

# Chapter 1. Malware Analysis Primer

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

## The Goals of Malware Analysis

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

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

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

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

higher detection rate and fewer false positives.

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

# Malware Analysis Techniques

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

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

## Basic Static Analysis

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

## Basic Dynamic Analysis

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

## Advanced Static Analysis

Advanced static analysis consists of reverse-engineering the malware's internals by loading the executable into a disassembler and looking at the program instructions in order to discover what the program does. The instructions are executed by the

CPU, so advanced static analysis tells you exactly what the program does. However, advanced static analysis has a steeper learning curve than basic static analysis and requires specialized knowledge of disassembly, code constructs, and Windows operating system concepts, all of which you'll learn in this book.

## **Advanced Dynamic Analysis**

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

# Types of Malware

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

- **Backdoor.** Malicious code that installs itself onto a computer to allow the attacker access. Backdoors usually let the attacker connect to the computer with little or no authentication and execute commands on the local system.
- **Botnet.** Similar to a backdoor, in that it allows the attacker access to the system, but all computers infected with the same botnet receive the same instructions from a single command-and-control server.
- **Downloader.** Malicious code that exists only to download other malicious code. Downloaders are commonly installed by attackers when they first gain access to a system. The downloader program will download and install additional malicious code.
- **Information-stealing malware.** Malware that collects information from a victim's computer and usually sends it to the attacker. Examples include sniffers, password hash grabbers, and keyloggers. This malware is typically used to gain access to online accounts such as email or online banking.
- **Launcher.** Malicious program used to launch other malicious programs. Usually, launchers use nontraditional techniques to launch other malicious programs in order to ensure stealth or greater access to a system.
- **Rootkit.** Malicious code designed to conceal the existence of other code. Rootkits are usually paired with other malware, such as a backdoor, to allow remote access to the attacker and make the code difficult for the victim to detect.
- **Scareware.** Malware designed to frighten an infected user into buying something. It usually has a user interface that makes it look like an antivirus or other security program. It informs users that there is malicious code on their system and that the only way to get rid of it is to buy their "software," when in

reality, the software it's selling does nothing more than remove the scareware.

- **Spam-sending malware.** Malware that infects a user's machine and then uses that machine to send spam. This malware generates income for attackers by allowing them to sell spam-sending services.
- **Worm or virus.** Malicious code that can copy itself and infect additional computers.

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

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

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

# General Rules for Malware Analysis

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

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

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

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

## **Part I. Basic Analysis**



# Chapter 2. 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 4](#).

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

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

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

## Antivirus Scanning: A Useful First Step

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

Because the various antivirus programs use different signatures and heuristics, it's useful to run several different antivirus programs against the same piece of

suspected malware. Websites such as VirusTotal (<http://www.virustotal.com/>) allow you to upload a file for scanning by multiple antivirus engines. VirusTotal generates a report that provides the total number of engines that marked the file as malicious, the malware name, and, if available, additional information about the malware.

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

```
C:\>md5deep c:\WINDOWS\system32\sol.exe
373e7a863a1a345c60edb9e20ec3231 c:\WINDOWS\system32\sol.exe
```

The GUI-based WinMD5 calculator, shown in **Figure 2-1**, can calculate and display hashes for several files at a time.

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

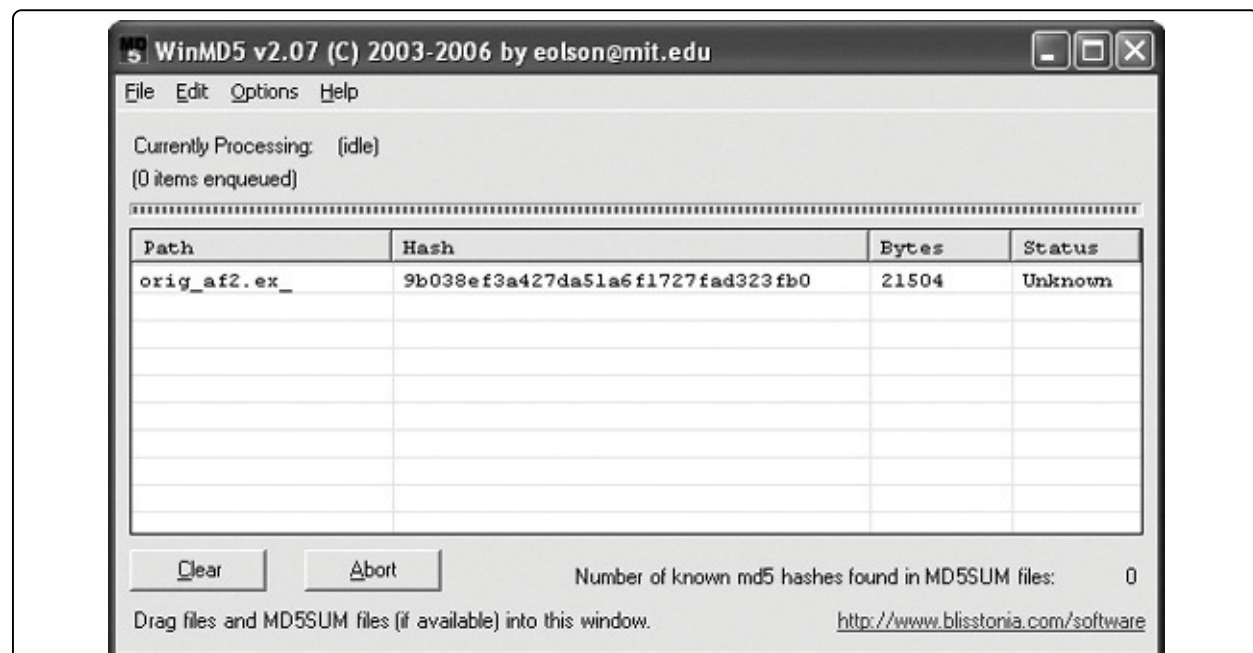


Figure 2-1. Output of WinMD5

## 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/ic4pIL>), 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 2-2** 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.

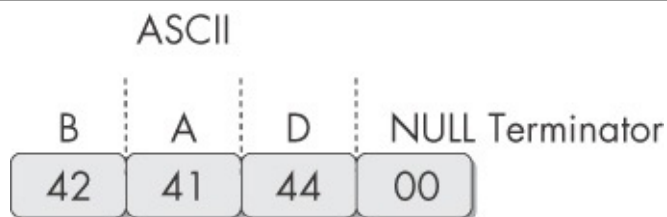
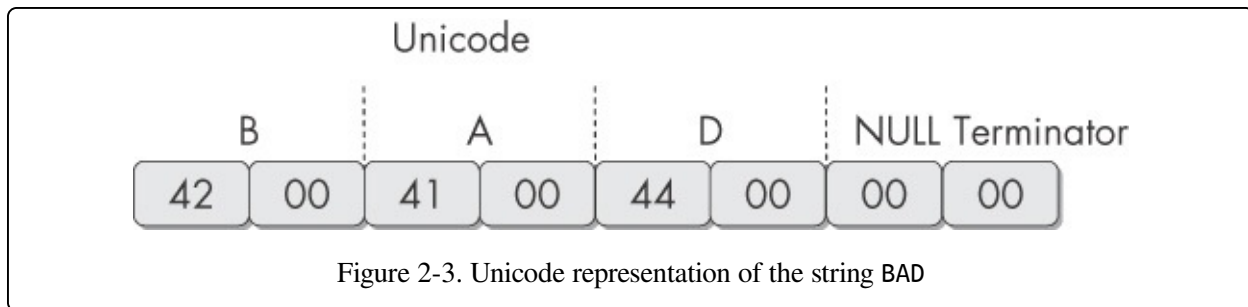


Figure 2-2. ASCII representation of the string BAD

**Figure 2-3** 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.



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

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

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

```
C:>strings bp6.ex_
VP3
VW3
t$@
D$4
99.124.22.1 4
e-@
GetLayout 1
GDI32.DLL 3
SetLayout 2
M}C
Mail system DLL is invalid.!Send Mail failed to send message. 5
```

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

On the other hand, the strings GetLayout at 1 and SetLayout at 2 are Windows functions used by the Windows graphics library. We can easily identify these as meaningful strings because Windows function names normally begin with a capital letter and subsequent words also begin with a capital letter.

GDI32.DLL at 3 is meaningful because it's the name of a common Windows dynamic link library (DLL) used by graphics programs. (DLL files contain executable code that is shared among multiple applications.) As you might imagine, the number 99.124.22.1 at 4 is an IP address—most likely one that the malware will use in some fashion.

Finally, at 5, Mail system DLL is invalid.!Send Mail failed to send message. is an error message. Often, the most useful information obtained by running Strings is found in error messages. This particular message reveals two things: The subject malware sends messages (probably through email), and it depends on a mail system DLL. This information suggests that we might want to check email logs for suspicious traffic, and that another DLL (Mail system DLL) might be associated with this particular malware. Note that the missing DLL itself is not necessarily malicious; malware often uses legitimate libraries and DLLs to further its goals.

# Packed and Obfuscated Malware

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

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

## NOTE

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

## Packing Files

When the packed program is run, a small wrapper program also runs to decompress the packed file and then run the unpacked file, as shown in [Figure 2-4](#). When a packed program is analyzed statically, only the small wrapper program can be dissected. ([Chapter 19](#) discusses packing and unpacking in more detail.)

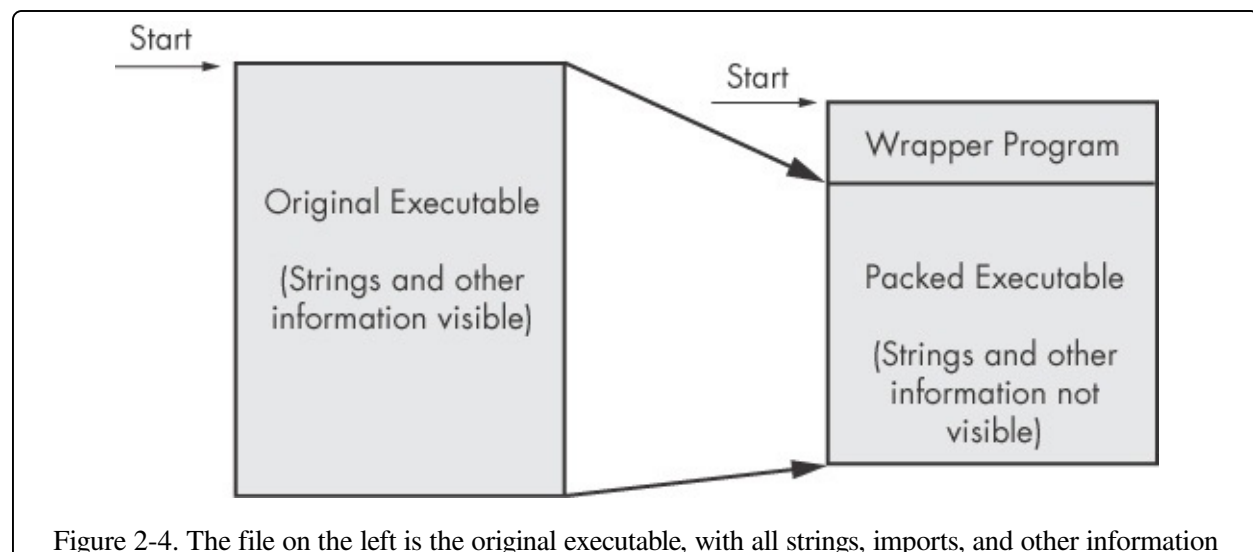


Figure 2-4. The file on the left is the original executable, with all strings, imports, and other information



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 2-5** shows information about the orig\_af2.ex\_ file as reported by PEiD.

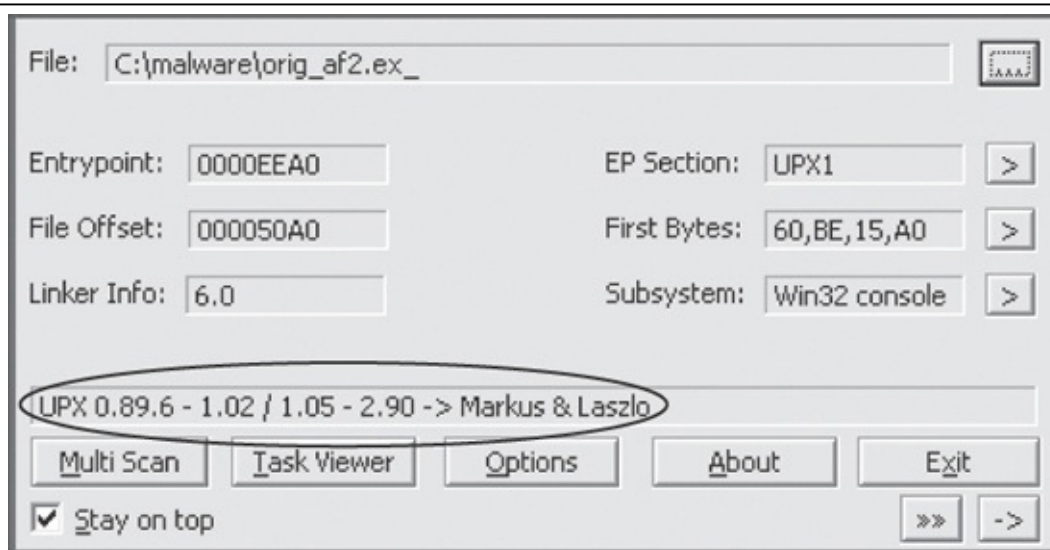


Figure 2-5. The PEiD program

### 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 19**.)

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 19**, 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 3](#) 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 2-6** shows the Dependency Walker's analysis of `SERVICES.EX_1`. The far left pane at **2** shows the program as well as the DLLs being imported, namely `KERNEL32.DLL` and `WS2_32.DLL`.

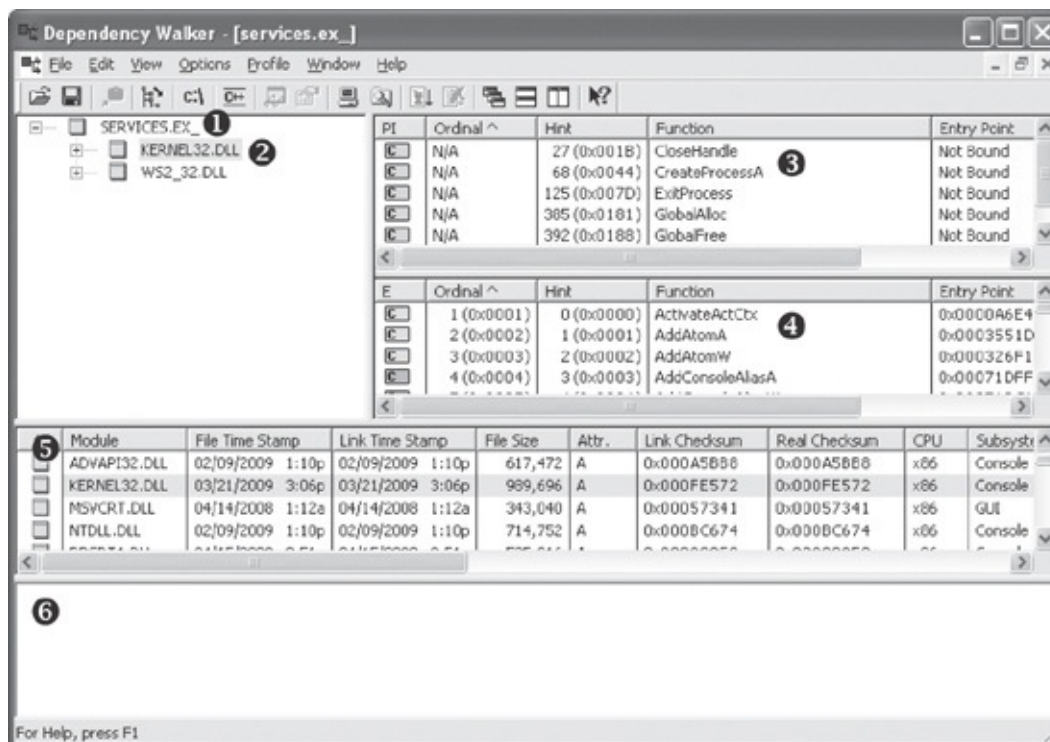


Figure 2-6. The Dependency Walker program

Clicking KERNEL32.DLL shows its imported functions in the upper-right pane at 3. 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 4 lists all functions in KERNEL32.DLL that can be imported—information that is not particularly useful to us. Notice the column in panes 3 and 4 labeled Ordinal. Executables can import functions by ordinal instead of name. When importing a function by ordinal, the name of the function never appears in the original executable, and it can be harder for an analyst to figure out which function is being used. When malware imports a function by ordinal, you can find out which function is being imported by looking up the ordinal value in the pane at 4.

The bottom two panes (5 and 6) list additional information about the versions of DLLs that would be loaded if you ran the program and any reported errors, respectively.

A program's DLLs can tell you a lot about its functionality. For example, [Table 2-1](#)

lists common DLLs and what they tell you about an application.

Table 2-1. Common DLLs

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

### FUNCTION NAMING CONVENTIONS

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

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

## Imported Functions

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


## Exported Functions

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

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

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

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

You can view export information using the Dependency Walker program discussed in [Exploring Dynamically Linked Functions with Dependency Walker](#). For a list of exported functions, click the name of the file you want to examine. Referring back to [Figure 2-6](#), window  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 2-2 shows an abridged list of functions imported by PotentialKeylogger.exe, as collected using Dependency Walker. Because we see so many imports, we can immediately conclude that this file is not packed.

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

Kernel32.dll	User32.dll	User32.dll (continued)
CreateDirectoryW	BeginDeferWindowPos	ShowWindow
CreateFileW	CallNextHookEx	ToUnicodeEx
CreateThread	CreateDialogParamW	TrackPopupMenu
DeleteFileW	CreateWindowExW	TrackPopupMenuEx
ExitProcess	DefWindowProcW	TranslateMessage
FindClose	DialogBoxParamW	UnhookWindowsHookEx
FindFirstFileW	EndDialog	UnregisterClassW
FindNextFileW	GetMessageW	UnregisterHotKey
GetCommandLineW	GetSystemMetrics	
GetCurrentProcess	GetWindowLongW	GDI32.dll
GetCurrentThread	GetWindowRect	GetStockObject
GetFileSize	GetWindowTextW	SetBkMode
GetModuleHandleW	InvalidateRect	SetTextColor
GetProcessHeap	IsDlgButtonChecked	
GetShortPathNameW	IsWindowEnabled	Shell32.dll
HeapAlloc	LoadCursorW	CommandLineToArgvW

HeapFree	LoadIconW	SHChangeNotify
IsDebuggerPresent	LoadMenuW	SHGetFolderPathW
MapViewOfFile	MapVirtualKeyW	ShellExecuteExW
<b>OpenProcess</b>	MapWindowPoints	ShellExecuteW
<b>ReadFile</b>	MessageBoxW	
SetFilePointer	<b>RegisterClassExW</b>	<b>Advapi32.dll</b>
<b>WriteFile</b>	<b>RegisterHotKey</b>	RegCloseKey
	SendMessageA	RegDeleteValueW
	SetClipboardData	RegOpenCurrentUser
	SetDlgItemTextW	RegOpenKeyExW
	<b>SetWindowTextW</b>	RegQueryValueExW
	<b>SetWindowsHookExW</b>	RegSetValueExW

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

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

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

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

The imports from Kernel32.dll in [Table 2-2](#) tell us that this software can open and

manipulate processes (such as `OpenProcess`, `GetCurrentProcess`, and `GetProcessHeap`) and files (such as `ReadFile`, `CreateFile`, and `WriteFile`). The functions `FindFirstFile` and `FindNextFile` are particularly interesting ones that we can use to search through directories.

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

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

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

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

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

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

event. The documentation for `SetWindowsHookEx` further explains that this function will be called when certain low-level keyboard events occur.

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

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

## PackedProgram.exe: A Dead End

Table 2-3 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 2-3. 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 4](#)) or unpacking (covered in [Chapter 19](#)).

# 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 2-4](#)).
- **.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 7](#).)
- **.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 2-4](#) lists the most common you'll encounter.

Table 2-4. 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 PView

The PE file format stores interesting information within its header. We can use the PView tool to browse through the information, as shown in [Figure 2-7](#).

In the figure, the left pane at **1** 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 **2**, contains basic information about the file. The Time Date Stamp description at **3** tells us when this executable was compiled, which can be very useful in malware analysis and incident response. For example, an old compile time suggests that this is an older attack, and antivirus programs might contain signatures for the malware. A new compile time suggests the reverse.

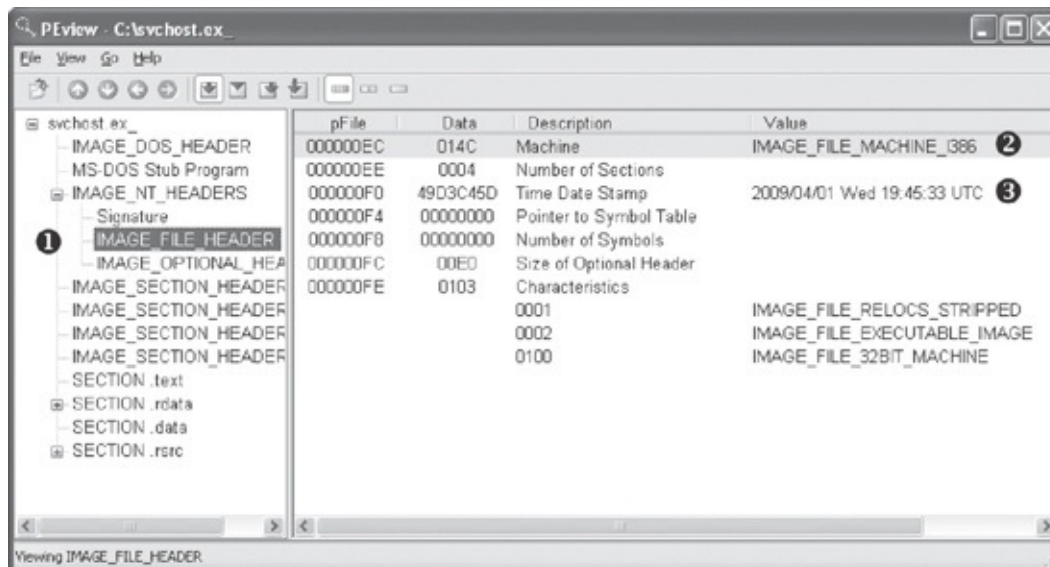


Figure 2-7. Viewing the IMAGE\_FILE\_HEADER in the PEview program

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

The IMAGE\_OPTIONAL\_HEADER section includes several important pieces of information. The Subsystem description indicates whether this is a console or GUI program. Console programs have the value IMAGE\_SUBSYSTEM\_WINDOWS\_CUI and run inside a command window. GUI programs have the value IMAGE\_SUBSYSTEM\_WINDOWS\_GUI and run within the Windows system. Less common subsystems such as Native or Xbox also are used.

The most interesting information comes from the section headers, which are in IMAGE\_SECTION\_HEADER, as shown in Figure 2-8. These headers are used to describe each section of a PE file. The compiler generally creates and names the sections of an executable, and the user has little control over these names. As a result, the sections are usually consistent from executable to executable (see Table 2-4), and any deviations may be suspicious.

For example, in Figure 2-8, Virtual Size at ① tells us how much space is allocated for a section during the loading process. The Size of Raw Data at ② shows how big the section is on disk. These two values should usually be equal, because data



should take up just as much space on the disk as it does in memory. Small differences are normal, and are due to differences between alignment in memory and on disk.

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

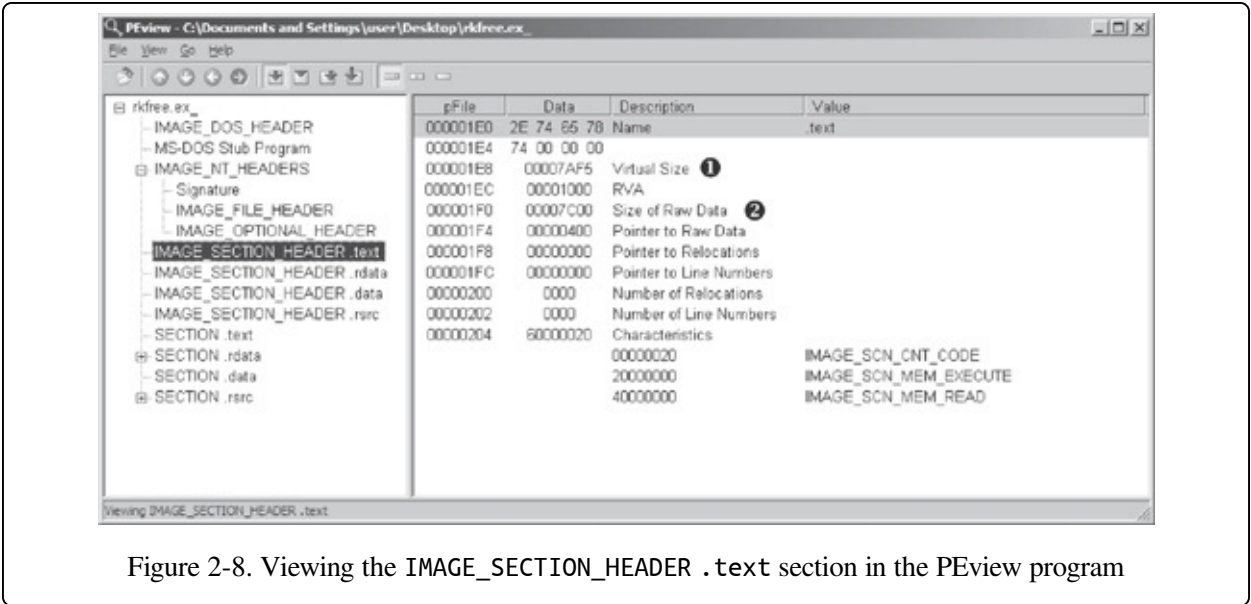


Figure 2-8. Viewing the IMAGE\_SECTION\_HEADER .text section in the PView program

Table 2-5 shows the sections from PotentialKeylogger.exe. As you can see, the `.text`, `.rdata`, and `.rsrc` sections each has a Virtual Size and Size of Raw Data value of about the same size. The `.data` section may seem suspicious because it has a much larger virtual size than raw data size, but this is normal for the `.data` section in Windows programs. But note that this information alone does not tell us that the program is not malicious; it simply shows that it is likely not packed and that the PE file header was generated by a compiler.

Table 2-5. Section Information for PotentialKeylogger.exe

Section	Virtual size	Size of raw data
.text	7AF5	7C00
.data	17A0	0200

.rdata	1AF5	1C00
.rsrc	72B8	7400

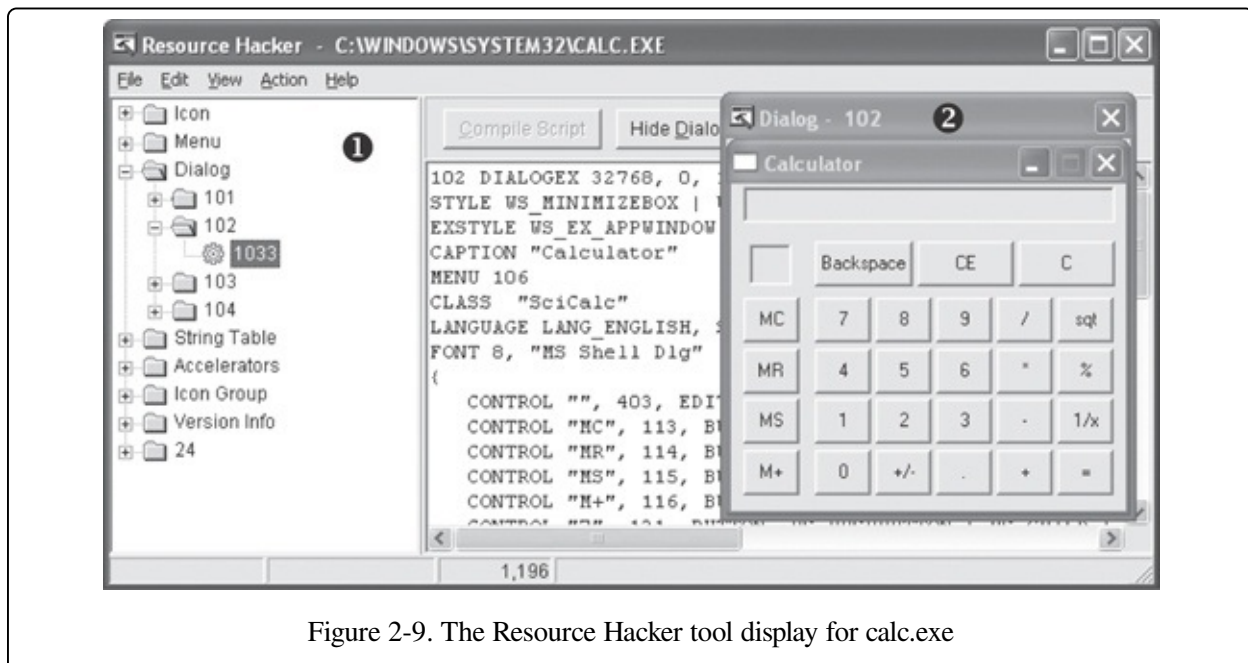
**Table 2-6** shows the sections from PackedProgram.exe. The sections in this file have a number of anomalies: The sections named `Dijfpds`, `.sdfuok`, and `Kijijl` are unusual, and the `.text`, `.data`, and `.rdata` sections are suspicious. The `.text` section has a Size of Raw Data value of 0, meaning that it takes up no space on disk, and its Virtual Size value is A000, which means that space will be allocated for the `.text` segment. This tells us that a packer will unpack the executable code to the allocated `.text` section.

