by ordinal instead of name. When importing a function by ordinal, the name of the function never appears in the original executable, and it can be harder for an analyst to figure out which function is being used. When malware imports a function by ordinal, you can find out which function is being imported by looking up the ordinal value in the pane at ❹.

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

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

Table 1-1: Common DLLs

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

FUNCTION NAMING CONVENTIONS

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

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

Imported Functions

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

Exported Functions

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

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

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

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

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

Static Analysis in Practice

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

PotentialKeylogger.exe: An Unpacked Executable

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

PackedProgram.exe: A Dead End

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

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

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

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

The PE File Headers and Sections

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

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

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

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

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

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

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

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

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

Examining PE Files with PEview

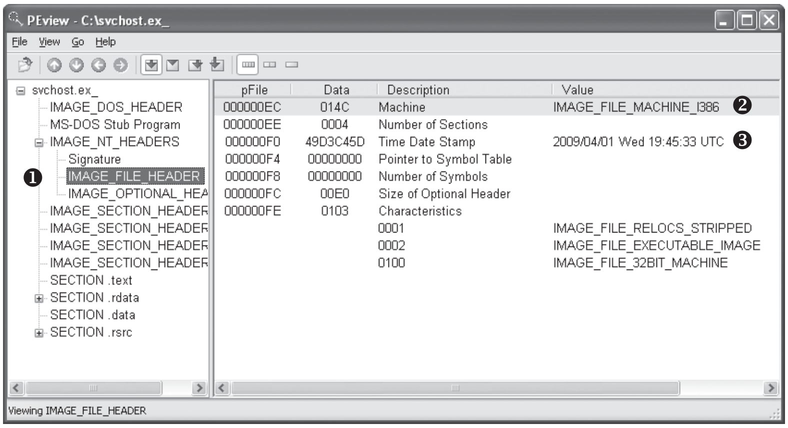
The PE file format stores interesting information within its header. We can use the PEview tool to browse through the information, as shown in Figure 1-7.

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

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

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

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



```
svchost.ex_
  IMAGE_DOS_HEADER
  MS-DOS Stub Program
  IMAGE_NT_HEADERS
    Signature
❶   IMAGE_FILE_HEADER
    IMAGE_OPTIONAL_HEA
  IMAGE_SECTION_HEADER
  IMAGE_SECTION_HEADER
  IMAGE_SECTION_HEADER
  IMAGE_SECTION_HEADER
  SECTION .text
  SECTION .rdata
  SECTION .data
  SECTION .rsrc
```

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

Viewing IMAGE_FILE_HEADER

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

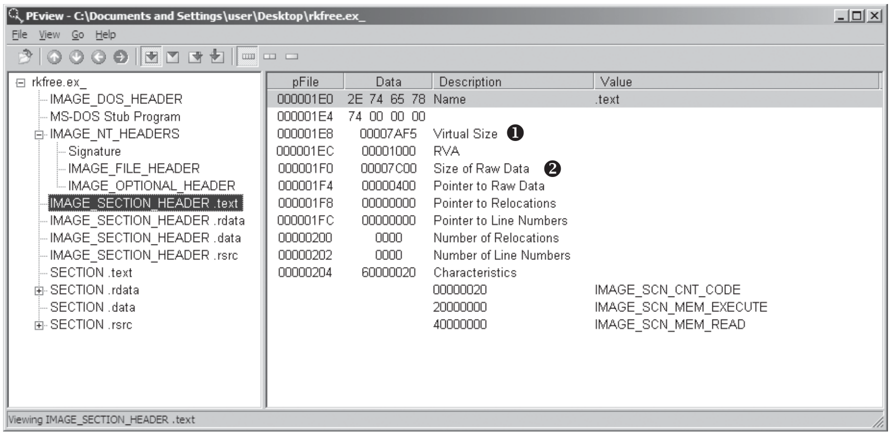
That said, the compile time is a bit problematic. All Delphi programs use a compile time of June 19, 1992. If you see that compile time, you're probably looking at a Delphi program, and you won't really know when it was compiled. In addition, a competent malware writer can easily fake the compile time.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

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

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

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

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



```
PEview - C:\Documents and Settings\user\Desktop\rkfree.ex_

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

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

Viewing IMAGE_SECTION_HEADER .text
```

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

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

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

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

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

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

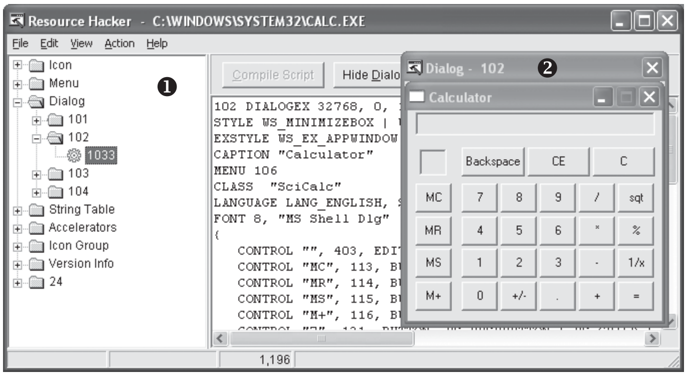
Name	Virtual size	Size of raw data
`.text`	A000	0000
`.data`	3000	0000
`.rdata`	4000	0000
`.rsrc`	19000	3400

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

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

Viewing the Resource Section with Resource Hacker

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



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

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

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

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

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

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

Using Other PE File Tools

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

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

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

PE Header Summary

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

Table 1-7: Information in the PE Header

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

Conclusion

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

LABS

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

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

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

Lab 1-1

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

Questions

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

Lab 1-2

Analyze the file *Lab01-02.exe*.