 2, 5

Bob is looking for a policy in which we adjust Objective **c** to isolate users from one another. We create the basic policy by adding one statement to the fundamental system policy. The statement directly addresses the risks of disclosure and forgery. It also addresses the risk of denial of service if one user accidentally damages a file belonging to another user. This yields the policy in Table 3.3.

However, Bob had a specific risk in mind:

6. A suite mate tampering with Bob's files.

To protect against that specific risk, we can add some more specific security policy statements to Bob's security policy. In particular, we can add policy statements to talk specifically about how Bob and his suitemates will use the computer. These additional statements appear in Table 3.4.

If Bob configures Windows in a friendly fashion, it will display a login screen that lists all users on the computer. If a suite mate clicks on "Suitemates," the computer will log in without a password. If a suite mate clicks on "Bob," the computer asks for the password.

TABLE 3.4 Policy additions for Bob's particular situation

#	Policy Statement	Risks
5	The system shall have two regular users: Bob and Suitemates.	4, 6
6	Bob shall have a password to protect his login.	2, 4, 5, 6
7	Suitemates shall not need to type in a password to log in.	1

TABLE 3.5 Policy for file sharing

#	Policy Statement	Risks
1	All users shall be able to execute customary application programs and operating system service programs.	1
2	Each user shall have a separate login and, optionally, a password.	4
3	Programs shall be protected from damage or other modifications by regular users.	1, 3
4	Files belonging to one user shall be readable by other users.	1
5	Files belonging to one user shall be protected from writing by other users.	1, 3, 5

This arrangement provides Bob with the safety he needs and the suitemates with access to his computer. The suitemates can't log in as Bob unless they somehow guess his password. On the other hand, they don't have to remember a password to log in as "Suitemates." When they log in as such, they won't have access to his files (per Policy 4).

User File-Sharing Policy

When people hide things, others get suspicious. There are some environments where it's more common to share information than to hide it. For example, many families want to share their information among members. Files may consist of letters, homework, family photos, and so on, and there may be bona fide reasons to share anything and everything. In engineering projects, it is fairly common to share information.

If we assume sharing is the norm, then Risk 2, disclosure, is no longer considered a risk. Instead, it becomes a "denial of service" if we can't read each others' files. This yields the policy in Table 3.5.

As with user separation, there are some systems that provide sharing by default. This is a tradition of Unix-based systems, which create file directories for individual users and, by default, grant other users access to those files. The resulting policy does not, of course, provide unrestricted sharing. A typical user-sharing policy provides *read-only* sharing, but protects files from writing except by the owner.

3.4 Security Controls for Files

In Section 2.5, we introduced the access matrix to describe access controls for processes in RAM. We can use the same matrix to describe access controls for files and other resources in an operating system. In general, the access matrix describes three things:

- What we are sharing (***objects***)

- With whom we share them (***subjects***)
- What ***rights*** each subject has to each object

If we make very precise statements about the subjects, objects, and rights we wish to implement, then the sharing is easy to implement. If we trust the system to block all accesses *except* those we specifically allow, then it's easy to verify the system's correct behavior. The more specific and mechanical we can be in our specification of subjects, objects, and rights, the more effectively we can implement and verify our policy.

In practice, however, a realistic policy statement is more abstract than that. We want our policy to associate *people* with *data*; however, that is not the same as associating *user identities* with *files*. We build mechanisms to relate the two as closely as possible, but they just aren't the same. We want to protect what is *in* the files, and not just the computer files themselves.

This is a security problem we also have with boundaries, fences, and other containers in general: The security mechanism controls what happens to the container, not the contents. For example, we could try to protect some valuables with a high fence, border alarms, and a strongly locked gate. Still, a helicopter could drop in and steal some items, bypassing the fence and raising no alarm.

We rely on a computer's operating system to protect its files. If an attacker starts the computer normally, our operating system will start up and protect our files. We still, however, face a Chain of Control risk. A system booted from a CD-ROM bypasses the hard drive's own operating system and allows unrestricted access to our files. We can rely only on operating system protections as long as two properties remain true:

1. The operating system's protections are always invoked when accessing our protected files, and
2. There is no way to bypass the operating system to access our protected files.

The terms *subject* and *object* are well-known in the information security community. They most often appear in highly formalized security specifications and aren't used routinely. We will use more specific terms like *process* or *user* instead of *subject*, and terms like *file* or *resource* instead of *object*.

3.4.1 Deny by Default: A Basic Principle

In a clean kitchen, everything has its proper place and everything is put away. Boxes and jars are closed and stored in cabinets. A few items may remain out if they are to be used soon, or if they are not going to draw insects, rodents, or other vermin.

In essence, the "default" is to deny access to vermin, at the very least. This arrangement also denies access to legitimate users (members of the household) to a small degree, because a person must take specific steps to gain access to the desired item: open a cabinet and a box or jar. A padlock provides a more explicit example: We can't open the door unless we have the key (Figure 3.10).

Figure 3.10
Deny by Default--a padlocked door.

Deny by Default provides the foundation for strong security. In computing, it means:

No access is allowed to anyone unless specifically granted.

As we saw in Section 2.4, every process starts out as an "island." A process has no access to resources unless the system gives the process a way to request them and reach them. It isn't easy to show that processes are truly isolated, but it is a fundamental design goal of every secure system. We need confidence that a system *only* grants access rights when told to do so. Otherwise we can't fully trust a system to enforce the permissions and restrictions we want.

The Opposite of Deny by Default

This may sound contradictory, but we can't trust a system to share things correctly unless it starts by blocking access to everything. This makes sense when we consider the opposite: a system that allows all accesses *except* the ones we specifically tell it to block. We're now faced with the challenge of listing *everyone* we might need to block. How can we possibly tell if that list is complete? Do we start a list of every person on the planet and remove the names of permitted users? This isn't practical.

For example, ***content control software*** (also called ***censorware***) classifies Internet websites according to content, and then blocks access to types of content the customer finds unacceptable. Some private organizations use this software to discourage employees from visiting personal, shopping, and entertainment sites with their workplace computers. A few countries, notably China and Saudi Arabia, apply such filtering to *all* Internet traffic that enters and leaves the country. We examine this further in Chapter 15.

