CHAPTER 5

Worms and Other Malware

This chapter provides a detailed look and some history as to how vulnerable software can impact the entire Internet. Malicious hackers write software that takes advantage of software vulnerabilities to spread worms and infiltrate many machines on the Internet, since much deployed software is vulnerable to attack. If you create software to be fundamentally less vulnerable to attack, then you can minimize the ease with which worms spread. In addition to describing how some worms have worked in detail, we describe other types of malware—such as rootkits, botnets, and keyloggers—and how these have posed threats to the security of the Internet and electronic commerce. The primary purpose of this chapter is to give you a sense of how badly things can go wrong, and give you an idea of what you are up against when you write code.

5.1. What Is a Worm?

A worm is a type of a virus. A *virus* is a computer program that is capable of making copies of itself and inserting those copies into other programs. One way that viruses can do this is through a floppy or USB disk. For instance, if someone inserts a disk into a computer that is infected with a virus, that virus may copy itself into programs that are on the disk. Then, when that disk inserted in other computers, the virus may copy itself and infect the new computers.

A *worm* is a virus that uses a network to copy itself onto other computers. The rate at which a traditional virus can spread is, to an extent, dependent upon how often humans put infected disks into computers. A worm, on the other hand, uses a network such as the Internet to spread. Millions of computers are always connected to the Internet. The rate at which a worm can propagate and spread to other computers is orders of magnitude faster than the rate at which viruses can spread for two reasons: (1) there are a large number of available computers to infect, and (2) the time required to connect to those computers is typically on the order of milliseconds. Given how pervasive networking is, this chapter will focus on worms instead of traditional viruses.

The material in this chapter on worms illustrates how security vulnerabilities can affect the entire Internet. At the same time, worms are simply one type of threat that can result from security vulnerabilities. In addition, while some worms exploit security vulnerabilities in software, not all of them do, and some of them rely on gullible users to spread.

5.2. An Abridged History of Worms

This section describes how worms can affect the operation of the entire Internet. We start with a description of the Morris worm, the first worm ever to be deployed on the Internet, in 1988. Then we cover Code Red, Nimda, Blaster, and SQL Slammer, a series of worms that caused significant damage between 2001 and 2003. Even though the Morris worm surfaced in 1988, a number of the lessons that we learned from it still hold true, and, to an extent, serve as evidence that the Internet community is still working on learning those lessons!

5.2.1. The Morris Worm: What It Did

The Morris worm was named after its creator, Robert Morris. Morris was a graduate student at Cornell University when he wrote the worm. When he first deployed the worm, it was able to infect over 6,000 computers in just a few hours.

The Morris worm used the Internet to propagate from one machine to the other, and it did not need any human assistance to spread. The Morris worm made copies of itself as it moved from one computer to the other. The act of copying itself caused substantial damage on its own. The amount of network traffic that was generated by the worm scanning for other computers to infect was extensive. The effort required by system administrators to even determine if a particular computer was infected, let alone remove it, was also significant.

The Morris worm took advantage of a vulnerability in a UNIX program called fingerd (the finger daemon server) to spread. The *fingerd* program is a server process that answers queries from a client program called finger. The *finger* client program allows users to find out if other users are logged onto a particular system. For example, if you wanted to see if your friend Bob was logged into an Internet host called "shrimp," you could type `finger @shrimp` and look for Bob's login ID in the output. The Morris worm took advantage of the fact that that fingerd was homogenously deployed on all UNIX systems. To propagate from one machine to another, the Morris worm exploited a buffer overflow vulnerability in the fingerd server. (We will cover how buffer overflows work in Chapter 6.)

In addition to the fingerd buffer overflow vulnerability, the Morris worm took advantage of a vulnerability in another piece of software that is deployed on most UNIX servers—the sendmail program. The *sendmail* program is used to route e-mails from one UNIX server to another. It allows mails to be routed to processes in addition to mailbox files. It has a "debug mode" built into it that allows a remote user to execute a command and send a mail to it, instead of just sending the mail to an already running process. The Morris worm took advantage of the debug mode to have the mails that it sends execute "arbitrary" code. In particular, Morris had the servers execute code that copies the worm from one machine to another. The debug mode feature should have been disabled on all of the production UNIX systems that it was installed on, but it was not (see the discussion on hardening in Section 3.6).

