CHAPTER

5

Beyond the Documentation

Twenty years ago, programs could almost exist in isolation, barely having to interface with anything other than the underlying hardware, with which they frequently communicated directly. Needless to say, things have changed quite a bit since then. Nowadays the average program runs on top of a humongous operating system and communicates with dozens of libraries, often developed by a number of different people.

This chapter deals with one of the most important applications of reversing: reversing for achieving interoperability. The idea is that by learning reversing techniques, software developers can more efficiently interoperate with third-party code (which is something *every* software developer does every day). That's possible because reversing provides the ultimate insight into the third-party's code—it takes you beyond the documentation.

In this chapter, I will be demonstrating the relatively extreme case where reversing techniques are used for learning how to use undocumented system APIs. I have chosen a relatively complex API set from the Windows native API, and I will be dissecting the functions in that API to the point where you fully understand what that each function does and how to use it. I consider this an extreme case because in many cases one does have *some* level of documentation—it just tends to be insufficient.

Reversing and Interoperability

For a software engineer, interoperability can be a nightmare. From the individual engineer's perspective, interoperability means getting the software to cooperate with software written by someone else. This other person can be someone else working in the same company on the same product or the developer of some entirely separate piece of software. Modern software components frequently interact: applications with operating systems, applications with libraries, and applications with other applications.

Getting software to communicate with other components of the same program, other programs, software libraries, and the operating system can be one of the biggest challenges in large-scale software development. In many cases, when you're dealing with a third-party library, you have no access to the source code of the component with which you're interfacing. In such cases you're forced to rely exclusively on vendor-supplied documentation. Any seasoned software developer knows that this rarely turns out to be a smooth and easy process. The documentation almost always neglects to mention certain functions, parameters, or entire features.

One excellent example is the Windows operating system, which has historically contained hundreds of such undocumented APIs. These APIs were kept undocumented for a variety of reasons, such as to maintain compatibility with other Windows platforms. In fact, many people have claimed that Windows APIs were kept undocumented to give Microsoft an edge over one software vendor or another. The Microsoft product could take advantage of a special undocumented API to provide better features, which would not be available to a competing software vendor.

This chapter teaches techniques for digging into any kind of third-party code on your own. These techniques can be useful in a variety of situations, for example when you have insufficient documentation (or no documentation at all) or when you are experiencing problems with third-party code and you have no choice but to try to solve these problems on your own. Sure, you should only consider this approach of digging into other people's code as a last resort and at least *try* and get answers through the conventional channels. Unfortunately, I've often found that going straight to the code is actually *faster* than trying to contact some company's customer support department when you have a very urgent and *very technical* question on your hands.

Laying the Ground Rules

Before starting the first reversing session, let's define some of the ground rules for every reversing session in this book. First of all, the reversing sessions in

this book are focused exclusively on offline code analysis, not on live analysis. This means that you'll primarily just read assembly language listings and try to decipher them, as opposed to running programs in the debugger and stepping through them. Even though in many cases you'll want to combine the two approaches, I've decided to only use offline analysis (dead listing) because it is easier to implement in the context of a written guide.

I could have described live debugging sessions throughout this book, but they would have been very difficult to follow, because any minor environmental difference (such as a different operating system version of even a different service pack) could create confusing differences between what you see on the screen on what's printed on the page. The benefit of using dead listings is that you will be able to follow along everything I do just by reading the code listings from the page and analyzing them with me.

In the next few chapters, you can expect to see quite a few longish, *uncommented* assembly language code listings, followed by detailed explanations of those listings. I have intentionally avoided commenting any of the code, because that would be outright cheating. The whole point is that you will look at raw assembly language code just as it will be presented to you in a real reversing session, and try to extract the information you're seeking from that code. I've made these analysis sessions very detailed, so you can easily follow the comprehension process as it takes place.

The disassembled listings in this book were produced using more than one disassembler, which makes sense considering that reversers rarely work with just a single tool throughout an entire project. Generally speaking, most of the code listings were produced using OllyDbg, which is one of the best freeware reversing tools available (it's actually distributed as shareware, but registration is performed free of charge—it's just a formality). Even though OllyDbg is a debugger, I find its internal disassembler quite powerful considering that it is 100 percent free—it provides highly accurate disassembly, and its code analysis engine is able to extract a decent amount of high-level information regarding the disassembled code.

Locating Undocumented APIs

As I've already mentioned, in this chapter you will be taking a group of undocumented Windows APIs and practicing your reversing skills on them. Before introducing the specific APIs you will be working with, let's take a quick look at how I found those APIs and how it is generally possible to locate such undocumented functions or APIs, regardless of whether they are part of the operating system or of some other third-party library.

The next section describes the first steps in dealing with undocumented code: how to find undocumented APIs and locate code that uses them.

What Are We Looking For?

Typically, the search for undocumented code starts with a requirement. What functionality is missing? Which software component can be expected to offer this functionality? This is where a general knowledge of the program in question comes into play. You need to be aware of the key executable modules that make up the program and to be familiar with the interfaces between those modules. Interfaces between binary modules are easy to observe simply by dumping the import and export directories of those modules (this is described in detail in Chapter 3).

In this particular case, I have decided to look for an interesting Windows API to dissect. Knowing that the majority of undocumented user-mode services in Windows are implemented in `NTDLL.DLL` (because that's where the native API is implemented), I simply dumped the export directory of `NTDLL.DLL` and visually scanned that list for *groups* of APIs that appear related (based on their names).

Of course, this is a somewhat unusual case. In most cases, you won't just be looking for undocumented APIs just because they're undocumented (unless you just find it really cool to use undocumented APIs and feel like trying it out) — you will have a specific feature in mind. In this case, you might want to search that export directory for relevant keywords. Suppose, for example, that you want to look for some kind of special memory allocation API. In such a case, you should just search the export list of `NTDLL.DLL` (or any DLL in which you suspect your API might be implemented) for some relevant keywords such as memory, alloc, and so on.

Once you find the name of an undocumented API and the name of the DLL that exports it, it's time to look for binaries that use it. Finding an executable that calls the API will serve two purposes. First, it might shed some additional light on what the API does. Second, it provides a live sample of how the API is used and exactly what data it receives as input and what it returns as output. Finding an example of how a function is used by live code can be invaluable when trying to learn how to use it.

There are many different approaches for locating APIs and code that uses them. The traditional approach uses a kernel-mode debugger such as Numega SoftICE or Microsoft WinDbg. Kernel-mode debuggers make it very easy to look for calls to a particular function *systemwide*, even if the function you're interested in is not a kernel-mode function. The idea is that you can install systemwide breakpoints that will get hit whenever *any process* calls some function. This greatly simplifies the process of finding code that uses a specific function. You could theoretically do this with a user-mode debugger such as OllyDbg but it would be far less effective because it would only show you calls made within the process you're currently debugging.

Case Study: The Generic Table API in NTDLL.DLL

Let's dive headfirst into our very first hands-on reverse-engineering session. In this session, I will be taking an undocumented group of Windows APIs and analyzing them until I gather enough information to use them in my own code. In fact, I've actually written a little program that uses these APIs, in order to demonstrate that it's really possible. Of course, the purpose of this chapter is not to serve as a guide for this particular API, but rather to provide a live demonstration of how reversing is performed on real-world code.

The particular API chosen for this chapter is the *generic table* API. This API is considered part of the Windows native API, which was discussed in Chapter 3.

The native API contains numerous APIs with different prefixes for different groups of functions. For this exercise, I've chosen a set of functions from the RTL group. These are the runtime library functions that typically aren't used for communicating with the operating system, but simply as a toolkit containing commonly required services such as string manipulation, data management, and so on.

Once you've locked on to the generic table API, the next step is to look through the list of exported symbols in `NTDLL.DLL` (which is where the generic table API is implemented) for every function that might be relevant. In this particular case any function that starts with the letters `Rtl` and mentions a generic table would probably be of interest. After dumping the `NTDLL.DLL` exports using DUMPBIN (see the section on DUMPBIN in Chapter 4) I searched for any `Rtl` APIs that contain the term `GenericTable` in them. I came up with the following function names.

```
RtlNumberGenericTableElements
RtlDeleteElementGenericTable
RtlGetElementGenericTable
RtlEnumerateGenericTable
RtlEnumerateGenericTableLikeADirectory
RtlEnumerateGenericTableWithoutSplaying
RtlInitializeGenericTable
RtlIsGenericTableEmpty
RtlInsertElementGenericTable
RtlLookupElementGenericTable
```

If you try this by yourself and go through the `NTDLL.DLL` export list, you'll probably notice that there are also versions of most of these APIs that have the suffix `Avl`. Since the generic table API is large enough as it is, I'll just ignore these functions for the purposes of this discussion.

From their names alone, you can make some educated guesses about these APIs. It's obvious that this is a group of APIs that manage some kind of a generic list (generic probably meaning that the elements can contain any type of data). There is an API for inserting, deleting, and searching for an element. `RtlNumberGenericTableElements` probably returns the total number of elements in the list, and `RtlGetElementGenericTable` most likely allows direct access to an element based on its index. Before you can start using a generic table you most likely need to call `RtlInitializeGenericTable` to initialize some kind of a root data structure.

Generally speaking, reversing sessions start with data—we must figure out the key data structures that are managed by the code. Because of this, it would be a good idea to start with `RtlInitializeGenericTable`, in the hope that it would shed some light on the generic table data structures.

As I've already explained, I will be relying exclusively on offline code analysis, and not on live debugging. If you want to try out the generic table code in a debugger, you can use `GenericTable.EXE`, which is a little program I have written based on my findings after reversing the generic table API. If you didn't have `GenericTable.EXE`, you'd have to either rely exclusively on a dead listing, or find some other piece of code that uses the generic table. In a quick search I conducted, I was only able to find kernel-mode components that do that (the generic table also has a kernel-mode implementation inside the Windows kernel), but no user-mode components. `GenericTable.EXE` is available along with its source code on this book's Web site at `www.wiley.com/go/eeilam`.

The following reversing session delves into each of the important functions in the generic table API and demonstrates its inner workings. It should be noted that I will be going a bit farther than I have to, just to demonstrate what can be achieved using advanced reverse-engineering techniques. If this were a real reversing session in which you simply needed the function prototypes in order to make use of the generic table API, you could probably stop a lot sooner, as soon as you had all of those function prototypes. In this session, I will proceed to go after the exact layout of the generic table data structures, but this is only done in order to demonstrate some of the more advanced reversing techniques.

RtlInitializeGenericTable

As I've said earlier, the best place to start the investigation of the generic table API is through its data structures. Even though you don't necessarily need to know everything about their layout, getting a general idea regarding their contents might help you figure out the *purpose* of the API. Having said that, let's start the investigation from a function that (judging from its name) is very likely to provide a few hints regarding those data structures: `RtlInitialize GenericTable` is a disassembly of `RtlInitializeGenericTable`, generated by OllyDbg (see Listing 5.1).

```
7C921A39    MOV EDI,EDI
7C921A3B    PUSH EBP
7C921A3C    MOV EBP,ESP
7C921A3E    MOV EAX,DWORD PTR SS:[EBP+8]
7C921A41    XOR EDX,EDX
7C921A43    LEA ECX,DWORD PTR DS:[EAX+4]
7C921A46    MOV DWORD PTR DS:[EAX],EDX
7C921A48    MOV DWORD PTR DS:[ECX+4],ECX
7C921A4B    MOV DWORD PTR DS:[ECX],ECX
7C921A4D    MOV DWORD PTR DS:[EAX+C],ECX
7C921A50    MOV ECX,DWORD PTR SS:[EBP+C]
7C921A53    MOV DWORD PTR DS:[EAX+18],ECX
7C921A56    MOV ECX,DWORD PTR SS:[EBP+10]
7C921A59    MOV DWORD PTR DS:[EAX+1C],ECX
7C921A5C    MOV ECX,DWORD PTR SS:[EBP+14]
7C921A5F    MOV DWORD PTR DS:[EAX+20],ECX
7C921A62    MOV ECX,DWORD PTR SS:[EBP+18]
7C921A65    MOV DWORD PTR DS:[EAX+14],EDX
7C921A68    MOV DWORD PTR DS:[EAX+10],EDX
7C921A6B    MOV DWORD PTR DS:[EAX+24],ECX
7C921A6E    POP EBP
7C921A6F    RET 14
```

Listing 5.1 Disassembly of `RtlInitializeGenericTable`.

Before attempting to determine what this function does and how it works let's start with the basics: what is the function's *calling convention* and how many parameters does it take? The calling convention is the layout that is used for passing parameters into the function and for defining who is responsible for clearing the stack once the function completes. There are several standard calling conventions, but Windows tends to use `stdcall` by default. `stdcall` functions are responsible for clearing their own stack, and they take parameters from the stack in their original left-to-right order (meaning that the caller must push parameters onto the stack in the reverse order). Calling conventions are discussed in depth in Appendix C.

In order to answer the questions about the function's calling convention, one basic step you can take is to find the `RET` instruction that terminates this function. In this particular function, you will quickly notice the `RET 14` instruction at the end. This is a `RET` instruction with a numeric operand, and it provides two important pieces of information. The operand passed to `RET` tells the processor how many bytes of stack to unwind (in addition to the return value). The very fact that the function is unwinding its own stack tells you that this is not a `cdecl` function because `cdecl` functions always let the caller unwind the stack. So, which calling convention is this?

Let's continue this process of elimination in order to determine the function's calling convention and observe that the function isn't taking any registers from the caller because every register that is accessed is initialized within the function itself. This shows that this isn't a `_fastcall` calling convention because `_fastcall` functions receive parameters through `ECX` and `EDX`, and yet these registers are initialized at the very beginning of this function.

The other common calling conventions are `stdcall` and the C++ member function calling convention. You know that this is not a C++ member function because you have its name from the export directory, and you know that it is *undecorated*. C++ functions are always *decorated* with the name of their class and the exact type of each parameter they receive. It is easy to detect decorated C++ names because they usually include numerous nonalphanumeric characters and more than one name (class name and method name at the minimum).

By process of elimination you've established that the function is an `stdcall`, and you now know that the number 14 after the `RET` instruction tells you how many parameters it receives. In this case, OllyDbg outputs hexadecimal numbers, so 14 in hexadecimal equals 20 in decimal. Because you're working in a 32-bit environment parameters are aligned to 32 bits, which are equivalent to 4 bytes, so you can assume that the function receives five parameters. It is possible that one of these parameters would be larger than 4 bytes, in which case the function receives less than five parameters, but it can't possibly be more than five because parameters are 32-bit aligned.

In looking at the function's prologue, you can see that it uses a standard `EBP` stack frame. The current value of `EBP` is saved on the stack, and `EBP` takes the value of `ESP`. This allows for convenient access to the parameters that were passed on the stack regardless of the current value of `ESP` while running the function (`ESP` constantly changes whenever the function pushes parameters into the stack while calling other functions). In this very popular layout, the first parameter is placed at `[EBP + 8]`, the second at `[ebp + c]`, and so on. If you're not sure why that is so please refer to Appendix C for a detailed explanation of stack frames.