Table 2-6. Section Information for PackedProgram.exe

Name	Virtual size	Size of raw data
.text	A000	0000
.data	3000	0000
.rdata	4000	0000
.rsrc	19000	3400
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 2-9** shows the Resource Hacker display for the Windows Calculator program, `calc.exe`.



The panel on the left shows all resources included in this executable. Each root folder shown in the left pane at **1** 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 2 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 2-9** 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-profile-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 2-7** reviews the key information that can be obtained from a PE header.

Table 2-7. 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

Q: 1. Upload the files to <http://www.VirusTotal.com/> and view the reports. Does either file match any existing antivirus signatures?

Q: 2. When were these files compiled?

Q: 3. Are there any indications that either of these files is packed or obfuscated? If so, what are these indicators?

Q: 4. Do any imports hint at what this malware does? If so, which imports are they?

Q: 5. Are there any other files or host-based indicators that you could look for on infected systems?

Q: 6. What network-based indicators could be used to find this malware on infected machines?

Q: 7. What would you guess is the purpose of these files?

---

## Lab 1-2

Analyze the file Lab01-02.exe.

### Questions

Q: 1. Upload the Lab01-02.exe file to <http://www.VirusTotal.com/>. Does it match any existing antivirus definitions?

Q: 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.

Q: 3. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?

Q: 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

Q: 1. Upload the Lab01-03.exe file to <http://www.VirusTotal.com/>. Does it match any existing antivirus definitions?

Q: 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.

Q: 3. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?

Q: 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

Q: 1. Upload the Lab01-04.exe file to <http://www.VirusTotal.com/>. Does it match any existing



antivirus definitions?

Q: 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.

Q: 3. When was this program compiled?

Q: 4. Do any imports hint at this program's functionality? If so, which imports are they and what do they tell you?

Q: 5. What host-or network-based indicators could be used to identify this malware on infected machines?

Q: 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?

# Chapter 3. 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 airgapped networks. These are isolated networks with machines that are disconnected from the Internet or any other networks to prevent the malware from spreading.

Airgapped 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.

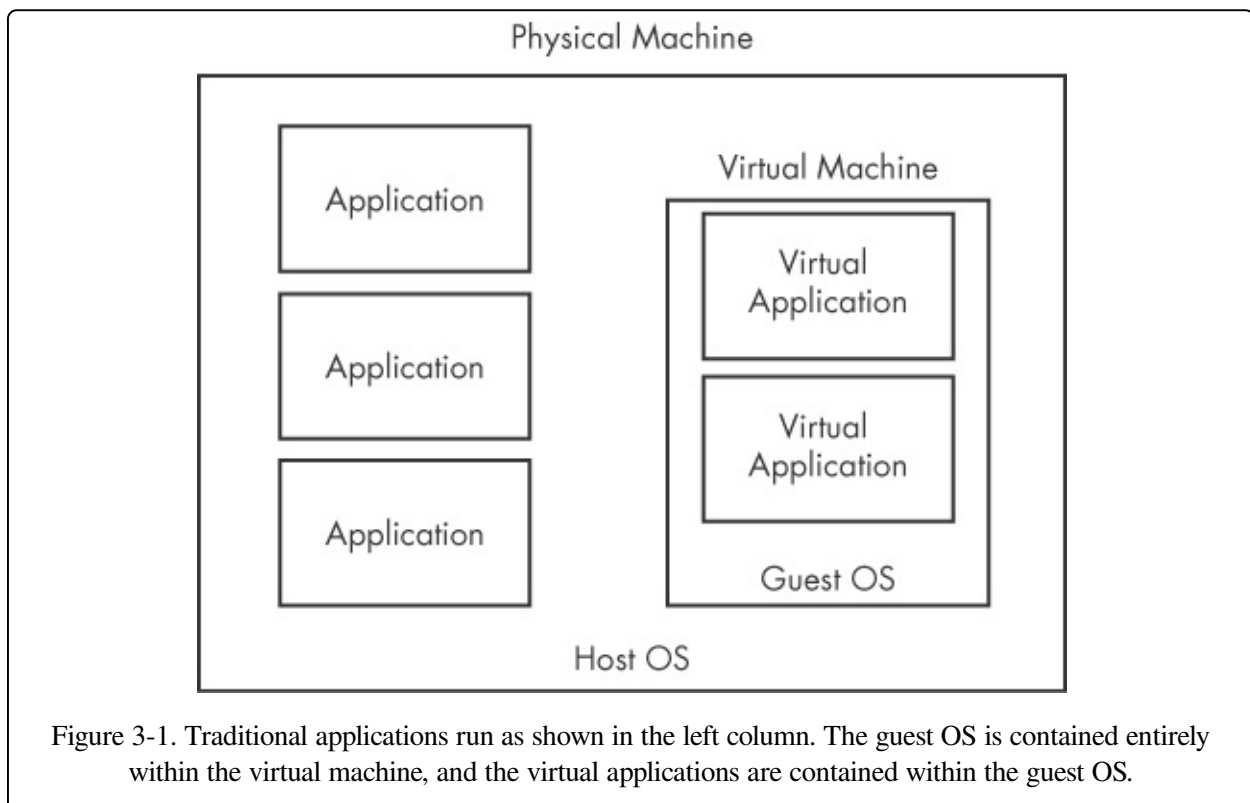
Another disadvantage to analyzing malware on physical rather than virtual machines is that malware can be difficult to remove. To avoid problems, most people who test malware on physical machines use a tool such as Norton Ghost to manage backup images of their operating systems (OSs), which they restore 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 3-1](#). 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.



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

There are many alternatives to VMware, such as Parallels, Microsoft Virtual PC, Microsoft Hyper-V, and Xen. These vary in host and guest OS support and

features. This book will focus on using VMware for virtualization, but if you prefer another virtualization tool, you should still find this discussion relevant.

# Creating Your Malware Analysis Machine

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

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

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

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

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

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

## Configuring VMware

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

to other computers.

When analyzing malware, you will probably want to observe the malware's network activity to help you understand the author's intention, to create signatures, or to exercise the program fully. VMware offers several networking options for virtual networking, as shown in **Figure 3-2** and discussed in the following sections.

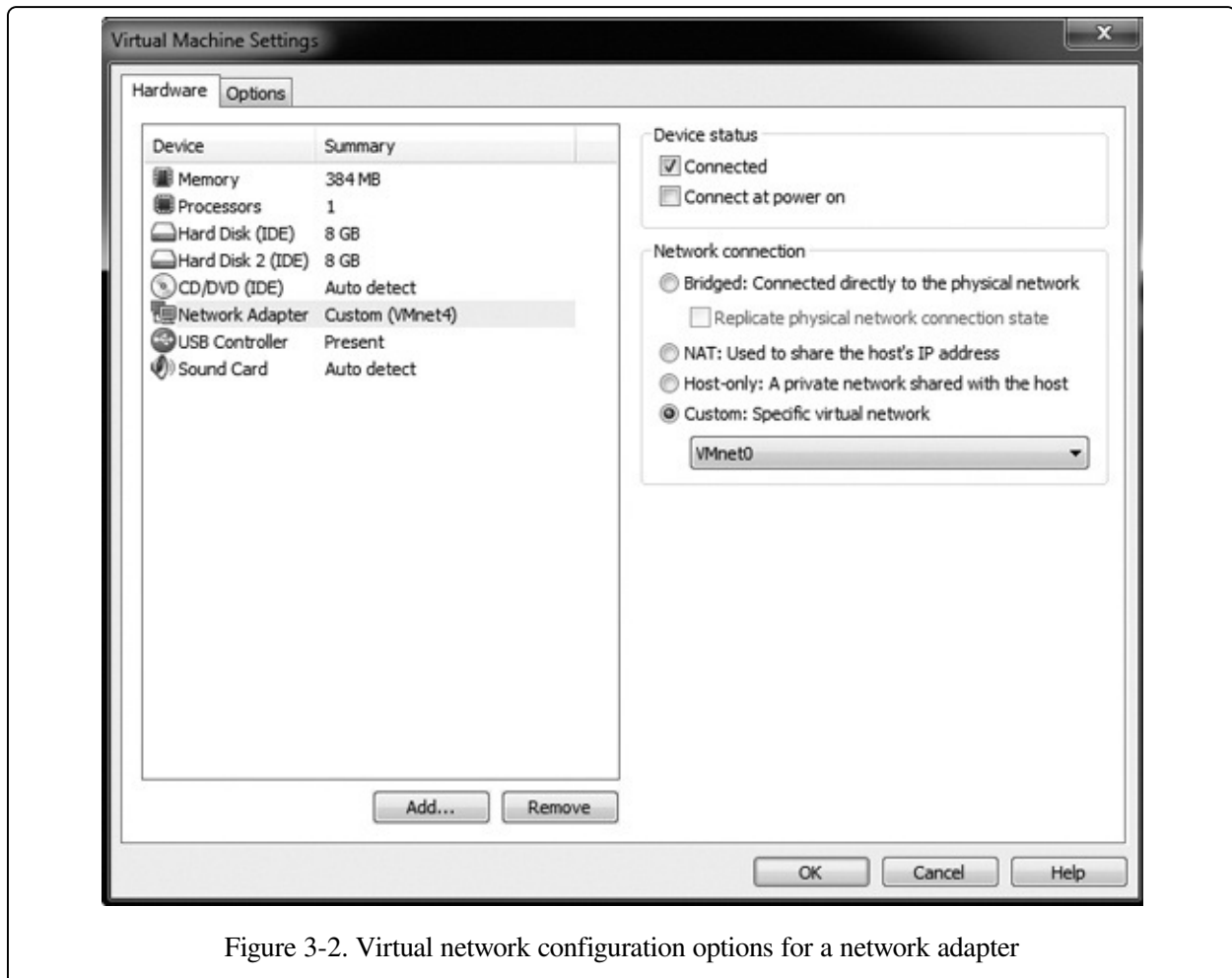


Figure 3-2. Virtual network configuration options for a network adapter

## Disconnecting the Network

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

Still, should you have reason to disconnect the network in VMware, you can do so either by removing the network adapter from the virtual machine or by

disconnecting the network adapter from the network by choosing **VM ► Removable Devices**.

You can also control whether a network adapter is connected automatically when the machine is turned on by checking the **Connect at power on** checkbox (see [Figure 3-2](#)).

## Setting Up Host-Only Networking

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

### NOTE

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

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

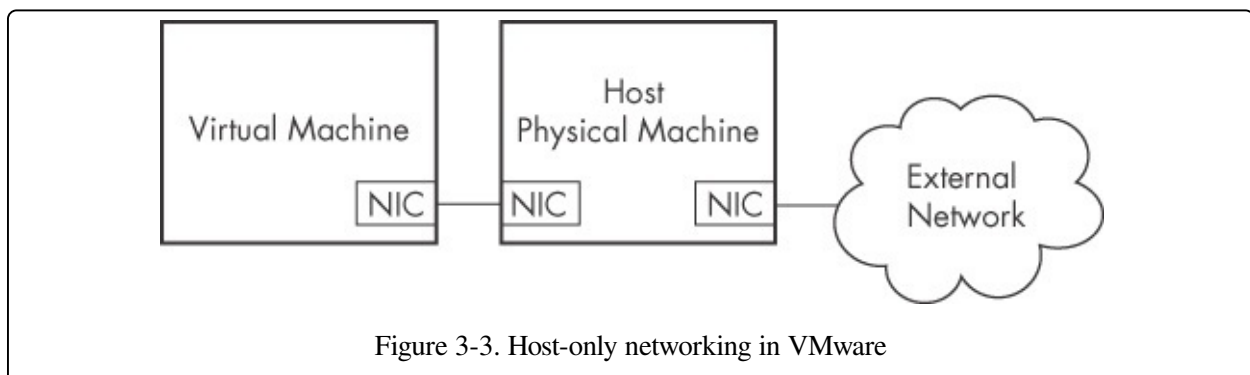


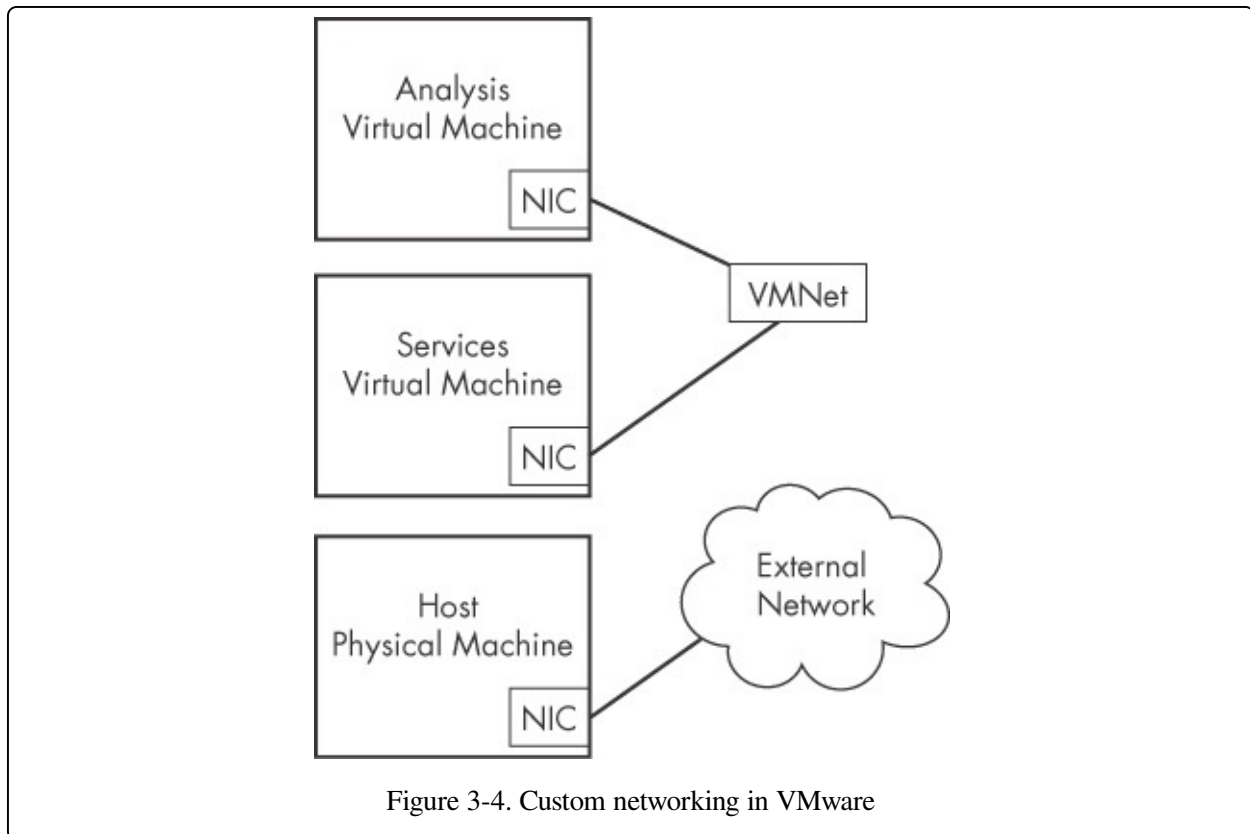
Figure 3-3. Host-only networking in VMware

## Using Multiple Virtual Machines

One last configuration combines the best of all options. It requires multiple virtual machines linked by a LAN but disconnected from the Internet and host machine,

so that the malware is connected to a network, but the network isn't connected to anything important.

**Figure 3-4** shows a custom configuration with two virtual machines connected to each other. In this configuration, one virtual machine is set up to analyze malware, and the second machine provides services. The two virtual machines are connected to the same VMNet virtual switch. In this case, the host machine is still connected to the external network, but not to the machine running the malware.



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



## Using Your Malware Analysis Machine

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

## Connecting Malware to the Internet

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

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

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

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

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

## Connecting and Disconnecting Peripheral Devices

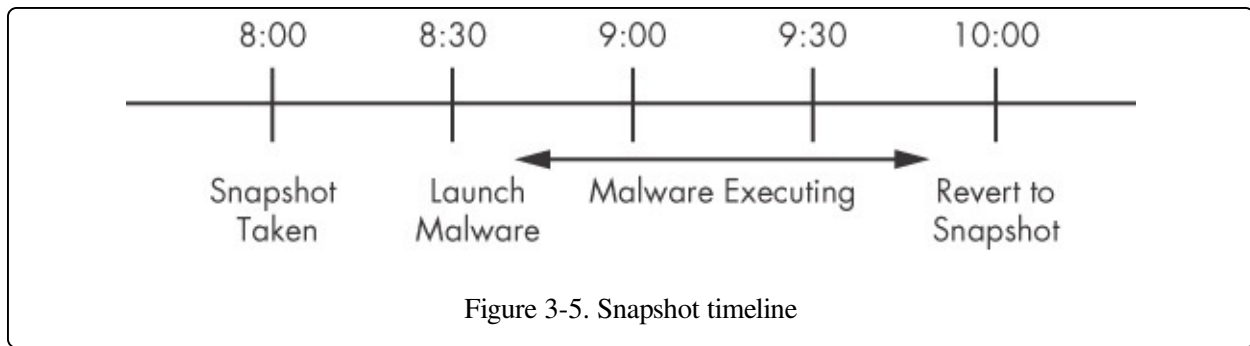
Peripheral devices, such as CD-ROMs and external USB storage drives, pose a particular problem for virtual machines. Most devices can be connected either to the physical machine or the virtual machine, but not both.

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

## Taking Snapshots

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

The timeline in **Figure 3-5** 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.



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 3-6](#). The two machine states are completely independent, and you can save as many snapshots as you have disk space.

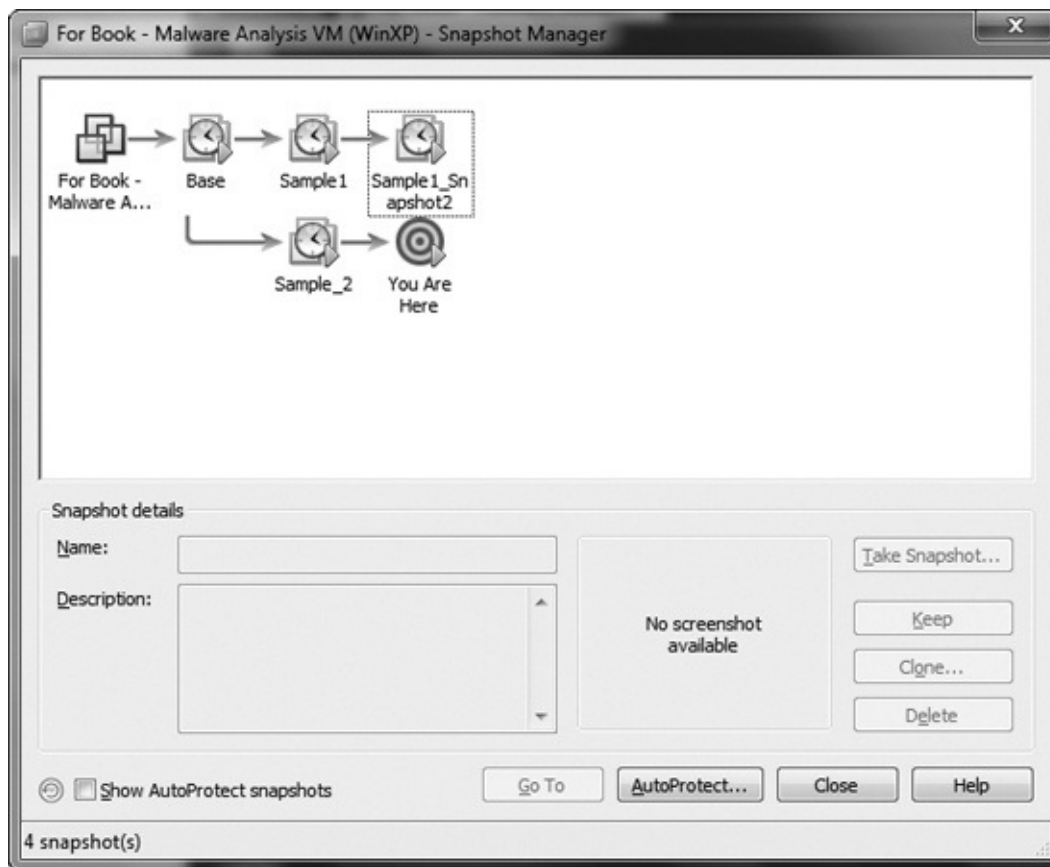


Figure 3-6. VMware Snapshot Manager

## 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 18 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 9](#).

## Conclusion

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

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

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

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

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

# Chapter 4. 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 4-1** 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.

<b>GFI SandBox™</b>		Analysis # 2307
		Sample: win32XYZ.exe (56476e02c29e5dbb9286b5f7b9e708f5)
<b>Table of Contents</b>		
<b>Analysis Summary</b>	.....	<b>3</b>
<b>Analysis Summary</b>	.....	<b>3</b>
<b>Digital Behavior Traits</b>	.....	<b>3</b>
<b>File Activity</b>	.....	<b>4</b>
<b>Stored Modified Files</b>	.....	<b>4</b>
<b>Created Mutexes</b>	.....	<b>5</b>
<b>Created Mutexes</b>	.....	<b>5</b>
<b>Registry Activity</b>	.....	<b>6</b>
<b>Set Values</b>	.....	<b>6</b>
<b>Network Activity</b>	.....	<b>7</b>
<b>Network Events</b>	.....	<b>7</b>
<b>Network Traffic</b>	.....	<b>8</b>
<b>DNS Requests</b>	.....	<b>9</b>
<b>VirusTotal Results</b>	.....	<b>10</b>

Figure 4-1. 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 4-1**, 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 8](#).)

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 2](#), 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 2](#). 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 4-2 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.

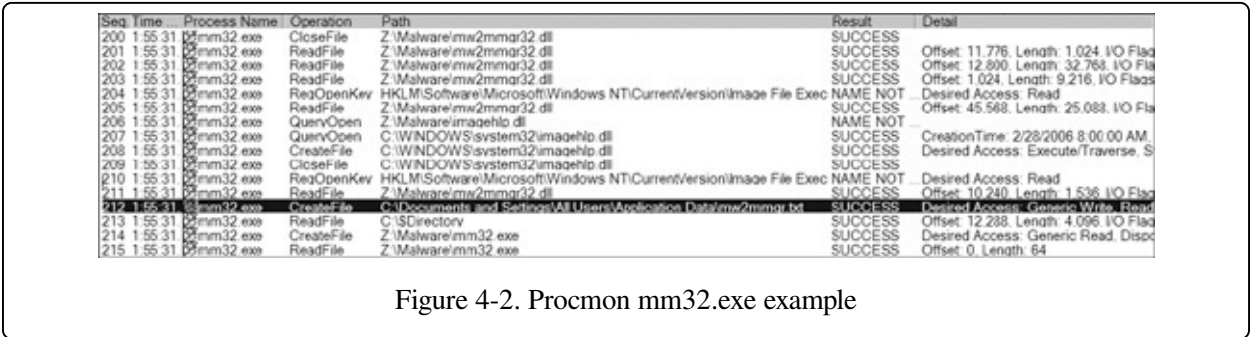


Figure 4-2. Procmon mm32.exe example

## Filtering in Procmon

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

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

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

To set a filter, choose **Filter ► Filter** to open the Filter menu, as shown in the top image of Figure 4-3. When setting a filter, first select a column to filter on using the drop-down box at the upper left, above the Reset button. The most important filters for malware analysis are Process Name, Operation, and Detail. Next, select a comparator, choosing from options such as Is, Contains, and Less Than. Finally,



choose whether this is a filter to include or exclude from display. Because, by default, the display will show all system calls, it is important to reduce the amount displayed.

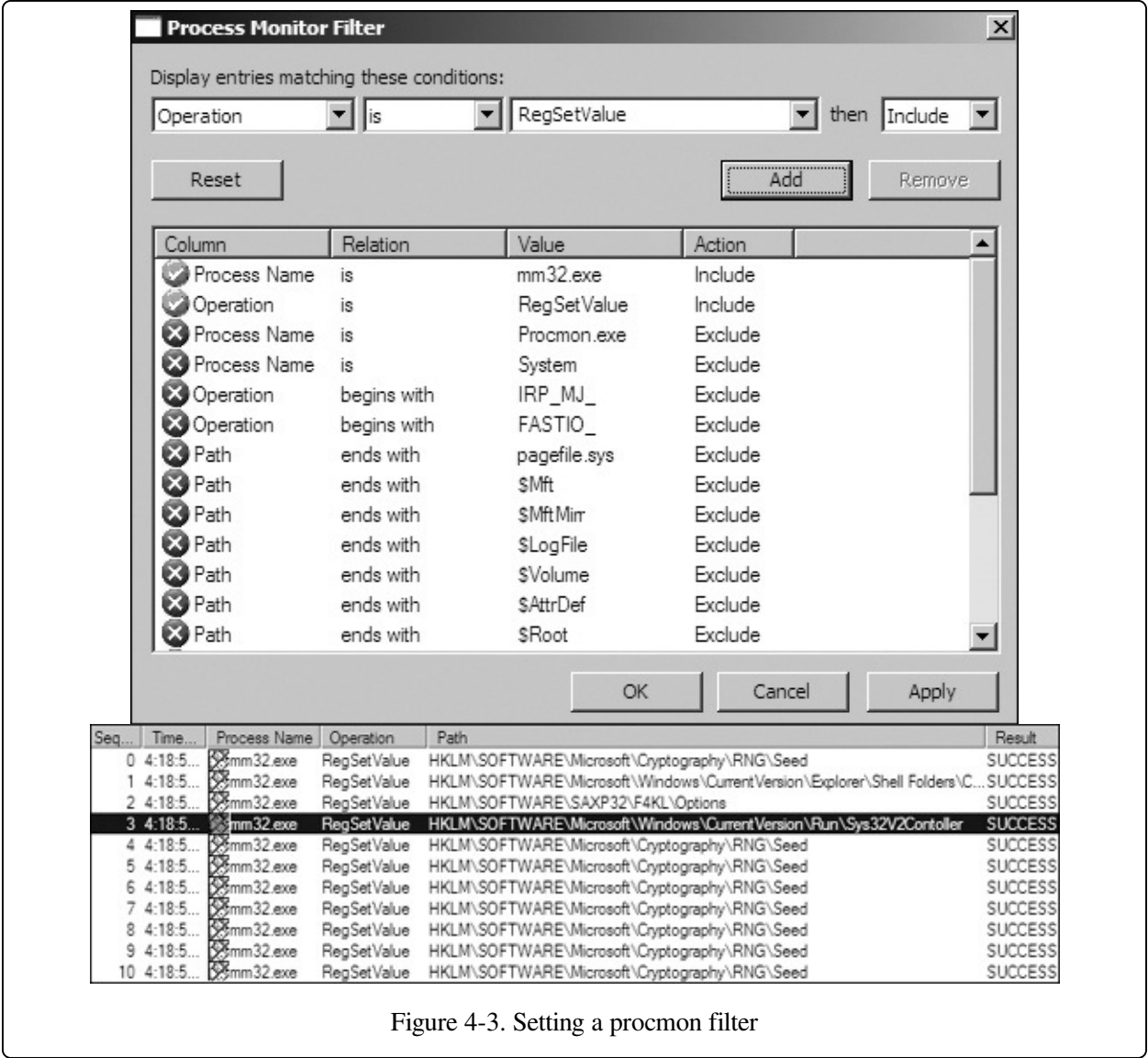


Figure 4-3. Setting a procmon filter

**NOTE**

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

As you can see in the first two rows of [Figure 4-3](#), we’re filtering on Process Name and Operation. We’ve added a filter on Process Name equal to mm32.exe that’s active when the Operation is set to RegSetValue.



After you've chosen a filter, click **Add** for each, and then click **Apply**. As a result of applying our filters, the display window shown in the lower image displays only 11 of the 39,351 events, making it easier for us to see that mm32.exe performed a RegSetValue of registry key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\Sys32V2Control (sequence number 3 using RegSetValue). Double-clicking this RegSetValue event will reveal the data written to this location, which is the current path to the malware.

