PC Magazine June 14, 1994 v13 n11 p337(6)

Dumping code and spying for Windows clues.

(PC Tech: Power Programming)

Author

Pietrek, Matt

Abstract

A guide to relatively simple methods of reverse- engineering Windows applications is presented. File- dumping utilities such as Microsoft's own EXEHDR or Borland's TDUMP dump information in a structured format to indicate what functions a program uses, but do not describe internal algorithms and data structures. 'Spy' programs found in many software development kits show the messages Windows sends and receives; some can see a program's calls to operating system API functions. Dumping out a file's contents shows what type it is and what it might be used for. Techniques for examining the output of such programs are presented. Message- spying tools are useful, but many programmers need more information; Periscope's WinScope and NuMega's BoundsChecker for Windows are among the programs that have raised spying to a new level by intercepting API calls. 'Probes' are put out at well- defined spots and take advantage of debug information when checking for bugs.

Full Text

In a perfect programming world, all of an operating system's useful interfaces and behaviors would be documented. On top of that, the operating system would be flexible and full- featured enough to support the demands of any reasonable program. In this perfect world, you would be able to look up anything of use in the documentation, allowing you to treat the operating system's internal components like a black box. You wouldn't need to understand the internal behaviors and data structures of such an operating system, because understanding and using the documented interfaces would be sufficient for writing your program, library, or device driver. The fact that words like undocumented and secrets appear in many book titles indicates that the current state of Windows' documentation is less than perfect.

In the absence of complete documentation, having the source code to the operating system can be a surrogate source of information. Although you would have to look at other people's code (shudder!), the answer to almost every operating- system question can be found with enough digging through the operating- system sources. In the Unix world, access to the operating- system sources is a fairly common thing. Unfortunately for us programmers who work with Microsoft Windows 3.1 and the Win32 operating systems (Windows NT, Chicago, and Win32s), operating- system source code isn't available.

Insufficient documentation and source code unavailability are not just a problem when you're working only with the operating system. An application you're developing may need to interact with another application that's exact behavior is unknown to you. A typical example of this is when a programmer spends a great deal of time trying to pin down what DDE messages Microsoft Excel sends and in what order. Another example from Windows 3.1 is the program that, when run, stops other programs from running. The cryptic "Insufficient memory to run this application" message from Windows isn't a great deal of help.

When you're faced with a lack of documentation and support resources, often the only remaining course of action is to take matters into your own hands and reverse- engineer the relevant code. To some people, the phrase reverse- engineer evokes images of immoral young hackers cracking the piracy- prevention code in a program so that they can sell copies of the program on the street corner for five dollars. Another common image in reverse engineering is the programmer who reverse- engineers a popular application program in order to sell a cloned version of the program without doing any original work. However, there are several different types of reverse engineering. Professional programmers who use these techniques rarely fit the above stereotypes. Rather, professionals use reverse engineering only as a last ditch effort to get past a particular problem when all other methods fail.

In this column, I'll discuss two relatively easy methods of reverse engineering used by Windows and Win32 programmers: file- dumping utilities and API- spying programs. In a future column, I'll describe the more complex process of reverse engineering with disassemblers. For each method, I'll describe the commonly available tools and give examples of how to use these tools.

REVERSE ENGINEERING OVERVIEW

The first step in reverse engineering is deciding which method to use. The easiest and most readily available method of learning the details of a piece of code is to use file- display programs, such as Borland's TDUMP or Microsoft's EXEHDR. These programs dump information in a structured format, which tells you, for example, the DLL functions a program uses.

These programs don't tell you about internal algorithms and data structures, though. Using file- dumping programs is relatively easy, but you may not be able to get all the information you need. To use an analogy, if you were trying to find out about a house and its occupants, file dumping is akin to conducting surveillance from across the street. You can see who enters and leaves the house, but what goes on inside, who knows?

The next level of snooping into the internals of a program is to use a spy program. Programs such as SPY, from the Microsoft Windows SDK, and WinSight, from Borland C++, show the Windows messages that a program sends and receives. Recently, programs such as Nu- Mega Technologies' Bounds- Checker for Windows and The Periscope Company's WinScope added the ability to see a program's calls to the operating system's API functions. You can even extend WinScope to spy on nonsystem DLLs. With all this information and a little bit of work, you can figure out how almost any piece of code is implemented. Returning to the above analogy, spy programs intercept the mail and listen in on phone conversations going into and out of the house.

Finally, when you need to know a program or DLL's internal algorithms or data structures, disassembly is the big gun. In the house- surveillance analogy, disassembly is like breaking down the front door and rifling through the house.

FILE- DUMPING TOOLS

The usual first step in reverse engineering a program is to dump out the file's contents. This is a quick and easy way to get a handle on what type of file you're dealing with and what the file might be used for. Figure 1 lists the capabilities of some well- known tools that dissect a file's contents. If you develop with Borland C++, try out TDUMP. EXE from the BIN directory. If you develop with the 16- bit Microsoft C/ C++, EXEHDR is your tool. If you use Microsoft Visual C/ C++ 32- Bit Edition, the DUMPBIN program in the BIN directory works with Portable Executables and COFF (Common Object File Format) .OBJs produced by Microsoft 32- bit compilers. (If you have Microsoft SDK for Win32, the DUMPBIN program is embedded inside LINK. EXE and can be invoked with LINK- dump <rest of command line>.) As Figure 1 shows, no one program does everything, so it's a good idea to have several of them on hand. A good combination that would cover all the bases is Borland's TDUMP and Microsoft's DUMPBIN.

Before continuing, a bit of explanation on the file formats in Figure 1 is in order. All .EXE programs that run under DOS, Windows, Windows NT, or OS/ 2 start with a 2- byte signature. Expressed in ASCII form, the signature is "MZ", the initials of Mark Zbikowski, an early Microsoft coder. The presence of the MZ signature means that the file contains a DOS .EXE program. When Microsoft designed the early versions of Windows and OS/ 2, it needed a different executable format but a format that wouldn't make DOS choke if it encountered a newer executable. Each of these extended executables has a small DOS program at the beginning that prints out a message saying this program requires Windows, OS/ 2, Windows NT, or so forth. Following the DOS stub program in these files is another signature word that identifies which operating system the program requires. OS/ 2 1. x and 16- bit Windows programs use the "NE" signature (for New Executable). Windows VxDs have an "LE" signature (Linear Executable), while 32- bit executables written for Win32 implementations have a "PE" (Portable Executable) signature. These Win32 files are also sometimes referred to as COFF files, because they're derived from the Unix System V COFF format.