Typically, a function would also allocate room for local variables by subtracting `ESP` with the number of bytes needed for local variable storage, but this doesn't happen in this function, indicating that the function doesn't store any local variables in the stack.

Let us go over the function from Listing 5.1 instruction by instruction and see what it does. As I mentioned earlier, you might want to do this using live analysis by stepping through this code in the debugger and actually *seeing* what happens during its execution using `GenericTable.EXE`. If you're feeling pretty comfortable with assembly language by now, you could probably just read through the code in Listing 5.1 without using `GenericTable.EXE`.

Let's dig further into the function and determine how it works and what it does.

```
7C921A3E    MOV EAX,DWORD PTR SS:[EBP+8]
7C921A41    XOR EDX,EDX
7C921A43    LEA ECX,DWORD PTR DS:[EAX+4]
```

The first line loads `[ebp+8]` into `EAX`. We've already established that `[ebp+8]` is the first parameter passed to the function. The second line performs a logical `XOR` of `EDX` against itself, which effectively sets `EDX` to zero. The compiler is using `XOR` because the machine code generated for `xor edx, edx` is shorter than `mov edx, 0`, which would have been far more intuitive. This gives a good idea of what reversers often have to go through—optimizing compilers always favor small and fast code to readable code.

The stack address is preceded by `ss:`. This means that the address is read using SS, the stack segment register. IA-32 processors support special memory management constructs called *segments*, but these are not used in Windows and can be safely ignored in most cases. There are several segment registers in IA-32 processors: CS, DS, FS, ES, and SS. On Windows, any mentioning of any of those can be safely ignored except for FS, which allows access to a small area of thread-local memory. Memory accesses that start with `FS:` are usually accessing that thread-local area. The remainder of code listings in this book only include segment register names when they're specifically called for.

The third instruction, `LEA`, might be a bit confusing when you first look at it. `LEA` (load effective address) is essentially an arithmetic instruction—it doesn't perform any actual memory access, but is commonly used for calculating addresses (though you can calculate general purpose integers with it). Don't let the `DWORD PTR` prefix fool you; this instruction is purely an arithmetic operation. In our particular case, the `LEA` instruction is equivalent to: `ECX = EAX + 4`.

You still don't know much about the data types you've encountered so far. Most importantly, you're not sure about the type of the first parameter you've received: `[ebp+8]`. Proceed to the next code snippet to see what else you can find out.

```
7C921A46    MOV DWORD PTR DS:[EAX],EDX
7C921A48    MOV DWORD PTR DS:[ECX+4],ECX
7C921A4B    MOV DWORD PTR DS:[ECX],ECX
7C921A4D    MOV DWORD PTR DS:[EAX+C],ECX
```

This code chunk exposes one very important piece of information: The first parameter in the function is a pointer to some data structure, and that data structure is being initialized by the function. It is very likely that this data structure is the key or root of the generic table, so figuring out the layout of this data structure will be key to your success in learning to use these generic tables.

One interesting thing about the data structure is the way it is accessed—using two different registers. Essentially, the function keeps two pointers into the data structure, EAX and ECX. EAX holds the original value passed through the first parameter, and ECX holds the address of EAX + 4. Some members are accessed using EAX and others via ECX.

Here's what the preceding code does, step by step.

1. Sets the first member of the structure to zero (using EDX). The structure is accessed via EAX.
2. Sets the third member of the structure to the address of the second member of the structure (this is the value stored in ECX: EAX + 4). This time the structure is accessed through ECX instead of EAX.
3. Sets the second member to the same address (the one stored in ECX).
4. Sets the fourth member to the same address (the one stored in ECX).

If you were to translate the snippet into C, it would look something like the following code:

```
UnknownStruct->Member1 = 0;
UnknownStruct->Member3 = &UnknownStruct->Member2;
UnkownStruct->Member2 = &UnknownStruct->Member2;
UnknownStruct->Member4 = &UnknownStruct->Member2;
```

At first glance this doesn't really tell us much about our structure, except that members 2, 3, and 4 (in offsets +4, +8, and +c) are all pointers. The last three members are initialized in a somewhat unusual fashion: They are all being initialized to point to the address of the second member. What could that possibly mean? Essentially it tells you that each of these members is a pointer to a group of three pointers (because that's what pointed to by UnknownStruct->Member2—a group of three pointers). The slightly confusing element here is the fact that this structure is pointing to itself, but this is most likely just a placeholder. If I had to guess I'd say these members will later be modified to point to other places.

Let's proceed to the next four lines in the disassembled function.

```
7C921A50    MOV ECX,DWORD PTR SS:[EBP+C]
7C921A53    MOV DWORD PTR DS:[EAX+18],ECX
7C921A56    MOV ECX,DWORD PTR SS:[EBP+10]
7C921A59    MOV DWORD PTR DS:[EAX+1C],ECX
```

The first two lines copy the value from the second parameter passed into the function into offset +18 in the present structure (offset +18 is the 7th member). The second two lines copy the third parameter into offset +1c in the structure (offset +1c is the 8th member). Converted to C, the preceding code would look like the following.

```
UnknownStruct->Member7 = Param2;
UnknownStruct->Member8 = Param3;
```

Let's proceed to the next section of `RtlInitializeGenericTable`.

```
7C921A5C    MOV ECX,DWORD PTR SS:[EBP+14]
7C921A5F    MOV DWORD PTR DS:[EAX+20],ECX
7C921A62    MOV ECX,DWORD PTR SS:[EBP+18]
7C921A65    MOV DWORD PTR DS:[EAX+14],EDX
7C921A68    MOV DWORD PTR DS:[EAX+10],EDX
7C921A6B    MOV DWORD PTR DS:[EAX+24],ECX
```

This is pretty much the same as before—the rest of the structure is being initialized. In this section, offset +20 is initialized to the value of the fourth parameter, offset +14 and +10 are both initialized to zero, and offset +24 is initialized to the value of the fifth parameter.

This concludes the structure initialization sequence in `RtlInitialize GenericTable`. Unfortunately, without looking at live values passed into this function in a debugger, you know little about the data types of the parameters or of the structure members. What you do know is that the structure is most likely 40 bytes long. You know this because the last offset that is accessed is +24. This means that the structure is 28 bytes long (in hexadecimal), which is 40 bytes in decimal. If you work with the assumption that each member in the structure is 4 bytes long, you can assume that our structure has 10 members. At this point, you can create a vague definition of the structure, which you will hopefully be able to improve on later.

```
struct TABLE
{
  UNKNOWN         Member1;
  UNKNOWN_PTR     Member2;
  UNKNOWN_PTR     Member3;
  UNKNOWN_PTR     Member4;
  UNKNOWN         Member5;
  UNKNOWN         Member6;
  UNKNOWN         Member7;
  UNKNOWN         Member8;
  UNKNOWN         Member9;
  UNKNOWN         Member10;
};
```

RtlNumberGenericTableElements

Let's proceed to investigate what is hopefully a simple function: `RtlNumber GenericTableElements`. The idea is that if the root data structure has a member that represents the total number of elements in the table, this function would expose it. If not, this function would iterate through all the elements

and just count them while doing that. The following is the OllyDbg output for `RtlNumberGenericTableElements`.

```
RtlNumberGenericTableElements:
7C923FD2    PUSH EBP
7C923FD3    MOV EBP,ESP
7C923FD5    MOV EAX,DWORD PTR [EBP+8]
7C923FD8    MOV EAX,DWORD PTR [EAX+14]
7C923FDB    POP EBP
7C923FDC    RET 4
```

Well, it seems that the question has been answered. This function simply takes a pointer to what one can only assume is the same structure as before, and returns whatever is in offset +14. Clearly, offset +14 contains the number of elements in a generic table data structure. Let's update the definition of the `TABLE` structure.

```
struct TABLE
{
  UNKNOWN         Member1;
  UNKNOWN_PTR     Member2;
  UNKNOWN_PTR     Member3;
  UNKNOWN_PTR     Member4;
  UNKNOWN         Member5;
  ULONG           NumberOfElements;
  UNKNOWN         Member7;
  UNKNOWN         Member8;
  UNKNOWN         Member9;
  UNKNOWN         Member10;
};
```

RtlIsGenericTableEmpty

There is one other (hopefully) trivial function in the generic table API that might shed some light on the data structure: `RtlIsGenericTableEmpty`. Of course, it is also possible that `RtlIsGenericTableEmpty` uses the same `NumberOfElements` member used in `RtlNumberGenericTableElements`. Let's take a look.

```
RtlIsGenericTableEmpty:
7C92715B    PUSH EBP
7C92715C    MOV EBP,ESP
7C92715E    MOV ECX,DWORD PTR [EBP+8]
7C927161    XOR EAX,EAX
7C927163    CMP DWORD PTR [ECX],EAX
7C927165    SETE AL
7C927168    POP EBP
7C927169    RET 4
```

As hoped, `RtlIsGenericTableEmpty` seems to be quite simple. The function loads `ECX` with the value of the first parameter (which should be the root data structure from before), and sets `EAX` to 0. The function then compares the first member (at offset +0) with `EAX`, and sets `AL` to 1 if they're equal using the `SETE` instruction (for more information on the `SETE` instruction refer to Appendix A).

Effectively what this function does is it checks whether offset +0 of the data structure is 0, and if it is the function returns `TRUE`. If it's not, the function returns zero. So, you now know that there must be some important member at offset +0 that is always nonzero when there are elements in the table. Again, we add this little bit of information to our data structure definition.

```
struct TABLE
{
  UNKNOWN_PTR    Member1; // This is nonzero when table has elements.
  UNKNOWN_PTR    Member2;
  UNKNOWN_PTR    Member3;
  UNKNOWN_PTR    Member4;
  UNKNOWN        Member5;
  ULONG          NumberOfElements;
  UNKNOWN        Member7;
  UNKNOWN        Member8;
  UNKNOWN        Member9;
  UNKNOWN        Member10;
};
```

RtlGetElementGenericTable

There are three functions in the generic table API that seem to be made for finding and retrieving elements. These are `RtlGetElementGenericTable`, `RtlEnumerateGenericTable`, and `RtlLookupElementGenericTable`. Based on their names, it's pretty easy to make some educated guesses on what they do. The easiest is `RtlEnumerateGenericTable` because it's obvious that it enumerates some or all of the elements in the list. The next question is what is the difference between `RtlGetElementGenericTable` and `RtlLookupElementGenericTable`? It's really impossible to know without looking at the code, but if I had to guess I'd say `RtlGetElementGenericTable` provides some kind of direct access to an element (probably using an index), and `RtlLookupElementGenericTable` has to search for the right element.

If I'm right, `RtlGetElementGenericTable` will probably be the simpler function of the two. Listing 5.2 presents the full disassembly for `RtlGetElementGenericTable`. See if you can figure some of it out by yourself before you proceed to the analysis that follows.

```
RtlGetElementGenericTable:
7C9624E0    PUSH EBP
7C9624E1    MOV EBP,ESP
7C9624E3    MOV ECX,DWORD PTR [EBP+8]
7C9624E6    MOV EDX,DWORD PTR [ECX+14]
7C9624E9    MOV EAX,DWORD PTR [ECX+C]
7C9624EC    PUSH EBX
7C9624ED    PUSH ESI
7C9624EE    MOV ESI,DWORD PTR [ECX+10]
7C9624F1    PUSH EDI
7C9624F2    MOV EDI,DWORD PTR [EBP+C]
7C9624F5    CMP EDI,-1
7C9624F8    LEA EBX,DWORD PTR [EDI+1]
7C9624FB    JE SHORT ntdll.7C962559
7C9624FD    CMP EBX,EDX
7C9624FF    JA SHORT ntdll.7C962559
7C962501    CMP ESI,EBX
7C962503    JE SHORT ntdll.7C962554
7C962505    JBE SHORT ntdll.7C96252B
7C962507    MOV EDX,ESI
7C962509    SHR EDX,1
7C96250B    CMP EBX,EDX
7C96250D    JBE SHORT ntdll.7C96251B
7C96250F    SUB ESI,EBX
7C962511    JE SHORT ntdll.7C96254E
7C962513    DEC ESI
7C962514    MOV EAX,DWORD PTR [EAX+4]
7C962517    JNZ SHORT ntdll.7C962513
7C962519    JMP SHORT ntdll.7C96254E
7C96251B    TEST EBX,EBX
7C96251D    LEA EAX,DWORD PTR [ECX+4]
7C962520    JE SHORT ntdll.7C96254E
7C962522    MOV EDX,EBX
7C962524    DEC EDX
7C962525    MOV EAX,DWORD PTR [EAX]
7C962527    JNZ SHORT ntdll.7C962524
7C962529    JMP SHORT ntdll.7C96254E
7C96252B    MOV EDI,EBX
7C96252D    SUB EDX,EBX
7C96252F    SUB EDI,ESI
7C962531    INC EDX
7C962532    CMP EDI,EDX
7C962534    JA SHORT ntdll.7C962541
7C962536    TEST EDI,EDI
7C962538    JE SHORT ntdll.7C96254E
7C96253A    DEC EDI
7C96253B    MOV EAX,DWORD PTR [EAX]
```

Listing 5.2 Disassembly of `RtlGetElementGenericTable`.

```
7C96253D    JNZ SHORT ntdll.7C96253A
7C96253F    JMP SHORT ntdll.7C96254E
7C962541    TEST EDX,EDX
7C962543    LEA EAX,DWORD PTR [ECX+4]
7C962546    JE SHORT ntdll.7C96254E
7C962548    DEC EDX
7C962549    MOV EAX,DWORD PTR [EAX+4]
7C96254C    JNZ SHORT ntdll.7C962548
7C96254E    MOV DWORD PTR [ECX+C],EAX
7C962551    MOV DWORD PTR [ECX+10],EBX
7C962554    ADD EAX,0C
7C962557    JMP SHORT ntdll.7C96255B
7C962559    XOR EAX,EAX
7C96255B    POP EDI
7C96255C    POP ESI
7C96255D    POP EBX
7C96255E    POP EBP
7C96255F    RET 8
```

Listing 5.2 *(continued)*

As you can see, `RtlGetElementGenericTable` is a somewhat more involved function compared to the ones you've looked at so far. The following sections provide a detailed analysis of the disassembled code from Listing 5.2.