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

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

- **Registry.** By examining registry operations, you can tell how a piece of malware installs itself in the registry.
- **File system.** Exploring file system interaction can show all files that the malware creates or configuration files it uses.
- **Process activity.** Investigating process activity can tell you whether the malware spawned additional processes.
- **Network.** Identifying network connections can show you any ports on which the malware is listening.

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

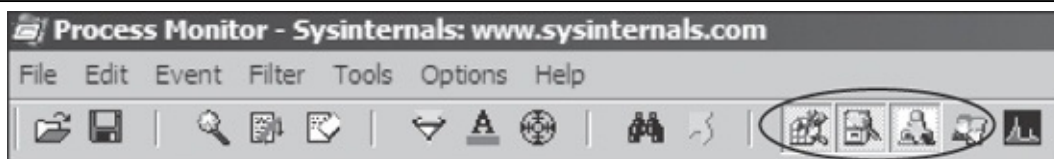


Figure 4-4. Filter buttons for procmon

#### **NOTE**

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

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

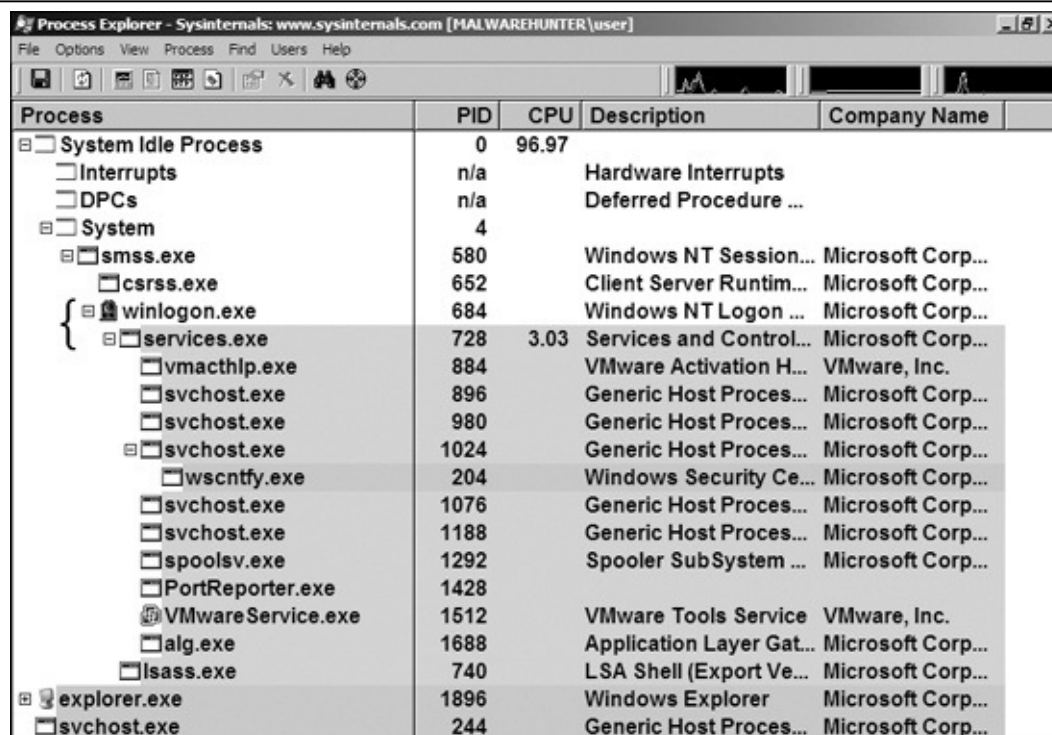
# Viewing Processes with Process Explorer

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

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

## The Process Explorer Display

Process Explorer monitors the processes running on a system and shows them in a tree structure that displays child and parent relationships. For example, in **Figure 4-5** you can see that services.exe is a child process of winlogon.exe, as indicated by the left curly bracket.



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

Figure 4-5. 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 4-6** 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.

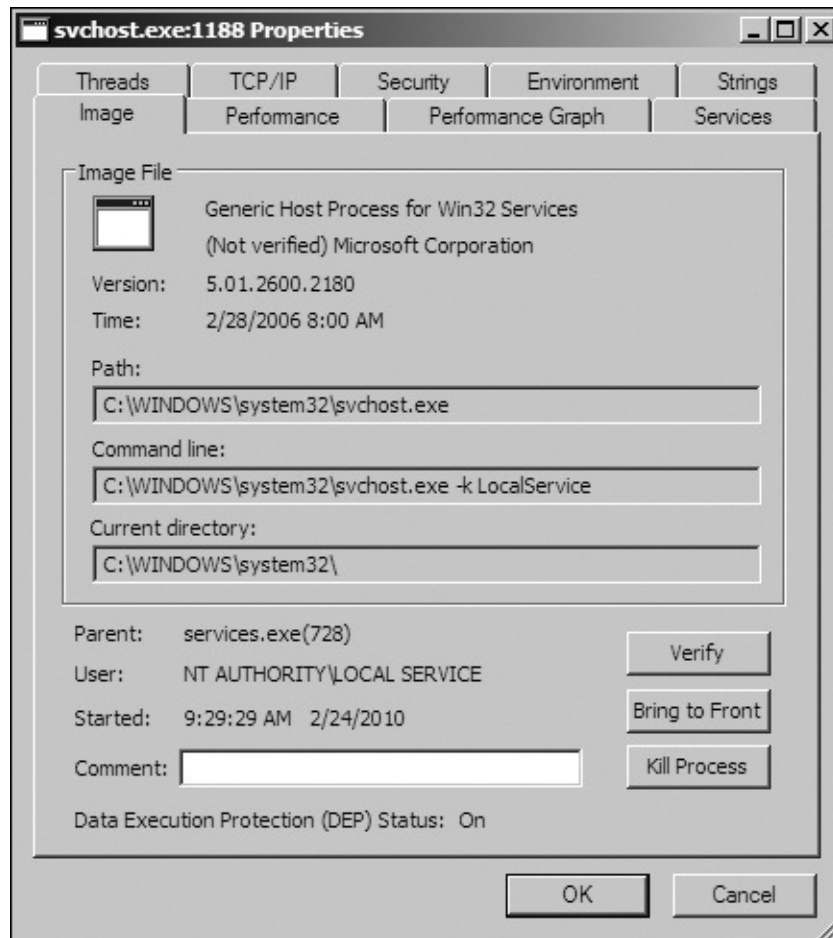


Figure 4-6. The Properties window, Image tab

## Using the Verify Option

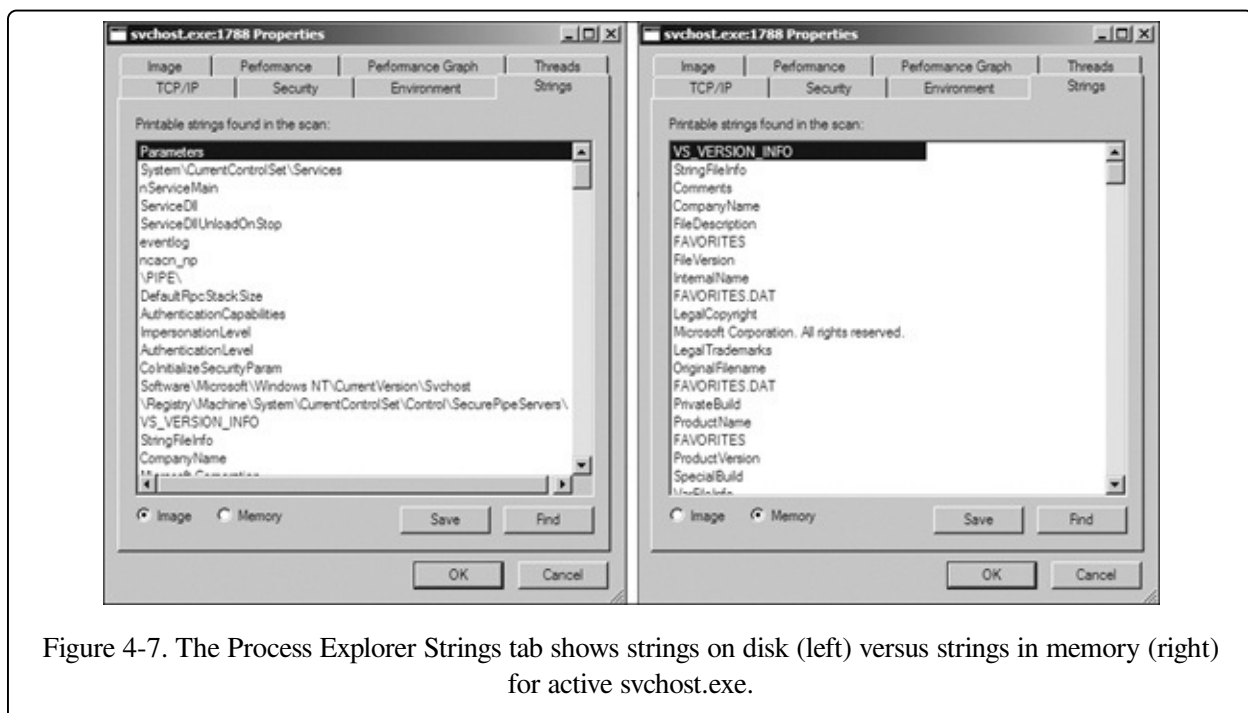
One particularly useful Process Explorer feature is the Verify button on the Image tab. Click this button to verify that the image on disk is, in fact, the Microsoft signed binary. Because Microsoft uses digital signatures for most of its core executables, when Process Explorer verifies that a signature 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 4-6**, the svchost.exe process is verified, yet it is actually malware. We'll discuss process replacement in more detail in **Chapter 13**.

## 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 4-7**. If the two string listings are drastically different, process replacement may have occurred. This string discrepancy is displayed in **Figure 4-7**. 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).



## 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 4-8**) is an open source registry comparison tool that allows you to take and compare two registry snapshots.

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

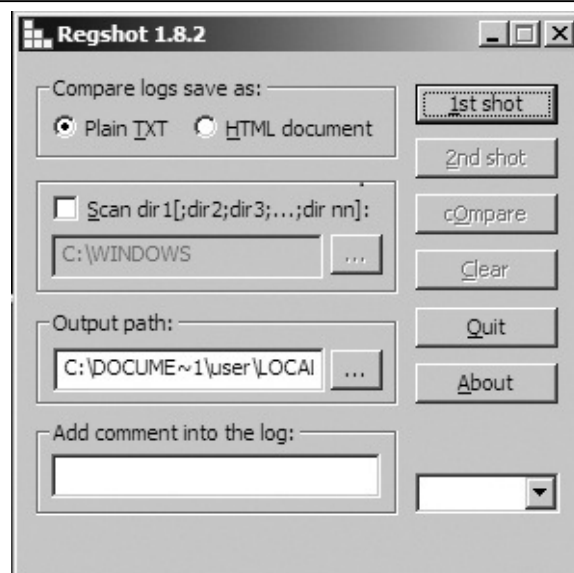


Figure 4-8. Regshot window

**Example 4-1** displays a subset of the results generated by Regshot during malware analysis. Registry snapshots were taken before and after running the spyware ckr.exe.

## Example 4-1. Regshot comparison results

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

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

```
-----
Values added:3
```



```

-----
1 HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\ckr:C:\WINDOWS\system32\
ckr.exe
...
...

-----
Values modified:2
-----
2 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
-----

```

As you can see ckr.exe creates a value at HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run as a persistence mechanism **1**. A certain amount of noise **2** 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 15](#). 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 3](#) 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](http://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 **2** and select the interface at **4**. 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 4-9](#), 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 **1**.

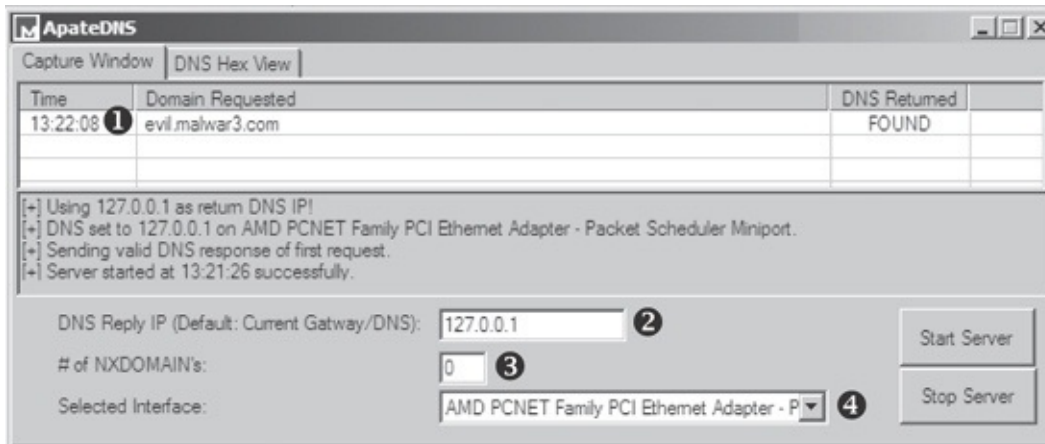


Figure 4-9. ApateDNS responding to a request for evil.malwar3.com

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 **3**. 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 4-9**. 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. **Example 4-2** shows an example.

#### Example 4-2. Netcat example listening on port 80

```
C:\> nc -l -p 80 1
POST cqframe.htm HTTP/1.1
Host: www.google.com 2
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> 3
```

The Netcat (nc) command **1** 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 **3**, but it does not immediately provide the shell. The network connection first appears as an HTTP POST request to `www.google.com` **2**, 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 4-10**:

- The Filter box **1** is used to filter the packets displayed.
- The packet listing **2** shows all packets that satisfy the display filter.
- The packet detail window **3** displays the contents of the currently selected packet (in this case, packet 47).
- The hex window **4** 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.

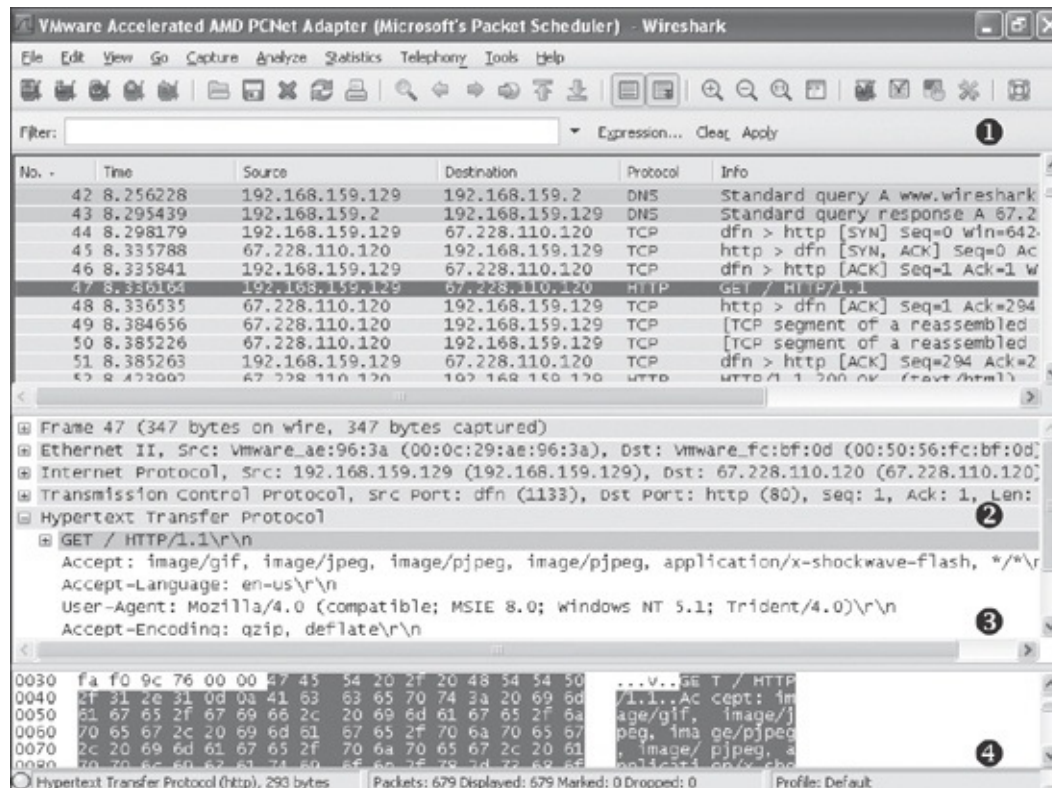


Figure 4-10. Wireshark DNS and HTTP example

To use Wireshark to view the contents of a TCP session, right-click any TCP packet and select **Follow TCP Stream**. As you can see in [Figure 4-11](#), both ends of the conversation are displayed in session order, with different colors showing each side of the connection.

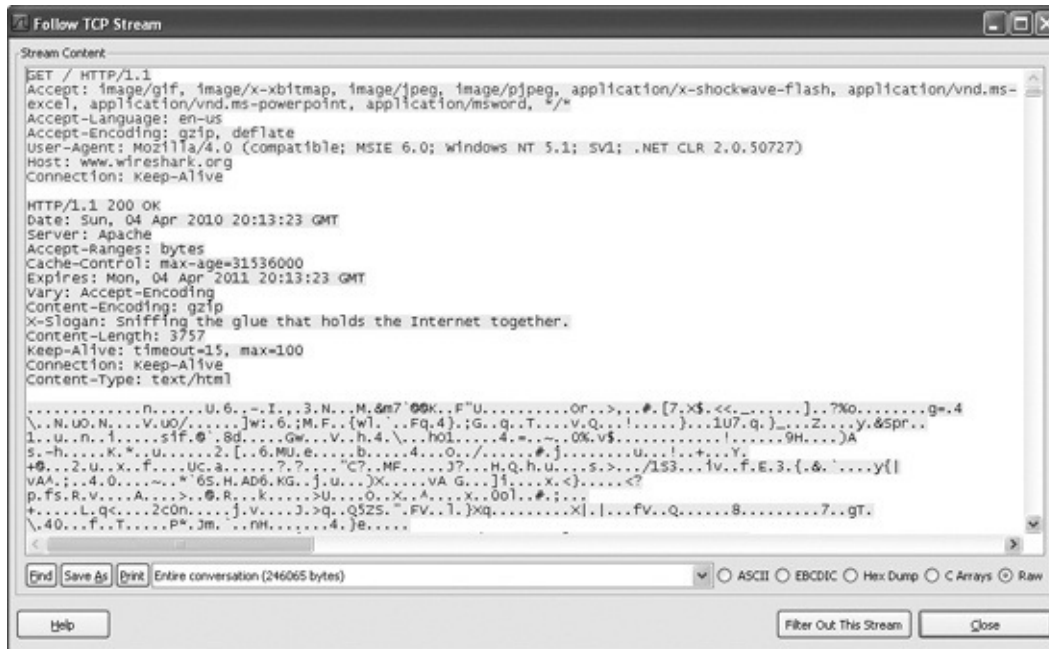


Figure 4-11. Wireshark's Follow TCP Stream window

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

### WARNING

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

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

**Chapter 15** 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. **Example 4-3** 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.

## Example 4-3. INetSim default emulated services

```
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)
```

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 it is scanned.



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

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

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

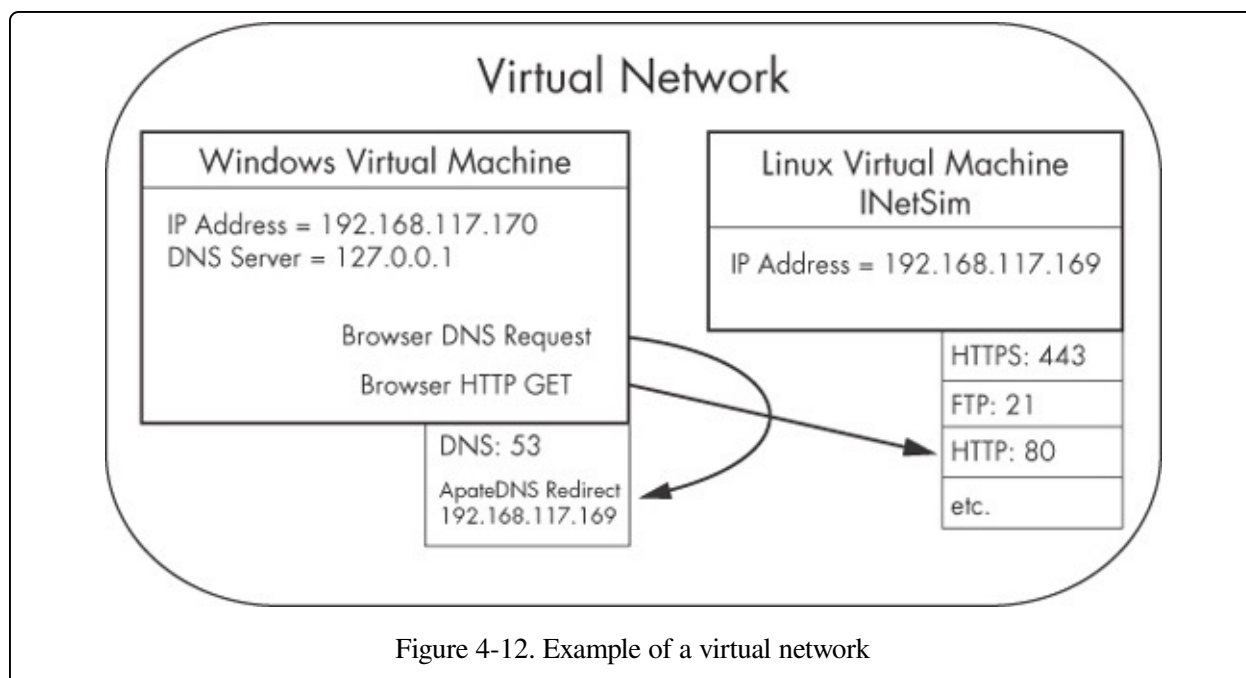
## Basic Dynamic Tools in Practice

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

1. Running procmon and setting a filter on the malware executable name and clearing out all events just before running.
2. Starting Process Explorer.
3. Gathering a first snapshot of the registry using Regshot.
4. Setting up your virtual network to your liking using INetSim and ApateDNS.
5. Setting up network traffic logging using Wireshark.

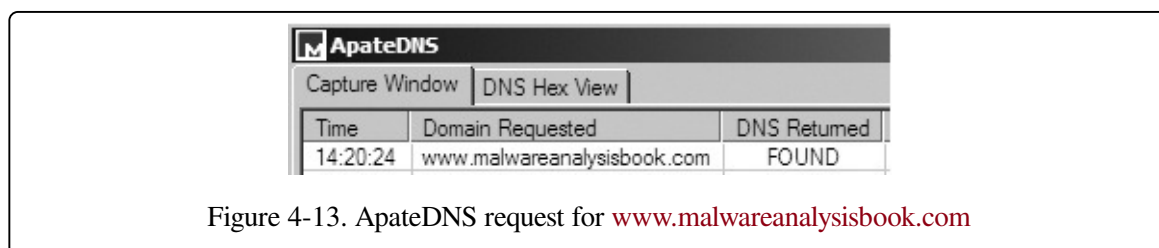
**Figure 4-12** shows a diagram of a virtual network that can be set up for malware analysis. This virtual network contains two hosts: the malware analysis Windows virtual machine and the Linux virtual machine running INetSim. The Linux virtual machine is listening on many ports, including HTTPS, FTP, and HTTP, through the use of INetSim. The Windows virtual machine is listening on port 53 for DNS requests through the use of ApateDNS. The DNS server for the Windows virtual machine has been configured to localhost (127.0.0.1). ApateDNS is configured to redirect you to the Linux virtual machine (192.168.117.169).

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

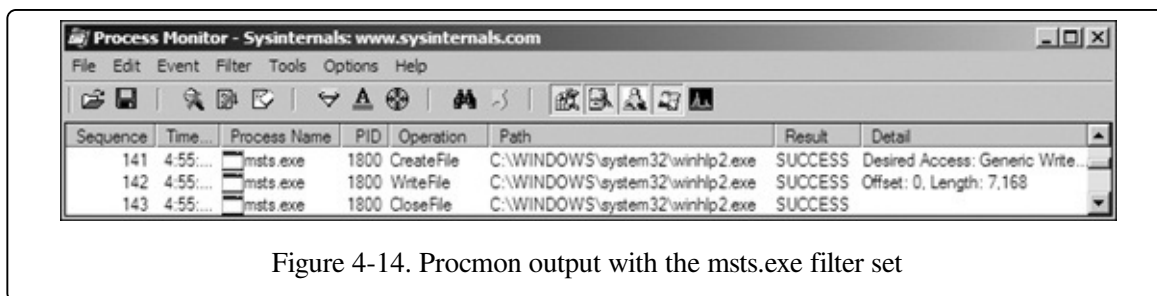


Let's see how this setup would work in practice by examining the malware msts.exe. We complete our initial setup and then run msts.exe on our malware analysis virtual machine. After some time, we stop event capture with procmon and run a second snapshot with Regshot. At this point we begin analysis as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

1. Examine ApatDNS to see if DNS requests were performed. As shown in [Figure 4-13](#), we notice that the malware performed a DNS request for [www.malwareanalysisbook.com](http://www.malwareanalysisbook.com).



2. Review the procmon results for file system modifications. In the procmon results shown in [Figure 4-14](#), we see CreateFile and WriteFile (sequence numbers 141 and 142) operations for C:\WINDOWS\system32\winhlp2.exe. Upon further investigation, we compare winhlp2.exe to msts.exe and see that they are identical. We conclude that the malware copies itself to that location.



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

```

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

```

- Use Process Explorer to examine the process to determine whether it creates mutexes or listens for incoming connections. The Process Explorer output in **Figure 4-15** shows that msts.exe creates a mutex (also known as a mutant) named Evil1 **1**. We discuss mutexes in depth in **Chapter 8**, but you should know that msts.exe likely created the mutex to ensure that only one version of the malware is running at a time. Mutexes can provide an excellent fingerprint for malware if they are unique enough.
- Review the INetSim logs for requests and attempted connections on standard services. The first line in the INetSim logs (shown next) tells us that the malware communicates over port 443, though not with standard Secure Sockets Layer (SSL), as shown next in the reported errors at **1**.

```

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

```

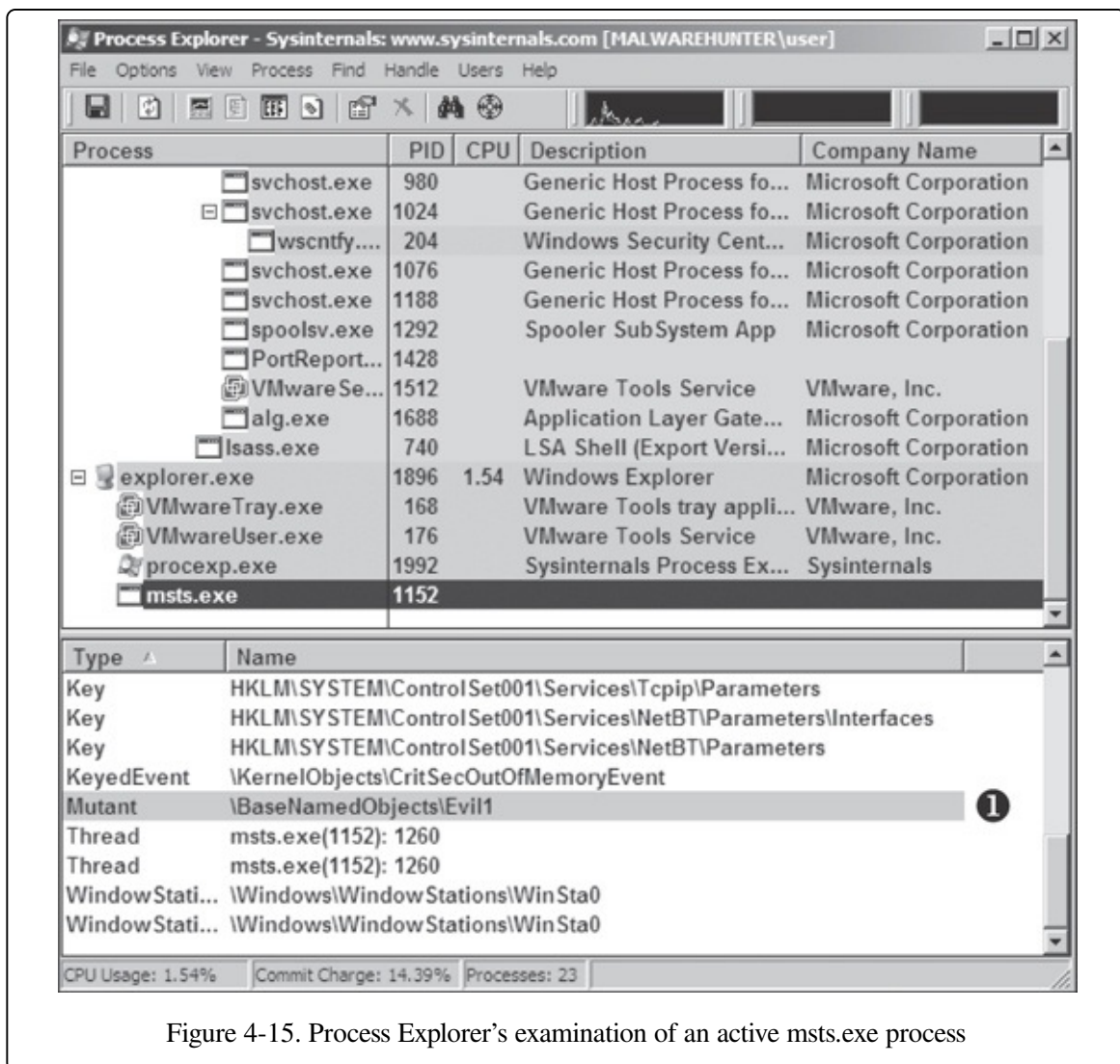


Figure 4-15. Process Explorer's examination of an active msts.exe process

- Review the Wireshark capture for network traffic generated by the malware. By using INetSim while capturing with Wireshark, we can capture the TCP handshake and the initial data packets sent by the malware. The contents of the TCP stream sent over port 443, as shown in [Figure 4-16](#), shows random ACSII data, which is often indicative of a custom protocol. When this happens, your best bet is to run the malware several more times to look for any consistency in the initial packets of the connection. (The resulting information could be used to draft a network-based signature, skills that we explore in [Chapter 15](#).)

