Chapter 4

Malware

Contents

4.1 Insider Attacks

This chapter is devoted to the ways that software systems can be attacked by ***malicious software***, which is also known as ***malware***. Malicious software is software whose existence or execution has negative and unintended consequences. We discuss various kinds of malware, including some case studies, and how systems and networks can be protected from malware.

We begin our coverage of malware with insider attacks. An ***insider attack*** is a security breach that is caused or facilitated by someone who is a part of the very organization that controls or builds the asset that should be protected. In the case of malware, an insider attack refers to a security hole that is created in a software system by one of its programmers. Such an attack is especially dangerous because it is initiated by someone that we should be able to trust. Unfortunately, such betrayals of trust are not uncommon.

Insider attack code can come embedded in a program that is part of a computer's operating system or in a program that is installed later by a user or system administrator. Either way, the embedded malware can initiate privilege escalation, can cause damage as a result of some event, or can itself be a means to install other malware.

4.1.1 Backdoors

A ***backdoor***, which is also sometimes called a ***trapdoor***, is a hidden feature or command in a program that allows a user to perform actions he or she would not normally be allowed to do. When used in a normal way, this program performs completely as expected and advertised. But if the hidden feature is activated, the program does something unexpected, often in violation of security policies, such as performing a privilege escalation. In addition, note that since a backdoor is a feature or command embedded in a program, backdoors are always created by one of the developers or administrators of the software. That is, they are a type of insider attack. (See Figure 4.1.)

Backdoors Inserted for Debugging Purposes

Some backdoors are put into programs for debugging purposes. For example, if a programmer is working on an elaborate biometric authentication system for a computer login program, she may wish to also provide a special command or password that can bypass the biometric system in the

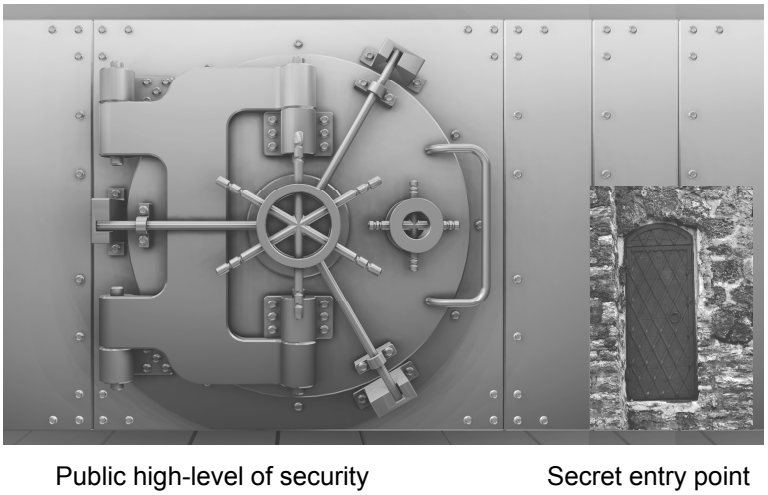

Figure 4.1: Metaphorical illustration of a software backdoor.

event of a failure. Such a backdoor serves a useful purpose during code development and debugging, since it helps prevent situations where a system under development could become unusable because of a programming error. For instance, if a programmer were unable to log in to a system due to a bug in its authentication mechanism, that system might become completely unusable. In these cases, a backdoor might be created that grants access when provided with a special command, such as letmeinBFIKU56, to prevent being locked out of the system when debugging. However, if such a backdoor remains in the program after finishing development, it can become a security risk that may allow an attacker to bypass authentication measures.

A backdoor left in a program even after it is fully debugged might not be intended to serve a malicious purpose, however. For instance, a biometric authentication system might contain a backdoor even after it is debugged, so as to provide a bypass mechanism in the case of an emergency or unanticipated problem. If a user is injured in a way that makes his biometric data invalid, for example if a cut on his hand significantly alters his fingerprint, then it would be useful if he could call the manufacturer of the biometric system to receive a one-time password in order to gain access to his system. Such a one-time password override could technically be considered as a backdoor, but it nevertheless serves a useful purpose. Of course, if a programmer never tells anyone at her company about such an override mechanism or if she inserts it as a way of gaining access to the system after it is deployed, then this backdoor has probably been left in for a malicious purpose.

Deliberate Backdoors

Sometimes programmers deliberately insert backdoors so they can perform malicious actions later that would not otherwise be allowed by the normal usage of their programs. For example, imagine what could happen if a programmer who is designing a digital entry system for a bank vault adds a backdoor that allows access to the vault through the use of a special sequence of keystrokes, known only to him. Such backdoors are clearly inserted for malicious purposes, and they have the potential for dramatic effects. For instance, the classic movie *War Games* features a backdoor at a dramatic high point. The backdoor in this case was a secret password that allowed access to a war-game simulation mode of a computer at the North American Aerospace Defense Command (NORAD).

Another more subtle way of creating a backdoor involves deliberately introducing a vulnerability into a program, such as a buffer overflow (see Section 3.4). Because the programmer knows about the vulnerability, it may be straightforward for him to exploit it and gain elevated privileges. In addition, this situation allows the programmer to simply feign ignorance on being accused of deliberately creating a backdoor—after all, software vulnerabilities are extremely common. Such attacks are sometimes employed against open source projects that allow volunteers to contribute code. An attacker may deliberately introduce an exploitable bug into the code of an open source project, allowing him to gain access to systems on other machines.

Easter Eggs

Software may include hidden features that can be accessed similarly to backdoors, known as ***Easter eggs***. An Easter egg is a harmless undocumented feature that is unlocked with a secret password or unusual set of inputs. For example, unlocking an Easter egg in a program could cause the display of a joke, a picture of the programmer, or a list of credits for the people who worked on that program. Specific examples of programs containing Easter eggs include early versions of the Unix operating system, which displayed funny responses to the command "make love," and the Solitaire game in Windows XP, which allows the user to win simply by simultaneously pressing Shift, Alt, and 2. In addition, movie DVDs sometimes contain Easter eggs that display deleted scenes, outtakes, or other extra content, by pushing unusual keystrokes at certain places on menu screens.

4.1.2 Logic Bombs

A ***logic bomb*** is a program that performs a malicious action as a result of a certain logic condition. (See Figure 4.2.) The classic example of a logic bomb is a programmer coding up the software for the payroll system who puts in code that makes the program crash should it ever process two consecutive payrolls without paying him. Another classic example combines a logic bomb with a backdoor, where a programmer puts in a logic bomb that will crash the program on a certain date. The logic bomb in this case can be disabled via a backdoor, but the programmer will only do so if he is paid for writing the program. This type of logic bomb is therefore a form of extortion.


```
if (trigger-condition = true) {
    unleash bomb;
}
```

Figure 4.2: A logic bomb.

The Y2K Problem

Note that for a piece of software to be a logic bomb, there has to be a malicious intent on the part of the programmer. Simple programming errors don't count. For example, programmers in the 20th century encoded

dates as two digits, xy, to imply $19xy$. When the year 2000 came, this practice caused several problems. Although Y2K didn't have the catastrophic effects that some were expecting, it did cause some problems with some credit-card transactions and other date-dependent calculations. In spite of these negative results, there was, as far as we know, no malice on the part of programmers encoding dates in this way. Instead, these programmers were trying to save some memory space with what they saw as the useless storage of two redundant digits. Because of a lack of malicious intent, the Y2K problem should not be considered a logic bomb although it had a similar effect.

Examples of Logic Bombs

An example of a logic bomb comes in the classic movie *Jurassic Park*, where the programmer, Nedry, installs a piece of code in the software for the park's security system that systematically turns off the locks on certain fences, gates, and doors to allow him to steal some dinosaur embryos.

A real-life logic bomb was reported to have been inserted in 2008 into the network software for Fannie Mae, a large financial enterprise sponsored by the United States government, by a software contractor, Rajendrasinh Makwana. He is said to have set a logic bomb to erase all of Fannie Mae's 4,000 server computers 3 months after he had been terminated. Fortunately, the code for this logic bomb was discovered prior to its activation date, which avoided a digital disaster that would have had major implications in the financial world.

The Omega Engineering Logic Bomb

An example of a logic bomb that was actually triggered and caused damage is one that programmer Tim Lloyd was convicted of using on his former employer, Omega Engineering Corporation. On July 31, 1996, a logic bomb was triggered on the server for Omega Engineering's manufacturing operations, which ultimately cost the company millions of dollars in damages and led to it laying off many of its employees.

When authorities investigated, they discovered that the files on the server were destroyed and that Tim Lloyd had been the administrator for that server. In addition, when they searched for backup tapes for the server, they only found two—at Tim Lloyd's house—and they were both erased.

The Logic Behind the Omega Engineering Time Bomb

In performing a forensic investigation of a true copy of the server's memory, agents for the U.S. Secret Service found a program containing the following sequence of six character strings:

7/30/96

- This was the event that triggered the logic bomb—a date that caused the remaining code to be executed only if the current date was later than July 30, 1996.

F:

- This focused subsequent commands to be run on the volume F, which contained the server's critical files.

F:\LOGIN\LOGIN 12345

- This is a login for a fictitious user, 12345, that had supervisory and destroy permissions, but (surprisingly) had no password. So all subsequent commands would run using the supervisory permissions of user 12345.

CD \PUBLIC

- This is a DOS command to change the current directory to the folder PUBLIC, which stored common programs and other public files on Omega Engineering's server.

FIX.EXE /Y F:*.*

- FIX.EXE was an exact copy of the DOS program DELTREE, which can delete an entire folder (and recursively its subfolders), except that FIX.EXE prints on the screen the words "fixing ..." instead of "deleting ..." for each file that is deleted. The /Y option confirms that each file should indeed be deleted, and the argument F:*.* identifies all the files on volume F as the ones to be deleted.

PURGE F:\/ALL

- Deleted files can often be easily recovered by a simple analysis of the disk. This command eliminates the information that would make such reconstruction easy, thereby making recovery of the deleted files difficult.

Thus, this program was a time bomb, which was designed to delete all the important files on Omega Engineering's server after July 30, 1996. Based in part on this evidence, Tim Lloyd was found guilty of computer sabotage.

4.1.3 Defenses Against Insider Attacks

Protecting a system against backdoors and logic bombs is not easy, since each of these types of malware is created by a trusted programmer (who clearly is not ***trustworthy***). But defense against these types of malware is not impossible. Possible defenses include the following:

- Avoid single points of failure. Let no one person be the only one to create backups or manage critical systems.
- Use code walk-throughs. Have each programmer present her source code to another programmer, line by line, so that he can help her identify any missing conditions or undetected logic errors. Assuming that there is no " sleight of hand," where she would present one set of source code during the code walk-through and install a different set later, she should be unable to discuss the code that defines a backdoor or logic bomb without her partner noticing.
- Use archiving and reporting tools. Several other software engineering tools, such as automatic documentation generators and software archiving tools, have a benefit of uncovering or documenting insider attacks, in addition to their primary goals of producing quality software. Software engineering tools often create visual artifacts or archival digests, which are often reviewed by managers, not just the programmers, so using these tools makes it harder for an insider who is a malware author to have her malicious code go undetected. Likewise, when program code is archived, it becomes harder for a team member to avoid the existence of malware source code to go undiscovered after an attack.
- Limit authority and permissions. Use a ***least privilege*** principle, which states that each program or user in a system should be given the least privilege required for them to do their job effectively. (See Section 1.1.4.)
- Physically secure critical systems. Important systems should be kept in locked rooms, with redundant HVAC and power systems, and protected against flood and fire.
- Monitor employee behavior. Be especially on the lookout for system administrators and programmers that have become disgruntled.
- Control software installations. Limit new software installations to programs that have been vetted and come from reliable sources.