Content control software often follows a policy of ***default permit***: Everything is allowed *except* sites on the prohibited list. This doesn't really block access to all "unacceptable" sites, it only blocks access to sites that have existed long enough to find their way onto the list of banned sites. Such systems can't be 100 percent effective, though they work well enough in practice to satisfy certain customers.

Islands, vaults, and other physical boundaries often implement a Deny by Default policy. People can't go through a physical fence that's too tall or risky to climb. The fence's gateway implements the permission to pass. Some fences are designed to restrict animals while others are intended to restrict people. The intended policy of a well-built fence is reflected in the structure of the fence and the lock on the gate. A fence intended to restrict animals may have a gate that people can easily operate.

3.4.2 Managing Access Rights

In order to enforce the correct access rights, the operating system needs to maintain a list of access rights. In theory, the system could maintain an access matrix. In practice, this isn't efficient.

For example, a typical Windows desktop contains over 13,000 separate files just for operating system components, and may contain five to ten times as many files for programs, media, and user-created documents. The complete access matrix might contain 100,000 rows, plus one column per user, or even one column per process. This is too large to manage.

Access rights follow well-known patterns in most systems. When one process starts another, the child process often inherits the parent's access rights, especially to files. When this happens, we can apply the same set of access rights to both the parent and its children. If we look at individual resources, particularly files, we find that the rights tend to cluster into three distinct groups: those granted to the owner, those granted to "the system" (including system administrators), and those granted to "everyone else." Files also may have a cluster of similar permissions granted to a "group" of users working on a shared project.

In other words, the access matrix contains a lot of redundant information. We can eliminate this redundancy by taking advantage of patterns in the access rights. When the system combines matching access rights, it yields a smaller set of rules that is easier to manage and enforce. Starting from the access matrix, there are two obvious strategies for combining access rights:

1. Cluster by column: associate access rights with users or processes
2. Cluster by row: associate access rights with resources like files.

When we cluster the rights by column, we implement ***capability-based security***. This is common in physical, real-world security situations: Bob, for example, carries keys to let him in to his suite, start his car, and open the front door back home. Bob's access rights are embodied in the keys, and the keys stay with the owner (the subject).

If Alice attends a concert, she buys tickets, and the tickets themselves impart access to the concert.

3.4.3 Capabilities

The Atlas system at Manchester and CTSS at MIT demonstrated that a computer could run many separate users at once, keeping them safely separated. Researchers at MIT, notably Earl Van Horn and Jack Dennis, tried to identify the essential features of such systems. Van Horn coined the term ***capability*** to describe the access rights a particular subject had for a particular object: the information in a single cell of an access matrix. Because operating systems always kept a list of resources belonging to each process, Dennis and Van Horn suggested that access control information be kept in that list.

When an operating system implements capability-based security, it keeps the list of access rights on a per-process basis. If a process has access rights to particular files or other resources, it carries those rights with it. The process also may be able to share or transfer its access rights to other processes. To print a file, for example, Bob's word-processing program passes the right to read his file to the printer's device driver.

This approach seemed reasonable because capability-based security seems to combine both the "island" and the "vault" access control strategies. A process can only locate a resource if it is on its capability list. If the resource appears on the list, then the system controls its access even further by including or omitting the individual access rights. When Butler Lampson introduced the access matrix in 1971, he predicted that capability-based access control would be less "cumbersome" than the column-based alternative.

Capabilities in Practice

Despite Lampson's prediction, capability-based security is not an obvious feature in major modern operating systems. The access controls that most of us see—those for protecting files and folders—rarely use capability lists. Several research systems were developed in the 1960s and 1970s that used capability lists, but very few commercial systems adopted them. The only really successful commercial system that used capability-based security was the IBM System/38 and its descendent, the AS-400.

We see capability-based security in more subtle places. For example, a process page table provides capabilities to use specific RAM areas. Network-based systems also use capabilities. For example, the Kerberos system (Section 13.4.2) distributes "tickets" to its users that in turn give those users access to particular resources. Microsoft Windows uses Kerberos to distribute user access rights between host computers on Windows networks. The U.S. government issues "public-key certificates" (Section 8.6) to members of the defense community. Each certificate identifies the individual and specifies certain access rights, like security clearances.

Resource-Oriented Permissions

Most operating systems, however, use the row-oriented approach: They store access rights with files and other resources. This is a less-common approach in real-world security; we mostly see it used for handling reservations in hotels and restaurants. If Bob takes his date to a popular restaurant, the maitre d' checks his name against the list of reservations, and only grants access if Bob is on the list. To take a typical airplane flight, Alice presents a photo ID at the airport, and the airline uses it to generate her boarding pass. Her access to the plane is based on her identity, not on possessing a ticket.

Within operating systems, the obvious example of this approach is in the file system. The system stores details about access rights along with other administrative data it maintains about a file. If we wish to find out about access rights to a particular file, we look at the file's properties. On Windows, we select the file and then select "Properties" from the file's menu. The "Security" tab lists the access rights. On Macintosh OS-X, we select the file and select "Information" from the menu. A section at the bottom of the Information window lists the access rights (Figure 3.11).

