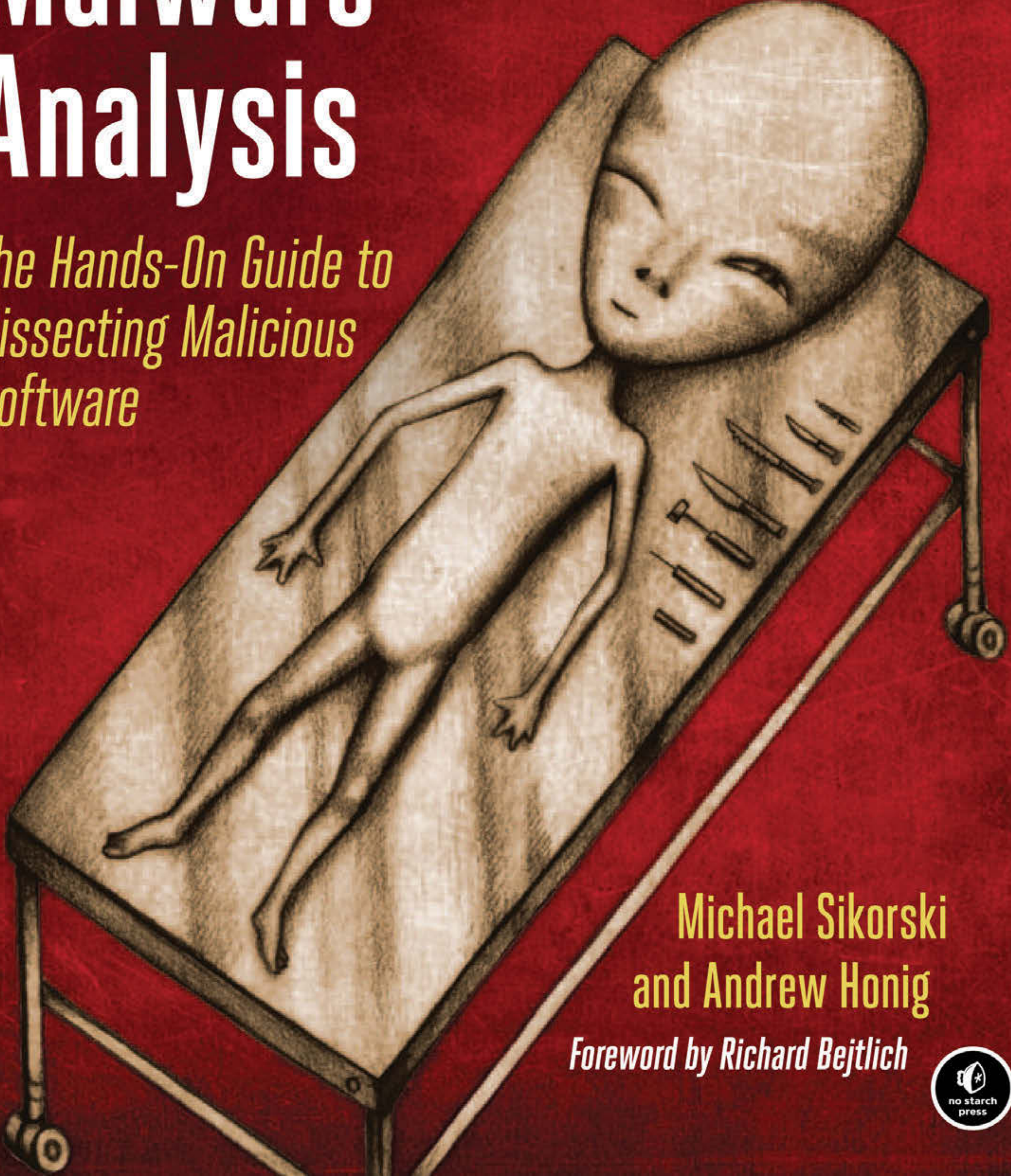
Practical Malware Analysis

The Hands-On Guide to Dissecting Malicious Software



Michael Sikorski
and Andrew Honig

Foreword by Richard Bejtlich

no starch press

PRAISE FOR *PRACTICAL MALWARE ANALYSIS*

Digital Forensics Book of the Year, FORENSIC 4CAST AWARDS 2013

"A hands-on introduction to malware analysis. I'd recommend it to anyone who wants to dissect Windows malware."
—**Ilfak Guilfanov**, CREATOR OF IDA PRO

"The book every malware analyst should keep handy."
—**Richard Bejtlich**, CSO OF MANDIANT & FOUNDER OF TAOSECURITY

"This book does exactly what it promises on the cover; it's crammed with detail and has an intensely practical approach, but it's well organised enough that you can keep it around as handy reference."
—**Mary Branscombe**, ZDNET

"If you're starting out in malware analysis, or if you are are coming to analysis from another discipline, I'd recommend having a nose."
—**Paul Baccas**, NAKED SECURITY FROM SOPHOS

"An excellent crash course in malware analysis."
—**Dino Dai Zovi**, INDEPENDENT SECURITY CONSULTANT

"The most comprehensive guide to analysis of malware, offering detailed coverage of all the essential skills required to understand the specific challenges presented by modern malware."
—**Chris Eagle**, SENIOR LECTURER OF COMPUTER SCIENCE AT THE NAVAL POSTGRADUATE SCHOOL

"A great introduction to malware analysis. All chapters contain detailed technical explanations and hands-on lab exercises to get you immediate exposure to real malware."
—**Sebastian Porst**, GOOGLE SOFTWARE ENGINEER

"Brings reverse-engineering to readers of all skill levels. Technically rich and accessible, the labs will lead you to a deeper understanding of the art and science of reverse-engineering. I strongly believe this will become the defacto text for learning malware analysis in the future."
—**Danny Quist**, PHD, FOUNDER OF OFFENSIVE COMPUTING

"An awesome book . . . written by knowledgeable authors who possess the rare gift of being able to communicate their knowledge through the written word."
—Richard Austin, IEEE CIPHER

"If you only read one malware book or are looking to break into the world of malware analysis, this is the book to get."
—Patrick Engebretson, IA PROFESSOR, DAKOTA STATE UNIVERSITY AND AUTHOR OF *The Basics of Hacking and Pen Testing*

"An excellent addition to the course materials for an advanced graduate level course on Software Security or Intrusion Detection Systems. The labs are especially useful to students in teaching the methods to reverse-engineer, analyze, and understand malicious software."
—Sal Stolfo, PROFESSOR, COLUMBIA UNIVERSITY

"The explanation of the tools is clear, the presentation of the process is lucid, and the actual detective work fascinating. All presented clearly and hitting just the right level so that developers with no previous experience in this particular area can participate fully. Highly recommended."
—Dr. Dobb's

"This book is like having your very own personal malware analysis teacher without the expensive training costs."
—Dustin Schultz, THEXPLOIT

"I highly recommend this book to anyone looking to get their feet wet in malware analysis or just looking for a good desktop reference on the subject."
—Pete Arzamendi, 403 LABS

"I do not see how anyone who has hands-on responsibility for security of Windows systems can rationalize not being familiar with these tools."
—Stephen Northcutt, SANS INSTITUTE

PRACTICAL MALWARE ANALYSIS

The Hands-On Guide to Dissecting Malicious Software

by Michael Sikorski and Andrew Honig

no starch
press

San Francisco

ISBN-10: 1-59327-290-1
ISBN-13: 978-1-59327-290-6

Publisher: William Pollock
Production Editor: Alison Law
Cover Illustration: Hugh D'Andrade
Interior Design: Octopod Studios
Developmental Editors: William Pollock and Tyler Ortman
Technical Reviewer: Stephen Lawler
Copyeditor: Marilyn Smith
Compositor: Riley Hoffman
Proofreader: Irene Barnard
Indexer: Nancy Guenther

For information on distribution, translations, or bulk sales, please contact No Starch Press, Inc. directly:

No Starch Press, Inc.
245 8th Street, San Francisco, CA 94103
phone: 1.415.863.9900; info@nostarch.com; www.nostarch.com

Library of Congress Cataloging-in-Publication Data

```
Sikorski, Michael.
  Practical malware analysis : the hands-on guide to dissecting malicious software / by Michael Sikorski,
Andrew Honig.
      p. cm.
  ISBN 978-1-59327-290-6 -- ISBN 1-59327-290-1
 1.  Malware (Computer software) 2.  Computer viruses. 3.  Debugging in computer science. 4.  Computer security.
I. Honig, Andrew. II. Title.
  QA76.76.C68S534 2012
  005.8'4--dc23
                                                                          2012000214
```

The information in this book is distributed on an "As Is" basis, without warranty. While every precaution has been taken in the preparation of this work, neither the authors nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

BRIEF CONTENTS

PART 1: BASIC ANALYSIS

PART 2: ADVANCED STATIC ANALYSIS

PART 3: ADVANCED DYNAMIC ANALYSIS

PART 4: MALWARE FUNCTIONALITY

PART 5: ANTI-REVERSE-ENGINEERING

PART 6: SPECIAL TOPICS

CONTENTS IN DETAIL

2
MALWARE ANALYSIS IN VIRTUAL MACHINES 29

3
BASIC DYNAMIC ANALYSIS 39

PART 2
ADVANCED STATIC ANALYSIS

4
A CRASH COURSE IN X86 DISASSEMBLY 65

5
IDA PRO 87

6
RECOGNIZING C CODE CONSTRUCTS IN ASSEMBLY 109

7
ANALYZING MALICIOUS WINDOWS PROGRAMS 135

PART 3
ADVANCED DYNAMIC ANALYSIS

PART 4
MALWARE FUNCTIONALITY

11
MALWARE BEHAVIOR 231

12
COVERT MALWARE LAUNCHING 253

PART 5
ANTI-REVERSE-ENGINEERING

15
ANTI-DISASSEMBLY 327

16
ANTI-DEBUGGING 351

17
ANTI-VIRTUAL MACHINE TECHNIQUES 369

ABOUT THE AUTHORS

Michael Sikorski is a computer security consultant at Mandiant. He reverse-engineers malicious software in support of incident response investigations and provides specialized research and development security solutions to the company's federal client base. Mike created a series of courses in malware analysis and teaches them to a variety of audiences including the FBI and Black Hat. He came to Mandiant from MIT Lincoln Laboratory, where he performed research in passive network mapping and penetration testing. Mike is also a graduate of the NSA's three-year System and Network Interdisciplinary Program (SNIP). While at the NSA, he contributed to research in reverse-engineering techniques and received multiple invention awards in the field of network analysis.

Andrew Honig is an information assurance expert for the Department of Defense. He teaches courses on software analysis, reverse-engineering, and Windows system programming at the National Cryptologic School and is a Certified Information Systems Security Professional. Andy is publicly credited with several zero-day exploits in VMware's virtualization products and has developed tools for detecting innovative malicious software, including malicious software in the kernel. An expert in analyzing and understanding both malicious and non-malicious software, he has over 10 years of experience as an analyst in the computer security industry.

About the Technical Reviewer

Stephen Lawler is the founder and president of a small computer software and security consulting firm. Stephen has been actively working in information security for over seven years, primarily in reverse-engineering, malware analysis, and vulnerability research. He was a member of the Mandiant Malware Analysis Team and assisted with high-profile computer intrusions affecting several Fortune 100 companies. Previously he worked in ManTech International's Security and Mission Assurance (SMA) division, where he discovered numerous zero-day vulnerabilities and software exploitation techniques as part of ongoing software assurance efforts. In a prior life that had nothing to do with computer security, he was lead developer for the sonar simulator component of the US Navy SMMTT program.

About the Contributing Authors

Nick Harbour is a malware analyst at Mandiant and a seasoned veteran of the reverse-engineering business. His 13-year career in information security began as a computer forensic examiner and researcher at the Department of Defense Computer Forensics Laboratory. For the last six years, Nick has been with Mandiant and has focused primarily on malware analysis. He is a researcher in the field of anti-reverse-engineering techniques, and he has written several packers and code obfuscation tools, such as PE-Scrambler. He has presented at Black Hat and Defcon several times on the topic of anti-reverse-engineering and anti-forensics techniques. He is the primary developer and teacher of a Black Hat Advanced Malware Analysis course.

Lindsey Lack is a technical director at Mandiant with over twelve years of experience in information security, specializing in malware reverse-engineering, network defense, and security operations. He has helped to create and operate a Security Operations Center, led research efforts in network defense, and developed secure hosting solutions. He has previously held positions at the National Information Assurance Research Laboratory, the Executive Office of the President (EOP), Cable and Wireless, and the US Army. In addition to a bachelor's degree in computer science from Stanford University, Lindsey has also received a master's degree in computer science with an emphasis in information assurance from the Naval Postgraduate School.

Jerrold "Jay" Smith is a principal consultant at Mandiant, where he specializes in malware reverse-engineering and forensic analysis. In this role, he has contributed to many incident responses assisting a range of clients from Fortune 500 companies. Prior to joining Mandiant, Jay was with the NSA, but he's not allowed to talk about that. Jay holds a bachelor's degree in electrical engineering and computer science from UC Berkeley and a master's degree in computer science from Johns Hopkins University.

FOREWORD

Few areas of digital security seem as asymmetric as those involving malware, defensive tools, and operating systems.

In the summer of 2011, I attended Peiter (Mudge) Zatko's keynote at Black Hat in Las Vegas, Nevada. During his talk, Mudge introduced the asymmetric nature of modern software. He explained how he analyzed 9,000 malware binaries and counted an average of 125 lines of code (LOC) for his sample set.

You might argue that Mudge's samples included only "simple" or "pedestrian" malware. You might ask, what about something truly "weaponized"? Something like (hold your breath)—Stuxnet? According to Larry L. Constantine,[1] Stuxnet included about 15,000 LOC and was therefore 120 times the size of a 125 LOC average malware sample. Stuxnet was highly specialized and targeted, probably accounting for its above-average size.

Leaving the malware world for a moment, the text editor I'm using (gedit, the GNOME text editor) includes *gedit.c* with 295 LOC—and *gedit.c* is only one of 128 total source files (along with 3 more directories) published

[1] *http://www.informit.com/articles/article.aspx?p=1686289*

in the GNOME GIT source code repository for gedit.[2] Counting all 128 files and 3 directories yields 70,484 LOC. The ratio of legitimate application LOC to malware is over 500 to 1. Compared to a fairly straightforward tool like a text editor, an average malware sample seems very efficient!

Mudge's 125 LOC number seemed a little low to me, because different definitions of "malware" exist. Many malicious applications exist as "suites," with many functions and infrastructure elements. To capture this sort of malware, I counted what you could reasonably consider to be the "source" elements of the Zeus Trojan (*.cpp*, *.obj*, *.h*, etc.) and counted 253,774 LOC. When comparing a program like Zeus to one of Mudge's average samples, we now see a ratio of over 2,000 to 1.

Mudge then compared malware LOC with counts for security products meant to intercept and defeat malicious software. He cited 10 million as his estimate for the LOC found in modern defensive products. To make the math easier, I imagine there are products with at least 12.5 million lines of code, bringing the ratio of offensive LOC to defensive LOC into the 100,000 to 1 level. In other words, for every 1 LOC of offensive firepower, defenders write 100,000 LOC of defensive bastion.

Mudge also compared malware LOC to the operating systems those malware samples are built to subvert. Analysts estimate Windows XP to be built from 45 million LOC, and no one knows how many LOC built Windows 7. Mudge cited 150 million as a count for modern operating systems, presumably thinking of the latest versions of Windows. Let's revise that downward to 125 million to simplify the math, and we have a 1 million to 1 ratio for size of the target operating system to size of the malicious weapon capable of abusing it.

Let's stop to summarize the perspective our LOC counting exercise has produced:

120:1	Stuxnet to average malware
500:1	Simple text editor to average malware
2,000:1	Malware suite to average malware
100,000:1	Defensive tool to average malware
1,000,000:1	Target operating system to average malware

From a defender's point of view, the ratios of defensive tools and target operating systems to average malware samples seem fairly bleak. Even swapping the malware suite size for the average size doesn't appear to improve the defender's situation very much! It looks like defenders (and their vendors) expend a lot of effort producing thousands of LOC, only to see it brutalized by nifty, nimble intruders sporting far fewer LOC.

What's a defender to do? The answer is to take a page out of the playbook used by any leader who is outgunned—redefine an "obstacle" as an "opportunity"! Forget about the size of the defensive tools and target operating systems—there's not a whole lot you can do about them. Rejoice in the fact that malware samples are as small (relatively speaking) as they are.

[2] *http://git.gnome.org/browse/gedit/tree/gedit?id=3.3.1*

Imagine trying to understand how a defensive tool works at the source code level, where those 12.5 million LOC are waiting. That's a daunting task, although some researchers assign themselves such pet projects. For one incredible example, read "Sophail: A Critical Analysis of Sophos Antivirus" by Tavis Ormandy,[3] also presented at Black Hat Las Vegas in 2011. This sort of mammoth analysis is the exception and not the rule.

Instead of worrying about millions of LOC (or hundreds or tens of thousands), settle into the area of one thousand or less—the place where a significant portion of the world's malware can be found. As a defender, your primary goal with respect to malware is to determine what it does, how it manifests in your environment, and what to do about it. When dealing with reasonably sized samples and the right skills, you have a chance to answer these questions and thereby reduce the risk to your enterprise.

If the malware authors are ready to provide the samples, the authors of the book you're reading are here to provide the skills. *Practical Malware Analysis* is the sort of book I think every malware analyst should keep handy. If you're a beginner, you're going to read the introductory, hands-on material you need to enter the fight. If you're an intermediate practitioner, it will take you to the next level. If you're an advanced engineer, you'll find those extra gems to push you even higher—and you'll be able to say "read this fine manual" when asked questions by those whom you mentor.

Practical Malware Analysis is really two books in one—first, it's a text showing readers how to analyze modern malware. You could have bought the book for that reason alone and benefited greatly from its instruction. However, the authors decided to go the extra mile and essentially write a second book. This additional tome could have been called *Applied Malware Analysis*, and it consists of the exercises, short answers, and detailed investigations presented at the end of each chapter and in Appendix C. The authors also wrote all the malware they use for examples, ensuring a rich yet safe environment for learning.

Therefore, rather than despair at the apparent asymmetries facing digital defenders, be glad that the malware in question takes the form it currently does. Armed with books like *Practical Malware Analysis*, you'll have the edge you need to better detect and respond to intrusions in your enterprise or that of your clients. The authors are experts in these realms, and you will find advice extracted from the front lines, not theorized in an isolated research lab. Enjoy reading this book and know that every piece of malware you reverse-engineer and scrutinize raises the opponent's costs by exposing his dark arts to the sunlight of knowledge.

Richard Bejtlich (@taosecurity)
Chief Security Officer, Mandiant and Founder of TaoSecurity
Manassas Park, Virginia
January 2, 2012

[3] *http://dl.packetstormsecurity.net/papers/virus/Sophail.pdf*

ACKNOWLEDGMENTS

Thanks to Lindsey Lack, Nick Harbour, and Jerrold "Jay" Smith for contributing chapters in their areas of expertise. Thanks to our technical reviewer Stephen Lawler who single-handedly reviewed over 50 labs and all of our chapters. Thanks to Seth Summersett, William Ballenthin, and Stephen Davis for contributing code for this book.

Special thanks go to everyone at No Starch Press for their effort. Alison, Bill, Travis, and Tyler: we were glad to work with you and everyone else at No Starch Press.

Individual Thanks

Mike: I dedicate this book to Rebecca—I couldn't have done this without having such a supportive and loving person in my life.

Andy: I'd like to thank Molly, Claire, and Eloise for being the best family a guy could have.

INTRODUCTION

The phone rings, and the networking guys tell you that you've been hacked and that your customers' sensitive information is being stolen from your network. You begin your investigation by checking your logs to identify the hosts involved. You scan the hosts with antivirus software to find the malicious program, and catch a lucky break when it detects a trojan horse named *TROJ.snapAK*. You delete the file in an attempt to clean things up, and you use network capture to create an intrusion detection system (IDS) signature to make sure no other machines are infected. Then you patch the hole that you think the attackers used to break in to ensure that it doesn't happen again.

Then, several days later, the networking guys are back, telling you that sensitive data is being stolen from your network. It seems like the same attack, but you have no idea what to do. Clearly, your IDS signature failed, because more machines are infected, and your antivirus software isn't providing enough protection to isolate the threat. Now upper management demands an explanation of what happened, and all you can tell them about the malware is that it was *TROJ.snapAK*. You don't have the answers to the most important questions, and you're looking kind of lame.

How do you determine exactly what *TROJ.snapAK* does so you can eliminate the threat? How do you write a more effective network signature? How can you find out if any other machines are infected with this malware? How can you make sure you've deleted the entire malware package and not just one part of it? How can you answer management's questions about what the malicious program does?

All you can do is tell your boss that you need to hire expensive outside consultants because you can't protect your own network. That's not really the best way to keep your job secure.

Ah, but fortunately, you were smart enough to pick up a copy of *Practical Malware Analysis*. The skills you'll learn in this book will teach you how to answer those hard questions and show you how to protect your network from malware.

What Is Malware Analysis?

Malicious software, or *malware*, plays a part in most computer intrusion and security incidents. Any software that does something that causes harm to a user, computer, or network can be considered malware, including viruses, trojan horses, worms, rootkits, scareware, and spyware. While the various malware incarnations do all sorts of different things (as you'll see throughout this book), as malware analysts, we have a core set of tools and techniques at our disposal for analyzing malware.

Malware analysis is the art of dissecting malware to understand how it works, how to identify it, and how to defeat or eliminate it. And you don't need to be an uber-hacker to perform malware analysis.

With millions of malicious programs in the wild, and more encountered every day, malware analysis is critical for anyone who responds to computer security incidents. And, with a shortage of malware analysis professionals, the skilled malware analyst is in serious demand.

