1

Introduction

The shrieking sound of my alarm clock startled me awake that morning. I had been having a strange dream in which computers controlled the world by creating a virtual reality simulation designed to imprison humans. Shaking off my dream, I prepared for another day at work. As usual, I groggily logged into my system to wade through the flood of e-mail that accumulates every night, looking for the real messages requiring "urgent" attention. While sorting through my e-mail, though, I realized my system didn't seem quite right. My computer was sluggish, not its usual snappy self.

I looked for aberrant programs sucking up extra CPU cycles, but found none that had gone awry. It was as though someone or something had snagged hundreds and hundreds of Megahertz from my 2-gigahertz processor. No visible programs were crunching the CPU; it was as though a ghost had invaded my machine. Perhaps I had misconfigured something the night before and had accidentally started a performance death spiral.

I spent the next few hours scouring my system looking for my mistake, but the system looked okay through and through. The config was the same as it had been the morning before. Running a variety of checks, I found no spurious programs, no strange files, and no unusual network traffic.

Then, I started to question the reality of what my machine was telling me about itself. Perhaps I'd been attacked, and the bad guy was tricking me. What if all the checks I was running were actually using the attacker's own code, which lied and told me that everything looked

good? I quickly backed up my system and booted to a CD-ROM I carry around for just such a problem. My handy-dandy CD was full of diagnostic tools. I eagerly scanned my hard drive looking for anomalies. Jackpot! The attacker had laced my system with malicious code designed to hide itself!

The bad guy had run several invisible programs designed to use *my* CPU in a brute-force cracking routine to determine the contents of a hidden encrypted file that the attacker loaded onto my system. His program was not only disguising itself, it was also guessing thousands of keys per second in an attempt to break open the encrypted file so the attacker could read it. I guess it was better for him to offload this processor-intensive activity to my machine and perhaps hundreds of others, rather than to tie up his own precious CPU. To this day, I have no idea of the contents of that mysterious encrypted file he was trying so desperately to open. I do, however, have a far greater sense of the malicious code he had used against my system.

And, that, dear reader, is what this book is all about: malicious code–how attackers install it, how they use it to evade detection, and how you can peer through their nefarious schemes to keep your systems safe. This book is designed to arm you with techniques and tools you need for the prevention, detection, and handling of malicious code attacks against your own computer systems and networks. We'll discuss how you can secure your systems in advance to stop such attacks, how you can detect any maliciousness that seeps through your defenses, and how you can analyze malware specimens that you encounter in the wild.

Defining the Problem

Malicious code planted on your computer gives an attacker remarkable control over your machine. Also known as *malware*, the code can act like an inside agent, carrying out the dastardly plan of an attacker inside your computer. If an attacker can install malicious code on your computers, or trick you into loading a malicious program, your very own computer systems act as the attacker's minions, doing the attacker's bidding. At the same time, your own systems don't follow your commands anymore. They are compromised, acting as evil double agents with real loyalty to the bad guys.

Who needs a human inside collaborator when an attacker can use malicious code to execute instructions on the inside? Human beings infiltrating your organization could get caught, arrested, and interro-

gated. Malicious code, on the other hand, might just get discovered, analyzed, and deleted, all of which are far better for the attacker than having a captured human accomplice in jail starting to spill secrets. Whether your organization is a commercial business, educational institution, government agency, or division of the military, malicious code can do some real damage.

But let's not get too far ahead of ourselves. So what is malware? Many definitions are lurking out there. For this book, let's use this working definition:

> *Malware is a set of instructions that run on your computer and make your system do something that an attacker wants it to do.*

Let's analyze this definition in a little more detail. First, what is a "set of instructions"? Note that the definition doesn't say software or programs, because to many people, these terms imply some sort of binary executable. Although much malicious code is implemented in binary executables, the overall malicious code problem extends far beyond that. Malicious code can be implemented in almost any conceivable computer language, with the limitation being the imagination of the computer attackers, and they tend to be quite an imaginative lot. Attackers have subverted a huge variety of binary executable types, scripting languages, word processing macro languages, and a host of other instruction sets to create malicious code.

Considering our definition again, you might ask what malicious code could make your computer do. Again, the sky's the limit, with very creative computer attackers devising new and ever more devious techniques for their code. Malicious code running on your computer could do any of the following:

- Delete sensitive configuration files from your hard drive, rendering your computer completely inoperable.
- Infect your computer and use it as a jumping-off point to spread to all of your friends' computers, making you the Typhoid Mary of the Internet.
- Monitor your keystrokes and let an attacker see everything you type.
- Gather information about you, your computing habits, the Web sites you visit, the time you stay connected, and so on.
- Send streaming video of your computer screen to an attacker, who can essentially remotely look over your shoulder as you use your computer.

- Grab video from an attached camera or audio from your microphone and send it out to an attacker across the network, turning you into the unwitting star of your own broadcast TV or radio show.
- Execute an attacker's commands on your system, just as if you had run the commands yourself.
- Steal files from your machine, especially sensitive ones containing personal, financial, or other sensitive information.
- Upload files onto your system, such as additional malicious code, stolen data, pirated software, or pornography, turning your system into a veritable cornucopia of illicit files for others to access.
- Bounce off your system as a jumping-off point to attack another machine, laundering the attacker's true source location to throw off law enforcement.
- Frame you for a crime, making all evidence of a caper committed by an attacker appear to point to you and your computer.
- Conceal an attacker's activities on your system, masking the attacker's presence by hiding files, processes, and network usage.