4.2 Computer Viruses

A ***computer virus***, or simply ***virus***, is computer code that can replicate itself by modifying other files or programs to insert code that is capable of further ***replication***. This self-replication property is what distinguishes computer viruses from other kinds of malware, such as logic bombs. Another distinguishing property of a virus is that replication requires some type of ***user assistance***, such as clicking on an email attachment or sharing a USB drive. Often, a computer virus will perform some malicious task as well, such as deleting important files or stealing passwords.

Computer viruses share a number of properties with biological viruses. When released, biological viruses use their environment to spread to uninfected cells. A virus can often lie dormant for a period of time, waiting until it encounters the right kind of uninfected cell. When a virus encounters such a cell, it attacks that cell's defenses at the margins. If it is able to penetrate the cell, the virus uses the cell's own reproductive processes to make copies of the virus instead, which eventually are released from the cell in great numbers, so as to repeat the process. (See Figure 4.3.) In this way, computer viruses mimic biological viruses, and we even use the biological term ***vectors*** to refer to vulnerabilities that malware, such as computer viruses, exploit to perform their attacks.

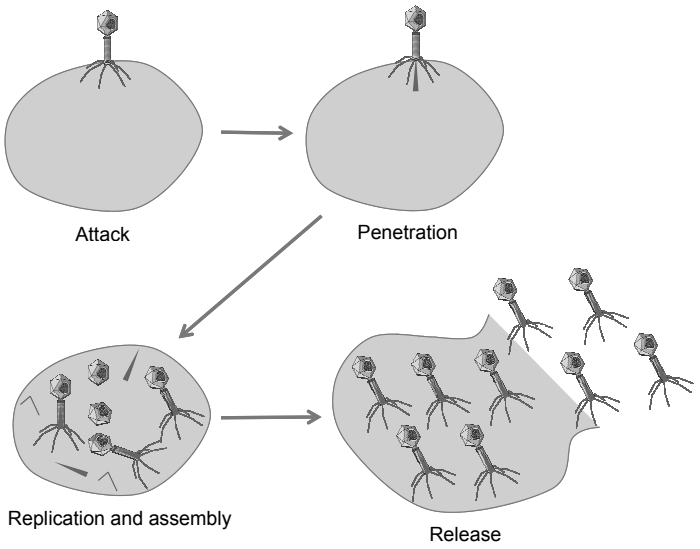

Figure 4.3: Four stages of a biological virus.

4.2.1 Virus Classification

Computer viruses follow four phases of execution:

1. ***Dormant phase***. During this phase, the virus just exists—the virus is laying low and avoiding detection.

2. ***Propagation phase***. During this phase, the virus is replicating itself, infecting new files on new systems.

3. ***Triggering phase***. In this phase, some logical condition causes the virus to move from a dormant or propagation phase to perform its intended action.

4. ***Action phase***. In this phase, the virus performs the malicious action that it was designed to perform, called ***payload***. This action could include something seemingly innocent, like displaying a silly picture on a computer's screen, or something quite malicious, such as deleting all essential files on the hard drive.

These phases characterize many different types of computer viruses. One way to classify the many varieties of viruses is according to the way they spread or the types of files that they infect.

Types of Viruses

A ***program virus***, also known as a ***file virus***, infects a program by modifying the file containing its object code. Once the infection occurs, a program virus is sure to be run each time the infected program executes. If the infected program is run often, as with a common operating system program or a popular video game, then the virus is more likely to be able to be maintained and to replicate. Thus, the most common and popular programs are also the most natural targets for program viruses. Figure 4.4 gives schematic examples of infected program files, which contain both the original program code and the virus code.

Several document preparation programs, such as Microsoft Word, support powerful macro systems that allow for automating sequences of commands. When used benevolently, macros provide, e.g., dynamic updating of facts, figures, names, and dates in documents when the underlying information changes. But macro systems often incorporate a rich set of operations, such as file manipulation and launching other applications. Since a macro can behave similarly to an executable program, it can become a target for viruses.

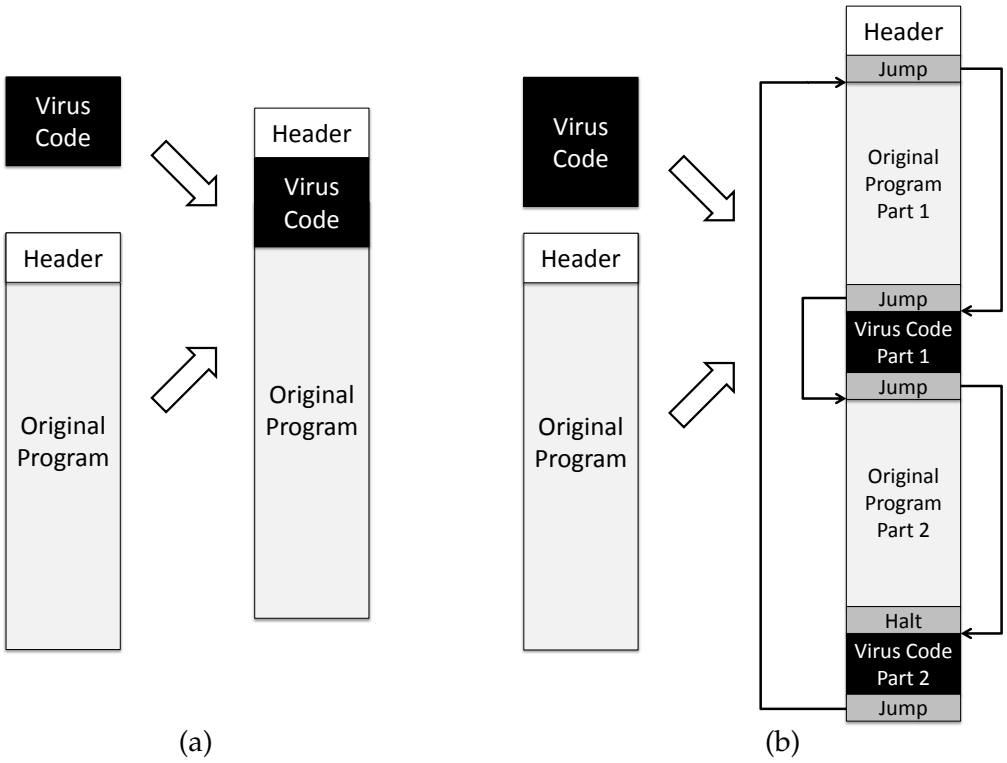

Figure 4.4: How a virus injects itself into a program file: (a) A simple injection at the beginning of a program. (b) A more complex injection that splits the virus code into two parts and injects them at different points in the program. Jump instructions are used to begin execution with the virus code and then pass control to the original program code.

A ***macro virus***, which is also known as a ***document virus***, is launched when a document is opened, at which time the virus then searches for other documents to infect. In addition, a macro virus can insert itself into the standard document template, which makes every newly created document infected. Finally, further propagation occurs when infected documents are emailed to other users.

A ***boot sector virus*** is a special type of program virus that infects the code in the boot sector of a drive, which is run each time the computer is turned on or restarted. This type of virus can be difficult to remove, since the boot program is the first program that a computer runs. Thus, if the boot sector is infected with a virus, then that virus can make sure it has copies of itself carefully placed in other operating system files. As a result, antivirus software routinely monitors the integrity of the boot sector.

Real-World Examples of Computer Viruses

Some real-world examples of computer viruses include the following:

- ***Jerusalem***. This is a virus that originated in the 1980s and infected DOS operating systems files. It was first discovered in Jerusalem, Israel. Once it becomes active on a computer, the Jerusalem virus loads itself into the main memory of the computer and infects other executable files that are run. In addition, it avoids reinfecting the files it has already injected itself into. Its destructive action is that if it is ever executed on a Friday the 13th, then it deletes every program file that is run. The Jerusalem virus has spawned a number of variants, such as Westwood, PQSR, Sunday, Anarkia, and Friday-15th, which cause havoc on other dates and have other slight differences from the original Jerusalem virus.

- ***Melissa***. This was the first recorded virus that spread itself via mass emailing. It is a macro virus that infects Microsoft Word 97 or 2000 documents and Excel 97 or 98 documents. Once an infected document is opened, the Melissa virus would email infected documents to the first 40 or 50 addresses in the victim's address book. It would also infect other Word and Excel documents. When initially launched, the Melissa virus spread so fast that a number of email servers had to be temporarily shut down because they were overloaded with so many emails. It has a number of variants, such as Papa, Syndicate, and Marauder, which differ in the messages and filenames included in the contents of the emails that are sent. In each case, the goal is to entice the recipient to open an enclosed document and further the spread of the virus.

- ***Elk Cloner***. This is a boot sector virus that infected Apple II operating systems in the early 1980s It infected systems by writing itself to the hard drive any time an infected disk was inserted. It was fairly harmless, however, in that its payload simply printed out a poem each 50th time the computer was booted.

- ***Sality***. This is a recent executable file virus. Once executed, it disables antivirus programs and infects other executable files. It obscures its presence in an executable file by modifying its entry point. It also checks if it is running on a computer with an Internet connection, and if so, it may connect to malware web sites and download other malware.

4.2.2 Defenses Against Viruses

Since computer viruses share many similarities with biological viruses, it is appropriate that we defend against them by gaining insight from how our bodies react to harmful intruders. When a virus enters a person's body and gets past her initial defenses, it may spread rapidly and infect many of her cells. As the virus spreads, however, that person's immune system learns to detect unique features of the attacking virus, and it mounts a response that is specifically tuned to attack infected cells.

Virus Signatures

A computer virus unleashed into the wild may get past the generic defenses of several systems and spread rapidly. This spread inevitably attracts the attention of systems managers, who in turn provide sample infected files to antivirus software companies. Experts then study the infected files looking for code fragments that are unique to this particular computer virus. Once they have located such a set of characteristic instructions, they can construct a character string that uniquely identifies this virus.

This character string is known as a ***signature*** for the virus; it amounts to a kind of digital fingerprint. Then, much like our immune system attacking a viral infection, a virus detection program is then loaded up with the signatures of all known viruses. Virus detection software packages have to be frequently updated, so that they always are using the most up-to-date database of virus signatures. Detecting the presence of a virus signature in a file is an instance of a ***pattern-matching*** problem, which consists of finding a search pattern in a text. Several efficient pattern matching algorithms have been devised, which are able to search for multiple patterns concurrently in a single scan of the file.

Virus Detection and Quarantine

Checking files for viruses can be done either by periodic scans of the entire file system or, more effectively, in real time by examining each newly created or modified file and each email attachment received. Real-time virus checking relies on intercepting system calls associated with file operations so that a file is scanned before it is written to disk. Any file that contains a part that matches a virus signature will be set aside into protected storage, known as a ***quarantine***. The programs put into quarantine can then be examined more closely to determine what should be done next. For example, a quarantined program might be deleted, replaced with its original (uninfected) version, or, in some cases, it might be directly modified to remove the virus code fragments (in a process not unlike surgery).

4.2.3 Encrypted Viruses

Because antivirus software systems target virus signatures, computer virus writers often try to hide their code. As illustrated in Figure 4.4, a virus may subdivide itself into multiple pieces and inject them into different locations in a program file. This approach can have some success, because it spreads a virus's signature all over the file, but reassembling the pieces of a virus will immediately reveal its code (and, hence, its signature). One additional technique used by virus writers to make the presence of their virus in a file more stealthy is to encrypt the main body of their program. (See Figure 4.5.)