Sometimes the most useful information you get from a file- dumping program is the names of the DLLs and the functions that a program or a DLL imports. Often just knowing that a program uses a certain function is enough to get you going when you're stuck. For example, Windows 3.0 had no documented way for a program to change the desktop wallpaper, yet the Control Panel application was able to change the wallpaper. The capability had to exist somewhere in Windows. By running TDUMP or EXEHDR on the Windows 3.0 Control Panel program, you would have seen that the program made use of the undocumented SetDeskWallPaper function. (In Windows 3.1, the documented SystemParametersInfo API function takes over this task.) Of course, not all exported function names are illuminating (GDI. EXE's Death and Resurrection functions come to mind), but more often than not they are helpful.

Finding the functions that a 16- bit NE program or a DLL uses is a two- step process. The NE file doesn't contain a simple list of all the functions imported from other DLLs. Instead, you first need to find the fix- up data for the executable's segments. (If you use EXEHDR, you'll need to use the /VERBOSE switch to get the fix- up information.) A TDUMP version of a typical sequence of fix- ups that occurs in the Windows 3.1 CALC. EXE is

...

```
PTR 0AD9h GDI. 91
PTR 0121h GDI. 93
PTR 00EAh USER. 89
PTR 0223h USER. 90
PTR 04ADh USER. 91
PTR 1DCAh USER. 92
```

...

The important information here is the module name and the import ordinal at the end of each line. This program is importing six functions, two from GDI and four from USER. Function names such as GDI. 91 aren't incredibly useful by themselves, so the second step of the process is to convert the module name (GDI) and ordinal (91) to a real function name. The translation between a function's name and its exported ordinal value is stored in the .EXE or .DLL file as well as in an import library. Since there's no easy way for Microsoft users to dump the contents of a 16- bit import library, I'll show how to get the function name from the DLL directly. Figure 2 shows a fragment of the Non- Resident Name Table section produced by running TDUMP on GDI. EXE. If you use EXEHDR, you'll find similar information in the section titled Exports:.

Look at the TDUMP output for GDI .EXE. Notice that the function GetTextExtent corresponds to GDI. 91. Putting two and two together, we can see that CALC. EXE calls GetTextExtent (GDI .91). The names of the other imported functions can be determined by repeating the above steps.

Finding the functions that a 32- bit PE file imports is much simpler. Running DUMPBIN on a Win32 program shows that a PE file contains a list of all the functions the program imports. The list of imported functions is sorted by module. The following is a portion of the imports section obtained by running DUMPBIN on Windows NT's USER32. DLL:

```
  9 BitBlt
 12 CopyEnhMetaFileW
 14 CopyMetaFileW
 15 CreateBitmap
 17 CreateBrushIndirect
 18 CreateCompatibleBitmap
 19 CreateCompatibleDC
```

The first number on each line is the hint ordinal. Win32 operating systems import functions by name, but the hint ordinal can speed up the process. The hint ordinal gives the loader a hint of where to start a binary search for the function name in the DLL that exports the functions.

Examining USER32. DLL's import table, we can see that USER32 calls GDI functions, such as BitBlt and CopyEnhMetaFileW. Note that when USER32 .DLL has a choice of calling either an ASCII or a Unicode version of the function, it calls the Unicode version (CopyEnhMetaFileW, CopyMetaFileW-- the W stands for wide character). This is consistent with Microsoft's claim that NT uses Unicode strings internally.

While it's very useful to know what API functions an .EXE or DLL uses, the flip side is equally important. File- dumping programs can show you what API functions a DLL exports for use by other programs and DLLs. The exported- function names are often a dead giveaway to what the DLL does and what its capabilities are. Sometimes a name by itself is enough information to enable you to guess what the parameters to an undocumented function are. Other times you'll need to use disassembly techniques along with the exported- function names to figure out how to call an undocumented DLL function.

Figure 3 shows the TDUMP output of the exports from a 16- bit NE file, SPELL .DLL. This DLL comes with Microsoft Word for Windows 2.0, but its API is not documented. Some of these function names give clues as to what they do and how you might call them. SpellVer probably doesn't take any parameters and probably returns a version number in the AX or DX: AX registers. It's very easy to write a small test program that tests this theory. You'd create a prototype for the function-- DWORD SpellVer(void);-- call LoadLibrary and GetProcAddress to get a pointer to the function, and then call SpellVer. Afterwards, you'd look at the DX and AX registers to see if they were set with values that resemble version numbers. Well, wouldn't you know, I wrote a small program to test this and discovered that the function always returns 0. Resorting to disassembly, I discovered that the SpellVer function actually takes three far pointers to WORDs (LPWORD). The lesson here is that file dumping is the easiest form of reverse engineering, but it doesn't always give you adequate information.

On the Win32 side of things, Figure 4 contains a portion of the output from running DUMPBIN on the Windows NT KERNEL32. DLL file. The figure shows two things of importance. First, several of the exported API functions have two variations (for example, AddAtomA and AddAtomW). AddAtomA is the version of AddAtom that uses ASCII strings, while AddAtomW is the equivalent function that uses Unicode strings. Second, the number at the end of each line is the relative virtual address (RVA) of the function in the module. A disassembler can use this address to match a symbolic name to a specific section of code.

Moving to a somewhat darker subject, we see in Figure 5 a selection of the DUMPBIN display of NTDLL .DLL's exported functions. Many of these undocumented functions uncovered by Microsoft's own DUMPBIN do the real work of creating processes, managing memory, and so on. You might be thinking, "This is nice, but I probably can't use NTDLL .DLL myself." Wrong! If you run DUMPBIN on a Windows NT program, such as WPERF. EXE, you'll see that they call undocumented NTDLL .DLL functions, such as NtQuerySystemInformation. So much for Microsoft's claim that undocumented API functions aren't of any use to application programs!

Besides the functions that an .EXE or a DLL imports or exports, you can often gain additional insight into a file by examining some of its text strings. One of the most useful text strings is the description field. The linker puts whatever you specify on the DESCRIPTION line of the .DEF file into the executable's description field. In 16- bit NE files, the description string is the first entry in the Non- Resident Name Table. Figure 6 shows some typical description strings from the files in the Windows 3.1 WINDOWS directory.

In 32- bit PE files, the Microsoft linker puts the description string somewhere in the .rdata section. Unfortunately, there doesn't appear to be any consistency in its placement. If you want to see these strings, your best bet is to do a raw hex dump of the .rdata section and look for an embedded ASCII string. Also, since the Microsoft Win32 tools generally don't need a .DEF file, you'll find many files that don't have a description string.

Another interesting place to look for useful strings after dumping an .EXE or a DLL is in the resource section. In both Win16 and Win32 programming, you can specify resources by ordinal number or by name. Sometimes dialogs have interesting names or hidden controls that are outside the dialog rectangle. String- table resources often contain goodies that you may ordinarily never see. For instance, in the TAIPEI. EXE game from Microsoft, the program rewards you with a proverb if you win a game. If you want to see all the possible proverbs, you can either master the game or cheat like I did and just dump out the string tables. Programs such as Borland's Resource Workshop let you view and edit the resources in any file interactively.

