**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.

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

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

• **Breach of confidentiality.** This type of violation involves unauthorized reading of data (or theft of information). Typically, a breach of confidentiality is the goal of an intruder. Capturing secret data from a system or a data stream, such as credit-card information or identity information for identity theft, can result directly in money for the intruder.

• **Breach of integrity.** This violation involves unauthorized modification of data. Such attacks can, for example, result in passing of liability to an innocent party or modification of the source code of an important commercial application.

**• Breach of availability.** This violation involves unauthorized destruction of data. Some crackers would rather wreak havoc and gain status or bragging rights than gain financially. Website defacement is a common example of this type of security breach.

• **Theft of service.** This violation involves unauthorized use of resources. For example, an intruder (or intrusion program) may install a daemon on a system that acts as a file server.

• **Denial of service.** This violation involves preventing legitimate use of the system. Denial-of-service (DOS) attacks are sometimes accidental. The original Internet worm turned into a DOS attack when a bug failed to delay its rapid spread.

Attackers use several standard methods in their attempts to breach security. The most common is masquerading, in which one participant in a communication pretends to be someone else (another host or another person). By masquerading, attackers breach authentication, the correctness of identification; they can then gain access that they would not normally be allowed or escalate their privileges—obtain privileges to which they would not normally be entitled. Another common attack is to replay a captured exchange of data. A replay attack consists of the malicious or fraudulent repeat of a valid data transmission. Sometimes the replay comprises the entire attack— for example, in a repeat of a request to transfer money. But frequently it is done along with message modification, again to escalate privileges. Consider the damage that could be done if a request for authentication had a legitimate user’s information replaced with an unauthorized user’s. Yet another kind of attack is the man-in-the-middle attack, in which an attacker sits in the data flow of a communication, masquerading as the sender to the receiver, and vice versa. In a network communication, a man-in-the-middle attack may be preceded by a session hijacking, in which an active communication session is intercepted. Several attack methods are depicted in Figure 1.93.

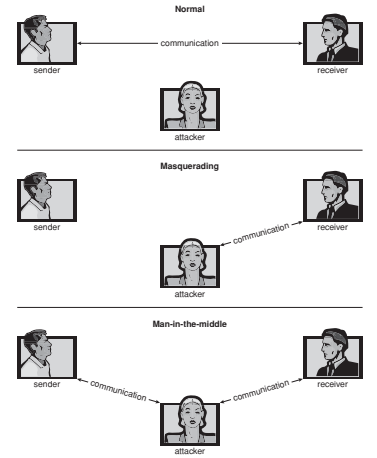


Figure 1.93 Standard security attacks.

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-ofservice attack, it is preferable to prevent the attack but sufficient to detect the attack so that countermeasures can be taken.

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

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

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

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

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

In the remainder of this chapter, we address security at the network and operating-system levels. Security at the physical and human levels, although important, is for the most part beyond the scope of this text. Security within the operating system and between operating systems is implemented in several ways, ranging from passwords for authentication through guarding against viruses to detecting intrusions. We start with an exploration of security threats.

**Program Threats**

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

**Trojan Horse**

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

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

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

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

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

**Trap Door**

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

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

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

**Code Injection**

Code injection is the exploitation of a computer bug that is caused by processing invalid data. Injection is used by an attacker to introduce code into a vulnerable computer program and change the course of execution. The result of successful code injection can be disastrous, for example by allowing computer worms to propagate.

Code injection vulnerabilities occur when an application sends untrusted data to an interpreter. Injection flaws are most often found in SQL, LDAP, XPath, or NoSQL queries; OS commands; XML parsers, SMTP headers, program arguments, etc. Injection flaws tend to be easier to discover when examining source code than via testing. Scanners and fuzzers can help find injection flaws. Injection can result in data loss or corruption, lack of accountability, or denial of access. Injection can sometimes lead to complete host takeover.

Code injection techniques are popular in system hacking or cracking to gain information, privilege escalation or unauthorized access to a system. Code injection can be used malevolently for many purposes.

**Stack and Buffer Overflow**

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

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

1. Overflow an input field, command-line argument, or input buffer—for example, on a network daemon—until it writes into the stack.

2. Overwrite the current return address on the stack with the address of the exploit code loaded in step 3.

3. Write a simple set of code for the next space in the stack that includes the commands that the attacker wishes to execute—for instance, spawn a shell.

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

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

Let’s look at a buffer-overflow exploit in more detail. Consider the simple C program shown in Figure 1.94. This program creates a character array of size BUFFER SIZE and copies the contents of the parameter provided on the command line—argv[1]. As long as the size of this parameter is less than BUFFER SIZE (we need one byte to store the null terminator), this program works properly. But consider what happens if the parameter provided on the command line is longer than BUFFER SIZE. In this scenario, the strcpy() function will begin copying from argv[1] until it encounters a null terminator (\0) or until the program crashes. Thus, this program suffers from a potential buffer-overflow problem in which copied data overflow the buffer array.