The possibilities are truly endless. This list is only a small sample of what an attacker could do with malicious code. Indeed, malicious code can do anything on your computer that you can, and perhaps even everything that your operating system can. However, the malicious code doesn't have your best interests in mind. It does what the attacker wants it to do.

Why Is Malicious Code So Prevalent?

Malicious code in the hands of a crafty attacker is indeed powerful. It's becoming even more of a problem because many of the very same factors fueling the evolution of the computer industry are making our systems even more vulnerable to malicious code. Specifically, malicious code writers benefit from the trends toward mixing data and executable instructions, increasingly homogenous computing environments, unprecedented connectivity, an ever-larger clueless user base, and an unfriendly world. Let's analyze each of these trends in more detail to see how we are creating an environment much more susceptible to malicious code.

Mixing Data and Executable Instructions: A Scary Combo

One of the primary reasons malicious code has flourished involves the ways computers mix different kinds of information. At the highest level, all information handled by modern computer systems can be broken down into two very general types of content: data and executable instructions. Data is readable, but isn't executed. The computer takes action *on* such content. Executable instructions, on the other hand, tell your machine *to* take some action. This content tells the computer what to do. If only we could keep these two types of information separate, we wouldn't have such a major problem with malicious code. Unfortunately, like a child running with scissors, most computer systems and programs throw caution to the wind and mix data and executable content thoroughly.

To understand the problems that mixing these types of information can cause, consider the following data content:

Here's the story… of a lovely lady
Who was bringing up three very lovely girls.
All of them had hair of gold… like their mother.
The youngest one in curls.

These lines are just plain data, meant to be heard at the start of a classic TV show. Although this is certainly very entertaining fare, we could jazz it up quite a bit if we add executable instructions to it. Suppose we had a human scripting language (which we'll abbreviate, HSL) that would tell people what to do while they were listening to such a song. We'd send the script right inside of the song for the sake of efficiency and flexibility. We might embed executable instructions in the form of a script in the next verse as follows:

Here's the story… of a man named Brady

```
<start HSL script> Go get your checkbook. <stop HSL script>
```

Who was busy with three boys of his own.

```
<start HSL script> Write a big check for the author of this book.
  <stop HSL script>
```

They were four men… living all together.

```
<start HSL script> Put the check in an envelope. <stop HSL script>
```

Yet they were all alone.

```
<start HSL script> Mail the envelope to the author of this book,
  care of the publisher. <stop HSL script>
```

If you were a clueless computer system, you might execute these embedded instructions while singing along with the song. Unfortunately for my checking account, however, you aren't clueless; you are a highly intelligent human being, able to carefully discern the impact of your actions. Therefore, you probably looked at the song and reviewed the embedded instructions, but didn't blindly execute them. Maybe I shouldn't be too hasty here. If, after reading that whole verse of the song, you do have an insatiable desire to send me money, go with your instincts! Don't let me stop you.

By mixing data with executable code, almost any information type on your system could include malicious code waiting for its chance to run and take over your machine. In the olden days, we just had to worry about executable binary programs. Now, with our mixing mania, every type of data is suspect, and every entry point for information could be an opening for malicious code. So, why do software architects and developers design computers that are so willing to mix data and executable instructions? As with so many things in the computer business, developers do it because it's cool, flexible, efficient, and might even help to increase market share. Additionally, some developers overlook the fact that a portion of their user base might be malicious. Let's zoom in on each of these aspects.

Cool: Dynamic, Interactive Content

If content is both viewable and executable, it can be more dynamic, interacting with a user in real time and even adapting to a specific environment. Such attributes in a computing system can be very powerful and profoundly cool. A classic illustration of this argument is the inclusion of various scripting languages embedded in Web pages. Plain, vanilla HTML can be used to create static Web pages. However, by extending HTML to include Javascript, VBScript, and other languages, Web site developers can create far more lively Web pages. With the appropriate scripts, such Web pages can feature animation and alter their behavior based on user input. Whole applications can be developed and seamlessly transmitted across the Web. That's just plain cool.

Flexible: Extendable Functionality

Beyond cool, by including its own custom language in addition to viewable data, a program can be extended by users and other developers in ways that the original program creator never envisioned. These extensions could make the program far more useful than it would otherwise

be. This concept is illustrated in various Microsoft Office® products that include macro languages, such as the Microsoft Word® word processor and the Microsoft Excel® spreadsheet. Developers can write small programs called macros that live inside of a document or spreadsheet. The resulting file can be turned from a mere document into an interactive form, checking user input for accuracy rather than just displaying data. It could even be considered a simple application, intelligently interacting with users and automatically populating various fields based on user input. However, this concept isn't limited to the Microsoft world. Many printers use PostScript, a language for defining page layout for display or printing. With a full language to describe page layout instead of just static images, developers can create far richer content. For example, using just PostScript, a developer can write a page that accesses the local file system to read data while rendering a picture. This functionality is certainly flexible but an attacker could subvert it by using it as a vehicle for malicious code.