Setup and Initialization

Just like the previous APIs, `RtlGetElementGenericTable` starts with a conventional stack frame setup sequence. This tells you that this function's parameters are going to be accessed using `EBP` instead of `ESP`. Let's examine the first few lines of `RtlGetElementGenericTable`.

```
7C9624E3    MOV ECX,DWORD PTR [EBP+8]
7C9624E6    MOV EDX,DWORD PTR [ECX+14]
7C9624E9    MOV EAX,DWORD PTR [ECX+C]
```

Generic table APIs all seem to take the root table data structure as their first parameter, and there is no reason to assume that `RtlGetElementGenericTable` is any different. In this sequence the function loads the root table pointer into `ECX`, and then loads the value stored at offset +14 into `EDX`. Recall that in the dissection of `RtlNumberGenericTableElements` it was established that offset +14 contains the total number of elements in the table. The next instruction loads the third pointer at offset +0c from the three pointer group into `EAX`. Let's proceed to the next sequence.

```
7C9624EC    PUSH EBX
7C9624ED    PUSH ESI
7C9624EE    MOV ESI,DWORD PTR [ECX+10]
7C9624F1    PUSH EDI
7C9624F2    MOV EDI,DWORD PTR [EBP+C]
7C9624F5    CMP EDI,-1
7C9624F8    LEA EBX,DWORD PTR [EDI+1]
7C9624FB    JE SHORT ntdll.7C962559
7C9624FD    CMP EBX,EDX
7C9624FF    JA SHORT ntdll.7C962559
```

This code starts out by pushing `EBX` and `ESI` into the stack in order to preserve their original values (we know this because there are no function calls anywhere to be seen). The code then proceeds to load the value from offset +10 of the root structure into `ESI`, and then pushes `EDI` in order to start using it. In the following instruction, `EDI` is loaded with the value pointed to by `EBP + C`.

You know that `EBP + C` points to the second parameter, just like `EBP + 8` pointed to the first parameter. So, the instruction at `ntdll.7C9624F2` loads `EDI` with the value of the second parameter passed into the function. Immediately afterward, `EDI` is compared against –1 and you see a classic case of interleaved code, which is a very common phenomena in code generated for modern IA-32 processors (see the section on execution environments in Chapter 2). Interleaved code means that instructions aren't placed in the code in their natural order, but instead pairs of interdependent instructions are interleaved so that in runtime the CPU has time to complete the first instruction before it must execute the second one. In this case, you can tell that the code is interleaved because the conditional jump doesn't immediately follow the `CMP` instruction. This is done to allow the highest level of parallelism during execution.

Following the comparison is another purely arithmetical application of the `LEA` instruction. This time, `LEA` is used simply to perform an `EBX = EDI + 1`. Typically, compilers would use `INC EDI`, but in this case the compiler wanted to keep both the original and the incremented value, so `LEA` is an excellent choice. It increments `EDI` by one and stores the result in `EBX`—the original value remains in `EDI`.

Next you can see the `JE` instruction that is related to the `CMP` instruction from `7C9624F5`. As a reminder, `EDI` (the second parameter passed to the function) was compared against –1. This instruction jumps to `ntdll.7C962559` if `EDI == -1`. If you go back to Listing 5.2 and take a quick look at the code at `ntdll.7C962559`, you can quickly see that it is a failure or error condition of some kind, because it sets `EAX` (the return value) to zero, pops the registers previously pushed onto the stack, and returns. So, if you were to translate the preceding conditional statement back into C, it would look like the following code:

```
if (Param2 == 0xffffffff)
                                                    return 0;
```

The last two instructions in the current chunk perform another check on that same parameter, except that this time the code is using EBX, which as you might recall is the incremented version of EDI. Here EBX is compared against EDX, and the program jumps to ntdll.7C962559 if EBX is greater. Notice that the jump target address, ntdll.7C962559, is the same as the address of the previous conditional jump. This is a strong indication that the two jumps are part of what was a single compound conditional statement in the source code. They are just two conditions tested within a single conditional statement.

Another interesting and informative hint you find here is the fact that the conditional jump instruction used is JA (jump if above), which uses the carry flag (CF). This indicates that EBX and EDX are both treated as unsigned values. If they were signed, the compiler would have used JG, which is the signed version of the instruction. For more information on signed and unsigned conditional codes refer to Appendix A.

If you try to put the pieces together, you'll discover that this last condition actually reveals an interesting piece of information about the second parameter passed to this function. Recall that EDX was loaded from offset +14 in the structure, and that this is the member that stores the total number of elements in the table. This indicates that the second parameter passed to RtlGetElement GenericTable is an index into the table. These last two instructions simply confirm that it is a valid index by comparing it against the total number of elements. This also sheds some light on why the index was incremented. It was done in order to properly compare the two, because the index is probably zero-based, and the total element count is certainly not. Now that you understand these two conditions and know that they both originated in the same conditional statement, you can safely assume that the validation done on the index parameter was done in one line and that the source code was probably something like the following:

```
ULONG AdjustedElementToGet = ElementToGet + 1;
if (ElementToGet == 0xffffffff ||
    AdjustedElementToGet > Table->TotalElements)
      return 0;
```

How can you tell whether ElementToGet + 1 was calculated within the if statement or if it was calculated into a variable first? You don't really know for sure, but when you look at all the references to EBX in Listing 5.2 you can see that the value ElementToGet + 1 is being used repeatedly throughout the function. This suggests that the value was calculated once into a local variable and that this variable was used in later references to the incremented value. The compiler has apparently assigned EBX to store this particular local variable rather than place it on the stack.

On the other hand, it is also possible that the source code contained multiple copies of the statement ElementToGet + 1, and that the compiler simply

optimized the code by automatically declaring a temporary variable to store the value instead of computing it each time it is needed. This is another case where you just don't know—this information was lost during the compilation process.

Let's proceed to the next code sequence:

```
7C962501    CMP ESI,EBX
7C962503    JE SHORT ntdll.7C962554
7C962505    JBE SHORT ntdll.7C96252B
7C962507    MOV EDX,ESI
7C962509    SHR EDX,1
7C96250B    CMP EBX,EDX
7C96250D    JBE SHORT ntdll.7C96251B
7C96250F    SUB ESI,EBX
7C962511    JE SHORT ntdll.7C96254E
```

This section starts out by comparing `ESI` (which was taken earlier from offset +10 at the table structure) against `EBX`. This exposes the fact that offset +10 also points to some kind of an index into the table (because it is compared against `EBX`, which you *know* is an index into the table), but you don't know exactly what that index is. If `ESI == EBX`, the code jumps to `ntdll.7C962554`, and if `ESI <= EBX`, it goes to `ntdll.7C96252B`. It is not clear at this point why the second jump uses `JBE` even though the operands couldn't be equal at this point or the first jump would have been taken.

Let's first explore what happens in `ntdll.7C962554`:

```
7C962554    ADD EAX,0C
7C962557    JMP SHORT ntdll.7C96255B
```

This code does `EAX = EAX + 12`, and unconditionally jumps to `ntdll.7C96255B`. If you go back to Listing 5.2, you can see that `ntdll.7C96255B` is right near the end of the function, so the preceding code snippet simply returns `EAX` + 12 to the caller. Recall that `EAX` was loaded earlier from the table structure at offset +C, and that while dissecting `RtlInitializeGenericTable`, you were working under the assumption that offsets +4, +8, and +C are all pointers into the same three-pointer data structure (they were all initialized to point at offset +4). At this point one, of these pointers is incremented by 12 and returned to the caller. This is a powerful hint about the structure of the generic tables. Let's examine the hints one by one:

- You know that there is a group of three pointers starting in offset +4 in the root data structure.
- You know that each one of these pointers point into another group of three pointers. Initially, they all point to themselves, but you can safely assume that this changes later on when the table is filled.

- You know that `RtlGetElementGenericTable` is returning the value of one of these pointers to the caller, but not before it is incremented by 12. Note that 12 also happens to be the total size of those three pointers.
- You have established that `RtlGetElementGenericTable` takes two parameters and that the first is the table data structure pointer and the second is an index into the table. You can safely assume that it returns the element through the return value.

All of this leads to one conclusion. `RtlGetElementGenericTable` is returning a pointer to an element, and adding 12 simply skips the element's header and gets directly to the element's data. It seems very likely that this header is another three-pointer data structure just like that in offset +4 in the root data structure. Furthermore, it would make sense that each of those pointers point to other items with three-pointer headers, just like this one. One other thing you have learned here is that offset +10 is the index of the cached element—the same element pointed to by the third pointer, at offset +c. The difference is that +c is a pointer to memory, and offset +10 is an index into the table, which is equivalent to an element number.

To me, this is the thrill of reversing—one by one gathering pieces of evidence and bringing them together to form partial explanations that slowly evolve into a full understanding of the code. In this particular case, we've made progress in what is undoubtedly the most important piece of the puzzle: the generic table data structure.

Logic and Structure

There is one key element that's been quietly overlooked in all of this: What is the structure of this function? Sure, you can treat all of those conditional and unconditional jumps as a bunch of `goto` instructions and still get away with understanding the flow of relatively simple code. On the other hand, what happens when there are too many of these jumps to the point where it gets hard to keep track of all of them? You need to start thinking the code's logic and structure, and the natural place to start is by trying to logically place all of these conditional and unconditional jumps. Remember, the assembly language code you're reversing was generated by a compiler, and the original code was probably written in C. In all likelihood all of this logic originated in neatly organized `if-else` statements. How do you reconstruct this layout?

Let's start with the first interesting conditional jump in Listing 5.2—the `JE` that goes to `ntdll.7C962554` (I'm ignoring the first two conditions that jump to `ntdll.7C962559` because we've already discussed those). How would you conditionally skip over so much code in a high-level language? Simple, the condition tested in the assembly language code is the opposite of what was

tested in the source code. That's because the processor needs to know whether to *skip* code, and high-level languages have a different perspective—which terms must be satisfied in order to *enter* a certain conditional block. In this case, the test of whether ESI equals EBX must have been originally stated as `if (ESI != EBX)`, and there was a very large chunk of code within those curly braces. The address to which JE is jumping is simply the code that comes right after the end of that conditional block.

It is important to realize that, according to this theory, every line between that JE and the address to which it jumps resides in a conditional block, so any additional conditions after this can be considered nested logic.

Let's take this logical analysis approach a bit further. The conditional jump that immediately follows the JE tests the same two registers, ESI and EBX, and jumps to ntdll.7C96252B if ESI ≤ EBX. Again, we're working under the assumption that the condition is reversed (a detailed discussion of when conditions are reversed and when they're not can be found in Appendix A). This means that the original condition in the source code must have been (ESI > EBX). If it isn't satisfied, the jump is taken, and the conditional block is skipped.

One important thing to notice about this particular condition is the unconditional JMP that comes right before ntdll.7C96252B. This means that ntdll.7C96252B is a chunk of code that *wouldn't* be executed if the conditional block is executed. This means that ntdll.7C96252B is only executed when the high-level conditional block is skipped. Why is that? When you think about it, this is a most popular high-level language programming construct: It is simply an if-else statement. The else block starts at ntdll.7C96252B, which is why there is an unconditional jump after the if block—we only want one of these blocks to run, not both.

Whenever you find a conditional jump that skips a code block that ends with a forward-pointing unconditional JMP, you're probably looking at an if-else block. The block being skipped is the if block, and the code after the unconditional JMP is the else block. The end of the else block is marked by the target address of the unconditional JMP.

For more information on compiler-generated logic please refer to Appendix A.

Let's now proceed to investigate the code chunk we were looking at earlier before we examined the code at ntdll.7C962554. Remember that we were at a condition that compared ESI (which is the index from offset +10) against EBX (which is apparently the index of the element we are trying to get). There were two conditional jumps. The first one (which has already been examined) is taken if the operands are equal, and the second goes to ntdll.7C96252B if ESI ≤ EBX. We'll go back to this conditional section later on. It's important to

realize that the code that follows these two jumps is only executed if `ESI > EBX`, because we've already tested and conditionally jumped if `ESI == EBX` or if `ESI < EBX`.

When none of the branches are taken, the code copies `ESI` into `EDX` and shifts it by one binary position to the right. Binary shifting is a common way to divide or multiply numbers by powers of two. Shifting integer x to the left by n bits is equivalent to $x \times 2^n$ and shifting right by n bits is equivalent to $x/2n$. In this case, right shifting `EDX` by one means $EDX/2^1$, or `EDX/2`. For more information on how to decipher arithmetic sequences refer to Appendix B.

Let's proceed to compare `EDX` (which now contains `ESI/2`) with `EBX` (which is the incremented index of the element we're after), and jump to `ntdll.7C96251B` if `EBX ≤ EDX`. Again, the comparison uses `JBE`, which assumes unsigned operands, so it's pretty safe to assume that table indexes are defined as unsigned integers. Let's ignore the conditional branch for a moment and proceed to the code that follows, as if the branch is not taken.

Here `EBX` is subtracted from `ESI` and the result is stored in `ESI`. The following instruction might be a bit confusing. You can see a `JE` (which is jump if equal) after the subtraction because subtraction and comparison are the same thing, except that in a comparison the result of the subtraction is discarded, and only the flags are kept. This `JE` branch will be taken if `EBX == ESI` before the subtraction or if `ESI == 0` after the subtraction (which are two different ways of looking at what is essentially the same thing). Notice that this exposes a redundancy in the code—you've already compared `EBX` against `ESI` earlier and exited the function if they were equal (remember the jump to `ntdll.7C962554`?), so `ESI` couldn't possibly be zero here. The programmer who wrote this code apparently had a pretty good reason to double-check that the code that follows this check is never reached when `ESI == EBX`. Let's now see why that is so.

Search Loop 1

At this point, you have completed the analysis of the code section starting at `ntdll.7C962501` and ending at `ntdll.7c962511`. The next sequence appears to be some kind of loop. Let's take a look at the code and try and figure out what it does.

```
7C962513    DEC ESI
7C962514    MOV EAX,DWORD PTR [EAX+4]
7C962517    JNZ SHORT ntdll.7C962513
7C962519    JMP SHORT ntdll.7C96254E
```

As I've mentioned, the first thing to notice about these instructions is that they form a loop. The `JNZ` will keep on jumping back to `ntdll.7C962513`

(which is the beginning of the loop) for as long as `ESI != 0`. What does this loop do? Remember that `EAX` is the third pointer from the three-pointer group in the root data structure, and that you're currently working under the assumption that each element starts with the same three-pointer structure. This loop really supports that assumption, because it takes offset +4 in what we believe is some element from the list and treats it as another pointer. Not definite proof, but substantial evidence that +4 is the second in a series of three pointers that precede each element in a generic table.

Apparently the earlier subtraction of `EBX` from `ESI` provided the exact number of elements you need to traverse in order to get from `EAX` to the element you are looking for (remember, you already know `ESI` is the index of the element pointed to by `EAX`). The question now is, in which direction are you moving relative to `EAX`? Are you going toward lower-indexed elements or higher-indexed elements? The answer is simple, because you've already compared `ESI` with `EBX` and branched out for cases where `ESI ≤ EBX`, so you know that in this particular case `ESI > EBX`. This tells you that by taking each element's offset +4 you are moving toward the lower-indexed elements in the table.

Recall that earlier I mentioned that the programmer must have really wanted to double-check cases where `ESI < EBX`? This loop clarifies that issue. If you ever got into this loop in a case where `ESI ≤ EBX`, `ESI` would immediately become a negative number because it is decremented at the very beginning. This would cause the loop to run unchecked until it either ran into an invalid pointer and crashed or (if the elements point back to each other in a loop) until `ESI` went back to zero again. In a 32-bit machine this would take 4,294,967,296 iterations, which may sound like a lot, but today's high-speed processors might actually complete this many iterations so quickly that if it happened rarely the programmer might actually miss it! This is why from a programmer's perspective crashing the program is sometimes better than letting it keep on running with the problem—it simplifies the program's stabilization process.