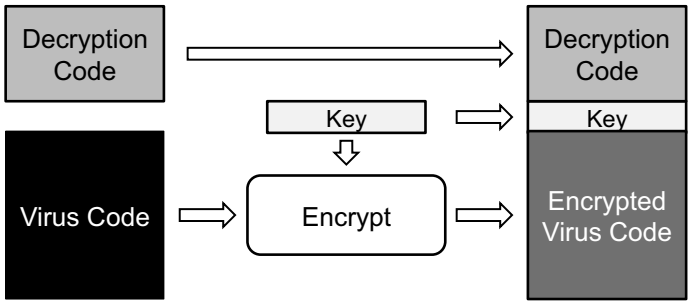

Figure 4.5: How an encrypted virus is structured.

By encrypting the main part of its code, a virus hides many of its distinguishing features, including its replication code and, more importantly, its payload, such as searching for and deleting important files. As illustrated in Figure 4.5, this modification results in the virus code taking on a different structure: the decryption code, the key, and the encrypted virus code. Alternatively, a short encryption key is used (e.g., a 16-bit key) and the decryption code is replaced by code for a brute-force decryption attack. The goal of encryption is to make it harder for an antivirus program to identify the virus. However, note that code for decrypting the main virus body must itself remain unencrypted. Interestingly, this requirement implies that an encrypted virus has a telltale structure, which itself is a kind of virus signature.

Even though this structure doesn't tell a security expert exactly what computations the virus performs, it does suggest to look for pieces of code that perform decryption as a way of locating potential computer viruses. In this way, the virus arms race continues, with the attack of signature-based detection being counterattacked with encrypted viruses, which in turn are themselves counterattacked with virus detection software that looks for encryption code.

4.2.4 Polymorphic and Metamorphic Viruses

Another technique used by viruses to fight back against signature-based detection is mutating as they replicate, thereby creating many different varieties of the same virus. Such viruses are known as ***polymorphic*** or ***metamorphic*** viruses. Although these terms are sometimes used interchangeably, a polymorphic virus achieves its ability of taking on many forms by using encryption, with each copy of the virus being encrypted using a different key. A metamorphic virus, on the other hand, uses noncryptographic obfuscation techniques, such as instruction reordering and the inclusion of useless instructions. Polymorphic and metamorphic viruses are difficult to detect, unfortunately, since they often have few fixed characteristic patterns of bits in their code that can be used to identify them.

Detecting Polymorphic Viruses

One way to detect a polymorphic virus is to focus on the fact that it must use a different encryption key each time the virus encrypts and replicates itself. This choice implies that the body of the computer virus must also include generic code for an encryption algorithm—so that it can encrypt copies of itself with new keys. A polymorphic virus might still have a signature related to its ability to encrypt itself. The encryption code may itself initially be encrypted, so a virus detection algorithm would, in this case, have to identify this decryption code first.

Detecting Metamorphic Viruses

Finding a single string that serves as the signature for a metamorphic virus may be impossible. Instead, we can use more complex signature schemes. A ***conjunction signature*** consists of a set of strings that must appear, in any order, in the infected file. A ***sequence signature*** consists of an ordered list of strings that must appear in the given order in the infected file. A ***probabilistic signature*** consists of a threshold value and a set of string-score pairs. A file is considered infected if the sum of the scores of the strings present in the file exceeds the threshold.

Metamorphic viruses also have an alternative detection strategy. If they have large amounts of pointless code, techniques similar to superfluous code detection methods used in optimizing compilers may be employed. In addition, a metamorphic virus must include code that can perform useless code injection, reorderings of independent instructions, and replacements of instructions with alternative equivalent instructions, all of which might be detected via virus signatures.

4.3 Malware Attacks

When malware was first discovered as a real-world risk to computer security, malicious software was distributed primarily via infected floppy disks. USB drives, CD-ROMs, and DVD-ROMs had not been invented yet, and the Internet was restricted to researchers in universities and industrial labs. Friends and coworkers would share files and collaborate using floppy disks and would inadvertently transmit computer viruses to each other. The explosive growth of the Internet gave rise to a whole new crop of malware, however, which didn't need to inject itself in files and didn't need to be transmitted via media sharing in order to spread.

4.3.1 Trojan Horses

Virgil's *Aeneid* tells the legend of the ***Trojan horse***—a large wooden horse given to the city of Troy as a peace offering. Unknown to the Trojans, the horse was full of dozens of Greek warriors, who snuck out of the horse in the dead of night after it had been brought inside the city, and opened the city gates so that their comrades could immediately attack. Given the powerful imagery that this story inspires, this legend serves as an apt metaphor for a type of malicious software. A ***Trojan horse*** (or ***Trojan***) is a malware program that appears to perform some useful task, but which also does something with negative consequences (e.g., launches a keylogger). (See Figure 4.6.) Trojan horses can be installed as part of the payload of other malware but are often installed by a user or administrator, either deliberately or accidentally.

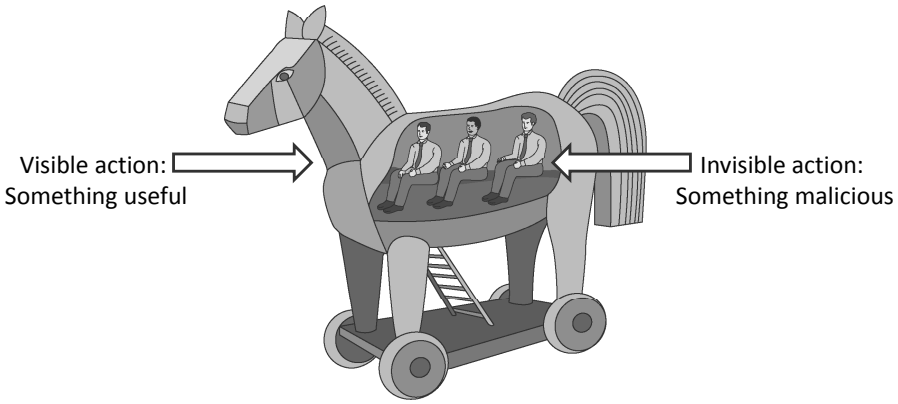

Figure 4.6: A Trojan horse.

Examples of Trojan Horses

A classic example of a Trojan horse is a utility program that performs a useful task better than an existing standard program, such as displaying folders and files of a file system in a beautiful way, while also performing a secret malicious task. By tricking an unsuspecting user into using the Trojan horse, the attacker is able to run his program with all the access rights of the user. If the user can read a proprietary document filled with company secrets, then the Trojan horse can too, and it can even secretly send what it finds to the attacker if the user is connected to the Internet. If the user can send signed emails, then the Trojan horse can too, all in the user's name. And if the user can automatically log in to an online banking system using a stored password, then the Trojan horse can too. The main risk of a Trojan horse is that it allows an attacker to perform a task as if he were another user, possibly even a system administrator.

A real-world example of a Trojan horse was used on one of of the authors while he was in college. A previously trusted friend, whom we will call "Tony," gave the author and several of his friends a program designed to indicate when and where members of this circle of friends were logged in on the campus computer network. In addition to this useful feature, this particular program also sent the friends' passwords to Tony's email account. This particular Trojan horse wasn't discovered until someone noticed a friend logged in at two places at the same time and, when they went to investigate, they found Tony at one of the two locations. Tony was ultimately caught by his own Trojan horse.

Other real-world Trojan horse examples include the following:

- ***The AIDS Trojan***. This was a Trojan horse program that claimed to provide important information about the AIDS disease (acquired immune deficiency syndrome). It was first distributed by mailing floppy disks in 1989. Running the program instead installed a Trojan horse, which would remain quiet until several restarts had occurred, at which time it would encrypt the user's hard drive. Then the Trojan would offer to give the user the password to decrypt the hard drive, but only after she paid a fee. Thus, the AIDS Trojan horse was a type of automated ransom.
- ***False upgrade to Internet Explorer***. This Trojan horse was sent via email as an executable file, which purported to be an upgrade to Microsoft's Internet Explorer. After installation, the program would make several modifications to the user's system. Because of this attack and others like it, most users have learned to avoid opening email attachments that are executable files, no matter what wonderful claims are made about the enclosed program.

- ***False antivirus software***. There have been several instances of this Trojan horse, which advertises itself as antivirus software. When installed, such Trojan horses typically modify the operating system to block real antimalware programs from executing, and then proceed to attempt to steal the user's passwords.

- ***Back Orifice***. First distributed in 1998, this program provided access to a remote computer over an encrypted network connection. Its features included executing commands, transferring files, and logging keystrokes. It was implemented as a service (see Section 3.1.2) for systems running Windows 95 or Windows 98. Thus, once installed, Back Orifice was automatically started whenever the machine was booted. While it had a useful functionality as a remote login and administration tool, Back Orifice was primarily used as a backdoor to steal information. The installation program was typically distributed via email as an executable attachment with an enticing name, such as PAMMY.EXE. When a user opened this attachment, the installation ran quickly and silently. Also, Back Orifice did not show up in Task Manager. Thus, most victims were unaware of the presence of this program.

- ***Mocmex***. In February 2008, it was discovered that several Chinese-made digital photo frames (actual picture frames that render digital images) contained a Trojan horse known as Mocmex. When an infected frame is plugged into a Windows machine, malware is copied from the frame to the computer and begins collecting and transmitting passwords. Mocmex is interesting because it is one of the first widely distributed viruses that takes advantage of an alternative media, in this case digital photo frames.

4.3.2 Computer Worms

A ***computer worm*** is a malware program that spreads copies of itself without the need to inject itself in other programs, and usually without human interaction. Thus, computer worms are technically not computer viruses (since they don't infect other programs), but some people nevertheless confuse the terms, since both spread by self-replication. In most cases, a computer worm will carry a malicious payload, such as deleting files or installing a backdoor.

Worm Propagation

Worms typically spread by exploiting vulnerabilities (e.g., buffer overflow) in applications run by Internet-connected computer systems that have a security hole. A worm then propagates by having each infected computer attempt to infect other target machines by connecting to them over the Internet. If a target machine is also vulnerable to this attack, then it will be infected and will try to infect some other machines in turn. Even if a machine is not vulnerable to a particular worm, it may have to endure repeated attack attempts from infected machines. Also, machines that have already been infected may be targeted for reinfection. (See Figure 4.7.)

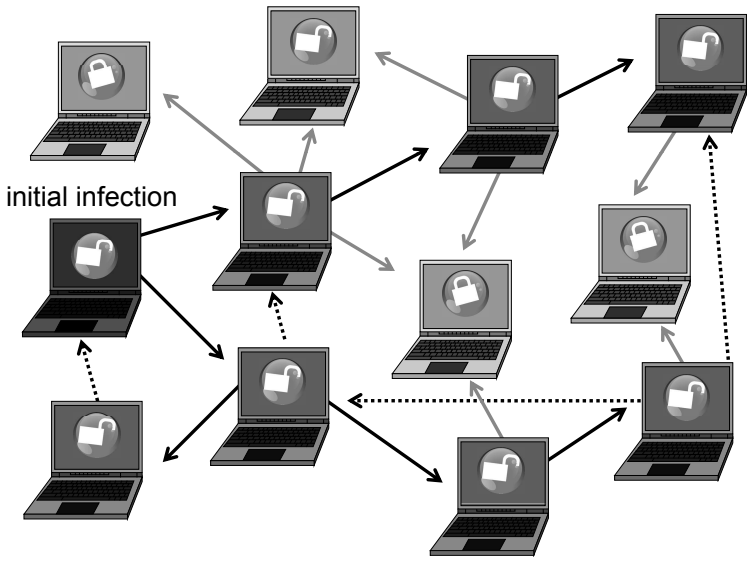

Figure 4.7: How a computer worm propagates through a computer network. Solid lines indicate successful infection attempts, dotted lines indicate reinfection attempts, and gray lines indicate unsuccessful attacks.