A third vulnerability that the Morris worm took advantage of was the use of two additional UNIX commands called `rexec` and `rsh`, both of which allow a user to remotely execute a command on another computer. The `rexec` command required a password. The Morris worm had a list of 432 common passwords hard-coded in it, and attempted to log into other machines using that list of passwords. Once it successfully logged into a machine with a given username and one of the passwords, it would attempt to log into additional machines to which the compromised machine was connected. In some cases, the `rexec` command was used to remotely execute a command on the additional machines with the guessed password.

In other cases, due to the way that the `rsh` command works, the additional machine would allow a login without a username and password because the machine was whitelisted by the user.

The Morris worm illustrates that security is not only a software quality issue, but also a systems deployment issue. Some systems are large and have so many different features that it is difficult to configure the programs correctly so that they have all of their unnecessary features turned off. Entire books have been written on how to harden Linux and Windows deployments. System administrators also should not allow their users to choose weak, easily guessable passwords, or allow their users to arbitrarily whitelist other machines just to make things more convenient.

5.2.2. The Morris Worm: What We Learned

There are three main things that the Morris worm did to propagate from one server to another: (1) it attempted to use a buffer overflow vulnerability in the finger daemon server, (2) it took advantage of the debug mode in sendmail, and (3) it tried to remotely log into hosts using common usernames and passwords.

There are many lessons to be learned from the Morris worm. A first is that "diversity" is good. The Morris worm took advantage of vulnerabilities in UNIX servers. Since most UNIX systems function the same way, the Morris worm was able to rely on certain vulnerabilities existing on all of these hosts. Therefore, from a security standpoint, homogeneity of operating systems makes it easier for an attacker to predictably exploit system vulnerabilities—if the vulnerability exists on one system, it exists on all of them. Heterogeneity in operating systems would have made the Morris worm's job harder. This lesson is still true today. Even with some of the more recent worms that have been successful at infecting many hosts, the large market share that some operating systems companies have can sometimes be a disadvantage from a security standpoint. For instance, if there is a vulnerability somewhere in the Microsoft Windows operating system, and 90 percent of the Internet population runs Microsoft, an attacker can rely on any particular vulnerability being on most of the machines they want to attack.

A second lesson that we learned is that large programs are particularly vulnerable to attack. The sendmail program is very large and has lots of lines of code. With any program that large, it is difficult to be able to go through it line by line and comb it for all possible security vulnerabilities. Big programs are likely to have more bugs and more exploitable vulnerabilities that worms and other attackers can take advantage of. At the same time, we should keep in mind that just because a program is small, it does not necessarily make it any less vulnerable to attack. The fingerd program is small compared to sendmail, yet the Morris worm was able to take advantage of a buffer overflow in it.

A third lesson from the Morris worm is the importance for users to choose good passwords. A good password is one that is hard for an attacker to guess. As was the case with the Morris worm, an attacking person or program can simply use a prepackaged list of common passwords to get access to some user account. Robert Morris and Ken Thompson determined as early as 1979 that if users are left to their own devices, they typically choose easily guessable passwords (Morris and Thompson 1979). If the users in 1988 had all chosen good passwords, then the Morris worm would have had a hard time trying to use its prepackaged list of 432 passwords. While the security community has been working hard to deploy alternative forms of authentication, passwords might be with us for some time. (Chapter 9 discusses password security in more depth.)

5.2.3. The Creation of CERT

Due to the damage and level of disruption that the Morris worm caused in 1988, the US government decided to fund an organization called the Computer Emergency Response Team (CERT). Carnegie Mellon University ran CERT as a research, development, and coordination center for emergency response to attacks. Since 1988, CERT has become a leading center not only on worm activity but also on software vulnerability announcements. CERT also works to raise awareness about our cyber-security even at the time of writing of this book.

5.2.4. The Code Red Worm

In 2001, the Code Red worm surfaced (CERT 2002). It exploited a buffer overflow vulnerability in the Microsoft IIS web server. The web server had an "indexing server" feature turned on by default. Code Red took advantage of the buffer overflow vulnerability in IIS to propagate. Once Code Red infected a particular machine, it started randomly scanning other IP addresses to try to connect to other IIS web servers at those IP addresses. It spread from one web server to another quickly (over 2,000 hosts per minute [Moore and Shannon 2002]).