The jackpot of reverse engineering with file dumpers is when you encounter a file that still contains debugging information. From the debug information in an executable, you can learn the names of all the executable's functions and variables. Along with the functions' and variables' names, the debug information contains the addresses of these symbols. The debug information also holds the types of the variables, the argument lists for the functions, and the layout of the structures and classes. In short, the debugging information contains almost everything about your program that you wouldn't want a competitor to know. I once shocked a programmer by telling him about a GP fault in his code and on what line it occurred. No, I didn't have the source code. The debug information alone was enough for me to pinpoint the problem. Your competitors may not be so nice! This is why it's important to check that you don't ship debugging information with your product. Many companies, including Microsoft, Borland, and Delrina, have been guilty of this in the past. Run SOUNDREC. EXE from Windows 3.1 through TDUMP or Microsoft's CVDUMP if you don't believe this.

While I've been focusing on file dumping of .EXEs and DLLs, you shouldn't overlook the wealth of information that can be found in other related files. In particular, .OBJ and .LIB files contain quite a bit of information about a given source module (or collection of modules). Borland's TDUMP takes apart Intel OMF (object module file) .OBJ files to show you public and external symbols, segment names, and so on. Symantec C++ includes the OBJ2ASM utility, which symbolically disassembles the code contained within an Intel OMF .OBJ file. The Win32 DUMPBIN program performs general purpose .OBJ file dumping as well as disassembly of COFF .OBJ files.

SPYING TOOLS

File dumping can be interesting and informative, but you often need to know more about the code in question. Tools that let you spy on a program's interactions with the operating system can be more than up to the task. The most familiar Windows spying tools are the message- spying programs, SPY, from Microsoft, and WINSIGHT, from Borland. Message- spying programs show the messages that a window receives and how the program responds to these messages.

While message- spying tools are useful, programmers often need more information to get to the root of what they're trying to figure out. Recently, programs such as Nu- Mega's Bounds- Checker for Windows and Periscope's WinScope have raised spy tools to a new level. Besides spying on messages, these programs intercept API calls that a program or its DLLs make. Additionally, these spy programs monitor and log window hooks, TOOLHELP notifications, and other callbacks.

These programs put "probes" at all the well- defined places where controls enter or exit the program's code-- such as window procedures and API calls. The information that passes across these boundaries is located in a predictable manner. For example, all window procedures are called with a consistent set of parameters on the stack (the HWND is at [BP+ 0E], the MSG number is at [BP+ 0C], and so on). Spy programs take advantage of this knowledge in order to save, analyze, and display the information.

The best spying tools are those that don't require any modification of the code you're spying on. To insert their probes, programs such as Bounds- Checker for Windows and WinScope rely solely on the information in the executable file and the calls it makes.

As a result, these programs can spy on almost any .EXE or DLL, even those that you can't relink or modify in some way. A second class of spying tools requires you to relink the code you wish to spy on.

These tools work by fooling the linker into resolving the program's API function calls to point to the tools' own code rather than to the operating- system DLLs. A closely related class of tools modifies the executable file after they've been linked. The net effect is the same. The spy program redirects function calls to the tool's own code, which logs the call before passing control to the operating system.

For 16- bit Windows applications, there are a variety of spying tools available. The primary purpose of Bounds- Checker for Windows (BCHKW) is to find bugs by intercepting all the Windows API function calls and certain C library calls that a program makes, and then validate the parameters. To give a clearer picture of the sequence of events that lead up to a bug, Bounds- Checker also watches window and dialog messages, hook callbacks, TOOLHELP notifications, and other assorted callbacks. If you save the trace information to a disk, the accompanying TVIEW program shows an expandable/ collapsible hierarchical view of your program's events. TVIEW can filter out repetitive sequences of function calls and messages that you're probably not interested in. A typical sequence of this sort would be

GetMessage

TranslateMessage

DispatchMessage

Window Message

DefWindowProc

Although Bounds- Checker takes advantage of debug information when it's checking for bugs, this information isn't necessary for its spying capabilities. As a result, you can run Bounds Checker for Windows with any Windows program.

Unlike Bounds- Checker for Windows, which concentrates on one program at a time, Periscope's WinScope is a systemwide spy tool. You'll see all the API function calls, hooks, and messages that occur anywhere in the system.

Sometimes this is very useful, while at other times it leads to information overflow. Fortunately, WinScope provides a very high level of customization to let you select what you want to spy on. You can enable or disable spying on individual windows, API functions, or groups of API functions. The same applies to Windows messages and hooks. Like Bounds- Checker for Windows, WinScope can save a copy of the memory that a function's far- pointer parameters point to. This enables you to see the actual strings and data structures that were passed to CreateWindow, to GetPrivateProfileString, and so on. WinScope uses the information in an NE file to hook calls to the Windows API, so you don't need to relink the code you want to spy on.

If you're willing to sacrifice usability and some features for price, you might consider Microsoft's API parameter profilers. Although both 16- and 32- bit versions of these spying tools appear on the Windows NT SDK, very few people know of their existence. The Microsoft profiler is crude in its implementation and requires you to modify the .EXEs or R. DLL, ZERNEL32. DLL, and ZSER32. DLL). Each DLL has the same base filename as an operating- system DLL but with the first letter changed to Z. These special DLLs have a small stub for each API function exported from the DLL they replace. You connect your program or DLL to these special DLLs with the APFCNVRT program (or APF32CVT for you Win32 users). APFCNVRT and APF32CVT modify the program's executable file so that that program imports its functions from the parameter- profiling DLLs rather than the normal operating- system DLLs. When you run the modified program, all calls to the affected DLLs go through the parameter- spying DLLs before they're passed on to the operating system.

Another Win32 spy program is Nu- Mega's Bounds- Checker32 for Windows NT (BCHK32NT), which spies on calls to the Win32 API, window messages, and hooks. For the most part, Bounds- Checker32 for Windows NT is similar to Bounds Checker for Windows. However, it includes features not present in its Windows 3.1 sibling. When a Win32 function fails, the function usually stores an error code with SetLastError indicating why the call failed. Bounds- Checker32 for Windows NT knows when an API function fails and records the error code. Also, since Win32 supports threads, Bounds- Checker32 for Windows NT saves the thread ID for each function call and window message. Its TVIEW program uses the thread information to provide additional filtering options, such as showing only the events for a specified thread.

When evaluating a spying tool, make sure you know what parts of the system the toolkit allows you to watch. In API spying, the program events that the spying tool can't show you are often the difference between figuring out what's going on and scratching your head in confusion.