Efficient: Flexible Software Building Blocks

By mixing executable instructions and data, developers can create small and simple software building blocks that can be tied together to create larger software projects. That's the idea behind object-oriented programming, a software concept that is infused in most major computer systems today. Instead of the old-fashioned separation of code and data, object-oriented programs create little . . . well, objects. Objects contain data that can be read, as you might expect. However, objects also include various actions that they can take on their own embedded data. Suppose, as an example, we have a virtual hamster object that includes a picture of a cuddly little hamster as data. This hypothetical object might also include some executable code called `Feed_Hamster` that runs and makes the hamster bigger. We could run lots of virtual hamster objects to create an entire community of the little virtual critters. By abusing the `Feed_Hamster` code, however, an attacker might be able to make the virtual hamster explode!

The object-oriented development paradigm is efficient because the objects I create can be used in a variety of different programs by me and other developers. Each sits on the shelf like a little building block, ready to be used and reused in many possibly disparate applications, such as a virtual hamster cage, a virtual traveling hamster circus, or even a simulation of virtual hamsters exhausting all resources in an environmental study.

Market Share: Making the Software World Go 'Round

Given all of the advantages of mixing data and executable instructions just described, the people who create computer systems know that a successful platform that mixes executable code and data can gain market share. Developers who realize the coolness, flexibility, and efficiencies of a platform will start to develop programs in it. With more developers working on your platform, you are more likely to get more customers buying your platform and the tools needed to support it. Viola! The creators of the platform realize increased market share, fame, and untold riches. Microsoft Windows itself is a classic example. The Windows operating system mixes executables and data all over the file system, but it is flexible enough that it has become a de facto standard for software development around the world.

Each of these factors is driving the computer industry ever deeper into combining data and executable instructions. As evidence, two of the hottest buzzwords this decade are Web services. Web services are an environment that allows applications distributed across the Internet to exchange data and executable code for seamless processing on multiple sites at the same time. With Web services, applications shoot bundles of executable instructions and data to each other across the network using eXtensible Markup Language (XML). My Web server might receive some XML from your server and execute the embedded instructions to conduct a search on your behalf. I sure hope you don't flood my systems with malicious code in your XML! Although it has been designed with a thorough security model, the Web services juggernaut promises to more thoroughly mix executable instructions and data at a level we've never seen before, potentially giving malicious code a new and deeper foothold on our systems.

In fact, with the way the computer industry is evolving, the separation of data and executable instructions seems almost passé. However, we face the rather significant problem of malicious code. A nasty person could write a series of instructions designed to accomplish some evil goal unanticipated by the developers of the language and users of the computer. These malicious instructions can be fed directly into some executable component of a target system, or they could be embedded in otherwise non-executable data and fed to the target. In fact, a majority of the malicious code examples covered in this book function just this way.

Malicious Users

Some developers write code assuming that users are kind, gentle souls, going about their day-to-day business with the purest of intentions. Because they expect their software to live in such a benign environment, developers often don't check the input from users to see if it would undermine the system. Of course, in the real world, the vast majority of systems are exposed to at least some malevolent users. An application on the Internet faces attack from the general public, as well as unscrupulous customers of the system. Even internal applications face disgruntled employees who might try to break the system from the inside out.

If a program isn't written with firm defenses in mind, an attacker could manipulate the system by providing executable instructions inside of user input. The attacker could then trick the system into running the executable instructions, thereby taking the machine over. In fact, this is precisely how numerous popular exploit techniques work.

For example, when a software developer doesn't check the size of user input, an attacker could provide oversized input, resulting in a buffer overflow attack. Buffer overflow vulnerabilities are extremely common, with new flaws discovered almost daily. To exploit a buffer overflow, an attacker provides user input that includes executable instructions to run on the victim machine. This malicious, executable input is large enough to overwrite certain data structures on the victim machine that control the flow of execution of code on the box. The attacker also embeds information in the user input that alters this flow of execution on the target system, so that the attacker's own code runs. By taking user input (which should be data) and treating it as executable instructions, the system falls under the attacker's control.

Beyond buffer overflows, consider web applications, such as online banking, electronic government, or other services, that utilize a Structured Query Language (SQL) database to store information. In a SQL injection attack against such applications, a bad guy sends database commands inside of user input. The user might be expected to provide an account number, but an attacker instead provides a line of SQL code that dumps information from the database in an unauthorized fashion. If the application doesn't screen out such a command, the database will execute it, giving the attacker raw access to a Web application's database. Again, because we have mixed executable instructions with user input, we've exposed our systems to attack.

Buffer overflows and SQL injection are but the tip of this exploit iceberg. Attackers have numerous vectors to sneak executable code into our systems along with standard user input. Clearly, developers must be extremely careful in the mixing of data and executable instructions, or else their systems will be highly vulnerable to attack.

Increasingly Homogeneous Computing Environments

Another trend contributing to the increasing problem of malicious code is the fact that we're all running the same types of computers and networks these days. Two decades ago, way back when pterodactyls flew the skies over the Earth, there were a lot of different kinds of computers and networks running around. We had minis, mainframes, and PCs, all with a huge variety of different operating system types and supported network protocols. There were numerous types of processor chips as well, with the Intel, Motorola, MIPS, Alpha, and Sparc lines being but a handful of examples. Malicious code back then could only attack a limited population. One of the single biggest impediments to the propagation of malicious code is a diverse computing base. My Apple II virus would be a fish out of water on your IBM mainframe. Likewise, if my evil worm expects certain support from a specific host operating system, and doesn't find that on your machine, it cannot take over.