That said, this is not a book on how to find malware. Our focus is on how to analyze malware once it has been found. We focus on malware found on the Windows operating system—by far the most common operating system in use today—but the skills you learn will serve you well when analyzing malware on any operating system. We also focus on executables, since they are the most common and the most difficult files that you'll encounter. At the same time, we've chosen to avoid discussing malicious scripts and Java programs. Instead, we dive deep into the methods used for dissecting advanced threats, such as backdoors, covert malware, and rootkits.

Prerequisites

Regardless of your background or experience with malware analysis, you'll find something useful in this book.

Chapters 1 through 3 discuss basic malware analysis techniques that even those with no security or programming experience will be able to use to perform malware triage. Chapters 4 through 14 cover more intermediate material that will arm you with the major tools and skills needed to analyze most malicious programs. These chapters do require some knowledge of programming. The more advanced material in Chapters 15 through 19 will be useful even for seasoned malware analysts because it covers strategies and techniques for analyzing even the most sophisticated malicious programs, such as programs utilizing anti-disassembly, anti-debugging, or packing techniques.

This book will teach you how and when to use various malware analysis techniques. Understanding when to use a particular technique can be as important as knowing the technique, because using the wrong technique in the wrong situation can be a frustrating waste of time. We don't cover every tool, because tools change all the time and it's the core skills that are important. Also, we use realistic malware samples throughout the book (which you can download from *http://www.practicalmalwareanalysis.com/* or *http://www.nostarch.com/malware.htm*) to expose you to the types of things that you'll see when analyzing real-world malware.

Practical, Hands-On Learning

Our extensive experience teaching professional reverse-engineering and malware analysis classes has taught us that students learn best when they get to practice the skills they are learning. We've found that the quality of the labs is as important as the quality of the lecture, and without a lab component, it's nearly impossible to learn how to analyze malware.

To that end, lab exercises at the end of most chapters allow you to practice the skills taught in that chapter. These labs challenge you with realistic malware designed to demonstrate the most common types of behavior that you'll encounter in real-world malware. The labs are designed to reinforce the concepts taught in the chapter without overwhelming you with unrelated information. Each lab includes one or more malicious files (which can be downloaded from *http://www.practicalmalwareanalysis.com/* or *http://www.nostarch.com/malware.htm*), some questions to guide you through the lab, short answers to the questions, and a detailed analysis of the malware.

The labs are meant to simulate realistic malware analysis scenarios. As such, they have generic filenames that provide no insight into the functionality of the malware. As with real malware, you'll start with no information, and you'll need to use the skills you've learned to gather clues and figure out what the malware does.

The amount of time required for each lab will depend on your experience. You can try to complete the lab yourself, or follow along with the detailed analysis to see how the various techniques are used in practice.

Most chapters contain three labs. The first lab is generally the easiest, and most readers should be able to complete it. The second lab is meant to be moderately difficult, and most readers will require some assistance from the solutions. The third lab is meant to be difficult, and only the most adept readers will be able to complete it without help from the solutions.

What's in the Book?

Practical Malware Analysis begins with easy methods that can be used to get information from relatively unsophisticated malicious programs, and proceeds with increasingly complicated techniques that can be used to tackle even the most sophisticated malicious programs. Here's what you'll find in each chapter:

- Chapter 0, "Malware Analysis Primer," establishes the overall process and methodology of analyzing malware.
- Chapter 1, "Basic Static Techniques," teaches ways to get information from an executable without running it.
- Chapter 2, "Malware Analysis in Virtual Machines," walks you through setting up virtual machines to use as a safe environment for running malware.
- Chapter 3, "Basic Dynamic Analysis," teaches easy-to-use but effective techniques for analyzing a malicious program by running it.
- Chapter 4, "A Crash Course in x86 Assembly," is an introduction to the x86 assembly language, which provides a foundation for using IDA Pro and performing in-depth analysis of malware.
- Chapter 5, "IDA Pro," shows you how to use IDA Pro, one of the most important malware analysis tools. We'll use IDA Pro throughout the remainder of the book.
- Chapter 6, "Recognizing C Code Constructs in Assembly," provides examples of C code in assembly and teaches you how to understand the high-level functionality of assembly code.
- Chapter 7, "Analyzing Malicious Windows Programs," covers a wide range of Windows-specific concepts that are necessary for understanding malicious Windows programs.
- Chapter 8, "Debugging," explains the basics of debugging and how to use a debugger for malware analysts.
- Chapter 9, "OllyDbg," shows you how to use OllyDbg, the most popular debugger for malware analysts.
- Chapter 10, "Kernel Debugging with WinDbg," covers how to use the WinDbg debugger to analyze kernel-mode malware and rootkits.
- Chapter 11, "Malware Behavior," describes common malware functionality and shows you how to recognize that functionality when analyzing malware.
- Chapter 12, "Covert Malware Launching," discusses how to analyze a particularly stealthy class of malicious programs that hide their execution within another process.
- Chapter 13, "Data Encoding," demonstrates how malware may encode data in order to make it harder to identify its activities in network traffic or on the victim host.

- Chapter 14, "Malware-Focused Network Signatures," teaches you how to use malware analysis to create network signatures that outperform signatures made from captured traffic alone.
- Chapter 15, "Anti-Disassembly," explains how some malware authors design their malware so that it is hard to disassemble, and how to recognize and defeat these techniques.
- Chapter 16, "Anti-Debugging," describes the tricks that malware authors use to make their code difficult to debug and how to overcome those roadblocks.
- Chapter 17, "Anti-Virtual Machine Techniques," demonstrates techniques used by malware to make it difficult to analyze in a virtual machine and how to bypass those techniques.
- Chapter 18, "Packers and Unpacking," teaches you how malware uses packing to hide its true purpose, and then provides a step-by-step approach for unpacking packed programs.
- Chapter 19, "Shellcode Analysis," explains what shellcode is and presents tips and tricks specific to analyzing malicious shellcode.
- Chapter 20, "C++ Analysis," instructs you on how C++ code looks different once it is compiled and how to perform analysis on malware created using C++.
- Chapter 21, "64-Bit Malware," discusses why malware authors may use 64-bit malware and what you need to know about the differences between x86 and x64.
- Appendix A, "Important Windows Functions," briefly describes Windows functions commonly used in malware.
- Appendix B, "Tools for Malware Analysis," lists useful tools for malware analysts.
- Appendix C, "Solutions to Labs," provides the solutions for the labs included in the chapters throughout the book.

Our goal throughout this book is to arm you with the skills to analyze and defeat malware of all types. As you'll see, we cover a lot of material and use labs to reinforce the material. By the time you've finished this book, you will have learned the skills you need to analyze any malware, including simple techniques for quickly analyzing ordinary malware and complex, sophisticated ones for analyzing even the most enigmatic malware.

Let's get started.

0

MALWARE ANALYSIS PRIMER

Before we get into the specifics of how to analyze malware, we need to define some terminology, cover common types of malware, and introduce the fundamental approaches to malware analysis. Any software that does something that causes detriment to the user, computer, or network—such as viruses, trojan horses, worms, rootkits, scareware, and spyware—can be considered *malware*. While malware appears in many different forms, common techniques are used to analyze malware. Your choice of which technique to employ will depend on your goals.

The Goals of Malware Analysis

The purpose of malware analysis is usually to provide the information you need to respond to a network intrusion. Your goals will typically be to determine exactly what happened, and to ensure that you've located all infected machines and files. When analyzing suspected malware, your goal will typically be to determine exactly what a particular suspect binary can do, how to detect it on your network, and how to measure and contain its damage.

Once you identify which files require full analysis, it's time to develop signatures to detect malware infections on your network. As you'll learn throughout this book, malware analysis can be used to develop host-based and network signatures.

Host-based signatures, or indicators, are used to detect malicious code on victim computers. These indicators often identify files created or modified by the malware or specific changes that it makes to the registry. Unlike antivirus signatures, malware indicators focus on what the malware does to a system, not on the characteristics of the malware itself, which makes them more effective in detecting malware that changes form or that has been deleted from the hard disk.

Network signatures are used to detect malicious code by monitoring network traffic. Network signatures can be created without malware analysis, but signatures created with the help of malware analysis are usually far more effective, offering a higher detection rate and fewer false positives.

After obtaining the signatures, the final objective is to figure out exactly how the malware works. This is often the most asked question by senior management, who want a full explanation of a major intrusion. The in-depth techniques you'll learn in this book will allow you to determine the purpose and capabilities of malicious programs.

Malware Analysis Techniques

Most often, when performing malware analysis, you'll have only the malware executable, which won't be human-readable. In order to make sense of it, you'll use a variety of tools and tricks, each revealing a small amount of information. You'll need to use a variety of tools in order to see the full picture.

There are two fundamental approaches to malware analysis: static and dynamic. *Static analysis* involves examining the malware without running it. *Dynamic analysis* involves running the malware. Both techniques are further categorized as basic or advanced.

Basic Static Analysis

Basic static analysis consists of examining the executable file without viewing the actual instructions. Basic static analysis can confirm whether a file is malicious, provide information about its functionality, and sometimes provide information that will allow you to produce simple network signatures. Basic static analysis is straightforward and can be quick, but it's largely ineffective against sophisticated malware, and it can miss important behaviors.

Basic Dynamic Analysis

Basic dynamic analysis techniques involve running the malware and observing its behavior on the system in order to remove the infection, produce effective signatures, or both. However, before you can run malware safely, you must set up an environment that will allow you to study the running malware without risk of damage to your system or network. Like basic static analysis techniques, basic dynamic analysis techniques can be used by most people without deep programming knowledge, but they won't be effective with all malware and can miss important functionality.

Advanced Static Analysis

Advanced static analysis consists of reverse-engineering the malware's internals by loading the executable into a disassembler and looking at the program instructions in order to discover what the program does. The instructions are executed by the CPU, so advanced static analysis tells you exactly what the program does. However, advanced static analysis has a steeper learning curve than basic static analysis and requires specialized knowledge of disassembly, code constructs, and Windows operating system concepts, all of which you'll learn in this book.

Advanced Dynamic Analysis

Advanced dynamic analysis uses a debugger to examine the internal state of a running malicious executable. Advanced dynamic analysis techniques provide another way to extract detailed information from an executable. These techniques are most useful when you're trying to obtain information that is difficult to gather with the other techniques. In this book, we'll show you how to use advanced dynamic analysis together with advanced static analysis in order to completely analyze suspected malware.

Types of Malware

When performing malware analysis, you will find that you can often speed up your analysis by making educated guesses about what the malware is trying to do and then confirming those hypotheses. Of course, you'll be able to make better guesses if you know the kinds of things that malware usually does. To that end, here are the categories that most malware falls into:

Backdoor Malicious code that installs itself onto a computer to allow the attacker access. Backdoors usually let the attacker connect to the computer with little or no authentication and execute commands on the local system.

Botnet Similar to a backdoor, in that it allows the attacker access to the system, but all computers infected with the same botnet receive the same instructions from a single command-and-control server.

Downloader Malicious code that exists only to download other malicious code. Downloaders are commonly installed by attackers when they first gain access to a system. The downloader program will download and install additional malicious code.

Information-stealing malware Malware that collects information from a victim's computer and usually sends it to the attacker. Examples include sniffers, password hash grabbers, and keyloggers. This malware is typically used to gain access to online accounts such as email or online banking.

Launcher Malicious program used to launch other malicious programs. Usually, launchers use nontraditional techniques to launch other malicious programs in order to ensure stealth or greater access to a system.

Rootkit Malicious code designed to conceal the existence of other code. Rootkits are usually paired with other malware, such as a backdoor, to allow remote access to the attacker and make the code difficult for the victim to detect.

Scareware Malware designed to frighten an infected user into buying something. It usually has a user interface that makes it look like an antivirus or other security program. It informs users that there is malicious code on their system and that the only way to get rid of it is to buy their "software," when in reality, the software it's selling does nothing more than remove the scareware.

Spam-sending malware Malware that infects a user's machine and then uses that machine to send spam. This malware generates income for attackers by allowing them to sell spam-sending services.

Worm or virus Malicious code that can copy itself and infect additional computers.

Malware often spans multiple categories. For example, a program might have a keylogger that collects passwords and a worm component that sends spam. Don't get too caught up in classifying malware according to its functionality.

Malware can also be classified based on whether the attacker's objective is mass or targeted. Mass malware, such as scareware, takes the shotgun approach and is designed to affect as many machines as possible. Of the two objectives, it's the most common, and is usually the less sophisticated and easier to detect and defend against because security software targets it.

Targeted malware, like a one-of-a-kind backdoor, is tailored to a specific organization. Targeted malware is a bigger threat to networks than mass malware, because it is not widespread and your security products probably won't protect you from it. Without a detailed analysis of targeted malware, it is nearly impossible to protect your network against that malware and to remove infections. Targeted malware is usually very sophisticated, and your analysis will often require the advanced analysis skills covered in this book.

General Rules for Malware Analysis

We'll finish this primer with several rules to keep in mind when performing analysis.

First, don't get too caught up in the details. Most malware programs are large and complex, and you can't possibly understand every detail. Focus instead on the key features. When you run into difficult and complex sections, try to get a general overview before you get stuck in the weeds.

Second, remember that different tools and approaches are available for different jobs. There is no one approach. Every situation is different, and the various tools and techniques that you'll learn will have similar and sometimes overlapping functionality. If you're not having luck with one tool, try another. If you get stuck, don't spend too long on any one issue; move on to something else. Try analyzing the malware from a different angle, or just try a different approach.

Finally, remember that malware analysis is like a cat-and-mouse game. As new malware analysis techniques are developed, malware authors respond with new techniques to thwart analysis. To succeed as a malware analyst, you must be able to recognize, understand, and defeat these techniques, and respond to changes in the art of malware analysis.

PART 1

BASIC ANALYSIS

1

BASIC STATIC TECHNIQUES

We begin our exploration of malware analysis with static analysis, which is usually the first step in studying malware. *Static analysis* describes the process of analyzing the code or structure of a program to determine its function. The program itself is not run at this time. In contrast, when performing *dynamic analysis*, the analyst actually runs the program, as you'll learn in Chapter 3.

This chapter discusses multiple ways to extract useful information from executables. In this chapter, we'll discuss the following techniques:

- Using antivirus tools to confirm maliciousness
- Using hashes to identify malware
- Gleaning information from a file's strings, functions, and headers

Each technique can provide different information, and the ones you use depend on your goals. Typically, you'll use several techniques to gather as much information as possible.

Antivirus Scanning: A Useful First Step

When first analyzing prospective malware, a good first step is to run it through multiple antivirus programs, which may already have identified it. But antivirus tools are certainly not perfect. They rely mainly on a database of identifiable pieces of known suspicious code (*file signatures*), as well as behavioral and pattern-matching analysis (*heuristics*) to identify suspect files. One problem is that malware writers can easily modify their code, thereby changing their program's signature and evading virus scanners. Also, rare malware often goes undetected by antivirus software because it's simply not in the database. Finally, heuristics, while often successful in identifying unknown malicious code, can be bypassed by new and unique malware.

Because the various antivirus programs use different signatures and heuristics, it's useful to run several different antivirus programs against the same piece of suspected malware. Websites such as VirusTotal (*http://www.virustotal.com/*) allow you to upload a file for scanning by multiple antivirus engines. VirusTotal generates a report that provides the total number of engines that marked the file as malicious, the malware name, and, if available, additional information about the malware.

Hashing: A Fingerprint for Malware

Hashing is a common method used to uniquely identify malware. The malicious software is run through a hashing program that produces a unique *hash* that identifies that malware (a sort of fingerprint). The Message-Digest Algorithm 5 (MD5) hash function is the one most commonly used for malware analysis, though the Secure Hash Algorithm 1 (SHA-1) is also popular.

For example, using the freely available md5deep program to calculate the hash of the Solitaire program that comes with Windows would generate the following output:

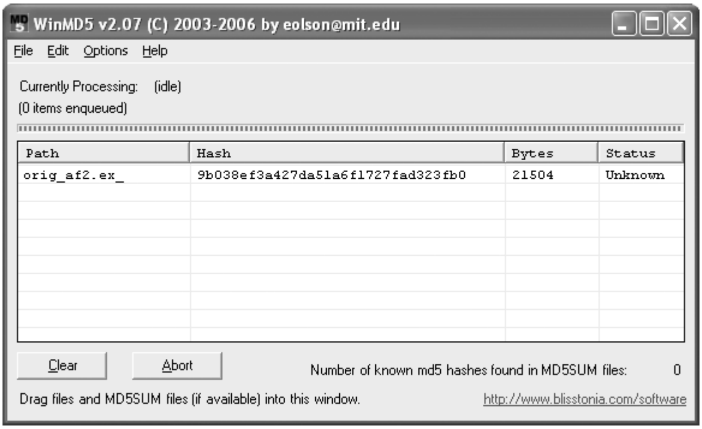
```
C:\>md5deep c:\WINDOWS\system32\sol.exe
373e7a863a1a345c60edb9e20ec32311  c:\WINDOWS\system32\sol.exe
```

The hash is **373e7a863a1a345c60edb9e20ec32311**.

The GUI-based WinMD5 calculator, shown in Figure 1-1, can calculate and display hashes for several files at a time.

Once you have a unique hash for a piece of malware, you can use it as follows:

- Use the hash as a label.
- Share that hash with other analysts to help them to identify malware.
- Search for that hash online to see if the file has already been identified.



Currently Processing: (idle)
(0 items enqueued)

Path	Hash	Bytes	Status
orig_af2.ex_	9b038ef3a427da51a6f1727fad323fb0	21504	Unknown

Number of known md5 hashes found in MD5SUM files: 0

Drag files and MD5SUM files (if available) into this window.

http://www.blisstonia.com/software

Figure 1-1: Output of WinMD5

Finding Strings

A *string* in a program is a sequence of characters such as "the." A program contains strings if it prints a message, connects to a URL, or copies a file to a specific location.

Searching through the strings can be a simple way to get hints about the functionality of a program. For example, if the program accesses a URL, then you will see the URL accessed stored as a string in the program. You can use the Strings program (*http://bit.ly/ic4plL*), to search an executable for strings, which are typically stored in either ASCII or Unicode format.

NOTE *Microsoft uses the term* wide character string *to describe its implementation of Unicode strings, which varies slightly from the Unicode standards. Throughout this book, when we refer to Unicode, we are referring to the Microsoft implementation.*

Both ASCII and Unicode formats store characters in sequences that end with a *NULL terminator* to indicate that the string is complete. ASCII strings use 1 byte per character, and Unicode uses 2 bytes per character.

Figure 1-2 shows the string BAD stored as ASCII. The ASCII string is stored as the bytes 0x42, 0x41, 0x44, and 0x00, where 0x42 is the ASCII representation of a capital letter *B*, 0x41 represents the letter *A*, and so on. The 0x00 at the end is the NULL terminator.

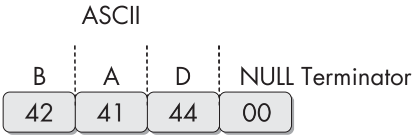


Figure 1-2: ASCII representation of the string BAD

Figure 1-3 shows the string BAD stored as Unicode. The Unicode string is stored as the bytes 0x42, 0x00, 0x41, and so on. A capital *B* is represented by the bytes 0x42 and 0x00, and the NULL terminator is two 0x00 bytes in a row.

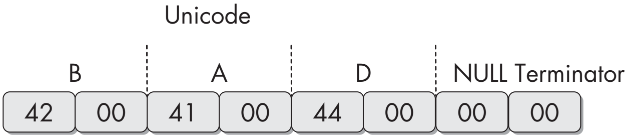


Figure 1-3: Unicode representation of the string BAD

When Strings searches an executable for ASCII and Unicode strings, it ignores context and formatting, so that it can analyze any file type and detect strings across an entire file (though this also means that it may identify bytes of characters as strings when they are not). Strings searches for a three-letter or greater sequence of ASCII and Unicode characters, followed by a string termination character.

Sometimes the strings detected by the Strings program are not actual strings. For example, if Strings finds the sequence of bytes 0x56, 0x50, 0x33, 0x00, it will interpret that as the string `VP3`. But those bytes may not actually represent that string; they could be a memory address, CPU instructions, or data used by the program. Strings leaves it up to the user to filter out the invalid strings.

Fortunately, most invalid strings are obvious, because they do not represent legitimate text. For example, the following excerpt shows the result of running Strings against the file *bp6.ex_*:

```
C:>strings bp6.ex_
VP3
VW3
t$@
D$4
99.124.22.1 ❹
e-@
GetLayout ❶
GDI32.DLL ❸
SetLayout ❷
M}C
Mail system DLL is invalid.!Send Mail failed to send message. ❺
```

In this example, the bold strings can be ignored. Typically, if a string is short and doesn't correspond to words, it's probably meaningless.

On the other hand, the strings `GetLayout` at ❶ and `SetLayout` at ❷ are Windows functions used by the Windows graphics library. We can easily identify these as meaningful strings because Windows function names normally begin with a capital letter and subsequent words also begin with a capital letter.

`GDI32.DLL` at ❸ is meaningful because it's the name of a common Windows *dynamic link library (DLL)* used by graphics programs. (DLL files contain executable code that is shared among multiple applications.)

As you might imagine, the number `99.124.22.1` at ❹ is an IP address—most likely one that the malware will use in some fashion.

Finally, at ❺, `Mail system DLL is invalid.!Send Mail failed to send message.` is an error message. Often, the most useful information obtained by running Strings is found in error messages. This particular message reveals two things: The subject malware sends messages (probably through email), and it depends on a mail system DLL. This information suggests that we might want to check email logs for suspicious traffic, and that another DLL (`Mail system DLL`) might be associated with this particular malware. Note that the missing DLL itself is not necessarily malicious; malware often uses legitimate libraries and DLLs to further its goals.