As a rule of thumb, the more data points (API function calls, window messages, hook callbacks, and TOOLHELP notifications) you record, the better. For instance, the Microsoft parameter profiler is useless if you're trying to figure out how a program uses TOOLHELP to perform some action. It doesn't have a ZOOLHELP. DLL to watch calls to TOOLHELP.

A SPYING ADVENTURE

Let's solve a real world problem to see these tools in action. A common question I hear relates to the Windows CLOCK. EXE program: "How do I make my program switch between having a title bar and not having a title bar, as CLOCK. EXE does?" You can find the answer by using a spy program to see what function calls CLOCK .EXE makes. I'm going to use the 32- bit CLOCK. EXE program from NT. I could just as easily have used the 16- bit CLOCK .EXE from Windows 3.1, though.

For my tool, I'll use Bounds- Checker32 for Windows NT, although any of the spying tools that spy on API functions I've mentioned would do. The first step is to run the program and collect the trace information. Run Bounds- Checker32 for Windows NT, select CLOCK. EXE from the File | Load dialog, and choose Run. After CLOCK starts up, click on Settings | No Title. Then shut down CLOCK. You can then examine the trace output and find the spot where the program responded to the No Title command. Figure 7 shows a text- file version of the relevant parts of the trace.

You might be wondering, "How did he know where to look in the trace information?" Whenever you select something from a menu, Windows delivers a WM_ COMMAND message to your program. Therefore, the first step in finding this sequence of events is to search for the string "WM_ COMMAND". If you followed the above steps, there should only be one WM_ COMMAND message in the entire event log.

For the sake of completeness, however, let's verify that the WM_ COMMAND message in Figure 7 is the correct one. In a WM_ COMMAND message, the WPARAM parameter holds the ID of the selected menu item. In the figure, WPARAM is 6. If you dump the resources in CLOCK. EXE with Resource Workshop or a similar program, you'll see that the No Title menu item has an ID of 6. We're now sure that we're looking at the right section of the event log.

After receiving the WM_ COMMAND message that tells CLOCK to turn off its title bar, CLOCK calls GetWindowLong, passing the GWL_ STYLE parameter. The next line of the figure shows that GetWindowLong returned a DWORD of 0x14CF0000. This value represents the WS_ xxx style bits passed to CreateWindow. You can decode these bits yourself by looking in WINDOWS. H:

```
#define WS_ VISIBLE         0x10000000L
#define WS_ CLIPSIBLINGS    0x04000000L
#define WS_ BORDER          0x00800000L
#define WS_ DLGFRAME        0x00400000L
#define WS_ SYSMENU         0x00080000L
#define WS_ THICKFRAME      0x00040000L
#define WS_ MINIMIZEBOX     0x00020000L
#define WS_ MAXIMIZEBOX     0x00010000L
                          ===========
                          0x14CF0000
```

For the sake of clarity, temporarily ignore the next two lines in the figure (I'll come back to them in a little while). After CLOCK has retrieved its style bits with GetWindowLong, it sets a slightly different set of style bits with the call to SetWindowLong. In this call, the style bits are 0x14840000. Apparently, CLOCK retrieves its WS_ xxx style bits, modifies a few of them, and then sends the revised style bits back to the window. So what styles did CLOCK change? Comparing the original 0x14CF0000 with the new 0x14840000, we can see that the new style DWORD is missing the following styles from the original value:

```
#define WS_ DLGFRAME        0x00400000L
#define WS_ SYSMENU         0x00080000L
#define WS_ MINIMIZEBOX     0x00020000L
#define WS_ MAXIMIZEBOX     0x00010000L
```

This is consistent with CLOCK behavior. When you select the No Title menu item, the system menu and the minimize and maximize buttons go away.

Let's return to the two skipped lines. The first of these lines is a call to SetWindowLong. This line appears to be setting the window's control ID (GWL_ ID) to 0. With only that information to go on, you're probably confused as to what the intent is. Here's the secret: All windows have an internal field that can be either a control ID or a menu handle. Child windows (such as dialog- box controls) use this field to hold their control ID. Top- level windows (such as CLOCK. EXE's) use this field to hold a menu handle (HMENU). (For official verification of this, refer to the hMenu field description in the documentation for CreateWindow.) Knowing this obscure fact, you can see that CLOCK. EXE sets its window's HMENU to 0. It may have been clearer if CLOCK had used SetMenu to change the HMENU value. But this seemingly gratuitous use of undocumented functionality can be explained. The last line in the trace calls SetWindowPos. This call forces Windows to repaint the window. Had the code called SetMenu, Windows would have forced the window to redraw prematurely-- that is, before the SetWindowLong call that changes the style bits and before SetWindowPos.

CONCLUSION

Using commonly available file- dumping and API- spying programs, you can peer into a program's operations to see how it works. In many cases, this is all the reverse engineering that you need to get the job done. In a follow- up column, you'll learn how to use disassemblers for those situations when you need to bring even more power to bear on the problem.

PC Magazine June 28, 1994 v13 n12 p275(6)

Disassembling Windows programs.

(PC Tech) (Power Programming)

Author

Pietrek, Matt

Abstract

A tutorial details the process of disassembling source code in Windows applications. Disassembly is typically the best way to accomplish the reverse engineering of a problematical algorithm. Knowing how to correlate high- level language code with the assembler code generated by the compiler can be an extremely valuable and time- saving skill. The basic steps in disassembly include obtaining a listing file by running the executable, labeling the known entities, inserting blank lines before instruction sequences, providing explanations of string- literal values, consolidating assembly instructions and converting conditional branch statements to their linguistic equivalencies. Other steps in the tutorial include the identification of functions and parameters, local and global variables, and identifying 'if' statements and string literals.

Full Text

In the last Power Programming column, I described two relatively easy approaches to reverse- engineering Microsoft Windows programs: dumping program files and using utilities that spy on a program's messages and API calls. In recent years, books such as Undocumented DOS, Undocumented Windows, and Windows Internals have furnished programmers with a third technique for reverse- engineering: disassembling program files. In this column, I'll show some basic techniques for reverse- engineering 16- and 32- bit Windows code via disassembly.

When other methods fail, disassembly is often the only way to crack open a mysterious algorithm or programming technique. And disassembling a program or a DLL isn't necessarily something you do only with other people's code. When you encounter a strange bug that's not immediately apparent from the source for your own code, knowing how to correlate high- level language code to the compiler- generated assembly code is an incredibly valuable skill. Likewise, disassembling your own code enables you to determine whether the compiler has generated optimal code for a heavily used routine. Yet another situation where you might disassemble your own code is when a program mysteriously crashes at a customer site. If the user can give you the address where the program blows up, you can disassemble your program at that address to see what the code is doing.