Once a system is infected, a worm must take steps to ensure that it persists on the victim machine and survives rebooting. On Windows machines, this is commonly achieved by modifying the ***Windows Registry***, a database used by the operating system that includes entries that tell the operating system to run certain programs and services or load device drivers on startup. One of the most common registry entries for this purpose is called

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run

Associating with this entry the path to the executable file of the worm will result in Windows executing the worm on startup. Thus, malware detection software always checks this entry (and other registry entries specifying programs to run at startup) for suspicious executable names.

The spread of a worm can be modeled using the classic epidemic theory. The model defines the following parameters:

- N: total number of ***vulnerable*** hosts
- $I(t)$: number of ***infected*** hosts at time t
- $S(t)$: number of ***susceptible*** hosts at time t, where we say that a host is susceptible if it is vulnerable but not infected yet
- β: infection rate, which is a constant associated with the speed of propagation of the worm

Starting from a single infected host, the change of $I(t)$ and $S(t)$ over time can be expressed by the following formulas:

$$I(0) = 1 \tag{4.1}$$
$$S(0) = N - 1 \tag{4.2}$$
$$I(t+1) = I(t) + \beta \cdot I(t) \cdot S(t) \tag{4.3}$$
$$S(t+1) = N - I(t+1) \tag{4.4}$$

Formula 4.3 states that the number of new infections, given by $I(t+1) - I(t)$, is proportional to the current number of infected hosts, $I(t)$, and to the number of susceptible hosts, $S(t)$. As shown in Figure 4.8, the propagation of the worm has three phases: slow start, fast spread, and slow finish. This theoretical model has been experimentally confirmed to be a good approximation of the actual propagation of worms in practice.

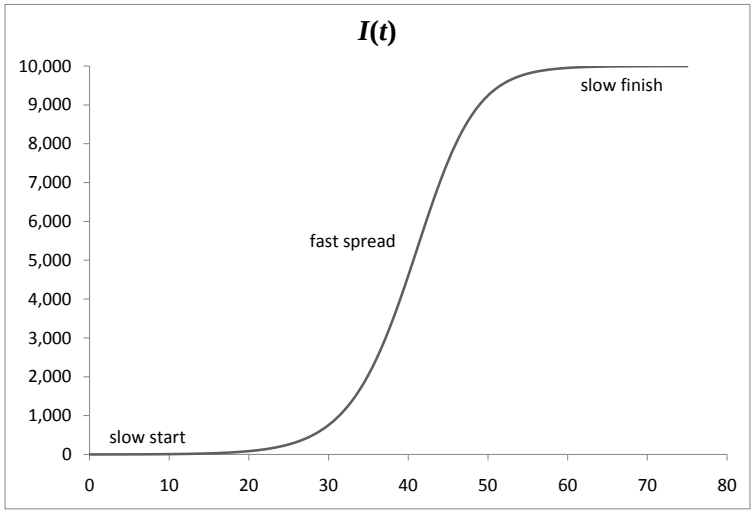

Figure 4.8: The epidemic model of the spread of a worm. The chart shows the number of infected hosts $I(t)$ as a function of time for a total population of $N = 10{,}000$ hosts and infection rate $\beta = 0.000025$, starting from a single host.

The Morris Worm

One of the first computer worms was launched in 1988 by Robert Morris, who was a Cornell University graduate student at the time. This worm didn't carry a malicious payload. Instead, it simply copied itself and spread across the Internet. The main problem with this worm was that it was designed to copy itself onto another vulnerable computer even if that computer was already infected. Interestingly, it would check if the target computer was already infected, but in one out of seven checks it would not trust a "yes" answer and would infect the target computer anyway.

Unfortunately, a probability of 1/7 for reinfection proved to be too high a reinfection rate for Internet-scale propagation, and the Morris worm quickly filled up the set of running processes on the computers it infected. These infected computers therefore suffered what amounted to a denial-of-service attack, since they ended up running many copies of the Morris worm and blocking out other jobs. It is estimated that ten percent of the computers connected to the Internet at that time became infected with the Morris worm, and the damage in lost productivity and the costs associated with cleaning up the Morris worm were estimated to be in the tens of millions of dollars. Robert Morris became the first person convicted under the 1986 Computer Fraud and Abuse Act. He is currently a faculty member at the Massachusetts Institute of Technology.

Some Other Real-World Examples of Computer Worms

Some other real-world examples of computer worms include the following:

- ***ILOVEYOU***. This is an ***email worm*** (a worm sent as an email attachment) first observed in 2000. This particular email worm is a Visual Basic program disguised as a love letter named LOVE-LETTER-FOR-YOU.TXT.vbs. The file extension, "vbs," indicates this is actually a Visual Basic program, which, when executed on a computer running Microsoft Windows, sends itself to everyone in the user's address book and then replaces documents and pictures on the user's hard drive with copies of itself.
- ***Code Red***. This is a computer worm observed on the Internet in 2001, which does not require human intervention to spread. It took advantage of a buffer-overflow vulnerability on computers running Microsoft's IIS web server (which was subsequently fixed) to spread. Its payload was designed to launch denial-of-service attacks on selected web sites from infected computers. Code Red was a fast spreading worm. On July 19, 2001 more than 350,000 vulnerable servers were infected in a few hours.

- ***Blaster***. This is a computer worm that exploited a previous buffer-overflow vulnerability in computers running Microsoft Windows XP and Windows 2000 in 2003. It spread by sending copies of itself to random computers on the Internet, hoping to find other machines with the buffer-overflow vulnerability. Its payload was designed to launch a denial-of-service attack against Microsoft's update web site.
- ***Mydoom***. This is an email worm that was observed in 2004 and seems to have been designed to set up a network of computers from which to send spam emails and launch denial-of-service attacks. It would spread by having users click on an attachment in an email message.
- ***Sasser***. This is a network worm discovered in 2004 that spread by exploiting a buffer-overflow vulnerability in computers running Microsoft Windows XP and Windows 2000. When first launched, it caused Delta Air Lines to cancel several flights, because its critical computers had become overwhelmed by the worm.
- ***Conficker***. This is a computer worm that was first observed in 2008. It targets computers running the Microsoft Windows operating system and is designed to allow the infected computer to be controlled by a third party, e.g., to launch denial-of-service attacks, install spyware, or send out spam emails. It includes a number of sophisticated malware techniques, including an ability to disable Safe Mode, disable AutoUpdate, and kill antimalware programs. It even has a mechanism to update itself from newer copies found on the Internet.

Designing a Worm

Developing a worm is a complex project consisting of the following tasks:

- Identify a vulnerability still unpatched in a popular application or operating system. Buffer overflow vulnerabilities (see Section 3.4) are among the most common ones exploited by worms.
- Write code for:
  - generating the target list of machines to attack, e.g., machines in the same local area network or machines at randomly generated Internet addresses
  - exploiting the vulnerability, e.g., with a stack-smashing attack (see Section 3.4.3)
  - querying/reporting if a host is already infected
  - installing and executing the payload
  - making the worm embedded into the operating system to survive reboots, e.g., installing it as a daemon (Linux) or service (Windows) (see Section 3.1.2)
- Install and launch the worm on a set of initial victims.

Detecting Worms

Note that the propagation process is similar to the traversal of a graph. Here, the nodes are vulnerable hosts and the edges are infection attempts. Referring the terminology used in the classic depth-first-search (DFS) algorithm, a successful infection corresponds to a discovery edge while detecting that a host is already infected corresponds to a back edge. However, while a DFS traversal is performed sequentially in a single thread of execution, the propagation of a worm is a distributed computation that is executed simultaneously by many different infected hosts.

To simplify the attacker's task, several toolkits for developing worms are marketed in the underground economy (see also Section 4.5.4).

The detection of worms can be performed with signature-based file-scanning techniques similar to those described for viruses. In addition, network-level scanning and filtering, which consists of analyzing the content of network packets before they are delivered to a machine, allows to detect and block worms in real time.

4.3.3 Rootkits

A ***rootkit*** is an especially stealthy type of malware. Rootkits typically alter system utilities or the operating system itself to prevent detection. For example, a rootkit that infects the Windows Process Monitor utility, which lists currently running processes, could hide by removing itself from the process list. Likewise, a rootkit might hide files on disk by infecting utilities that allow the user to browse files, such as Windows Explorer. Rootkits are often used to hide the malicious actions of other types of malware, such as Trojan horses and viruses.

Concealment

Rootkits employ several techniques to achieve stealth. Software can either run in user-mode, which includes ordinary program execution, or kernel-mode, which is used for low-level, privileged operating system routines. Accordingly, rootkits may operate in either of these two modes.

Some user-mode rootkits work by altering system utilities or libraries on disk. While this approach may be the simplest, it is easily detected, because checking the integrity of files can be performed offline by using a cryptographic hash function, as detailed below. Other user-mode rootkits insert code into another user-mode process's address space in order to alter its behavior, using techniques such as DLL injection. While these tactics

may be effective, they are easily detected by antirootkit software, which frequently runs at the kernel level.

Kernel-mode rootkits are considered more difficult to detect, because they work at the lowest levels of the operating system. Kernel rootkits in Windows are typically loaded as device drivers, because the device driver system is modular—it allows users to load arbitrary code into the kernel. While this feature is intended to allow developers to easily install drivers for keyboards, audio, or video devices, rootkit developers use device drivers to subvert the security of a system. Even though few Linux rootkits have emerged, kernel-mode Linux rootkits are typically loaded using the Loadable Kernel Module (LKM) system, which functions similarly to Windows device drivers.

Once rootkit code is loaded into the kernel, several techniques may be employed to achieve stealth. One of the most common methods is known as ***function hooking***. Because the rootkit is running with kernel privileges, it can directly modify kernel memory to replace operating system functions with customized versions that steal information or hide the existence of the rootkit. For example, a rootkit might replace a kernel function that enumerates files in a directory with a nearly identical version that is designed to skip over particular files that are part of the rootkit. This way, every program that uses this function will be unable to detect the rootkit. Kernel function hooking is powerful in that rootkit developers only have to alter one function, as opposed to patching every system utility that lists directory contents.

Another kernel-mode rootkit technique involves modifying the internal data structures the kernel uses for bookkeeping purposes. For example, the Windows kernel keeps a list of information on the device drivers that are currently loaded into memory. A rootkit might modify this data structure directly to remove itself from the list and potentially avoid detection. A rootkit that performs this action may be difficult to remove without rebooting the system, because its bookkeeping information would no longer be accessible to the functions that unload device drivers.

Once a system is infected, a rootkit must take steps to ensure that it persists on the victim machine and survives rebooting, including the modification of appropriate entries in the Windows Registry. Since antirootkit software searches the registry for suspicious entries, to avoid detection, some rootkits modify the kernel functions that list registry entries. This is one example of the arms race that takes place between rootkit and antirootkit software, which are constantly engaged in a complex game of hide-and-seek.

Detecting Rootkits

Rootkits are sneaky, but they are not impossible to detect. User-mode rootkits can be detected by checking for modifications to files on disk. On Windows, important code libraries are digitally signed, so that any tampering would invalidate the digital signature and be detected. Another commonly employed technique is to periodically compute a cryptographic hash function for critical system components while the system is offline. This hash can be recomputed while the system is online, and if the hashes do not match, then a rootkit may be altering these files. (See Figure 4.9.) In addition, kernel-mode antirootkit software can detect code injection in system processes.