Packed and Obfuscated Malware

Malware writers often use packing or obfuscation to make their files more difficult to detect or analyze. *Obfuscated* programs are ones whose execution the malware author has attempted to hide. *Packed* programs are a subset of obfuscated programs in which the malicious program is compressed and cannot be analyzed. Both techniques will severely limit your attempts to statically analyze the malware.

Legitimate programs almost always include many strings. Malware that is packed or obfuscated contains very few strings. If upon searching a program with Strings, you find that it has only a few strings, it is probably either obfuscated or packed, suggesting that it may be malicious. You'll likely need to throw more than static analysis at it in order to investigate further.

NOTE *Packed and obfuscated code will often include at least the functions* `LoadLibrary` *and* `GetProcAddress`*, which are used to load and gain access to additional functions.*

Packing Files

When the packed program is run, a small wrapper program also runs to decompress the packed file and then run the unpacked file, as shown in Figure 1-4. When a packed program is analyzed statically, only the small wrapper program can be dissected. (Chapter 18 discusses packing and unpacking in more detail.)

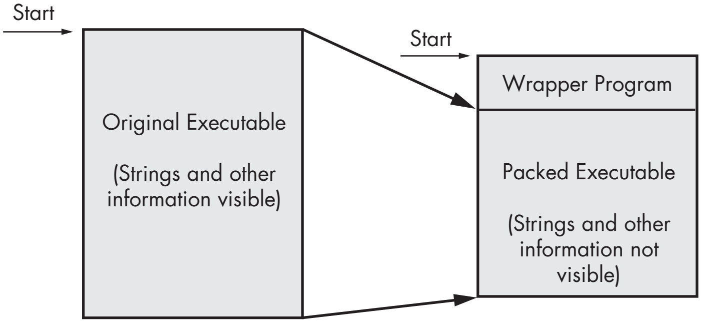
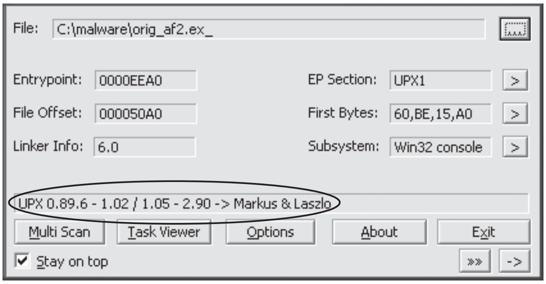

Start
Original Executable
(Strings and other information visible)
Start
Wrapper Program
Packed Executable
(Strings and other information not visible)

Figure 1-4: The file on the left is the original executable, with all strings, imports, and other information visible. On the right is a packed executable. All of the packed file's strings, imports, and other information are compressed and invisible to most static analysis tools.

Detecting Packers with PEiD

One way to detect packed files is with the PEiD program. You can use PEiD to detect the type of packer or compiler employed to build an application, which makes analyzing the packed file much easier. Figure 1-5 shows information about the *orig_af2.ex_* file as reported by PEiD.



File: C:\malware\orig_af2.ex_

Entrypoint: 0000EEA0

EP Section: UPX1

File Offset: 000050A0

First Bytes: 60,BE,15,A0

Linker Info: 6.0

Subsystem: Win32 console

UPX 0.89.6 - 1.02 / 1.05 - 2.90 -> Markus & Laszlo

☑ Stay on top

Figure 1-5: The PEiD program

NOTE *Development and support for PEiD has been discontinued since April 2011, but it's still the best tool available for packer and compiler detection. In many cases, it will also identify which packer was used to pack the file.*

As you can see, PEiD has identified the file as being packed with UPX version 0.89.6-1.02 or 1.05-2.90. (Just ignore the other information shown here for now. We'll examine this program in more detail in Chapter 18.)

When a program is packed, you must unpack it in order to be able to perform any analysis. The unpacking process is often complex and is covered in detail in Chapter 18, but the UPX packing program is so popular and easy to use for unpacking that it deserves special mention here. For example, to unpack malware packed with UPX, you would simply download UPX (*http://upx.sourceforge.net/*) and run it like so, using the packed program as input:

```
upx -d PackedProgram.exe
```

NOTE *Many PEiD plug-ins will run the malware executable without warning! (See Chapter 2 to learn how to set up a safe environment for running malware.) Also, like all programs, especially those used for malware analysis, PEiD can be subject to vulnerabilities. For example, PEiD version 0.92 contained a buffer overflow that allowed an attacker to execute arbitrary code. This would have allowed a clever malware writer to write a program to exploit the malware analyst's machine. Be sure to use the latest version of PEiD.*

Portable Executable File Format

So far, we have discussed tools that scan executables without regard to their format. However, the format of a file can reveal a lot about the program's functionality.

The Portable Executable (PE) file format is used by Windows executables, object code, and DLLs. The PE file format is a data structure that contains the information necessary for the Windows OS loader to manage the wrapped executable code. Nearly every file with executable code that is loaded by Windows is in the PE file format, though some legacy file formats do appear on rare occasion in malware.

PE files begin with a header that includes information about the code, the type of application, required library functions, and space requirements. The information in the PE header is of great value to the malware analyst.

Linked Libraries and Functions

One of the most useful pieces of information that we can gather about an executable is the list of functions that it imports. *Imports* are functions used by one program that are actually stored in a different program, such as code libraries that contain functionality common to many programs. Code libraries can be connected to the main executable by *linking*.

Programmers link imports to their programs so that they don't need to re-implement certain functionality in multiple programs. Code libraries can be linked statically, at runtime, or dynamically. Knowing how the library code is linked is critical to our understanding of malware because the information we can find in the PE file header depends on how the library code has been linked. We'll discuss several tools for viewing an executable's imported functions in this section.

Static, Runtime, and Dynamic Linking

Static linking is the least commonly used method of linking libraries, although it is common in UNIX and Linux programs. When a library is statically linked to an executable, all code from that library is copied into the executable, which makes the executable grow in size. When analyzing code, it's difficult to differentiate between statically linked code and the executable's own code, because nothing in the PE file header indicates that the file contains linked code.

While unpopular in friendly programs, *runtime linking* is commonly used in malware, especially when it's packed or obfuscated. Executables that use runtime linking connect to libraries only when that function is needed, not at program start, as with dynamically linked programs.

Several Microsoft Windows functions allow programmers to import linked functions not listed in a program's file header. Of these, the two most commonly used are `LoadLibrary` and `GetProcAddress`. `LdrGetProcAddress` and `LdrLoadDll` are also used. `LoadLibrary` and `GetProcAddress` allow a program to access any function in any library on the system, which means that when these functions are used, you can't tell statically which functions are being linked to by the suspect program.

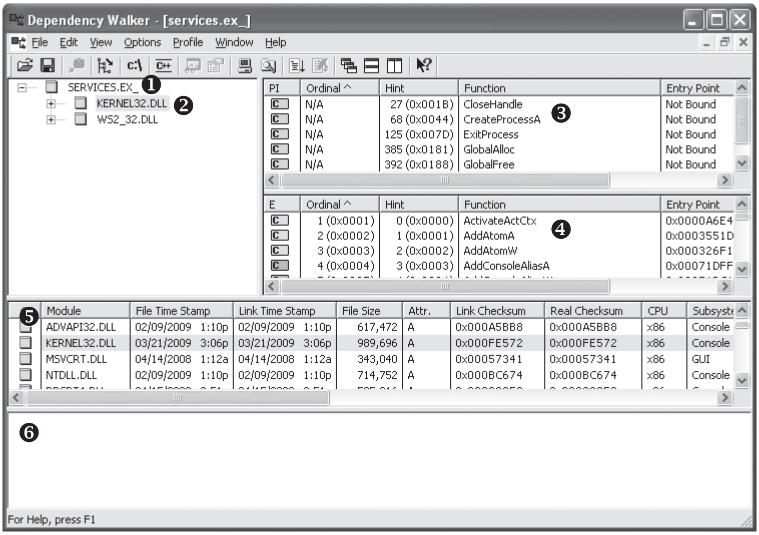
Of all linking methods, *dynamic linking* is the most common and the most interesting for malware analysts. When libraries are dynamically linked, the host OS searches for the necessary libraries when the program is loaded. When the program calls the linked library function, that function executes within the library.

The PE file header stores information about every library that will be loaded and every function that will be used by the program. The libraries used and functions called are often the most important parts of a program, and identifying them is particularly important, because it allows us to guess at what the program does. For example, if a program imports the function `URLDownloadToFile`, you might guess that it connects to the Internet to download some content that it then stores in a local file.

Exploring Dynamically Linked Functions with Dependency Walker

The Dependency Walker program (*http://www.dependencywalker.com/*), distributed with some versions of Microsoft Visual Studio and other Microsoft development packages, lists only dynamically linked functions in an executable.

Figure 1-6 shows the Dependency Walker's analysis of *SERVICES.EX_* ❶. The far left pane at ❷ shows the program as well as the DLLs being imported, namely *KERNEL32.DLL* and *WS2_32.DLL*.


Dependency Walker - [services.ex_]
File Edit View Options Profile Window Help
SERVICES.EX_ ❶
KERNEL32.DLL ❷
WS2_32.DLL
❸
❹
❺
❻
For Help, press F1

PI	Ordinal ^	Hint	Function	Entry Point
C	N/A	27 (0x001B)	CloseHandle	Not Bound
C	N/A	68 (0x0044)	CreateProcessA	Not Bound
C	N/A	125 (0x007D)	ExitProcess	Not Bound
C	N/A	385 (0x0181)	GlobalAlloc	Not Bound
C	N/A	392 (0x0188)	GlobalFree	Not Bound

E	Ordinal ^	Hint	Function	Entry Point
C	1 (0x0001)	0 (0x0000)	ActivateActCtx	0x0000A6E4
C	2 (0x0002)	1 (0x0001)	AddAtomA	0x0003551D
C	3 (0x0003)	2 (0x0002)	AddAtomW	0x000326F1
C	4 (0x0004)	3 (0x0003)	AddConsoleAliasA	0x00071DFF

Module	File Time Stamp	Link Time Stamp	File Size	Attr.	Link Checksum	Real Checksum	CPU	Subsyst
ADVAPI32.DLL	02/09/2009 1:10p	02/09/2009 1:10p	617,472	A	0x000A5BB8	0x000A5BB8	x86	Console
KERNEL32.DLL	03/21/2009 3:06p	03/21/2009 3:06p	989,696	A	0x000FE572	0x000FE572	x86	Console
MSVCRT.DLL	04/14/2008 1:12a	04/14/2008 1:12a	343,040	A	0x00057341	0x00057341	x86	GUI
NTDLL.DLL	02/09/2009 1:10p	02/09/2009 1:10p	714,752	A	0x000BC674	0x000BC674	x86	Console

Figure 1-6: The Dependency Walker program

Clicking *KERNEL32.DLL* shows its imported functions in the upper-right pane at ❸. We see several functions, but the most interesting is `CreateProcessA`, which tells us that the program will probably create another process, and suggests that when running the program, we should watch for the launch of additional programs.

The middle right pane at ❹ lists all functions in *KERNEL32.DLL* that can be imported—information that is not particularly useful to us. Notice the column in panes ❸ and ❹ labeled Ordinal. Executables can import functions by ordinal instead of name. When importing a function by ordinal, the name of the function never appears in the original executable, and it can be harder for an analyst to figure out which function is being used. When malware imports a function by ordinal, you can find out which function is being imported by looking up the ordinal value in the pane at ❹.

The bottom two panes (❺ and ❻) list additional information about the versions of DLLs that would be loaded if you ran the program and any reported errors, respectively.

A program's DLLs can tell you a lot about its functionality. For example, Table 1-1 lists common DLLs and what they tell you about an application.

Table 1-1: Common DLLs

DLL	Description
Kernel32.dll	This is a very common DLL that contains core functionality, such as access and manipulation of memory, files, and hardware.
Advapi32.dll	This DLL provides access to advanced core Windows components such as the Service Manager and Registry.
User32.dll	This DLL contains all the user-interface components, such as buttons, scroll bars, and components for controlling and responding to user actions.
Gdi32.dll	This DLL contains functions for displaying and manipulating graphics.
Ntdll.dll	This DLL is the interface to the Windows kernel. Executables generally do not import this file directly, although it is always imported indirectly by *Kernel32.dll*. If an executable imports this file, it means that the author intended to use functionality not normally available to Windows programs. Some tasks, such as hiding functionality or manipulating processes, will use this interface.
WSock32.dll and *Ws2_32.dll*	These are networking DLLs. A program that accesses either of these most likely connects to a network or performs network-related tasks.
Wininet.dll	This DLL contains higher-level networking functions that implement protocols such as FTP, HTTP, and NTP.

FUNCTION NAMING CONVENTIONS

When evaluating unfamiliar Windows functions, a few naming conventions are worth noting because they come up often and might confuse you if you don't recognize them. For example, you will often encounter function names with an `Ex` suffix, such as `CreateWindowEx`. When Microsoft updates a function and the new function is incompatible with the old one, Microsoft continues to support the old function. The new function is given the same name as the old function, with an added `Ex` suffix. Functions that have been significantly updated twice have two `Ex` suffixes in their names.

Many functions that take strings as parameters include an `A` or a `W` at the end of their names, such as `CreateDirectoryW`. This letter does *not* appear in the documentation for the function; it simply indicates that the function accepts a string parameter and that there are two different versions of the function: one for ASCII strings and one for wide character strings. Remember to drop the trailing `A` or `W` when searching for the function in the Microsoft documentation.

Imported Functions

The PE file header also includes information about specific functions used by an executable. The names of these Windows functions can give you a good idea about what the executable does. Microsoft does an excellent job of documenting the Windows API through the Microsoft Developer Network (MSDN) library. (You'll also find a list of functions commonly used by malware in Appendix A.)

Exported Functions

Like imports, DLLs and EXEs export functions to interact with other programs and code. Typically, a DLL implements one or more functions and exports them for use by an executable that can then import and use them.

The PE file contains information about which functions a file exports. Because DLLs are specifically implemented to provide functionality used by EXEs, exported functions are most common in DLLs. EXEs are not designed to provide functionality for other EXEs, and exported functions are rare. If you discover exports in an executable, they often will provide useful information.

In many cases, software authors name their exported functions in a way that provides useful information. One common convention is to use the name used in the Microsoft documentation. For example, in order to run a program as a service, you must first define a `ServiceMain` function. The presence of an exported function called `ServiceMain` tells you that the malware runs as part of a service.

Unfortunately, while the Microsoft documentation calls this function `ServiceMain`, and it's common for programmers to do the same, the function can have any name. Therefore, the names of exported functions are actually of limited use against sophisticated malware. If malware uses exports, it will often either omit names entirely or use unclear or misleading names.

You can view export information using the Dependency Walker program discussed in "Exploring Dynamically Linked Functions with Dependency Walker" on page 16. For a list of exported functions, click the name of the file you want to examine. Referring back to Figure 1-6, window ❹ shows all of a file's exported functions.

Static Analysis in Practice

Now that you understand the basics of static analysis, let's examine some real malware. We'll look at a potential keylogger and then a packed program.

PotentialKeylogger.exe: An Unpacked Executable

Table 1-2 shows an abridged list of functions imported by *PotentialKeylogger.exe*, as collected using Dependency Walker. Because we see so many imports, we can immediately conclude that this file is not packed.

Table 1-2: An Abridged List of DLLs and Functions Imported from *PotentialKeylogger.exe*

Kernel32.dll	User32.dll	User32.dll (continued)
CreateDirectoryW	BeginDeferWindowPos	**ShowWindow**
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	**GDI32.dll**
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	**Shell32.dll**
HeapAlloc	LoadCursorW	CommandLineToArgvW
HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
OpenProcess	MapWindowPoints	ShellExecuteW
ReadFile	MessageBoxW	
SetFilePointer	**RegisterClassExW**	**Advapi32.dll**
WriteFile	**RegisterHotKey**	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	SetWindowTextW	RegQueryValueExW
	SetWindowsHookExW	RegSetValueExW

Like most average-sized programs, this executable contains a large number of imported functions. Unfortunately, only a small minority of those functions are particularly interesting for malware analysis. Throughout this book, we will cover the imports for malicious software, focusing on the most interesting functions from a malware analysis standpoint.

When you are not sure what a function does, you will need to look it up. To help guide your analysis, Appendix A lists many of the functions of greatest interest to malware analysts. If a function is not listed in Appendix A, search for it on MSDN online.

As a new analyst, you will spend time looking up many functions that aren't very interesting, but you'll quickly start to learn which functions could be important and which ones are not. For the purposes of this example, we will show you a large number of imports that are uninteresting, so you can become familiar with looking at a lot of data and focusing on some key nuggets of information.

Normally, we wouldn't know that this malware is a potential keylogger, and we would need to look for functions that provide the clues. We will be focusing on only the functions that provide hints to the functionality of the program.

The imports from *Kernel32.dll* in Table 1-2 tell us that this software can open and manipulate processes (such as `OpenProcess`, `GetCurrentProcess`, and `GetProcessHeap`) and files (such as `ReadFile`, `CreateFile`, and `WriteFile`). The functions `FindFirstFile` and `FindNextFile` are particularly interesting ones that we can use to search through directories.

The imports from *User32.dll* are even more interesting. The large number of GUI manipulation functions (such as `RegisterClassEx`, `SetWindowText`, and `ShowWindow`) indicates a high likelihood that this program has a GUI (though the GUI is not necessarily displayed to the user).

The function `SetWindowsHookEx` is commonly used in spyware and is the most popular way that keyloggers receive keyboard inputs. This function has some legitimate uses, but if you suspect malware and you see this function, you are probably looking at keylogging functionality.

The function `RegisterHotKey` is also interesting. It registers a hotkey (such as CTRL-SHIFT-P) so that whenever the user presses that hotkey combination, the application is notified. No matter which application is currently active, a hotkey will bring the user to this application.

The imports from *GDI32.dll* are graphics-related and simply confirm that the program probably has a GUI. The imports from *Shell32.dll* tell us that this program can launch other programs—a feature common to both malware and legitimate programs.

The imports from *Advapi32.dll* tell us that this program uses the registry, which in turn tells us that we should search for strings that look like registry keys. Registry strings look a lot like directories. In this case, we found the string `Software\Microsoft\Windows\CurrentVersion\Run`, which is a registry key (commonly used by malware) that controls which programs are automatically run when Windows starts up.

This executable also has several exports: `LowLevelKeyboardProc` and `LowLevelMouseProc`. Microsoft's documentation says, "The `LowLevelKeyboardProc` hook procedure is an application-defined or library-defined callback function used with the `SetWindowsHookEx` function." In other words, this function is used with `SetWindowsHookEx` to specify which function will be called when a specified event occurs—in this case, the low-level keyboard event. The documentation for `SetWindowsHookEx` further explains that this function will be called when certain low-level keyboard events occur.

The Microsoft documentation uses the name `LowLevelKeyboardProc`, and the programmer in this case did as well. We were able to get valuable information because the programmer didn't obscure the name of an export.

Using the information gleaned from a static analysis of these imports and exports, we can draw some significant conclusions or formulate some hypotheses about this malware. For one, it seems likely that this is a local keylogger that uses `SetWindowsHookEx` to record keystrokes. We can also surmise that it has a GUI that is displayed only to a specific user, and that the hotkey registered with `RegisterHotKey` specifies the hotkey that the malicious user enters to see the keylogger GUI and access recorded keystrokes. We can further speculate from the registry function and the existence of `Software\Microsoft\Windows\CurrentVersion\Run` that this program sets itself to load at system startup.

PackedProgram.exe: A Dead End

Table 1-3 shows a complete list of the functions imported by a second piece of unknown malware. The brevity of this list tells us that this program is packed or obfuscated, which is further confirmed by the fact that this program has no readable strings. A Windows compiler would not create a program that imports such a small number of functions; even a Hello, World program would have more.

Table 1-3: DLLs and Functions Imported from *PackedProgram.exe*

Kernel32.dll	User32.dll
`GetModuleHandleA`	`MessageBoxA`
`LoadLibraryA`	
`GetProcAddress`	
`ExitProcess`	
`VirtualAlloc`	
`VirtualFree`	

The fact that this program is packed is a valuable piece of information, but its packed nature also prevents us from learning anything more about the program using basic static analysis. We'll need to try more advanced analysis techniques such as dynamic analysis (covered in Chapter 3) or unpacking (covered in Chapter 18).

The PE File Headers and Sections

PE file headers can provide considerably more information than just imports. The PE file format contains a header followed by a series of sections. The header contains metadata about the file itself. Following the header are the actual sections of the file, each of which contains useful information. As we progress through the book, we will continue to discuss strategies for viewing the information in each of these sections. The following are the most common and interesting sections in a PE file:

`.text` The `.text` section contains the instructions that the CPU executes. All other sections store data and supporting information. Generally, this is the only section that can execute, and it should be the only section that includes code.

`.rdata` The `.rdata` section typically contains the import and export information, which is the same information available from both Dependency

Walker and PEview. This section can also store other read-only data used by the program. Sometimes a file will contain an `.idata` and `.edata` section, which store the import and export information (see Table 1-4).

`.data` The `.data` section contains the program's global data, which is accessible from anywhere in the program. Local data is not stored in this section, or anywhere else in the PE file. (We address this topic in Chapter 6.)

`.rsrc` The `.rsrc` section includes the resources used by the executable that are not considered part of the executable, such as icons, images, menus, and strings. Strings can be stored either in the `.rsrc` section or in the main program, but they are often stored in the `.rsrc` section for multilanguage support.