The most familiar PC- based disassembler is probably V- Communications' Sourcer. By itself, Sourcer works with DOS .EXE and .COM files. With add- on components that produce script files, Sourcer also takes apart 16- bit NE (New Executable) files, VxD LE (Linear Executable) files, and Win32 PE (Portable Executable) files. Other disassemblers for the PC include Eclectic Software's Win2Asm and DisxDoc Professional, from RJSwantek. If you're only concerned with Win32 files and price, it's hard to beat the DUMPBIN program that comes with the Microsoft SDK for Win32. The DUMPBIN program is embedded in LINK. EXE: Use the syntax LINK /DUMP /DISASM filename, where filename is the name of a COFF (Common Object File Format) .OBJ or a Win32 PE format executable. For you Microsoft Visual C++ 32- Bit Edition users, from the command line, type DUMPBIN /DISASM filename.

If you just want to tinker around with the disassembly techniques shown here, you may be able to get away with using the disassembler in your debugger. By dumping the contents of several disassembly windows into a file, you can get a somewhat reasonable listing. This is tedious and time consuming, however. If you're at all serious about disassembly, get a real file- based disassembler. They're well worth the cost.

I CAME, I SAW, I DISASSEMBLED

Although there is no one correct approach to reverse- engineering a piece of code, my approach can be summed up as "divide and conquer." Starting with the raw output from a disassembler, I don't tackle an entire function in one big piece. Instead, I break up the raw listing into small, manageable pieces. My ultimate goal is to work a disassembly listing into a piece of commented C pseudo- code. My book, Windows Internals, has many examples of this. Another use of disassemblers is to create an .ASM file that you can modify and then reassemble (when you don't have the source code, for example). This aspect of disassembly won't be covered in this column.

I'll first describe in general terms the steps I take when reverse- engineering. Then, I'll jump into the nitty- gritty details of identifying functions, function return values, parameters, local variables, global variables, string literals, and branch statements.

THE STRATEGY

The steps in disassembling a function are

1. Run the executable through a disassembler to get a listing file. If you're interested in only one function, you might find it helpful to delete other code that comes before or after the relevant function. This makes the file more manageable in your editor.

2. Go through the function and label all the known entities with names that are more descriptive. By known entities, I mean arguments to the function, local variables, and global variables. The idea is to do all the easy work first. When doing a puzzle, most people do the borders first. This reduces the number of unknown pieces you have to sort through and gives a better context from which to fill in the remaining pieces. This same concept applies to disassembly.

3. Disassembly listings often contain long sequences of instructions with no intervening blank lines. It's helpful to insert blank lines before and after instruction sequences that logically belong together. This sounds vague, but it's not hard in practice. An example of such a sequence is the code that pushes parameters onto the stack and then calls another function. A helpful guideline is to try to create sequences of instructions that form a single statement in the program's source code.

Later on in the disassembly process, you may need to decode branch statements. In high- level languages, these are statements such as if, while, do, and switch. I've found that putting a blank line after each jump instruction makes it much easier to understand the listing. I do this quickly with an editor macro that searches for instructions that start with the letter J and inserts a blank line following that instruction.

4. If the function looks like it uses any string- literal values, add comments that contain the string. Put the comments near the function calls that use the string. Later on, this will help reduce several lines of assembly code down to one C statement.

5. Consolidate groups of assembly instructions into single pseudo- code statements. Find the instruction sequences that constitute calls to other functions for which you know the names and parameters. Study what's being pushed onto the stack and construct what the arguments to the function should look like.

6. Convert conditional branch statements into their high- level- language equivalents.

7. Repeat the preceding steps as necessary.

This last step sounds trite, but it's not supposed to be. For all but the most trivial functions, reverse engineering is an iterative process. You make a pass through the code, doing as much as you can with the information you currently have. You then step back, look at how the picture has changed, and make another pass. By figuring out one piece of the puzzle, a dozen more may quickly fall into place.

Having discussed in broad strokes how to disassemble a function, let's look at some common code sequences and code- generation conventions. This will help you translate raw assembly code into its high- level- language equivalent.

IDENTIFYING FUNCTIONS

The first thing to do with the raw output from a disassembler is to figure out where the functions start and end. To find the start of a function, look for some sort of standard prologue code generated by a compiler. Of course, there is always the possibility that the code was written in assembly language, so there may not be any sort of prologue code.

The standard compiler- generated prologue code is some variation of this:

* save original (E) BP register on the stack

* assign the stack pointer to the (E) BP register

* decrement the stack pointer to make room for local variables

* save the calling function's register variables on the stack

Expressed in 16- bit assembly language, the above looks like this:

```
PUSH BP         ; Save caller's BP frame
MOV BP, SP      ; Set up new BP
                ; frame
SUB SP, XX      ; XX is the number of
                ; bytes need for local
                ; variables
PUSH SI         ; DI and SI are
PUSH DI         ; commonly used as
                ; register variables
```

When 80286 or better code generation is enabled, look for

```
ENTER XX, 0     ; XX is the number of
PUSH SI         ; bytes needed for
PUSH DI         ; local variables
```

The above sequences are the full- blown prologues. In real- world code, parts or all of the prologue may be missing or different. If the function doesn't alter a register- variable register--(E) SI and (E) DI, for example-- the prologue won't bother to save it. In 32- bit code, EBX is sometimes used as a register variable; in 16- bit code, BX usually isn't.

In 16- bit code, if the function doesn't take any parameters or use any local variables, the compiler may omit

```
PUSH BP
MOV BP, SP
```

In 32- bit code, even if the function takes parameters and uses local variables, the compiler may still not set up an EBP frame. The 32- bit addressing modes of 386 and above CPUs allow the compiler to address parameters and local variables with the ESP register:

```
MOV EAX,[ ESP+ 1C]
```

Recognizing the function epilogue is a little trickier. If the compiler's optimizer is turned on, there may be multiple places where the function does a RET or RETF to the calling function. Assuming the function has a single epilogue at the end of the function, the full- blown 16- bit epilogue looks something like this:

```
POP DI     ; Restore caller's
           ; register variables
POP SI
ADD SP, XX ; or LEAVE for
POP BP     ; 80286 and up
RETF 8     ; Far return, pop 8 bytes
           ; off the stack. Near
           ; return is a RET.
```

When determining where one routine starts and another ends, remember that right after the end of one routine, you're likely to find the start of another. If you see something that looks like epilogue code, verify it by looking for something that looks like prologue code for another function right after it.

FUNCTION RETURN VALUES

In 32- bit code, functions return their values in EAX; 16- bit code uses AX for returning 16- bit values and the DX: AX combination for returning 32- bit values. If the code is written in assembly language, however, all bets are off, since the programmer can return values anywhere. For instance, KRNL386. EXE returns values in CX all over the place. One common assembler convention is that if the routine needs to return only a success or a failure code, the routine sets or clears the carry flag (CF) as appropriate. You can ferret out these routines by looking for JC and JNC instructions right after CALL instructions.