CODE RED ATTACKS THE WHITE HOUSE

Once the Code Red worm had infected a particular web server, it conducted a DDoS attack against the White House's web site if the date happened to be between the 20th and 27th of the month. The DDoS attack consisted of sending lots of data packets to the White House's web site. This sometimes resulted in using all the bandwidth of the client's Internet connection, and in slowing down or taking the White House web site offline. In addition to the DDoS attack, the worm defaced the home page of the web server that it infected.

Code Red was interesting because it was able to spread at speeds that humans simply could not keep up with. In order for the worm to spread from one web server to another, it only had to construct an IP address, connect to the web server at that IP address, and exploit the buffer overflow vulnerability at the other web server. The entire process took milliseconds. Human response takes minutes or hours. Since the worm was able to spread to thousands of machines within minutes, there was little that anyone could do to react to the attack quickly enough to curtail it.

Another interesting characteristic of the Code Red worm is that it spread rampantly, even though there was virus scanning software running on some of the machines it infected. Virus scanning utilities often scan for infected files—they look for particular bit patterns (sometimes called signatures) in files that may be infected with viruses. However, to prevent being detected, Code Red would just stay resident in the web server's memory. Code Red did not write any files to disk, and, as a result, was able to evade automated detection by some typical virus scanners. At the same time, a user could check if her machine was infected simply by viewing the home page returned by her web server. Anyone visiting an infected web server was alerted, since Code Red defaced the front page of the web server. Unlike most worms, Code Red was much more easily detectable by humans than some virus scanners.

Because Code Red was resident only in memory, it could be eliminated from a particular web server just by rebooting the machine. Yet, even if you rebooted an infected web server, it would typically get reinfected very quickly! So many other infected web servers were continuously scanning for victims that it wasn't long before one of them happened to construct the IP address for your server and reinfect it. As such, firewalls were used to block traffic from being sent to web servers to prevent reinfection.

5.2.5. The Nimda Worm

The Nimda worm was very interesting since it took some of what Code Red did and it made it a lot worse. Nimda not only spread from web server to web server, but it employed multiple propagation vectors. A *propagation vector*, in the context of worms, is a method by which the worm spreads to another machine. The Morris worm, for instance, had three propagation vectors: the fingerd buffer overflow vulnerability, the sendmail debug mode vulnerability, and password-guessing remote login. Code Red, by comparison, only used one propagation vector.

Like Code Red, Nimda spread from web server to web server. In addition, Nimda spread from web servers to web clients by infecting files on the web server. Whenever a web browser connected to that web server and downloaded an infected file, it also became infected. Nimda used the infected client to continue to spread the worm. Nimda sent out e-mails from the infected client to other machines containing the worm's code as a payload. (A *payload* is the data that the worm carries when it travels from one machine to another.) Therefore, Nimda took all of what Code Red did, packaged in a couple of other different propagation vectors, and thereby increased its ability to spread aggressively.

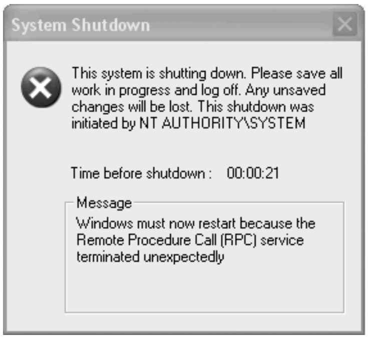
The Code Red and Nimda worms spread so quickly that it caught the attention of many academics and researchers. There are now entire workshops and conferences studying the speed at which worms spread and the potential defenses that we might be able to use as countermeasures (e.g., the Workshop on Rapid Malcode, held in association with the ACM Conference on Computer and Communications Security). Some projects presented at such conferences explore the commonalities between some of the mathematical models that can be used to understand both biological spread of viruses and the technological spread of worms.