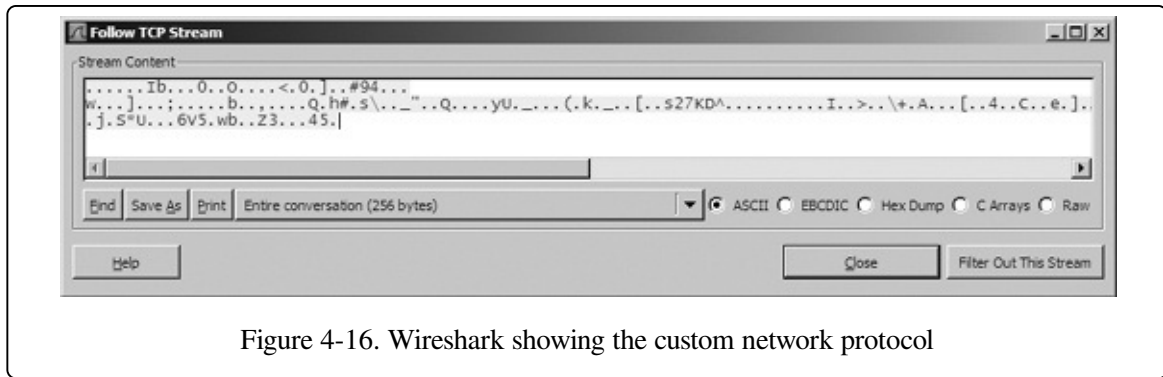


Figure 4-16. Wireshark showing the custom network protocol

## Conclusion

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

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

# Labs

## Lab 3-1

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

### Questions

Q: 1. What are this malware's imports and strings?

Q: 2. What are the malware's host-based indicators?

Q: 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

Q: 1. How can you get this malware to install itself?

Q: 2. How would you get this malware to run after installation?

Q: 3. How can you find the process under which this malware is running?

Q: 4. Which filters could you set in order to use procmon to glean information?

Q: 5. What are the malware's host-based indicators?

Q: 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

Q: 1. What do you notice when monitoring this malware with Process Explorer?

Q: 2. Can you identify any live memory modifications?



---

Q: 3. What are the malware's host-based indicators?

Q: 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 10** labs.)

### Questions

Q: 1. What happens when you run this file?

Q: 2. What is causing the roadblock in dynamic analysis?

---

Q: 3. Are there other ways to run this program?

---

## **Part II. Advanced Static Analysis**

# Chapter 5. A Crash Course in x86 Disassembly

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

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

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

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

## Levels of Abstraction

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

**Figure 5-1** shows the three coding levels involved in malware analysis. Malware authors create programs at the high-level language level and use a compiler to generate machine code to be run by the CPU. Conversely, malware analysts and reverse engineers operate at the low-level language level; we use a disassembler to generate assembly code that we can read and analyze to figure out how a program operates.

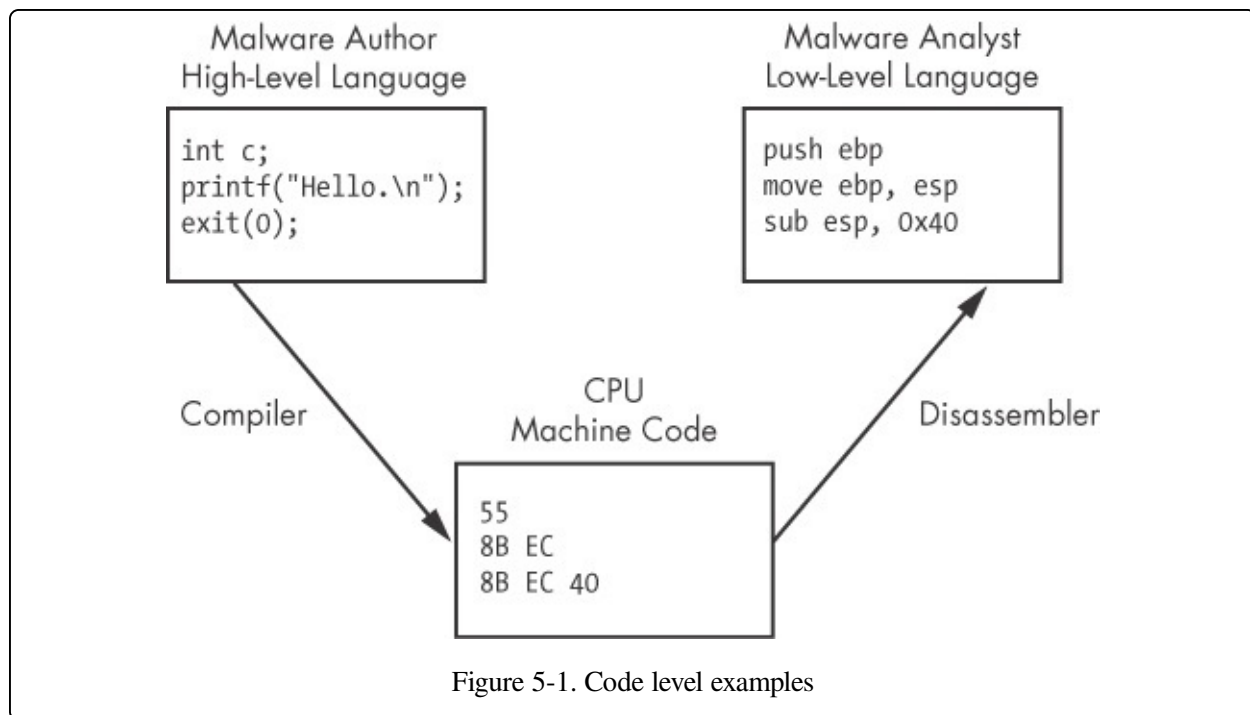


Figure 5-1 shows a simplified model, but computer systems are generally described with the following six different levels of abstraction. We list these levels starting from the bottom. Higher levels of abstraction are placed near the top with more specific concepts underneath, so the lower you get, the less portable the level will be across computer systems.

- **Hardware.** The hardware level, the only physical level, consists of electrical circuits that implement complex combinations of logical operators such as XOR, AND, OR, and NOT gates, known as digital logic. Because of its physical nature, hardware cannot be easily manipulated by software.
- **Microcode.** The microcode level is also known as firmware. Microcode operates only on the exact circuitry for which it was designed. It contains microinstructions that translate from the higher machine-code level to provide a way to interface with the hardware. When performing malware analysis, we usually don't worry about the microcode because it is often specific to the computer hardware for which it was written.
- **Machine code.** The machine code level consists of opcodes, hexadecimal digits that tell the processor what you want it to do. Machine code is typically implemented with several microcode instructions so that the underlying hardware can execute the code. Machine code is created when a computer

program written in a high-level language is compiled.

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

#### NOTE

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

- **High-level languages.** Most computer programmers operate at the level of high-level languages. High-level languages provide strong abstraction from the machine level and make it easy to use programming logic and flow-control mechanisms. High-level languages include C, C++, and others. These languages are typically turned into machine code by a compiler through a process known as compilation.
- **Interpreted languages.** Interpreted languages are at the top level. Many programmers use interpreted languages such as C#, Perl, .NET, and Java. The code at this level is not compiled into machine code; instead, it is translated into bytecode. Bytecode is an intermediate representation that is specific to the programming language. Bytecode executes within an interpreter, which is a program that translates bytecode into executable machine code on the fly at runtime. An interpreter provides an automatic level of abstraction when compared to traditional compiled code, because it can handle errors and memory management on its own, independent of the OS.

# Reverse-Engineering

When malware is stored on a disk, it is typically in binary form at the machine code level. As discussed, machine code is the form of code that the computer can run quickly and efficiently. When we disassemble malware (as shown in [Figure 5-1](#)), we take the malware binary as input and generate assembly language code as output, usually with a disassembler. ([Chapter 6](#) 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 22](#) 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 5-2**. 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.

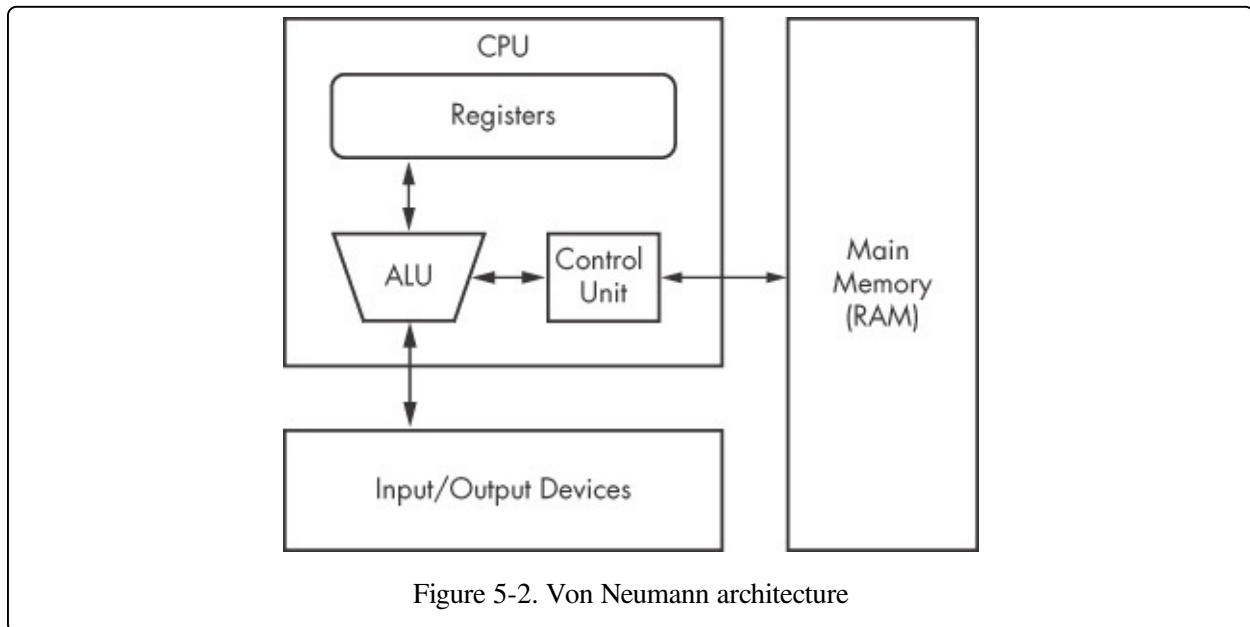
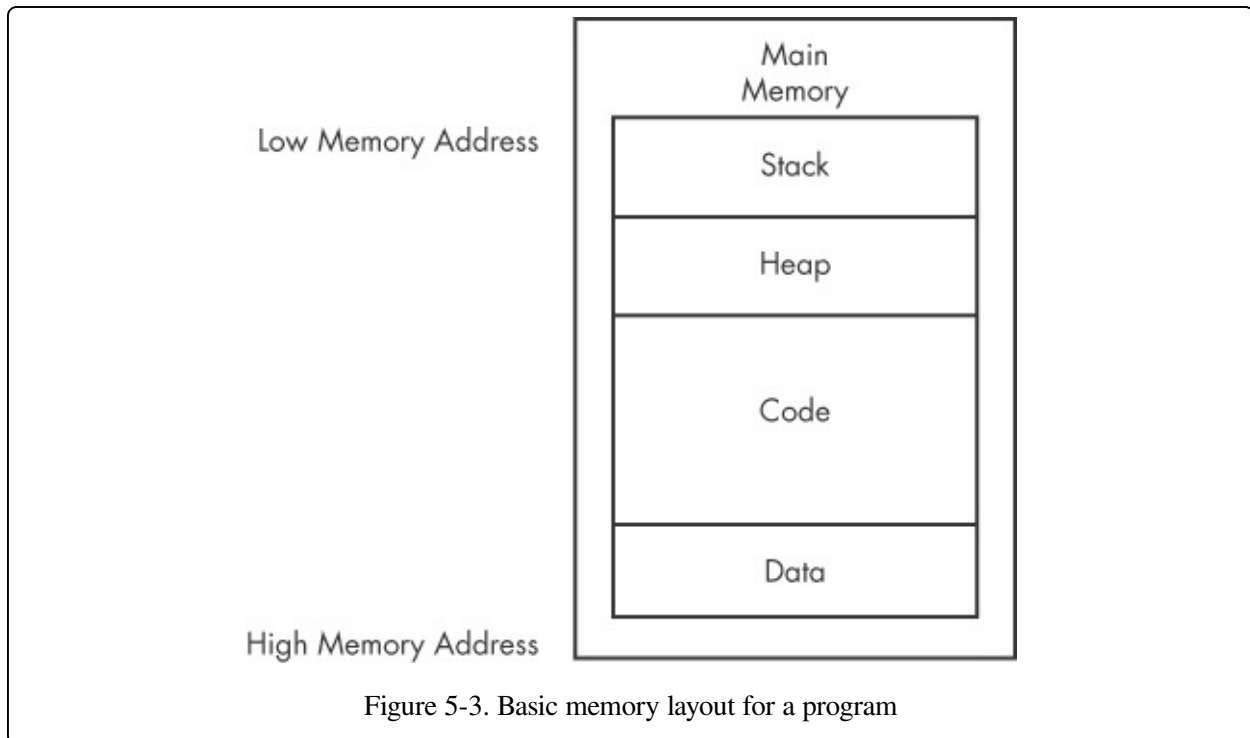


Figure 5-2. Von Neumann architecture

As you can see in **Figure 5-2**, 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 5-3](#).



- **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 5-3](#) 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 5-1](#), 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 5-1. Instruction Format

Mnemonic	Destination operand	Source operand
<code>mov</code>	<code>ecx</code>	<code>0x42</code>

## 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 5-2](#), 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 5-2. Instruction Opcodes

Instruction	<code>mov ecx, 0x42</code>			
Opcodes	<code>B9</code>	<code>42</code>	<code>00</code>	<code>00 00</code>

`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 5-1](#).
- Register operands refer to registers, such as `ecx` in [Table 5-1](#).
- 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 5-3](#) 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 5-3](#) 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 5-3. 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)			

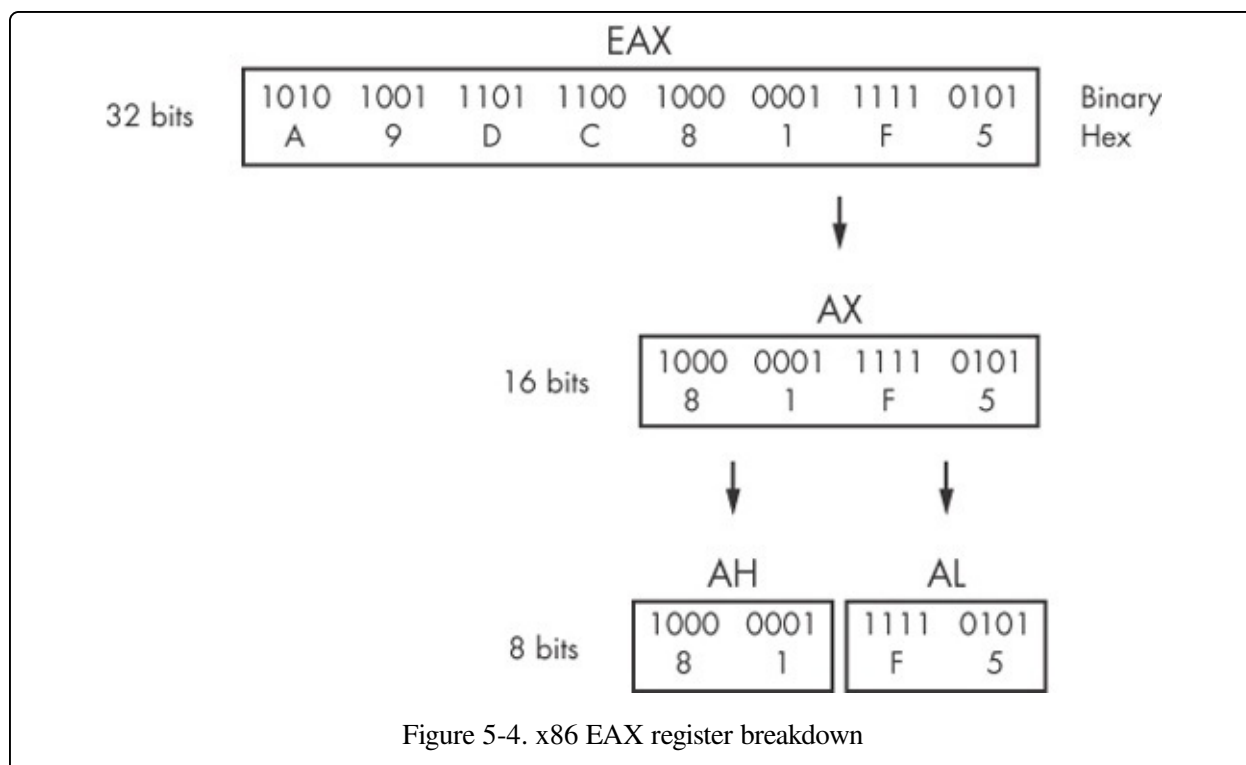
All general registers are 32 bits in size and can be referenced as either 32 or 16 bits in assembly code. For example, EDX is used to reference the full 32-bit register, and DX is used to reference the lower 16 bits of the EDX register.

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 5-3** lists the possible references for each general register. The EAX register breakdown is illustrated in **Figure 5-4**. 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.



Some x86 instructions use specific registers by definition. For example, the multiplication and division instructions always use EAX and EDX.

In addition to instruction definitions, general registers can be used in a consistent fashion throughout a program. The use of registers in a consistent fashion across compiled code is known as a convention. Knowledge of the conventions used by compilers allows a malware analyst to examine the code more quickly, because time isn't wasted figuring out the context of how a register is being used. For example, the EAX generally contains the return value for function calls. Therefore, if you see the EAX register used immediately after a function call, you are probably seeing the code manipulate the return value.

## Flags

The EFLAGS register is a status register. In the x86 architecture, it is 32 bits in size, and each bit is a flag. During execution, each flag is either set (1) or cleared (0) to control CPU operations or indicate the results of a CPU operation. The following flags are most important to malware analysis:

- **ZF**. The zero flag is set when the result of an operation is equal to zero; otherwise, it is cleared.

- **CF**. The carry flag is set when the result of an operation is too large or too small for the destination operand; otherwise, it is cleared.
- **SF**. The sign flag is set when the result of an operation is negative or cleared when the result is positive. This flag is also set when the most significant bit is set after an arithmetic operation.
- **TF**. The trap flag is used for debugging. The x86 processor will execute only one instruction at a time if this flag is set.

#### 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 5-4** 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 5-4** uses an

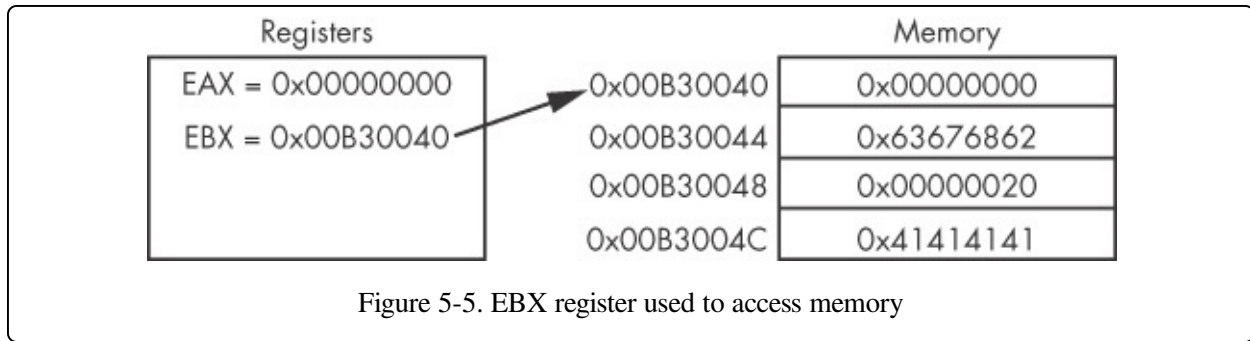
equation to calculate a memory address. This saves space, because it does not require separate instructions to perform the calculation contained within the brackets. Performing calculations such as this within an instruction is not possible unless you are calculating a memory address. For example, `mov eax, ebx+esi*4` (without the brackets) is an invalid instruction.

Table 5-4. `mov` Instruction Examples

Instruction	Description
<code>mov eax, ebx</code>	Copies the contents of EBX into the EAX register
<code>mov eax, 0x42</code>	Copies the value 0x42 into the EAX register
<code>mov eax, [0x4037C4]</code>	Copies the 4 bytes at the memory location 0x4037C4 into the EAX register
<code>mov eax, [ebx]</code>	Copies the 4 bytes at the memory location specified by the EBX register into the EAX register
<code>mov eax, [ebx+esi*4]</code>	Copies the 4 bytes at the memory location specified by the result of the equation <code>ebx+esi*4</code> 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 data at the memory address specified by `EBX+8`. Therefore, `lea eax, [ebx+8]` would be the same as `mov eax, ebx+8`; however, a `mov` instruction like that is invalid.

Figure 5-5 shows values for registers EAX and EBX on the left and the information contained in memory on the right. EBX is set to 0xB30040. At address 0xB30048 is the value 0x20. The instruction `mov eax, [ebx+8]` places the value 0x20 (obtained from memory) into EAX, and the instruction `lea eax, [ebx+8]` places the value 0xB30048 into EAX.



The `lea` instruction is not used exclusively to refer to memory addresses. It is useful when calculating values, because it requires fewer instructions. For example, it is common to see an instruction such as `lea ebx, [eax*5+5]`, where `eax` is a number, rather than a memory address. This instruction is the functional equivalent of `ebx = (eax+1)*5`, but the former is shorter or more efficient for the compiler to use instead of a total of four instructions (for example `inc eax; mov ecx, 5; mul ecx; mov ebx, eax`).

## Arithmetic

x86 assembly includes many instructions for arithmetic, ranging from basic addition and subtraction to logical operators. We'll cover the most commonly used instructions in this section.

Addition or subtraction adds or subtracts a value from a destination operand. The format of the addition instruction is `add destination, value`. The format of the subtraction instruction is `sub destination, value`. The `sub` instruction modifies two important flags: the zero flag (ZF) and carry flag (CF). The ZF is set if the result is zero, and CF is set if the destination is less than the value subtracted. The `inc` and `dec` instructions increment or decrement a register by one. [Table 5-5](#) shows examples of the addition and subtraction instructions.

Table 5-5. Addition and Subtraction Instruction Examples

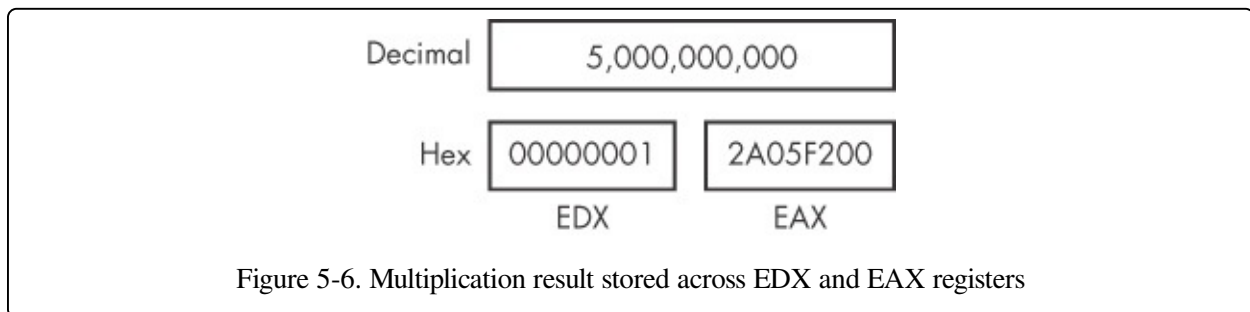
Instruction	Description
<code>sub eax, 0x10</code>	Subtracts 0x10 from EAX
<code>add eax, ebx</code>	Adds EBX to EAX and stores the result in EAX
<code>inc edx</code>	Increments EDX by 1

dec ecx	Decrements ECX by 1
---------	---------------------

Multiplication and division both act on a predefined register, so the command is simply the instruction, plus the value that the register will be multiplied or divided by. The format of the `mul` instruction is `mul value`. Similarly, the format of `div` instruction is `div value`. The assignment of the register on which a `mul` or `div` instruction acts can occur many instructions earlier, so you might need to search through a program to find it.

The `mul value` instruction always multiplies `eax` by `value`. Therefore, EAX must be set up appropriately before the multiplication occurs. The result is stored as a 64-bit value across two registers: EDX and EAX. EDX stores the most significant 32 bits of the operations, and EAX stores the least significant 32 bits. Figure 5-6 depicts the values in EDX and EAX when the decimal result of multiplication is 5,000,000,000 and is too large to fit in a single register.

The `div value` instruction does the same thing as `mul`, except in the opposite direction: It divides the 64 bits across EDX and EAX by `value`. Therefore, the EDX and EAX registers must be set up appropriately before the division occurs. The result of the division operation is stored in EAX, and the remainder is stored in EDX.



A programmer obtains the remainder of a division operation by using an operation known as modulo, which will be compiled into assembly through the use of the EDX register after the `div` instruction (since it contains the remainder). Table 5-6 shows examples of the `mul` and `div` instructions. The instructions `imul` and `idiv` are the signed versions of the `mul` and `div` instructions.

Table 5-6. Multiplication and Division Instruction Examples

Instruction	Description
-------------	-------------



`mul 0x50` Multiplies EAX by 0x50 and stores the result in EDX:EAX

`div 0x75` Divides EDX:EAX by 0x75 and stores the result in EAX and the remainder in EDX

Logical operators such as OR, AND, and XOR are used in x86 architecture. The corresponding instructions operate similar to how `add` and `sub` operate. They perform the specified operation between the source and destination operands and store the result in the destination. The `xor` instruction is frequently encountered in disassembly. For example, `xor eax, eax` is a quick way to set the EAX register to zero. This is done for optimization, because this instruction requires only 2 bytes, whereas `mov eax, 0` requires 5 bytes.

The `shr` and `shl` instructions are used to shift registers. The format of the `shr` instruction is `shr destination, count`, and the `shl` instruction has the same format. The `shr` and `shl` instructions shift the bits in the destination operand to the right and left, respectively, by the number of bits specified in the count operand. Bits shifted beyond the destination boundary are first shifted into the CF flag. Zero bits are filled in during the shift. For example, if you have the binary value 1000 and shift it right by 1, the result is 0100. At the end of the shift instruction, the CF flag contains the last bit shifted out of the destination operand.

The rotation instructions, `ror` and `rol`, are similar to the shift instructions, except the shifted bits that “fall off” with the shift operation are rotated to the other end. In other words, during a right rotation (`ror`) the least significant bits are rotated to the most significant position. Left rotation (`rol`) is the exact opposite. [Table 5-7](#) displays examples of these instructions.

Table 5-7. Common Logical and Shifting Arithmetic Instructions

Instruction	Description
<code>xor eax, eax</code>	Clears the EAX register
<code>or eax, 0x7575</code>	Performs the logical or operation on EAX with 0x7575
<code>mov eax, 0xA</code> <code>shl</code>	Shifts the EAX register to the left 2 bits; these two instructions result in EAX = 0x28, because 1010 (0xA in binary) shifted 2 bits left is 101000 (0x28)

`eax, 2`

```
mov bl, 0xA    Rotates the BL register to the right 2 bits; these two instructions result in BL =  
ror bl, 2       10000010, because 1010 rotated 2 bits right is 10000010
```

Shifting is often used in place of multiplication as an optimization. Shifting is simpler and faster than multiplication, because you don't need to set up registers and move data around, as you do for multiplication. The `shl eax, 1` instruction computes the same result as multiplying EAX by two. Shifting to the left two bit positions multiplies the operand by four, and shifting to the left three bit positions multiplies the operand by eight. Shifting an operand to the left  $n$  bits multiplies it by  $2^n$ .