Now, however, things have changed. The computer revolution has brought a major consolidation in platform types and networks. It seems that everything runs on Windows or UNIX, and uses TCP/IP to communicate. Processors based on Intel's x86 instruction set seem to dominate the planet. Even those increasingly rare holdout systems that don't rely on these standards (such as a pure MVS mainframe or a VAX box) are still probably accessed through a UNIX or Windows system front end, running an Intel processor or clone, on a TCP/IP network. Even at the application level, we see widespread support of HTML, Java, and PDF files across a number of different application types.

And things are poised to condense even more. Several of the major UNIX vendors, including IBM (maker of the AIX flavor of UNIX), Sun Microsystems (of Solaris UNIX fame), and HP (owner of the HP-UX variety of UNIX) have announced their increasing support of Linux. Although AIX, Solaris, and HP-UX haven't been abandoned, Linux appears to be the wave of the future for UNIX-like environments.

What does this trend mean for malicious code? A homogenous computing environment is extremely fertile soil for malicious code. The evil little program I wrote on my $400 beat-up Linux laptop could

infect your gazillion-dollar mainframe running Linux. Likewise, a nation-state could create some malicious code that would infect a hundred million Windows boxes worldwide. Because our computing ecosystem has less diversity, a single piece of malicious code could have an immense impact.

Unprecedented Connectivity

At the same time we're condensing on a small number of operating systems and protocols, we're greatly increasing our interconnectedness. We used to see islands of computer connectivity. My corporate network wasn't jacked into your government network. The phone system didn't have indirect data connections with university machines. The automatic teller machine (ATM) network was carefully segmented from the Internet.

My, how that has changed! Now, it seems that all computers are connected together, whether we want them to be or not. My laptop is connected to the Internet, which is connected to a pharmaceutical company's DMZ, which connects to their internal network, which connects to their manufacturing plant network, which connects to their manufacturing systems, which make the medicines we all give to our children. Malicious code could jump from system to system, quickly wreaking havoc throughout that supply line.

Two major computer glitches illustrate this concept of unwanted hyperconnectivity. Back in 1999, off the coast of Guam, a United States Navy ship detected the Melissa macro-virus on board [1]. Somehow, due to unprecedented connectivity, the unclassified network of the *USS Blue Ridge* was under attack from Melissa, out in the middle of the water halfway around the world! Additionally, in January 2003, the SQL Slammer worm started ripping through the Internet, sucking up massive amounts of bandwidth. During its voracious spread, it managed to hop into some cash machine networks. By tying up links on the ATM network, more than 13,000 cash machines in North America were out of commission for several hours. The same worm managed to impact police, fire, and emergency 911 services as well. Both of these examples show how easily malicious code can spread to computer systems that aren't obviously connected together.

Ever Larger Clueless User Base

In the last decade, the knowledge base of the average computer user has declined significantly. At the same time, their computers and network connections have grown more powerful and become even juicier targets for an attacker. Today's average computer users don't understand the complexities of their own machines and the risks posed by malicious code. I don't think we in the computer industry should design systems that expect users to understand their systems at a fine-grained level. The average Joe or Jane User wants to treat his or her computer like an appliance, in a manner similar to a refrigerator or a stereo. Who could imagine a refrigerator that can get a virus, or a worm infecting a stereo?

However, our computers and protocols have been built around an assumption that users will understand the concerns and trade-offs associated with various risky behaviors such as downloading code from the Internet and installing it, surfing to Web sites that might hose a system, and not applying patches to system software and applications. For most users, that's a pretty poor assumption. We have made systems that, at best, offer a poorly worded techno-babble warning to Joe and Jane User as they run highly risky software or forget to apply a system patch that they don't understand and typically ignore. Most of the time, there is no warning at all! We shouldn't be surprised when malicious code proliferates in such an environment.

The World Just Isn't a Friendly Place

I don't know if you've noticed it, but the world can be a pretty unfriendly place. Over the past couple years, international events have underscored the fact that we live in a tremendously unstable world. We've had wars and terrorism for millennia, but international "incidents" sure seem to have intensified in recent times.

Although I'd hate to see it, it's conceivable that terrorist organizations could move beyond physical attacks and attempt to undermine the computing infrastructure of a target country. Beyond the terrorist threat, we also face the possibility of a cyberattack associated with military action between countries. In addition to lobbing bullets and bombs, countries could turn to cyberattacks in an attempt to disable their adversaries' military and civilian computer infrastructure. Countries around the world are spending billions of dollars on cyberwarfare capabilities. I don't want to be too much of a pessimist. However, it

seems to me highly likely that malicious code, with its ability to clog networks and even let an attacker take over systems, will be turned into a weapon of war or terror in the future, if it hasn't already.

Types of Malicious Code

On that cheery note, we turn our attention to the multitude of malicious code categories available to attackers today. About a decade ago, when I first started working in computer security, I was overwhelmed at all of the avenues available to an attacker for squeezing executable instructions into a target machine. An attacker could shoot scripts across the Web, overflow buffers with executable commands, send programs in e-mail, overwrite my operating system, tweak my kernel … all of the different possibilities boggled my mind. And the possibilities have only increased in the last 10 years. Each mechanism used by the bad guys for implementation and delivery of malicious code is quite different, and requires specific understanding and defenses.