When our loop ends the code takes an unconditional jump to `ntdll.7C96254E`. Let's see what happens there.

```
7C96254E    MOV DWORD PTR [ECX+C],EAX
7C962551    MOV DWORD PTR [ECX+10],EBX
```

Well, very interesting indeed. Here, you can get a clear view on what offsets +C and +10 in the root data structure contain. It appears that this is some kind of an optimization for quickly searching and traversing the table. Offset +C receives the pointer to the element you've been looking for (the one you've reached by going through the loop), and offset +10 receives that element's index. Clearly the reason this is done is so that repeated calls to this function

(and possibly to other functions that traverse the list) would require as few iterations as possible. This code then proceeds into `ntdll.7C962554`, which you've already looked at. `ntdll.7C962554` skips the element's header by adding 12 and returns that pointer to the caller.

You've now established the basics of how this function works, and a little bit about how a generic table is laid out. Let's proceed with the other major cases that were skipped over earlier.

Let's start with the case where the condition `ESI < EBX` is satisfied (the actual check is for `ESI ≤ EBX`, but you could never be here if `ESI == EBX`). Here is the code that executes in this case.

```
7C96252B    MOV EDI,EBX
7C96252D    SUB EDX,EBX
7C96252F    SUB EDI,ESI
7C962531    INC EDX
7C962532    CMP EDI,EDX
7C962534    JA SHORT ntdll.7C962541
7C962536    TEST EDI,EDI
7C962538    JE SHORT ntdll.7C96254E
```

This code performs `EDX = (Table->TotalElements – ElementToGet + 1) + 1` and `EDI = ElementToGet + 1 – LastIndexFound`. In plain English, `EDX` now has the distance (in elements) from the element you're looking for to the end of the list, and `EDI` has the distance from the element you're looking for to the last index found.

Search Loop 2

Having calculated the two distances above, you now reach an important junction in which you enter one of two search loops. Let's start by looking at the first conditional branch that jumps to `ntdll.7C962541` if `EDI > EDX`.

```
7C962541    TEST EDX,EDX
7C962543    LEA EAX,DWORD PTR [ECX+4]
7C962546    JE SHORT ntdll.7C96254E
7C962548    DEC EDX
7C962549    MOV EAX,DWORD PTR [EAX+4]
7C96254C    JNZ SHORT ntdll.7C962548
```

This snippet checks that `EDX != 0`, and starts looping on elements starting with the element pointed by offset +4 of the root table data structure. Like the previous loop you've seen, this loop also traverses the elements using offset +4 in each element. The difference with this loop is the starting pointer. The previous loop you saw started with offset + c in the root data structure, which is a

pointer to the last element found. This loop starts with offset +4. Which element does offset +4 point to? How can you tell? There is one hint available.

Let's see how many elements this loop traverses, and how you get to that number. The number of iterations is stored in `EDX`, which you got by calculating the distance between the last element in the table and the element that you're looking for. This loop takes you the distance between the end of the list and the element you're looking for. This means that offset +4 in the root structure points to the last element in the list! By taking offset +4 in each element you are going backward in the list toward the beginning. This makes sense, because in the previous loop (the one at `ntdll.7C962513)` you established that taking each element's offset +4 takes you "backward" in the list, toward the lowered-indexed elements. This loop does the same thing, except that it starts from the very end of the list. All `RtlGetElementGenericTable` is doing is it's trying to find the right element in the lowest possible number of iterations.

By the time `EDX` gets to zero, you know that you've found the element. The code then flows into `ntdll.7C96254E`, which you've examined before. This is the code that caches the element you've found into offsets +c and +10 of the root data structure. This code flows right into the area in the function that returns the pointer to our element's data to the caller.

What happens when (in the previous sequence) `EDI == 0`, and the jump to `ntdll.7C96254E` is taken? This simply skips the loop and goes straight to the caching of the found element, followed by returning it to the caller. In this case, the function returns the previously found element—the one whose pointer is cached in offset +c of the root data structure.

Search Loop 3

If neither of the previous two branches is taken, you know that `EDI < EDX` (because you've examined all other possible options). In this case, you know that you must move forward in the list (toward higher-indexed elements) in order to get from the cached element in offset +c to the element you are looking for. Here is the forward-searching loop:

```
7C962513    DEC ESI
7C962514    MOV EAX,DWORD PTR [EAX+4]
7C962517    JNZ SHORT ntdll.7C962513
7C962519    JMP SHORT ntdll.7C96254E
```

The most important thing to notice about this loop is that it is using a different pointer in the element's header. The backward-searching loops you encountered earlier were both using offset +4 in the element's header, and this one is using offset +0. That's really an easy one—this is clearly a linked list of some sort, where offset +0 stores the `NextElement` pointer and offset +4 stores the `PrevElement` pointer. Also, this loop is using `EDI` as the counter,

and EDI contains the distance between the cached element and the element that you're looking for.

Search Loop 4

There is one other significant search case that hasn't been covered yet. Remember how before we got into the first backward-searching loop we tested for a case where the index was lower than LastIndexFound / 2? Let's see what the function does when we get there:

```
7C96251B    TEST EBX,EBX
7C96251D    LEA EAX,DWORD PTR [ECX+4]
7C962520    JE SHORT ntdll.7C96254E
7C962522    MOV EDX,EBX
7C962524    DEC EDX
7C962525    MOV EAX,DWORD PTR [EAX]
7C962527    JNZ SHORT ntdll.7C962524
7C962529    JMP SHORT ntdll.7C96254E
```

This sequence starts with the element at offset +4 in the root data structure, which is the one we've previously defined as the last element in the list. It then starts looping on elements using offset +0 in each element's header. Offset +0 has just been established as the element's NextElement pointer, so what's going on? How could we possibly be going forward from the last element in the list? It seems that we must revise our definition of offset +4 in the root data structure a little bit. It is not really the last element in the list, but it is the head of a *circular linked list*. The term circular means that the NextElement pointer in the last element of the list points back to the beginning and that the PrevElement pointer in the first element points to the last element.

Because in this case the index is lower than LastIndexFound / 2, it would just be inefficient to start our search from the last element found. Instead, we start the search from the first element in the list and move forward until we find the right element.

Reconstructing the Source Code

This concludes the detailed analysis of RtlGetElementGenericTable. It is not a trivial function, and it includes several slightly confusing control flow constructs and some data structure manipulation. Just to demonstrate the power of reversing and just how accurate the analysis is, I've attempted to reconstruct the source code of that function, along with a tentative declaration of what must be inside the TABLE data structure. Listing 5.3 shows what you currently know about the TABLE data structure. Listing 5.4 contains my reconstructed source code for RtlGetElementGenericTable.

```
struct TABLE
{
  PVOID Unknown1;
  LIST_ENTRY *LLHead;
  LIST_ENTRY *SomeEntry;
  LIST_ENTRY *LastElementFound;
  ULONG LastElementIndex;
  ULONG NumberOfElements;
  ULONG Unknown1;
  ULONG Unknown2;
  ULONG Unknown3;
  ULONG Unknown4;
};
```

Listing 5.3 The contents of the `TABLE` data structure, based on what has been learned so far.

```
PVOID stdcall MyRtlGetElementGenericTable(TABLE *Table, ULONG
ElementToGet)
{
  ULONG TotalElementCount = Table->NumberOfElements;
  LIST_ENTRY *ElementFound = Table->LastElementFound;
  ULONG LastElementFound = Table->LastElementIndex;
  ULONG AdjustedElementToGet = ElementToGet + 1;

  if (ElementToGet == -1 || AdjustedElementToGet > TotalElementCount)
    return 0;

  // If the element is the last element found, we just return it.
  if (AdjustedElementToGet != LastIndexFound)
  {
    // If the element isn't LastElementFound, go search for it:
    if (LastIndexFound > AdjustedElementToGet)
    {
      // The element is located somewhere between the first element and
      // the LastElementIndex. Let's determine which direction would
      // get us there the fastest.
      ULONG HalfWayFromLastFound = LastIndexFound / 2;
      if (AdjustedElementToGet > HalfWayFromLastFound)
      {
        // We start at LastElementFound (because we're closer to it) and
        // move backward toward the beginning of the list.
        ULONG ElementsToGo = LastIndexFound - AdjustedElementToGet;

        while(ElementsToGo--)
          ElementFound = ElementFound->Blink;
```

Listing 5.4 A source-code level reconstruction of RtlGetElementGenericTable.

```
      }
      else
      {
        // We start at the beginning of the list and move forward:
        ULONG ElementsToGo = AdjustedElementToGet;
        ElementFound = (LIST_ENTRY *) &Table->LLHead;

        while(ElementsToGo--)
          ElementFound = ElementFound->Flink;
      }
    }
    else
    {
      // The element has a higher index than LastElementIndex. Let's see
      // if it's closer to the end of the list or to LastElementIndex:
      ULONG ElementsToLastFound = AdjustedElementToGet - LastIndexFound;
      ULONG ElementsToEnd = TotalElementCount - AdjustedElementToGet+ 1;

      if (ElementsToLastFound <= ElementsToEnd)
      {
        // The element is closer (or at the same distance) to the last
        // element found than to the end of the list. We traverse the
        // list forward starting at LastElementFound.
        while (ElementsToLastFound--)
          ElementFound = ElementFound->Flink;
      }
      else
      {
        // The element is closer to the end of the list than to the last
        // element found. We start at the head pointer and traverse the
        // list backward.
        ElementFound = (LIST_ENTRY *) &Table->LLHead;
        while (ElementsToEnd--)
          ElementFound = ElementFound->Blink;
      }
    }

    // Cache the element for next time.
    Table->LastElementFound = ElementFound;
    Table->LastElementIndex = AdjustedElementToGet;
  }

  // Skip the header and return the element.
  // Note that we don't have a full definition for the element struct
  // yet, so I'm just incrementing by 3 ULONGs.
  return (PVOID) ((PULONG) ElementFound + 3);
}
```

Listing 5.4 *(continued)*

It's quite amazing to think that with a few clever deductions and a solid understanding of assembly language you can convert those two pages of assembly language code to the function in Listing 5.4. This function does everything the disassembled code does at the same order and implements the exact same logic.

If you're wondering just how close my approximation is to the original source code, here's something to consider: If compiled using the right compiler version and the right set of flags, the preceding source code will produce the *exact* same binary code as the function we disassembled earlier from `NTDLL`, byte for byte. The compiler in question is the one shipped with Microsoft Visual C++ .NET 2003—*Microsoft 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86*.

If you'd like to try this out for yourself, keep in mind that Windows is not built using the compiler's default settings. The following are the optimization and code generation flags I used in order to get binary code that was identical to the one in `NTDLL`. The four optimization flags are: `/Ox` for enabling maximum optimizations, `/Og` for enabling global optimizations, `/Os` for favoring code size (as opposed to code speed), and `/Oy-` for ensuring the use of frame pointers. I also had `/GA` enabled, which optimizes the code specifically for Windows applications.

Standard reversing practices rarely require such a highly accurate reconstruction of a function's source code. Simply figuring out the basic data structures and the generally idea of the logic that takes place in the function is enough for most purposes. Determining the exact compiler version and compiler flags in order to produce the exact same binary code as the one we started with is a nice exercise, but it has limited practical value for most purposes.

Whew! You've just completed your first attempt at reversing a fairly complicated and involved function. If you've never attempted reversing before, don't worry if you missed parts of this session—it'll be easier to go back to this function once you develop a full understanding of the data structures. In my opinion, reading through such a long reversing session can often be much more productive when you already know the general idea of what the code does and how data is laid out.

RtlInsertElementGenericTable

Let's proceed to see how an element is added to the table by looking at `RtlInsertElementGenericTable`. Listing 5.5 contains the disassembly of `RtlInsertElementGenericTable`.

```
7C924DC0    PUSH EBP
7C924DC1    MOV EBP,ESP
7C924DC3    PUSH EDI
7C924DC4    MOV EDI,DWORD PTR [EBP+8]
7C924DC7    LEA EAX,DWORD PTR [EBP+8]
7C924DCA    PUSH EAX
7C924DCB    PUSH DWORD PTR [EBP+C]
7C924DCE    CALL ntdll.7C92147B
7C924DD3    PUSH EAX
7C924DD4    PUSH DWORD PTR [EBP+8]
7C924DD7    PUSH DWORD PTR [EBP+14]
7C924DDA    PUSH DWORD PTR [EBP+10]
7C924DDD    PUSH DWORD PTR [EBP+C]
7C924DE0    PUSH EDI
7C924DE1    CALL ntdll.7C924DF0
7C924DE6    POP EDI
7C924DE7    POP EBP
7C924DE8    RET 10
```

Listing 5.5 A disassembly of RtlInsertElementGenericTable, produced using OllyDbg.

We've already discussed the first two instructions—they create the stack frame. The instruction that follows pushes `EDI` onto the stack. Generally speaking, there are three common scenarios where the `PUSH` instruction is used in a function:

- When saving the value of a register that is about to be used as a local variable by the function. The value is then typically popped out of the stack near the end of the function. This is easy to detect because the value must be popped *into the same register*.
- When pushing a parameter onto the stack before making a function call.
- When copying a value, a `PUSH` instruction is sometimes immediately followed by a `POP` that loads that value into some other register. This is a fairly unusual sequence, but some compilers generate it from time to time.

In the function we must try and figure out whether `EDI` is being pushed as the last parameter of `ntdll.7C92147B`, which is called right afterward, or if it is a register whose value is being saved. Because you can see that `EDI` is overwritten with a new value immediately after the `PUSH`, and you can also see that it's popped back from the stack at the very end of the function, you know that the compiler is just saving the value of `EDI` in order to be able to use that register as a local variable within the function.

The next two instructions in the function are somewhat interesting.

```
7C924DC4    MOV EDI,DWORD PTR [EBP+8]
7C924DC7    LEA EAX,DWORD PTR [EBP+8]
```

The first line loads the value of the first parameter passed into the function (we've already established that `[ebp+8]` is the address of the first parameter in a function) into the local variable, `EDI`. The second loads the *pointer* to the first parameter into `EAX`. Notice that difference between the `MOV` and `LEA` instructions in this sequence. `MOV` actually goes to memory and retrieves the value pointed to by `[ebp+8]` while `LEA` simply calculates `EBP + 8` and loads that number into `EAX`.

One question that quickly arises is whether `EAX` is another local variable, just like `EDI`. In order to answer that, let's examine the code that immediately follows.

```
7C924DCA    PUSH EAX
7C924DCB    PUSH DWORD PTR [EBP+C]
7C924DCE    CALL ntdll.7C92147B
```

You can see that the first parameter pushed onto the stack is the value of `EAX`, which strongly suggests that `EAX` was not assigned for a local variable, but was used as temporary storage by the compiler because two instructions were needed into order to push the pointer of the first parameter onto the stack. This is a very common limitation in assembly language: Most instructions aren't capable of receiving complex arguments like `LEA` and `MOV` can. Because of this, the compiler must use `MOV` or `LEA` and store their output into a register and then use that register in the instruction that follows.