During malware analysis, if you encounter a function containing only the instructions `xor`, `or`, `and`, `shl`, `ror`, `shr`, or `rol` repeatedly and seemingly randomly, you have probably encountered an encryption or compression function. Don't get bogged down trying to analyze each instruction unless you really need to do so. Instead, your best bet in most cases is to mark this as an encryption routine and move on.

## NOP

The final simple instruction, `nop`, does nothing. When it's issued, execution simply proceeds to the next instruction. The instruction `nop` is actually a pseudonym for `xhcg eax, eax`, but since exchanging EAX with itself does nothing, it is popularly referred to as NOP (no operation).

The opcode for this instruction is 0x90. It is commonly used in a NOP sled for buffer overflow attacks, when attackers don't have perfect control of their exploitation. It provides execution padding, which reduces the risk that the malicious shellcode will start executing in the middle, and therefore malfunction. We discuss `nop` sleds and shellcode in depth in [Chapter 20](#).

## 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 5-7](#)).

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 7](#) 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 5-8](#) 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).
3. Through the use of a function prologue, space is allocated on the stack for local variables and EBP (the base pointer) is pushed onto the stack. This is done to save EBP for the calling function.
4. The function performs its work.
5. Through the use of a function epilogue, the stack is restored. ESP is adjusted to free the local variables, and EBP is restored so that the calling function can address its variables properly. The `leave` instruction can be used as an epilogue because it sets ESP to equal EBP and pops EBP off the stack.
6. The function returns by calling the `ret` instruction. This pops the return address off the stack and into EIP, so that the program will continue executing from where the original call was made.
7. The stack is adjusted to remove the arguments that were sent, unless they'll be used again later.

## Stack Layout

As discussed, the stack is allocated in a top-down fashion, with the higher memory addresses used first. **Figure 5-7** shows how the stack is laid out in memory. Each time a call is performed, a new stack frame is generated. A function maintains its own stack frame until it returns, at which time the caller's stack frame is restored and execution is transferred back to the calling function.

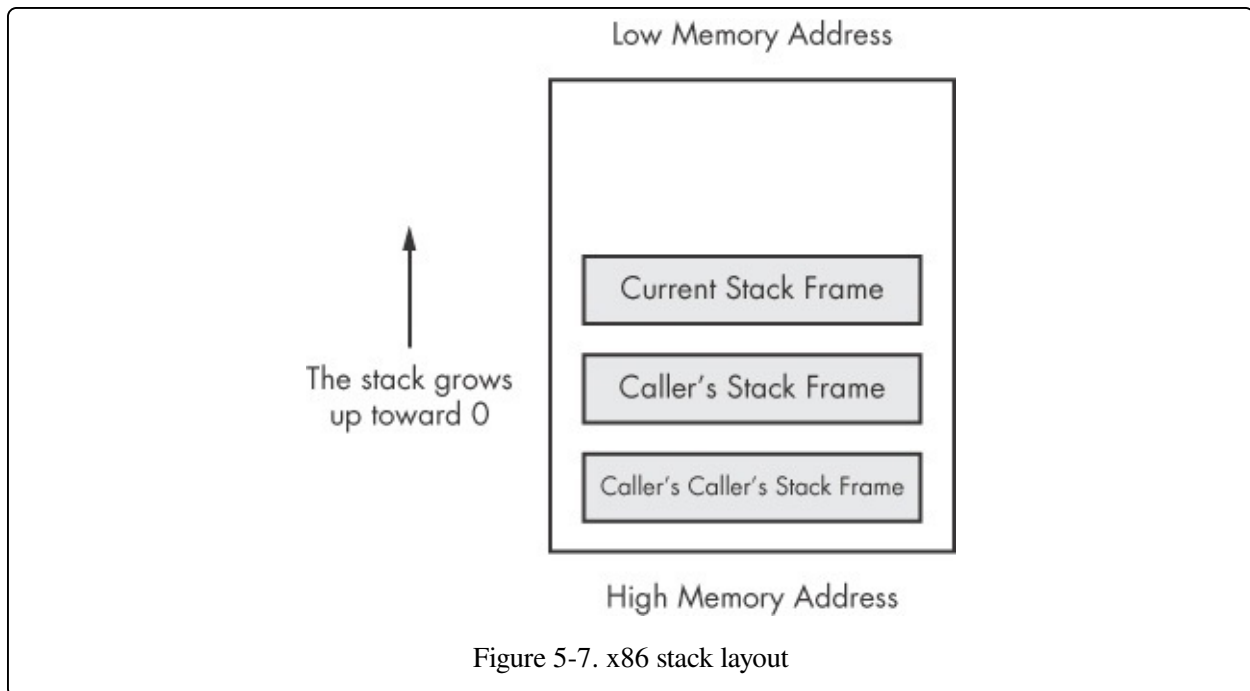
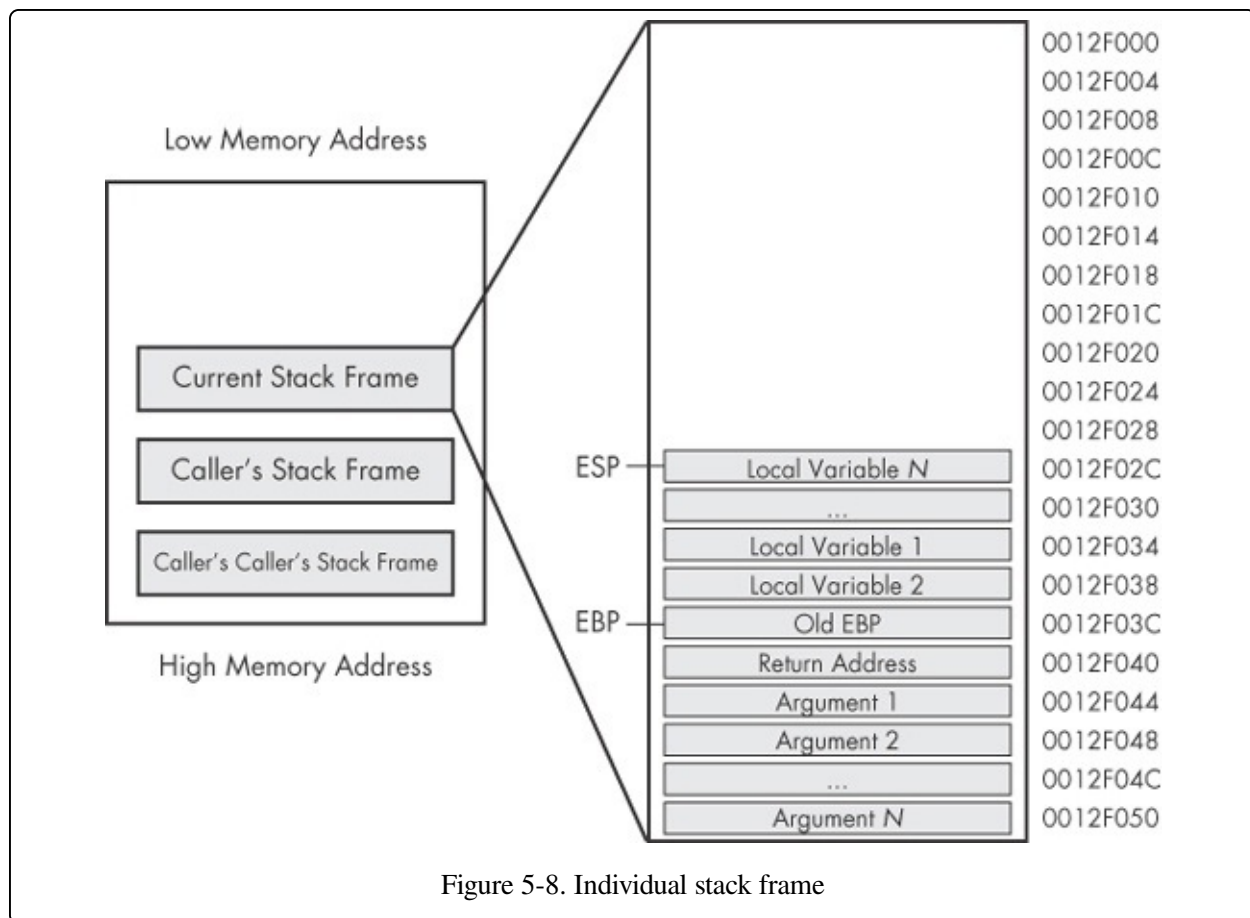


Figure 5-8 shows a dissection of one of the individual stack frames from Figure 5-7. The memory locations of individual items are also displayed. In this diagram, ESP would point to the top of the stack, which is the memory address 0x12F02C. EBP would be set to 0x12F03C throughout the duration of the function, so that the local variables and arguments can be referenced using EBP. The arguments that are pushed onto the stack before the call are shown at the bottom of the stack frame. Next, it contains the return address that is put on the stack automatically by the call instruction. The old EBP is next on the stack; this is the EBP from the caller's stack frame.

When information is pushed onto the stack, ESP will be decreased. In the example in Figure 5-8, if the instruction `push eax` were executed, ESP would be decremented by four and would contain 0x12F028, and the data contained in EAX would be copied to 0x12F028. If the instruction `pop ebx` were executed, the data at 0x12F028 would be moved into the EBX register, and then ESP would be incremented by four.



It is possible to read data from the stack without using the `push` or `pop` instructions. For example, the instruction `mov eax, ss:[esp]` will directly access the top of the stack. This is identical to `pop eax`, except the ESP register is not impacted. The convention used depends on the compiler and how the compiler is configured. (We discuss this in more detail in [Chapter 7](#).)

The x86 architecture provides additional instructions for popping and pushing, the most popular of which are `pusha` and `pushad`. These instructions push all the registers onto the stack and are commonly used with `popa` and `popad`, which pop all the registers off the stack. The `pusha` and `pushad` functions operate as follows:

- `pusha` pushes the 16-bit registers on the stack in the following order: AX, CX, DX, BX, SP, BP, SI, DI.
- `pushad` pushes the 32-bit registers on the stack in the following order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI.

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 5-8](#) shows how the `cmp` instruction impacts the flags.

Table 5-8. `cmp` Instruction and Flags

<code>cmp dst, src</code>	ZF	CF
<code>dst = src</code>	1	0
<code>dst &lt; src</code>	0	1
<code>dst &gt; src</code>	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 5-9** shows the most common conditional jump instructions and details of how they operate. `Jcc` is the shorthand for generally describing conditional jumps.

Table 5-9. Conditional Jumps

Instruction	Description
<code>jz loc</code>	Jump to specified location if <code>ZF = 1</code> .
<code>jnz loc</code>	Jump to specified location if <code>ZF = 0</code> .
<code>je loc</code>	Same as <code>jz</code> , but commonly used after a <code>cmp</code> instruction. Jump will occur if the destination operand equals the source operand.
<code>jne loc</code>	Same as <code>jnz</code> , but commonly used after a <code>cmp</code> . Jump will occur if the destination operand is not equal to the source operand.
<code>jg loc</code>	Performs signed comparison jump after a <code>cmp</code> if the destination operand is greater than the source operand.
<code>jge loc</code>	Performs signed comparison jump after a <code>cmp</code> if the destination operand is greater than or equal to the source operand.
<code>ja loc</code>	Same as <code>jg</code> , but an unsigned comparison is performed.
<code>jae loc</code>	Same as <code>jge</code> , but an unsigned comparison is performed.
<code>jl loc</code>	Performs signed comparison jump after a <code>cmp</code> if the destination operand is less than the source operand.
<code>jle loc</code>	Performs signed comparison jump after a <code>cmp</code> if the destination operand is less than or equal to the source operand.
<code>jb loc</code>	Same as <code>jl</code> , but an unsigned comparison is performed.



<code>jbe loc</code>	Same as <code>jle</code> , but an unsigned comparison is performed.
<code>jo loc</code>	Jump if the previous instruction set the overflow flag (OF = 1).
<code>js loc</code>	Jump if the sign flag is set (SF = 1).
<code>jecxz loc</code>	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 2](#).)

The most common data buffer manipulation instructions are `movsx`, `cmps`, `stos`, and `scas`, 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.

In x86, the repeat prefixes are used for multibyte operations. The `rep` instruction increments the ESI and EDI offsets, and decrements the ECX register. The `rep` prefix will continue until ECX = 0. The `repe/repz` and `repne/repnz` prefixes will continue until ECX = 0 or until the ZF = 1 or 0. This is illustrated in [Table 5-10](#). Therefore, in most data buffer manipulation instructions, ESI, EDI, and ECX must be properly initialized for the `rep` instruction to be useful.

Table 5-10. `rep` Instruction Termination Requirements

Instruction	Description
-------------	-------------

<code>rep</code>	Repeat until ECX = 0
<code>repe, repz</code>	Repeat until ECX = 0 or ZF = 0
<code>repne, repnz</code>	Repeat until ECX = 0 or ZF = 1

The `movsb` instruction is used to move a sequence of bytes from one location to another. The `rep` prefix is commonly used with `movsb` to copy a sequence of bytes, with size defined by ECX. The `rep movsb` instruction is the logical equivalent of the C `memcpy` function. The `movsb` instruction grabs the byte at address ESI, stores it at address EDI, and then increments or decrements the ESI and EDI registers by one according to the setting of the direction flag (DF). If DF = 0, they are incremented; otherwise, they are decremented.

You rarely see this in compiled C code, but in shellcode, people will sometimes flip the direction flag so they can store data in the reverse direction. If the `rep` prefix is present, the ECX is checked to see if it contains zero. If not, then the instruction moves the byte from ESI to EDI and decrements the ECX register. This process repeats until ECX = 0.

The `cmpsb` instruction is used to compare two sequences of bytes to determine whether they contain the same data. The `cmpsb` instruction subtracts the value at location EDI from the value at ESI and updates the flags. It is typically used with the `repe` prefix. When coupled with the `repe` prefix, the `cmpsb` instruction compares each byte of the two sequences until it finds a difference between the sequences or reaches the end of the comparison. The `cmpsb` instruction obtains the byte at address ESI, compares the value at location EDI to set the flags, and then increments the ESI and EDI registers by one. If the `repe` prefix is present, ECX is checked and the flags are also checked, but if ECX = 0 or ZF = 0, the operation will stop repeating. This is equivalent to the C function `memcmp`.

The `scasb` instruction is used to search for a single value in a sequence of bytes. The value is defined by the AL register. This works in the same way as `cmpsb`, but it compares the byte located at address ESI to AL, rather than to EDI. The `repe` operation will continue until the byte is found or ECX = 0. If the value is found in the sequence of bytes, ESI stores the location of that value.

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 5-11 displays some common `rep` instructions and describes their operation.

Table 5-11. `rep` Instruction Examples

Instruction	Description
<code>repe cmpsb</code>	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 <code>ECX = 0</code> or the buffers are not equal.
<code>rep stosb</code>	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 <code>xor eax, eax</code> .
<code>rep movsb</code>	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 <code>ECX = 0</code> .
<code>repne scasb</code>	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 <code>ECX = 0</code> 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

```

**Example 5-1** shows the C code for a simple program.

#### Example 5-1. C code, main method example

```

int main(int argc, char* argv[])
{
    if (argc != 3) {return 0;}

    if (strncmp(argv[1], "-r", 2) == 0){
        DeleteFileA(argv[2]);
    }
    return 0;
}

```

**Example 5-2** shows the C code from **Example 5-1** in compiled form. This example will help you understand how the parameters listed in Table 4-12 are accessed in assembly code. `argc` is compared to 3 at **1**, and `argv[1]` is compared to `-r` at **2** through the use of a `strncmp`. Notice how `argv[1]` is accessed: First the location of the beginning of the array is loaded into `eax`, and then 4 (the offset) is added to `eax` to get `argv[1]`. The number 4 is used because each entry in the `argv` array is an address to a string, and each address is 4 bytes in size on a 32-bit system. If `-r` is provided on the command line, the code starting at **3** will be executed, which is when we see `argv[2]` accessed at offset 8 relative to `argv` and provided as an argument to the `DeleteFileA` function.

#### Example 5-2. Assembly code, C main method parameters

004113CE	cmp	[ebp+ <b>argc</b> ], 3 <b>1</b>
004113D2	jz	short loc_4113D8
004113D4	xor	eax, eax
004113D6	jmp	short loc_411414
004113D8	mov	esi, esp
004113DA	push	2 ; MaxCount
004113DC	push	offset Str2 ; "-r"
004113E1	mov	eax, [ebp+ <b>argv</b> ]
004113E4	mov	ecx, [eax+4]
004113E7	push	ecx ; Str1
004113E8	call	strncmp <b>2</b>
004113F8	test	eax, eax
004113FA	jnz	short loc_411412
004113FC	mov	esi, esp <b>3</b>
004113FE	mov	eax, [ebp+ <b>argv</b> ]
00411401	mov	ecx, [eax+8]
00411404	push	ecx ; lpFileName

## 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 I**, and Volume 3B: System Programming Guide, **Part II*****

- 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 7** 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.

# Chapter 6. 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 6-1** 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 **1** with Intel x86 architecture **2**. 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 **3**. 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 10**. 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 **4** in **Figure 6-1**, and you'll see an input box where you can specify the new virtual base address in which to load the file.



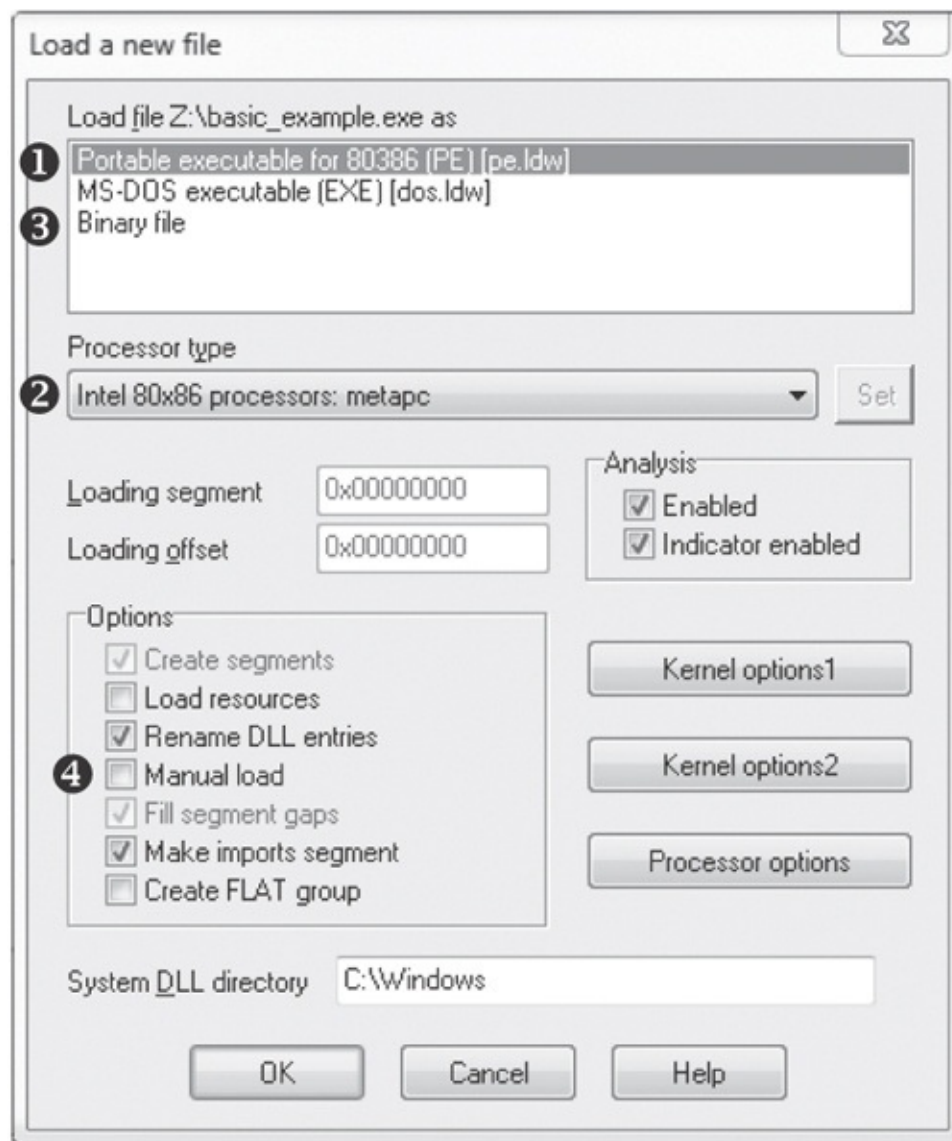


Figure 6-1. Loading a file in IDA Pro

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 6-2**. 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 6-2**) 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**.)

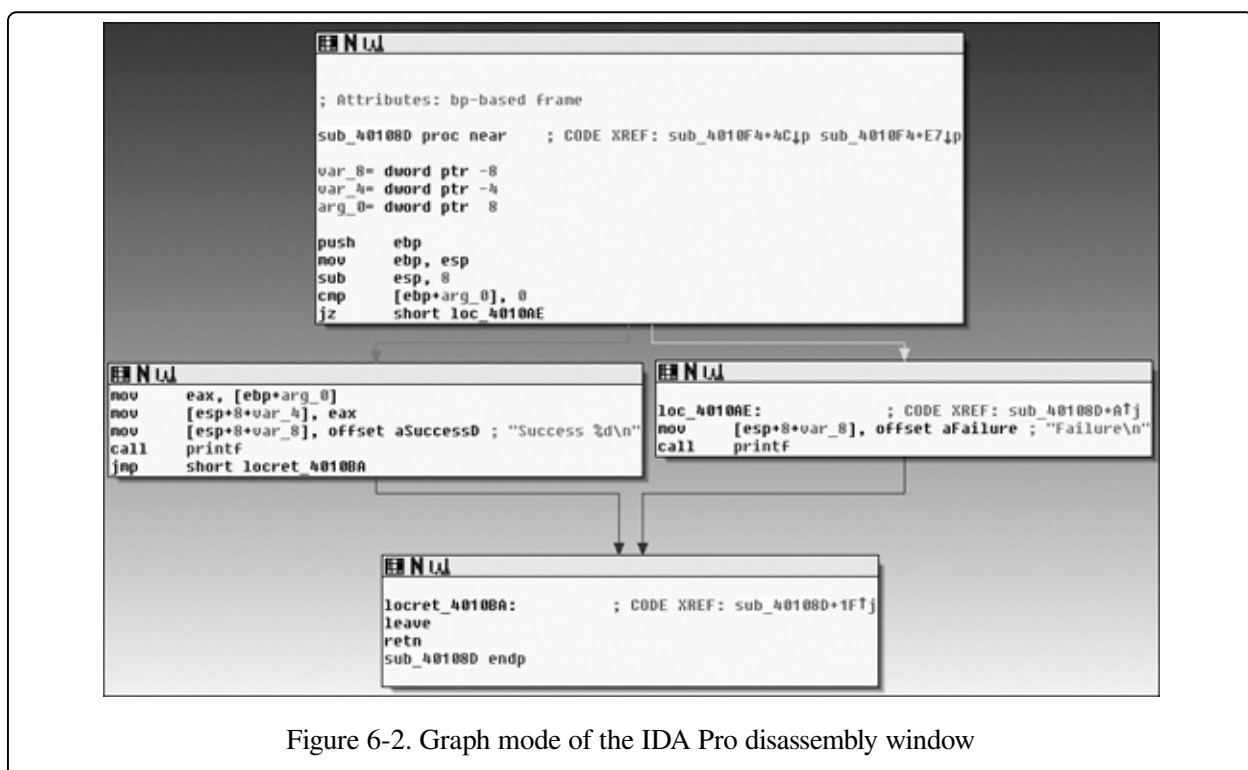


Figure 6-2. Graph mode of the IDA Pro disassembly window

In graph mode, the color and direction of the arrows help show the program's flow during analysis. The arrow's color tells you whether the path is based on a particular decision having been made: red if a conditional jump is not taken, green if the jump is taken, and blue for an unconditional jump. The arrow direction shows the program's flow; upward arrows typically denote a loop situation. Highlighting text in graph mode highlights every instance of that text in the disassembly window.

## Text Mode

The text mode of the disassembly window is a more traditional view, and you must use it to view data regions of a binary. **Figure 6-3** displays the text mode view of a disassembled function. It displays the memory address (0040105B) and section name (.text) in which the opcodes (83EC18) will reside in memory **1**.

The left portion of the text-mode display is known as the arrows window and shows the program's nonlinear flow. Solid lines mark unconditional jumps, and dashed lines mark conditional jumps. Arrows facing up indicate a loop. The example includes the stack layout for the function at **2** and a comment (beginning with a semicolon) that was automatically added by IDA Pro **3**.

### NOTE

If you are still learning assembly code, you should find the auto comments feature of IDA Pro useful. To turn on this feature, select **Options ► General**, and then check the **Auto comments** checkbox. This adds additional comments throughout the disassembly window to aid your analysis.



- **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 [Example 6-1](#). Double-clicking any of these links **1** will display the target location in the disassembly window.

Example 6-1. Navigational links within the disassembly window

```
00401075      jnz      short 1 loc_40107E
00401077      mov      [ebp+var_10], 1
0040107E loc_40107E:      ; CODE XREF: 1 2 sub_401040+35j
```

```

0040107E      cmp     [ebp+var_C], 0
00401082      jnz     short 1 loc_401097
00401084      mov     eax, [ebp+var_4]
00401087      mov     [esp+18h+var_14], eax
0040108B      mov     [esp+18h+var_18], offset 1 aPrintNumberD ; "Print Number=
%d\n"
00401092      call    1printf
00401097      call    1sub_4010A0

```

The following are the most common types of links:

- Sub links are links to the start of functions such as `printf` and `sub_4010A0`.
- Loc links are links to jump destinations such as `loc_40107E` and `loc_401097`.
- Offset links are links to an offset in memory.

Cross-references (shown at **2** in the listing) are useful for jumping the display to the referencing location: `0x401075` in this example. Because strings are typically references, they are also navigational links. For example, `aPrintNumberD` can be used to jump the display to where that string is defined in memory.

## Exploring Your History

IDA Pro's forward and back buttons, shown in **Figure 6-4**, make it easy to move through your history, just as you would move through a history of web pages in a browser. Each time you navigate to a new location within the disassembly window, that location is added to your history.

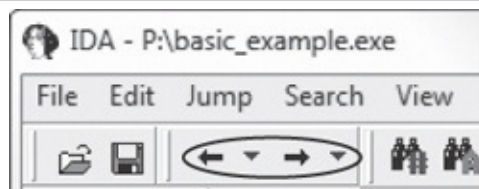


Figure 6-4. Navigational buttons

## Navigation Band

The horizontal color band at the base of the toolbar is the navigation band, which presents a color-coded linear view of the loaded binary's address space. The colors offer insight into the file contents at that location in the file as follows:

- Light blue is library code as recognized by FLIRT.
- Red is compiler-generated code.

- Dark blue is user-written code.

You should perform malware analysis in the dark-blue region. If you start getting lost in messy code, the navigational band can help you get back on track. IDA Pro's default colors for data are pink for imports, gray for defined data, and brown for undefined data.

#### NOTE

If you have an older version of IDA Pro, your FLIRT signatures may not be up to date and you can end up with a lot of library code in the dark-blue region. FLIRT isn't perfect, and sometimes it won't recognize and label all library code properly.

## Jump to Location

To jump to any virtual memory address, simply press the G key on your keyboard while in the disassembly window. A dialog box appears, asking for a virtual memory address or named location, such as `sub_401730` or `printf`.

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 6-5**. We notice that **Bad key** is used at 0x401104 **1**, so we jump to that location in the disassembly window by double-clicking the entry in the search window.



Figure 6-5. 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

**Example 6-2** shows a code cross-reference at **1** 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 **2** tells us which jump takes us to this location, which in this example corresponds to the location marked at **3**. We know this because at offset 0x19 into sub\_401000 is the jmp at memory address 0x401019.

Example 6-2. Code cross-references