As an overview to the rest of the book, let's take a look at the different categories of malicious code. Think of me as a zookeeper taking you to look at some ferocious animals. Right now, we'll take a brisk walk past the cages of a variety of these beasties. Later, throughout the rest of this book, we'll get a chance to study each specimen in much more detail. The major categories of malicious code, as well as their defining characteristics and significant examples, are shown in Table 1.1. Note that the defining characteristics are based on the mechanisms used by the malicious code to spread, as well as the impact it has on the target system. Keep in mind that some malware crosses the boundaries between these individual definitions, a theme we'll discuss in more detail in Chapter 9.

Table 1.1
Types of Malicious Code

Type of Malicious Code	Defining Characteristics	Significant Examples	Covered In
Virus	Infects a host file (e.g., executable, word processing document, etc.) Self-replicates. Usually requires human interaction to replicate (by opening a file, reading e-mail, booting a system, or executing an infected program).	Michelangelo, CIH	Chapter 2

Table 1.1
Types of Malicious Code (Continued)

Type of Malicious Code	Defining Characteristics	Significant Examples	Covered In
Worm	Spreads across a network. Self-replicates. Usually does not require human interaction to spread.	Morris Worm, Code Red, SQL Slammer	Chapter 3
Malicious mobile code	Consists of lightweight programs that are downloaded from a remote system and executed locally with minimal or no user intervention. Typically written in Javascript, VBScript, Java, or ActiveX.	Cross Site Scripting	Chapter 4
Backdoor	Bypass normal security controls to give an attacker access.	Netcat and Virtual Network Computing (VNC): Both can be used legitimately as remote administration tools, or illegitimately as attack tools.	Chapter 5
Trojan horse	Disguises itself as useful program while masking hidden malicious purpose.	Setiri, Hydan	Chapter 6
User-level RootKit	Replace or modify executable programs used by system administrators and users.	Linux RootKit (LRK) family, Universal RootKit, FakeGINA	Chapter 7
Kernel-level RootKits	Manipulate the heart of the operating system, the kernel, to hide and create backdoors.	Adore Kernel Intrusion System	Chapter 8
Combination malware	Combine various techniques already described to increase effectiveness.	Lion, Bugbear.B	Chapter 9

People frequently confuse these categories of malicious code, and use inappropriate terms for various attacks. I hear otherwise freakishly brilliant people mistakenly refer to a worm as a Trojan horse. Others talk about RootKits, but accidentally call them viruses. Sure, this improper use of terminology is confusing, but the issue goes beyond mere semantics. If you don't understand the differences in the categories of malicious code, you won't be able to see how specific defenses can help. If you think a RootKit is synonymous with a virus, you might think you've handled the problem with your antivirus tool. However, you've only scratched the surface of true defenses for that problem. Sure, some of the defenses apply against multiple types of attack. Yet a clear understanding of each malicious code vector will help to make sure you have the comprehensive defenses you require. One of the main purposes of this book is to clarify the differences in various types of malicious code so you can apply the appropriate defenses in your environment.

Although it is immensely useful to get this terminology correct when referring to malicious code and the associated defenses, it should be noted that there is some crossover between these breeds. Some tools are both viruses and worms. Likewise, some worms carry backdoors or RootKits. Most of the developers of these tools don't sit down to create a single tool in a single category. No, they brainstorm about the capabilities they desire, and sling some code to accomplish their varied goals. You can't send a worm to do a kernel-level RootKit's job, unless the worm carries a kernel-level RootKit embedded in it. This intermingling gives rise to the combination malware category included in Table 1-1.

Malicious Code History

Although we've witnessed a huge increase in malicious code attacks in the last few years, malware is certainly not new. Attackers have been churning out highly effective evil programs for decades. However, with the constant evolutionary improvement in the capabilities of these attack tools, and the rapid spread of the Internet into every nook and cranny of our economy, today's malicious code has far greater impact than the attacks of yesteryear. Let's take a nostalgic stroll down memory lane to get an idea of the roots of malicious code and to understand the direction these tools are heading in the future. Figure 1.1 shows a plot of these major malicious code events over the past 20 or so years.