Section names are often consistent across a compiler, but can vary across different compilers. For example, Visual Studio uses `.text` for executable code, but Borland Delphi uses `CODE`. Windows doesn't care about the actual name since it uses other information in the PE header to determine how a section is used. Furthermore, the section names are sometimes obfuscated to make analysis more difficult. Luckily, the default names are used most of the time. Table 1-4 lists the most common you'll encounter.

Table 1-4: Sections of a PE File for a Windows Executable

Executable	Description
`.text`	Contains the executable code
`.rdata`	Holds read-only data that is globally accessible within the program
`.data`	Stores global data accessed throughout the program
`.idata`	Sometimes present and stores the import function information; if this section is not present, the import function information is stored in the `.rdata` section
`.edata`	Sometimes present and stores the export function information; if this section is not present, the export function information is stored in the `.rdata` section
`.pdata`	Present only in 64-bit executables and stores exception-handling information
`.rsrc`	Stores resources needed by the executable
`.reloc`	Contains information for relocation of library files

Examining PE Files with PEview

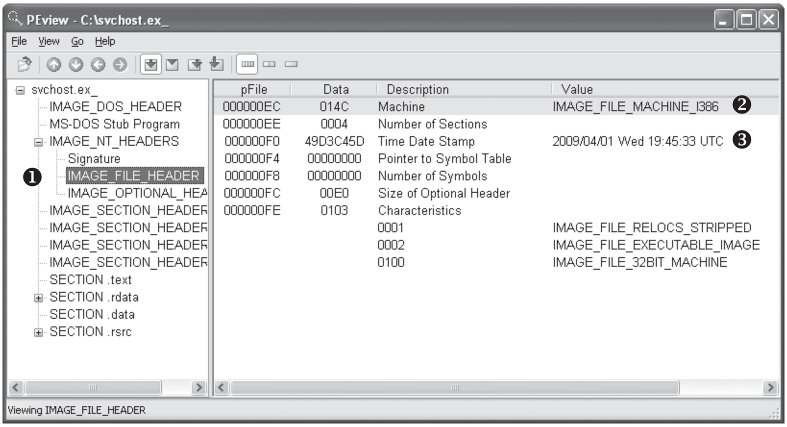
The PE file format stores interesting information within its header. We can use the PEview tool to browse through the information, as shown in Figure 1-7.

In the figure, the left pane at ❶ displays the main parts of a PE header. The `IMAGE_FILE_HEADER` entry is highlighted because it is currently selected.

The first two parts of the PE header—the `IMAGE_DOS_HEADER` and MS-DOS Stub Program—are historical and offer no information of particular interest to us.

The next section of the PE header, `IMAGE_NT_HEADERS`, shows the NT headers. The signature is always the same and can be ignored.

The `IMAGE_FILE_HEADER` entry, highlighted and displayed in the right panel at ❷, contains basic information about the file. The Time Date Stamp description at ❸ tells us when this executable was compiled, which can be very useful in malware analysis and incident response. For example, an old compile time suggests that this is an older attack, and antivirus programs might contain signatures for the malware. A new compile time suggests the reverse.



pFile	Data	Description	Value
000000EC	014C	Machine	IMAGE_FILE_MACHINE_I386 ❷
000000EE	0004	Number of Sections	
000000F0	49D3C45D	Time Date Stamp	2009/04/01 Wed 19:45:33 UTC ❸
000000F4	00000000	Pointer to Symbol Table	
000000F8	00000000	Number of Symbols	
000000FC	00E0	Size of Optional Header	
000000FE	0103	Characteristics	
		0001	IMAGE_FILE_RELOCS_STRIPPED
		0002	IMAGE_FILE_EXECUTABLE_IMAGE
		0100	IMAGE_FILE_32BIT_MACHINE

Figure 1-7: Viewing the `IMAGE_FILE_HEADER` *in the PEview program*

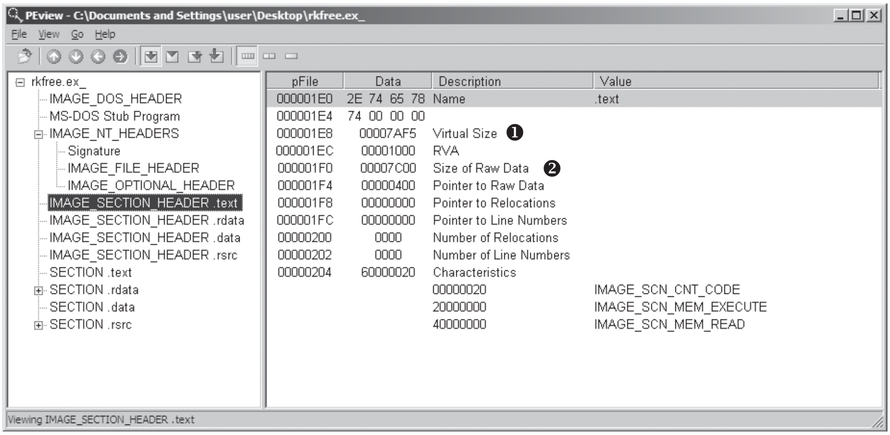
That said, the compile time is a bit problematic. All Delphi programs use a compile time of June 19, 1992. If you see that compile time, you're probably looking at a Delphi program, and you won't really know when it was compiled. In addition, a competent malware writer can easily fake the compile time. If you see a compile time that makes no sense, it probably was faked.

The `IMAGE_OPTIONAL_HEADER` section includes several important pieces of information. The Subsystem description indicates whether this is a console or GUI program. Console programs have the value `IMAGE_SUBSYSTEM_WINDOWS_CUI` and run inside a command window. GUI programs have the value `IMAGE_SUBSYSTEM_WINDOWS_GUI` and run within the Windows system. Less common subsystems such as Native or Xbox also are used.

The most interesting information comes from the section headers, which are in `IMAGE_SECTION_HEADER`, as shown in Figure 1-8. These headers are used to describe each section of a PE file. The compiler generally creates and names the sections of an executable, and the user has little control over these names. As a result, the sections are usually consistent from executable to executable (see Table 1-4), and any deviations may be suspicious.

For example, in Figure 1-8, Virtual Size at ❶ tells us how much space is allocated for a section during the loading process. The Size of Raw Data at ❷ shows how big the section is on disk. These two values should usually be equal, because data should take up just as much space on the disk as it does in memory. Small differences are normal, and are due to differences between alignment in memory and on disk.

The section sizes can be useful in detecting packed executables. For example, if the Virtual Size is much larger than the Size of Raw Data, you know that the section takes up more space in memory than it does on disk. This is often indicative of packed code, particularly if the `.text` section is larger in memory than on disk.



```
⊟ rkfree.ex_
   IMAGE_DOS_HEADER
   MS-DOS Stub Program
 ⊟ IMAGE_NT_HEADERS
     Signature
     IMAGE_FILE_HEADER
     IMAGE_OPTIONAL_HEADER
   IMAGE_SECTION_HEADER .text
   IMAGE_SECTION_HEADER .rdata
   IMAGE_SECTION_HEADER .data
   IMAGE_SECTION_HEADER .rsrc
   SECTION .text
 ⊞ SECTION .rdata
   SECTION .data
 ⊞ SECTION .rsrc
```

pFile	Data	Description	Value
000001E0	2E 74 65 78	Name	.text
000001E4	74 00 00 00		
000001E8	00007AF5	Virtual Size ❶	
000001EC	00001000	RVA	
000001F0	00007C00	Size of Raw Data ❷	
000001F4	00000400	Pointer to Raw Data	
000001F8	00000000	Pointer to Relocations	
000001FC	00000000	Pointer to Line Numbers	
00000200	0000	Number of Relocations	
00000202	0000	Number of Line Numbers	
00000204	60000020	Characteristics	
		00000020	IMAGE_SCN_CNT_CODE
		20000000	IMAGE_SCN_MEM_EXECUTE
		40000000	IMAGE_SCN_MEM_READ

Viewing IMAGE_SECTION_HEADER .text

Figure 1-8: Viewing the `IMAGE_SECTION_HEADER .text` *section in the PEview program*

Table 1-5 shows the sections from *PotentialKeylogger.exe*. As you can see, the `.text`, `.rdata`, and `.rsrc` sections each has a Virtual Size and Size of Raw Data value of about the same size. The `.data` section may seem suspicious because it has a much larger virtual size than raw data size, but this is normal for the `.data` section in Windows programs. But note that this information alone does not tell us that the program is not malicious; it simply shows that it is likely not packed and that the PE file header was generated by a compiler.

Table 1-5: Section Information for *PotentialKeylogger.exe*

Section	Virtual size	Size of raw data
`.text`	7AF5	7C00
`.data`	17A0	0200
`.rdata`	1AF5	1C00
`.rsrc`	72B8	7400

Table 1-6 shows the sections from *PackedProgram.exe*. The sections in this file have a number of anomalies: The sections named `Dijfpds`, `.sdfuok`, and `Kijijl` are unusual, and the `.text`, `.data`, and `.rdata` sections are suspicious. The `.text` section has a Size of Raw Data value of 0, meaning that it takes up no space on disk, and its Virtual Size value is A000, which means that space will be allocated for the `.text` segment. This tells us that a packer will unpack the executable code to the allocated `.text` section.

Table 1-6: Section Information for *PackedProgram.exe*

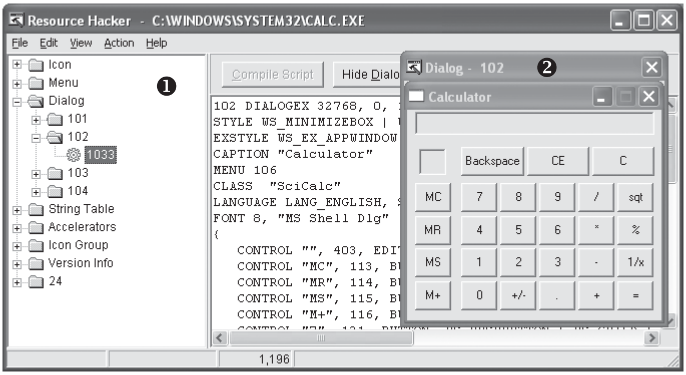
Name	Virtual size	Size of raw data
`.text`	A000	0000
`.data`	3000	0000
`.rdata`	4000	0000
`.rsrc`	19000	3400

Table 1-6: Section Information for *PackedProgram.exe* (continued)

Name	Virtual size	Size of raw data
`Dijfpds`	20000	0000
`.sdfuok`	34000	3313F
`Kijijl`	1000	0200

Viewing the Resource Section with Resource Hacker

Now that we're finished looking at the header for the PE file, we can look at some of the sections. The only section we can examine without additional knowledge from later chapters is the resource section. You can use the free Resource Hacker tool found at *http://www.angusj.com/* to browse the `.rsrc` section. When you click through the items in Resource Hacker, you'll see the strings, icons, and menus. The menus displayed are identical to what the program uses. Figure 1-9 shows the Resource Hacker display for the Windows Calculator program, *calc.exe*.



```
102 DIALOGEX 32768, 0,
STYLE WS_MINIMIZEBOX |
EXSTYLE WS_EX_APPWINDOW
CAPTION "Calculator"
MENU 106
CLASS  "SciCalc"
LANGUAGE LANG_ENGLISH,
FONT 8, "MS Shell Dlg"
{
   CONTROL "", 403, EDIT
   CONTROL "MC", 113, B
   CONTROL "MR", 114, B
   CONTROL "MS", 115, B
   CONTROL "M+", 116, B
```

Figure 1-9: The Resource Hacker tool display for calc.exe

The panel on the left shows all resources included in this executable. Each root folder shown in the left pane at ❶ stores a different type of resource. The informative sections for malware analysis include:

- The Icon section lists images shown when the executable is in a file listing.
- The Menu section stores all menus that appear in various windows, such as the File, Edit, and View menus. This section contains the names of all the menus, as well as the text shown for each. The names should give you a good idea of their functionality.
- The Dialog section contains the program's dialog menus. The dialog at ❷ shows what the user will see when running *calc.exe*. If we knew nothing else about *calc.exe*, we could identify it as a calculator program simply by looking at this dialog menu.
- The String Table section stores strings.
- The Version Info section contains a version number and often the company name and a copyright statement.

The .rsrc section shown in Figure 1-9 is typical of Windows applications and can include whatever a programmer requires.

NOTE *Malware, and occasionally legitimate software, often store an embedded program or driver here and, before the program runs, they extract the embedded executable or driver. Resource Hacker lets you extract these files for individual analysis.*

Using Other PE File Tools

Many other tools are available for browsing a PE header. Two of the most useful tools are PEBrowse Professional and PE Explorer.

PEBrowse Professional (*http://www.smidgeonsoft.prohosting.com/pebrowse-pro-file-viewer.html*) is similar to PEview. It allows you to look at the bytes from each section and shows the parsed data. PEBrowse Professional does the better job of presenting information from the resource (.rsrc) section.

PE Explorer (*http://www.heaventools.com/*) has a rich GUI that allows you to navigate through the various parts of the PE file. You can edit certain parts of the PE file, and its included resource editor is great for browsing and editing the file's resources. The tool's main drawback is that it is not free.

PE Header Summary

The PE header contains useful information for the malware analyst, and we will continue to examine it in subsequent chapters. Table 1-7 reviews the key information that can be obtained from a PE header.

Table 1-7: Information in the PE Header

Field	Information revealed
Imports	Functions from other libraries that are used by the malware
Exports	Functions in the malware that are meant to be called by other programs or libraries
Time Date Stamp	Time when the program was compiled
Sections	Names of sections in the file and their sizes on disk and in memory
Subsystem	Indicates whether the program is a command-line or GUI application
Resources	Strings, icons, menus, and other information included in the file

Conclusion

Using a suite of relatively simple tools, we can perform static analysis on malware to gain a certain amount of insight into its function. But static analysis is typically only the first step, and further analysis is usually necessary. The next step is setting up a safe environment so you can run the malware and perform basic dynamic analysis, as you'll see in the next two chapters.

LABS

The purpose of the labs is to give you an opportunity to practice the skills taught in the chapter. In order to simulate realistic malware analysis you will be given little or no information about the program you are analyzing. Like all of the labs throughout this book, the basic static analysis lab files have been given generic names to simulate unknown malware, which typically use meaningless or misleading names.

Each of the labs consists of a malicious file, a few questions, short answers to the questions, and a detailed analysis of the malware. The solutions to the labs are included in Appendix C.

The labs include two sections of answers. The first section consists of short answers, which should be used if you did the lab yourself and just want to check your work. The second section includes detailed explanations for you to follow along with our solution and learn how we found the answers to the questions posed in each lab.

Lab 1-1

This lab uses the files *Lab01-01.exe* and *Lab01-01.dll*. Use the tools and techniques described in the chapter to gain information about the files and answer the questions below.

Questions

1. Upload the files to *http://www.VirusTotal.com/* and view the reports. Does either file match any existing antivirus signatures?
2. When were these files compiled?
3. Are there any indications that either of these files is packed or obfuscated? If so, what are these indicators?
4. Do any imports hint at what this malware does? If so, which imports are they?
5. Are there any other files or host-based indicators that you could look for on infected systems?
6. What network-based indicators could be used to find this malware on infected machines?
7. What would you guess is the purpose of these files?

Lab 1-2

Analyze the file *Lab01-02.exe*.

Questions

1. Upload the *Lab01-02.exe* file to *http://www.VirusTotal.com/*. Does it match any existing antivirus definitions?
2. Are there any indications that this file is packed or obfuscated? If so, what are these indicators? If the file is packed, unpack it if possible.
3. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?
4. What host- or network-based indicators could be used to identify this malware on infected machines?

Lab 1-3

Analyze the file *Lab01-03.exe*.

Questions

1. Upload the *Lab01-03.exe* file to *http://www.VirusTotal.com/*. Does it match any existing antivirus definitions?
2. Are there any indications that this file is packed or obfuscated? If so, what are these indicators? If the file is packed, unpack it if possible.
3. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?
4. What host- or network-based indicators could be used to identify this malware on infected machines?

Lab 1-4

Analyze the file *Lab01-04.exe*.

Questions

1. Upload the *Lab01-04.exe* file to *http://www.VirusTotal.com/*. Does it match any existing antivirus definitions?
2. Are there any indications that this file is packed or obfuscated? If so, what are these indicators? If the file is packed, unpack it if possible.
3. When was this program compiled?
4. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?
5. What host- or network-based indicators could be used to identify this malware on infected machines?
6. This file has one resource in the resource section. Use Resource Hacker to examine that resource, and then use it to extract the resource. What can you learn from the resource?

2

MALWARE ANALYSIS IN VIRTUAL MACHINES

Before you can run malware to perform dynamic analysis, you must set up a safe environment. Fresh malware can be full of surprises, and if you run it on a production machine, it can quickly spread to other machines on the network and be very difficult to remove. A safe environment will allow you to investigate the malware without exposing your machine or other machines on the network to unexpected and unnecessary risk.

You can use dedicated physical or virtual machines to study malware safely. Malware can be analyzed using individual physical machines on *air-gapped networks*. These are isolated networks with machines that are disconnected from the Internet or any other networks to prevent the malware from spreading.

Air-gapped networks allow you to run malware in a real environment without putting other computers at risk. One disadvantage of this test scenario, however, is the lack of an Internet connection. Many pieces of malware depend on a live Internet connection for updates, command and control, and other features.

Another disadvantage to analyzing malware on physical rather than virtual machines is that malware can be difficult to remove. To avoid problems, most people who test malware on physical machines use a tool such as Norton Ghost to manage backup images of their operating systems (OSs), which they restore on their machines after they've completed their analysis.

The main advantage to using physical machines for malware analysis is that malware can sometimes execute differently on virtual machines. As you're analyzing malware on a virtual machine, some malware can detect that it's being run in a virtual machine, and it will behave differently to thwart analysis.

Because of the risks and disadvantages that come with using physical machines to analyze malware, virtual machines are most commonly used for dynamic analysis. In this chapter, we'll focus on using virtual machines for malware analysis.

The Structure of a Virtual Machine

Virtual machines are like a computer inside a computer, as illustrated in Figure 2-1. A guest OS is installed within the host OS on a virtual machine, and the OS running in the virtual machine is kept isolated from the host OS. Malware running on a virtual machine cannot harm the host OS. And if the malware damages the virtual machine, you can simply reinstall the OS in the virtual machine or return the virtual machine to a clean state.

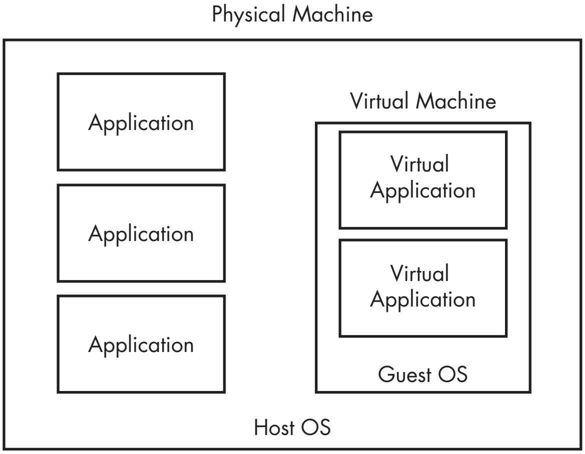


Figure 2-1: Traditional applications run as shown in the left column. The guest OS is contained entirely within the virtual machine, and the virtual applications are contained within the guest OS.

VMware offers a popular series of desktop virtualization products that can be used for analyzing malware on virtual machines. VMware Player is free and can be used to create and run virtual machines, but it lacks some features necessary for effective malware analysis. VMware Workstation costs a little under $200 and is generally the better choice for malware analysis. It includes features such as snapshotting, which allows you to save the current state of a virtual machine, and the ability to clone or copy an existing virtual machine.

There are many alternatives to VMware, such as Parallels, Microsoft Virtual PC, Microsoft Hyper-V, and Xen. These vary in host and guest OS support and features. This book will focus on using VMware for virtualization, but if you prefer another virtualization tool, you should still find this discussion relevant.

Creating Your Malware Analysis Machine

Of course, before you can use a virtual machine for malware analysis, you need to create one. This book is not specifically about virtualization, so we won't walk you through all of the details. When presented with options, your best bet, unless you know that you have different requirements, is to choose the default hardware configurations. Choose the hard drive size based on your needs.

VMware uses disk space intelligently and will resize its virtual disk dynamically based on your need for storage. For example, if you create a 20GB hard drive but store only 4GB of data on it, VMware will shrink the size of the virtual hard drive accordingly. A virtual drive size of 20GB is typically a good beginning. That amount should be enough to store the guest OS and any tools that you might need for malware analysis. VMware will make a lot of choices for you and, in most cases, these choices will do the job.

Next, you'll install your OS and applications. Most malware and malware analysis tools run on Windows, so you will likely install Windows as your virtual OS. As of this writing, Windows XP is still the most popular OS (surprisingly) and the target for most malware. We'll focus our explorations on Windows XP.

After you've installed the OS, you can install any required applications. You can always install applications later, but it is usually easier if you set up everything at once. Appendix B has a list of useful applications for malware analysis.