```
00401000      sub_401000      proc near      ; 1CODE XREF: main+3p
00401000      push      ebp
00401001      mov       ebp, esp
00401003  loc401003:      ; 2CODE 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 3
```

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 6-6**, which shows a list of cross-references for sub\_408980, you can see that this function is called 64 times (“Line 1 of 64”).

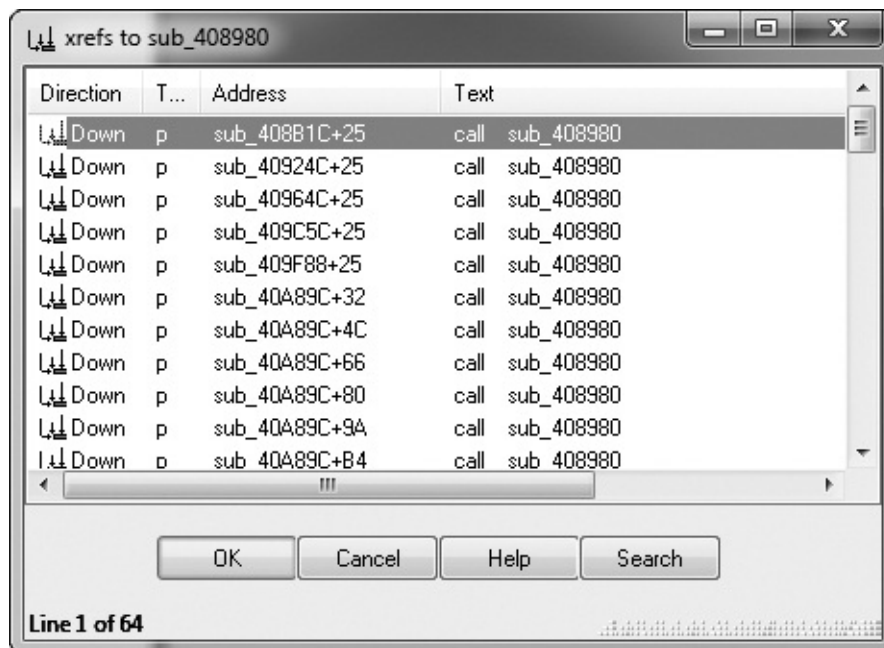


Figure 6-6. Xrefs window

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 [Example 6-3](#). For example, you can see the data cross-reference to the DWORD 0x7F000001 at [1](#). 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>.

Example 6-3. Data cross-references

```
0040C000 dword_40C000    dd 7F000001h          ; 1DATA XREF: sub_401020+14r
0040C004 aHostnamePort    db '<Hostname> <Port>',0Ah,0 ; DATA XREF: sub_401000+30
```

Recall from [Chapter 2](#) 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. **Example 6-4** shows an example of a function that has been recognized by IDA Pro.

## Example 6-4. Function and stack example

```
00401020 ; ===== S U B R O U T I N E =====
00401020
00401020 ; Attributes: ebp-based frame 1
00401020
00401020 function      proc near          ; CODE XREF: main+1Cp
00401020
00401020 varC             = dword ptr -0Ch 2
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 3
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
```

Notice how IDA Pro tells us that this is an EBP-based stack frame used in the function **1**, 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 5** that local variables will be at a negative offset relative to EBP and arguments will be at a positive offset. You can see at **2** 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 **3**; 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 6-7**. Four of these graphing options utilize cross-references.



Figure 6-7. 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 6-1**.

Table 6-1. 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 <b>Chapter 7</b> .)
	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 <b>Figure 6-8</b> . 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.
	Graphs the cross-references from the	This is a useful way to see a series of function calls. For example, <b>Figure 6-9</b> displays this type of graph for a single function. Notice how sub_4011f0 calls sub_401110, which then calls gethostbyname. This view can quickly tell you what a function does and what the functions do

currently  
selected  
symbol

underneath it. This is the easiest way to get a quick overview of the  
function.



Graphs a  
user-specified  
cross-  
reference  
graph

Use this option to build a custom graph. You can specify the graph's  
recursive depth, the symbols used, the to or from symbol, and the types of  
nodes to exclude from the graph. This is the only way to modify graphs  
generated by IDA Pro for display in WinGraph32.

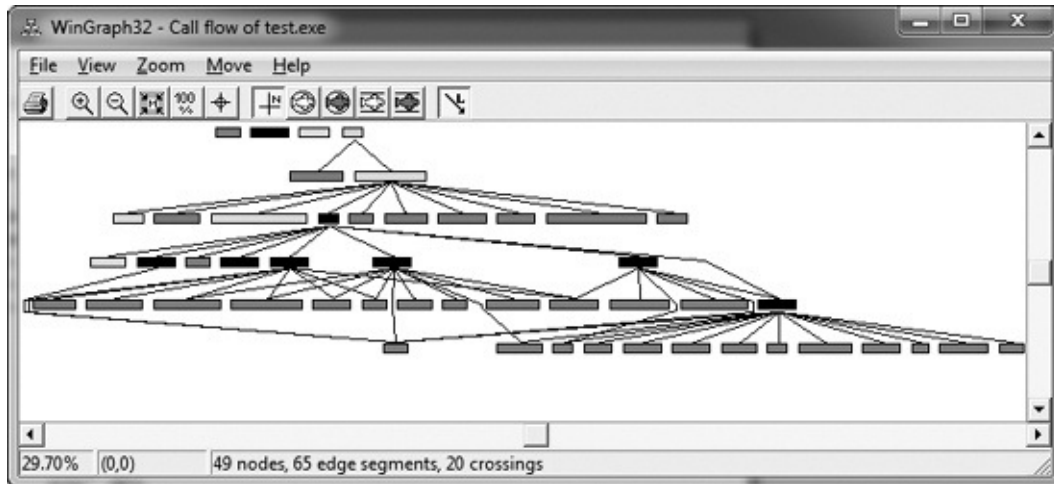


Figure 6-8. Cross-reference graph of a program

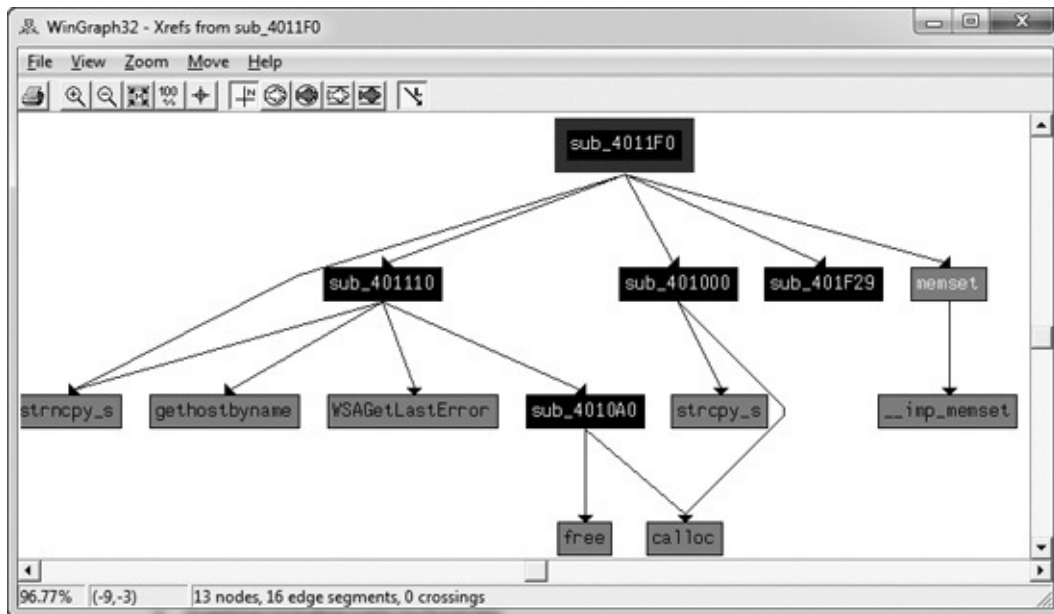


Figure 6-9. Cross-reference graph of a single function (sub\_4011F0)

# 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 6-2** 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.

Table 6-2. Function Operand Manipulation

Without renamed arguments	With renamed arguments
004013C8 mov eax, [ebp+arg_4]	004013C8 mov eax, [ebp+port_str]
004013CB push eax	004013CB push eax
004013CC call _atoi	004013CC call _atoi
004013D1 add esp, 4	004013D1 add esp, 4
004013D4 mov [ebp+var_598], ax	004013D4 mov [ebp+port], ax
004013DB movzx ecx, [ebp+var_598]	004013DB movzx ecx, [ebp+port]
004013E2 test ecx, ecx	004013E2 test ecx, ecx
004013E4 jnz short loc_4013F8	004013E4 jnz short loc_4013F8
004013E6 push offset aError	004013E6 push offset aError
004013EB call printf	004013EB call printf
004013F0 add esp, 4	004013F0 add esp, 4
004013F3 jmp loc_4016FB	004013F3 jmp loc_4016FB
004013F8 ; -----	004013F8 ; -----
004013F8	004013F8
004013F8 loc_4013F8:	004013F8 loc_4013F8:
004013F8 movzx edx, [ebp+var_598]	004013F8 movzx edx, [ebp+port]
004013FF push edx	004013FF push edx
00401400 call ds:htons	00401400 call ds:htons

Figure 6-10 shows an example of modifying operands in an instruction, where 62h is compared to the local variable var\_4. If you were to right-click 62h, you would be presented with options to change the 62h into 98 in decimal, 142o in octal,

1100010b in binary, or the character b in ASCII—whatever suits your needs and your situation.

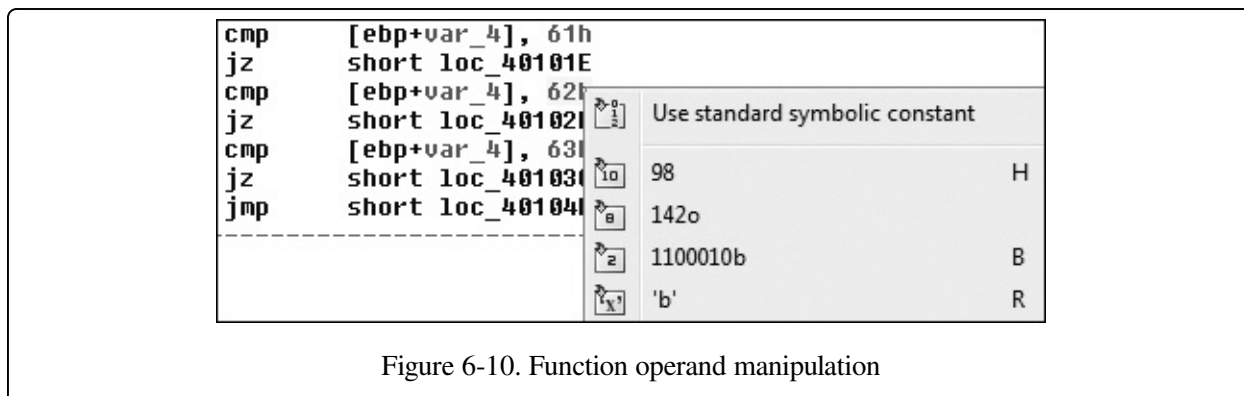


Figure 6-10. Function operand manipulation

To change whether an operand references memory or stays as data, press the O key on your keyboard. For example, suppose when you're analyzing disassembly with a link to `loc_410000`, you trace the link back and see the following instructions:

```
mov eax, loc_410000
add ebx, eax
mul ebx
```

At the assembly level, everything is a number, but IDA Pro has mislabeled the number 4259840 (0x410000 in hex) as a reference to the address 410000. To correct this mistake, press the O key to change this address to the number 410000h and remove the offending cross-reference from the disassembly window.

## 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 6-10) on an operand in your disassembly. Figure 6-11 shows the window that appears when you select Use Standard Symbolic Constant on the value `0x80000000`.

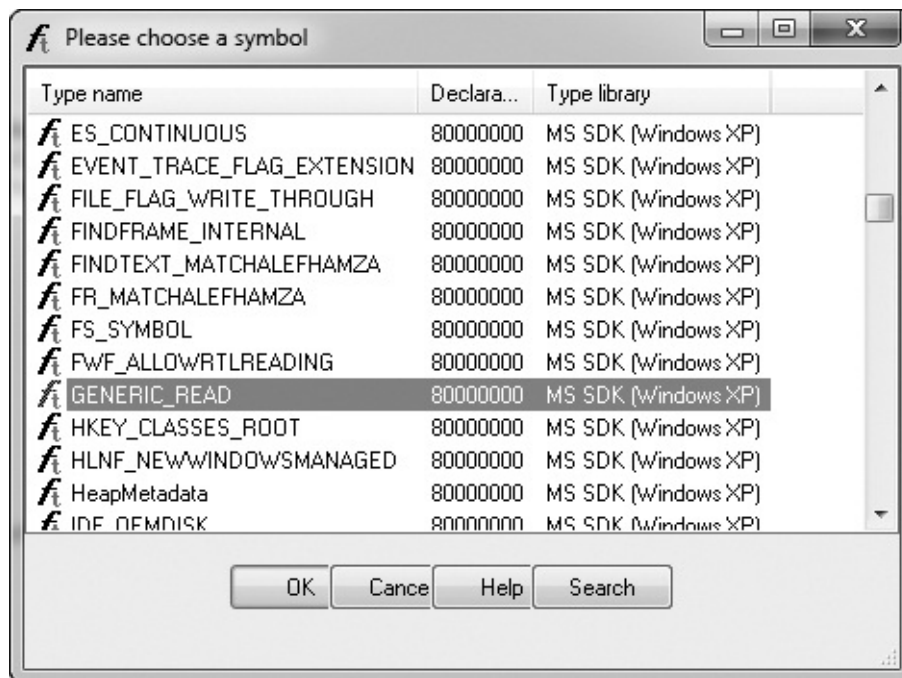


Figure 6-11. Standard symbolic constant window

The code snippets in [Table 6-3](#) 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 8](#), 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 `gnuunix` (GNU C++ UNIX) libraries.

Table 6-3. Code Before and After Standard Symbolic Constants

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

## 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 6-4](#) shows a malicious PDF document named `paycuts.pdf`. At offset 0x8387 into the file, we discover shellcode (defined as raw bytes) at [1](#), 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 [2](#).

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 6-12**. 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.

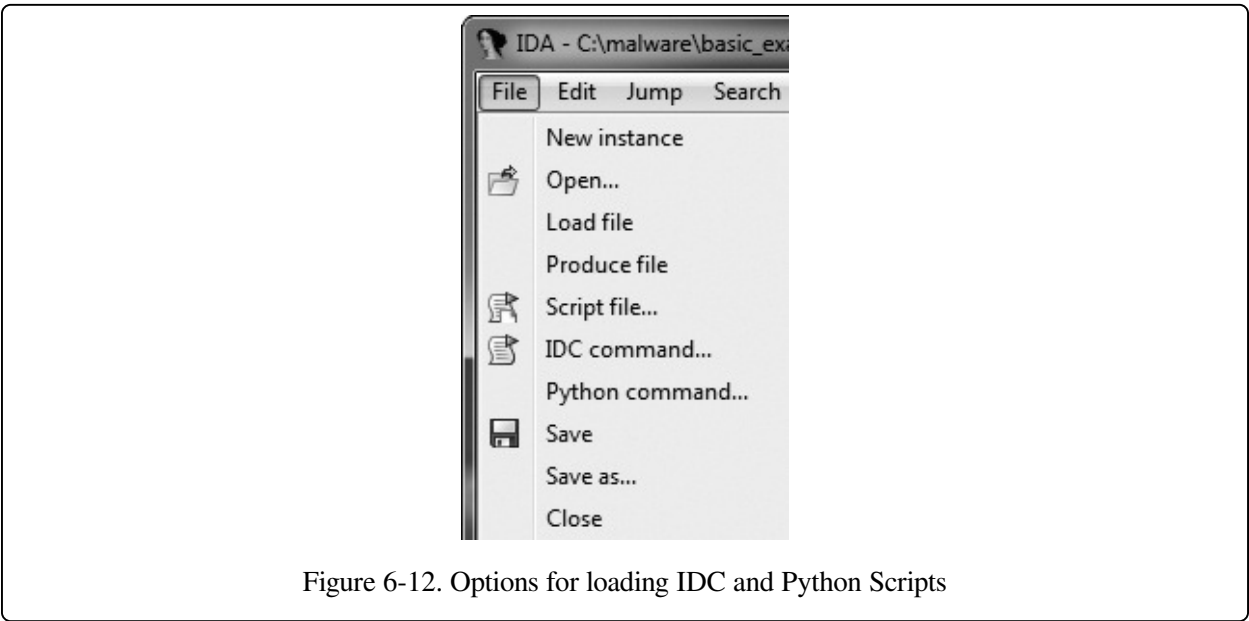


Figure 6-12. Options for loading IDC and Python Scripts

Table 6-4. Manually Disassembling Shellcode in the paycuts.pdf Document

File before pressing C	File after pressing C
00008384 db 28h ; (	00008384 db 28h ; (
28h ; (	00008385 db 0FCh ; n
00008385 db 0FCh ; n	00008386 db 10h
00008386 db 10h	00008387 nop
00008387 db 10h	00008388 nop
00008387 db 10h	00008389 mov ebx, eax
	0000838B add ebx, 28h ; '('

90h ; É 1	0000838E add dword ptr [ebx], 1Bh
00008388 db	00008391 mov ebx, [ebx]
90h ; É	00008393 xor ecx, ecx
00008389 db	00008395
8Bh ; Ĩ	00008395 loc_8395: ; CODE
0000838A db	XREF: seg000:000083A0j
0D8h ; +	00008395 xor byte ptr [ebx], 97h 2
0000838B db	00008398 inc ebx
83h ; â	00008399 inc ecx
0000838C db	0000839A cmp ecx, 700h
0C3h ; +	000083A0 jnz short loc_8395
0000838D db	000083A2 retn 7B1Ch
28h ; (	000083A2 ; -----000083A5
0000838E db	db 16h
83h ; â	000083A6 db 7Bh ; {
0000838F db	000083A7 db 8Fh ; Å
3	
00008390 db	
1Bh	
00008391 db	
8Bh ; Ĩ	
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
```

---

## 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 2](#), 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 [Example 6-5](#).

Example 6-5. IDC script generated by the PEiD KANAL plug-in

```
#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):
        SB0X (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 SB0X (for decryption)");
}
```

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.

**Example 6-6** 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.

Example 6-6. Useful Python script to color all function calls

```
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)
```

## 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.



ynamics 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

Q: 1. What is the address of `DllMain`?

Q: 2. Use the Imports window to browse to `gethostbyname`. Where is the import located?

Q: 3. How many functions call `gethostbyname`?

Q: 4. Focusing on the call to `gethostbyname` located at 0x10001757, can you figure out which DNS request will be made?

Q: 5. How many local variables has IDA Pro recognized for the subroutine at 0x10001656?

Q: 6. How many parameters has IDA Pro recognized for the subroutine at 0x10001656?

Q: 7. Use the Strings window to locate the string `\cmd.exe /c` in the disassembly. Where is it located?

Q: 8. What is happening in the area of code that references `\cmd.exe /c`?

Q: 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.)

Q: 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)?

Q: 11. What does the export `PSLIST` do?

Q: 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?

Q: 13. How many Windows API functions does `DllMain` call directly? How many at a depth of 2?

Q: 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?

Q: 15. At 0x10001701 is a call to `socket`. What are the three parameters?

Q: 16. Using the MSDN page for `socket` and the named symbolic constants functionality in IDA Pro, can you make the parameters more meaningful? What are the parameters after you apply changes?

Q: 17. Search for usage of the `in` instruction (opcode 0xED). This instruction is used with a magic string VMXh to perform VMware detection. Is that in use in this malware? Using the cross-references to the function that executes the `in` instruction, is there further evidence of VMware detection?

Q: 18. Jump your cursor to 0x1001D988. What do you find?

Q: 19. If you have the IDA Python plug-in installed (included with the commercial version of IDA Pro), run Lab05-01.py, an IDA Pro Python script provided with the malware for this book. (Make sure the cursor is at 0x1001D988.) What happens after you run the script?

Q: 20. With the cursor in the same location, how do you turn this data into a single ASCII string?

Q: 21. Open the script with a text editor. How does it work?

# Chapter 7. Recognizing C Code Constructs in Assembly

In [Chapter 5](#), 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 reverse: Your goal as a malware analyst will be to go from disassembly to high-level constructs. Learning in this reverse direction is often easier, because computer programmers are accustomed to reading and understanding source code.

This chapter will focus on how the most common and difficult constructs, such as loops and conditional statements, are compiled. After you've built a foundation with these, you'll learn how to develop a high-level picture of code functionality quickly.

In addition to discussing the different constructs, we'll also examine the differences between compilers, because compiler versions and settings can impact how a particular construct appears in disassembly. We'll evaluate two different ways that `switch` statements and function calls can be compiled using different compilers.

This chapter will dig fairly deeply into C code constructs, so the more you understand about C and programming in general, the more you'll get out of it. For help with the C language, have a look at the classic *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, 1988). Most malware is written in C, although it is sometimes written in Delphi and C++. C is a simple language with a close relationship to assembly, so it is the most logical place for a new malware analyst to start.

As you read this chapter, remember that your goal is to understand the overall functionality of a program, not to analyze every single instruction. Keep this in mind, and don't get bogged down with the minutiae. Focus on the way programs work in general, not on how they do each particular thing.

## Global vs. Local Variables

Global variables can be accessed and used by any function in a program. Local variables can be accessed only by the function in which they are defined. Both global and local variables are declared similarly in C, but they look completely different in assembly.

Following are two examples of C code for both global and local variables. Notice the subtle difference between the two. The global example, [Example 7-1](#), defines *x* and *y* variables outside the function. In the local example, [Example 7-2](#), the variables are defined within the function.

Example 7-1. A simple program with two global variables

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

void main()
{
    x = x+y;
    printf("Total = %d\n", x);
}
```

Example 7-2. A simple program with two local variables

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

    x = x+y;
    printf("Total = %d\n", x);
}
```

The difference between the global and local variables in these C code examples is small, and in this case the program result is the same. But the disassembly, shown in [Example 7-3](#) and [Example 7-4](#), is quite different. The global variables are referenced by memory addresses, and the local variables are referenced by the stack addresses.

In [Example 7-3](#), the global variable `x` is signified by  `dword_40CF60`, a memory location at `0x40CF60`. Notice that `x` is changed in memory when `eax` is moved into  `dword_40CF60` at **1**. All subsequent functions that utilize this variable will be impacted.

Example 7-3. Assembly code for the global variable example in [Example 7-1](#)

```
00401003      mov     eax, dword_40CF60
00401008      add     eax, dword_40C000
0040100E      mov     dword_40CF60, eax 1
00401013      mov     ecx, dword_40CF60
00401019      push    ecx
0040101A      push    offset aTotalD ; "total = %d\n"
0040101F      call    printf
```

In [Example 7-4](#) and [Example 7-5](#), the local variable `x` is located on the stack at a constant offset relative to `ebp`. In [Example 7-4](#), memory location `[ebp-4]` is used consistently throughout this function to reference the local variable `x`. This tells us that `ebp-4` is a stack-based local variable that is referenced only in the function in which it is defined.

Example 7-4. Assembly code for the local variable example in [Example 7-2](#), without labeling

```
00401006      mov     dword ptr [ebp-4], 0
0040100D      mov     dword ptr [ebp-8], 1
00401014      mov     eax, [ebp-4]
00401017      add     eax, [ebp-8]
0040101A      mov     [ebp-4], eax
0040101D      mov     ecx, [ebp-4]
00401020      push    ecx
00401021      push    offset aTotalD ; "total = %d\n"
00401026      call    printf
```

In [Example 7-5](#), `x` has been nicely labeled by IDA Pro Disassembler with the dummy name `var_4`. As we discussed in [Chapter 6](#), dummy names can be renamed to meaningful names that reflect their function. Having this local variable named `var_4` instead of `-4` simplifies your analysis, because once you rename `var_4` to `x`, you won't need to track the offset `-4` in your head throughout the

function.

Example 7-5. Assembly code for the local variable example shown in [Example 7-2](#), with labeling

```
00401006      mov     [ebp+var_4], 0
0040100D      mov     [ebp+var_8], 1
00401014      mov     eax, [ebp+var_4]
00401017      add     eax, [ebp+var_8]
0040101A      mov     [ebp+var_4], eax
0040101D      mov     ecx, [ebp+var_4]
00401020      push    ecx
00401021      push    offset aTotalD ; "total = %d\n"
00401026      call    printf
```



# Disassembling Arithmetic Operations

Many different types of math operations can be performed in C programming, and we'll present the disassembly of those operations in this section.

**Example 7-6** shows the C code for two variables and a variety of arithmetic operations. Two of these are the `--` and `++` operations, which are used to decrement by 1 and increment by 1, respectively. The `%` operation performs the modulo between the two variables, which is the remainder after performing a division operation.

Example 7-6. C code with two variables and a variety of arithmetic

```
int a = 0;
int b = 1;
a = a + 11;
a = a - b;
a--;
b++;
b = a % 3;
```

**Example 7-7** shows the assembly for the C code shown in **Example 7-6**, which can be broken down to translate back to C.

Example 7-7. Assembly code for the arithmetic example in **Example 7-6**

```
00401006    mov     [ebp+var_4], 0
0040100D    mov     [ebp+var_8], 1
00401014    mov     eax, [ebp+var_4] 1
00401017    add     eax, 0Bh
0040101A    mov     [ebp+var_4], eax
0040101D    mov     ecx, [ebp+var_4]
00401020    sub     ecx, [ebp+var_8] 2
00401023    mov     [ebp+var_4], ecx
00401026    mov     edx, [ebp+var_4]
00401029    sub     edx, 1 3
0040102C    mov     [ebp+var_4], edx
0040102F    mov     eax, [ebp+var_8]
00401032    add     eax, 1 4
00401035    mov     [ebp+var_8], eax
00401038    mov     eax, [ebp+var_4]
0040103B    cdq
0040103C    mov     ecx, 3
00401041    idiv    ecx
00401043    mov     [ebp+var_8], edx 5
```

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` **1**, and then `0x0b` is added

to `eax`, thereby incrementing `a` by 11. `b` is then subtracted from `a` **2**. (The compiler decided to use the `sub` and `add` instructions **3** and **4**, instead of the `inc` and `dec` functions.)

The final five assembly instructions implement the modulo. When performing the `div` or `idiv` instruction **5**, 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` **5**.

# 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.

**Example 7-8** displays a simple `if` statement in C with the assembly for this code shown in **Example 7-9**. Notice the conditional jump `jnz` at **2**. There must be a conditional jump for an `if` statement, but not all conditional jumps correspond to `if` statements.

Example 7-8. C code `if` statement example

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

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

Example 7-9. Assembly code for the `if` statement example in **Example 7-8**

```
00401006      mov     [ebp+var_8], 1
0040100D      mov     [ebp+var_4], 2
00401014      mov     eax, [ebp+var_8]
00401017      cmp     eax, [ebp+var_4] 1
0040101A      jnz     short loc_40102B 2
0040101C      push    offset aXEqualsY_ ; "x equals y.\n"
00401021      call    printf
00401026      add     esp, 4
00401029      jmp     short loc_401038 3
0040102B loc_40102B:
0040102B      push    offset aXIsNotEqualToY ; "x is not equal to y.\n"
00401030      call    printf
```

As you can see in **Example 7-9**, a decision must be made before the code inside the `if` statement in **Example 7-8** will execute. This decision corresponds to the conditional jump (`jnz`) shown at **2**. 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 **1**. 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 **3**. 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 7-1](#). This feature is the default view for analyzing functions.

[Figure 7-1](#) shows a graph of the assembly code example in [Example 7-9](#). As you can see, two different paths ([1](#) and [2](#)) of code execution lead to the end of the function, and each path prints a different string. Code path [1](#) will print "x equals y.", and [2](#) will print "x is not equal to y."

IDA Pro adds `false` [1](#) and `true` [2](#) 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.