OS-X provides a specific permission ("Read Only" or "Read/Write") associated with the user who owns the file and a separate permission for the rest of the world. OS-X

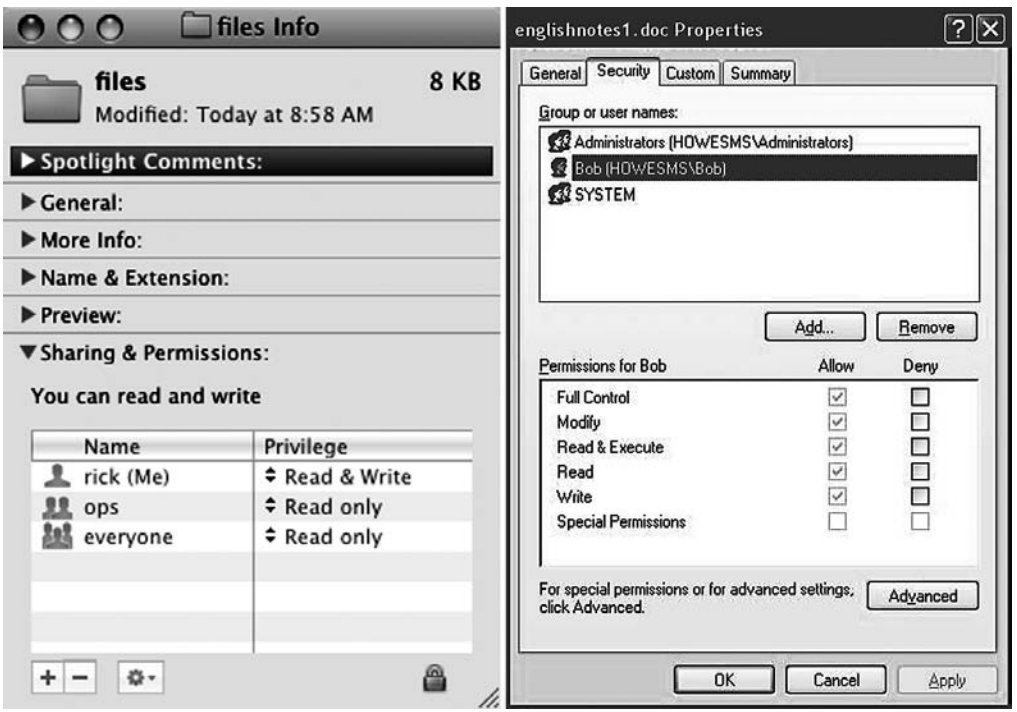

Figure 3.11

Resource-oriented permissions OS-X and Windows.

doesn't display any rights if the world has no rights. The display also shows if there are rights granted to a particular "group" of users.

3.5 File Security Controls

Let us look at the access permissions required by users on Bob's system to implement the policy for Bob's tower computer. The policy appears in Tables 3.3 and 3.4. First, let us look at access rights for executable programs. Figure 3.12 illustrates the appropriate access rights, using two executable files as examples. Regular users receive Read and Execute access. The "System" user receives full access to the file.

Figure 3.13 uses a slightly different approach to show access rights to user files. Instead of showing example files, it shows a "pile" of files to represent those belonging to the two regular users.

The two diagrams represent all of the users on Bob's system. In larger systems, we might use a single oval to represent all "regular" users and present permissions in those terms. We also could represent permissions with separate ovals to represent "the file owner," the "system," and "other users."

3.5.1 File Permission Flags

Real systems rarely use diagrams to establish access rights; instead, there is a notation or a tabular format to use. The simplest approach is to provide a set of ***permission flags*** to identify rights for a specific set of users.

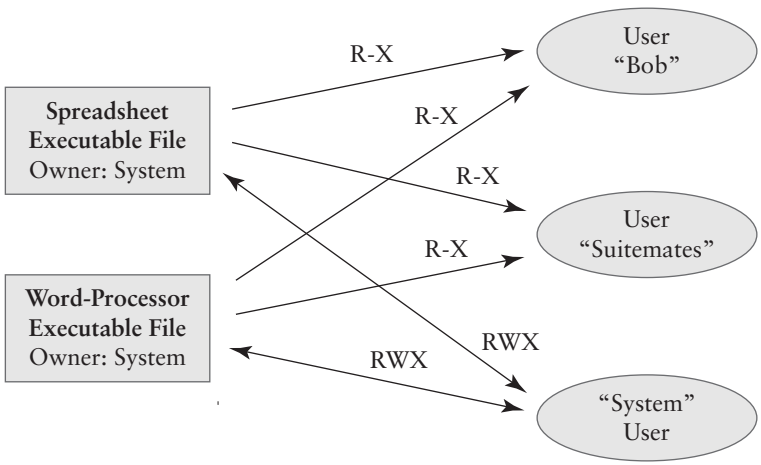

Figure 3.12

Access rights to executable files.

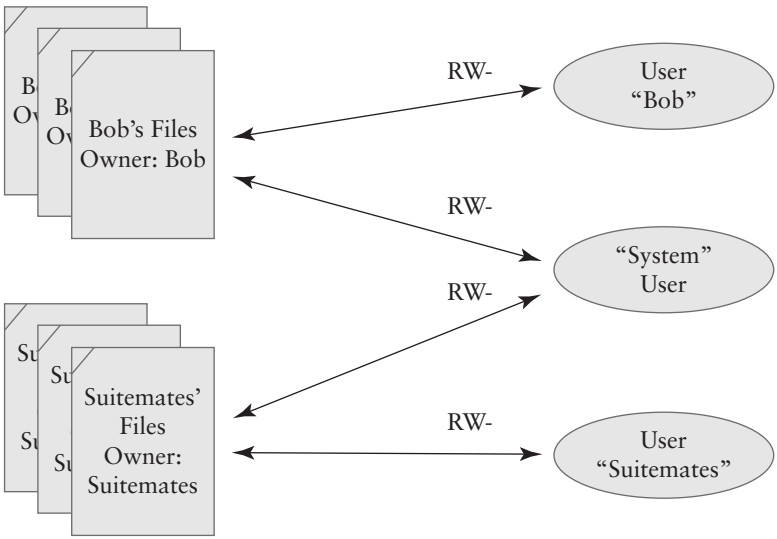

Figure 3.13

Access rights to personal files.

In Figure 3.14, we have three sets of flags. Each set specifies the basic rights of Read, Write, and Execute (or Seek for directories), and associates the rights with one of three specific classes of users:

- ***Owner rights***—the user who owns the file in question
- ***System rights***—privileged processes run by the system
- ***World rights***—all users other than the owner and the system

This allows us to define access rights with a compact set of nine flags, along with the user identity of the file's owner.