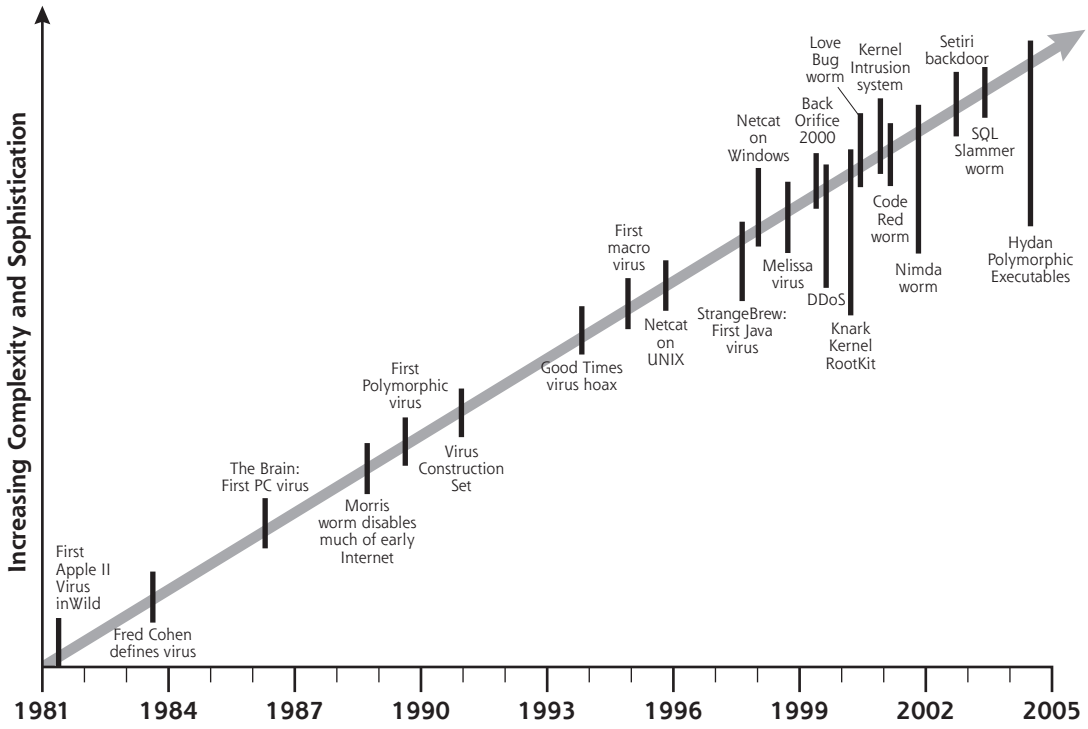

Figure 1.1
More than 20 years of malicious code.

Don't worry if you do not yet understand all of the tools and concepts described in Figure 1.1. The remainder of the book will address each of these issues in far more detail. At this point, however, the major themes I want you to note in Figure 1.1 include these:

- *The increasing complexity and sophistication of malicious software:* We went from fairly simple Apple II viruses that infected games to the complex kernel manipulation tools and powerful worms of this new millennium. The newer tools are very crafty in their rapid infection and extreme stealth techniques.
- *Acceleration of the rate of release of innovative tools and techniques:* New concepts in malicious code started slowly, but have certainly picked up steam over time. Especially over the past five years, we've seen the rapid release of amazing new tools, and this trend is only increasing. Just when I think I've seen it all, the computer underground releases an astonishing (and sometimes frightening) new tool.

- *Movement from viruses to worms to kernel-level exploitation:* In the olden days of malicious code, most of the action revolved around viruses and infecting executable programs. Over the past five years, however, we've seen a major focus on worms, as well as exploiting systems at the kernel level.

These three themes are very intertwined, and feed off of each other as malicious code authors borrow ideas and innovate. By tracing through these significant milestones in malicious code history, we can pay special attention to each of these important trends:

- 1981–1982–*First Reported Computer Viruses*: At least three separate viruses, including Elk Cloner, were discovered in games for the Apple II computer system, although the word *virus* wasn't yet applied to this malicious code.
- 1983–*Formal Definition of Computer Virus:* Fred Cohen defines a computer virus as "a program that can infect other programs by modifying them to include a, possibly evolved, version of itself" [2].
- 1986–*First PC Virus*: The so-called Brain virus infected Microsoft DOS systems, an important harbinger of malicious code to come, as the popular DOS and later Windows operating systems would become a primary target for viruses and worms [3].
- 1988–*Morris Internet Worm:* Written by Robert Tappan Morris, Jr., and released in November, this primordial worm disabled much of the early Internet, making news headlines around the globe.
- 1990–*First Polymorphic Viruses:* To evade antivirus systems, these viruses altered their own appearance every time they ran, opening up the frontier of polymorphic code that is still being explored in research today.
- 1991–*Virus Construction Set (VCS) Released:* In March, this tool hit the bulletin board system community and gave aspiring virus writers a simple toolkit to create their own customized malicious code.
- 1994–*Good Times Virus Hoax:* This virus didn't infect computers. Instead, it was entirely fictional. However, concern about this virus spread from human to human via word of mouth as frightened people warned others about impending doom from this totally bogus malicious code scam [4].
- 1995–*First Macro Viruses:* This particularly nasty strain of viruses was implemented in Microsoft Word macro languages, infecting

document files. These techniques soon spread to other macro languages in other programs.

- 1996–*Netcat released for UNIX:* This tool written by Hobbit remains *the* most popular backdoor for UNIX systems to this day. Although it has a myriad of legitimate and illicit uses, Netcat is often abused as a backdoor.
- 1998–*Netcat released for Windows:* Netcat is no slouch on Windows systems either. Written by Weld Pond, it is used as an extremely popular backdoor on Windows systems as well.
- 1998–*Back Orifice:* This tool released in July by Cult of the Dead Cow (cDc), a hacking group, allowed for remote control of Windows systems across the network, another increasingly popular feature set.
- 1998–*First Java Virus:* The StrangeBrew virus infected other Java programs, bringing virus concerns into the realm of Web-based applications.
- 1999–*The Melissa Virus/Worm*: Released in March, this Microsoft Word macro virus infected thousands of computer systems around the globe by spreading through e-mail. It was both a virus and a worm in that it infected a document file, yet propagated via the network.
- 1999–*Back Orifice 2000 (BO2K)*: In July, cDc released this completely rewritten version of Back Orifice for remote control of a Windows system. The new version sported a slick point-and-click interface, an Application Programming Interface (API) for extending its functionality, and remote control of the mouse, keyboard, and screen.
- 1999–*Distributed Denial of Service Agents:* In late summer, the Tribe Flood Network (TFN) and Trin00 denial of service agents were released. These tools offered an attacker control of dozens, hundreds, or even thousands of machines with an installed zombie via a single client machine. With a centralized point of coordination, these distributed agents could launch a devastating flood or other attack.
- 1999–*Knark Kernel-Level RootKit:* In November, someone called Creed released this tool built on earlier ideas for kernel manipulation on Linux systems. Knark included a complete toolkit for tweaking the Linux kernel so an attacker could very effectively hide files, processes, and network activity.