To go back to the code, you can quickly see that there is a function, `ntdll.7C92147B`, that takes two parameters. Remember that in the `stdcall` calling convention (which is the convention used by most Windows code) parameters are always pushed onto the stack in the reverse order, so the first `PUSH` instruction (the one that pushes `EAX`) is really pushing the second parameter. The first parameter that `ntdll.7C92147B` receives is `[ebp+C]`, which is the second parameter that was passed to `RtlInsertElementGenericTable`.

RtlLocateNodeGenericTable

Let's now follow the function call made from `RtlInsertElementGenericTable` into `ntdll.7C92147B` and analyze that function, which I have tentatively titled `RtlLocateNodeGenericTable`. The full disassembly of that function is presented in Listing 5.6.

```
7C92147B    MOV EDI,EDI
7C92147D    PUSH EBP
7C92147E    MOV EBP,ESP
7C921480    PUSH ESI
7C921481    MOV ESI,DWORD PTR [EDI]
7C921483    TEST ESI,ESI
7C921485    JE ntdll.7C924E8C
7C92148B    LEA EAX,DWORD PTR [ESI+18]
7C92148E    PUSH EAX
7C92148F    PUSH DWORD PTR [EBP+8]
7C921492    PUSH EDI
7C921493    CALL DWORD PTR [EDI+18]
7C921496    TEST EAX,EAX
7C921498    JE ntdll.7C924F14
7C92149E    CMP EAX,1
7C9214A1    JNZ SHORT ntdll.7C9214BB
7C9214A3    MOV EAX,DWORD PTR [ESI+8]
7C9214A6    TEST EAX,EAX
7C9214A8    JNZ ntdll.7C924F22
7C9214AE    PUSH 3
7C9214B0    POP EAX
7C9214B1    MOV ECX,DWORD PTR [EBP+C]
7C9214B4    MOV DWORD PTR [ECX],ESI
7C9214B6    POP ESI
7C9214B7    POP EBP
7C9214B8    RET 8
7C9214BB    XOR EAX,EAX
7C9214BD    INC EAX
7C9214BE    JMP SHORT ntdll.7C9214B1
```

Listing 5.6 Disassembly of the internal, nonexported function at `ntdll.7C92147B`.

Before even beginning to reverse this function, there are a couple of slight oddities about the very first few lines in Listing 5.6 that must be considered. Notice the first line: `MOV EDI, EDI`. It does nothing! It is essentially dead code that was put in place by the compiler as a placeholder, in case someone wanted to *trap* this function. Trapping means that some external component adds a `JMP` instruction that is used as a notification whenever the trapped function is called. By placing this instruction at the beginning of every function, Microsoft essentially set an infrastructure for trapping functions inside `NTDLL`. Note that these placeholders are only implemented in more recent versions of Windows (in Windows XP, they were introduced in Service Pack 2), so you may or may not see them on your system.

The next few lines also exhibit a peculiarity. After setting up the traditional stack frame, the function is reading a value from `EDI`, even though that register has not been accessed in this function up to this point. Isn't `EDI`'s value just going to be random at this point?

If you look at `RtlInsertElementGenericTable` again (in Listing 5.5), it seems that the value of the first parameter passed to that function (which is probably the address of the root `TABLE` data structure) is loaded into `EDI` before the function from Listing 5.6 is called. This implies that the compiler is simply using `EDI` in order to directly pass that pointer into `RtlLocateNode GenericTable`, but the question is which calling convention passes parameters through `EDI`? The answer is that no standard calling convention does that, but the compiler has chosen to do this anyway. This indicates that the compiler controls all *points of entry* into this function.

Generally speaking, when a function is defined within an object file, the compiler has no way of knowing what its scope is going to be. It might be exported by the linker and called by other modules, or it might be internal to the executable but called from other object files. In any case, the compiler must honor the specified calling convention in order to ensure compatibility with those unknown callers. The only exception to this rule occurs when a function is explicitly defined as local to the current object file using the `static` keyword. This informs the compiler that only functions within the current source file may call the function, which allows the compiler to give such static functions nonstandard interfaces that might be more efficient.

In this particular case, the compiler is taking advantage of the `static` keyword by avoiding stack usage as much as possible and simply passing some of the parameters through registers. This is possible because the compiler is taking advantage of having full control of register allocation in both the caller and the callee.

Judging by the number of bytes passed on the stack (8 from looking at the `RET` instruction), and by the fact that `EDI` is being used without ever being initialized, we can safely assume that this function takes three parameters. Their order is unknown to us because of that register, but judging from the previous functions we can safely assume that the root data structure is always passed as the first parameter. As I said, `RtlInsertElementGenericTable` loads `EDI` with the value of the first parameter passed on to it, so we pretty much know that `EDI` contains our root data structure.

Let's now proceed to examine the first lines of the actual body of this function.

```
7C921481    MOV ESI,DWORD PTR [EDI]
7C921483    TEST ESI,ESI
7C921485    JE ntdll.7C924E8C
```

In this snippet, you can quickly see that `EDI` is being treated as a pointer to something, which supports the assumption about its being the table data structure. In this case, the first member (offset +0) is being tested for zero (remember that you're reversing the conditions), and the function jumps to `ntdll.7C924E8C` if that condition is satisfied.

You might have noticed an interesting fact: the address `ntdll.7C924E8C` is *far away* from the address of the current code you're looking at! In fact, that code was not even included in Listing 5.6—it resides in an entirely separate region in the executable file. How can that be—why would a function be scattered throughout the module like that? The reason this is done has to do with some Windows memory management issues.

Remember we talked about working sets in Chapter 3? While building executable modules, one of the primary concerns is to arrange the module in a way that would allow the module to consume as little physical memory as possible while it is loaded into memory. Because Windows only allocates physical memory to areas that are in active use, this module (and pretty much every other component in Windows) is arranged in a special layout where popular code sections are placed at the beginning of the module, while more esoteric code sequences that are rarely executed are pushed toward the end. This process is called *working-set tuning*, and is discussed in detail in Appendix A.

For now just try to think of what you can learn from the fact that this conditional block has been relocated and sent to a higher memory address. It most likely means that this conditional block is *rarely executed*! Granted, there are various reasons why a certain conditional block would rarely be executed, but there is one primary explanation that is probably true for 90 percent of such conditional blocks: the block implements some sort of error-handling code. Error-handling code is a typical case in which conditional statements are created that are rarely, if ever, actually executed.

Let's now proceed to examine the code at `ntdll.7C924E8C` and see if it is indeed an error-handling statement.

```
7C924E8C    XOR EAX,EAX
7C924E8E    JMP ntdll.7C9214B6
```

As expected, all this sequence does is set `EAX` to zero and jump back to the function's epilogue. Again, this is not definite, but all evidence indicates that this is an error condition.

At this point, you can proceed to the code that follows the conditional statement at `ntdll.7C92148B`, which is clearly the body of the function.

The Callback

The body of `RtlLocateNodeGenericTable` performs a somewhat unusual function call that appears to be the focal point of this entire function. Let's take a look at that code.

```
7C92148B    LEA EAX,DWORD PTR [ESI+18]
7C92148E    PUSH EAX
7C92148F    PUSH DWORD PTR [EBP+8]
7C921492    PUSH EDI
7C921493    CALL DWORD PTR [EDI+18]
```

```
7C921496    TEST EAX,EAX
7C921498    JE ntdll.7C924F14
7C92149E    CMP EAX,1
7C9214A1    JNZ SHORT ntdll.7C9214BB
```

This snippet does something interesting that you haven't encountered so far. It is obvious that the first five instructions are all part of the same function call sequence, but notice the address that is being called. It is not a hard-coded address as usual, but rather the value at offset +18 in `EDI`. This exposes another member in the root table data structure at offset +18 as a callback function of some sort. If you go back to `RtlInitializeGenericTable`, you'll see that that offset +18 was loaded from the second parameter passed to that function. This means that offset +18 contains some kind of a user-defined callback.

The function seems to take three parameters, the first being the table data structure; the second, the second parameter passed to the current function; and the third, `ESI + 18`. Remember that `ESI` was loaded earlier with the value at offset +0 of the root structure. This indicates that offset +0 contains some other data structure and that the callback is getting a pointer to offset +18 at this structure. You don't really know what this data structure is at this point.

Once the callback function returns, you can test its return value and jump to `ntdll.7C924F14` if it is zero. Again, that address is outside of the main body of the function. Another error handling code? Let's find out. The following is the code snippet found at `ntdll.7C924F14`.

```
7C924F14    MOV EAX,DWORD PTR [ESI+4]
7C924F17    TEST EAX,EAX
7C924F19    JNZ SHORT ntdll.7C924F22
7C924F1B    PUSH 2
7C924F1D    JMP ntdll.7C9214B0
7C924F22    MOV ESI,EAX
7C924F24    JMP ntdll.7C92148B
```

This snippet loads offset +4 from the unknown structure in `ESI` and tests if it is zero. If it is nonzero, the code jumps to `ntdll.7C924F22`, a two-line segment that jumps back to `ntdll.7C92148B` (which is back inside the main body of our function), but not before it loads `ESI` with the value from offset +4 in the unknown data structure (which is currently stored in `EAX`). If offset +4 at the unknown structure is zero, the code pushes the number 2 onto the stack and jumps back into `ntdll.7C9214B0`, which is another address at the main body of `RtlLocateNodeGenericTable`.

It is important at this point to keep track of the various branches you've encountered in the code so far. This is a bit more confusing than it could have been because of the way the function is scattered throughout the module. Essentially, the test for offset +4 at the unknown structure has one of two outcomes. If the value is zero the function returns to the caller (`ntdll.7C9214B0` is near the

very end of the function). If there is a nonzero value at that offset, the code loads that value into ESI and jumps back to ntdll.7C92148B, which is the callback calling code you just examined.

It looks like you're looking at a loop that constantly calls into the callback and traverses some kind of linked list that starts at offset +0 of the root data structure. Each item seems to be at least 0x1c bytes long, because offset +18 of that structure is passed as the last parameter in the callback.

Let's see what happens when the callback returns a nonzero value.

```
7C92149E    CMP EAX,1
7C9214A1    JNZ SHORT ntdll.7C9214BB
7C9214A3    MOV EAX,DWORD PTR [ESI+8]
7C9214A6    TEST EAX,EAX
7C9214A8    JNZ ntdll.7C924F22
7C9214AE    PUSH 3
7C9214B0    POP EAX
7C9214B1    MOV ECX,DWORD PTR [EBP+C]
7C9214B4    MOV DWORD PTR [ECX],ESI
7C9214B6    POP ESI
7C9214B7    POP EBP
7C9214B8    RET 8
```

First of all, it seems that the callback returns some kind of a number and not a pointer. This could be a Boolean, but you don't know for sure yet. The first check tests for ReturnValue != 1 and loads offset +8 into EAX if that condition is not satisfied. Offset +8 in ESI is then tested for a nonzero value, and if it is zero the code sets EAX to 3 (using the PUSH-POP method described earlier), and proceeds to what is clearly this function's epilogue. At this point, it becomes clear that the reason for loading the value 3 into EAX was to return the value 3 to the caller. Notice how the second parameter is treated as a pointer, and that this pointer receives the current value of ESI, which is that unknown structure we discussed. This is important because it seems that this function is traversing a different list than the one you've encountered so far. Apparently, there is some kind of a linked list that starts at offset +0 in the root table data structure.

So far you've seen what happens when the callback returns 0 or when it returns 1. When the callback returns some other value, the conditional jump you looked at earlier is taken and execution continues at ntdll.7C9214BB. Here is the code at that address:

```
7C9214BB    XOR EAX,EAX
7C9214BD    INC EAX
7C9214BE    JMP SHORT ntdll.7C9214B1
```

This snippet sets EAX to 1 and jumps back into ntdll.7C9214B1, that you've just examined. Recall that that sequence doesn't affect EAX, so it is effectively returning 1 to the caller.

If you go back to the code that immediately follows the invocation of the callback, you can see that when the check for `ESI` offset +8 finds a nonzero value, the code jumps to `ntdll.7C924F22`, which is an address you've already looked at. This is the code that loads `ESI` from `EAX` and jumps back to the beginning of the loop.

At this point, you have gathered enough information to make some educated guesses on this function. This function loops on code that calls some callback and acts differently based on the return value received. The callback function receives items in what appears to be some kind of a linked list. The first item in that list is accessed through offset +0 in the root data structure.

The continuation of the loop and the direction in which it goes depend on the callback's return value.

1. If the callback returns 0, the loop continues on offset +4 in the current item. If offset +4 contains zero, the function returns 2.
2. If the callback returns 1, the function loads the next item from offset +8 in the current item. If offset +8 contains zero the function returns 3. When offset +8 is non-`NULL`, the function continues looping on offset +4 starting with the new item.
3. If the callback returns any other value, the loop terminates and the current item is returned. The return value is 1.

High-Level Theories

It is useful to take a little break from all of these bits, bytes, and branches, and look at the big picture. What are we seeing here, what does this function do? It's hard to tell at this point, but the repeated callback calls and the direction changes based on the callback return values indicate that the callback might be used for determining the relative position of an element within the list. This is probably defined as an element comparison callback that receives two elements and compares them. The three return values probably indicate *smaller than*, *larger than*, or *equal*.

It's hard to tell at this point which return value means what. If we were to draw on our previous conclusions regarding the arrangement of next and previous pointers we see that the next pointer comes first and is followed by the previous pointer. Based on that arrangement we can make the following guesses:

- A return value of 0 from the callback means that the new element is higher valued than the current element and that we need to move forward in the list.
- A return value of 1 would indicate that the new element is lower valued than the current element and that we need to move backward in the list.

- Any value other than 1 or 0 indicates that the new element is identical to one already in the list and that it shouldn't be added.

You've made good progress, but there are several pieces that just don't seem to fit in. For instance, assuming that offsets +4 and +8 in the new unknown structure do indeed point to a linked list, what is the point of looping on offset +4 (which is supposedly the next pointer), and then when finding a lower-valued element to take one element from offset +8 (supposedly the `prev` pointer) only to keep looping on offset +4? If this were a linked list, this would mean that if you found a lower-valued element you'd go back one element, and then keep moving forward. It's not clear how such a sequence could be useful, which suggests that this just isn't a linked list. More likely, this is a tree structure of some sort, where offset +4 points to one side of the tree (let's assume it's the one with higher-valued elements), and offset +8 points to the other side.