Because the operating system already keeps track of the owner of every file and other major resources, our solution doesn't need to explicitly identify users. All it needs to do

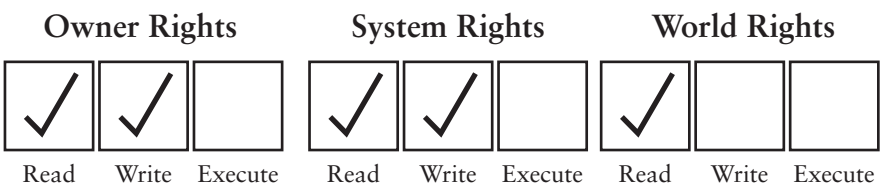

Figure 3.14

A set of file permission flags.

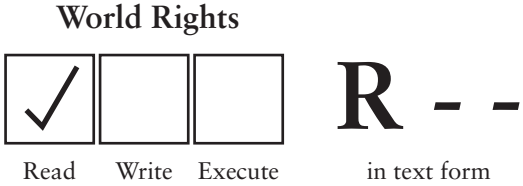

Figure 3.15

Flags for compact access rules.

is identify the rights granted to the owner, the system, and the rest of the user community (the "world"). A user with administrative rights receives "System" access rights to files and resources. Most systems also can grant access rights to predefined "groups" of users; we will discuss that in Chapter 4.

Although we've reduced the rights to a very compact set, they still carry a great deal of redundancy. Typically, owners get full access to their files. Administrators and privileged user identities associated with system operations, like a "System" user, typically get full access to everything. The important differences in access rights arise when we specify access rights for the rest of the user community.

In most examples and exercises in this chapter, we use a simplified set of rights called ***compact access rules***. Figure 3.15 illustrates such a rule.

Compact access rules reflect two common assumptions. First, they assume that the owner of a file or folder has full access rights to it. Second, they assume that system processes, or those running under a "System" user ID, have full access rights to all files and folders on the computer. We may safely use these assumptions while working problems in this chapter.

We abbreviate a compact access rule with three letters. As shown in Figure 3.15, the letters represent read, write, and execute rights. Thus, the permission R-E allows read and execute to other users, while --- allows no access to other users.

System and Owner Access Rights in Practice

If our examples omit owner and system access rights, we should ask some obvious questions: Why would *any* system allow us to specify owner and system access rights? Why would we ever want to restrict access by a file's owner or by the system? The answer is, of course, Least Privilege. Access restrictions sometimes make sense for individual owners or for the overall system.

For example, some owners explicitly set valuable files to be read only simply to protect the files from accidental damage. If the owner edits the file by mistake or runs a program that might change it somehow, the file system will deny access.

Operating systems are large and complex, so Least Privilege helps keep components from injuring one another. Early versions of the Unix system, for example, ran most

system processes with the "root" user identity. As we saw in Section 2.3, this led to disaster when the Morris worm hit. Modern Unix systems, and their derivatives like OS-X and Linux, run as few processes as possible using the "root" user identity.

When we create a new file on Microsoft Windows, it will, by default, be accessible by the "SYSTEM" user identity and by administrators. We can remove those permissions and still, as the owner, read and write the file. The access permissions apply to any processes that belong to the "SYSTEM" user identity, while the true "kernel" of the operating system is not really restricted by such rules. Thus, we can still read and write our file even though "SYSTEM" processes cannot.

On some high-security systems, the security policy forbids administrators to look at certain types of sensitive information stored on the computer. Such systems should block access to that information, even when using a "SYSTEM" user identity, unless the specific individual is allowed to use the sensitive data. Thus, some administrators may be able to look at the highly sensitive data while others are blocked from that data.

In this chapter, we focus on simple cases and assume that both the file's owner and the system have full access rights to it. The only access rights we need to specify are for "world" access to files.

3.5.2 Security Controls to Enforce Bob's Policy

The diagrams are useful to illustrate how users can (or can't) share information on the system. To actually implement a policy, we produce a more detailed list of security controls. Bob's isolation policy, described in Tables 3.3 and 3.4, requires some general security controls and some specific access rules. These appear in the following tables.

First, we specify the general security controls in Table 3.6. These controls establish user identities and passwords on Bob's computer. These controls follow the format introduced in Section 2.6.

TABLE 3.6 Security controls to implement Bob's isolation policy

#	Control Category	Description	Policy Statement
1	Logical	Bob has a personal user identity on the computer.	2, 5, 6
2	Logical	Bob must use a password to log in.	6
3	Logical	Suitemates have a single user identity on the computer that they all share.	2, 7
4	Logical	Suitemates do not need to use a password to log in.	7

TABLE 3.7 Compact access rules for Bob's isolation policy

#	File Type	File Owner	World Access	Policy Statement
6	Executable programs	System	R-X	3, 4
7	Bob's data files	Bob	---	5
8	Bob's directory	Bob	---	5
9	Suitemates' data files	Suitemates	---	5
10	Suitemates' directory	Suitemates	---	5

To specify access rules, we need a different format. We will use a simple format that makes use of compact access rules; it lists the three-letter abbreviation for the "World" access rights, and also identifies the file's owner.

To simplify matters further, we list "types" of files instead of listing all files individually. We assume that the specified access rights are associated with all files of each specified type and owner. The resulting table of compact access rules appears in Table 3.7.

Note that the "File owner" column does not consistently refer to Bob even though he is the owner of the computer. The column indicates the computer user that owns the particular file. These rules assume Bob logs in as the user "Bob" and that his suitemates log in as "Suitemates." If Bob always logs in as an administrator, then he gets "System" access rights, which gives him access to every file on his computer.

3.5.3 States and State Diagrams

States and ***state diagrams*** give us a different way of organizing and looking at parts of a complex system. While we can always look at a system by breaking it into pieces and examining individual parts, a state diagram gives us a way to systematically break up its *behavior*.

Engineers apply state diagrams to all kinds of systems. At the circuit level, engineers can implement a diagram directly in hardware. At the system level, states and state diagrams provide a way to break complicated system behavior into separate and more comprehensible activities.

The individual "states" reduce the system's behavior into a manageable number of separate situations. The state diagram gives us a simple image of how the system behaves. For example, Figure 3.16 presents a state diagram for a door.

