Part

II

Other Platforms—Windows, Solaris, OS/X, and Cisco

Now that you have completed the introductory section on vulnerability development for the Linux/IA32 platform, we explore more difficult and tricky operating systems and exploitation concepts. We move into the world of Windows, where we detail some interesting exploitation concepts from a Window's hacker point of view. The first chapter in this part, Chapter 6, will help you understand how Windows is different from the Linux/IA32 content in Part I. We move right into Windows shellcode in Chapter 7, and then delve into some more advanced Windows content in Chapter 8. Finally, we round out the Windows content with a chapter on overcoming filters for Windows in Chapter 9. The concepts for circumventing various filters can be applied to any hostile code injection scenario.

The other chapters in this section show you how to discover and exploit vulnerabilities for the Solaris and OS X operating systems and the Cisco platform. Because Solaris runs on an entirely different architecture than the Linux and Windows content described thus far, it may at first appear alien to you. The two Solaris chapters will have you hacking Solaris on SPARC like a champ, introducing the Solaris platform in Chapter 10 and delving into more advanced concepts in Chapter 11, such as abusing the Procedure Linkage Table and the use of native blowfish encryption in shellcode.

Chapter 12 introduces OS X and walks through the peculiarities of writing exploits on the Intel and PowerPC platforms. Chapter 13 discusses the various Cisco platforms and techniques that can help you find and exploit bugs on them, and Chapter 14 discusses the various exploit protection mechanisms that have recently (and in some cases, not so recently) been introduced into most common operating systems and compilers.

Once you've completed Part II, you should have a basic grip on most of the techniques that you need to understand and write exploits on pretty much every operating system out there, as well as a keen understanding of the various obstacles that OS and compiler vendors put in the way.

CHAPTER

6

The Wild World of Windows

We have reached the point in the book in which all operating systems will be defined by their differences from Linux. This chapter will give experienced Windows hackers a fresh perspective on Microsoft issues and at the same time allow Unix-oriented hackers to gain a good grasp of Windows internals. At the end of this chapter, you should be able to write a basic Windows exploit and avoid some of the common pitfalls that will stand in your way when you attempt more complex exploits.

You'll also gain an understanding of how to use basic Windows debugging tools. Along the way you'll develop an understanding of the Windows security and programming model and a basic knowledge of Distributed Component Object Model (DCOM) and Portable Executable–Common File Format (PE-COFF). In short, this chapter contains everything an expert-level hacker with years of real-world experience would have loved to know when first learning to attack Windows platforms.

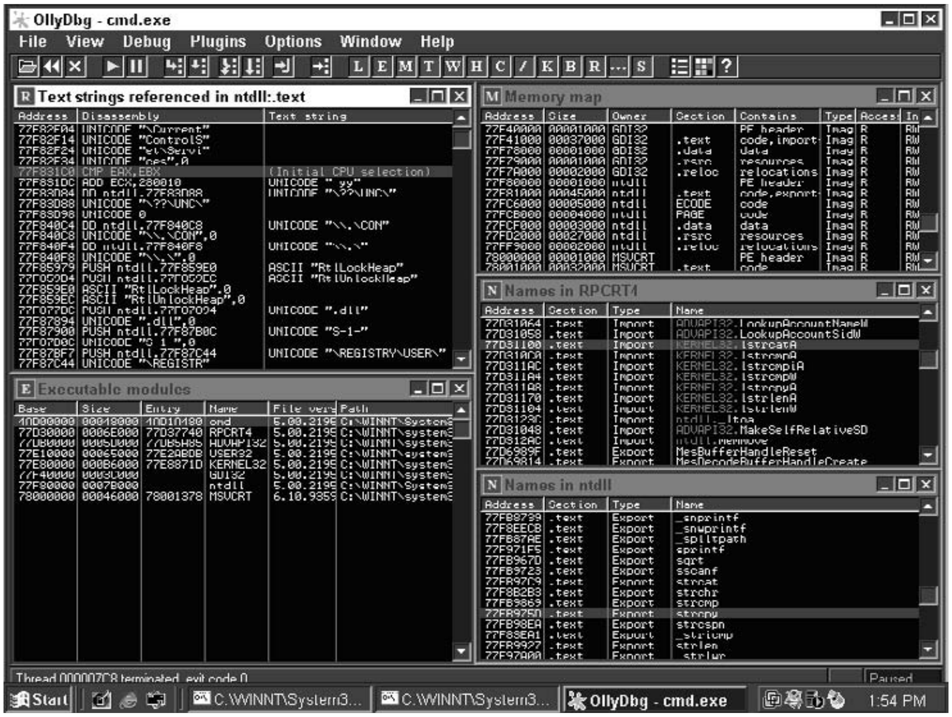
How Does Windows Differ from Linux?