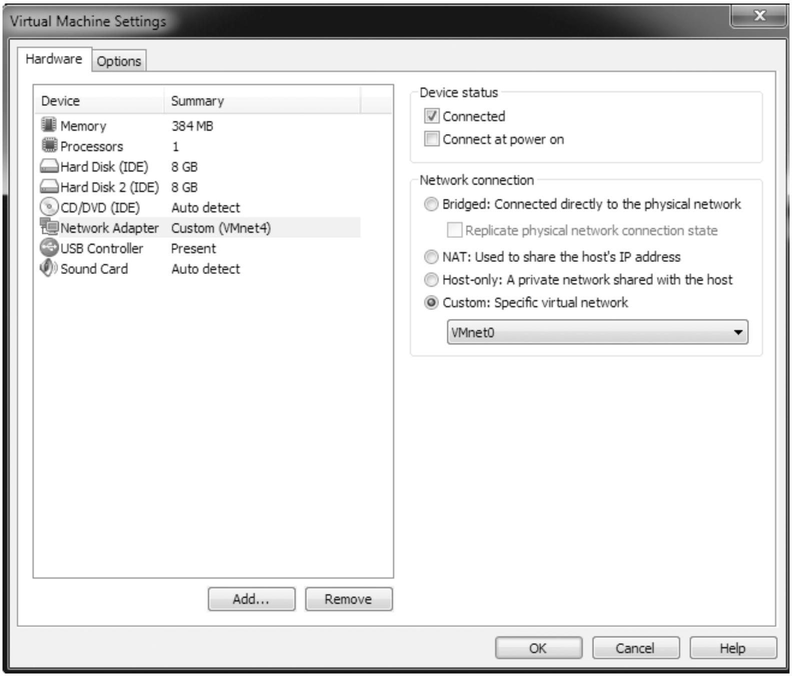
Next, you'll install VMware Tools. From the VMware menu, select **VM ▸ Install VMware Tools** to begin the installation. VMware Tools improves the user experience by making the mouse and keyboard more responsive. It also allows access to shared folders, drag-and-drop file transfer, and various other useful features we'll discuss in this chapter.

After you've installed VMware, it's time for some configuration.

Configuring VMware

Most malware includes network functionality. For example, a worm will perform network attacks against other machines in an effort to spread itself. But you would not want to allow a worm access to your own network, because it could to spread to other computers.

When analyzing malware, you will probably want to observe the malware's network activity to help you understand the author's intention, to create signatures, or to exercise the program fully. VMware offers several networking options for virtual networking, as shown in Figure 2-2 and discussed in the following sections.



Device	Summary
Memory	384 MB
Processors	1
Hard Disk (IDE)	8 GB
Hard Disk 2 (IDE)	8 GB
CD/DVD (IDE)	Auto detect
Network Adapter	Custom (VMnet4)
USB Controller	Present
Sound Card	Auto detect

Device status

☑ Connected

☐ Connect at power on

Network connection

○ Bridged: Connected directly to the physical network

☐ Replicate physical network connection state

○ NAT: Used to share the host's IP address

○ Host-only: A private network shared with the host

● Custom: Specific virtual network

VMnet0

Figure 2-2: Virtual network configuration options for a network adapter

Disconnecting the Network

Although you can configure a virtual machine to have no network connectivity, it's usually not a good idea to disconnect the network. Doing so will be useful only in certain cases. Without network connectivity, you won't be able to analyze malicious network activity.

Still, should you have reason to disconnect the network in VMware, you can do so either by removing the network adapter from the virtual machine or by disconnecting the network adapter from the network by choosing **VM ▸ Removable Devices**.

You can also control whether a network adapter is connected automatically when the machine is turned on by checking the **Connect at power on** checkbox (see Figure 2-2).

Setting Up Host-Only Networking

Host-only networking, a feature that creates a separate private LAN between the host OS and the guest OS, is commonly used for malware analysis. A host-only LAN is not connected to the Internet, which means that the malware is contained within your virtual machine but allowed some network connectivity.

NOTE *When configuring your host computer, ensure that it is fully patched, as protection in case the malware you're testing tries to spread. It's a good idea to configure a restrictive firewall to the host from the virtual machine to help prevent the malware from spreading to your host. The Microsoft firewall that comes with Windows XP Service Pack 2 and later is well documented and provides sufficient protection. Even if patches are up to date, however, the malware could spread by using a zero-day exploit against the host OS.*

Figure 2-3 illustrates the network configuration for host-only networking. When host-only networking is enabled, VMware creates a virtual network adapter in the host and virtual machines, and connects the two without touching the host's physical network adapter. The host's physical network adapter is still connected to the Internet or other external network.

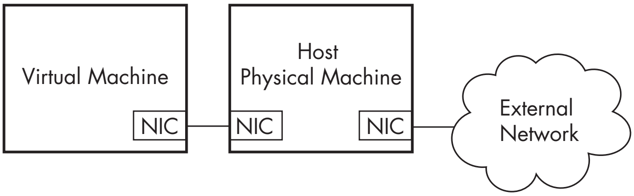


Figure 2-3: Host-only networking in VMware

Using Multiple Virtual Machines

One last configuration combines the best of all options. It requires multiple virtual machines linked by a LAN but disconnected from the Internet and host machine, so that the malware is connected to a network, but the network isn't connected to anything important.

Figure 2-4 shows a custom configuration with two virtual machines connected to each other. In this configuration, one virtual machine is set up to analyze malware, and the second machine provides services. The two virtual machines are connected to the same VMNet virtual switch. In this case, the host machine is still connected to the external network, but not to the machine running the malware.

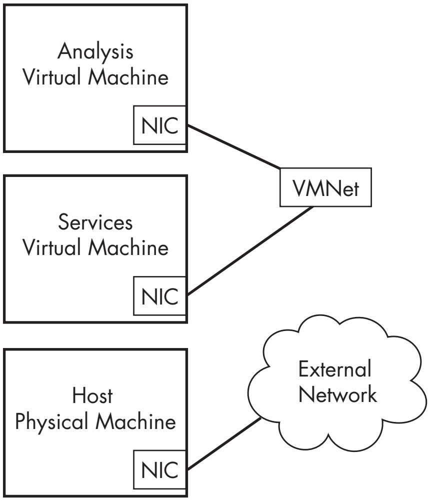


Figure 2-4: Custom networking in VMware

When using more than one virtual machine for analysis, you'll find it useful to combine the machines as a *virtual machine team*. When your machines are joined as part of a virtual machine team, you will be able to manage their power and network settings together. To create a new virtual machine team, choose **File ▸ New ▸ Team**.

Using Your Malware Analysis Machine

To exercise the functionality of your subject malware as much as possible, you must simulate all network services on which the malware relies. For example, malware commonly connects to an HTTP server to download additional malware. To observe this activity, you'll need to give the malware access to a Domain Name System (DNS) server to resolve the server's IP address, as well as an HTTP server to respond to requests. With the custom network configuration just described, the machine providing services should be running the services required for the malware to communicate. (We'll discuss a variety of tools useful for simulating network services in the next chapter.)

Connecting Malware to the Internet

Sometimes you'll want to connect your malware-running machine to the Internet to provide a more realistic analysis environment, despite the obvious risks. The biggest risk, of course, is that your computer will perform malicious activity, such as spreading malware to additional hosts, becoming a node in a distributed denial-of-service attack, or simply spamming. Another risk is that the malware writer could notice that you are connecting to the malware server and trying to analyze the malware.

You should never connect malware to the Internet without first performing some analysis to determine what the malware might do when connected. Then connect only if you are comfortable with the risks.

The most common way to connect a virtual machine to the Internet using VMware is with a *bridged network adapter*, which allows the virtual machine to be connected to the same network interface as the physical machine. Another way to connect malware running on a virtual machine to the Internet is to use VMware's Network Address Translation (NAT) mode.

NAT mode shares the host's IP connection to the Internet. The host acts like a router and translates all requests from the virtual machine so that they come from the host's IP address. This mode is useful when the host is connected to the network, but the network configuration makes it difficult, if not impossible, to connect the virtual machine's adapter to the same network.

For example, if the host is using a wireless adapter, NAT mode can be easily used to connect the virtual machine to the network, even if the wireless network has Wi-Fi Protected Access (WPA) or Wired Equivalent Privacy (WEP) enabled. Or, if the host adapter is connected to a network that allows only certain network adapters to connect, NAT mode allows the virtual machine to connect through the host, thereby avoiding the network's access control settings.

Connecting and Disconnecting Peripheral Devices

Peripheral devices, such as CD-ROMs and external USB storage drives, pose a particular problem for virtual machines. Most devices can be connected either to the physical machine or the virtual machine, but not both.

The VMware interface allows you to connect and disconnect external devices to virtual machines. If you connect a USB device to a machine while the virtual machine window is active, VMware will connect the USB device to the guest and not the host, which may be undesirable, considering the growing popularity of worms that spread via USB storage devices. To modify this setting, choose **VM ▸ Settings ▸ USB Controller** and uncheck the **Automatically connect new USB devices** checkbox to prevent USB devices from being connected to the virtual machine.

Taking Snapshots

Taking *snapshots* is a concept unique to virtual machines. VMware's virtual machine snapshots allow you save a computer's current state and return to that point later, similar to a Windows restore point.

The timeline in Figure 2-5 illustrates how taking snapshots works. At 8:00 you take a snapshot of the computer. Shortly after that, you run the malware sample. At 10:00, you revert to the snapshot. The OS, software, and other components of the machine return to the same state they were in at 8:00, and everything that occurred between 8:00 and 10:00 is erased as though it never happened. As you can see, taking snapshots is an extremely powerful tool. It's like a built-in undo feature that saves you the hassle of needing to reinstall your OS.

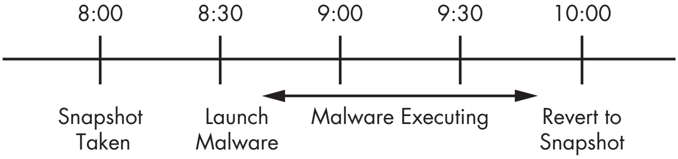


Figure 2-5: Snapshot timeline

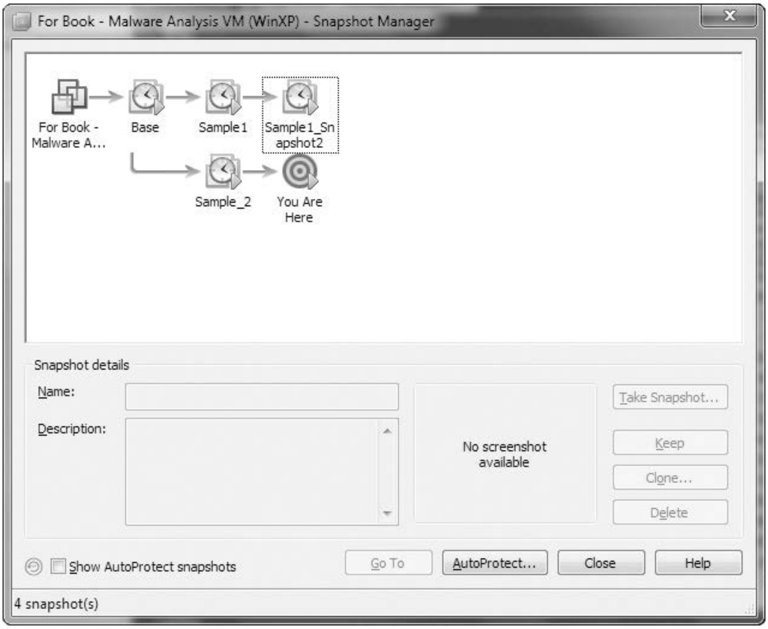
After you've installed your OS and malware analysis tools, and you have configured the network, take a snapshot. Use that snapshot as your base, clean-slate snapshot. Next, run your malware, complete your analysis, and then save your data and revert to the base snapshot, so that you can do it all over again.

But what if you're in the middle of analyzing malware and you want to do something different with your virtual machine without erasing *all* of your progress? VMware's Snapshot Manager allows you to return to any snapshot at any time, no matter which additional snapshots have been taken since then or what has happened to the machine. In addition, you can branch your snapshots so that they follow different paths. Take a look at the following example workflow:

1. While analyzing malware sample 1, you get frustrated and want to try another sample.
2. You take a snapshot of the malware analysis of sample 1.
3. You return to the base image.

4. You begin to analyze malware sample 2.
5. You take a snapshot to take a break.

When you return to your virtual machine, you can access either snapshot at any time, as shown in Figure 2-6. The two machine states are completely independent, and you can save as many snapshots as you have disk space.



No screenshot available

4 snapshot(s)

Figure 2-6: VMware Snapshot Manager

Transferring Files from a Virtual Machine

One drawback of using snapshots is that any work undertaken on the virtual machine is lost when you revert to an earlier snapshot. You can, however, save your work before loading the earlier snapshot by transferring any files that you want to keep to the host OS using VMware's drag-and-drop feature. As long as VMware Tools is installed in the guest OS and both systems are running Windows, you should be able to drag and drop a file directly from the guest OS to the host OS. This is the simplest and easiest way to transfer files.

Another way to transfer your data is with VMware's shared folders. A *shared folder* is accessible from both the host and the guest OS, similar to a shared Windows folder.

The Risks of Using VMware for Malware Analysis

Some malware can detect when it is running within a virtual machine, and many techniques have been published to detect just such a situation. VMware does not consider this a vulnerability and does not take explicit steps to avoid detection, but some malware will execute differently when running on a virtual machine to make life difficult for malware analysts. (Chapter 17 discusses such anti-VMware techniques in more detail.)

And, like all software, VMware occasionally has vulnerabilities. These can be exploited, causing the host OS to crash, or even used to run code on the host OS. Although only few public tools or well-documented ways exist to exploit VMware, vulnerabilities have been found in the shared folders feature, and tools have been released to exploit the drag-and-drop functionality. Make sure that you keep your VMware version fully patched.

And, of course, even after you take all possible precautions, some risk is always present when you're analyzing malware. Whatever you do, and even if you are running your analysis in a virtual machine, you should avoid performing malware analysis on any critical or sensitive machine.

Record/Replay: Running Your Computer in Reverse

One of VMware's more interesting features is record/replay. This feature in VMware Workstation records everything that happens so that you can replay the recording at a later time. The recording offers 100 percent fidelity; every instruction that executed during the original recording is executed during a replay. Even if the recording includes a one-in-a-million race condition that you can't replicate, it will be included in the replay.

VMware also has a movie-capture feature that records only the video output, but record/replay actually executes the CPU instructions of the OS and programs. And, unlike a movie, you can interrupt the execution at any point to interact with the computer and make changes in the virtual machine. For example, if you make a mistake in a program that lacks an undo feature, you can restore your virtual machine to the point prior to that mistake to do something different.

As we introduce more tools throughout this book, we'll examine many more powerful ways to use record/replay. We'll return to this feature in Chapter 8.

Conclusion

Running and analyzing malware using VMware and virtual machines involves the following steps:

1. Start with a clean snapshot with no malware running on it.
2. Transfer the malware to the virtual machine.
3. Conduct your analysis on the virtual machine.
4. Take your notes, screenshots, and data from the virtual machine and transfer it to the physical machine.
5. Revert the virtual machine to the clean snapshot.

As new malware analysis tools are released and existing tools are updated, you will need to update your clean base image. Simply install the tools and updates, and then take a new, clean snapshot.

To analyze malware, you usually need to run the malware to observe its behavior. When running malware, you must be careful not to infect your computer or networks. VMware allows you to run malware in a safe, controllable environment, and it provides the tools you need to clean the malware when you have finished analyzing it.

Throughout this book, when we discuss running malware, we assume that you are running the malware within a virtual machine.

3

BASIC DYNAMIC ANALYSIS

Dynamic analysis is any examination performed after executing malware. Dynamic analysis techniques are the second step in the malware analysis process. Dynamic analysis is typically performed after basic static analysis has reached a dead end, whether due to obfuscation, packing, or the analyst having exhausted the available static analysis techniques. It can involve monitoring malware as it runs or examining the system after the malware has executed.

Unlike static analysis, dynamic analysis lets you observe the malware's true functionality, because, for example, the existence of an action string in a binary does not mean the action will actually execute. Dynamic analysis is also an efficient way to identify malware functionality. For example, if your malware is a keylogger, dynamic analysis can allow you to locate the keylogger's log file on the system, discover the kinds of records it keeps, decipher where it sends its information, and so on. This kind of insight would be more difficult to gain using only basic static techniques.

Although dynamic analysis techniques are extremely powerful, they should be performed only after basic static analysis has been completed, because dynamic analysis can put your network and system at risk. Dynamic techniques do have their limitations, because not all code paths may execute when a piece of malware is run. For example, in the case of command-line malware that requires arguments, each argument could execute different program functionality, and without knowing the options you wouldn't be able to dynamically examine all of the program's functionality. Your best bet will be to use advanced dynamic or static techniques to figure out how to force the malware to execute all of its functionality. This chapter describes the basic dynamic analysis techniques.

Sandboxes: The Quick-and-Dirty Approach

Several all-in-one software products can be used to perform basic dynamic analysis, and the most popular ones use sandbox technology. A *sandbox* is a security mechanism for running untrusted programs in a safe environment without fear of harming "real" systems. Sandboxes comprise virtualized environments that often simulate network services in some fashion to ensure that the software or malware being tested will function normally.

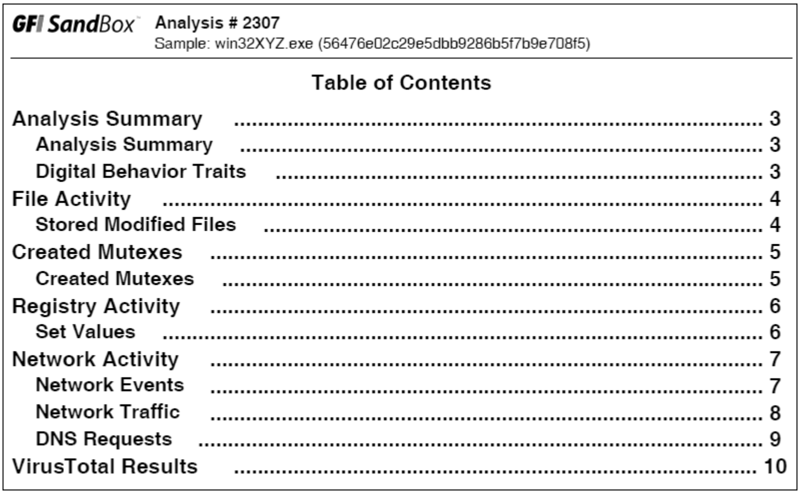
Using a Malware Sandbox

Many malware sandboxes—such as Norman SandBox, GFI Sandbox, Anubis, Joe Sandbox, ThreatExpert, BitBlaze, and Comodo Instant Malware Analysis—will analyze malware for free. Currently, Norman SandBox and GFI Sandbox (formerly CWSandbox) are the most popular among computer-security professionals.

These sandboxes provide easy-to-understand output and are great for initial triage, as long as you are willing to submit your malware to the sandbox websites. Even though the sandboxes are automated, you might choose not to submit malware that contains company information to a public website.

NOTE *You can purchase sandbox tools for in-house use, but they are extremely expensive. Instead, you can discover everything that these sandboxes can find using the basic techniques discussed in this chapter. Of course, if you have a lot of malware to analyze, it might be worth purchasing a sandbox software package that can be configured to process malware quickly.*

Most sandboxes work similarly, so we'll focus on one example, GFI Sandbox. Figure 3-1 shows the table of contents for a PDF report generated by running a file through GFI Sandbox's automated analysis. The malware report includes a variety of details on the malware, such as the network activity it performs, the files it creates, the results of scanning with VirusTotal, and so on.



GFI SandBox Analysis # 2307
Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)

Table of Contents

Analysis Summary 3
  Analysis Summary 3
  Digital Behavior Traits 3
File Activity 4
  Stored Modified Files 4
Created Mutexes 5
  Created Mutexes 5
Registry Activity 6
  Set Values 6
Network Activity 7
  Network Events 7
  Network Traffic 8
  DNS Requests 9
VirusTotal Results 10

Figure 3-1: GFI Sandbox sample results for win32XYZ.exe

Reports generated by GFI Sandbox vary in the number of sections they contain, based on what the analysis finds. The GFI Sandbox report has six sections in Figure 3-1, as follows:

- The Analysis Summary section lists static analysis information and a high-level overview of the dynamic analysis results.
- The File Activity section lists files that are opened, created, or deleted for each process impacted by the malware.
- The Created Mutexes section lists mutexes created by the malware.
- The Registry Activity section lists changes to the registry.
- The Network Activity section includes network activity spawned by the malware, including setting up a listening port or performing a DNS request.
- The VirusTotal Results section lists the results of a VirusTotal scan of the malware.

Sandbox Drawbacks

Malware sandboxes do have a few major drawbacks. For example, the sandbox simply runs the executable, without command-line options. If the malware executable requires command-line options, it will not execute any code that runs only when an option is provided. In addition, if your subject malware is waiting for a command-and-control packet to be returned before launching a backdoor, the backdoor will not be launched in the sandbox.

The sandbox also may not record all events, because neither you nor the sandbox may wait long enough. For example, if the malware is set to sleep for a day before it performs malicious activity, you may miss that event. (Most sandboxes hook the `Sleep` function and set it to sleep only briefly, but there is more than one way to sleep, and the sandboxes cannot account for all of these.)

Other potential drawbacks include the following:

- Malware often detects when it is running in a virtual machine, and if a virtual machine is detected, the malware might stop running or behave differently. Not all sandboxes take this issue into account.
- Some malware requires the presence of certain registry keys or files on the system that might not be found in the sandbox. These might be required to contain legitimate data, such as commands or encryption keys.
- If the malware is a DLL, certain exported functions will not be invoked properly, because a DLL will not run as easily as an executable.
- The sandbox environment OS may not be correct for the malware. For example, the malware might crash on Windows XP but run correctly in Windows 7.
- A sandbox cannot tell you what the malware does. It may report basic functionality, but it cannot tell you that the malware is a custom Security Accounts Manager (SAM) hash dump utility or an encrypted keylogging backdoor, for example. Those are conclusions that you must draw on your own.

Running Malware

Basic dynamic analysis techniques will be rendered useless if you can't get the malware running. Here we focus on running the majority of malware you will encounter (EXEs and DLLs). Although you'll usually find it simple enough to run executable malware by double-clicking the executable or running the file from the command line, it can be tricky to launch malicious DLLs because Windows doesn't know how to run them automatically. (We'll discuss DLL internals in depth in Chapter 7.)