5.2.6. The Blaster and SQL Slammer Worms

In 2003, the Blaster and SQL Slammer worms surfaced. Blaster, like Code Red, took advantage of a buffer overflow vulnerability in Microsoft's operating system. Instead of attacking a web server, however, Blaster attacked a Distributed Component Object Model (DCOM) service that was running as part of the operating system.[1] Microsoft deployed a patch for the vulnerability at `http://windowsupdate.microsoft.com` on July 16, 2003. While users could have downloaded the patch and inoculated their systems against an attack that took advantage of such a buffer overflow vulnerability, many unfortunately did not. Patching a system is inconvenient and gets in the way of the "real work" that users are interested in doing. On August 11, 2003, even though the DCOM vulnerability was announced and a patch was deployed, the Blaster worm was still able to take advantage of the vulnerability to launch its attack.

1. DCOM allows programs to make remote procedure calls (RPCs) from one machine to another.

The Blaster worm used an exploit that would cause the user's system to start shutting down. The dialog box shown in Figure 5-1 would pop up on a user's screen once their host was infected.


System Shutdown

This system is shutting down. Please save all work in progress and log off. Any unsaved changes will be lost. This shutdown was initiated by NT AUTHORITY\SYSTEM

Time before shutdown : 00:00:21

Message
Windows must now restart because the Remote Procedure Call (RPC) service terminated unexpectedly

Figure 5-1. *The System Shutdown dialog box caused by the Blaster worm*

The Blaster worm attacked hosts running versions of Windows NT, 2000, and XP. The host did not need to be running a web server. Most users were surprised by the dialog box that popped up as their system shut down. Some users thought that the dialog box might be due to some operating system bug, which they assumed could be corrected by simply by letting their system reboot.

Once the worm caused the system to shut down and reboot, the worm issued a DDoS attack against the Windows Update site (`http://windowsupdate.microsoft.com`). So even when users realized that their PCs may have been infected with a worm, when they tried to go to the Windows Update site to patch their systems, the deluge of DoS traffic sent to the site from their own computers prevented them from doing so.

The Blaster worm coupled some characteristics of the previous worms (exploiting a buffer overflow and randomly scanning for new hosts to which to propagate) with a DDoS attack against a web site that had the patch to fix the problem.

SQL Slammer was another worm that appeared the same year as Blaster. SQL Slammer, like Blaster and some of the other previous worms, took advantage of a buffer overflow vulnerability. However, instead of attacking an operating system service (as in the case of Blaster) or the web server application (as in the case of Code Red), SQL Slammer attacked the Microsoft SQL Server database application.

There are many "mission critical" types of applications that depend upon databases such as Microsoft SQL Server. Once SQL Slammer hit a particular database server, it disabled that server and continued scanning random IP addresses for other SQL Server machines that it could infect. The excessive traffic generated by the SQL Slammer worm as it scanned for other SQL servers to infect caused outages in approximately 13,000 Bank of America ATMs, which prevented users from withdrawing money. In addition, Continental Airlines' systems were affected—some regional flights were canceled and others were delayed (Sieberg and Bush 2003). The SQL Slammer worm was serious enough that the White House was notified.

Another interesting characteristic of SQL Slammer is that it took a single UDP packet of only 376 bytes to exploit the buffer overflow vulnerability to propagate the worm. Since UDP is connectionless, the worm was able to spread extremely quickly. The worm infected at least 75,000 hosts, and 90 percent of them were infected within the first 10 minutes of the worm's release (Moore et al. 2003).

While the SQL Slammer worm caused most of its outages primarily because of the traffic that it generated by scanning other SQL servers, the worm could have been much worse if, for example, it had been designed to delete data, change data, or publish data on the Internet.

IT COULD HAVE BEEN EVEN WORSE

It is important to note that all the worms described in this chapter did not create as much damage as their capability could have permitted. Some of these worms were written by teenage hackers looking for fame. Other more dangerous worms have been developed since then. For example, the Witty worm was engineered to be much more effective than the worms we have described thus far, and was very successful at executing targeted attacks against security products used to protect users' PCs themselves (Shannon and Moore 2004; Schneier 2004).

One positive side effect from these worms is that the security community started to act more aggressively to deal with them. For example, system administrators have become more aware of the threats than they were five or ten years prior. They started deploying preventive and containment measures, and they put more processes in place to deal with the attacks. Researchers started spending a lot more time thinking about how worms spread so that new technologies to contain these attacks could be developed. Incident response teams were formed at some companies that did not already have them, and those companies formed relationships with the authorities to coordinate in the case of a widespread attack. Organizations that did not already have a disaster recovery plan in place created one.