- 2000–*Love Bug:* In May, this VBScript worm shut down tens of thousands of systems around the world as it spread via several Microsoft Outlook weaknesses.
- 2001–*Code Red Worm:* In July, this worm spread via a buffer overflow in Microsoft's IIS Web server product. Over 250,000 machines fell victim in less than eight hours.
- 2001–*Kernel Intrusion System:* Also in July, this tool by Optyx revolutionized the manipulation of Linux kernels by including an easy-to-use graphical user interface (GUI) and extremely effective hiding mechanisms.
- 2001–*Nimda Worm:* Only a week after the September 11 terrorist attacks, this extremely virulent worm included numerous methods for infecting Windows machines, including Web server buffer overflows, Web browser exploits, Outlook e-mail attacks, and file sharing.
- 2002–*Setiri Backdoor:* Although never formally released, this Trojan horse tool has the ability to bypass personal firewalls, network firewalls, and Network Address Translation devices by posing as an invisible browser.
- 2003–*SQL Slammer Worm:* In January 2003, this worm spread rapidly, disabling several Internet service providers in South Korea and briefly causing problems throughout the world.
- 2003–*Hydan Executable Steganography Tool:* In February, this tool offered its users the ability to hide data inside of executables using polymorphic coding techniques on Linux, BSD, and Windows executables. These concepts could also be extended for antivirus and intrusion detection system evasion.

Things didn't stop there, however. Attackers continue to hone their wares, coming up with newer and nastier malicious code on a regular basis. Throughout this book, we'll explore many specimens from this list, as well as trends on the malicious code of the future.

Why This Book?

Just between you and me, have you noticed how the information security bookshelf at your favorite bookstore (whether it's real-world or virtual) is burgeoning under the weight of tons of titles? Some of them are incredibly helpful. However, it seems that a brand-spanking new security book is competing for your attention every 47 seconds, and you might be wondering how this book is different and why you should read it.

First, as discussed earlier in this chapter, controlling malicious code is an extremely relevant topic. System administrators, network personnel, home users, and especially security practitioners need to defend their network from these attacks, which are getting nastier all the time. Worms, Trojan horses, and RootKits are not a thing of the past. They are a sign of the even nastier stuff to come, and you better be ready. This book will help you get the skills you need to handle such attacks.

Second, our focus here will be on practicality. Throughout the book, we'll discuss time-tested, real-world actions you can take to secure your systems from attack. Our goal will be to give you the concepts and skills you need to do your job as a system, network, or security administrator. The book also includes a full chapter devoted to analysis tools for scrutinizing malicious code under a microscope. Following the tips in Chapter 11, you'll be able to construct a top-notch defender's toolkit to analyze the malicious code you discover in the wild.

Third, this books aims to build on what was covered in other books before, in an effort to make malicious code defenses understandable and practical. A while back, I wrote a book titled *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses.* That earlier book describes the end-to-end process attackers used in compromising systems. *Counter Hack* gives you the big picture of computer attacks, from reconnaissance to covering tracks. This book is not a second edition of *Counter Hack*, nor is it a regurgitation of that book. This book focuses like a laser beam on one of the biggest areas of concern: malicious code. We addressed malicious code in just one chapter of *Counter Hack.* Here, we get to focus a dozen chapters on one of the most interesting and rapidly developing areas of computer attacks, getting into far more depth on this topic than my earlier book. Additionally, attackers haven't been resting on their laurels since the release of *Counter Hack.* This book includes some of the more late-breaking tools and techniques, as most of the action in computer attacks and techniques over the past few years has dealt with newer and nastier malicious code tricks.

Finally, this book tries to encourage you to have fun with this stuff. Don't be intimidated by your computer, the attackers, or malicious code. The book uses a little irreverent humor here and there, but (I hope) stays within the bounds of good taste (well, we'll at least try, exploding virtual hamsters notwithstanding). With a tiny of bit humor, this book tries to encourage you to get comfortable with and actually test some of the tools we'll cover. I strongly encourage you to run the attack and defensive tools we'll discuss in a laboratory of your own to

see how they work. Chapter 11 tells you how you can build your very own low-cost experimental mininetwork for analysis of malicious code and the associated defenses. However, make sure you experiment on a lab network, physically disconnected from your production network and the Internet. In such a controlled environment, you can feel free to safely mess around with these nasty tools so you can be ready if and when a bad guy unleashes them on your production environment.

What To Expect

Throughout this book, we'll use a few standard pictures and phrases to refer to recurring ideas. As we're discussing various attacks against computer systems, we'll show the attack using illustrations. For any figure in this book where we need to differentiate between the attacking system and the victim machine, we'll illustrate the attacking machine with a black hat, as shown in Figure 1.2. That way, you'll be able to quickly determine where the bad guy sits in the overall architecture of the attack.