Let's take a look at how you can launch DLLs to be successful in performing dynamic analysis.

The program *rundll32.exe* is included with all modern versions of Windows. It provides a container for running a DLL using this syntax:

```
C:\>rundll32.exe DLLname, Export arguments
```

The `Export` value must be a function name or ordinal selected from the exported function table in the DLL. As you learned in Chapter 1, you can use a tool such as PEview or PE Explorer to view the Export table. For example, the file *rip.dll* has the following exports:

```
Install
Uninstall
```

`Install` appears to be a likely way to launch *rip.dll*, so let's launch the malware as follows:

```
C:\>rundll32.exe rip.dll, Install
```

Malware can also have functions that are exported by ordinal—that is, as an exported function with only an ordinal number, which we discussed in depth in Chapter 1. In this case, you can still call those functions with *rundll32.exe* using the following command, where 5 is the ordinal number that you want to call, prepended with the # character:

```
C:\>rundll32.exe xyzzy.dll, #5
```

Because malicious DLLs frequently run most of their code in DLLMain (called from the DLL entry point), and because DLLMain is executed whenever the DLL is loaded, you can often get information dynamically by forcing the DLL to load using *rundll32.exe*. Alternatively, you can even turn a DLL into an executable by modifying the PE header and changing its extension to force Windows to load the DLL as it would an executable.

To modify the PE header, wipe the IMAGE_FILE_DLL (0x2000) flag from the Characteristics field in the IMAGE_FILE_HEADER. While this change won't run any imported functions, it will run the DLLMain method, and it may cause the malware to crash or terminate unexpectedly. However, as long as your changes cause the malware to execute its malicious payload, and you can collect information for your analysis, the rest doesn't matter.

DLL malware may also need to be installed as a service, sometimes with a convenient export such as InstallService, as listed in *ipr32x.dll*:

```
C:\>rundll32 ipr32x.dll,InstallService ServiceName
C:\>net start ServiceName
```

The *ServiceName* argument must be provided to the malware so it can be installed and run. The net start command is used to start a service on a Windows system.

NOTE *When you see a* ServiceMain *function without a convenient exported function such as* Install *or* InstallService, *you may need to install the service manually. You can do this by using the Windows* sc *command or by modifying the registry for an unused service, and then using* net start *on that service. The service entries are located in the registry at* HKLM\SYSTEM\CurrentControlSet\Services.

Monitoring with Process Monitor

Process Monitor, or procmon, is an advanced monitoring tool for Windows that provides a way to monitor certain registry, file system, network, process, and thread activity. It combines and enhances the functionality of two legacy tools: FileMon and RegMon.

Although procmon captures a lot of data, it doesn't capture everything. For example, it can miss the device driver activity of a user-mode component talking to a rootkit via device I/O controls, as well as certain GUI calls, such as SetWindowsHookEx. Although procmon can be a useful tool, it usually should not be used for logging network activity, because it does not work consistently across Microsoft Windows versions.

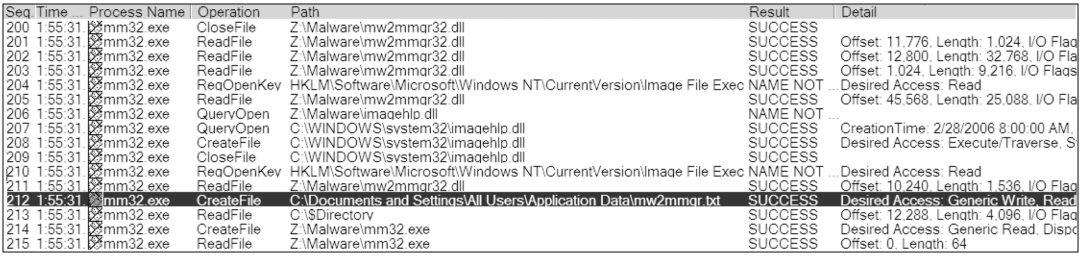
WARNING *Throughout this chapter, we will use tools to test malware dynamically. When you test malware, be sure to protect your computers and networks by using a virtual machine, as discussed in the previous chapter.*

Procmon monitors all system calls it can gather as soon as it is run. Because many system calls exist on a Windows machine (sometimes more than 50,000 events a minute), it's usually impossible to look through them all. As a result, because procmon uses RAM to log events until it is told to stop capturing, it can crash a virtual machine using all available memory. To avoid this, run procmon for limited periods of time. To stop procmon from capturing events, choose **File ▸ Capture Events**. Before using procmon for analysis, first clear all currently captured events to remove irrelevant data by choosing **Edit ▸ Clear Display**. Next, run the subject malware with capture turned on. After a few minutes, you can discontinue event capture.

The Procmon Display

Procmon displays configurable columns containing information about individual events, including the event's sequence number, timestamp, name of the process causing the event, event operation, path used by the event, and result of the event. This detailed information can be too long to fit on the screen, or it can be otherwise difficult to read. If you find either to be the case, you can view the full details of a particular event by double-clicking its row.

Figure 3-2 shows a collection of procmon events that occurred on a machine running a piece of malware named *mm32.exe*. Reading the Operation column will quickly tell you which operations *mm32.exe* performed on this system, including registry and file system accesses. One entry of note is the creation of a file *C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt* at sequence number 212 using `CreateFile`. The word *SUCCESS* in the Result column tells you that this operation was successful.



Seq.	Time ...	Process Name	Operation	Path	Result	Detail
200	1:55:31.	mm32.exe	CloseFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	
201	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 11,776, Length: 1,024, I/O Flag
202	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 12,800, Length: 32,768, I/O Fla
203	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 1,024, Length: 9,216, I/O Flags
204	1:55:31.	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT ...	Desired Access: Read
205	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 45,568, Length: 25,088, I/O Fla
206	1:55:31.	mm32.exe	QueryOpen	Z:\Malware\imagehlp.dll	NAME NOT ...	
207	1:55:31.	mm32.exe	QueryOpen	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	CreationTime: 2/28/2006 8:00:00 AM,
208	1:55:31.	mm32.exe	CreateFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	Desired Access: Execute/Traverse, S
209	1:55:31.	mm32.exe	CloseFile	C:\WINDOWS\system32\imagehlp.dll	SUCCESS	
210	1:55:31.	mm32.exe	RegOpenKey	HKLM\Software\Microsoft\Windows NT\CurrentVersion\Image File Exec	NAME NOT ...	Desired Access: Read
211	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mw2mmgr32.dll	SUCCESS	Offset: 10,240, Length: 1,536, I/O Flag
212	1:55:31.	mm32.exe	CreateFile	C:\Documents and Settings\All Users\Application Data\mw2mmgr.txt	SUCCESS	Desired Access: Generic Write, Read
213	1:55:31.	mm32.exe	ReadFile	C:\$Directory	SUCCESS	Offset: 12,288, Length: 4,096, I/O Flag
214	1:55:31.	mm32.exe	CreateFile	Z:\Malware\mm32.exe	SUCCESS	Desired Access: Generic Read, Dispo
215	1:55:31.	mm32.exe	ReadFile	Z:\Malware\mm32.exe	SUCCESS	Offset: 0, Length: 64

Figure 3-2: Procmon mm32.exe *example*

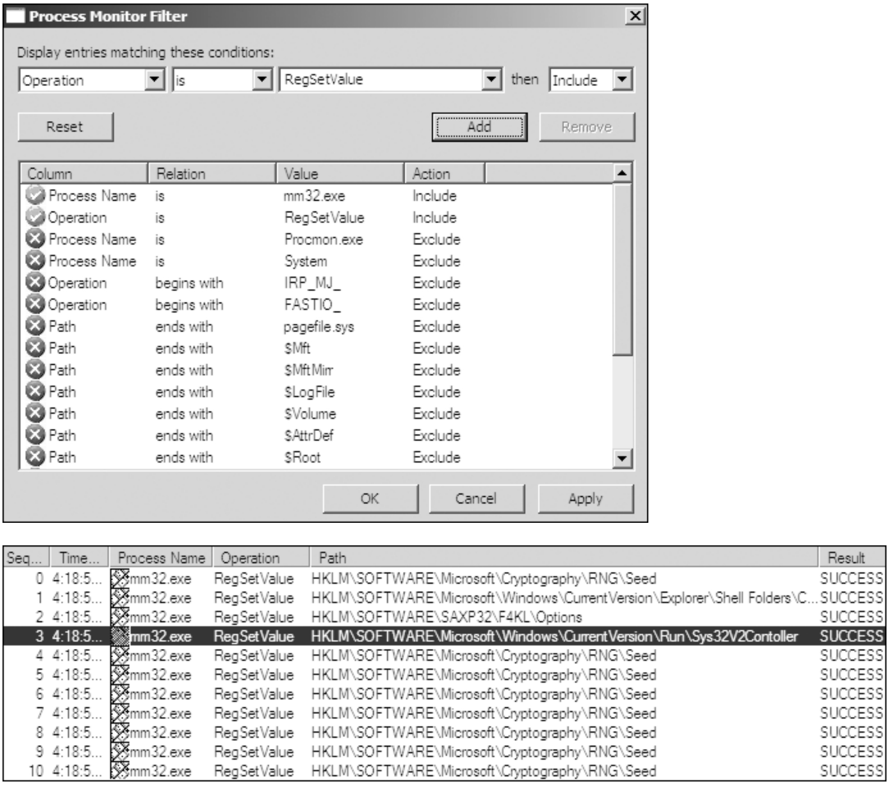
Filtering in Procmon

It's not always easy to find information in procmon when you are looking through thousands of events, one by one. That's where procmon's filtering capability is key.

You can set procmon to filter on one executable running on the system. This feature is particularly useful for malware analysis, because you can set a filter on the piece of malware you are running. You can also filter on individual system calls such as `RegSetValue`, `CreateFile`, `WriteFile`, or other suspicious or destructive calls.

When procmon filtering is turned on, it filters through recorded events only. All recorded events are still available even though the filter shows only a limited display. Setting a filter is not a way to prevent procmon from consuming too much memory.

To set a filter, choose **Filter ▸ Filter** to open the Filter menu, as shown in the top image of Figure 3-3. When setting a filter, first select a column to filter on using the drop-down box at the upper left, above the Reset button. The most important filters for malware analysis are Process Name, Operation, and Detail. Next, select a comparator, choosing from options such as Is, Contains, and Less Than. Finally, choose whether this is a filter to include or exclude from display. Because, by default, the display will show all system calls, it is important to reduce the amount displayed.



Column	Relation	Value	Action
✓ Process Name	is	mm32.exe	Include
✓ Operation	is	RegSetValue	Include
✗ Process Name	is	Procmon.exe	Exclude
✗ Process Name	is	System	Exclude
✗ Operation	begins with	IRP_MJ_	Exclude
✗ Operation	begins with	FASTIO_	Exclude
✗ Path	ends with	pagefile.sys	Exclude
✗ Path	ends with	$Mft	Exclude
✗ Path	ends with	$MftMirr	Exclude
✗ Path	ends with	$LogFile	Exclude
✗ Path	ends with	$Volume	Exclude
✗ Path	ends with	$AttrDef	Exclude
✗ Path	ends with	$Root	Exclude

Seq...	Time...	Process Name	Operation	Path	Result
0	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
1	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\C...	SUCCESS
2	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\SAXP32\F4KL\Options	SUCCESS
3	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Contoller	SUCCESS
4	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
5	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
6	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
7	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
8	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
9	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS
10	4:18:5...	mm32.exe	RegSetValue	HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed	SUCCESS

Figure 3-3: Setting a procmon filter

NOTE *Procmon uses some basic filters by default. For example, it contains a filter that excludes* procmon.exe *and one that excludes the pagefile from logging, because it is accessed often and provides no useful information.*

As you can see in the first two rows of Figure 3-3, we're filtering on Process Name and Operation. We've added a filter on Process Name equal to *mm32.exe* that's active when the Operation is set to `RegSetValue`.

After you've chosen a filter, click **Add** for each, and then click **Apply**. As a result of applying our filters, the display window shown in the lower image displays only 11 of the 39,351 events, making it easier for us to see that *mm32.exe* performed a `RegSetValue` of registry key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Controller` (sequence number 3 using `RegSetValue`). Double-clicking this `RegSetValue` event will reveal the data written to this location, which is the current path to the malware.

If the malware extracted another executable and ran it, don't worry, because that information is still there. Remember that the filter controls only the display. All of the system calls that occurred when you ran the malware are captured, including system calls from malware that was extracted by the original executable. If you see any malware extracted, change the filter to display the extracted name, and then click **Apply**. The events related to the extracted malware will be displayed.

Procmon provides helpful automatic filters on its toolbar. The four filters circled in Figure 3-4 filter by the following categories:

Registry By examining registry operations, you can tell how a piece of malware installs itself in the registry.

File system Exploring file system interaction can show all files that the malware creates or configuration files it uses.

Process activity Investigating process activity can tell you whether the malware spawned additional processes.

Network Identifying network connections can show you any ports on which the malware is listening.

All four filters are selected by default. To turn off a filter, simply click the icon in the toolbar corresponding to the category.

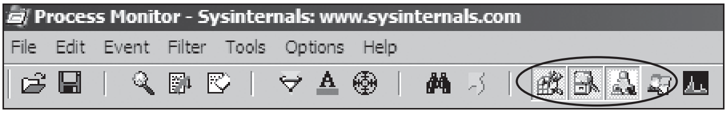

Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help

Figure 3-4: Filter buttons for procmon

NOTE *If your malware runs at boot time, use procmon's boot logging options to install procmon as a startup driver to capture startup events.*

Analysis of procmon's recorded events takes practice and patience, since many events are simply part of the standard way that executables start up. The more you use procmon, the easier you will find it to quickly review the event listing.

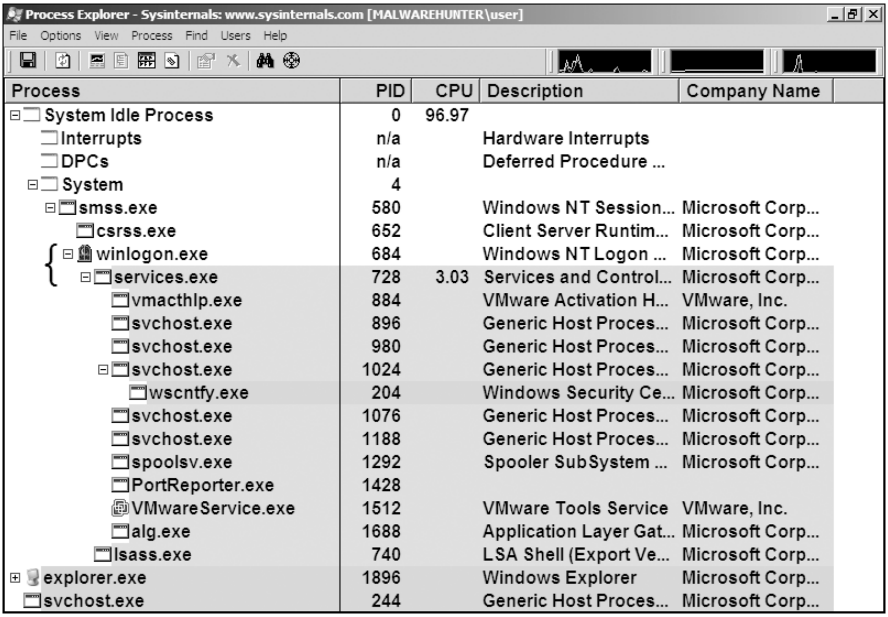
Viewing Processes with Process Explorer

The Process Explorer, free from Microsoft, is an extremely powerful task manager that should be running when you are performing dynamic analysis. It can provide valuable insight into the processes currently running on a system.

You can use Process Explorer to list active processes, DLLs loaded by a process, various process properties, and overall system information. You can also use it to kill a process, log out users, and launch and validate processes.

The Process Explorer Display

Process Explorer monitors the processes running on a system and shows them in a tree structure that displays child and parent relationships. For example, in Figure 3-5 you can see that *services.exe* is a child process of *winlogon.exe*, as indicated by the left curly bracket.


Process Explorer - Sysinternals: www.sysinternals.com [MALWAREHUNTER\user]
File
Options
View
Process
Find
Users
Help

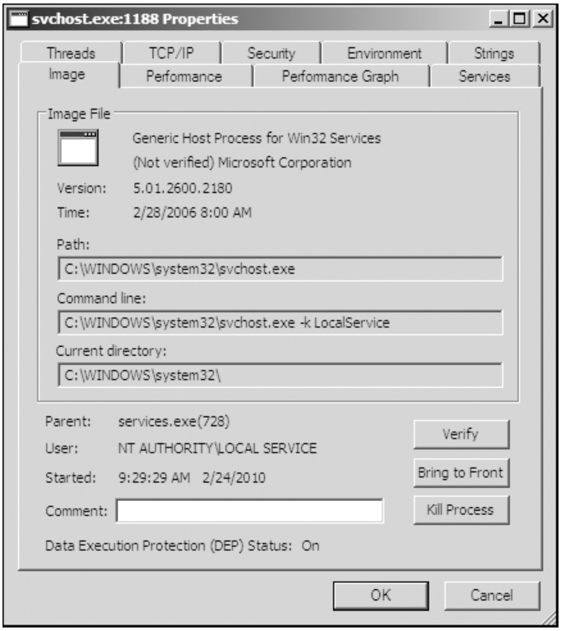
Process	PID	CPU	Description	Company Name
System Idle Process	0	96.97		
Interrupts	n/a		Hardware Interrupts	
DPCs	n/a		Deferred Procedure ...	
System	4			
smss.exe	580		Windows NT Session...	Microsoft Corp...
csrss.exe	652		Client Server Runtim...	Microsoft Corp...
winlogon.exe	684		Windows NT Logon ...	Microsoft Corp...
services.exe	728	3.03	Services and Control...	Microsoft Corp...
vmacthlp.exe	884		VMware Activation H...	VMware, Inc.
svchost.exe	896		Generic Host Proces...	Microsoft Corp...
svchost.exe	980		Generic Host Proces...	Microsoft Corp...
svchost.exe	1024		Generic Host Proces...	Microsoft Corp...
wscntfy.exe	204		Windows Security Ce...	Microsoft Corp...
svchost.exe	1076		Generic Host Proces...	Microsoft Corp...
svchost.exe	1188		Generic Host Proces...	Microsoft Corp...
spoolsv.exe	1292		Spooler SubSystem ...	Microsoft Corp...
PortReporter.exe	1428			
VMwareService.exe	1512		VMware Tools Service	VMware, Inc.
alg.exe	1688		Application Layer Gat...	Microsoft Corp...
lsass.exe	740		LSA Shell (Export Ve...	Microsoft Corp...
explorer.exe	1896		Windows Explorer	Microsoft Corp...
svchost.exe	244		Generic Host Proces...	Microsoft Corp...

Figure 3-5: Process Explorer examining svchost.exe *malware*

Process Explorer shows five columns: Process (the process name), PID (the process identifier), CPU (CPU usage), Description, and Company Name. The view updates every second. By default, services are highlighted in pink, processes in blue, new processes in green, and terminated processes in red. Green and red highlights are temporary, and are removed after the process has started or terminated. When analyzing malware, watch the Process Explorer window for changes or new processes, and be sure to investigate them thoroughly.

Process Explorer can display quite a bit of information for each process. For example, when the DLL information display window is active, you can click a process to see all DLLs it loaded into memory. You can change the DLL display window to the Handles window, which shows all handles held by the process, including file handles, mutexes, events, and so on.

The Properties window shown in Figure 3-6 opens when you double-click a process name. This window can provide some particularly useful information about your subject malware. The Threads tab shows all active threads, the TCP/IP tab displays active connections or ports on which the process is listening, and the Image tab (opened in the figure) shows the path on disk to the executable.



Generic Host Process for Win32 Services
(Not verified) Microsoft Corporation

Version: 5.01.2600.2180
Time: 2/28/2006 8:00 AM

Path:
C:\WINDOWS\system32\svchost.exe

Command line:
C:\WINDOWS\system32\svchost.exe -k LocalService

Current directory:
C:\WINDOWS\system32\

Parent: services.exe(728)
User: NT AUTHORITY\LOCAL SERVICE
Started: 9:29:29 AM 2/24/2010
Comment:

Data Execution Protection (DEP) Status: On

Figure 3-6: The Properties window, Image tab

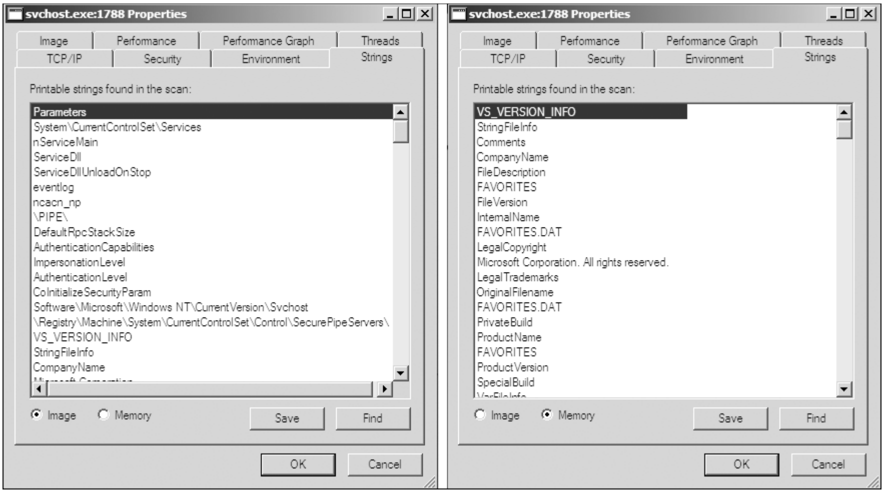
Using the Verify Option

One particularly useful Process Explorer feature is the Verify button on the Image tab. Click this button to verify that the image on disk is, in fact, the Microsoft signed binary. Because Microsoft uses digital signatures for most of its core executables, when Process Explorer verifies that a signature is valid, you can be sure that the file is actually the executable from Microsoft. This feature is particularly useful for verifying that the Windows file on disk has not been corrupted; malware often replaces authentic Windows files with its own in an attempt to hide.