IDENTIFYING PARAMETERS

If you know the parameters for the function you're taking apart, labeling them in the assembly code is particularly easy. With the exception of fast- call functions (where parameters are passed in registers), compilers always pass arguments to a function or a procedure on the stack. By adding up the sizes of each parameter passed, you can quickly locate where each parameter resides on the stack. Before showing an example of this, I need to do a quick review of compiler calling conventions used in Windows and Win32.

In 16- bit Windows code, almost all exported functions use the Pascal calling convention. In the Pascal calling convention, the calling code pushes parameters onto the stack from the leftmost parameter to the rightmost parameter. Thus, the 16- bit code generated for a call to foo(0x10, 0x20, 0x30) would look something like this:

```
PUSH 0010h
PUSH 0020h
PUSH 0030h
CALL FAR PTR FOO
```

Besides specifying that parameters are passed from left to right, the Pascal calling convention also dictates that the called function removes the arguments from the stack before returning. In the above example, the foo() function needs to pop 6 bytes off the stack before it returns. It will probably do this with a RETF 6 instruction.

The opposite of the Pascal calling convention is the C calling convention. The standard C/ C++ runtime library functions use the C calling convention. In the C calling convention, parameters are passed from the rightmost to the leftmost. The code that calls the function is responsible for removing the parameters from the stack after the call returns. A call to foo(0x10, 0x20, 0x30) using the C calling convention would appear like this:

```
PUSH 0030h ; Parameters pushed
PUSH 0020h ; right- to- left
PUSH 0010h ;
CALL FAR PTR FOO
ADD SP, 06h ; remove parameters
          ; from the stack
```

You shouldn't always expect to see an ADD (E) SP, XX after a C- style call. If the compiler pushed only one or two parameters, the compiler will sometimes pop the parameters into an unneeded register to remove them from the stack.

For Win32, Microsoft has adopted the stdcall calling convention for almost all functions that the operating system exports. The stdcall convention is a hybrid of the C and Pascal conventions. The caller pushes the parameters from right to left as in the C style. The called function cleans the parameters off the stack as the Pascal style does. (Incidentally, when you use a stdcall function with a Microsoft Win32 compiler, the compiler internally adds an @xx to the end of the function name. The xx is a string representing the number of bytes that the function expects as parameters-- for instance, _GetWindowLong@ 8 or _PeekMessage@ 20. If you dump out the import libraries created for a DLL, you'll see this mangled version of the function name.)

After you've figured out which calling convention the function you're examining uses, you can determine where the parameters are on the stack. Knowing the offset of a parameter relative to the stack frame, you can look for the instructions that reference that memory location and replace the assembly- language address with a symbolic name. Having meaningful parameter names when working on a disassembly listing is immensely helpful.

After a function has executed its prologue code, the stack frame looks like this:

* parameters
* return address (placed by the CALL instruction)
* previous (E) BP (pushed by the prologue code)

The (E) BP register points to where the previous (E) BP value is saved. All of the parameters can now be accessed within the function as positive displacements from BP or EBP. This is a key point worth restating: Instructions that access memory using addresses such as [BP+ xx] or [EBP+ xx] are referring to the routine's parameters.

For a far 16- bit function such as those exported by Windows 3. x, the actual stack frame looks like this:

* parameters (starting at BP+ 06)
* return CS (at BP+ 04)
* return IP (at BP+ 02)
* previous BP (at BP+ 00)

Assuming word- size arguments and the Pascal calling convention, the last parameter to the function will be at [BP+ 06], the second to last parameter at [BP+ 08], and so on. If there are any DWORD parameters, the calculations need to be adjusted accordingly.

Let's look at a real- world example to get a better feel for stack parameters. A window procedure for a 16- bit program has the following declaration:

```
LRESULT WINAPI WndProc( HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
```

Inside the WndProc code, the stack frame looks like this (note the parameters have been pushed from left to right):

```
hWnd        WORD PTR [BP+ 0E]
msg         WORD PTR [BP+ 0C]
wParam      WORD PTR [BP+ 0A]
lParam D    WORD PTR [BP+ 06]
return CS   WORD PTR [BP+ 04]
return IP   WORD PTR [BP+ 02]
previous BP WORD PTR [BP+ 00]
```

Armed with this knowledge, you can use your editor's search- and- replace feature to find all the references to [BP+ 0E] and replace them with the much more meaningful [hWnd]. Likewise, you can replace [BP+ 0C] with [msg] and so forth.

Now let's look at the 32- bit equivalent to the above window procedure. In Win32, all parameters are 32 bits. The return address is a 32- bit near pointer, and the code uses EBP rather than BP. And don't forget that the window procedure uses the stdcall convention, so the parameters appear in the reverse order from the equivalent 16- bit code. The stack frame for a 32- bit window procedure looks like this:

```
lParam        DWORD PTR [EBP+ 14]
wParam        DWORD PTR [EBP+ 10]
msg           DWORD PTR [EBP+ 0C]
hWnd          DWORD PTR [EBP+ 08]
return EIP    DWORD PTR [EBP+ 04]
previous EBP  DWORD PTR [EBP+ 00]
```

Now that I've described the layout of the normal stack frame for a 32- bit function, it's time to tell you the bad news that I briefly alluded to earlier. A 32- bit compiler has the option of not generating standard EBP frames; instead, the code may address the parameters with an offset from ESP (for example, [ESP+ 14]). If this thought alarms you, it should! The value of ESP changes throughout the function as it pushes parameters in preparation for calling other routines. Thus, an [lParam] that's at [ESP+ 14] early on in the function could later be found at [ESP+ 18] if the code pushes a DWORD onto the stack. If the code pushes a second DWORD, [lParam] would be at [ESP+ 1C] and so on.

If you're taking apart a function that you don't know the parameters for, there are still some tricks you can use. You'll definitely want to figure out how many bytes of parameters the function expects. Look at the exit prologue of the function. Does it pop parameters off the stack with something like RETF 8? If so, you know how many bytes of parameters the function takes (in this case, 8). If the function's exit code doesn't remove anything from the stack, find a place in the code where there's a call to the function. Is the next instruction after the call something like ADD ESP, 12? If so, the function takes 12 bytes of parameters.

Beyond knowing how many bytes of parameters a function takes, you can often glean information by studying the code that pushes the parameters onto the stack. For instance, consider the following disassembly- listing fragment from a Win32 program:

```
CALL GetFocus
PUSH EAX
CALL GetCurrentThread
PUSH EAX
CALL DoSomething
```

It's easy to see that the DoSomething function takes two parameters, an HWND and a thread HANDLE. How do you know that? Both GetFocus and GetCurrentThread are Win32 functions that return a value in EAX. After calling GetFocus, EAX holds an HWND value. After calling GetCurrentThread, EAX holds a thread HANDLE. The code pushes both of these values onto the stack and calls a function. By logical deduction, the DoSomething function expects an HWND and a HANDLE as parameters. This means you can calculate where DoSomething's parameters are on the stack and give them meaningful names.