The beauty of this tree theory is that it would explain why the loop would take offset +8 from the current element and then keep looping on offset +4. Assuming that offset +4 does indeed point to the right node and that offset +8 points to the left node, it makes total sense. The function is looping toward higher-valued elements by constantly moving to the next node on the right until it finds a node whose middle element is higher-valued than the element you're looking for (which would indicate that the element is somewhere in the left node). Whenever that happens the function moves to the left node and then continues to move to the right from there until the element is found. This is the classic *binary search algorithm* defined in Donald E. Knuth. *The Art of Computer Programming - Volume 3: Sorting and Searching (Second Edition).* Addison Wesley. [Knuth3]. Of course, this function is probably not searching for an existing element, but is rather looking for a place to fit the new element.

Callback Parameters

Let's take another look at the parameters passed to the callback and try to guess their meaning. We already know what the first parameter is—it is read from `EDI`, which is the root data structure. We also know that the third parameter is the current node in what we *believe* is a binary search, but why is the callback taking offset +18 in that structure? It is likely that +18 is not exactly an offset into a structure, but is rather just the total size of the element's headers. By adding 18 to the element pointer the function is simply skipping these headers and is getting to the actual element data, which is of course implementation-specific.

The second parameter of the callback is taken from the first parameter passed to the function. What could it possible be? Since we think that this function is some kind of an element comparison callback, we can safely assume that the second parameter points to the new element. It would have to be because if it isn't, what would the comparison callback compare? This means

that the callback takes a TABLE pointer, a pointer to the data of the element being added, and a pointer to the data of the current element. The function is comparing the new element with the data of the element we're currently traversing. Let's try and define a prototype for the callback.

```
typedef int (stdcall * TABLE_COMPARE_ELEMENTS) (
     TABLE   *pTable,
     PVOID    pElement1,
     PVOID    pElement2
     );
```

Summarizing the Findings

Let's try and summarize all that has been learned about RtlLocateNode GenericTable. Because we have a working theory on the parameters passed into it, let's revisit the code in RtlInsertElementGenericTable that called into RtlLocateNodeGenericTable, just to try and use this knowledge to learn something about the parameters that RtlInsertElement GenericTable takes. The following is the sequence that calls RtlLocate NodeGenericTable from RtlInsertElementGenericTable.

```
7C924DC7    LEA EAX,DWORD PTR [EBP+8]
7C924DCA    PUSH EAX
7C924DCB    PUSH DWORD PTR [EBP+C]
7C924DCE    CALL ntdll.7C92147B
```

It looks like the second parameter passed to RtlInsertElementGeneric Table at [ebp+C] is the new element currently being inserted. Because you now know that ntdll.7C92147B (RtlLocateNodeGenericTable) locates a node in the generic table, you can now give it an estimated prototype.

```
int RtlLocateNodeGenericTable (
     TABLE  *pTable,
     PVOID   ElementToLocate,
     NODE  **NodeFound;
     );
```

There are still many open questions regarding the data layout of the generic table. For example, what was that linked list we encountered in RtlGet ElementGenericTable and how is it related to the binary tree structure we've found?

RtlRealInsertElementWorker

After ntdll.7C92147B returns, RtlInsertElementGenericTable proceeds by calling ntdll.7C924DF0, which is presented in Listing 5.7. You don't have to think much to know that since the previous function only searched for

the right node where to insert the element, surely this function must do the actual insertion into the table.

Before looking at the implementation of the function, let's go back and look at how it's called from `RtlInsertElementGenericTable`. Since you now have some information on some of the data that `RtlInsertElementGenericTable` deals with, you might be able to learn a bit about this function before you even start actually disassembling it. Here's the sequence in `RtlInsertElementGenericTable` that calls the function.

```
7C924DD3    PUSH EAX
7C924DD4    PUSH DWORD PTR [EBP+8]
7C924DD7    PUSH DWORD PTR [EBP+14]
7C924DDA    PUSH DWORD PTR [EBP+10]
7C924DDD    PUSH DWORD PTR [EBP+C]
7C924DE0    PUSH EDI
7C924DE1    CALL ntdll.7C924DF0
```

It appears that `ntdll.7C924DF0` takes six parameters. Let's go over each one and see if we can figure out what it contains.

Argument 6 This snippet starts right after the call to position the new element, so the sixth argument is essentially the return value from `ntdll.7C92147B`, which could either be 1, 2, or 3.

Argument 5 This is the address of the first parameter passed to `RtlInsertElementGenericTable`. However, it no longer contains the value passed to `RtlInsertElementGenericTable` from the caller. It has been used for receiving a binary tree node pointer from the search function. This is essentially the pointer to the node to which the new element will be added.

Argument 4 This is the fourth parameter passed to `RtlInsertElementGenericTable`. You don't currently know what it contains.

Argument 3 This is the third parameter passed to `RtlInsertElementGenericTable`. You don't currently know what it contains.

Argument 2 Based on our previous assessment, the second parameter passed to `RtlInsertElementGenericTable` is the actual element we'll be adding.

Argument 1 `EDI` contains the root table data structure.

Let's try to take all of this information and use it to make a temporary prototype for this function.

```
UNKNOWN RtlRealInsertElementWorker(
      TABLE *pTable,
      PVOID ElementData,
      UNKNOWN Unknown1,
      UNKNOWN Unknown2,
```

```
    NODE *pNode,
    ULONG SearchResult
    );
```

You now have some basic information on `RtlRealInsertElement Worker`. At this point, you're ready to take on the complete listing and try to figure out exactly how it works. The full disassembly of `RtlRealInsert ElementWorker` is presented in Listing 5.7.

```
7C924DF0    MOV EDI,EDI
7C924DF2    PUSH EBP
7C924DF3    MOV EBP,ESP
7C924DF5    CMP DWORD PTR [EBP+1C],1
7C924DF9    PUSH EBX
7C924DFA    PUSH ESI
7C924DFB    PUSH EDI
7C924DFC    JE ntdll.7C935D5D
7C924E02    MOV EDI,DWORD PTR [EBP+10]
7C924E05    MOV ESI,DWORD PTR [EBP+8]
7C924E08    LEA EAX,DWORD PTR [EDI+18]
7C924E0B    PUSH EAX
7C924E0C    PUSH ESI
7C924E0D    CALL DWORD PTR [ESI+1C]
7C924E10    MOV EBX,EAX
7C924E12    TEST EBX,EBX
7C924E14    JE ntdll.7C94D4BE
7C924E1A    AND DWORD PTR [EBX+4],0
7C924E1E    AND DWORD PTR [EBX+8],0
7C924E22    MOV DWORD PTR [EBX],EBX
7C924E24    LEA ECX,DWORD PTR [ESI+4]
7C924E27    MOV EDX,DWORD PTR [ECX+4]
7C924E2A    LEA EAX,DWORD PTR [EBX+C]
7C924E2D    MOV DWORD PTR [EAX],ECX
7C924E2F    MOV DWORD PTR [EAX+4],EDX
7C924E32    MOV DWORD PTR [EDX],EAX
7C924E34    MOV DWORD PTR [ECX+4],EAX
7C924E37    INC DWORD PTR [ESI+14]
7C924E3A    CMP DWORD PTR [EBP+1C],0
7C924E3E    JE SHORT ntdll.7C924E88
7C924E40    CMP DWORD PTR [EBP+1C],2
7C924E44    MOV EAX,DWORD PTR [EBP+18]
7C924E47    JE ntdll.7C924F0C
7C924E4D    MOV DWORD PTR [EAX+8],EBX
7C924E50    MOV DWORD PTR [EBX],EAX
7C924E52    MOV ESI,DWORD PTR [EBP+C]
7C924E55    MOV ECX,EDI
7C924E57    MOV EAX,ECX
```

Listing 5.7 Disassembly of function at `ntdll.7C924DF0`.

```
7C924E59    SHR ECX,2
7C924E5C    LEA EDI,DWORD PTR [EBX+18]
7C924E5F    REP MOVS DWORD PTR ES:[EDI],DWORD PTR [ESI]
7C924E61    MOV ECX,EAX
7C924E63    AND ECX,3
7C924E66    REP MOVS BYTE PTR ES:[EDI],BYTE PTR [ESI]
7C924E68    PUSH EBX
7C924E69    CALL ntdll.RtlSplay
7C924E6E    MOV ECX,DWORD PTR [EBP+8]
7C924E71    MOV DWORD PTR [ECX],EAX
7C924E73    MOV EAX,DWORD PTR [EBP+14]
7C924E76    TEST EAX,EAX
7C924E78    JNZ ntdll.7C935D4F
7C924E7E    LEA EAX,DWORD PTR [EBX+18]
7C924E81    POP EDI
7C924E82    POP ESI
7C924E83    POP EBX
7C924E84    POP EBP
7C924E85    RET 18
7C924E88    MOV DWORD PTR [ESI],EBX
7C924E8A    JMP SHORT ntdll.7C924E52
7C924E8C    XOR EAX,EAX
7C924E8E    JMP ntdll.7C9214B6
```

Listing 5.7 *(continued)*

Like the function at Listing 5.6, this one also starts with that dummy `MOV EDI, EDI` instruction. However, unlike the previous function, this one doesn't seem to receive any parameters through registers, indicating that it was probably not defined using the `static` keyword. This function starts out by checking the value of the `SearchResult` parameter (the last parameter it takes), and making one of those remote, out of function jumps if `SearchResult == 1`. We'll deal with this condition later.

For now, here's the code that gets executed when that condition isn't satisfied.

```
7C924E02    MOV EDI,DWORD PTR [EBP+10]
7C924E05    MOV ESI,DWORD PTR [EBP+8]
7C924E08    LEA EAX,DWORD PTR [EDI+18]
7C924E0B    PUSH EAX
7C924E0C    PUSH ESI
7C924E0D    CALL DWORD PTR [ESI+1C]
```

It seems that the `TABLE` data structure contains another callback pointer. Offset +1c appears to be another callback function that takes two parameters. Let's examine those parameters and try to figure out what the callback does. The first parameter comes from `ESI` and is quite clearly the `TABLE` pointer. What does

the second parameter contain? Essentially, it is the value of the third parameter passed to `RtlRealInsertElementWorker` plus 18 bytes (hex). When you looked earlier at the parameters that `RtlRealInsertElementWorker` takes, you had no idea what the third parameter was, but the number `0x18` sounds somehow familiar. Remember how `RtlLocateNodeGenericTable` added `0x18` (24 in decimal) to the pointer of the current element before it passed it to the `TABLE_COMPARE_ELEMENTS` callback? I suspected that adding 24 bytes was a way of skipping the element's header and getting to the actual data. This corroborates that assumption—it looks like elements in a generic table are each stored with 24-byte headers that are followed by the element's data.

Let's dig further into this function to try and figure out how it works and what the callback does. Here's what happens after the callback returns.

```
7C924E10    MOV EBX,EAX
7C924E12    TEST EBX,EBX
7C924E14    JE ntdll.7C94D4BE
7C924E1A    AND DWORD PTR [EBX+4],0
7C924E1E    AND DWORD PTR [EBX+8],0
7C924E22    MOV DWORD PTR [EBX],EBX
7C924E24    LEA ECX,DWORD PTR [ESI+4]
7C924E27    MOV EDX,DWORD PTR [ECX+4]
7C924E2A    LEA EAX,DWORD PTR [EBX+C]
7C924E2D    MOV DWORD PTR [EAX],ECX
7C924E2F    MOV DWORD PTR [EAX+4],EDX
7C924E32    MOV DWORD PTR [EDX],EAX
7C924E34    MOV DWORD PTR [ECX+4],EAX
7C924E37    INC DWORD PTR [ESI+14]
7C924E3A    CMP DWORD PTR [EBP+1C],0
7C924E3E    JE SHORT ntdll.7C924E88
7C924E40    CMP DWORD PTR [EBP+1C],2
7C924E44    MOV EAX,DWORD PTR [EBP+18]
7C924E47    JE ntdll.7C924F0C
7C924E4D    MOV DWORD PTR [EAX+8],EBX
7C924E50    MOV DWORD PTR [EBX],EAX
```

This code tests the return value from the callback. If it's zero, the function jumps into a remote block. Let's take a quick look at that block.

```
7C94D4BE    MOV EAX,DWORD PTR [EBP+14]
7C94D4C1    TEST EAX,EAX
7C94D4C3    JE SHORT ntdll.7C94D4C7
7C94D4C5    MOV BYTE PTR [EAX],BL
7C94D4C7    XOR EAX,EAX
7C94D4C9    JMP ntdll.7C924E81
```

This appears to be some kind of failure mode that essentially returns 0 to the caller. Notice how this sequence checks whether the fourth parameter at

[ebp+14] is nonzero. If it is, the function is treating it as a pointer, writing a single byte containing 0 (because we *know* EBX is going to be zero at this point) into the address pointed by it. It would appear that the fourth parameter is a pointer to some Boolean that's used for notifying the caller of the function's success or failure.

Let's proceed to look at what happens when the callback returns a non-NULL value. It's not difficult to see that this code is initializing the header of the newly allocated element, using the callback's return value as the address. Before we try to figure out the details of this initialization, let's pause for a second and try to realize what this tells us about the callback function we just observed. It looks as if the purpose of the callback function was to allocate memory for the newly created element. We know this because EBX now contains the return value from the callback, and it's definitely being used as a pointer to a new element that's currently being initialized. With this information, let's try to define this callback.

```
typedef NODE * ( _stdcall * TABLE_ALLOCATE_ELEMENT) (
     TABLE  *pTable,
     ULONG   ElementSize
     );
```

How did I know that the second parameter is the element's size? It's simple. This is a value that was passed along from the caller of RtlInsertElement GenericTable into RtlRealInsertElementWorker, was incremented by 24, and was finally fed into TABLE_ALLOCATE_ELEMENT. Clearly the application calling RtlInsertElementGenericTable is supplying the size of this element, and the function is adding 24 because that's the length of the node's header. Because of this we now also know that the third parameter passed into RtlRealInsertElementWorker is the user-supplied element length. We've also found out that the fourth parameter is an optional pointer into some Boolean that contains the outcome of this function. Let's correct the original prototype.

```
UNKNOWN RtlRealInsertElementWorker(
      TABLE *pTable,
      PVOID ElementData,
      ULONG ElementSize,
      BOOLEAN *pResult OPTIONAL,
      NODE *pNode,
      ULONG SearchResult
      );
```

You may notice that we've been accumulating quite a bit of information on the parameters that RtlInsertElementGenericTable takes. We're now ready to start looking at the prototype for RtlInsertElementGenericTable.

```
UNKNOWN NTAPI RtlInsertElementGenericTable(
     TABLE   *pTable,
     PVOID    ElementData,
     ULONG    DataLength,
     BOOLEAN *pResult OPTIONAL,
     );
```

At this point in the game, you've gained quite a bit of knowledge on this API and associated data structures. There's probably no real need to even try and figure out each and every member in a node's header, but let's look at that code sequence and try and figure out how the new element is linked into the existing data structure.

Linking the Element

First of all, you can see that the function is accessing the element header through `EBX`, and then it loads `EAX` with `EBX + c`, and accesses members through `EAX`. This indicates that there is some kind of a data structure at offset +c of the element's header. Why else would the compiler access these members through another register? Why not just use `EBX` for accessing all the members?