The Verify button verifies the image on disk rather than in memory, and it is useless if an attacker uses *process replacement*, which involves running a process on the system and overwriting its memory space with a malicious executable. Process replacement provides the malware with the same privileges as the process it is replacing, so that the malware appears to be executing as a legitimate process, but it leaves a fingerprint: The image in memory will differ from the image on disk. For example, in Figure 3-6, the *svchost.exe* process is verified, yet it is actually malware. We'll discuss process replacement in more detail in Chapter 12.

Comparing Strings

One way to recognize process replacement is to use the Strings tab in the Process Properties window to compare the strings contained in the disk executable (image) against the strings in memory for that same executable running in memory. You can toggle between these string views using the buttons at the bottom-left corner, as shown in Figure 3-7. If the two string listings are drastically different, process replacement may have occurred. This string discrepancy is displayed in Figure 3-7. For example, the string `FAVORITES.DAT` appears multiple times in the right half of the figure (*svchost.exe* in memory), but it cannot be found in the left half of the figure (*svchost.exe* on disk).



```
Parameters
System\CurrentControlSet\Services
nServiceMain
ServiceDll
ServiceDllUnloadOnStop
eventlog
ncacn_np
\PIPE\
DefaultRpcStackSize
AuthenticationCapabilities
ImpersonationLevel
AuthenticationLevel
CoInitializeSecurityParam
Software\Microsoft\Windows NT\CurrentVersion\Svchost
\Registry\Machine\System\CurrentControlSet\Control\SecurePipeServers\
VS_VERSION_INFO
StringFileInfo
CompanyName
```

```
VS_VERSION_INFO
StringFileInfo
Comments
CompanyName
FileDescription
FAVORITES
FileVersion
InternalName
FAVORITES.DAT
LegalCopyright
Microsoft Corporation. All rights reserved.
LegalTrademarks
OriginalFilename
FAVORITES.DAT
PrivateBuild
ProductName
FAVORITES
ProductVersion
SpecialBuild
```

Figure 3-7: The Process Explorer Strings tab shows strings on disk (left) versus strings in memory (right) for active svchost.exe.

Using Dependency Walker

Process Explorer allows you to launch *depends.exe* (Dependency Walker) on a running process by right-clicking a process name and selecting **Launch Depends.** It also lets you search for a handle or DLL by choosing **Find ▸ Find Handle or DLL**.

The Find DLL option is particularly useful when you find a malicious DLL on disk and want to know if any running processes use that DLL. The Verify button verifies the EXE file on disk, but not every DLL loaded during runtime. To determine whether a DLL is loaded into a process after load time, you can compare the DLL list in Process Explorer to the imports shown in Dependency Walker.

Analyzing Malicious Documents

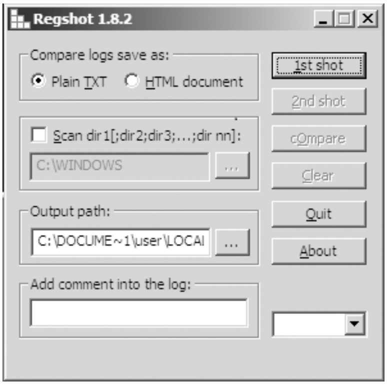
You can also use Process Explorer to analyze malicious documents, such as PDFs and Word documents. A quick way to determine whether a document is malicious is to open Process Explorer and then open the suspected malicious document. If the document launches any processes, you should see them in Process Explorer, and be able to locate the malware on disk via the Image tab of the Properties window.

NOTE *Opening a malicious document while using monitoring tools can be a quick way to determine whether a document is malicious; however, you will have success running only vulnerable versions of the document viewer. In practice, it is best to use intentionally unpatched versions of the viewing application to ensure that the exploitation will be successful. The easiest way to do this is with multiple snapshots of your analysis virtual machine, each with old versions of document viewers such as Adobe Reader and Microsoft Word.*

Comparing Registry Snapshots with Regshot

Regshot (shown in Figure 3-8) is an open source registry comparison tool that allows you to take and compare two registry snapshots.

To use Regshot for malware analysis, simply take the first shot by clicking the **1st Shot** button, and then run the malware and wait for it to finish making any system changes. Next, take the second shot by clicking the **2nd Shot** button. Finally, click the **Compare** button to compare the two snapshots.



Compare logs save as: ● Plain TXT ○ HTML document

□ Scan dir1[;dir2;dir3;...;dir nn]: C:\WINDOWS

Output path: C:\DOCUME~1\user\LOCAl

Add comment into the log:

Figure 3-8: Regshot window

Listing 3-1 displays a subset of the results generated by Regshot during malware analysis. Registry snapshots were taken before and after running the spyware *ckr.exe*.

```
Regshot
Comments:
Datetime: <date>
Computer: MALWAREANALYSIS
Username: username

----------------------------------
Keys added: 0
----------------------------------
```

```
  ---------------------------------
  Values added:3
  ---------------------------------
❶ HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\
  ckr.exe
  ...
  ...

  ---------------------------------
  Values modified:2
  ---------------------------------
❷ HKLM\SOFTWARE\Microsoft\Cryptography\RNG\Seed: 00 43 7C 25 9C 68 DE 59 C6 C8
  9D C3 1D E6 DC 87 1C 3A C4 E4 D9 0A B1 BA C1 FB 80 EB 83 25 74 C4 C5 E2 2F CE
  4E E8 AC C8 49 E8 E8 10 3F 13 F6 A1 72 92 28 8A 01 3A 16 52 86 36 12 3C C7 EB
  5F 99 19 1D 80 8C 8E BD 58 3A DB 18 06 3D 14 8F 22 A4
  ...

  ---------------------------------
  Total changes:5
  ---------------------------------
```

Listing 3-1: Regshot comparison results

As you can see *ckr.exe* creates a value at `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` as a persistence mechanism ❶. A certain amount of noise ❷ is typical in these results, because the random-number generator seed is constantly updated in the registry.

As with procmon, your analysis of these results requires patient scanning to find nuggets of interest.

Faking a Network

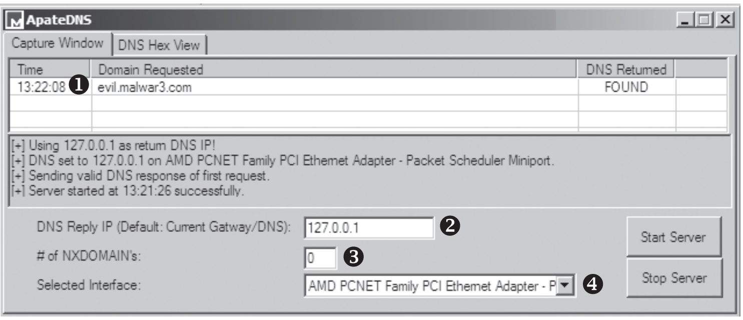
Malware often beacons out and eventually communicates with a command-and-control server, as we'll discuss in depth in Chapter 14. You can create a fake network and quickly obtain network indicators, without actually connecting to the Internet. These indicators can include DNS names, IP addresses, and packet signatures.

To fake a network successfully, you must prevent the malware from realizing that it is executing in a virtualized environment. (See Chapter 2 for a discussion on setting up virtual networks with VMware.) By combining the tools discussed here with a solid virtual machine network setup, you will greatly increase your chances of success.

Using ApateDNS

ApateDNS, a free tool from Mandiant (*www.mandiant.com/products/research/mandiant_apatedns/download*), is the quickest way to see DNS requests made by malware. ApateDNS spoofs DNS responses to a user-specified IP address by listening on UDP port 53 on the local machine. It responds to DNS requests with the DNS response set to an IP address you specify. ApateDNS can display the hexadecimal and ASCII results of all requests it receives.

To use ApateDNS, set the IP address you want sent in DNS responses at ❷ and select the interface at ❹. Next, press the **Start Server** button; this will automatically start the DNS server and change the DNS settings to localhost. Next, run your malware and watch as DNS requests appear in the ApateDNS window. For example, in Figure 3-9, we redirect the DNS requests made by malware known as *RShell*. We see that the DNS information is requested for *evil.malwar3.com* and that request was made at 13:22:08 ❶.



Time	Domain Requested	DNS Returned
13:22:08 ❶	evil.malwar3.com	FOUND

```
[+] Using 127.0.0.1 as return DNS IP!
[+] DNS set to 127.0.0.1 on AMD PCNET Family PCI Ethernet Adapter - Packet Scheduler Miniport.
[+] Sending valid DNS response of first request.
[+] Server started at 13:21:26 successfully.
```

DNS Reply IP (Default: Current Gatway/DNS): 127.0.0.1 ❷

# of NXDOMAIN's: 0 ❸

Selected Interface: AMD PCNET Family PCI Ethernet Adapter - P ❹

Figure 3-9: ApateDNS responding to a request for evil.malwar3.com

In the example shown in the figure, we redirect DNS requests to 127.0.0.1 (localhost), but you may want to change this address to point to something external, such as a fake web server running on a Linux virtual machine. Because the IP address will differ from that of your Windows malware analysis virtual machine, be sure to enter the appropriate IP address before starting the server. By default ApateDNS will use the current gateway or current DNS settings to insert into DNS responses.

You can catch additional domains used by a malware sample through the use of the nonexistent domain (NXDOMAIN) option at ❸. Malware will often loop through the different domains it has stored if the first or second domains are not found. Using this NXDOMAIN option can trick malware into giving you additional domains it has in its configuration.

Monitoring with Netcat

Netcat, the "TCP/IP Swiss Army knife," can be used over both inbound and outbound connections for port scanning, tunneling, proxying, port forwarding, and much more. In listen mode, Netcat acts as a server, while in connect mode it acts as a client. Netcat takes data from standard input for transmission over the network. All the data it receives is output to the screen via standard output.

Let's look at how you can use Netcat to analyze the malware *RShell* from Figure 3-9. Using ApateDNS, we redirect the DNS request for *evil.malwar3.com* to our local host. Assuming that the malware is going out over port 80 (a common choice), we can use Netcat to listen for connections before executing the malware.

Malware frequently uses port 80 or 443 (HTTP or HTTPS traffic, respectively), because these ports are typically not blocked or monitored as outbound connections. Listing 3-2 shows an example.

```
C:\> nc -l -p 80 ❶
POST /cq/frame.htm HTTP/1.1
Host: www.google.com ❷
User-Agent: Mozilla/5.0 (Windows; Windows NT 5.1; TWFsd2FyZUh1bnRlcg==;
rv:1.38)
Accept: text/html, application
Accept-Language: en-US, en:q=
Accept-Encoding: gzip, deflate
Keep-Alive: 300
Content-Type: application/x-form-urlencoded
Content-Length

Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

Z:\Malware> ❸
```

Listing 3-2: Netcat example listening on port 80

The Netcat (`nc`) command ❶ shows the options required to listen on a port. The `-l` flag means listen, and `-p` (with a port number) specifies the port on which to listen. The malware connects to our Netcat listener because we're using ApateDNS for redirection. As you can see, *RShell* is a reverse shell ❸, but it does not immediately provide the shell. The network connection first appears as an HTTP `POST` request to `www.google.com` ❷, fake `POST` data that *RShell* probably inserts to obfuscate its reverse shell, because network analysts frequently look only at the start of a session.

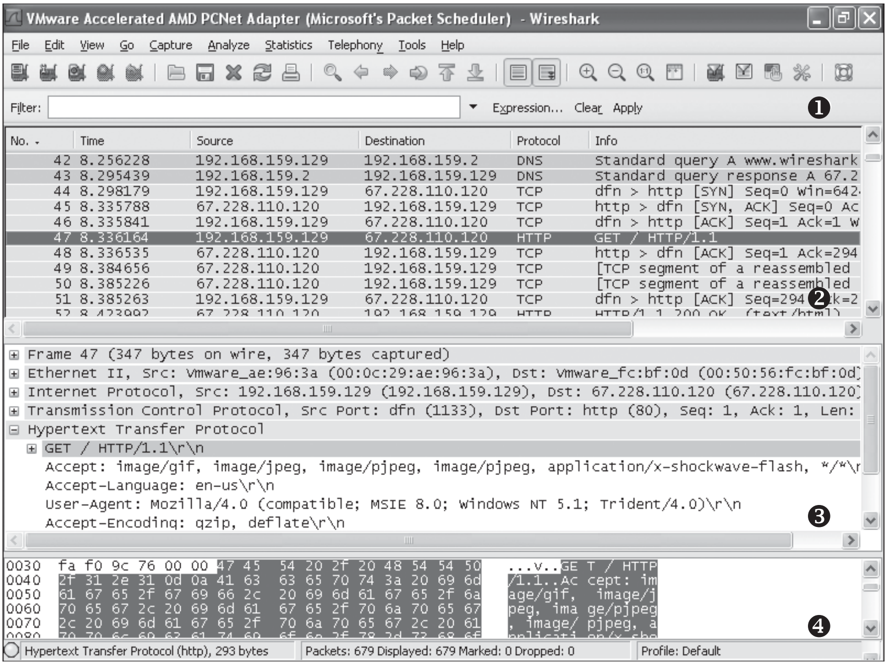
Packet Sniffing with Wireshark

Wireshark is an *open source sniffer*, a packet capture tool that intercepts and logs network traffic. Wireshark provides visualization, packet-stream analysis, and in-depth analysis of individual packets.

Like many tools discussed in this book, Wireshark can be used for both good and evil. It can be used to analyze internal networks and network usage, debug application issues, and study protocols in action. But it can also be used to sniff passwords, reverse-engineer network protocols, steal sensitive information, and listen in on the online chatter at your local coffee shop.

The Wireshark display has four parts, as shown in Figure 3-10:

- The Filter box ❶ is used to filter the packets displayed.
- The packet listing ❷ shows all packets that satisfy the display filter.
- The packet detail window ❸ displays the contents of the currently selected packet (in this case, packet 47).
- The hex window ❹ displays the hex contents of the current packet. The hex window is linked with the packet detail window and will highlight any fields you select.


VMware Accelerated AMD PCNet Adapter (Microsoft's Packet Scheduler) - Wireshark
File Edit View Go Capture Analyze Statistics Telephony Tools Help
Filter:
Expression... Clear Apply
❶
❷
❸
❹

```
No.   Time       Source            Destination       Protocol  Info
   42 8.256228   192.168.159.129   192.168.159.2     DNS       Standard query A www.wireshark
   43 8.295439   192.168.159.2     192.168.159.129   DNS       Standard query response A 67.2
   44 8.298179   192.168.159.129   67.228.110.120    TCP       dfn > http [SYN] Seq=0 Win=642
   45 8.335788   67.228.110.120    192.168.159.129   TCP       http > dfn [SYN, ACK] Seq=0 Ac
   46 8.335841   192.168.159.129   67.228.110.120    TCP       dfn > http [ACK] Seq=1 Ack=1 W
   47 8.336164   192.168.159.129   67.228.110.120    HTTP      GET / HTTP/1.1
   48 8.336535   67.228.110.120    192.168.159.129   TCP       http > dfn [ACK] Seq=1 Ack=294
   49 8.384656   67.228.110.120    192.168.159.129   TCP       [TCP segment of a reassembled
   50 8.385226   67.228.110.120    192.168.159.129   TCP       [TCP segment of a reassembled
   51 8.385263   192.168.159.129   67.228.110.120    TCP       dfn > http [ACK] Seq=294 Ack=2
   52 8.423992   67.228.110.120    192.168.159.129   HTTP      HTTP/1.1 200 OK  (text/html)
```

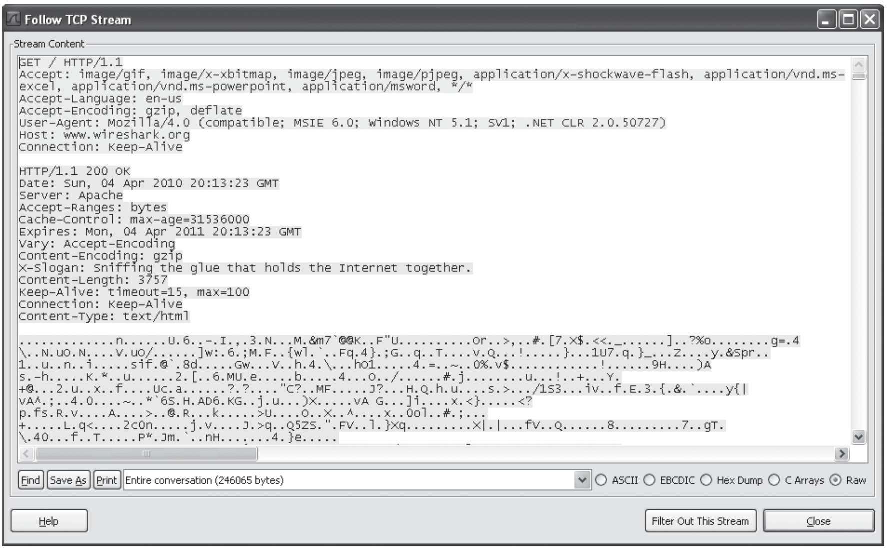
```
⊞ Frame 47 (347 bytes on wire, 347 bytes captured)
⊞ Ethernet II, Src: Vmware_ae:96:3a (00:0c:29:ae:96:3a), Dst: Vmware_fc:bf:0d (00:50:56:fc:bf:0d)
⊞ Internet Protocol, Src: 192.168.159.129 (192.168.159.129), Dst: 67.228.110.120 (67.228.110.120)
⊞ Transmission Control Protocol, Src Port: dfn (1133), Dst Port: http (80), Seq: 1, Ack: 1, Len:
⊟ Hypertext Transfer Protocol
  ⊞ GET / HTTP/1.1\r\n
    Accept: image/gif, image/jpeg, image/pjpeg, image/pjpeg, application/x-shockwave-flash, */*\r
    Accept-Language: en-us\r\n
    User-Agent: Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 5.1; Trident/4.0)\r\n
    Accept-Encoding: gzip, deflate\r\n
```

```
0030  fa f0 9c 76 00 00 47 45  54 20 2f 20 48 54 54 50   ...v..GE T / HTTP
0040  2f 31 2e 31 0d 0a 41 63  63 65 70 74 3a 20 69 6d   /1.1..Ac cept: im
0050  61 67 65 2f 67 69 66 2c  20 69 6d 61 67 65 2f 6a   age/gif,  image/j
0060  70 65 67 2c 20 69 6d 61  67 65 2f 70 6a 70 65 67   peg, ima ge/pjpeg
0070  2c 20 69 6d 61 67 65 2f  70 6a 70 65 67 2c 20 61   , image/ pjpeg, a
0080  70 70 6c 69 63 61 74 69  6f 6e 2f 78 2d 73 68 6f   pplicati on/x-sho
```

Hypertext Transfer Protocol (http), 293 bytes | Packets: 679 Displayed: 679 Marked: 0 Dropped: 0 | Profile: Default

Figure 3-10: Wireshark DNS and HTTP example

To use Wireshark to view the contents of a TCP session, right-click any TCP packet and select **Follow TCP Stream**. As you can see in Figure 3-11, both ends of the conversation are displayed in session order, with different colors showing each side of the connection.



```
GET / HTTP/1.1
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, application/x-shockwave-flash, application/vnd.ms-
excel, application/vnd.ms-powerpoint, application/msword, */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; .NET CLR 2.0.50727)
Host: www.wireshark.org
Connection: Keep-Alive

HTTP/1.1 200 OK
Date: Sun, 04 Apr 2010 20:13:23 GMT
Server: Apache
Accept-Ranges: bytes
Cache-Control: max-age=31536000
Expires: Mon, 04 Apr 2011 20:13:23 GMT
Vary: Accept-Encoding
Content-Encoding: gzip
X-Slogan: Sniffing the glue that holds the Internet together.
Content-Length: 3757
Keep-Alive: timeout=15, max=100
Connection: Keep-Alive
Content-Type: text/html

.............n......U.6..-.I.,.3.N...M.&m7`@@K..F"U.........Or..>,..#.[7.X$.<<._......]..?%o........g=.4
\..N.uO.N....V.uO/......]w:.6.;M.F..{wl.`..Fq.4}.;G..q..T....v.Q...!.....}...1U7.q.}_...Z....y.&Spr..
1..u..n..i.....sif.@`.8d.....Gw...V..h.4.\...ho1.....4.=..~..0%.v$............!......9H....)A
s.-h.....K.*..u......2.[..6.MU.e.....b.....4...O../......#.j.........u...!..+...Y.
+@...2.u..x..f....Uc.a......?.?...."C?..MF.....J?...H.Q.h.u....s.>.../1S3...iv..f.E.3.{.&.`....y{|
vA^.;..4.0....~..*`6S.H.AD6.KG..j.u...)X.....vA G...]i....x.<}.....<?
p.fs.R.v....A....>..@.R...k.....>U....O..X..^....x..0ol..#.;...
+.....L.q<....2c0n.....j.v....J.>q..Q5ZS.".FV..l.}Xq.........X|.|...fV..Q......8.........7..gT.
\.40...f..T.....P*.Jm.`..nH.......4.}e.....
```

Entire conversation (246065 bytes)   ○ ASCII ○ EBCDIC ○ Hex Dump ○ C Arrays ● Raw

Figure 3-11: Wireshark's Follow TCP Stream window

To capture packets, choose **Capture ▸ Interfaces** and select the interface you want to use to collect packets. Options include using promiscuous mode or setting a capture filter.