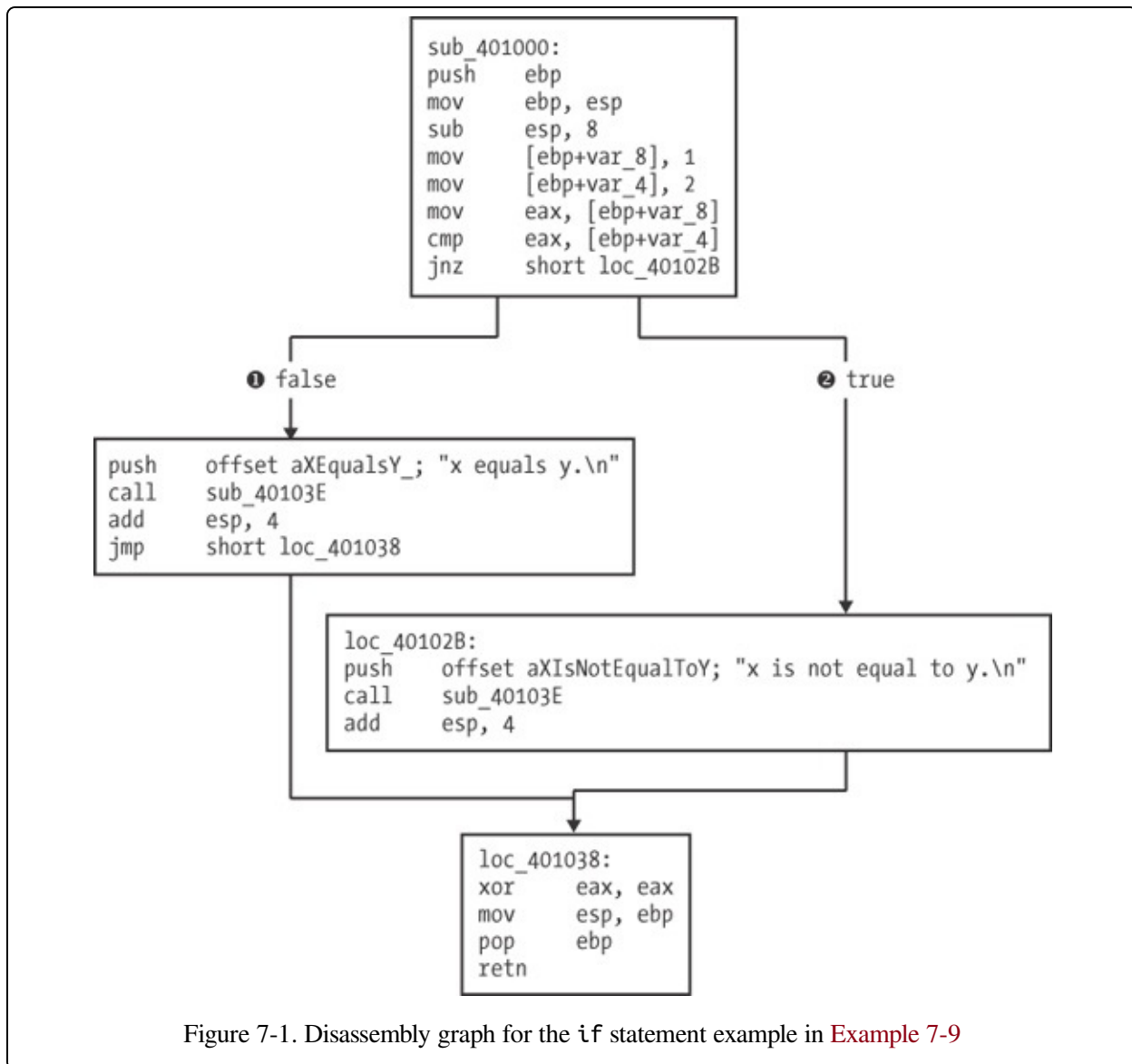
## Recognizing Nested if Statements

[Example 7-10](#) shows C code for a nested `if` statement that is similar to [Example 7-8](#), 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.

Example 7-10. C code for a nested `if` statement

```
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");
    }
}
```



Despite this minor change to the C code, the assembly code is more complicated, as shown in Example 7-11.

Example 7-11. Assembly code for the nested `if` statement example shown in Example 7-10

```

00401006      mov     [ebp+var_8], 0
0040100D      mov     [ebp+var_4], 1
00401014      mov     [ebp+var_C], 2
0040101B      mov     eax, [ebp+var_8]
0040101E      cmp     eax, [ebp+var_4]
00401021      jnz     short loc_401047 1
00401023      cmp     [ebp+var_C], 0
00401027      jnz     short loc_401038 2
00401029      push    offset aZIsZeroAndXY_ ; "z is zero and x = y.\n"
0040102E      call    printf

```

```

00401033      add     esp, 4
00401036      jmp     short loc_401045
00401038 loc_401038:
00401038      push    offset aZIsNonZeroAndX ; "z is non-zero and x = y.\n"
0040103D      call    printf
00401042      add     esp, 4
00401045 loc_401045:
00401045      jmp     short loc_401069
00401047 loc_401047:
00401047      cmp     [ebp+var_C], 0
0040104B      jnz     short loc_40105C 3
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

```

As you can see, three different conditional jumps occur. The first occurs if `var_4` does not equal `var_8` at **1**. The other two occur if `var_C` is not equal to zero at **2** and **3**.

# 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.

**Example 7-12** shows an example of a for loop.

Example 7-12. C code for a for loop

```
int i;

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

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 **Example 7-13**, **1** corresponds to the initialization step. The code between **3** and **4** corresponds to the increment that is initially jumped over at **2** with a jump instruction. The comparison occurs at **5**, and at **6**, the decision is made by the conditional jump. If the jump is not taken, the `printf` instruction will execute, and an unconditional jump occurs at **7**, which causes the increment to occur.

Example 7-13. Assembly code for the for loop example in **Example 7-12**

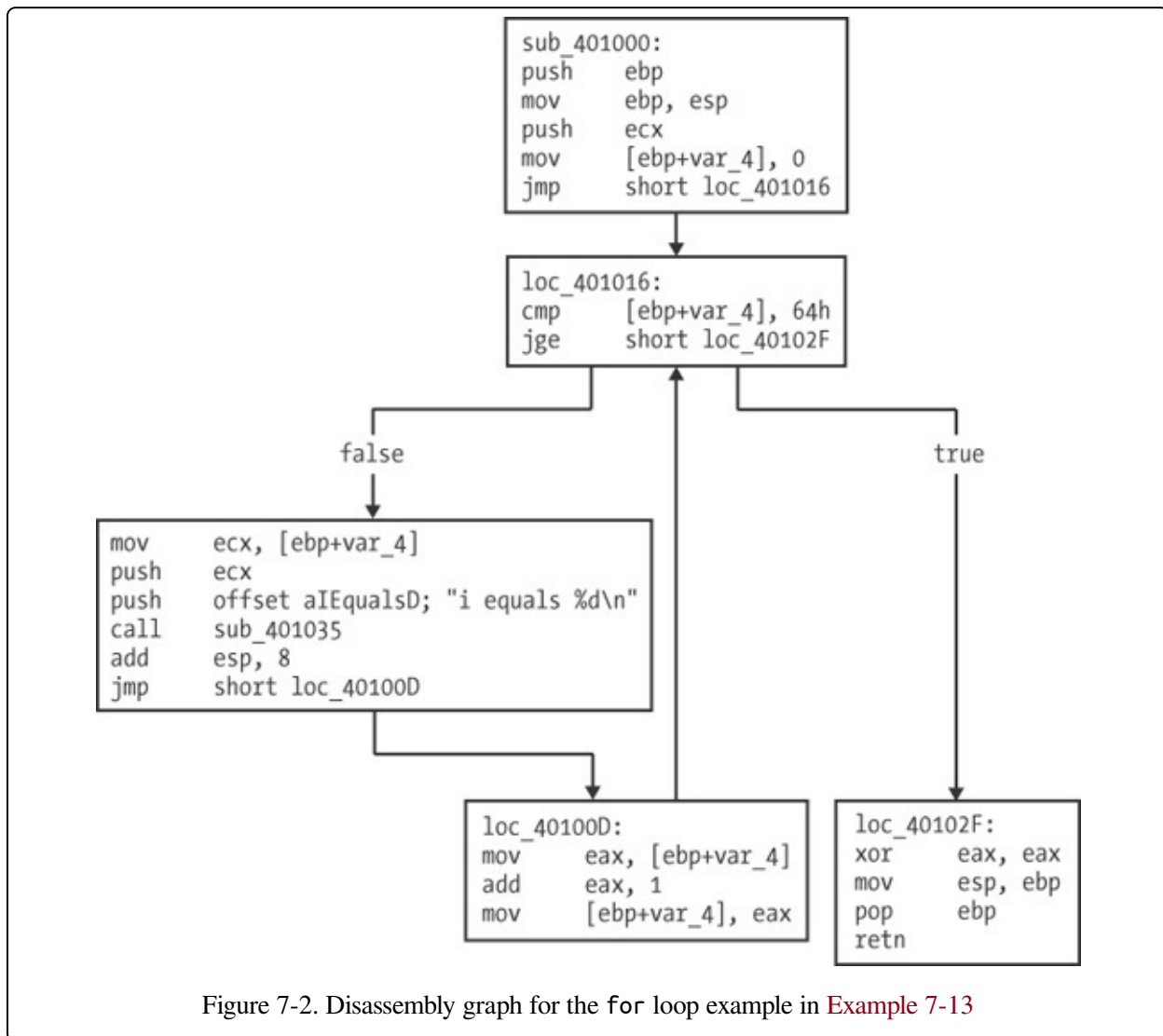
```
00401004      mov     [ebp+var_4], 0 1
0040100B      jmp     short loc_401016 2
0040100D loc_40100D:
0040100D      mov     eax, [ebp+var_4] 3
00401010      add     eax, 1
00401013      mov     [ebp+var_4], eax 4
00401016 loc_401016:
```

```

00401016    cmp     [ebp+var_4], 64h 5
0040101A    jge     short loc_40102F 6
0040101C    mov     ecx, [ebp+var_4]
0040101F    push    ecx
00401020    push    offset aID ; "i equals %d\n"
00401025    call    printf
0040102A    add     esp, 8
0040102D    jmp     short loc_40100D 7

```

A for loop can be recognized using IDA Pro's graphing mode, as shown in [Figure 7-2](#).



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 5](#) 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 [Example 7-14](#) will continue to loop until the status returned from `checkResult` is 0.

Example 7-14. C code for a `while` loop

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

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

The assembly code in [Example 7-15](#) looks similar to the `for` loop, except that it lacks an increment section. A conditional jump occurs at **1** and an unconditional jump at **2**, but the only way for this code to stop executing repeatedly is for that conditional jump to occur.

Example 7-15. Assembly code for the `while` loop example in [Example 7-14](#)

```
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 1
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 2
```

# Understanding Function Call Conventions

In [Chapter 5](#), 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 8](#)).

We will use the pseudocode in [Example 7-16](#) to describe each of the calling conventions.

Example 7-16. Pseudocode for a function call

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

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 5](#) 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. [Example 7-17](#) shows an example

of what the disassembly would look like if the code in [Example 7-16](#) were compiled to use `cdecl`.

Example 7-17. `cdecl` function call

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

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 [Example 7-17](#) 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 [Example 7-16](#) 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.

**Example 7-18** 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`.

Example 7-18. C code for a function call

```
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));
}
```

The assembly code for the `adder` function is consistent across compilers and is displayed in **Example 7-19**. As you can see, this code adds `arg_0` to `arg_4` and stores the result in `EAX`. (As discussed in **Chapter 5**, `EAX` stores the return value.)

Example 7-19. Assembly code for the `adder` function in **Example 7-18**

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

**Table 7-1** 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 7-1. 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	004010A5	mov	[esp+4], eax
00401764	push	eax	004010A9	mov	[esp], offset
00401765	push	offset	TheFunctionRet		
TheFunctionRet			004010B0	call	printf
0040176A	call	ds: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

**Example 7-20** 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.

Example 7-20. C code for a three-option switch statement

```
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;
}
```

This switch statement has been compiled into the assembly code shown in **Example 7-21**. It contains a series of conditional jumps between **1** and **2**. The conditional jump determination is made by the comparison that occurs directly before each jump.

The switch statement has three options, shown at **3**, **4**, and **5**. 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 7-3**.)

Example 7-21. Assembly code for the switch statement example in **Example 7-20**

```
00401013      cmp     [ebp+var_8], 1
00401017      jz      short loc_401027 1
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

```

**Figure 7-3** 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 **Example 7-20** the value of `i` were 3, three different comparisons would take place before the third case was executed. In

**Example 7-22**, we add one case to **Example 7-20** (as you can see by comparing the listings), but the assembly code generated is drastically different.

**Example 7-22.** C code for a four-option switch statement

```
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;
}
```



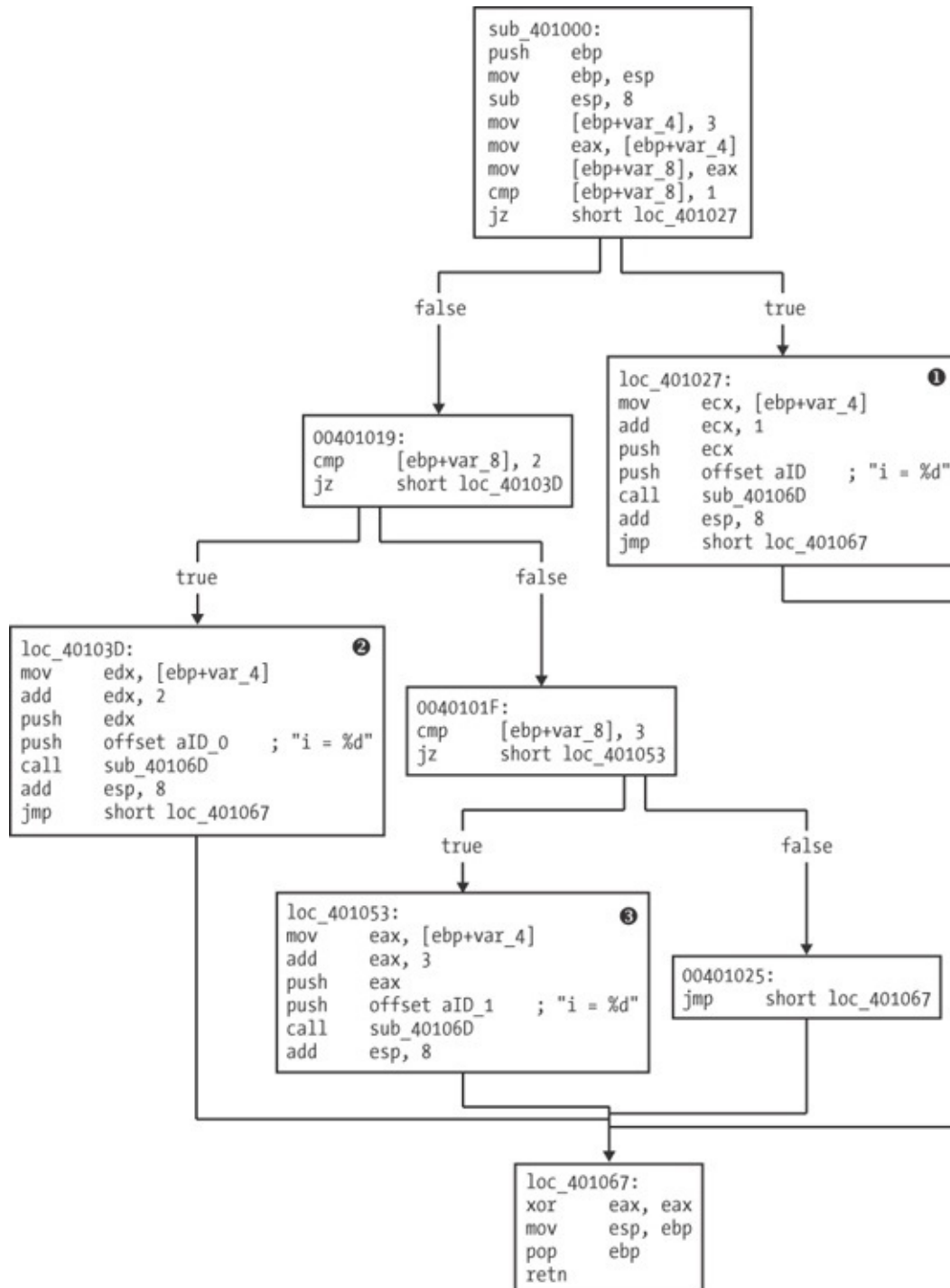


Figure 7-3. Disassembly graph of the if style switch statement example in Example 7-21

The more efficient assembly code in Example 7-23 uses a jump table, shown at 2, which defines offsets to additional memory locations. The switch variable is used as an index into the jump table.

In this example, `ecx` contains the switch variable, and 1 is subtracted from it in the first line. In the C code, the switch table range is 1 through 4, and the assembly code must adjust it to 0 through 3 so that the jump table can be properly indexed. The jump instruction at **1** is where the target is based on the jump table.

In this jump instruction, `edx` is multiplied by 4 and added to the base of the jump table (0x401088) to determine which case code block to jump to. It is multiplied by 4 because each entry in the jump table is an address that is 4 bytes in size.

Example 7-23. Assembly code for the switch statement example in [Example 7-22](#)

```

00401016      sub     ecx, 1
00401019      mov     [ebp+var_8], ecx
0040101C      cmp     [ebp+var_8], 3
00401020      ja      short loc_401082
00401022      mov     edx, [ebp+var_8]
00401025      jmp     ds:off_401088[edx*4] 1
0040102C      loc_40102C:
                ...
00401040      jmp     short loc_401082
00401042      loc_401042:
                ...
00401056      jmp     short loc_401082
00401058      loc_401058:
                ...
0040106C      jmp     short loc_401082
0040106E      loc_40106E:
                ...
00401082      loc_401082:
00401082      xor     eax, eax
00401084      mov     esp, ebp
00401086      pop     ebp
00401087      retn
00401087      _main  endp
00401088      2off_401088 dd offset loc_40102C
0040108C      dd offset loc_401042
00401090      dd offset loc_401058
00401094      dd offset loc_40106E

```

The graph in [Figure 7-4](#) for this type of switch statement is clearer than the standard disassembly view.

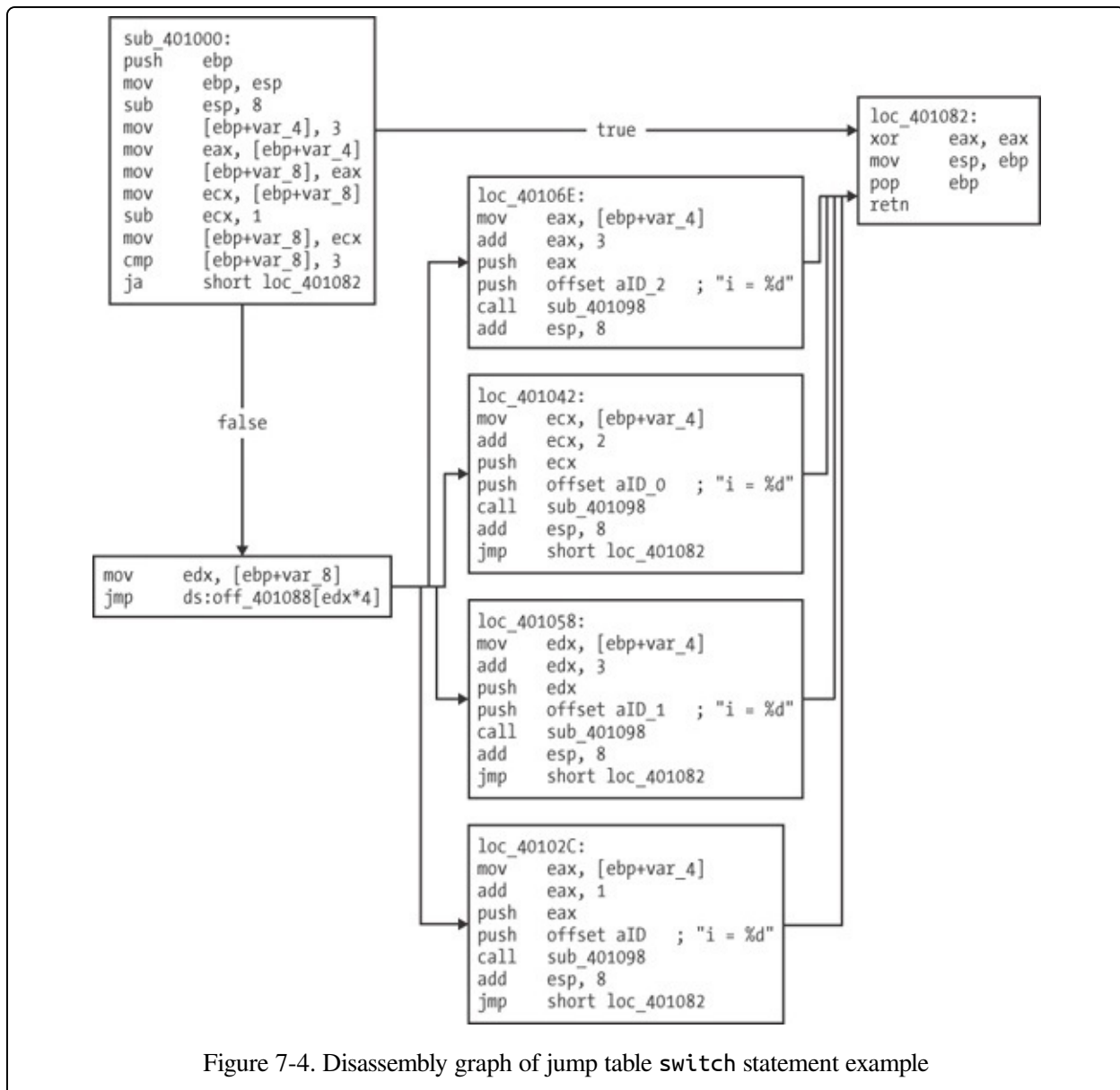


Figure 7-4. Disassembly graph of jump table switch statement example

As you can see, each of the four cases is broken down clearly into separate assembly code chunks. These chunks appear one after another in a column after the jump table determines which one to use. Notice that all of these boxes and the initial box terminate at the right box, which is the end of the function.

# 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.

**Example 7-24** 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.

Example 7-24. C code for an array

```
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;
    }
}
```

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. **Example 7-25** shows the assembly code for **Example 7-24**.

Example 7-25. Assembly code for the array in **Example 7-24**

```
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 1
00401028      mov     eax, [ebp+var_18]
0040102B      mov     ecx, [ebp+var_18]
0040102E      mov     dword_40A000[ecx*4], eax 2
00401035      jmp     short loc_40100F
```

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 **1** and **2** 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 **Example 7-26**, we define a structure at **1** 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 **2** is a global variable.

Example 7-26. C code for a struct example

```
struct my_structure { 1
    int x[5];
    char y;
    double z;
};

struct my_structure *gms; 2

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);
}
```

Structures (like arrays) are accessed with a base address used as a starting pointer. It is difficult to determine whether nearby data types are part of the same struct or whether they just happen to be next to each other. Depending on the structure's context, your ability to identify a structure can have a significant impact on your ability to analyze malware.

**Example 7-27** shows the `main` function from **Example 7-26**, disassembled. Since

the struct `gms` is a global variable, its base address will be the memory location `dword_40EA30` as shown in [Example 7-27](#). The base address of this structure is passed to the `sub_401000` (test) function via the `push eax` at [1](#).

Example 7-27. Assembly code for the `main` function in the struct example in [Example 7-26](#)

```
00401050      push     ebp
00401051      mov      ebp, esp
00401053      push     20h
00401055      call     malloc
0040105A      add      esp, 4
0040105D      mov      dword_40EA30, eax
00401062      mov      eax, dword_40EA30
00401067      push     eax 1
00401068      call     sub_401000
0040106D      add      esp, 4
00401070      xor      eax, eax
00401072      pop      ebp
00401073      retn
```

[Example 7-28](#) shows the disassembly of the `test` method shown in [Example 7-26](#). `arg_0` is the base address of the structure. Offset `0x14` stores the character within the struct, and `0x61` corresponds to the letter `a` in ASCII.

Example 7-28. Assembly code for the `test` function in the struct example in [Example 7-26](#)

```
00401000      push     ebp
00401001      mov      ebp, esp
00401003      push     ecx
00401004      mov      eax, [ebp+arg_0]
00401007      mov      byte ptr [eax+14h], 61h
0040100B      mov      ecx, [ebp+arg_0]
0040100E      fld      ds:dbl_40B120 1
00401014      fstp     qword ptr [ecx+18h]
00401017      mov      [ebp+var_4], 0
0040101E      jmp      short loc_401029
00401020 loc_401020:
00401020      mov      edx, [ebp+var_4]
00401023      add      edx, 1
00401026      mov      [ebp+var_4], edx
00401029 loc_401029:
00401029      cmp      [ebp+var_4], 5
0040102D      jge      short loc_40103D
0040102F      mov      eax, [ebp+var_4]
00401032      mov      ecx, [ebp+arg_0]
00401035      mov      edx, [ebp+var_4]
00401038      mov      [ecx+eax*4], edx 2
0040103B      jmp      short loc_401020
0040103D loc_40103D:
0040103D      mov      esp, ebp
0040103F      pop      ebp
00401040      retn
```

We can tell that offset 0x18 is a double because it is used as part of a floating-point instruction at **1**. 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 **2**. 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.

**Example 7-29** 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 **1** creates 10 nodes and fills them with data. The second loop at **2** iterates over all the records and prints their contents.

Example 7-29. C code for a linked list traversal

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

typedef struct node pnode;

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

    head = NULL;

    for(i=1;i<=10;i++) 1
    {
        curr = (pnode *)malloc(sizeof(pnode));
        curr->x = i;
        curr->next = head;
        head = curr;
    }

    curr = head;

    while(curr) 2
    {
        printf("%d\n", curr->x);
        curr = curr->next ;
    }
}
```

The best way to understand the disassembly is to identify the two code constructs

within the `main` method. And that is, of course, the crux of this chapter: Your ability to recognize these constructs makes the analysis easier.

In [Example 7-30](#), we identify the `for` loop first. `var_C` corresponds to `i`, which is the counter for the loop. `var_8` corresponds to the `head` variable, and `var_4` is the `curr` variable. `var_4` is a pointer to a struct with two variables that are assigned values (shown at [1](#) and [2](#)).

The `while` loop ([3](#) through [5](#)) executes the iteration through the linked list. Within the loop, `var_4` is set to the next record in the list at [4](#).

Example 7-30. Assembly code for the linked list traversal example in [Example 7-29](#)

```

0040106A      mov     [ebp+var_8], 0
00401071      mov     [ebp+var_C], 1
00401078
00401078 loc_401078:
00401078      cmp     [ebp+var_C], 0Ah
0040107C      jg      short loc_4010AB
0040107E      mov     [esp+18h+var_18], 8
00401085      call    malloc
0040108A      mov     [ebp+var_4], eax
0040108D      mov     edx, [ebp+var_4]
00401090      mov     eax, [ebp+var_C]
00401093      mov     [edx], eax 1
00401095      mov     edx, [ebp+var_4]
00401098      mov     eax, [ebp+var_8]
0040109B      mov     [edx+4], eax 2
0040109E      mov     eax, [ebp+var_4]
004010A1      mov     [ebp+var_8], eax
004010A4      lea     eax, [ebp+var_C]
004010A7      inc     dword ptr [eax]
004010A9      jmp     short loc_401078
004010AB loc_4010AB:
004010AB      mov     eax, [ebp+var_8]
004010AE      mov     [ebp+var_4], eax
004010B1
004010B1 loc_4010B1:
004010B1      cmp     [ebp+var_4], 0 3
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 4
004010D5      jmp     short loc_4010B1 5

```

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 `4`, `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

Q: 1. What is the major code construct found in the only subroutine called by `main`?

Q: 2. What is the subroutine located at 0x40105F?

Q: 3. What is the purpose of this program?

## Lab 6-2

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

### Questions

Q: 1. What operation does the first subroutine called by `main` perform?

Q: 2. What is the subroutine located at 0x40117F?

Q: 3. What does the second subroutine called by `main` do?

Q: 4. What type of code construct is used in this subroutine?

Q: 5. Are there any network-based indicators for this program?

Q: 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

Q: 1. Compare the calls in `main` to **Lab 6-2 Solutions**'s `main` method. What is the new function called from `main`?

Q: 2. What parameters does this new function take?

Q: 3. What major code construct does this function contain?

Q: 4. What can this function do?

Q: 5. Are there any host-based indicators for this malware?

Q: 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

Q: 1. What is the difference between the calls made from the `main` method in **Lab 6-3 Solutions** and **Lab 6-4 Solutions**?

Q: 2. What new code construct has been added to `main`?

Q: 3. What is the difference between this lab's `parse HTML` function and those of the previous labs?

Q: 4. How long will this program run? (Assume that it is connected to the Internet.)

Q: 5. Are there any new network-based indicators for this malware?

Q: 6. What is the purpose of this malware?

# Chapter 8. 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 8-1** lists some of the most common Windows API types (there are many more). Each type's prefix follows it in parentheses.

Table 8-1. 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 <code>HModule</code> , <code>HInstance</code> , and <code>HKey</code> .
Long Pointer (LP)	A pointer to another type. For example, <code>LPByte</code> is a pointer to a byte, and <code>LPCSTR</code> is a pointer to a character string. Strings are usually prefixed by <code>LP</code> because they are actually pointers. Occasionally, you will see <code>Pointer (P)...</code> prefixing another type instead of <code>LP</code> ; in 32-bit systems, this is the same as <code>LP</code> . The difference was meaningful in 16-bit systems.
Callback	Represents a function that will be called by the Windows API. For example, the <code>InternetSetStatusCallback</code> 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 8-1**, 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 8-1**) 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.