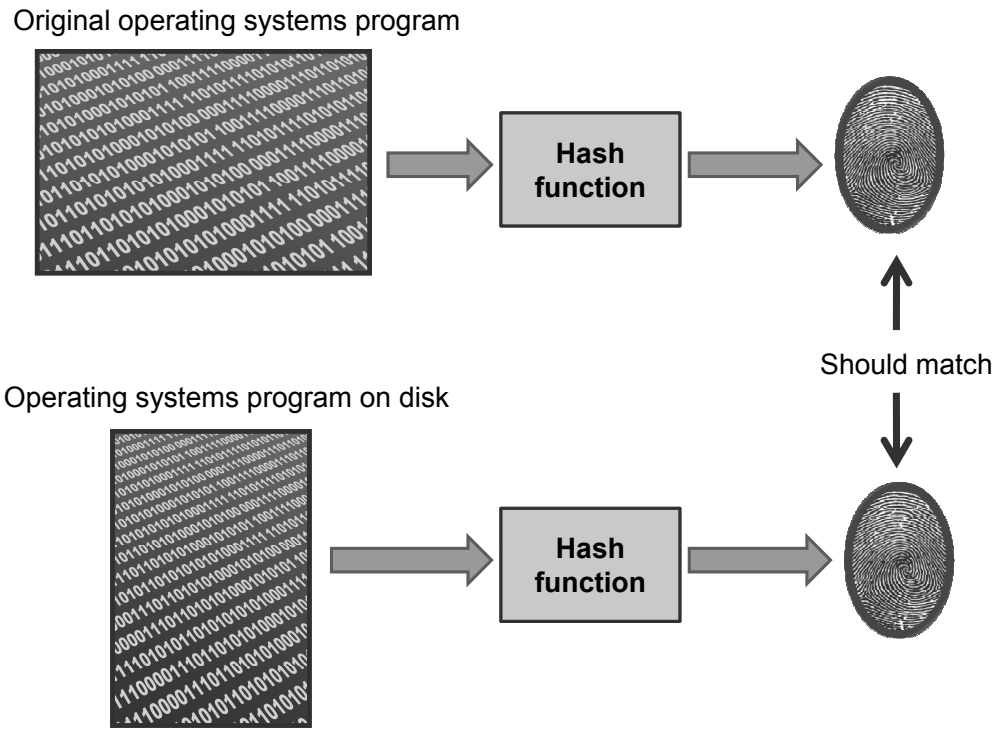

Figure 4.9: How a cryptographic hash function can be used to detect a corrupt operating systems program file on a disk drive.

Kernel-mode rootkits can be more difficult to detect. As discussed, most kernel rootkits do not alter system files on disk, but rather perform their operations on kernel memory. Most antirootkit applications detect kernel rootkits by searching for evidence of techniques such as function hooking. Such rootkit detectors may keep signatures of certain kernel functions that

are likely to be targeted by rootkits, and inspect kernel memory to determine if any modifications have been made to these functions. However, because kernel rootkits operate at the highest level of system privileges, they may preemptively detect antirootkit software and prevent it from achieving its goals. Therefore, sometimes an in-depth offline analysis of an infected system, including inspection of the registry and boot records, is required to defeat rootkits.

A simple and powerful detection technique for rootkits consists of performing two scans of the file system, one using high-level system calls, which are likely infected by the rootkit, and another using low-level disk reading programs that access the content of the disk using primitive block-access methods. If the two scans result in different images of the file system, then it is likely that a rootkit is present. This method is effective for both user-mode and kernel-mode rootkits. Of course, a sophisticated rootkit might anticipate this test and infect both the system calls that list folders and files and also the low-level disk access methods that read a disk one block at a time.

Given the difficulty of guaranteeing the removal of rootkits, users are often advised to reformat their hard drive on suspicion of infection, rather than risking continued compromise by failing to remove all traces of rootkit activity.

An Example Real-World Rootkit

One of the most famous rootkits was included in the copy protection software found on some CDs distributed by Sony BMG in 2005. This rootkit would install itself on PCs running the Microsoft Windows operating system whenever someone put one of the CDs in their optical disk drive (e.g., to rip music off the CD). This automatic installation relied on the default "AutoPlay" option of Windows XP, which executed the commands listed in a designated file on the CD (`autorun.inf`). This rootkit would then infect a number of important files so that the rootkit would not be detected by system utilities. The rootkit's primary intent was to enforce copy protection of the music content on the infected CDs. The rootkit included on these CDs was not intended to be malicious, but since it would hide any process or program with a name that started with a certain string, some malicious code writers soon exploited this fact. Fortunately, it did not have a wide distribution, as it was included in only about 50 CDs. Soon after it was discovered, there were well-advertised ways of removing the copy-protection rootkit.

4.3.4 Zero-Day Attacks

Signature-based methods for detecting viruses and computer worms depend on the ability to find code patterns that uniquely identify malware that is spreading "in the wild." This process takes time, and a new computer virus or worm can continue to spread as workers are trying to find good signatures and push these signatures out to their customers. An ideal goal for antimalware software is that it be able to detect a computer virus or worm even before anyone knows of its existence. Unfortunately, this goal is made difficult by unknown vulnerabilities.

A ***zero-day attack*** is an attack that exploits a vulnerability that was previously unknown, even to the software designers who created the system containing this vulnerability. The term comes from the idea of a fictional timer that starts the moment software designers know about a vulnerability until the day that they publish a patch for it. It is not uncommon for software developers to be aware of nonpublic vulnerabilities, for which they work hard to design a fix. If a malware attack exploits a vulnerability that the developers didn't know about, however, such an attack is said to occur on "day zero" of their awareness of it.

Zero-day attacks pose a unique problem for intrusion detection, since by definition they are not recognizable by simple signature-based schemes. There are two common primary methods for detecting zero-day attacks, both of which are based on ***heuristics***, that is, rules of thumb that perform well in practice. The first heuristic is to continually scan programs for instructions that involve doing something that is potentially malicious, such as deleting files or sending information over the Internet. Such potentially malicious instructions could be tipp-offs that a program has been infected or that a computer worm has been launched. Further inspection is needed, however, since there is a risk of ***false positive*** responses with this approach, because legitimate programs may also perform these types of actions.

Another heuristic for combating zero-day attacks is to run programs in an isolated run-time environment that monitors how they interact with the "outside world." Potentially dangerous actions, like reading and writing to existing files, writing to a system folder, or sending and receiving packets on the Internet, are flagged. A user running such a detection program in the background would be alerted each time an untrusted program performs one of these potentially unsafe actions. Such a run-time environment, which is a type of ***virtual machine***, is sometimes referred to as a ***sandbox***. The challenge in using such a system is that it should not be constantly annoying the user with false-positive actions being performed by legitimate software.

4.3.5 Botnets

The earliest malware was developed for research purposes, but before long malware was being used to conduct destructive pranks and targeted attacks against individuals and organizations. With the widespread adoption of the Internet and the vast amounts of sensitive information stored on home computers, information theft and spam have become lucrative criminal ventures. When this potential for profit became well-known, malware transitioned from the past time of bored teenagers to professionally coded software deployed by criminal organizations.

Because criminal enterprises are interested in these illegal activities on a mass scale, it became desirable to control vast networks of compromised computers, using them as nodes in a spam operation or stealing information from their owners. Such networks are known as ***botnets***, and are centrally controlled by an owner known as a ***bot herder***. Botnets can be truly massive in size—at the time of this writing, the largest botnets are estimated to contain several million compromised machines, and it has been conjectured that up to one quarter of all computers connected to the Internet are part of some botnet.

How Botnets are Created and Controlled

One of the key properties of a botnet is a central ***command-and-control*** mechanism. Once bot software is installed on a compromised computer, via a worm, Trojan horse, or some other malware package, the infected machine, known as a ***zombie***, contacts a central control server to request commands. This way, bot herders can issue commands at will that affect potentially millions of machines, without the need to control each zombie individually.

Early botnets hosted command-and-control stations at static IP addresses that were coded into bot software on each infected machine. This allowed authorities to easily shut down botnets by tracking down control servers and shutting them down. To prevent authorities from shutting down botnets so easily, many botnets now change the address of the command-and-control server daily, dynamically generating and registering the domain name using the current date, for example. To avoid detection, zombie machines often use unexpected channels to receive commands, including services such as Internet Relay Chat (IRC), Twitter, and Instant Messaging.

Botnet Uses

Once a botnet has been assembled, its owner can begin exploiting it to perform illegal activity. Some of the largest botnets harvest credit card numbers, bank account credentials, and other personal information on a terrifying scale.

Other botnets are used to send millions of spam e-mails. Due to the massive total bandwidth under the control of a single person or organization, some botnets have been used to launch distributed denial-of-service attacks against major web sites, even including smaller government infrastructures. As computer use becomes even more pervasive globally, botnets continue to pose a serious threat for illegal activity.

The Zeus Botnet Kit

Zeus is a toolkit for building and deploying a customized Trojan botnet. The attacker can specify the payload to be deployed and the type of information to be capture. Available payloads include not only classic spyware (see Section 4.4.2), but also more sophisticated attacks, such as the following:

- Grab username and password only for specific web sites specified by the attacker.

- Add new form fields to a web page to induce the user to provide additional information. For example, a modified page of a banking site may prompt the user to enter the date of birth and Social Security number for "additional protection."

Zeus has been used extensively to steal credentials for social network sites, banking sites, and shopping sites. In the period from July 2008 to June 2009, Symantec detected more than 150,000 hosts infected with Zeus Trojan horses. In January 2010, NetWitness discovered a Zeus botnet consisting of more than 74,000 zombies in 196 different countries, including many hosts that are part of the networks of government agencies, educational institutions, and large corporations.

4.4 Privacy-Invasive Software

Another category of malware is ***privacy-invasive software***. This class of malicious software targets a user's privacy or information that a user considers sensitive or valuable.

Consent and Intent

Privacy-invasive software can be installed on a user's computer as a result of her visiting a certain web site or as a payload inside a computer virus, network worm, or Trojan horse email attachment. The software invades a user's computer to either operate in the background performing privacy-invasive actions against the user's consent or to immediately gather sensitive or valuable information against the user's wishes.

The intent behind privacy-invasive software is usually commercial. The agent who launched it could, for example, be interested in generating revenue from pop-up advertisements. He could alternatively be interested in stealing information about a user that he can resell to interested parties, or he may have a direct commercial interest in the privacy-invasive action the software is engaging in. In any case, the privacy violations performed by such malware are rarely merely for the sake of curiosity or vandalism.

4.4.1 Adware

Adware is a form of privacy-invasive software that displays advertisements on a user's screen against their consent. Since advertisements are pervasive on the Internet and often are embedded in software packages as a way of reducing the initial purchase price of the software, it is important to stress that malicious adware displays advertisements ***against a user's consent***.

How Adware Works

Typically, an adware program is installed on a user's computer because he visits an infected web page, opens an infected email attachment, installs a shareware or freeware program that has the adware embedded in a Trojan horse, or as the result of being victimized by a computer virus or worm. Once it is installed and running in the background, an adware program will periodically pop up an advertisement on the user's screen. (See Figure 4.10.)