The Windows NT team made a few design decisions early on that profoundly affected every resulting architecture. The NT project was in full swing in 1989, with its first release in 1991 as Windows NT 3.1. Most of the internals originally were inspired by VMS, although there were several major differences between VMS and NT, notably an inclusion of kernel threads in the early versions of the NT kernel. This chapter visits some major features of NT that may not be recognizable to someone used to Linux or Unix internals.

Win32 API and PE-COFF

OllyDbg, a full-featured, assembler-level, analyzing debugger that runs on Windows (see Figure 6-1), is a powerful tool for binary analysis. You will best understand the content in this chapter when working with a binary analysis debugger such as OllyDbg. To apply what you learn here, you will need a tool with its features. OllyDbg is distributed under a shareware license and found at `http://www.ollydbg.de/`.The native API for Windows programs is the 32-bit Windows API, which a Linux programmer can think of simply as a collection of all the shared libraries available in `/usr/lib`.

NOTE If you are a little rusty on the Windows API or are entirely new to it, you can read an excellent online tutorial on the Windows API by Brook Miles at `http://www.winprog.org/tutorial/`**.**



Text strings referenced in ntdll:.text

```
Address   Disassembly                  Text string
77F82F04  UNICODE "\Current"
77F82F14  UNICODE "ControlS"
77F82F24  UNICODE "et\Servi"
77F82F34  UNICODE "ces",0
77F831C0  CMP EAX,EBX                  (Initial CPU selection)
77F831DC  ADD ECX,280010               UNICODE " yy"
77F83D84  DD ntdll.77F83D88            UNICODE "\??\UNC\"
77F83D88  UNICODE "\??\UNC\"
77F83D98  UNICODE 0
77F840C4  DD ntdll.77F840C8            UNICODE "\\.\CON"
77F840C8  UNICODE "\\.\CON",0
77F840F4  DD ntdll.77F840F8            UNICODE "\\.\"
77F840F8  UNICODE "\\.\",0
77F85979  PUSH ntdll.77F859E0          ASCII "RtlLockHeap"
77F852D4  PUSH ntdll.77F852EC          ASCII "RtlUnlockHeap"
77F859E0  ASCII "RtlLockHeap",0
77F859EC  ASCII "RtlUnlockHeap",0
77F077DC  PUSH ntdll.77F07024          UNICODE ".dll"
77F87894  UNICODE ".dll",0
77F87900  PUSH ntdll.77F87B0C          UNICODE "S-1-"
77F07D0C  UNICODE "S 1 ",0
77F87BF7  PUSH ntdll.77F87C44          UNICODE "\REGISTRY\USER\"
77F87C44  UNICODE "\REGISTR"
```

Executable modules

```
Base      Size      Entry     Name      File vers Path
40D00000  00018000  40D10480  cmd       5.00.2195 C:\WINNT\System3
77D30000  0006E000  77D37740  RPCRT4    5.00.2195 C:\WINNT\system3
77DB0000  0005D000  77DB5A85  ADVAPI32  5.00.2195 C:\WINNT\system3
77E10000  00065000  77E2ABDB  USER32    5.00.2195 C:\WINNT\system3
77E80000  000B6000  77E8871D  KERNEL32  5.00.2195 C:\WINNT\system3
77F40000  0003C000            GDI32     5.00.2195 C:\WINNT\system3
77F80000  0007B000            ntdll     5.00.2195 C:\WINNT\System3
78000000  00046000  78001378  MSVCRT    6.10.9359 C:\WINNT\system3
```