Additionally, when referring to the perpetrators of an attack, we'll use the word *attacker* or the phrase *bad guy*. We won't use the word *hacker*, as that terminology has become too loaded with political baggage. Some people think of hackers as noble explorers, whereas others assume the word implies criminal wrongdoing. By using the words *attacker* and *bad guy*, we'll sidestep such controversies, which often spread more heat than light.

Also, it's important to note that this book is operating system agnostic. We don't worship at the shrine of Linux, Solaris, or Windows, but instead mention attack techniques that could function in a variety of

Figure 1.2
In this book, the attacker's machines are illustrated with a black hat.

operating system environments. Throughout the book, we'll discuss attacks against both Windows and UNIX systems, jumping back and forth between the two operating systems to illustrate various points.

This approach is based on my own strong feeling that to be a solid security person, you need to be ready to operate in both a Windows and a UNIX environment, as most organizations have some mix of the two classes of operating systems. If you are prepared for attacks against both types of systems, your defenses will be far better, and you will be more valuable to your employer. Using this philosophy, most chapters include attacks against Windows and UNIX, with a given tool from either side to illustrate the point. If we cover a particular attack against Windows, keep in mind that analogous attacks are available for UNIX, and vice versa.

In some of the later chapters of the book (especially Chapters 7 and 8, which deal with RootKits), the malware undermines components of the operating system itself. Therefore, because such attacks are often highly operating-system-specific, we'll split those chapters in half, first dealing with UNIX-oriented attacks and later dealing with Windows attacks.

Although various chapters cover both Windows and UNIX-based tools, each chapter of this book deals with a specific type of malicious code. For each type of malware, we start by introducing the concepts that classify each type, exploring the defining characteristics of the breed. Then, each chapter describes the techniques used by that type of malware, as well as prominent examples, so you can understand what you are up against on your systems. This discussion includes a description of the current capabilities of the latest tools, as well as future evolutionary trends for that type of attack. Finally, we get to the must useful stuff; each chapter includes a description of the defenses needed to handle that type of malicious code. The chapters in this book include the following:

Chapter 1: Introduction: That's this intro . . . you probably figured that out already!

Chapter 2: Viruses: Viruses were the very first malicious code examples unleashed more than 20 years ago. They've had the most time to evolve, and include some highly innovative strategies that are being borrowed by other malicious code tools. This chapter describes the current virus threat and what you need to do to stop this vector of attack.

Chapter 3: Worms: By spreading via a network, worms can pack a wallop, conquering hundreds of thousands of systems in a matter of

hours. Given their inherent power, worms are getting a huge amount of research and development attention, which we'll analyze in this chapter.

Chapter 4: Malicious Mobile Code: Attackers are devising novel ways for delivering malicious code via the World-Wide Web and e-mail. If you run a Web browser or e-mail reader (and who doesn't?), this chapter describes the different types of malicious mobile code, as well as how you can defend your browsers from attack.

Chapter 5: Backdoors: Attackers use backdoors to access a system and bypass normal security controls. State-of-the-art backdoors give the attacker significant control over a target system. This chapter explores the most popular and powerful backdoors available today.

Chapter 6: Trojan Horses: By posing as a nice, happy program, a Trojan horse tricks users and administrators. These programs look fun or useful, but really hide a sinister plot to undermine your security from within. This chapter identifies classic Trojan horse strategies and shows you how to stop them in their tracks.

Chapter 7: User-Level RootKits: By replacing the programs built into your operating system with RootKits, an attacker can hide on your machine without your knowledge. This chapter discusses user-level RootKits so you can defend your network against such shenanigans.

Chapter 8: Kernel-Level Modifications: If attackers can modify the heart of your operating system, the kernel itself, they can achieve complete domination of your system in a highly invisible fashion. In this chapter, we'll look at this active area of new development and recommend solid practices for stopping kernel-level attacks.

Chapter 9: Going Deeper and Combo Malware: The techniques discussed throughout this book aren't static. Sometime in the future, attackers might try undermining our hardware, with BIOS and CPU-level attacks. Furthermore, attackers are developing newer attacks by cobbling various types of malicious software together into Frankenstein-like monsters. This chapter addresses such deeper malware as well as combinations of various malicious code types.

Chapter 10: Putting It All Together: There's nothing like real-world examples to help clarify abstract concepts. In this chapter, we'll go over three sample scenarios of malicious code attacks, and determine how various organizations could have prevented disaster. Each scenario has a movie theme, just to keep it fun. Let's learn from the mistakes of others and improve our security.

Chapter 11: Malware Analysis: This chapter gives you recipes for creating your own malicious code analysis laboratory using cheap hardware and software.

Chapter 12: Conclusion: In this chapter, we'll go over some future predictions and areas where you can get more information about malicious code.

References

[1] Colleen O'Hara and FSW Staff, "Agencies Fight off 'Melissa' Macro Virus," *Federal Computer Week*, April 5, 1999, *www.fcw.com/fcw/articles/1999/FCW_040599_261.asp*

[2] Fred Cohen, *Computer Viruses: Theory and Experiments*, Fred Cohen & Associates, 1984, *http://all.net/books/virus/index.html*

[3] Joe Wells, "Virus Timeline," IBM Research, August 1996, *www.research.ibm.com/antivirus/timeline.htm*

[4] CIAC, U.S. Department of Energy, "The Good Times Virus Is an Urban Legend," December, 1994, *http://ciac.llnl.gov/ciac/notes/Notes04c.shtml*