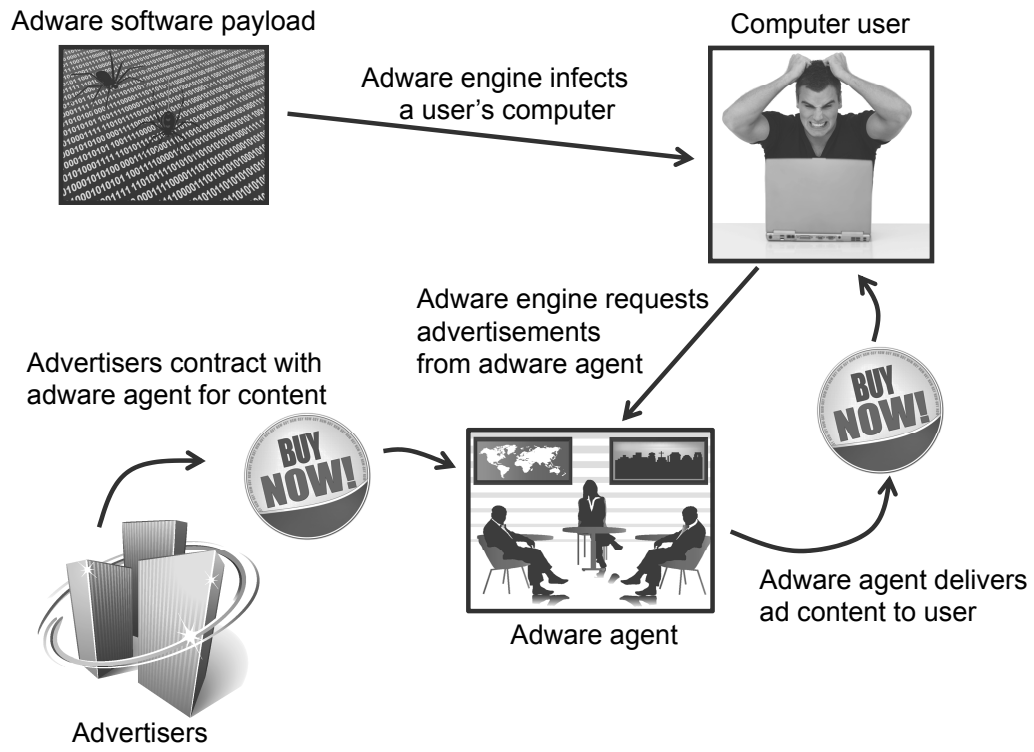

Figure 4.10: How adware works.

Adware Actions

Adware pop-ups are often triggered by the user opening their web browser, so that they might be fooled into thinking the pop-up is a part of their browser's startup page. Alternately, these advertisements might also pop up seemingly at random. Either way, such an advertisement might be created simply to make an impression on the user, similarly to a television commercial or magazine advertisement, or it could have some functionality built in, so that if the user clicks it or tries to close it, then it might display another advertisement or redirect the user to a web page for the product being advertised.

No matter how it is installed or how it operates, adware is an example of privacy-invasive software, because it violates the user's ability to control the content being displayed on his or her computer screen. Moreover, adware often monitors the usage patterns and web page visits of a user, so as to better target the advertisements that are displayed on his or her computer screen. Such instances of adware are also examples of the type of privacy-invasive software we discuss next.

4.4.2 Spyware

Spyware is privacy-invasive software that is installed on a user's computer without his consent and which gathers information about a user, his computer, or his computer usage without his consent. A spyware infection will typically involve the use of one or more programs that are always running in the background, collecting information. Periodically, these programs will contact a data collection agent and upload information it has gathered from the user. (See Figure 4.11.) In order to continue running even after a computer has been rebooted, such an infection will often involve creating modifications in the operating system so that the spyware software is always run as a part of the computer's startup sequence.

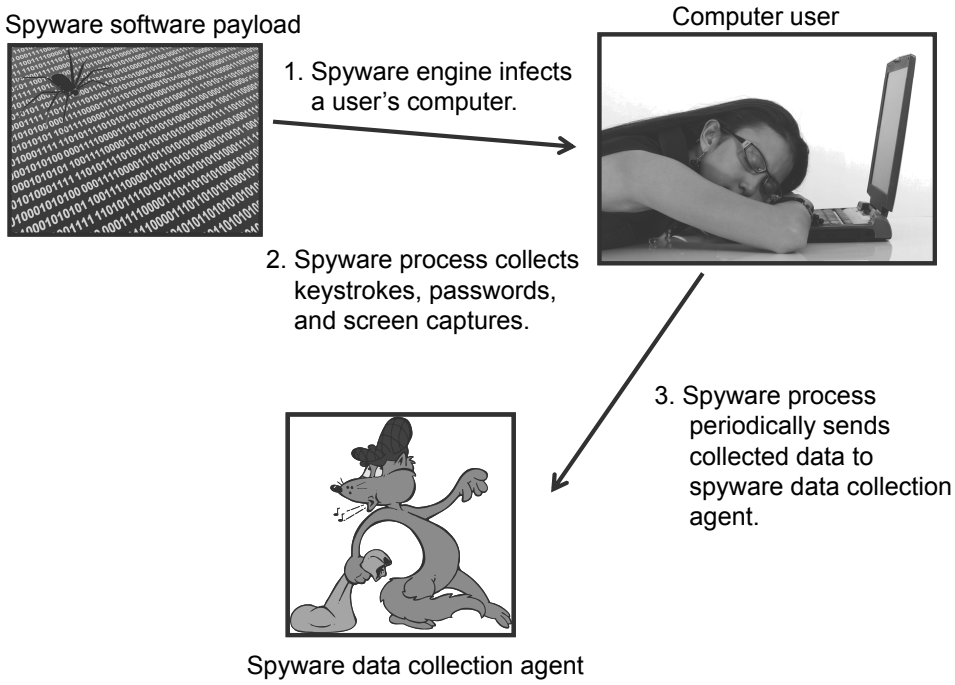

Figure 4.11: How spyware works as a background process.

A user typically doesn't know that his computer is infected with spyware. His only indication may be that his computer might run a little more slowly, but such a performance degradation usually only occurs if a computer has multiple infections. Naturally, a spyware infection will do whatever it can to hide its existence from a computer user, possibly using rootkit hiding tricks (recall Section 4.3.3). A spyware infection might even

go to the length of removing competing adware and spyware, so as to make it harder for a user to notice that unwanted software is running.

Spyware can be categorized by the actions it performs. In particular, we discuss some of the different actions that might be performed by spyware below.

Keylogging

Keystroke logging, or ***keylogging***, is the act of monitoring the actions of a computer's keyboard by recording each key that is pressed (Section 2.4.2). Typically, keyloggers aim to capture sensitive user credentials, such as passwords, login information, and other secret information. Software keyloggers are often installed as ***drivers***, which are part of the operating system and act as an intermediary between hardware and software. Such a keylogger might work by intercepting each keystroke returned by the hardware and recording it to a secret location, before passing it on to the operating system as usual. Writing such a keyboard driver may be difficult, so one potentially simpler approach might take advantage of existing methods of interacting with the existing keyboard driver. For example, many operating systems allow applications to register as keyboard listeners, which are notified each time a key is pressed. Other keyloggers repeatedly ***poll*** the state of the keyboard using existing operating system functions. While this type of keylogger may be easiest to write, it may be easy to detect, because it may require a high amount of CPU usage in order to sustain regular polling of the keyboard state. One final keylogging technique involves using rootkit techniques to hook operating system functions that handle keystrokes and secretly log data.

Screen Capturing

Taking a digital snapshot of a user's screen can reveal a great deal of personal information, and most operating systems provide simple ways of performing such screen captures. Thus, a spyware infection that uses periodic screen captures can greatly compromise the privacy of a user. The challenge to a spyware author wishing to take advantage of screen capturing is that, unlike keystrokes, saving a digital image of a computer screen at a fast enough frequency to grab useful personal information will require a lot of computer time and memory. Thus, spyware that wishes to remain anonymous must perform screen captures at a relatively low rate, or risk detection. Nevertheless, the amount of information gained can be significant, so even with its detection risks, performing screen captures is an alluring option for spyware authors.

Tracking Cookies

Web browser cookies (which are discussed in detail in Section 7.1.4) provide a way for web sites to maintain state between multiple visits of the same user, as a way of "remembering" that user and providing a personalized browsing experience. When a user visits a certain web site for the first time, this site can request that a small file, called a ***cookie***, be placed on her computer to store useful information about her for this site. This feature is beneficial, for instance, if it allows the user to avoid retyping her login and password each time she visits an online movie rental web site or if it allows the site to remember her preferences with a search engine or news-feed web site.

Unfortunately, cookies can also be used for tracking purposes. A group of web sites could conspire to install cookies of a certain name and type, so that they can collectively track when a user visits any of their web sites. Likewise, an advertising company with web banners on many web sites could use tracking cookies to determine which of its customer web sites are visited by a particular user. Thus, cookies used in this way can be viewed as a type of spyware, even though there is no software installed on the user's computer.

Data Harvesting

Another type of spyware avoids the troubles and risks associated with monitoring a user's actions and instead searches through the files on his computer looking for personal or proprietary data. Such programs are ***data harvesters***. Examples include programs that search a user's contact list to collect email addresses that can be used for spamming purposes, possibly even with falsified "From" fields that make it look like the email is coming from another victim. Other examples include programs that look for documents, spreadsheets, PowerPoint presentations, etc., that might contain data of interest to the spyware's master.

As we have seen, spyware authors can gather information using several different techniques. In every case, in order for spyware authors to gain access to information collected on infected machines, there must be a mechanism that allows spyware to communicate with its "master." Because this communication may be detected by antispyware measures, stealth must be employed by spyware to perform the transmission of data as surreptitiously as possible.

Gray Zones

The intent behind a spyware installation is not always malicious, however. For instance, parents might have legitimate reasons for tracking the online behavior of their children and employers might likewise have good reasons for monitoring the ways in which their employees are using company computers at work. These spyware installations are in an ethical gray zone, however, because a parent or employer who owns a computer might knowingly install monitoring software that uses some of the above techniques and is otherwise unknown or disapproved of by the users of that computer. Moreover, it is also possible that the same privacy-invasive software that was originally installed for an ethically justifiable reason could also be used unethically in the future.

Consider, for instance, the privacy concerns surrounding the cameras commonly installed on personal computers and laptops these days. Several computer security companies are marketing software that creates a backdoor on such a computer that allows a third party to capture images from that computer's camera using a special password. The intended use for this software is to take some pictures of the thief in the event that a user's computer is stolen, which is clearly a worthwhile application. But now suppose such computers are laptops owned by a university or high school that is loaning these computers out to disadvantaged students. Antitheft imaging software installed on these computers could allow school administrators to spy on students in their homes through the cameras on their school laptops. Such a use would likely be a violation of laws on illegal wiretapping, and, in fact, a school district in suburban Philadelphia was accused of such a violation in a lawsuit filed in 2010.

Another gray zone concerns companies that provide software or software services in exchange for collecting information about a user. For example, an online email service might perform keyword searches in a user's email messages and display advertisements that are matched with the words used in their messages. For example, an email invitation to a biking trip might be displayed along with an advertisement for bicycles. Similarly, a browser toolbar or desktop searching tool might also collect and communicate information about a user to the company providing these tools. If the user truly makes an informed consent to allow for such monitoring and data collection in exchange for the services or software provided by the company, then this would not be considered spyware. But if this consent is simply assumed or is buried deep in an unreadable user agreement, then one could certainly make the argument that the company has now crossed the line into spyware.