IDENTIFYING LOCAL VARIABLES

Like function parameters, a routine's local variables are also usually found on the stack. (The exception to this rule is when the compiler keeps a heavily used local variable in a register.) The key difference in distinguishing between a parameter and a local variable is that the code references local variables with a negative offset from the stack frame-- for example, [BP- 4] in 16- bit code or [EBP- 4] in 32- bit code.

Unlike parameters, there's no mechanical method for determining the types, uses, and locations of local variables. Instead, you have to examine how the function's code uses a particular memory location. Sometimes you can easily determine a local variable's meaning and label it. Consider the following Win32 code snippet:

```
PUSH DWORD PTR [EBP+ 08]
CALL GetParent
MOV [EBP- 0C], EAX
```

The Win32 GetParent function takes an HWND parameter and returns that window's parent HWND in EAX. Since the code snippet copies EAX into [EBP- 0C], it's obvious that [EBP- 0C] is an HWND. Additionally, you can make a wild guess that this variable is probably called something like hWndParent in the original source code. Once you've gotten this far, it's time to use your editor's search- and- replace feature to change all [EBP- 0C] 's to [hWndParent]. Look at your disassembly listing after you do this. Odds are it's starting to become more clear.

Sometimes it's easier to identify local variables from their use as parameters to other functions. This Win32 assembly fragment shows such an example:

```
LEA EAX,[ EBP- 30] ; Get value of
                   ; EBP- 30h into EAX
PUSH EAX           ; Push it as an LPRECT
PUSH [EBP+ 08]     ; HWND parameter
CALL GetWindowRect ; Call into
                   ; USER32 to get the RECT coordinates
```

From examining the Microsoft Windows Software Development Kit (SDK), we know that GetWindowRect takes an HWND and a pointer to a RECT structure that has to be filled in. Since GetWindowRect is a stdcall function, the RECT pointer should be pushed first, followed by the HWND. In the listing, we see that for the LPRECT parameter, the code pushes an address 30h bytes below the EBP value. Therefore, there must be a local variable of type RECT at [EBP- 30]. This is a bonanza of information! Since WINDEF. H contains the format of a RECT structure (four DWORDS), we can now figure out where all of the RECT's fields are on the stack:

```
RECT. left = [EBP- 30]
RECT. top = [EBP- 2C]
RECT. right = [EBP- 28]
RECT. bottom = [EBP- 24]
```

IDENTIFYING GLOBAL VARIABLES

Determining that a function uses a global variable is particularly easy. Almost any memory reference that uses a hard- coded address is a global variable. Global variables don't require the assistance of registers such as EBP to address them. In 32- bit code, a global- variable reference looks something like

```
MOV EAX, [00464398]
```

If you're lucky and happen to have symbol information (debug information that somebody accidentally left in the executable, for example), the disassembler may already have replaced the [00464398] with the name used in the program's source. If not, you should find all the instructions that use that memory location and replace that address with the symbolic name. If you don't have symbolic information, try to figure out what the variable is used for and then make up your own name.

In 16- bit code, identifying global variables is pretty much the same as in 32- bit code, albeit with 16- bit addresses rather than 32- bit. If the code you're working with has multiple data segments, however, you'll need to be extra careful. The problem is that the same offset can be used in several data segments. When accessing global variables in a segment other than the default DGROUP, the code usually sets up a segment register (usually ES) to point to that segment. The code then accesses variables in the segment with hard- coded offsets-- for instance,

```
MOV AX, ES:[001C]
```

The lesson here is to be careful when replacing global- variable addresses with symbolic names. Make sure the segment register really is pointing to the segment that you think it is. Otherwise, you run the risk of labeling a global variable with the wrong name.

If you have symbolic information for an executable file but encounter a memory location that's not in the list of global variables, there are two probable explanations. First, that memory location might be used for a static variable. If your symbol information includes only public symbols, the variable won't show up in the list. The other possibility is that you're looking at a member of a structure or an array. For instance, a 16- bit program has a global variable MSG MyMsg that ends up in the program's DGROUP segment at offset 0364h. The wParam field lies 4 bytes into the MSG structure. MyMsg. wParam will therefore be at offset 0368h in the data segment. Symbol information generated for this executable will include a public symbol called MyMsg at offset 0364h but will contain nothing about offset 0368h. This is where experience comes in. You might guess that MyMsg is a structure of type MSG. Working from that assumption, you would figure out the addresses of the individual structure members and look for code that uses those memory locations. If the code uses the memory locations in a manner consistent with your hypothesis, you've probably guessed correctly.

IDENTIFYING STRING LITERALS

Many API functions take strings as parameters. By matching up the ASCII strings with the functions that use them, you can often get a much better idea of what the code is doing. For instance, in a 16- bit program, you might encounter

```
PUSH DS
PUSH 0437
CALL GETMODULEHANDLE
```

Turning to your trusty API documentation, you see that the GetModuleHandle function takes one argument, an LPSTR. The PUSH instructions are pushing the address of a string onto the stack as the parameter to GetModuleHandle. At address DS: 0437, therefore, there must be a null- terminated string (for example, "USER"). If your disassembler has done a hexadecimal and ASCII dump of the file's data sections, go to the address and retrieve the string. Back in the code that referenced the string literal, make a comment that includes the retrieved string-- for instance, you would use

```
PUSH DS
PUSH 0437 ; = "USER"
CALL GETMODULEHANDLE
```

If the code you're disassembling uses a lot of string literals, you'll be amazed how much clearer it becomes after you do this.

IDENTIFYING IF STATEMENTS

The simplest type of conditional execution code to figure out is a simple if statement:

if (some test)
do some sequence of code

Viewed from the disassembly- listing level, there are three primary types of tests that you encounter:

* equality: if (a == b), if (a != b)
* Boolean TRUE/ FALSE: if (a), if (! a)
* bitfield tests: if (a & 0x0040)

Although compilers generate different code sequences for each type of test, the goal in each case is to set or clear the CPU's zero flag (ZF). After setting or clearing the zero flag, the code uses the JZ or JNZ conditional branch instructions either to execute the next section of code or to skip over it. JZ stands for "jump if zero" while JNZ means "jump if not zero." In the spirit of keeping assembly language confusing, the JZ instruction mnemonic can also be expressed as JE (jump if equal). Likewise, JNZ can also be coded as JNE (jump if not equal).

The idea behind the "test, then conditionally jump" model is this: If the test expression resolves to FALSE, the CPU takes the conditional jump, and the code inside the 's or the BEGIN/ END block doesn't get executed. If the expression evaluates to TRUE, the conditional jump isn't taken, and control falls into the code inside the block in C or between a BEGIN/ END block in Pascal.