To create a state diagram, we look at the system and divide its behavior into a small number of completely separate conditions. Each condition becomes a "state." Then we

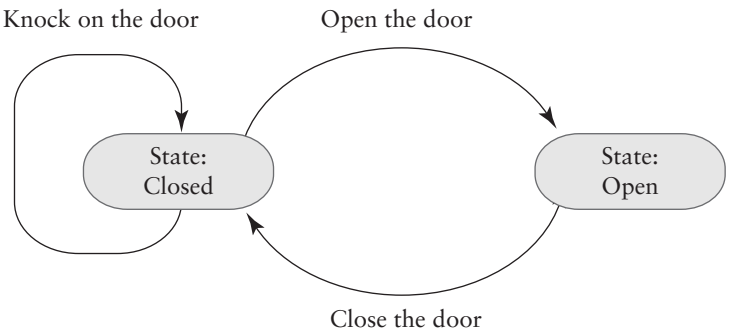

Figure 3.16
State diagram showing a door's operation.

identify events that take place, especially those that cause the system to change from one state into another.

In Figure 3.16, we have two states because a door can only be "Open" or "Closed." We have three possible events: knocking, opening, and closing. The door actually changes state when we open or close it. Knocking causes no change at all.

For the state diagram to be useful, it must capture *all* relevant and possible events, especially those that change states. Figure 3.16 presents a very limited view of a door, because it says nothing about locking or unlocking, or even about responses to knocking. In fact, knocking is clearly unnecessary, because the diagram shows the door opens and closes regardless of whether knocking took place.

Information States

We also talk about data or information as having states. Here are the fundamental ***information states***:

- ***Storage state***—the information is stored and is not currently being processed. We also call this state "data at rest."
- ***Processing state***—the information is being used by an active process to make decisions or to transform the information into another form. This is "data in use."
- ***Transmission state***—the information is being moved from one storage area to another. This is "data in motion."

Figure 3.17 shows a state diagram for Bob's essay file. We initially create files in an application program, so Bob creates it in the Processing state. When he saves the file, he puts it into the Storage state. If he moves it onto a removable drive, it enters the Transmission state.

In this diagram, we process either new or stored data. We save the result as stored data. We don't use processing to transport data; we simply move the data to a portable

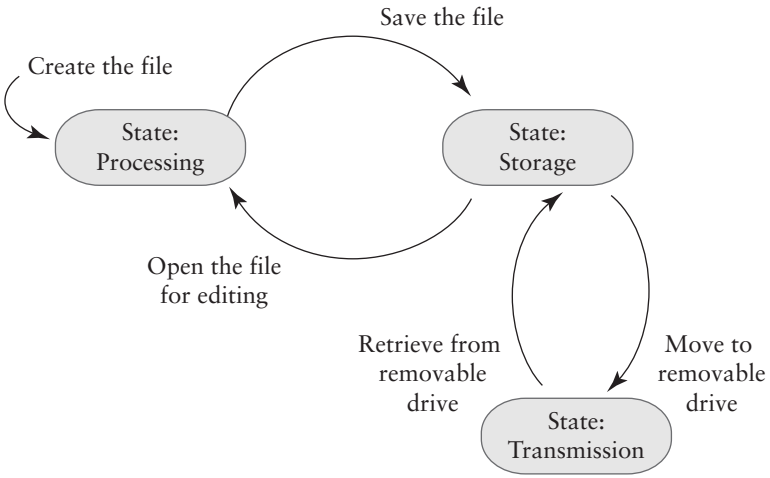

Figure 3.17

Information states of the essay file.

medium. In this environment, data never moves directly from the Processing state to the Transmission state.

A closed door has different security implications than an open one. The same is true for information states. Each reflects a different access control environment. The storage state involves files and their access rights. The processing state refers to the world of processes and RAM-based access control.

When we transmit information between programs or processes inside an operating system, the storage and processing controls protect those operations. When we move a file to removable storage, like a USB drive, we move it away from the protections enforced by our operating system. This also happens when we transmit information across a network, as we will see in Chapter 10.

3.6 Patching Security Flaws

Technical systems *always* have flaws. In the 19th century, telegraph and telephone technicians spoke of "bugs in the wire" as the source of unexplained buzzing noises and other hard-to-fix problems. We still chase bugs in technical hardware and in software.

Although system developers would like to build error-free software, this has proven impractical. Social and economic reasons encourage people to buy lower quality software at lower prices. Software vendors then provide bug fixes when existing flaws cause too much trouble, or worse, open customer computers to attack.

The Patching Process

Most software developers use a carefully designed process to handle bug fixes. The process collects problem reports and eventually develops a ***software patch*** to fix the problem. A patch is a piece of binary data that modifies the instructions in a program to correct a problem. Most modern computing systems have special software to apply patches to existing software.

The alternative to patching is to reinstall a new copy of the original software. Modern software packages often grow quite large. It is more efficient to distribute a patch, even if it contains several million bytes of data, than to replace a 100-million byte program.

There are always more problems to fix than resources to fix them. Developers use the process to prioritize flaws, develop fixes, and distribute fixes. A typical process follows these steps:

1. Collect error reports. These may come directly from individual customers, from companies that use the software, or from news reports of problems. In the worst case, error reports may come from national flaw reports, like the Common Vulnerabilities and Exposures database.
2. Prioritize errors and assign for investigation. A team of software engineers reviews the error reports and assigns the highest priority ones to engineers to investigate and to try to fix.
3. The engineer develops a fix for the software problem. This yields a change to the software containing the flaw.
4. Another team of engineers reviews proposed bug fixes for release in an upcoming patch. The team selects the fixes to include, and integrates those fixes into a new patch.
5. Test the patch. A test team applies the patch to different versions of the software and tests them for correctness and stability. The vendor doesn't want to release a bug fix that makes matters worse.
6. Release the patch. Software engineers package the bug fixes into a patch file for automatic installation. They place the patch on the vendor's website and ensure that automatic patching software can find and install it.

Many bugs may take weeks or months to work their way through this process. Higher priority bugs, like those that pose a serious security risk, may be fixed sooner. Given a sufficiently serious bug, the process may only take a matter of days.