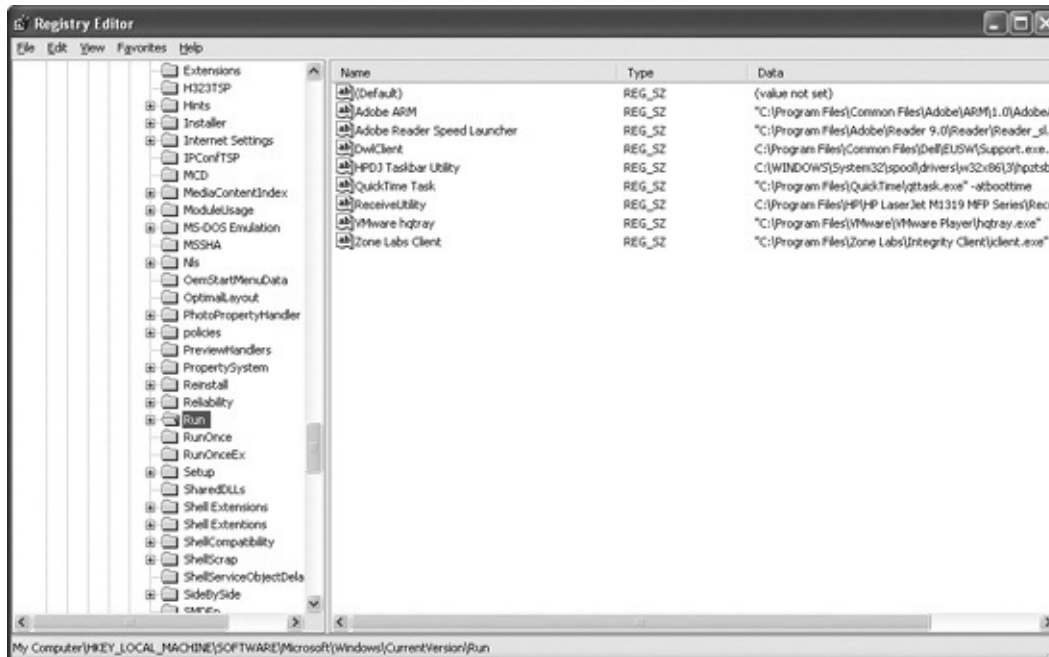


Figure 8-1. The Regedit tool

## 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

**Example 8-1** 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*, *RegSetValueEx*, and so on, remember to drop the trailing *W* or *A* character.

**Example 8-1.** Code that modifies registry settings

```
0040286F  push    2                ; samDesired
00402871  push    eax              ; ulOptions
00402872  push    offset SubKey    ;
"Software\\Microsoft\\Windows\\CurrentVersion\\Run"
00402877  push    HKEY_LOCAL_MACHINE ; hKey
0040287C  1call   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  2call   ds:lstrlenW
0040288F  lea     edx, [eax+eax+2]
00402893  3push    edx              ; cbData
00402894  mov     edx, [esp+428h+hKey]
00402898  4lea     eax, [esp+428h+Data]
0040289C  push    eax              ; lpData
0040289D  push    1                ; dwType
0040289F  push    0                ; Reserved
004028A1  5lea     ecx, [esp+434h+ValueName]
004028A8  push    ecx              ; lpValueName
004028A9  push    edx              ; hKey
004028AA  call    ds:RegSetValueExW
```

**Example 8-1** 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 **1** with the parameters needed to open a handle to the registry key HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run. The value name at **5** and data at **4** are stored on the stack as parameters to this function, and are shown here as having been labeled by IDA Pro. The call to strlenW at **2** is needed in order to get the size of the data, which is given as a parameter to the RegSetValueEx function at **3**.

## 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.

**Example 8-2** shows an example of a .reg file.

### Example 8-2. Sample .reg file

Windows Registry Editor Version 5.00

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

The first line in **Example 8-2** 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 15](#). 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 8-2](#).

Table 8-2. Berkeley Compatible Sockets Networking Functions

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

**Example 8-3** 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.

**Example 8-3.** A simplified program with a server socket

```
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
```

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.

DLLs are also a useful code-reuse mechanism. For example, large software companies will create DLLs with some functionality that is common to many of their applications. Then, when they distribute the applications, they distribute the main .exe and any DLLs that application uses. This allows them to maintain a single library of common code and distribute it only when needed.

## How Malware Authors Use DLLs

Malware writers use DLLs in three ways:

### To store malicious code

- Sometimes, malware authors find it more advantageous to store malicious code in a DLL, rather than in an .exe file. Some malware attaches to other processes,

but each process can contain only one .exe file. Malware sometimes uses DLLs to load itself into another process.

### **By using Windows DLLs**

- Nearly all malware uses the basic Windows DLLs found on every system. The Windows DLLs contain the functionality needed to interact with the OS. The way that a malicious program uses the Windows DLLs often offers tremendous insight to the malware analyst. The imports that you learned about in [Chapter 2](#) and the functions covered throughout this chapter are all imported from the Windows DLLs. Throughout the balance of this chapter, we will continue to cover functions from specific DLLs and describe how malware uses them.

### **By using third-party DLLs**

- Malware can also use third-party DLLs to interact with other programs. When you see malware that imports functions from a third-party DLL, you can infer that it is interacting with that program to accomplish its goals. For example, it might use the Mozilla Firefox DLL to connect back to a server, rather than connecting directly through the Windows API. Malware might also be distributed with a customized DLL to use functionality from a library not already installed on the victim's machine; for example, to use encryption functionality that is distributed as a DLL.

## **Basic DLL Structure**

Under the hood, DLL files look almost exactly like .exe files. DLLs use the PE file format, and only a single flag indicates that the file is a DLL and not an .exe. DLLs often have more exports and generally fewer imports. Other than that, there's no real difference between a DLL and an .exe.

The main DLL function is `DLLMain`. It has no label and is not an export in the DLL, but it is specified in the PE header as the file's entry point. The function is called to notify the DLL whenever a process loads or unloads the library, creates a new thread, or finishes an existing thread. This notification allows the DLL to manage any per-process or per-thread resources.

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 host-based firewalls and other security mechanisms. Or it could create an instance of Internet Explorer and then use that program to access malicious content.

Malware commonly uses `CreateProcess` to create a simple remote shell with just a single function call. One of the parameters to the `CreateProcess` function, the `STARTUPINFO` struct, includes a handle to the standard input, standard output, and standard error streams for a process. A malicious program could set these values to a socket, so that when the program writes to standard output, it is really writing to the socket, thereby allowing an attacker to execute a shell remotely without running anything other than the call to `CreateProcess`.

**Example 8-4** shows how `CreateProcess` could be used to create a simple remote shell. Prior to this snippet, code would have opened a socket to a remote location. The handle to the socket is stored on the stack and entered into the `STARTUPINFO` structure. Then `CreateProcess` is called, and all input and output for the process is routed through the socket.

Example 8-4. Sample code using the `CreateProcess` call

```
004010DA  mov     eax, dword ptr [esp+58h+SocketHandle]
004010DE  lea     edx, [esp+58h+StartupInfo]
004010E2  push    ecx                ; lpProcessInformation
004010E3  push    edx                ; lpStartupInfo
004010E4  1mov    [esp+60h+StartupInfo.hStdError], eax
004010E8  2mov    [esp+60h+StartupInfo.hStdOutput], eax
004010EC  3mov    [esp+60h+StartupInfo.hStdInput], eax
004010F0  4mov    eax, dword_403098
004010F5  push    0                  ; lpCurrentDirectory
004010F7  push    0                  ; lpEnvironment
004010F9  push    0                  ; dwCreationFlags
004010FB  mov     dword ptr [esp+6Ch+CommandLine], eax
004010FF  push    1                  ; bInheritHandles
00401101  push    0                  ; lpThreadAttributes
00401103  lea     eax, [esp+74h+CommandLine]
00401107  push    0                  ; lpProcessAttributes
00401109  5push    eax                ; lpCommandLine
0040110A  push    0                  ; lpApplicationName
0040110C  mov     [esp+80h+StartupInfo.dwFlags], 101h
00401114  6call    ds:CreateProcessA
```



In the first line of code, the stack variable `SocketHandle` is placed into `EAX`. (The socket handle is initialized outside this function.) The `lpStartupInfo` structure for the process stores the standard output **1**, standard input **2**, and standard error **3** that will be used for the new process. The socket is placed into the `lpStartupInfo` structure for all three values (**1**, **2**, **3**). The access to `dword_403098` at **4** contains the command line of the program to be executed, which is eventually pushed on the stack as a parameter **5**. The call to `CreateProcess` at **6** has 10 parameters, but all except `lpCommandLine`, `lpProcessInformation`, and `lpStartupInfo` are either 0 or 1. (Some represent NULL values and others represent flags, but none are interesting for malware analysis.)

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 [Example 8-4](#)). 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 2](#), 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 2](#)) 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.

**Example 8-5** shows an example of accessing a local variable and pushing it on the stack.

Example 8-5. Accessing a local variable and pushing it on the stack

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

In **Example 8-5**, the code at **1** 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.

## Creating a Thread

The `CreateThread` function is used to create new threads. The function's caller specifies a start address, which is often called the `start` function. Execution begins at the start address and continues until the function returns, although the function does not need to return, and the thread can run until the process ends. When analyzing code that calls `CreateThread`, you will need to analyze the `start` function in addition to analyzing the rest of the code in the function that calls `CreateThread`.

The caller of `CreateThread` can specify the function where the thread starts and a single parameter to be passed to the `start` function. The parameter can be any value, depending on the function where the thread will start.

Malware can use `CreateThread` in multiple ways, such as the following:

- Malware can use `CreateThread` to load a new malicious library into a process, with `CreateThread` called and the address of `LoadLibrary` specified as the start address. (The argument passed to `CreateThread` is the name of the library to be loaded. The new DLL is loaded into memory in the process, and `DllMain` is called.)
- Malware can create two new threads for input and output: one to listen on a socket or pipe and then output that to standard input of a process, and the other to read from standard output and send that to a socket or pipe. The malware's goal is to send all information to a single socket or pipe in order to communicate seamlessly with the running application.

**Example 8-6** shows how to recognize the second technique by identifying two `CreateThread` calls near each other. (Only the system calls for `ThreadFunction1` and `ThreadFunction2` are shown.) This code calls `CreateThread` twice. The arguments are `lpStartAddress` values, which tell us where to look for the code that will run when these threads start.

#### Example 8-6. Main function of thread example

```

004016EE lea     eax, [ebp+ThreadId]
004016F4 push    eax                ; lpThreadId
004016F5 push    0                  ; dwCreationFlags
004016F7 push    0                  ; lpParameter
004016F9 push    1offset ThreadFunction1 ; lpStartAddress
004016FE push    0                  ; dwStackSize
00401700 lea     ecx, [ebp+ThreadAttributes]
00401706 push    ecx                ; lpThreadAttributes
00401707 call   2ds:CreateThread
0040170D mov     [ebp+var_59C], eax
00401713 lea     edx, [ebp+ThreadId]
00401719 push    edx                ; lpThreadId
0040171A push    0                  ; dwCreationFlags
0040171C push    0                  ; lpParameter
0040171E push    3offset ThreadFunction2 ; lpStartAddress
00401723 push    0                  ; dwStackSize
00401725 lea     eax, [ebp+ThreadAttributes]
0040172B push    eax                ; lpThreadAttributes
0040172C call   4ds:CreateThread

```

In **Example 8-6**, we have labeled the start function `ThreadFunction1` **1** for the first call to `CreateThread` **2** and `ThreadFunction2` **3** for the second call **4**. To determine the purpose of these two threads, we first navigate to `ThreadFunction1`. As shown in **Example 8-7**, 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.

#### Example 8-7. ThreadFunction1 of thread example

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

As shown in [Example 8-8](#), 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.

#### Example 8-8. ThreadFunction2 of thread example

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

#### 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 [Example 8-9](#).

Example 8-9. Using a mutex to ensure that only one copy of malware is running on a system

```
00401000  push  offset Name      ; "HGL345"
00401005  push  0                 ; bInheritHandle
00401007  push  1F0001h           ; dwDesiredAccess
0040100C  1call ds:__imp__OpenMutexW@12 ; OpenMutexW(x,x,x)
00401012  2test  eax, eax
00401014  3jz    short loc_40101E
00401016  push  0                 ; int
00401018  4call ds:__imp__exit
0040101E  push  offset Name      ; "HGL345"
00401023  push  0                 ; bInitialOwner
00401025  push  0                 ; lpMutexAttributes
00401027  5call ds:__imp__CreateMutexW@12 ; CreateMutexW(x,x,x)
```

The code in [Example 8-9](#) 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 **1**. If the return value is `NULL` at **2**, it jumps (at **3**) over the `exit` call and continues to execute. If the return value is not `NULL`, it calls `exit` at **4**, and the process will exit. If the code continues to execute, the mutex is created at **5** 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.

Services can be installed and manipulated via a few Windows API functions, which are prime targets for malware. There are several key functions to look for:

- **OpenSCManager**. Returns a handle to the service control manager, which is used for all subsequent service-related function calls. All code that will interact with services will call this function.
- **CreateService**. Adds a new service to the service control manager, and allows the caller to specify whether the service will start automatically at boot time or must be started manually.
- **StartService**. Starts a service, and is used only if the service is set to be started manually.

The Windows OS supports several different service types, which execute in unique ways. The one most commonly used by malware is the `WIN32_SHARE_PROCESS` type, which stores the code for the service in a DLL, and combines several different services in a single, shared process. In Task Manager, you can find several instances of a process called `svchost.exe`, which are running `WIN32_SHARE_PROCESS`-type services.

The `WIN32_OWN_PROCESS` type is also used because it stores the code in an `.exe` file and runs as an independent process.

The final common service type is `KERNEL_DRIVER`, which is used for loading code into the kernel. (We discuss malware running in the kernel later in this chapter and extensively in [Chapter 11](#).)

The information about services on a local system is stored in the registry. Each

service has a subkey under HKLM\SYSTEM\CurrentControlSet\Services. For example, **Figure 8-2** shows the registry entries for HKLM\SYSTEM\CurrentControlSet\Services\VMware NAT Service.

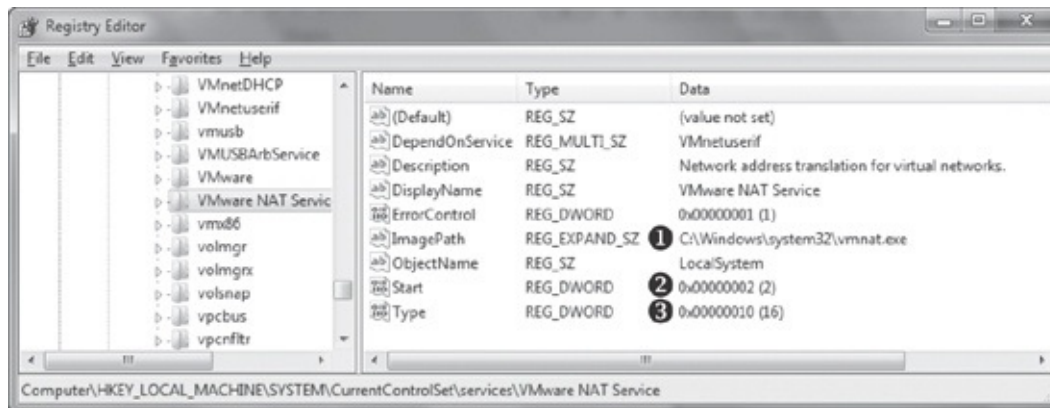


Figure 8-2. Registry entry for VMware NAT service

The code for the VMware NAT service is stored at C:\Windows\system32\vmnat.exe **1**. The type value of 0x10 **3** corresponds to WIN32\_OWN\_PROCESS, and the start value of 0x02 **2** corresponds to AUTO\_START.

The SC program is a command-line tool included with Windows that you can use to investigate and manipulate services. It includes commands for adding, deleting, starting, stopping, and querying services. For example, the qc command queries a service's configuration options by accessing the same information as the registry entry shown in **Figure 8-2** in a more readable way. **Example 8-10** shows the SC program in action.

**Example 8-10.** The query configuration information command of the SC program

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

```
SERVICE_NAME: VMware NAT Service  
        TYPE               : 10      1 WIN32_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
```

**Example 8-10** 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` **1**. 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 12](#).)

## 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 malware analyst can search for these calls to determine whether a program is using COM functionality. However, knowing that a piece of malware uses a COM object as a client does not provide much information, because COM objects are diverse and widespread. Once you determine that a program uses COM, you'll need to find a couple of identifiers of the object being used to continue analysis.

## CLSIDs, IIDs, and the Use of COM Objects



COM objects are accessed via their globally unique identifiers (GUIDs) known as class identifiers (CLSIDs) and interface identifiers (IIDs).

The `CoCreateInstance` function is used to get access to COM functionality. One common function used by malware is `Navigate`, which allows a program to launch Internet Explorer and access a web address. The `Navigate` function is part of the `IWebBrowser2` interface, which specifies a list of functions that must be implemented, but does not specify which program will provide that functionality. The program that provides the functionality is the COM class that implements the `IWebBrowser2` interface. In most cases, the `IWebBrowser2` interface is implemented by Internet Explorer. Interfaces are identified with a GUID called an IID, and classes are identified with a GUID called a CLSID.

Consider an example piece of malware that uses the `Navigate` function from the `IWebBrowser2` interface implemented by Internet Explorer. The malware first calls the `CoCreateInstance` function. The function accepts the CLSID and the IID of the object that the malware is requesting. The OS then searches for the class information, and loads the program that will perform the functionality, if it isn't already running. The `CoCreateInstance` class returns a pointer that points to a structure that contains function pointers. To use the functionality of the COM server, the malware will call a function whose pointer is stored in the structure returned from `CoCreateInstance`. **Example 8-11** shows how some code gets access to an `IWebBrowser2` object.

#### Example 8-11. Accessing a COM object with `CoCreateInstance`

```
00401024 lea     eax, [esp+18h+PointerToComObject]
00401028 push    eax                ; ppv
00401029 push    1offset IID_IWebBrowser2 ; riid
0040102E push    4                  ; dwClsContext
00401030 push    0                  ; pUnkOuter
00401032 push    2offset stru_40211C ; rclsid
00401037 call    CoCreateInstance
```

In order to understand the code, click the structures that store the IID and CLSID at **1** and **2**. The code specifies the IID `D30C1661-CDAF-11D0-8A3E-00C04FC9E26E`, which represents the `IWebBrowser2` interface, and the CLSID `0002DF01-0000-0000-C000-000000000046`, which represents Internet Explorer. IDA Pro can recognize and label the IID for `IWebBrowser2`, since it's commonly used. Software developers can create their own IIDs, so IDA Pro can't always label

the IID used by a program, and it is never able to label the CLSID, because disassembly doesn't contain the necessary information.

When a program calls `CoCreateInstance`, the OS uses information in the registry to determine which file contains the requested COM code. The `HKLM\SOFTWARE\Classes\CLSID\` and `HKCU\SOFTWARE\Classes\CLSID` registry keys store the information about which code to execute for the COM server. The value of `C:\Program Files\Internet Explorer\iexplore.exe`, stored in the `LocalServer32` subkey of the registry key `HKLM\SOFTWARE\Classes\CLSID\0002DF01-0000-0000-C000-000000000046`, identifies the executable that will be loaded when `CoCreateInstance` is called.

Once the structure is returned from the `CoCreateInstance` call, the COM client calls a function whose location is stored at an offset in the structure. **Example 8-12** shows the call. The reference to the COM object is stored on the stack, and then moved into `EAX`. Then the first value in the structure points to a table of function pointers. At an offset of `0x2C` in the table is the `Navigate` function that is called.

#### Example 8-12. Calling a COM function

```
0040105E  push    ecx
0040105F  push    ecx
00401060  push    ecx
00401061  mov     esi, eax
00401063  mov     eax, [esp+24h+PointerToComObject]
00401067  mov     edx, [eax]
00401069  mov     edx, [edx+12Ch]
0040106C  push    ecx
0040106D  push    esi
0040106E  push    eax
0040106F  call    edx
```

In order to identify what a malicious program is doing when it calls a COM function, malware analysts must determine which offset a function is stored at, which can be tricky. IDA Pro stores the offsets and structures for common interfaces, which can be explored via the structure subview. Press the **INSERT** key to add a structure, and then click **Add Standard Structure**. The name of the structure to add is `InterfaceNameVtbl`. In our `Navigate` example, we add the `IWebBrowser2Vtbl` structure. Once the structure is added, right-click the offset at **1** in the disassembly to change the label from `2Ch` to the function name `IWebBrowser2Vtbl.Navigate`. Now IDA Pro will add comments to the `call` instruction and the parameters being pushed onto the stack.

For functions not available in IDA Pro, one strategy for identifying the function called by a COM client is to check the header files for the interface specified in the call to `CoCreateInstance`. The header files are included with Microsoft Visual Studio and the platform SDK, and can also be found on the Internet. The functions are usually declared in the same order in the header file and in the function table. For example, the `Navigate` function is the twelfth function in the .h file, which corresponds to an offset of `0x2C`. The first function is at 0, and each function takes up 4 bytes.

In the previous example, Internet Explorer was loaded as its own process when `CoCreateInstance` was called, but this is not always the case. Some COM 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. **Example 8-13** shows disassembly for the first few lines of a function that has exception handling.

**Example 8-13.** Storing exception-handling information in `fs:0`

```
01006170  push  1offset loc_10061C0
01006175  mov    eax, large fs:0
0100617B  push  2eax
0100617C  mov    large fs:0, esp
```

At the beginning of the function, an exception-handling frame is put onto the stack at **1**. 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 **2**, 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 (**Chapter 9–Chapter 11**, **Chapter 16**, and **Chapter 17**).

## 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

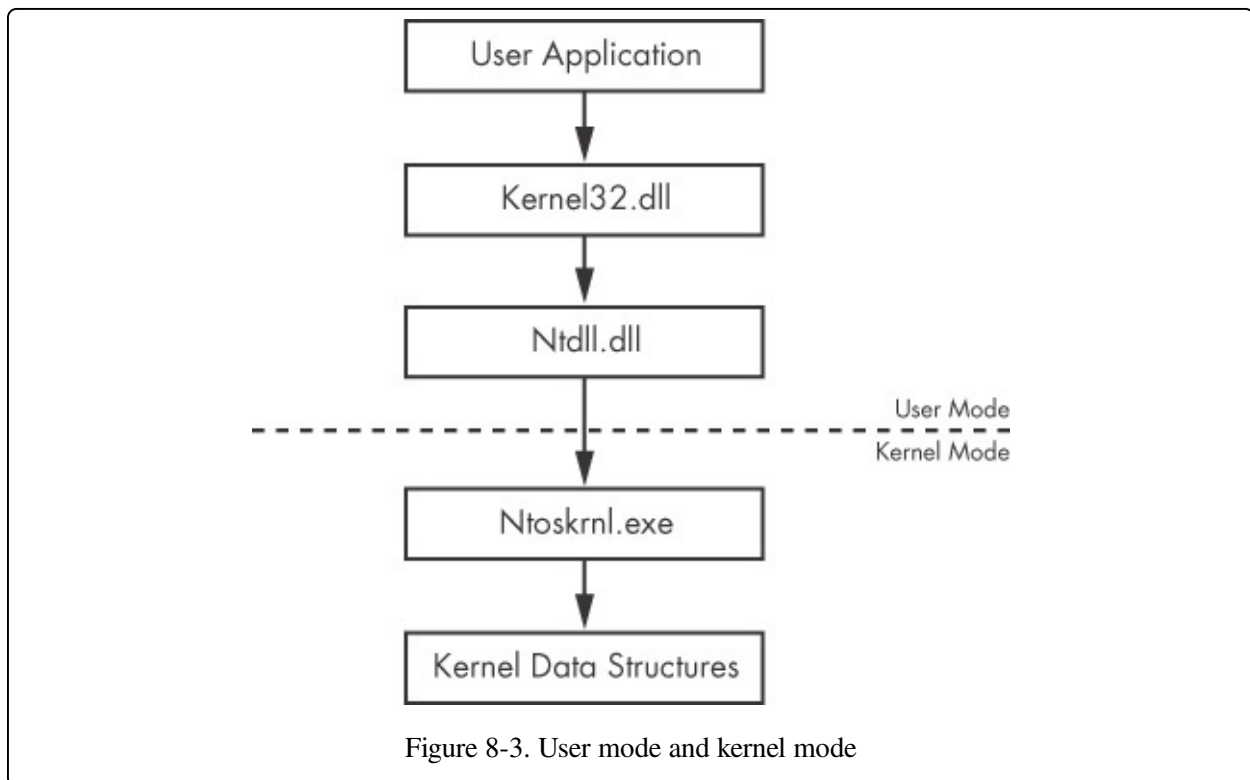
running in user space. Within kernel space, any distinction between processes running as a privileged or unprivileged user is removed. Additionally, the OS's auditing features don't apply to the kernel. For these reasons, nearly all rootkits utilize code running in the kernel.

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 11](#).)

# 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 8-3** illustrates how this works for most API calls.



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.

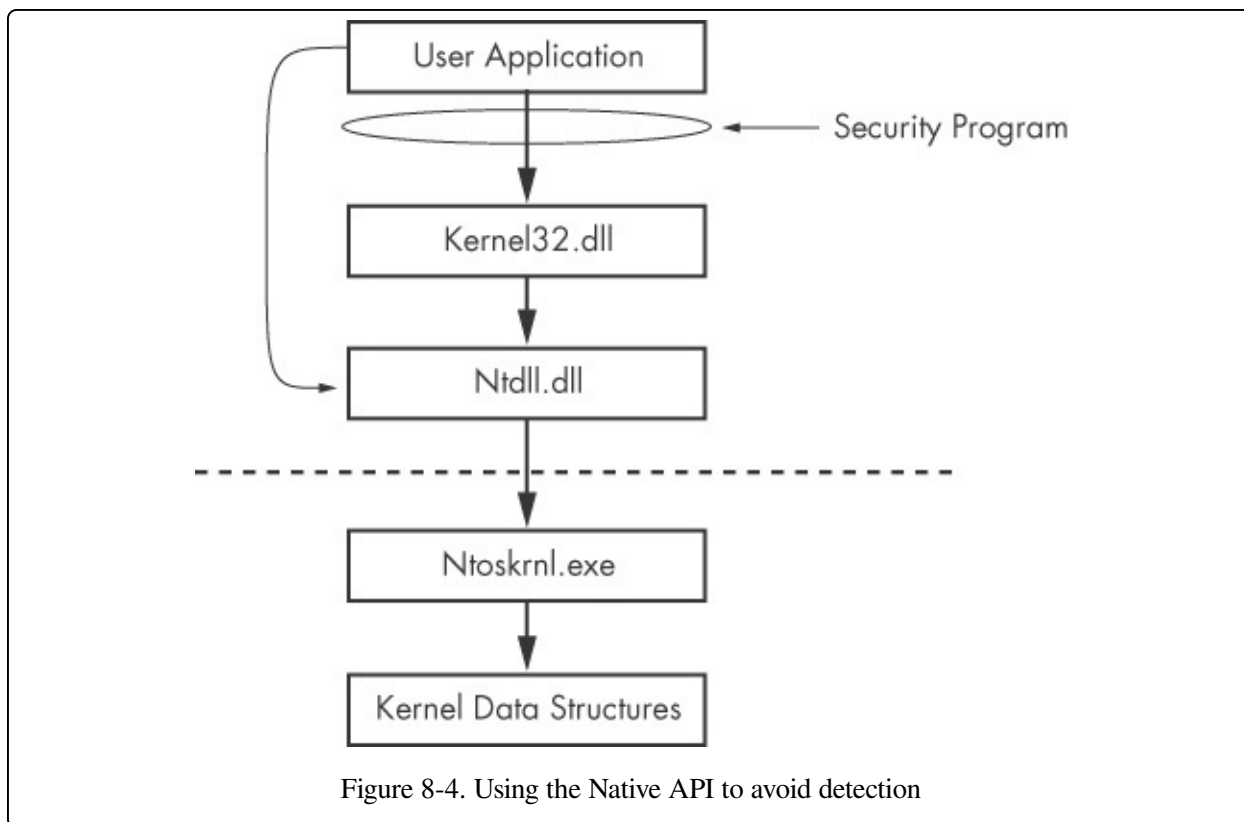
The ntdll functions use APIs and structures just like the ones used in the kernel. These functions make up the Native API. Programs are not supposed to call the Native API, but nothing in the OS prevents them from doing so. Although Microsoft does not provide thorough documentation on the Native API, there are websites and books that document these functions. The best reference is Windows NT/2000 Native API Reference by Gary Nebbett (Sams, 2000), although it is quite old. Online resources such as <http://undocumented.ntinternals.net/> can provide more recent information.

Calling the Native API directly is attractive for malware writers because it allows them to do things that might not otherwise be possible. There is a lot of functionality that is not exposed in the regular Windows API, but can be accomplished by calling the Native API directly.

Additionally, calling the Native API directly is sometimes stealthier. Many antivirus and host-protection products monitor the system calls made by a process. If the process calls the Native API function directly, it may be able to evade a poorly designed security product.

**Figure 8-4** shows a diagram of a system call with a poorly designed security program monitoring calls to kernel32.dll. In order to bypass the security program, some hypothetical malware uses the Native API. Instead of calling the Windows functions `ReadFile` and `WriteFile`, this malware calls the functions `NtReadFile` and `NtWriteFile`. These functions are in ntdll.dll and are not monitored by the security program. A well-designed security program will monitor calls at all levels, including the kernel, to ensure that this tactic doesn't work.





There are a series of Native API calls that can be used to get information about the system, processes, threads, handles, and other items. These include `NtQuerySystemInformation`, `NtQueryInformationProcess`, `NtQueryInformationThread`, `NtQueryInformationFile`, and `NtQueryInformationKey`. These calls provide much more detailed information than any available Win32 calls, and some of these functions allow you to set fine-grained attributes for files, processes, threads, and so on.

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.