Warning: What I've just described here is the simple version of what occurs. In the real world, the generated code might be more complex. In 16- bit code, there might be a JZ or JNZ instruction whose only job is to jump over a regular JMP statement. The JMP statement will jump to the end of the conditional block. This happens if the code inside the if block was longer than 127 bytes, the limit of a 16- bit JE or JNE instruction.

For equality tests, compilers use the CMP instruction. This snippet of output produced with DUMPBIN /DISASM shows an example:

```
0000101E: cmp dword ptr [ebp- 04], 04
00001022: jne 0000102E
00001028: inc byte ptr [ebp- 04]
0000102B: inc byte ptr [ebp- 08]
0000102E: ...
```

The first instruction compares the DWORD at [EBP- 04] with the value 4. If they're the same, the CMP instruction sets the zero flag; otherwise, it clears the zero flag. The next instruction (JNE) jumps over the code that follows but only if the zero flag was cleared. Therefore, the two INC instructions execute only if the zero flag was set. The zero flag could be set only if [EBP- 04] was equal to 4. Expressed in C code, the above snippet could look something like this:

```
if ( SomeVariable1 == 4 )

SomeVariable1++; // INC [EBP- 04]
SomeVariable2++; // INC [EBP= 08]
```

In 16- bit code, when the code inside the 's is greater than 128 bytes in length, the above expression could be

```
; Is the value 4?
0176: CMP WORD PTR [BP- 02], 04
; Yes? jump to  code
017A: JE 017F
; No? Jump around  code
017C: JMP 085A
017F: ... ; start of the  code
```

When the expression in the if statement is concerned only with whether the expression is TRUE or FALSE, the compiler has alternative code- generation options. In some cases, the generated code can look like the if statement described earlier. For example, the expression if (MyVariable) could also be written as if (MyVariable != 0). Another sequence occurs when the expression's value is in a register. When this happens, the compiler can use a smaller instruction to determine whether the value is TRUE (nonzero) or FALSE (0). The shorter instruction is an OR register, register instruction, like this:

```
0000102E: call 00001000
00001033: or eax, eax
00001035: je 0000103E
0000103B: inc byte ptr [ebp- 04]
0000103E: ...
```

In this code, the first instruction calls a function that returns its value in EAX. Rather than using 3 bytes with a CMP EAX, 0, the compiler uses an OR instruction. The OR instruction does a logical OR on all the bits in EAX. Only if none of them are set (EAX == 0) will the zero flag be set.

The third instruction sequence you'll see generated for if statements occurs when individual bits are involved. Many WORDs and DWORDs in Windows programming are composed of a set of 1- bit flags, such as the WS_ xxx style bits you pass to CreateWindow. Code that needs to test whether a bit is set uses the language's bitwise AND operator (in C, the & operator). Consider the following C fragment:

DWORD winFlags = GetWinFlags(); if (winFlags & WF_ CPU386) is386 = TRUE;

The assembly code generated for the if statement looks like this:

```
0000102E: test byte ptr [ebp- 08], 04
00001032: je 0000103F
00001038: mov dword ptr [ebp- 0C], 00000001
0000103F: ....
```

where 0004h corresponds to WF_ CPU386. The first instruction uses the CPU's TEST instruction to see whether the bit representing the value 4 is set. The TEST instruction performs a logical AND on the corresponding bits in the two operands but doesn't overwrite either of the operands. Instead, TEST sets the zero flag if the result doesn't have any bits set. If any bits were set, TEST clears the zero flag. If the bit representing the value 4 was set in [EBP- 08], the zero flag will be clear, the JE instruction won't jump, and the DWORD at [EBP- 0C] will be incremented.

Only slightly more complicated than an if statement is an if- else statement. Consider the following elaboration of a previous example:

```
if ( i == 4 )
i++;

j++;

 else j--;
```

The compiler generates the following code:

```
0000101E: cmp dword ptr [ebp- 04], 04
00001022: jne 00001033
00001028: inc byte ptr [ebp- 04]
0000102B: inc byte ptr [ebp- 08]
0000102E: jmp 00001036
00001033: dec byte ptr [ebp- 08]
00001036: ...
```

The first two instructions look identical to the code for a plain if statement. The JMP instruction a bit later is the key. After executing the code when the expression is true, this JMP instruction skips over the code generated for the else clause. The target address of the JMP instruction identifies where the else clause's code ends. The key things to look for in identifying if- else statements are these: Does the initial JE/ JNE instruction jump to an instruction that's immediately after a JMP instruction? Does the JMP instruction transfer to a higher address (that is, does it goes forward in memory rather than backward)?

I've covered only the basics of branching constructs here. There are certainly more complicated things you will encounter. I haven't covered if statements with multiple conditions, for loops, while loops, and other branching constructs. Almost everything you'll encounter, however, can be broken down into combinations and variations of the code sequences that I've shown here.

PUTTING IT ALL TOGETHER

After you've done all the mechanical portions of the disassembly process, it's time to take the intermediate results and work them into lines of understandable C code. For example, you can replace the following:

```
PUSH DS
PUSH 0437 ; DS: 0437 = "USER"
CALL GETMODULEHANDLE
MOV SI, AX
with its C equivalent:
SomeLocalVariable = GetModuleHandle
( "USER" );
```

Not every group of assembly instructions converts so easily. There are no hard- and- fast rules. This is where experience and gut instincts come in. Try to put yourself in the position of the programmer who wrote the code. Look for common patterns that might establish a coding convention. One useful technique for learning the subtleties of this process is to disassemble and analyze your own programs. This helps get a feel for how a compiler expresses common high- level- language expressions in assembly language.

FURTHER READING

If you'd like to delve deeper into the topic of reverse engineering through disassembly, check out the following books:

* Undocumented DOS, Second Edition, by Andrew Schulman, Ralf Brown, David Maxey, Raymond J. Michels, and Jim Kyle, Addison- Wesley. ISBN: 0- 201- 63287- X.

* Undocumented Windows, by Andrew Schulman, David Maxey, and Matt Pietrek, Addison- Wesley. ISBN: 0- 201- 60834- 0.

* Windows Internals, by Matt Pietrek, Addison- Wesley. ISBN: 0- 201- 62217- 3.

NOT JUST FOR WIZARDS

Disassembly is messy, imprecise, and frustrating work. On the other hand, it can be an extremely valuable skill that few programmers take the time to master. If you're not already using these tools and techniques, I hope I've taken away the mystique. If you have a firm grounding in hardware and operating- system basics, disassembly can be just another part of your toolbox rather than something reserved for programming wizards.

Matt Pietrek is the author of Windows Internals (Addison- Wesley, 1993) and is a contributing editor of Microsoft Systems Journal. He is also a software architect at Nu- Mega technologies