4.5 Countermeasures

The success of malware depends on a number of factors, including:

- ***Diversity***. Malware typically targets a vulnerability in a specific system, such as a particular web browser or operating system. In order to infect as many hosts as possible, the vulnerable software must be widely used. Malware is much more likely to successfully attack software that is used by the most people, as opposed to software that is only one of several equally viable options.

- ***Robustness***. If software contains bugs that make it vulnerable to exploitation, it is naturally more vulnerable to malware attacks. Good software design and coding practices are essential to protect end users from such attacks.

- ***Auto-execution***. Code that resides on USB drives, CD ROMs, and other removable media, as well as web sites, and can run without the direct approval of a user. Such execution paths can be natural vectors for malware attacks.

- ***Privilege***. When programs or users are given more privilege than they need to perform their required tasks, there is a risk that a malware infection could be leveraged into a privilege escalation, which can cause further infection and damage.

4.5.1 Best Practices

There are a number simple precautions that can be used to help protect systems from malware. For instance, best practices that follow immediately from the risks embodied in the factors listed above include the following:

- Employ system diversity as much as possible, including the use of multiple operating systems, document preparation systems, web browsers, and image/video processing systems. Such use of diversity can help limit damage from software-specific vulnerabilities, including those that would be exploited by zero-day attacks.

- Try to limit software installations to systems that come from trusted sources, including large corporations, which have to deal with public-relations nightmares when their software is exploited by malware,

and popular open-source software foundations, which have "many eyes" to catch vulnerabilities (or insider attacks).

- Turn off auto-execution. Allowing for auto-execution is a minor convenience, which is usually not worth the risks it enables. Most auto-execution actions can easily be performed manually anyway.

- Employ a principle of least privilege for sensitive systems and data paths. Limiting people and software to just the privileges they need to perform their work helps to avoid privilege escalation attacks.

Additional Best Practices

In addition, there are number of other best practices to avoid malware infection and minimize the damage caused by malware. These include the following:

- Avoid freeware and shareware unless it comes with verifiable guarantees about the absence of spyware and/or adware from reputable sources. Ideally, software should be digitally signed, so that such guarantees can be enforced and the integrity of the provided software can be checked using cryptographic hash functions.

- Avoid peer-to-peer (P2P) music and video sharing systems, which are often hotbeds for adware, spyware, computer worms, and computer viruses.

- Install a network monitor that blocks the installation of known instances of privacy-invasive software or the downloading of web pages from known malware web sites.

- Install a network firewall, which blocks the transmission of data to unauthorized locations, such as computers or email addresses of spyware sources.

- Use physical tokens, e.g., smartcards (Section 2.3.3), or biometrics (Section 2.3.5) in addition to passwords for authentication, so that even if a keylogger can capture the username and password, more information is required to compromise a user's account.

- Keep all software up-to-date. Computer worms usually don't require direct interaction with humans. Instead, they spread by exploiting vulnerabilities in computers connected to a network. Therefore, the best way to thwart computer worms is to keep all programs updated

with the latest security patches. For example, the Morris worm spread itself by exploiting known vulnerabilities in several Unix systems programs, including the program that forwards emails (sendmail), the program that tells who is logged into the computer (finger), and the program that allows users to login over the network (rsh).

- Avoid weak passwords. This is an often-mentioned best practice, but it should not be ignored. Incidentally, the Morris worm also exploited weak passwords, so encouraging users to pick good passwords is another way to defend against computer worms.

- Use malware-detection and eradication software. The software designers for reputable computer security companies spend a great deal of time and effort on methods for detecting and eliminating malware infections. It would be foolish not to benefit from their expertise.

Detecting Malware from its Behavior

Regarding the last of the recommended best practices listed above, we have already mentioned some of the signature-based methods that are used for detecting computer virus infections, and similar techniques can be used for computer worms. In addition, there are several behavioral properties that can be used to identify and remove malware, such as the following:

- Rootkits necessarily must modify critical operating systems files or alter memory and/or registry entries.

- Adware needs a method for downloading advertisements and displaying them on a user's computer screen.

- Data harvesters need to make calls to operating system routines that read the contents of a large number of files.

- Spyware makes calls to low-level routines to collect events generated by a user, and it must also periodically communicate its collected data back to its master.

Thus, an adware/spyware removal tool can look for these behaviors as indicators that a program is an instance of privacy-invasive software. Also, as widely-distributed examples of privacy-invasive software become public, a removal tool can be updated with patterns and signatures that can identify specific types of infections.

4.5.2 The Impossibility of Detecting All Malware

Ideally, it would be great if we could write a program that could detect every possible instance of malware, including the polymorphic and encrypted viruses. Unfortunately, the existence of such a perfect malware-detection program is impossible.

The argument showing that no such program is possible is based on the principle of "proof by contradiction," which allows us to show that something is impossible because its existence would disprove a fact that we know to be true. Suppose, for the sake of leading to such a contradiction, that there is a program, SuperKiller, that can detect all malware, that is, all programs that act in malicious ways (assuming we could formally define what this means).

Given the existence of SuperKiller, an especially ingenious malware writer could write a malware program, UltraWorm, which runs the SuperKiller program as a subroutine (e.g., to remove rival malware). But this also allows for contradictory behavior, because the code for the UltraWorm would itself be contained in a file that could be given to the SuperKiller program as input.

Consider what could happen if the UltraWorm gave the subroutine SuperKiller the code for UltraWorm (itself) as input. If SuperKiller says UltraWorm is not a computer virus, then UltraWorm could replicate itself, do something malicious, and then terminate. If, on the other hand, SuperKiller says that UltraWorm is malware, then UltraWorm could terminate without doing anything more. For example, the pseudocode for such an UltraWorm could be as follows:

```
UltraWorm():
    if (SuperKiller(UltraWorm) = true) then
        Terminate execution.
    else
        Output UltraWorm.
        Do something malicious.
        Terminate execution.
```

Thus, if SuperKiller says that this UltraWorm is malware, then in reality it is not malware, and if SuperKiller says that UltraWorm is not malware, then in reality it is. This is clearly a contradiction, because we are operating under the assumption that SuperKiller works perfectly and detects all malware, so we must conclude that the SuperKiller program cannot exist. Therefore, there is no foolproof way to detect all malware.

Duality, Undecidability, and Related Concepts

The argument above is admittedly more of a proof sketch than an actual proof that a perfect malware detector is impossible. But this argument can be made rigorous simply by using a more formal definition of malware based on a formal model of computation, such as the model of computation known as a ***Turing machine***. The gist of the formal proof that a perfect malware detector is impossible would still resemble the argument above, however, so we will not give such a formal proof here.

In addition to providing a fun, mind-bending moment, the intellectual exercise above, showing the impossibility of a perfect malware detector, has a number of practical implications and is related to several interesting concepts in computer security and computer science in general.

The first we mention is the related concept of ***duality***, which is a central theme in theoretical computer science. Duality is the property of computer programs that they exist both as a string of characters—the characters that describe a program—and as a functioning computation—the actions that the program performs. In fact, if we think about it, it is precisely because of duality that computer viruses and worms can exist in the first place! That is, duality is what enables a program to perform a computation that involves the replication of the description of that program, so that it can spread. In addition, duality is also the reason that the fictional program Ultraworm could perform contradictory actions should a program like SuperKiller exist.

Another concept related to the discussion above is the fact that testing a program to see if it is malware is not the only computation that it is impossible. There are, in fact, many questions about programs that are ***undecidable***. The most famous such question is that of testing whether a program will terminate or if it instead goes into an infinite loop. This question, which is known as the ***halting problem***, is also undecidable, and the argument that proves this point is similar to that used above to prove that the SuperKiller program cannot exist. That is, if there is a program, HaltTester, which can always check if a program will terminate or not, then we can create a program, Crasher, which feeds itself to the HaltTester program and goes into an infinite loop if and only if HaltTester says that Crasher terminates. So Crasher would have an infinite loop if HaltTester says it doesn't, and Crasher would terminate if HaltTester says it won't. Thus, a program like HaltTester is impossible. Indeed, for related reasons, it is impossible to test if a program has any nontrivial input-output behavior.

4.5.3 The Malware Detection Arms Race

Ironically, the fact that detecting any nontrivial behavior about programs is impossible actually can also provide a comforting thought to the companies that manufacture and sell malware-detection software. While it proves that they will never ultimately succeed in their quest for the perfect malware detector, it also shows that they will never be put out of business by some competitor who could create such a program.

The impossibility of perfect malware detection provides a business plan for malware-detecting companies—keep improving the malware-detection software to detect most kinds of malware, realizing that you will always be able to make improvements, because the ultimate goal is impossible. So an ultra-perfect virus-detection program is impossible, but the software companies selling malware-detection software can continue to sell ever-improving versions of their software.

Misuse of Malware-Detection Software

Unfortunately, honest computer users are not the only people buying malware-detection software from the companies that manufacture it—computer virus and worm writers are buying this software too! Their use of virus-detection software is for a more nefarious purpose, however.

At a high level, malware designers use of existing malware-detection software mimics the way that UltraWorm used the SuperKiller program, in the discussion given above. These attackers use existing malware-detection software for quality control.

For example, each time such an attacker writes a new computer virus, V, he runs the existing malware-detection software on V. If these malware detectors say that V is a virus, then the attacker will never launch an attack using V, and, instead, he will head back to his design lab to work on a new, improved version of V. If, on the other hand, V is not flagged as a virus by any of the existing malware-detection software packages, then the attacker will then launch an attack using V. In so doing, the virus writer is buying his virus time to spread "in the wild" before people discover that it exists and write new programs to detect and kill it. And so the arms race of computer viruses and virus-detection programs continues.

4.5.4 Economics of Malware

Malware is increasing at an alarming rate (see Figure 4.12). Moreover, according to several accounts, a prime motivation for the production of malware is economic. That is, malware can make money for its creator.

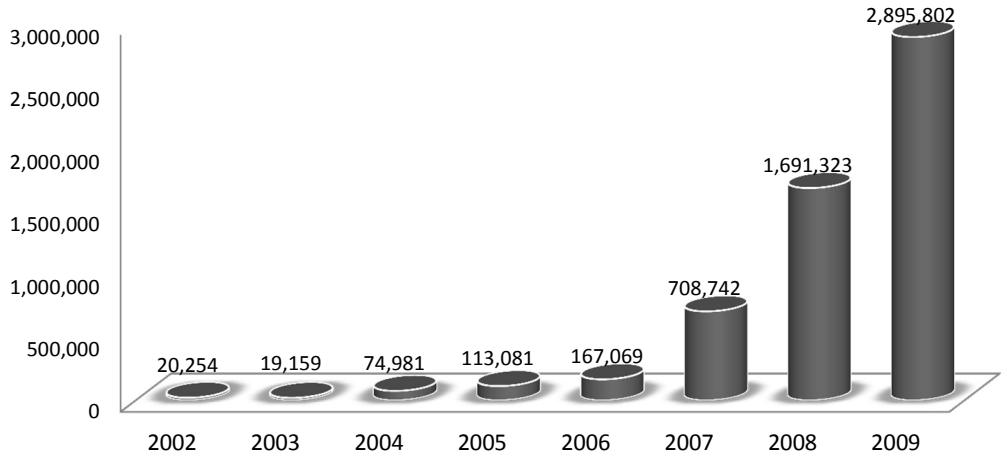

Figure 4.12: New malicious code signatures. Source: Symantec Corp.