Security Flaws and Exploits

In the ideal case, a small number of people in the security community find out about a software security flaw and the vendor develops a patch for it. The general public doesn't really learn about the flaw until the patch appears. Then a race begins between black-

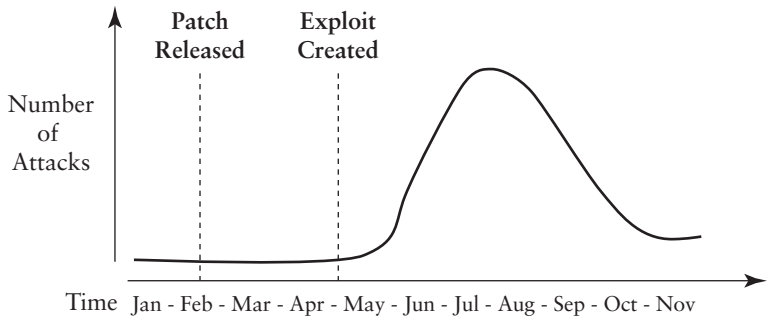

Figure 3.18

Time line for attacking unpatched flaws.

hat hackers who want to use that vulnerability to attack systems, and end users who want to patch their systems.

Many end users fail to patch their computers, even when patches are available for known vulnerabilities. Figure 3.18 summarizes the typical patching experience in the late 1990s. The vendor issues a patch in January. By March, black hats have reverse-engineered the patch to figure out the vulnerability, and they construct an ***exploit***: malware that uses the vulnerability in an attack. The black hats then search out and attack the unpatched systems.

Using network scanning software, black hats could search the Internet and identify unpatched computers. The exploit software then successfully attacks those systems. Matters may have improved somewhat in recent years, since the patching process has become more efficient, reliable, and automatic. Technophobic users can set their computers to update software automatically. Even so, countless computers often go unpatched.

Windows of Vulnerability

Today, every potential vulnerability represents a race between the white-hat and black-hat communities. White hats, including responsible security researchers and vendors, try to find security flaws and deploy patches before exploits arise. Black hats try to identify flaws and construct exploits before the patches appear. The ideal goal for a black hat is to construct a ***zero-day exploit***: one for which no patch yet exists.

Figure 3.19 illustrates this race between black hats and white hats using a state diagram. The diagram follows the history of a security vulnerability, relative to a particular piece of software installed by a particular end user. The software starts out in the Hardened state, which means that the user has patched it for all known weaknesses. The software enters the Flawed state when someone—white hat or black—finds a flaw.

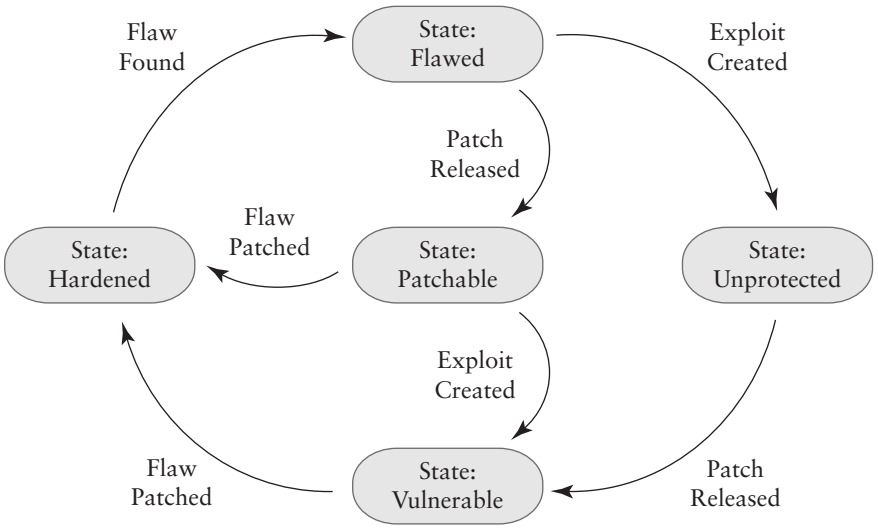

Figure 3.19

Software vulnerability state diagram.

If matters progress as shown in Figure 3.18, then the vendor releases a patch and the software enters the Patchable state. If the user immediately patches the software, it returns to the Hardened state. Otherwise, it moves to the Vulnerable state when an exploit appears.

The software remains vulnerable while an exploit exists and the software remains unpatched. We call this the ***window of vulnerability***: the period of time during which a system is unprotected from an exploit. As long as the software remains unpatched in the face of a possible exploit, it is vulnerable to attack.

If, however, we face a zero-day exploit, then the software moves to the Unprotected state. The user has no choice but to be stuck in a window of vulnerability until the vendor releases the patch. The only alternative is to stop using the vulnerable software.

The Unprotected state also may arise when we install new software from a CD or other distribution media. The distribution disk may contain an older version of the software that lacks the latest patches. Thus, the newly installed software may start out in the Unprotected state.

This was a particular problem with Windows XP, the version used in the early 2000s. Various estimates in 2005 suggested that an unpatched Windows XP machine would only last a matter of minutes before being attacked on the Internet. When a user installed Windows XP from a distribution CD, the system would remain Unprotected while it connected to the Internet and downloaded the patches needed to reach a Hardened state.

3.7 Process Example: The Horse

Helen eyed the horse suspiciously. She accompanied Husband Number Three, who was known as Deiphobus, Prince of Troy.

Following Helen's kidnapping 10 years earlier (or was it an elopement—no one knew for sure), the Greeks had been making war on the city of Troy.

Now the Greek camp was empty and burned. Their ships were gone. All that remained was a huge wooden statue of a horse (Figure 3.20). The Greeks also had left a messenger; he claimed the horse was an offering to the goddess Athena for their safe trip home.

The people of Troy already had dragged the huge statue through the gates and into the city.

Helen walked around the horse with her prince, examining it closely. Were warriors hidden inside? Helen called out the names of famous Greek warriors, trying to mimic the sound of their wives' voices. No Greek voices responded. Perhaps she suspected what Odysseus told her later; he was keeping the Greeks silent while she tried to trick them into speaking.