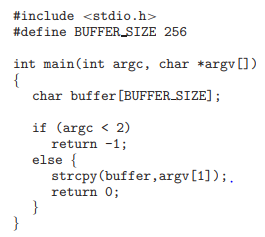


Figure 1.94 C program with buffer-overflow condition.

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

Furthermore, lack of bounds checking is not the only possible cause of the behavior of the program in Figure 1.94. The program could instead have been carefully designed to compromise the integrity of the system. We now consider the possible security vulnerabilities of a buffer overflow.

When a function is invoked in a typical computer architecture, the variables defined locally to the function (sometimes known as automatic variables), the parameters passed to the function, and the address to which control returns once the function exits are stored in a stack frame. The layout for a typical stack frame is shown in Figure 1.95. Examining the stack frame from top to bottom, we first see the parameters passed to the function, followed by any automatic variables declared in the function. We next see the frame pointer, which is the address of the beginning of the stack frame. Finally, we have the return address, which specifies where to return control once the function exits. The frame pointer must be saved on the stack, as the value of the stack pointer can vary during the function call. The saved frame pointer allows relative access to parameters and automatic variables.

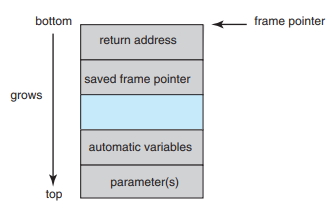


Figure 1.95 The layout for a typical stack frame.

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

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

#include <stdio.h>

int main(int argc, char \*argv[])

{

execvp(‘‘\bin\sh’’,‘‘\bin \sh’’, NULL);

return 0;

}

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

Refer again to the program shown in Figure 1.94. Let’s assume that when the main() function is called in that program, the stack frame appears as shown in Figure 1.96(a). Using a debugger, the programmer then finds the address of buffer[0] in the stack. That address is the location of the code the attacker wants executed. The binary sequence is appended with the necessary amount of NO-OP instructions (for NO-OPeration) to fill the stack frame up to the location of the return address, and the location of buffer[0], the new return address, is added. The attack is complete when the attacker gives this constructed binary sequence as input to the process. The process then copies the binary sequence from argv[1] to position buffer[0] in the stack frame. Now, when control returns from main(), instead of returning to the location specified by the old value of the return address, we return to the modified shell code, which runs with the access rights of the attacked process! Figure 1.96(b) contains the modified shell code.

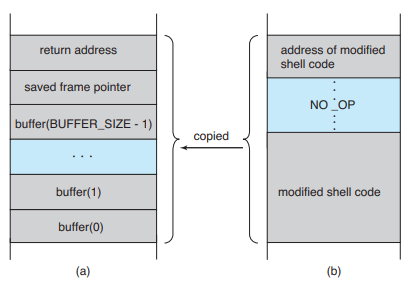


Figure 1.96 Hypothetical stack frame for Figure 15.2, (a) before and (b) after.

There are many ways to exploit potential buffer-overflow problems. In this example, we considered the possibility that the program being attacked— the code shown in Figure 1.94—ran with system-wide permissions. However, the code segment that runs once the value of the return address has been modified might perform any type of malicious act, such as deleting files, opening network ports for further exploitation, and so on.

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

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

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

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

**Viruses**

Another form of program threat is a virus. A virus is a fragment of code embedded in a legitimate program. Viruses are self-replicating and are designed to “infect” other programs. They can wreak havoc in a system by modifying or destroying files and causing system crashes and program malfunctions. As with most penetration attacks, viruses are very specific to architectures, operating systems, and applications. Viruses are a particular problem for users of PCs. UNIX and other multiuser operating systems generally are not susceptible to viruses because the executable programs are protected from writing by the operating system. Even if a virus does infect such a program, its powers usually are limited because other aspects of the system are protected.

Viruses are usually borne via e-mail, with spam the most common vector. They can also spread when users download viral programs from Internet file-sharing services or exchange infected disks. Another common form of virus transmission uses Microsoft Office files, such as Microsoft Word documents. These documents can contain macros (or Visual Basic programs) that programs in the Office suite (Word, PowerPoint, and Excel) will execute automatically. Because these programs run under the user’s own account, the macros can run largely unconstrained (for example, deleting user files at will). Commonly, the virus will also e-mail itself to others in the user’s contact list. Here is a code sample that shows how simple it is to write a Visual Basic macro that a virus could use to format the hard drive of a Windows computer as soon as the file containing the macro was opened:

Sub AutoOpen()

Dim oFS

Set oFS = CreateObject(’’Scripting.FileSystemObject’’)

vs = Shell(’’c: command.com /k format c:’’,vbHide)

End Sub

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

• **File**. A standard file virus infects a system by appending itself to a file. It changes the start of the program so that execution jumps to its code. After it executes, it returns control to the program so that its execution is not noticed. File viruses are sometimes known as parasitic viruses, as they leave no full files behind and leave the host program still functional.

• **Boot**. A boot virus infects the boot sector of the system, executing every time the system is booted and before the operating system is loaded. It watches for other bootable media and infects them. These viruses are also known as memory viruses, because they do not appear in the file system. Figure 1.97 shows how a boot virus works.