According to a recent report from Symantec Corporation, in examining fraudulent chat servers, there is an underground economy for fraudulent tools, services, and products, with some of the following statistics:

- The number one product being advertised on fraudulent chat servers was credit card information. Number two was bank account information. Together these accounted for more than half of the advertised products and services.
- The total value of all advertised goods and services on the observed underground economy servers was over $276 million. This does not include the value of what could be stolen using, say, fraudulent credit card or bank account information. It is just the advertised value of the goods and services themselves.
- Attack tools are being sold directly; hence, they have an economic value to their designers. The average price for a keystroke logger was $23, for instance. The highest priced attack tool was for botnets, which had an average price of $225.

Thus, there are strong economic incentives for malware designers to create malware that can be sold and used by others, to launch spyware attacks that can gather credit card and bank account information, and to exploit operating systems vulnerabilities to create botnets.

4.6 Exercises

For help with exercises, please visit **securitybook.net**.

Reinforcement

R-4.1 In a ***salami-slicing*** attack, a program performs a large number of small, hardly noticeable malicious actions, which add up to a large aggregate malicious action. In a classic example, a programmer for a bank has 1 cent of the monthly interest calculation on each bank customer's account transfered into his account. Thus, if the bank has 1,000,000 customers, then this programmer would get $10,000 each month from this salami slicing attack. What type of malware is such a program?

R-4.2 In the Tim Lloyd logic bomb attack on Omega Engineering, what type of vulnerability was the existence of the user, "12345," an example of?

R-4.3 Viruses that perform no explicit malicious behaviors are called ***bacteria*** or ***rabbits***. Explain how such seemingly benign viruses can still have negative impacts on computer systems.

R-4.4 Explain briefly the differences between polymorphic viruses and metamorphic viruses.

R-4.5 Describe the main differences between a virus, worm, and Trojan horse. How are these types of malware similar?

R-4.6 Bobby says that a computer virus ate his homework, which was saved as a Word document. What kind of virus is the most likely culprit?

R-4.7 Dwight has a computer game, StarGazer, which he plays at work. StarGazer has a secret feature—it pops up an image of a spreadsheet on the screen any time he hits Shift-T on his keyboard, so that it looks like the user is actually working. Dwight uses this feature any time his boss walks by while he is playing StarGazer. What is this "feature" of StarGazer called?

R-4.8 There was an email joke chain letter that called itself the ***Amish virus***. It stated that its author had no computer available in order to write it; hence, it can't run as an executable program or document macro. Instead, it asked the recipient to forward the Amish virus to several friends and then randomly delete some files on his or her hard drive. Is the Amish virus a true email virus? Why or why not?

R-4.9 Explain why a spyware infection that collects mouse moves and clicks without also performing screen captures would not be very useful for a malware author to implement.

R-4.10 Explain why it is often beneficial for an adware author to include spyware in his adware.

R-4.11 Jack encrypts all his email and insists that everyone who sends him email encrypt it as well. What kind of spyware attack is Jack trying to avoid?

R-4.12 Pam's boss, Alan, says that she needs to write her software so that it is protected against the security risks of today and tomorrow. How is this even possible, given that we don't even know what the security risks of tomorrow are?

R-4.13 Eve installed some spyware software on 100 USB flash drives and has designed this software to autoload from these drives along with some nude photos. She then painted the logo of a well-known adult magazine on each one and randomly scattered these flash drives in the parking lots of several of the big defense companies in her town. What type of malware attack is this and what vulnerability is she trying to exploit in order to get her malware code past the network firewalls of these companies?

R-4.14 XYZ Company has just designed a new web browser and they are initiating a major marketing campaign to get this browser to become the exclusive browser used by everyone on the Internet. Why would you expect a reduction in Internet security if this marketing campaign succeeds?

R-4.15 What would be the financial advantage for a malware designer to create lots of different malicious code instances that all exploit the same vulnerability yet have different malware signatures?

R-4.16 Your boss just bought a new malware-detection software program from the ABC Security company for his home computer for $1,000. He says that it was worth the cost, because it says right on the box that this program will "detect all computer viruses, both now and in the future, without any need for updates of virus descriptions." What should you tell your boss?

Creativity

C-4.1 Explain why any computer worm that operates without human intervention is likely to either be self-defeating or inherently detectable.

C-4.2 Describe a malware attack that causes the victim to receive physical advertisements.

C-4.3 You are given the task of detecting the occurrences of a polymorphic virus that conceals itself as follows. The body, C, of the virus code is obfuscated by XORing it with a byte sequence, T, derived from a six-byte secret key, K, that changes from instance to instance of the virus in a random way. The sequence T is derived by merely repeating over and over the given key K. The length of the body of the virus code is a multiple of six—padding is added otherwise. Thus, the obfuscated body is $T \oplus C$, where $T = K||K|| \cdots$ and $||$ denotes string concatenation. The virus inserts itself to the infected program at an unpredictable location.

An infected file contains a ***loader*** that reads the key K, unhides the body C of the virus code by XORing the obfuscated version with the sequence T (derived from K), and finally launches C. The loader code, key K, and the obfuscated body are inserted at random positions of infected programs. At some point of the execution of the infected program, the loader gets called, which unhides the virus and then executes it. Assume that you have obtained the body C of the virus code and a set of programs that are suspected to be infected. You want to detect the occurrences of this virus among the suspected programs without having to actually emulate the execution of the programs. Give an algorithm to do this in polynomial time in the length of the program. Assume that the loader of the virus is a short piece of code that can be commonly found in legitimate programs. Therefore, it *cannot* be used as a signature of our virus. Hence, looking for the loader is not an acceptable solution. Remember, the loader is in binary, and as such, extracting information from it is nontrivial, i.e., wrong.

C-4.4 Suppose there is a new computer virus, H1NQ, which is both polymorphic and metamorphic. Mike has a new malware-detection program, QSniffer, that is 95% accurate at detecting H1NQ. That is, if a computer is infected with H1NQ, then QSniffer will correctly detect this fact 95% of the time, and if a computer is not infected, then QSniffer will correctly detect this fact 95% of the time. It turns out that the H1NQ virus will only infect any given computer with a probability of 1%. Nevertheless, you are nervous and run QSniffer on your computer, and it unfortunately says that your computer is infected with H1NQ. What is the probability that your computer really is infected?

C-4.5 Like a computer virus, a ***quine*** is a computer program that copies itself. But, unlike a virus, a quine outputs a copy of its source code

when it is run, rather than its object code. Give an example of a quine in Java, C, or some other high-level language.

C-4.6 In accepting the ACM Turing Award, Ken Thompson described a devious Trojan horse attack on a Unix system, which most people now refer to as ***Thompson's rigged compiler***. This attack first changes the binary version of the login program to add a backdoor, say, to allow a new user, 12345, that has password, 67890, which is never checked against the password file. Thus, the attacker can always login to this computer using this username and password. Then the attack changes the binary version of the C compiler, so that it first checks if it is compiling the source code for the login program, and, if so, it reinserts the backdoor in the binary version. Thus, a system administrator cannot remove this Trojan horse simply by recompiling the login program. In fact, the attack goes a step further, so that the C compiler also checks if it is compiling the source code of the C compiler itself, and, if so, it inserts the extra code that reinserts the backdoor for when it is compiling the login program. So recompiling the C compiler won't fix this attack either, and if anyone examines the source code for the login program or the C compiler, they won't notice that anything is wrong. Now suppose your Unix system has been compromised in this way (which you confirm by logging in as 12345). How can you fix it, without using any outside resources (like a fresh copy of the operating system)?

C-4.7 Discuss how you would handle the following situations:

(1) You are a system administrator who needs to defend against self-propagating worms. What are three things you can do to make your users safer?

(2) You have a suspected polymorphic virus. What are some steps you can take to correctly identify when you are being infected or propagating it?

(3) You suspect you may have a rootkit installed on your system that is telling the music company whether or not you are violating copyright with an audio CD you recently bought. How might you detect this intrusion without using any outside tools?

(4) If you are a virus writer, name four techniques you would use to make your virus more difficult to detect.

C-4.8 Suppose you want to use an Internet cafe to login to your personal account on a bank web site, but you suspect that the computers in this cafe are infected with software keyloggers. Assuming that you can have both a web browser window and a text editing window

open at the same time, describe a scheme that allows you to type in your userID and password so that a keylogger, used in isolation of any screen captures or mouse event captures, would not be able to discover your userID and password.

C-4.9 Suppose that a metamorphic virus, DoomShift, is 99% useless bytes and 1% useful bytes. Unfortunately, DoomShift has infected the login program on your Unix system and increased its size from 54K bytes to 1,054K bytes; hence, 1,000K bytes of the login program now consists of the DoomShift virus. Barb has a cleanup program, DoomSweep, that is able to prune away the useless bytes of the DoomShift virus, so that in any infected file it will consist of 98% useless bytes and 2% useful bytes. If you apply DoomSweep to the infected login program, what will be its new size?

C-4.10 Each time a malware designer, Pierre, sells a product on a chat server in the underground economy for fraudulent products and services, there is a chance that he will get caught and be fined by law enforcement officials. Suppose the probability that Pierre will get caught because of any one sale of malware is p, and this value is known to both Pierre and the law enforcement officials. What should be the minimum fine for selling a keystroke logger so that it is not worth the effort for a rational malware designer like Pierre to sell it? What about the minimum fine for selling a botnet?

Projects

P-4.1 You want to maliciously infiltrate someone's computer and make it patient zero for your new worm. There are a few ways you could plant your first computer disease vector. You have physical access to your target's computer, but no tools or methods to get access to any passwords. It is a Windows XP machine. Your only tool is EBCD (http://ebcd.pcministry.com/) and you must evade detection, lest you be caught for your dastardly deeds. How will you complete the following steps for your master plan:
(1) Gain administrative access?
(2) Add your worm as a service in Windows?
(3) Cover your tracks so it does not show up in log files or other telltale clues?

P-4.2 Write a software keylogger and test it while you fill out a web form or type in the contents of a document using your favorite document preparation software. If the operating system of your computer supports both keyboard listening events and keyboard

polling, write two versions of your keylogger and compare their respective computational overheads.

P-4.3 Write a simulator that can track how a computer worm propagates in a network of 1 million computers, such that n of these computers are vulnerable to this particular worm. In each step of the simulation, each infected computer randomly picks d other computers and tries to infect them. If a computer is attacked, then it is infected only if it is vulnerable. And if an infected computer is attacked, it will be reinfected according to a random reinfection probability, p. Run a number of experimental simulations for various values of the parameters n, d, and p, including the cases $p = 0$, $p = 1/2$, and $p = 1$, keeping track of how many infections and reinfections occur on each vulnerable computer, as well as the total numbers of each category of vulnerable computer. Try to find parameter values that cause the worm propagation to die out after a few rounds without infecting all the vulnerable computers and also try to find parameter values that cause the worm to overload the vulnerable computers to a point of saturation.

P-4.4 Write a term paper that discusses the business model for adware. Use articles you can find on the Internet, say, from Google Scholar, as source material for this paper. Include in your paper the risks, benefits, and costs for advertisers, adware designers, and the people who run adware servers.

Chapter Notes

Fred Cohen initiated the formal study of computer viruses and showed the undecidability of virus detection [17]. The book by Peter Szor gives a detailed coverage of computer viruses, including advanced detection techniques [99]. A sophisticated model of worm spreading, motivated by the analysis of Code Red propagation, is provided by Zou, Gong, and Towsley [113]. Methods for building and detecting rootkits are provided in the book by Hoglund and Butler [40]. The data for Figure 4.12 comes from the April 2009 *Internet Security Threat Report*, by Symantec Corporation. The discussion on the economics for malware is based on the November 2008 *Symantec Report on the Underground Economy*, by Symantec Corporation.