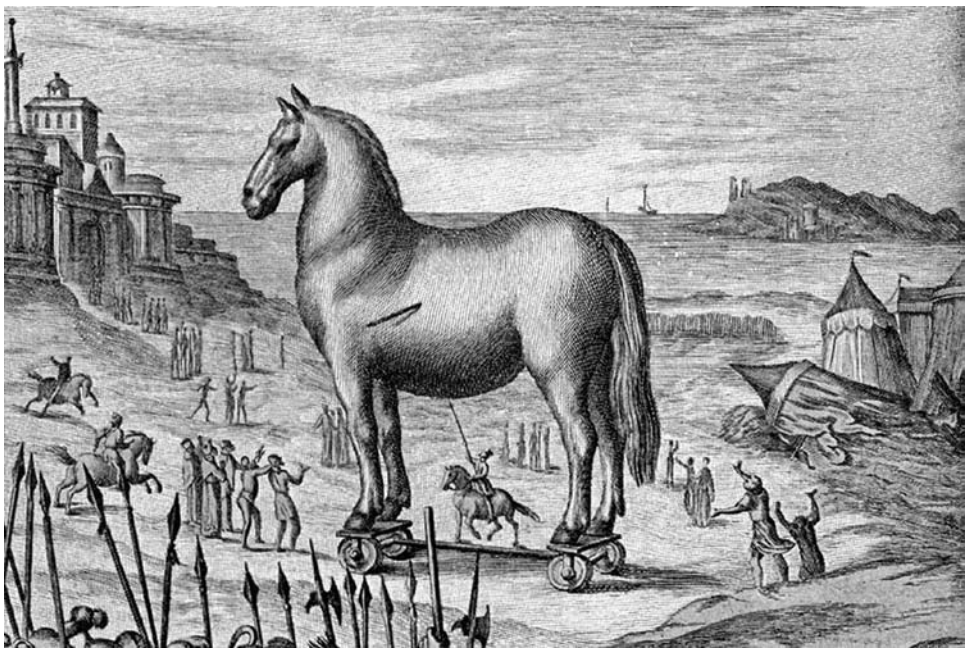

Figure 3.20

Drawing of the Trojan horse. From *The Children's Hour,* Tappan, ed., 1907.

Like Helen, many Trojans saw the horse as a menace. They said it should be broken up, burned, or dragged to the city's heights and pushed off, letting it fall to its ruin.

Others argued that the statue should be kept as an offering to the gods while the city celebrated its victory against the Greeks. This argument was enough to satisfy most people and the celebrations began in earnest.

Helen's suspicions were, of course, correct.

Hours later, a force of Greeks emerged from the statue. They found the city asleep after the night of festivities. The soldiers opened the city gate to admit the rest of the Greek army. Troy soon fell, as the Greeks defeated the Trojans inside the city walls.

3.7.1 Troy: A High-Level Analysis

Although the technology of the Trojan horse might not directly apply to modern computing, it is a popular symbol in computer security. When a computer program contains secret, malicious behavior that may injure its unsuspecting user, we call it a Trojan horse. For some experts, a "Trojan" virus is one that performs a damaging act in addition to spreading itself around. Other experts consider the spreading process itself to be damaging, and classify all viruses as Trojan horse programs.

However, this terminology is not relevant to this particular section: we will apply the six-part security process to the original Trojan horse.

Assets: For this discussion, we will focus on a single Trojan asset: their city. For the city to thrive, the people need food and trade, and conquerors must be kept away.

Risks: During the war, the Trojans were at risk of attack from the Greeks. The ancient city of Troy was protected by high walls that made it invulnerable to direct assaults by soldiers on foot or on horseback. Primitive weapons of the time could not breach those walls while defenders fought back.

The remaining weakness was at the city gates, which needed to be opened for trade and provisioning, and closed when the Greek army moved to attack the city.

Policy: Since we are focusing on the safety of the city of Troy itself, the security policy would simply forbid enemy warriors from entering the city. The policy would further specify the boundary to be the city's defensible walls. If the Greeks were kept outside, the Trojan defenders could rely on strong gates and high walls.

Implementation: We have specified the policy in terms of the city's boundary, and the boundary is implemented by the city's defensive walls.

The city has a main gate. Just like a door has two states, the city's defenses had two states, based on the condition of the main gate.

In the first state, the city gates were closed. We assume that the city does not contain enemy warriors. If that is true, then as long as the gate remain closed, no enemy warriors can invade the city. However, the gate cannot remain closed indefinitely. It must

be opened occasionally to allow traffic for trade and provisions, or the population would starve.

In the second state, the gate was opened to allow food and traders to enter the city. Enemy warriors, however, were not allowed in the city. If any were spotted, they were captured. If an enemy army approached the city while the gate was opened, the city switched to its other state by closing its gate.

Monitoring: Trojan warriors guarded the city against the entry of Greek warriors and, in general, the Greeks stayed out. On one occasion, however, the warrior Odysseus himself dressed as a beggar and snuck into the city. Small incidents like that did not lead to disaster.

The Trojan horse allowed a team of Greek warriors into the city. Helen clearly suspected this, but her security analysis was not convincing to the rest of the city. The horse was left unguarded as the city prematurely celebrated its victory.

Recovery: Unfortunately for Troy, the final Greek attack was thoroughly successful. The archeological site attributed to Troy was destroyed in the 13th century BC and another city did not develop there for over 1200 years.

3.7.2 Analyzing the Security Failure

Let us examine the fall of Troy in terms of the process just described. The Trojans suffered failures in monitoring, in implementation, and in policy.

Monitoring: Despite the care shown by Helen and her prince in examining the horse, they did not detect the warriors hidden inside. They did not post a guard around the horse, despite their suspicions.

Implementation: There was no process in place to prevent the unusual Greek gift, the wooden horse, from being dragged into the city by the city's populace. Once the horse was in the city, it remained unguarded while the city celebrated victory over the Greeks.