Users have also become a little (but not much) more sensitive and paranoid regarding which web sites they visit, and which e-mail attachments they open.

In addition, programmers are becoming smarter about writing their code such that it is more secure. You are, after all, reading this book! In early 2002, Microsoft put all their development projects on hold, spent a lot of time examining their software, and enrolled their programmers in training courses on how to prevent security vulnerabilities. Hopefully, over time, other companies will follow suit because, if nothing else, the worms that we've discussed have demonstrated that having a vulnerable software infrastructure is not a tenable situation.

5.3. More Malware

Worms that propagate from one machine to another are just one software-based tool that an attacker can use as part of a system infiltration. In this section, we describe other types of malware that an attacker might employ. Some of the worms that we have described take advantage of software vulnerabilities to spread, but other types of malware may not. Some malware, for instance, may rely on social engineering–based attacks to dupe users into downloading and installing it. Since 2003, the types of malware that we list have become more prevalent, and we provide some more up-to-date information and some case studies of them at `www.learnsecurity.com/ntk`.

Here are some other types of malware that you need to be aware of:

Rootkits: A *rootkit* is a set of impostor operating system tools (tools that list the set of active processes, allow users to change passwords, etc.) that are meant to replace the standard version of those tools such that the activities of an attacker that has compromised the system can be hidden. Once a rootkit is successfully installed, the impostor version of the operating system tools becomes the default version. A system administrator may inadvertently use the impostor version of the tools and may be unable to see processes that the attacker is running, files or log entries that result from the attacker's activity, and even network connections to other machines created by the attacker.

Botnets: Once an attacker compromises ("owns") a machine, the attacker can add that machine to a larger network of compromised machines. A *botnet* is a network of software robots that attackers use to control large numbers of machines at once. A botnet of machines can be used, for example, to launch a DDoS attack in which each of the machines assimilated into the botnet is instructed to flood a particular victim with IP packets. If an attacker installs a rootkit on each machine in a botnet, the existence of the botnet could remain quite hidden until the time at which a significant attack is launched.

Spyware: *Spyware* is software that monitors the activity of a system and some or all of its users without their consent. For example, spyware may collect information about what web pages a user visits, what search queries a user enters, and what electronic commerce transactions a user conducts. Spyware may report such activity to an unauthorized party for marketing purposes or other financial gain.

Keyloggers: A *keylogger* is a type of spyware that monitors user keyboard or mouse input and reports some or all such activity to an adversary. Keyloggers are often used to steal usernames, passwords, credit card numbers, bank account numbers, and PINs.

Adware: *Adware* is software that shows advertisements to users potentially (but not necessarily) without their consent. In some cases, adware provides the user with the option of paying for software in exchange for not having to see ads.

Trojan horses: Also known simply as a Trojan, a *Trojan horse* is software that claims to perform one function but performs an additional or different function than advertised once installed. For example, a program that appears to be a game but really deletes a user's hard disk is an example of a Trojan.[2]

Clickbots: A *clickbot* is a software robot that clicks on ads (issues HTTP requests for advertiser web pages) to help an attacker conduct click fraud. Some clickbots can be purchased, while others are malware that spreads like worms and are part of larger botnets. They can receive instructions from a botnet master server as to what ads to click, and how often and when to click them. Some clickbots are really just special cases of a bot in a botnet.

2. The term *Trojan horse* comes from a battle in which the ancient Greeks provided a "gift" to the Trojans—a large wooden horse that was hollow on the inside, and filled with Greek soldiers. The Greeks left the horse at the gates of Troy, and sailed far enough to appear that they had left for good. The Trojans in Troy let the horse into their city gates thinking it was a gift. In the middle of the night, the Greeks hiding inside the horse came out and opened the city gates to allow the Greek soldiers into the city, and they destroyed the city of Troy!

While there are some prevention and detection techniques that can be used to deter worms, rootkits, botnets, and other threats, an implementation vulnerability such as a buffer overflow can open the door to these threats. Buffer overflow vulnerabilities alone are extremely dangerous, and a sound way to deal with them is to prevent them from entering your code altogether, to whatever extent possible. We spend the next chapter on buffer overflow vulnerabilities.