• **Macro**. Most viruses are written in a low-level language, such as assembly or C. Macro viruses are written in a high-level language, such as Visual Basic. These viruses are triggered when a program capable of executing the macro is run. For example, a macro virus could be contained in a spreadsheet file.

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

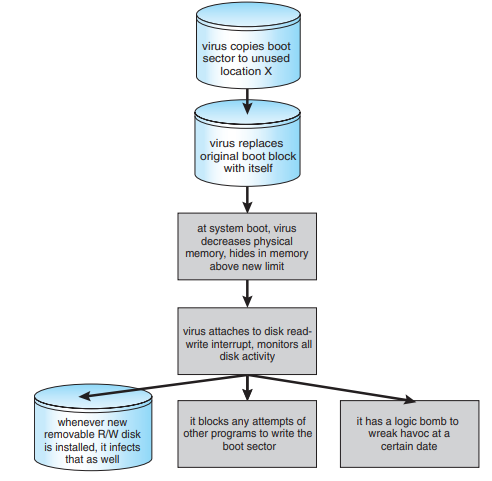


Figure 1.97 A boot-sector computer virus.

• **Polymorphic**. A polymorphic virus changes each time it is installed to avoid detection by antivirus software. The changes do not affect the virus’s functionality but rather change the virus’s signature. A virus signature is a pattern that can be used to identify a virus, typically a series of bytes that make up the virus code.

• **Encrypted**. An encrypted virus includes decryption code along with the encrypted virus, again to avoid detection. The virus first decrypts and then executes.

• **Stealth**. This tricky virus attempts to avoid detection by modifying parts of the system that could be used to detect it. For example, it could modify the read system call so that if the file it has modified is read, the original form of the code is returned rather than the infected code.

• **Tunneling**. This virus attempts to bypass detection by an antivirus scanner by installing itself in the interrupt-handler chain. Similar viruses install themselves in device drivers.

• **Multipartite**. A virus of this type is able to infect multiple parts of a system, including boot sectors, memory, and files. This makes it difficult to detect and contain.

• **Armored**. An armored virus is coded to make it hard for antivirus researchers to unravel and understand. It can also be compressed to avoid detection and disinfection. In addition, virus droppers and other full files that are part of a virus infestation are frequently hidden via file attributes or unviewable file names.

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

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

**System and Network Threats**

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

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

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

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

**Worms**

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

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

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

The worm was made up of two programs, a grappling hook (also called a bootstrap or vector) program and the main program. Named l1.c, the grappling hook consisted of 99 lines of C code compiled and run on each machine it accessed. Once established on the computer system under attack, the grappling hook connected to the machine where it originated and uploaded a copy of the main worm onto the hooked system (Figure 1.98). The main program proceeded to search for other machines to which the newly infected system could connect easily. In these actions, Morris exploited the UNIX networking utility rsh for easy remote task execution. By setting up special files that list host–login name pairs, users can omit entering a password each time they access a remote account on the paired list. The worm searched these special files for site names that would allow remote execution without a password. Where remote shells were established, the worm program was uploaded and began executing anew.

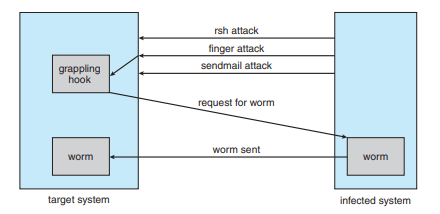


Figure 1.98 The Morris Internet worm.

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

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

finger user-name@hostname

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

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

Once in place, the main worm systematically attempted to discover user passwords. It began by trying simple cases of no password or passwords constructed of account–user-name combinations, then used comparisons with an internal dictionary of 432 favorite password choices, and then went to the final stage of trying each word in the standard UNIX on-line dictionary as a possible password. This elaborate and efficient three-stage password-cracking algorithm enabled the worm to gain access to other user accounts on the infected system. The worm then searched for rsh data files in these newly broken accounts and used them as described previously to gain access to user accounts on remote systems.

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

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

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

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

Sobig.F was launched by being uploaded to a pornography newsgroup via an account created with a stolen credit card. It was disguised as a photo. The virus targeted Microsoft Windows systems and used its own SMTP engine to e-mail itself to all the addresses found on an infected system. It used a variety of subject lines to help avoid detection, including “Thank You!” “Your details, and “Re: Approved.” It also used a random address on the host as the “From:” address, making it difficult to determine from the message which machine was the infected source. Sobig.F included an attachment for the target e-mail reader to click on, again with a variety of names. If this payload was executed, it stored a program called WINPPR32.EXE in the default Windows directory, along with a text file. It also modified the Windows registry.

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

**Port Scanning**

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

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

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

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

**Denial of Service**

As mentioned earlier, denial-of-service attacks are aimed not at gaining information or stealing resources but rather at disrupting legitimate use of a system or facility. Most such attacks involve systems that the attacker has not penetrated. Launching an attack that prevents legitimate use is frequently easier than breaking into a machine or facility.

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

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

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

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