Policy: The Greeks had burned their camp and their ships were gone. Many Trojans believed that the Greek threat was over and that wartime security policies no longer applied. No lookouts on the walls or guards at the city gates sounded the alarm when the Greeks marched up to the city. The populace had, in effect, nullified the policies by ignoring them.

Failures teach important lessons about security, even when the event doesn't involve computers or technology.

3.8 Chapter Resources

IMPORTANT TERMS INTRODUCED

capability
capability-based security
censorware
compact access rules
content control software
default permit
Deny by Default
directory path
drive-by download
execute right
exploit
file folder
file-sharing policy
global policy
hierarchical directory
information states
interpreter
isolation policy
macro
objects
owner rights
permission flags
processing state
rights
root directory
scripting language
seek right
shell script
software patch
state diagrams
states
storage state
subject
system rights
tailored policy
transmission state
Trojan
window of vulnerability
world rights
zero-day exploit

ACRONYMS INTRODUCED

CRUD—Create, read, update, delete access rights
MBR—Master boot record
PDF—Adobe portable document format
RWX—Read, write, execute access rights

3.8.1 Review Questions

R1. Explain the role of a file name and path in locating a file on a hard drive.

R2. Describe the four basic access rights for files and other resources in general.

R3. Give reasons why a user would protect a file from read or write access by other users.

R4. How does the operating system decide what permissions to apply when a user creates a new file?

R5. Explain how the four basic access rights of files and directories interact.

R6. What does it mean to have "Execute" access to a file?

R7. What is "Seek" access and how is it different from "Read" access?

R8. Describe the format of an executable file.

R9. Why would we restrict access to executable files?

R10. Describe how a virus operates and spreads.

R11. Explain the difference between a virus, a worm, and a Trojan.

R12. Summarize the policy for enforcing *isolation* among users.

R13. Summarize the policy to provide *file sharing* among users.

R14. When we wish to specify file-access rights, which elements serve as *subjects* and *objects* in the access matrix?

R15. Explain the difference between a default permit policy and one that enforces Deny by Default.

R16. Name the two requirements that must remain true in order for an operating system to enforce its policy.

R17. Why do we say that "capabilities" represent the clustering of access rights "by column"?

R18. Give examples of systems that use capability-based security.

R19. Do most modern operating systems specify file permissions with a "cluster by row" or "cluster by column" strategy?

R20. Summarize the information needed to specify a file's access rights using permission flags.

R21. Describe the differences between listing customary security controls (Table 3.6) and compact access rules (Table 3.7).

R22. Describe the differences between an access matrix and a table of compact access rules.

R23. Explain how the Morris worm took advantage of a failure to use Least Privilege.

R24. Describe the components of a state diagram.

R25. List the typical steps a vendor follows to release a software patch.

R26. Explain two different situations in which a window of vulnerability might arise.

3.8.2 Exercises

E1. Search the directory on your computer. Locate the root folder. From there, locate the following files and provide the full path and file name for each:

a. The word-processing program you typically use.

b. A text or word-processing file in your "documents" directory (the directory in which programs typically save files for you).

c. A file containing your "preferences" for a program like a web browser or word processor.

d. A file you downloaded from the Internet using your browser.

E2. Can you create files in the root directory of your system? If so, create a file or folder and then delete it. If the attempt fails, describe the error message displayed.

E3. Determine whether the system you use implements an isolation policy or a user-sharing policy. Describe the steps you took to verify your conclusion.

E4. Determine whether the system you use will allow a file's owner to block access to an owned file. Create a file and try to remove the owner's Read or Write permission from the file. Did the system remove the right or not? How could you tell?

E5. Find out about antivirus software on the computer you use.

a. Is antivirus software installed?
b. What kind is it?
c. Can you tell if it works? Why or why not?
d. Has it ever reported a virus to you? If so, describe how it handled the virus.

E6. Using the story of the Trojan horse as an analogy, give two examples of "Trojan" attacks in computer systems. Explain why the examples fit the analogy of the Trojan horse.

E7. Search the Internet for information on different malware packages, like those discussed in the text. Find a description of a malware package *not* described in the text. Provide the following information about the malware:

a. What is the malware called?
b. How does it propagate?
c. What does it do to the computers it infects?

E8. Bob would like to be able to look at files in the "Suitemates" folders without having to log in as Suitemates. Take these steps to create a security plan to achieve this goal.

a. Create a revised policy to reflect this objective.
b. Draw a diagram illustrating access rights that implement this policy.
c. Create a table of compact access rules that implements the revised policy. Use the format shown in Table 3.7.

E9. Create a diagram portraying access rules that enforces the file-sharing policy described in Table 3.5.

E10. Create a table of compact access rules that enforces the file-sharing policy described in Table 3.5. The compact access rules should use the same format as Table 3.7.

E11. Alice's computer was infected by Virus X, which attached itself to all of her applications. Bob lent her a file-searching utility, which was stored on a USB stick. This utility, however, was infected by Virus Y, which then infected all of Alice's applications, too. Thus, each application contained *two* virus infections. Draw a diagram based on Figure 3.7 to show both infections in a single application file.

Riko is writing a program. Bob's computer contains a compiler that will take Riko's source code (the program she's written) and produce an executable program file (with a ".exe" suffix). Thus, we have three users of interest: Bob, Riko, and Suitemates, and these files: the compiler, Riko's written program, and the executable program built by the compiler. We need to implement the policy in Table 3.8.

TABLE 3.8 Policy for protecting Riko's custom program

#	Policy Statement	Risks
1	Everyone shall have execute access to the compiler program.	1, 2
2	Riko shall have full access to the program's source code.	2, 4, 5
3	Riko shall have full access to the program's executable file.	2, 5
4	Bob shall have read and execute access to the program's executable file.	1, 2, 4, 5

Answer the following questions based on the scenario just described.

E12. Draw a diagram showing the user and file access rights that implement the file-sharing policy in Table 3.8. Create a diagram similar to Figure 3.12 that includes the correct users, files, and permissions. Indicate access rights with arrows and labels (RWX). Be sure the access rights allow the work just described and still achieve Least Privilege.

E13. Construct an access matrix that provides the user and file access rights needed to implement the policy in Table 3.8.