WARNING *Wireshark is known to have many security vulnerabilities, so be sure to run it in a safe environment.*

Wireshark can help you to understand how malware is performing network communication by sniffing packets as the malware communicates. To use Wireshark for this purpose, connect to the Internet or simulate an Internet connection, and then start Wireshark's packet capture and run the malware. (You can use Netcat to simulate an Internet connection.)

Chapter 14 discusses protocol analysis and additional uses of Wireshark in more detail.

Using INetSim

INetSim is a free, Linux-based software suite for simulating common Internet services. The easiest way to run INetSim if your base operating system is Microsoft Windows is to install it on a Linux virtual machine and set it up on the same virtual network as your malware analysis virtual machine.

INetSim is the best free tool for providing fake services, allowing you to analyze the network behavior of unknown malware samples by emulating services such as HTTP, HTTPS, FTP, IRC, DNS, SMTP, and others. Listing 3-3 displays all services that INetSim emulates by default, all of which (including the default ports used) are shown here as the program is starting up.

```
 * dns 53/udp/tcp - started (PID 9992)
 * http 80/tcp - started (PID 9993)
 * https 443/tcp - started (PID 9994)
 * smtp 25/tcp - started (PID 9995)
 * irc 6667/tcp - started (PID 10002)
 * smtps 465/tcp - started (PID 9996)
 * ntp 123/udp - started (PID 10003)
 * pop3 110/tcp - started (PID 9997)
 * finger 79/tcp - started (PID 10004)
 * syslog 514/udp - started (PID 10006)
 * tftp 69/udp - started (PID 10001)
 * pop3s 995/tcp - started (PID 9998)
 * time 37/tcp - started (PID 10007)
 * ftp 21/tcp - started (PID 9999)
 * ident 113/tcp - started (PID 10005)
 * time 37/udp - started (PID 10008)
 * ftps 990/tcp - started (PID 10000)
 * daytime 13/tcp - started (PID 10009)
 * daytime 13/udp - started (PID 10010)
 * echo 7/tcp - started (PID 10011)
 * echo 7/udp - started (PID 10012)
 * discard 9/udp - started (PID 10014)
```

```
  * discard 9/tcp - started (PID 10013)
  * quotd 17/tcp - started (PID 10015)
  * quotd 17/udp - started (PID 10016)
  * chargen 19/tcp - started (PID 10017)
  * dummy 1/udp - started (PID 10020)
  * chargen 19/udp - started (PID 10018)
  * dummy 1/tcp - started (PID 10019)
```

Listing 3-3: INetSim default emulated services

INetSim does its best to look like a real server, and it has many easily configurable features to ensure success. For example, by default, it returns the banner of Microsoft IIS web server if is it scanned.

Some of INetSim's best features are built into its HTTP and HTTPS server simulation. For example, INetSim can serve almost any file requested. For example, if a piece of malware requests a JPEG from a website to continue its operation, INetSim will respond with a properly formatted JPEG. Although that image might not be the file your malware is looking for, the server does not return a 404 or another error, and its response, even if incorrect, can keep the malware running.

INetSim can also record all inbound requests and connections, which you'll find particularly useful for determining whether the malware is connected to a standard service or to see the requests it is making. And INetSim is extremely configurable. For example, you can set the page or item returned after a request, so if you realize that your subject malware is looking for a particular web page before it will continue execution, you can provide that page. You can also modify the port on which various services listen, which can be useful if malware is using nonstandard ports.

And because INetSim is built with malware analysis in mind, it offers many unique features, such as its Dummy service, a feature that logs all data received from the client, regardless of the port. The Dummy service is most useful for capturing all traffic sent from the client to ports not bound to any other service module. You can use it to record all ports to which the malware connects and the corresponding data that is sent. At least the TCP handshake will complete, and additional data can be gathered.

Basic Dynamic Tools in Practice

All the tools discussed in this chapter can be used in concert to maximize the amount of information gleaned during dynamic analysis. In this section, we'll look at all the tools discussed in the chapter as we present a sample setup for malware analysis. Your setup might include the following:

1. Running procmon and setting a filter on the malware executable name and clearing out all events just before running.
2. Starting Process Explorer.
3. Gathering a first snapshot of the registry using Regshot.

4. Setting up your virtual network to your liking using INetSim and ApateDNS.
5. Setting up network traffic logging using Wireshark.

Figure 3-12 shows a diagram of a virtual network that can be set up for malware analysis. This virtual network contains two hosts: the malware analysis Windows virtual machine and the Linux virtual machine running INetSim. The Linux virtual machine is listening on many ports, including HTTPS, FTP, and HTTP, through the use of INetSim. The Windows virtual machine is listening on port 53 for DNS requests through the use of ApateDNS. The DNS server for the Windows virtual machine has been configured to localhost (127.0.0.1). ApateDNS is configured to redirect you to the Linux virtual machine (192.168.117.169).

If you attempt to browse to a website using the Windows virtual machine, the DNS request will be resolved by ApateDNS redirecting you to the Linux virtual machine. The browser will then perform a `GET` request over port 80 to the INetSim server listening on that port on the Linux virtual machine.

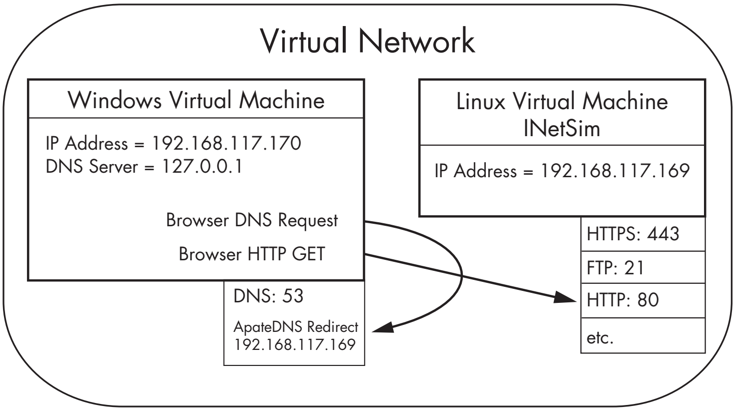
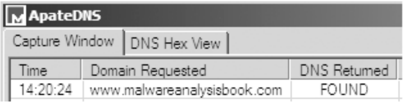


Figure 3-12: Example of a virtual network

Let's see how this setup would work in practice by examining the malware *msts.exe*. We complete our initial setup and then run *msts.exe* on our malware analysis virtual machine. After some time, we stop event capture with procmon and run a second snapshot with Regshot. At this point we begin analysis as follows:

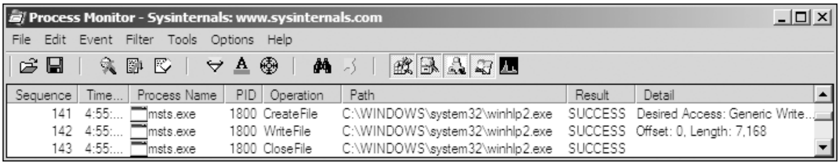
1. Examine ApateDNS to see if DNS requests were performed. As shown in Figure 3-13, we notice that the malware performed a DNS request for *www.malwareanalysisbook.com*.



Time	Domain Requested	DNS Returned
14:20:24	www.malwareanalysisbook.com	FOUND

Figure 3-13: ApateDNS request for www.malwareanalysisbook.com

2. Review the procmon results for file system modifications. In the procmon results shown in Figure 3-14, we see `CreateFile` and `WriteFile` (sequence numbers 141 and 142) operations for *C:\WINDOWS\system32\winhlp2.exe*. Upon further investigation, we compare *winhlp2.exe* to *msts.exe* and see that they are identical. We conclude that the malware copies itself to that location.


Process Monitor - Sysinternals: www.sysinternals.com
File Edit Event Filter Tools Options Help

Sequence	Time...	Process Name	PID	Operation	Path	Result	Detail
141	4:55:...	msts.exe	1800	CreateFile	C:\WINDOWS\system32\winhlp2.exe	SUCCESS	Desired Access: Generic Write...
142	4:55:...	msts.exe	1800	WriteFile	C:\WINDOWS\system32\winhlp2.exe	SUCCESS	Offset: 0, Length: 7,168
143	4:55:...	msts.exe	1800	CloseFile	C:\WINDOWS\system32\winhlp2.exe	SUCCESS	

Figure 3-14: Procmon output with the msts.exe *filter set*

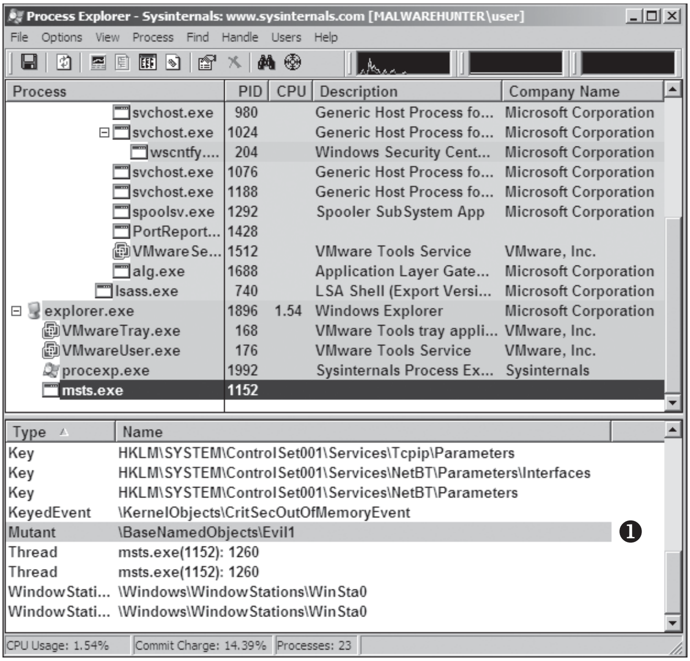
3. Compare the two snapshots taken with Regshot to identify changes. Reviewing the Regshot results, shown next, we see that the malware installed the autorun registry value `winhlp` at `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run` location. The data written to that value is where the malware copied itself (*C:\WINDOWS\system32\winhlp2.exe*), and that newly copied binary will execute upon system reboot.

```
Values added:3
----------------------------------
HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\winhlp: C:\WINDOWS\system32\winhlp2.exe
```

4. Use Process Explorer to examine the process to determine whether it creates mutexes or listens for incoming connections. The Process Explorer output in Figure 3-15 shows that *msts.exe* creates a mutex (also known as a *mutant*) named `Evil1` ❶. We discuss mutexes in depth in Chapter 7, but you should know that *msts.exe* likely created the mutex to ensure that only one version of the malware is running at a time. Mutexes can provide an excellent fingerprint for malware if they are unique enough.

5. Review the INetSim logs for requests and attempted connections on standard services. The first line in the INetSim logs (shown next) tells us that the malware communicates over port 443, though not with standard Secure Sockets Layer (SSL), as shown next in the reported errors at ❶.

```
  [2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] connect
  [2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043]
❶ Error setting up SSL:  SSL accept attempt failed with unknown error
  Error:140760FC:SSL routines:SSL23_GET_CLIENT_HELLO:unknown protocol
  [2010-X] [15013] [https 443/tcp 15199] [192.168.117.128:1043] disconnect
```


Process Explorer - Sysinternals: www.sysinternals.com [MALWAREHUNTER\user]
File Options View Process Find Handle Users Help
❶

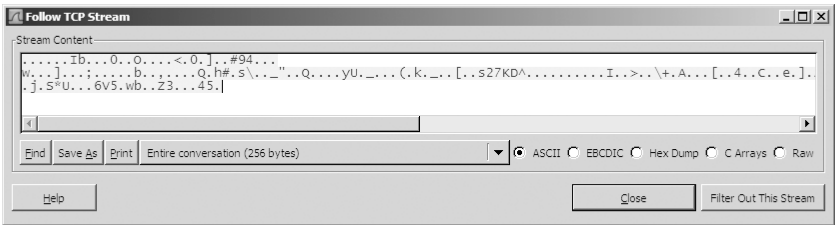
Process	PID	CPU	Description	Company Name
svchost.exe	980		Generic Host Process fo...	Microsoft Corporation
svchost.exe	1024		Generic Host Process fo...	Microsoft Corporation
wscntfy....	204		Windows Security Cent...	Microsoft Corporation
svchost.exe	1076		Generic Host Process fo...	Microsoft Corporation
svchost.exe	1188		Generic Host Process fo...	Microsoft Corporation
spoolsv.exe	1292		Spooler SubSystem App	Microsoft Corporation
PortReport...	1428			
VMwareSe...	1512		VMware Tools Service	VMware, Inc.
alg.exe	1688		Application Layer Gate...	Microsoft Corporation
lsass.exe	740		LSA Shell (Export Versi...	Microsoft Corporation
explorer.exe	1896	1.54	Windows Explorer	Microsoft Corporation
VMwareTray.exe	168		VMware Tools tray appli...	VMware, Inc.
VMwareUser.exe	176		VMware Tools Service	VMware, Inc.
procexp.exe	1992		Sysinternals Process Ex...	Sysinternals
msts.exe	1152			

Type	Name
Key	HKLM\SYSTEM\ControlSet001\Services\Tcpip\Parameters
Key	HKLM\SYSTEM\ControlSet001\Services\NetBT\Parameters\Interfaces
Key	HKLM\SYSTEM\ControlSet001\Services\NetBT\Parameters
KeyedEvent	\KernelObjects\CritSecOutOfMemoryEvent
Mutant	\BaseNamedObjects\Evil1
Thread	msts.exe(1152): 1260
Thread	msts.exe(1152): 1260
WindowStati...	\Windows\WindowStations\WinSta0
WindowStati...	\Windows\WindowStations\WinSta0

CPU Usage: 1.54% Commit Charge: 14.39% Processes: 23

Figure 3-15: Process Explorer's examination of an active msts.exe *process*

6. Review the Wireshark capture for network traffic generated by the malware. By using INetSim while capturing with Wireshark, we can capture the TCP handshake and the initial data packets sent by the malware. The contents of the TCP stream sent over port 443, as shown in Figure 3-16, shows random ACSII data, which is often indicative of a custom protocol. When this happens, your best bet is to run the malware several more times to look for any consistency in the initial packets of the connection. (The resulting information could be used to draft a network-based signature, skills that we explore in Chapter 14.)



```
......Ib...0..o....<.0.]..#94...
w...]...;.....b..,....Q.h#.s\.._"..Q....yU._...(.k._..[..s27KD^.........I..>..\+.A...[..4..C..e.]..
.j.S*U...6V5.wb..Z3...45.
```

Entire conversation (256 bytes) ● ASCII ○ EBCDIC ○ Hex Dump ○ C Arrays ○ Raw

Figure 3-16: Wireshark showing the custom network protocol

Conclusion

Basic dynamic analysis of malware can assist and confirm your basic static analysis findings. Most of the tools described in this chapter are free and easy to use, and they provide considerable detail.

However, basic dynamic analysis techniques have their deficiencies, so we won't stop here. For example, to understand the networking component in the *msts.exe* fully, you would need to reverse-engineer the protocol to determine how best to continue your analysis. The next step is to perform advanced static analysis techniques with disassembly and dissection at the binary level, which is discussed in the next chapter.

LABS

Lab 3-1

Analyze the malware found in the file *Lab03-01.exe* using basic dynamic analysis tools.

Questions

1. What are this malware's imports and strings?
2. What are the malware's host-based indicators?
3. Are there any useful network-based signatures for this malware? If so, what are they?

Lab 3-2

Analyze the malware found in the file *Lab03-02.dll* using basic dynamic analysis tools.

Questions

1. How can you get this malware to install itself?
2. How would you get this malware to run after installation?
3. How can you find the process under which this malware is running?
4. Which filters could you set in order to use procmon to glean information?
5. What are the malware's host-based indicators?
6. Are there any useful network-based signatures for this malware?

Lab 3-3

Execute the malware found in the file *Lab03-03.exe* while monitoring it using basic dynamic analysis tools in a safe environment.

Questions

1. What do you notice when monitoring this malware with Process Explorer?
2. Can you identify any live memory modifications?
3. What are the malware's host-based indicators?
4. What is the purpose of this program?

Lab 3-4

Analyze the malware found in the file *Lab03-04.exe* using basic dynamic analysis tools. (This program is analyzed further in the Chapter 9 labs.)

Questions

1. What happens when you run this file?
2. What is causing the roadblock in dynamic analysis?
3. Are there other ways to run this program?

PART 2

ADVANCED STATIC ANALYSIS

4

A CRASH COURSE IN X86 DISASSEMBLY

As discussed in previous chapters, basic static and dynamic malware analysis methods are good for initial triage, but they do not provide enough information to analyze malware completely.

Basic static techniques are like looking at the outside of a body during an autopsy. You can use static analysis to draw some preliminary conclusions, but more in-depth analysis is required to get the whole story. For example, you might find that a particular function is imported, but you won't know how it's used or whether it's used at all.

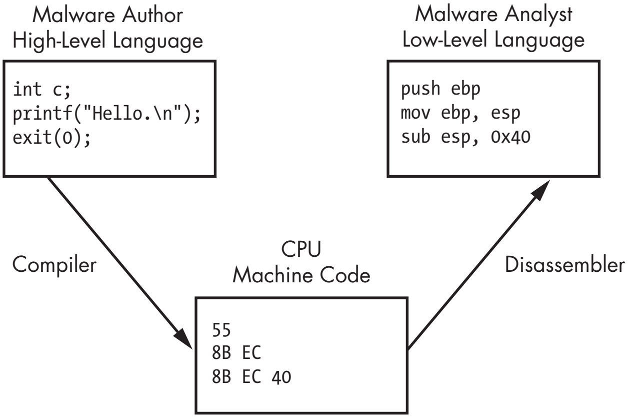
Basic dynamic techniques also have shortcomings. For example, basic dynamic analysis can tell you how your subject malware responds when it receives a specially designed packet, but you can learn the format of that packet only by digging deeper. That's where disassembly comes in, as you'll learn in this chapter.

Disassembly is a specialized skill that can be daunting to those new to programming. But don't be discouraged; this chapter will give you a basic understanding of disassembly to get you off on the right foot.

Levels of Abstraction

In traditional computer architecture, a computer system can be represented as several *levels of abstraction* that create a way of hiding the implementation details. For example, you can run the Windows OS on many different types of hardware, because the underlying hardware is abstracted from the OS.

Figure 4-1 shows the three coding levels involved in malware analysis. Malware authors create programs at the high-level language level and use a compiler to generate machine code to be run by the CPU. Conversely, malware analysts and reverse engineers operate at the low-level language level; we use a disassembler to generate assembly code that we can read and analyze to figure out how a program operates.



```
int c;
printf("Hello.\n");
exit(0);
```

```
push ebp
mov ebp, esp
sub esp, 0x40
```

```
55
8B EC
8B EC 40
```

Figure 4-1: Code level examples

Figure 4-1 shows a simplified model, but computer systems are generally described with the following six different levels of abstraction. We list these levels starting from the bottom. Higher levels of abstraction are placed near the top with more specific concepts underneath, so the lower you get, the less portable the level will be across computer systems.

Hardware The hardware level, the only physical level, consists of electrical circuits that implement complex combinations of logical operators such as XOR, AND, OR, and NOT gates, known as *digital logic*. Because of its physical nature, hardware cannot be easily manipulated by software.

Microcode The microcode level is also known as *firmware*. Microcode operates only on the exact circuitry for which it was designed. It contains microinstructions that translate from the higher machine-code level to provide a way to interface with the hardware. When performing malware analysis, we usually don't worry about the microcode because it is often specific to the computer hardware for which it was written.

Machine code The machine code level consists of *opcodes*, hexadecimal digits that tell the processor what you want it to do. Machine code is typically implemented with several microcode instructions so that the underlying hardware can execute the code. Machine code is created when a computer program written in a high-level language is compiled.

Low-level languages A low-level language is a human-readable version of a computer architecture's instruction set. The most common low-level language is assembly language. Malware analysts operate at the low-level languages level because the machine code is too difficult for a human to comprehend. We use a disassembler to generate low-level language text, which consists of simple mnemonics such as `mov` and `jmp`. Many different dialects of assembly language exist, and we'll explore each in turn.

NOTE *Assembly is the highest level language that can be reliably and consistently recovered from machine code when high-level language source code is not available.*

High-level languages Most computer programmers operate at the level of high-level languages. High-level languages provide strong abstraction from the machine level and make it easy to use programming logic and flow-control mechanisms. High-level languages include C, C++, and others. These languages are typically turned into machine code by a compiler through a process known as *compilation*.

Interpreted languages Interpreted languages are at the top level. Many programmers use interpreted languages such as C#, Perl, .NET, and Java. The code at this level is not compiled into machine code; instead, it is translated into bytecode. *Bytecode* is an intermediate representation that is specific to the programming language. Bytecode executes within an *interpreter*, which is a program that translates bytecode into executable machine code on the fly at runtime. An interpreter provides an automatic level of abstraction when compared to traditional compiled code, because it can handle errors and memory management on its own, independent of the OS.

Reverse-Engineering

When malware is stored on a disk, it is typically in *binary* form at the machine code level. As discussed, machine code is the form of code that the computer can run quickly and efficiently. When we disassemble malware (as shown in Figure 4-1), we take the malware binary as input and generate assembly language code as output, usually with a *disassembler*. (Chapter 5 discusses the most popular disassembler, IDA Pro.)

Assembly language is actually a class of languages. Each assembly dialect is typically used to program a single family of microprocessors, such as x86, x64, SPARC, PowerPC, MIPS, and ARM. x86 is by far the most popular architecture for PCs.