Also, you're now seeing distinct proof that the generic table maintains both a linked list and a tree. `EAX` is loaded with the starting address of the linked list header (`LIST_ENTRY *`), and `EBX` is used for accessing the binary tree members. The function checks the `SearchResult` parameter before the tree node gets attached to the rest of the tree. If it is 0, the code jumps to `ntdll.7C924E88`, which is right after the end of the function's main body. Here is the code for that condition.

```
7C924E88    MOV DWORD PTR [ESI],EBX
7C924E8A    JMP SHORT ntdll.7C924E52
```

In this case, the node is attached as the root of the tree. If `SearchResult` is nonzero, the code proceeds into what is clearly an `if-else` block that is entered when `SearchResult != 2`. If that conditional block is entered (when `SearchResult != 2`), the code takes the `pNode` parameter (which is the node that was found in `RtlLocateNodeGenericTable`), and attaches the newly created node as the left child (offset +8). If `SearchResult == 2`, the code jumps to the following sequence.

```
7C924F0C    MOV DWORD PTR [EAX+4],EBX
7C924F0F    JMP ntdll.7C924E50
```

Here the newly created element is attached as the right child of `pNode` (offset +4). Clearly, the search result indicates whether the new element is smaller or larger than the value represented by `pNode`. Immediately after the '`if-else`'

block a pointer to pNode is stored in offset +0 at the new entry. This indicates that offset +0 in the node header contains a pointer to the parent element. You can now properly define the node header data structure.

```
struct NODE
{
  NODE        *ParentNode;
  NODE        *RightChild;
  NODE        *LeftChild;
  LIST_ENTRY  LLEntry;
  ULONG       Unknown;
};
```

Copying the Element

After allocating the new node and attaching it to pNode, you reach an interesting sequence that is actually quite common and is one that you're probably going to see quite often while reversing IA-32 assembly language code. Let's take a look.

```
7C924E52    MOV ESI,DWORD PTR [EBP+C]
7C924E55    MOV ECX,EDI
7C924E57    MOV EAX,ECX
7C924E59    SHR ECX,2
7C924E5C    LEA EDI,DWORD PTR [EBX+18]
7C924E5F    REP MOVS DWORD PTR ES:[EDI],DWORD PTR [ESI]
7C924E61    MOV ECX,EAX
7C924E63    AND ECX,3
7C924E66    REP MOVS BYTE PTR ES:[EDI],BYTE PTR [ESI]
```

This code loads ESI with ElementData, EDI with the end of the new node's header, ECX with ElementSize * 4, and starts copying the element data, 4 bytes at a time. Notice that there are two copying sequences. The first is for 4-byte chunks, and the second checks whether there are any bytes left to be copied, and copies those (notice how the first MOVS takes DWORD PTR arguments and the second takes BYTE PTR operands).

I say that this is a common sequence because this is a classic memcpy implementation. In fact, it is very likely that the source code contained a memcpy call and that the compiler simply implemented it as an intrinsic function (intrinsic functions are briefly discussed in Chapter 7).

Splaying the Table

Let's proceed to the next code sequence. Notice that there are two different paths that could have gotten us to this point. One is through the path I have just covered in which the callback is called and the structure is initialized, and

the other is taken when `SearchResult == 1` at that first branch in the beginning of the function (at `ntdll.7C924DFC`). Notice that this branch doesn't go straight to where we are now—it goes through a relocated block at `ntdll.7C935D5D`. Regardless of how we got here, let's look at where we are now.

```
7C924E68    PUSH EBX
7C924E69    CALL ntdll.RtlSplay
7C924E6E    MOV ECX,DWORD PTR [EBP+8]
7C924E71    MOV DWORD PTR [ECX],EAX
7C924E73    MOV EAX,DWORD PTR [EBP+14]
7C924E76    TEST EAX,EAX
7C924E78    JNZ ntdll.7C935D4F
7C924E7E    LEA EAX,DWORD PTR [EBX+18]
```

This sequence calls a function called `RtlSplay` (whose name you have because it is exported—remember, I'm *not* using the Windows debug symbol files!). `RtlSplay` takes one parameter. If `SearchResult == 1` that parameter is the `pNode` parameter passed to `RtlRealInsertElementWorker`. If it's anything else, `RtlSplay` takes a pointer to the new element that was just inserted. Afterward the tree root pointer at `pTable` is set to the return value of `RtlSplay`, which indicates that `RtlSplay` returns a tree node, but you don't really know what that node is at the moment.

The code that follows checks for the optional Boolean pointer and if it exists it is set to `TRUE` if `SearchResult != 1`. The function then loads the return value into `EAX`. It turns out that `RtlRealInsertElementWorker` simply returns the pointer to the data of the newly allocated element. Here's a corrected prototype for `RtlRealInsertElementWorker`.

```
PVOID RtlRealInsertElementWorker(
      TABLE *pTable,
      PVOID ElementData,
      ULONG ElementSize,
      BOOLEAN *pResult OPTIONAL,
      NODE *pNode,
      ULONG SearchResult
      );
```

Also, because `RtlInsertElementGenericTable` returns the return value of `RtlRealInsertElementWorker`, you can also update the prototype for `RtlInsertElementGenericTable`.

```
PVOID NTAPI RtlInsertElementGenericTable(
     TABLE   *pTable,
     PVOID    ElementData,
     ULONG    DataLength,
     BOOLEAN *pResult OPTIONAL,
     );
```

Splay Trees

At this point, one thing you're still not sure about is that `RtlSplay` function. I will not include it here because it is quite long and convoluted, and on top of that it appears to be distributed throughout the module, which makes it even more difficult to read. The fact is that you can pretty much start using the generic table without understanding `RtlSplay`, but you should probably still take a quick look at what it does, just to make sure you fully understand the generic table data structure.

The algorithm implemented in `RtlSplay` is quite involved, but a quick examination of what it does shows that it has something to do with the rebalancing of the tree structure. In binary trees, rebalancing is the process of restructuring the tree so that the elements are divided as evenly as possible under each side of each node. Normally, rebalancing means that an algorithm must check that the root node actually represents the median value represented by the tree. However, because elements in the generic table are user-defined, `RtlSplay` would have to make a callback into the user's code in order to compare elements, and there is no such callback in this function.

A more careful inspection of `RtlSplay` reveals that it's basically taking the specified node and moving it upward in the tree (you might want to run `RtlSplay` in a debugger in order to get a clear view of this process). Eventually, the function returns the pointer to the same node it originally starts with, except that now this node is the root of the entire tree, and the rest of the elements are distributed between the current element's left and right child nodes.

Once I realized that this is what `RtlSplay` does the picture became a bit clearer. It turns out that the generic table is implemented using a *splay tree* [Tarjan] Robert Endre Tarjan, Daniel Dominic Sleator. *Self-adjusting binary search trees*. Journal of the ACM (JACM). Volume 32 , Issue 3, July 1985, which is essentially a binary tree with a unique organization scheme. The problem of properly organizing a binary tree has been heavily researched and there are quite a few techniques that deal with it (If you're patient, Knuth provides an in-depth examination of most of them in [Knuth3] Donald E. Knuth. *The Art of Computer Programming—Volume 3: Sorting and Searching (Second Edition).* Addison Wesley. The primary goal is, of course, to be able to reach elements using the lowest possible number of iterations.

A splay tree (also known as a *self-adjusting binary search tree*) is an interesting solution to this problem, where every node that is touched (in any operation) is immediately brought to the top of the tree. This makes the tree act like a cache of sorts, whereby the most recently used items are always readily available, and the least used items are tucked at the bottom of the tree. By definition, splay trees always rotate the most recently used item to the top of the tree. This is why

you're seeing a call to `RtlSplay` immediately after adding a new element (the new element becomes the root of the tree), and you should also see a call to the same function after deleting and even just searching for an element.

Figures 5.1 through 5.5 demonstrate how `RtlSplay` progressively raises the newly added item in the tree's hierarchy until it becomes the root node.

RtlLookupElementGenericTable

Remember how before you started digging into the generic table I mentioned two functions (`RtlGetElementGenericTable` and `RtlLookupElementGenericTable`) that appeared to be responsible for retrieving elements? Because you know that `RtlGetElementGenericTable` searches for an element by its index, `RtlLookupElementGenericTable` must be the one that provides some sort of search capabilities for a generic table. Let's have a look at `RtlLookupElementGenericTable` (see Listing 5.8).

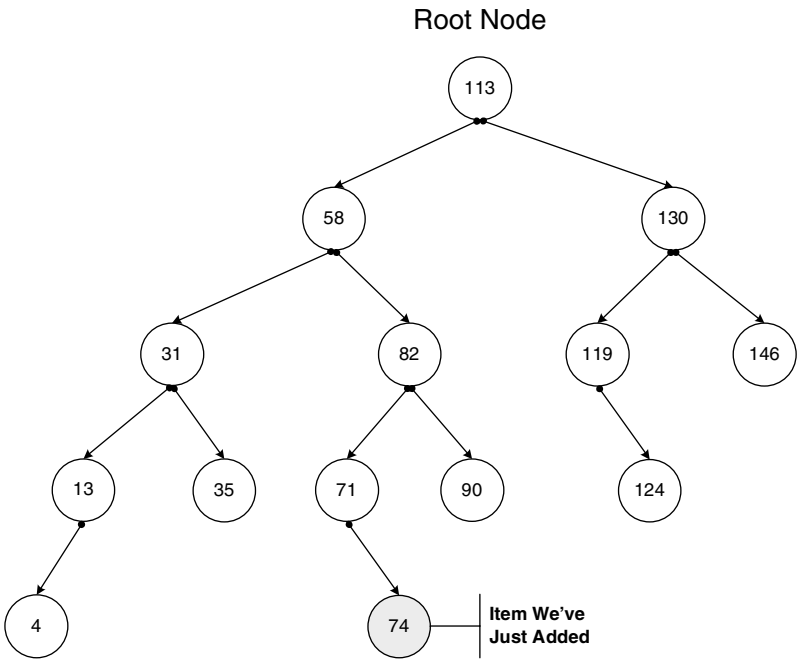

Figure 5.1 Binary tree after adding a new item. New item is connected to the tree at the most appropriate position, but no other items are moved.

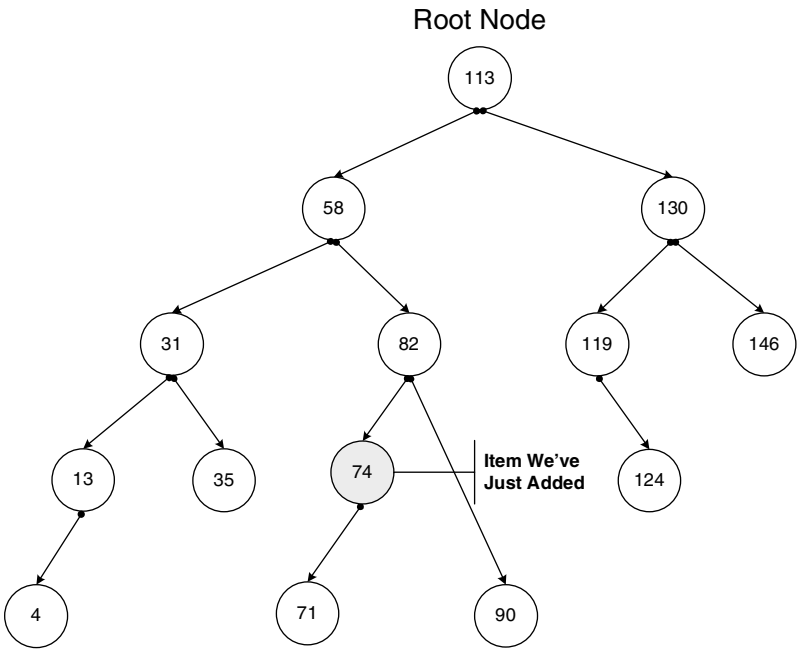

Figure 5.2 Binary tree after first splaying step. The new item has been moved up by one level, toward the root of the tree. The previous parent of our new item is now its child.

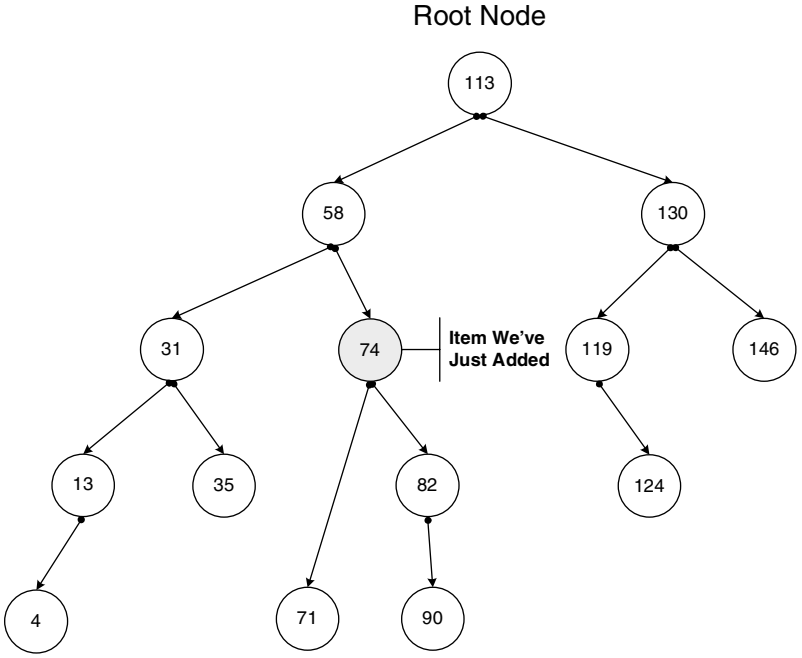

Figure 5.3 Binary tree after second splaying step. The new item has been moved up by another level.

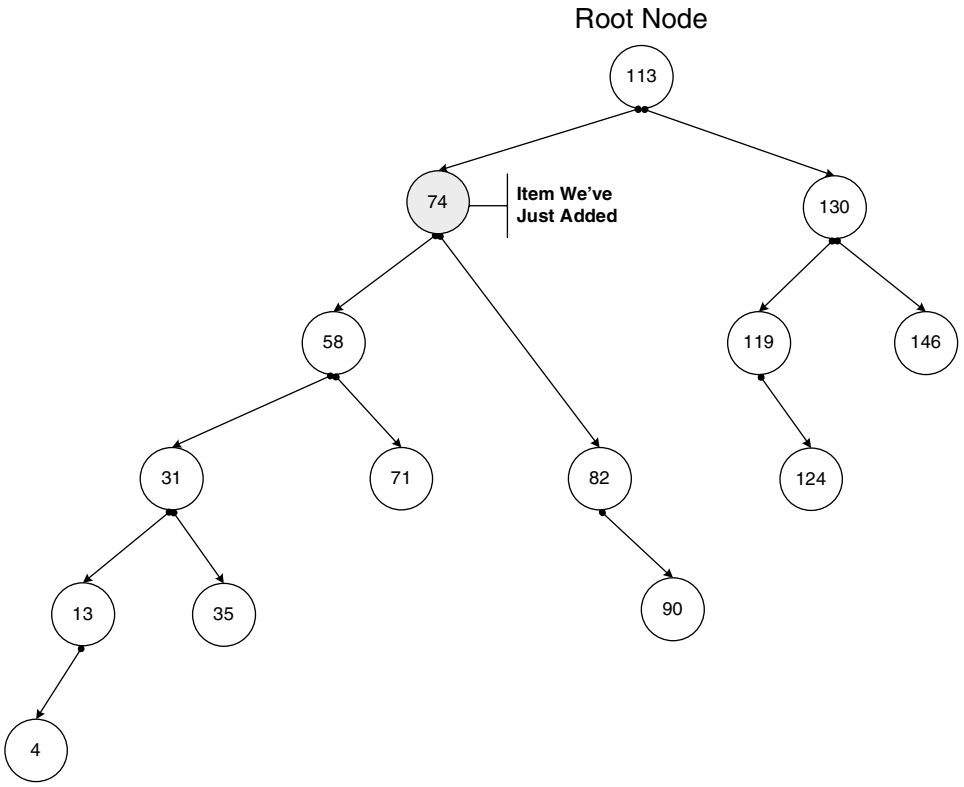

Figure 5.4 Binary tree after third splaying step. The new item has been moved up by yet another level.

```
7C9215BB    PUSH EBP
7C9215BC    MOV EBP,ESP
7C9215BE    LEA EAX,DWORD PTR [EBP+C]
7C9215C1    PUSH EAX
7C9215C2    LEA EAX,DWORD PTR [EBP+8]
7C9215C5    PUSH EAX
7C9215C6    PUSH DWORD PTR [EBP+C]
7C9215C9    PUSH DWORD PTR [EBP+8]
7C9215CC    CALL ntdll.7C9215DA
7C9215D1    POP EBP
7C9215D2    RET 8
```

Listing 5.8 Disassembly of RtlLookupElementGenericTable.

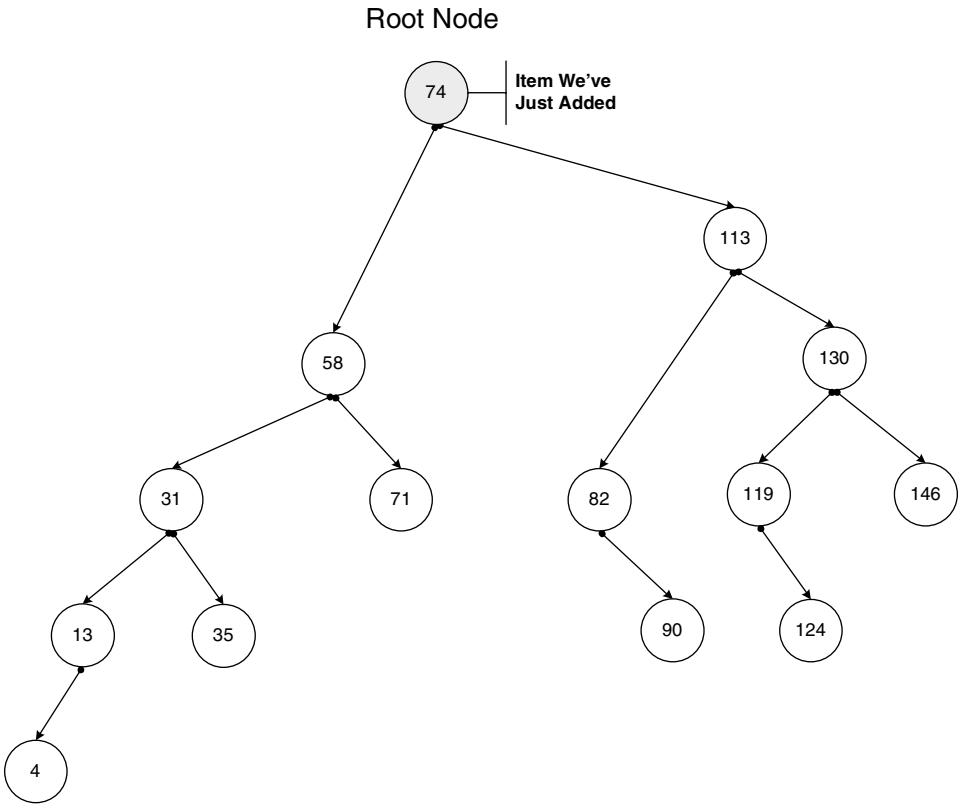

Figure 5.5 Binary after splaying process. The new item is now the root node, and the rest of the tree is centered on it.

From its name, you can guess that RtlLookupElementGenericTable performs a binary tree search on the generic table, and that it probably takes the `TABLE` structure and an element data pointer for its parameters. It appears that the actual implementation resides in `ntdll.7C9215DA`, so let's take a look at that function. Notice the clever stack use in the call to this function. The first two parameters are the same parameters that were passed to `RtlLookupElementGenericTable`. The second two parameters are apparently pointers to some kind of output values that `ntdll.7C9215DA` returns. They're apparently not used, but instead of allocating local variables that would contain them, the compiler is simply using the stack area that was used for passing parameters into the function. Those stack slots are no longer needed after they are read and passed on to `ntdll.7C9215DA`. Listing 5.9 shows the disassembly for `ntdll.7C9215DA`.

```
7C9215DA    MOV EDI,EDI
7C9215DC    PUSH EBP
7C9215DD    MOV EBP,ESP
7C9215DF    PUSH ESI
7C9215E0    MOV ESI,DWORD PTR [EBP+10]
7C9215E3    PUSH EDI
7C9215E4    MOV EDI,DWORD PTR [EBP+8]
7C9215E7    PUSH ESI
7C9215E8    PUSH DWORD PTR [EBP+C]
7C9215EB    CALL ntdll.7C92147B
7C9215F0    TEST EAX,EAX
7C9215F2    MOV ECX,DWORD PTR [EBP+14]
7C9215F5    MOV DWORD PTR [ECX],EAX
7C9215F7    JE SHORT ntdll.7C9215FE
7C9215F9    CMP EAX,1
7C9215FC    JE SHORT ntdll.7C921606
7C9215FE    XOR EAX,EAX
7C921600    POP EDI
7C921601    POP ESI
7C921602    POP EBP
7C921603    RET 10
7C921606    PUSH DWORD PTR [ESI]
7C921608    CALL ntdll.RtlSplay
7C92160D    MOV DWORD PTR [EDI],EAX
7C92160F    MOV EAX,DWORD PTR [ESI]
7C921611    ADD EAX,18
7C921614    JMP SHORT ntdll.7C921600
```

Listing 5.9 Disassembly of ntdll.7C9215DA, tentatively titled RtlLookupElementGeneric TableWorker.

At this point, you're familiar enough with the generic table that you hardly need to investigate much about this function—we've discussed the two core functions that this API uses: `RtlLocateNodeGenericTable` (`ntdll.7C92147B`) and `RtlSplay`. `RtlLocateNodeGenericTable` is used for the actual locating of the element in question, just as it was used in `RtlInsertElementGenericTable`. After `RtlLocateNodeGenericTable` returns, `RtlSplay` is called because, as mentioned earlier, splay trees are always splayed after adding, removing, or searching for an element. Of course, `RtlSplay` is only actually called if `RtlLocateNodeGenericTable` locates the element sought.

Based on the parameters passed into `RtlLocateNodeGenericTable`, you can immediately see that `RtlLookupElementGenericTable` takes the `TABLE` pointer and the `Element` pointer as its two parameters. As for the return value, the `add eax, 18` shows that the function takes the located node

and skips its header to get to the return value. As you would expect, this function returns the pointer to the found element's data.

RtlDeleteElementGenericTable

So we've covered the basic usage cases of adding, retrieving, and searching for elements in the generic table. One case that hasn't been covered yet is *deletion*. How are elements deleted from the generic table? Let's take a quick look at `RtlDeleteElementGenericTable`.

```
7C924FFF    MOV EDI,EDI
7C925001    PUSH EBP
7C925002    MOV EBP,ESP
7C925004    PUSH EDI
7C925005    MOV EDI,DWORD PTR [EBP+8]
7C925008    LEA EAX,DWORD PTR [EBP+C]
7C92500B    PUSH EAX
7C92500C    PUSH DWORD PTR [EBP+C]
7C92500F    CALL ntdll.7C92147B
7C925014    TEST EAX,EAX
7C925016    JE SHORT ntdll.7C92504E
7C925018    CMP EAX,1
7C92501B    JNZ SHORT ntdll.7C92504E
7C92501D    PUSH ESI
7C92501E    MOV ESI,DWORD PTR [EBP+C]
7C925021    PUSH ESI
7C925022    CALL ntdll.RtlDelete
7C925027    MOV DWORD PTR [EDI],EAX
7C925029    MOV EAX,DWORD PTR [ESI+C]
7C92502C    MOV ECX,DWORD PTR [ESI+10]
7C92502F    MOV DWORD PTR [ECX],EAX
7C925031    MOV DWORD PTR [EAX+4],ECX
7C925034    DEC DWORD PTR [EDI+14]
7C925037    AND DWORD PTR [EDI+10],0
7C92503B    PUSH ESI
7C92503C    LEA EAX,DWORD PTR [EDI+4]
7C92503F    PUSH EDI
7C925040    MOV DWORD PTR [EDI+C],EAX
7C925043    CALL DWORD PTR [EDI+20]
7C925046    MOV AL,1
7C925048    POP ESI
7C925049    POP EDI
7C92504A    POP EBP
7C92504B    RET 8
7C92504E    XOR AL,AL
7C925050    JMP SHORT ntdll.7C925049
```

Listing 5.10 Disassembly of RtlDeleteElementGenericTable.

RtlDeleteElementGenericTable has three primary steps. First of all it uses the famous RtlLocateNodeGenericTable (ntdll.7C92147B) for locating the element to be removed. It then calls the (exported) RtlDelete to actually remove the element. I will not go into the actual algorithm that RtlDelete implements in order to remove elements from the tree, but one thing that's important about it is that after performing the actual removal it also calls RtlSplay in order to restructure the table.

The last function call made by RtlDeleteElementGenericTable is actually quite interesting. It appears to be a callback into user code, where the callback function pointer is accessed from offset +20 in the TABLE structure. It is pretty easy to guess that this is the element-free callback that frees the memory allocated in the TABLE_ALLOCATE_ELEMENT callback earlier. Here is a prototype for TABLE_FREE_ELEMENT:

```
typedef void ( _stdcall * TABLE_FREE_ELEMENT) (
     TABLE  *pTable,
     PVOID   Element
     );
```

There are two things to note here. First of all, TABLE_FREE_ELEMENT clearly doesn't have a return value, and if it does RtlDeleteElementGenericTable certainly ignores it (see how right after the callback returns AL is set to 1). Second, keep in mind that the Element pointer is going to be a pointer to the beginning of the NODE data structure, and not to the beginning of the element's data, as you've been seeing all along. That's because the caller allocated this entire memory block, including the header, so it's now up to the caller to free this entire memory block.

RtlDeleteElementGenericTable returns a Boolean that is set to TRUE if an element is found by RtlLocateNodeGenericTable, and FALSE if RtlLocateNodeGenericTable returns NULL.

Putting the Pieces Together

Whenever a reversing session of this magnitude is completed, it is advisable to prepare a little document that describes your findings. It is an elegant way to summarize the information obtained while reversing, not to mention that most of us tend to forget this stuff as soon as we get up to get a cup of coffee or a glass of chocolate milk (my personal favorite).

The following listings can be seen as a formal definition of the generic table API, which is based on the conclusions from our reversing sessions. Listing 5.11 presents the internal data structures, Listing 5.12 presents the callbacks prototypes, and Listing 5.13 presents the function prototypes for the APIs.

```
struct NODE
{
  NODE                      *ParentNode;
  NODE                      *RightChild;
  NODE                      *LeftChild;
  LIST_ENTRY                LLEntry;
  ULONG                     Unknown;
};

struct TABLE
{
  NODE                      *TopNode;
  LIST_ENTRY                LLHead;
  LIST_ENTRY                *LastElementFound;
  ULONG                     LastElementIndex;
  ULONG                     NumberOfElements;
  TABLE_COMPARE_ELEMENTS    CompareElements;
  TABLE_ALLOCATE_ELEMENT    AllocateElement;
  TABLE_FREE_ELEMENT        FreeElement;
  ULONG                     Unknown;
};
```

Listing 5.11 Definitions of internal generic table data structures discovered in this chapter.

```
typedf int (NTAPI * TABLE_COMPARE_ELEMENTS) (
    TABLE   *pTable,
    PVOID    pElement1,
    PVOID    pElement2
    );

typedef NODE * (NTAPI * TABLE_ALLOCATE_ELEMENT) (
    TABLE    *pTable,
    ULONG    TotalElementSize
    );

typedef void (NTAPI * TABLE_FREE_ELEMENT) (
    TABLE    *pTable,
    PVOID    Element
    );
```

Listing 5.12 Prototypes of generic table callback functions that must be implemented by the caller.

```
void NTAPI RtlInitializeGenericTable(
    TABLE *pGenericTable,
    TABLE_COMPARE_ELEMENTS CompareElements,
    TABLE_ALLOCATE_ELEMENT AllocateElement,
    TABLE_FREE_ELEMENT FreeElement,
    ULONG Unknown
    );

ULONG NTAPI RtlNumberGenericTableElements(
    TABLE *pGenericTable
    );

BOOLEAN NTAPI RtlIsGenericTableEmpty(
    TABLE *pGenericTable
    );

PVOID NTAPI RtlGetElementGenericTable(
    TABLE *pGenericTable,
    ULONG ElementNumber
    );

PVOID NTAPI RtlInsertElementGenericTable(
    TABLE *pGenericTable,
    PVOID ElementData,
    ULONG DataLength,
    OUT BOOLEAN *IsNewElement
    );

PVOID NTAPI RtlLookupElementGenericTable(
    TABLE *pGenericTable,
    PVOID ElementToFind
    );

BOOLEAN NTAPI RtlDeleteElementGenericTable(
    TABLE *pGenericTable,
    PVOID ElementToFind
    );
```

Listing 5.13 Prototypes of the basic generic table APIs.

Conclusion

In this chapter, I demonstrated how to investigate, use, and document a reasonably complicated set of functions. If there is one important moral to this

story, it is that reversing is always about meeting the low-level with the high-level. If you just keep tracing through registers and bytes, you'll never really get anywhere. The secret is to always keep your eye on the big picture that's slowly materializing in front of you while you're reversing. I've tried to demonstrate this process as clearly as possible in this chapter. If you feel as if you've missed some of the steps we took in order to get to this point, fear not. I highly recommend that you go over this chapter more than once, and perhaps use a live debugger to step through this code while reading the text.