Memory map

```
Address   Size      Owner   Section  Contains       Type  Access In
77F40000  00001000  GDI32            PE header      Imag  R      RW
77F41000  00037000  GDI32   .text    code,import    Imag  R      RW
77F78000  00001000  GDI32   .data    data           Imag  R      RW
77F79000  00001000  GDI32   .rsrc    resources      Imag  R      RW
77F7A000  00002000  GDI32   .reloc   relocations    Imag  R      RW
77F80000  00001000  ntdll            PE header      Imag  R      RW
77F81000  00045000  ntdll   .text    code,export    Imag  R      RW
77FC6000  00005000  ntdll   ECODE    code           Imag  R      RW
77FCB000  00004000  ntdll   PAGE     code           Imag  R      RW
77FCF000  00003000  ntdll   .data    data           Imag  R      RW
77FD2000  00027000  ntdll   .rsrc    resources      Imag  R      RW
77FF9000  00002000  ntdll   .reloc   relocations    Imag  R      RW
78000000  00001000  MSVCRT           PE header      Imag  R      RW
78001000  00032000  MSVCRT  .text    code           Imag  R      RW
```

Names in RPCRT4

```
Address   Section  Type    Name
77D31064  .text    Import  ADVAPI32.LookupAccountNameW
77D31058  .text    Import  ADVAPI32.LookupAccountSidW
77D31100  .text    Import  KERNEL32.lstrcatA
77D310A0  .text    Import  KERNEL32.lstrcmpA
77D311AC  .text    Import  KERNEL32.lstrcmpiA
77D311A4  .text    Import  KERNEL32.lstrcmpW
77D311A8  .text    Import  KERNEL32.lstrcpyA
77D31170  .text    Import  KERNEL32.lstrlenA
77D31104  .text    Import  KERNEL32.lstrlenW
77D3123C  .text    Import  ntdll._ltoa
77D31048  .text    Import  ADVAPI32.MakeSelfRelativeSD
77D312AC  .text    Import  ntdll.memmove
77D6989F  .text    Export  MesBufferHandleReset
77D69814  .text    Export  MesDecodeBufferHandleCreate
```

Names in ntdll

```
Address   Section  Type    Name
77FB8739  .text    Export  _snprintf
77F8EECB  .text    Export  _snwprintf
77FB87AE  .text    Export  _splitpath
77F971F5  .text    Export  sprintf
77FB967D  .text    Export  sqrt
77FB9723  .text    Export  sscanf
77FB97C9  .text    Export  strcat
77F8B2B3  .text    Export  strchr
77FB9869  .text    Export  strcmp
77FB975D  .text    Export  strcpy
77FB98EA  .text    Export  strcspn
77F83EA1  .text    Export  _stricmp
77FB9927  .text    Export  strlen
77F97A00  .text    Export  _strlwr
```

```
Thread 000007C8 terminated, exit code 0                Paused
```

Figure 6-1: OllyDbg can show you all the information you need about any DLLs loaded into memory.

A skilled Linux programmer can write a program that talks directly to the kernel, for example by using the `open()` or `write()` syscalls. No such luck on Windows. Each new service pack and release of Windows NT changes the kernel interface, and a corresponding set of libraries (known as Dynamic Link Libraries [DLLs]) are included with the release to make programs continue to work. DLLs provide a way for a process to call a function that is not part of its own executable code. The executable code for the function is located in a DLL, containing one or more functions that are compiled, linked, and stored separately from the processes using them. The Windows API is implemented as an orderly set of DLLs, so any process using the Win32 API uses dynamic linking.

This gives the Windows Kernel Team a way to change their internal APIs, or to add complex new functionality to them, while still providing a reasonably stable API for program developers to use. In contrast, you can't add a new argument to a syscall in any Unix variant without a horde of programmers calling foul.

Like any modern operating system, Windows uses a relocatable file format that gets loaded at runtime to provide the functionality of shared libraries. In Linux, these would be `.so` files, but in Windows these are DLLs. Much like a `.so` is an ELF file, a DLL is a PE-COFF file (also referred to as PE—portable executable). PE-COFF was derived from the Unix COFF format. PE files are portable because they can be loaded on every 32-bit Windows platform; the PE loader accepts this file format.

A PE file has an import and export table at the beginning of the file that indicates both what files the PE needs to find and what functions inside those files it needs. The export indicates what functions the DLL provides. It also marks where in the file, once loaded into memory, to find the functions. *The import table lists all the functions that the PE file uses that are in DLLs, as well as listing the name of the DLL in which the imported function resides.*

Most PE files are relocatable. Like ELF files, a PE file is composed of various sections; the `.reloc` section can be used to relocate the DLL in memory. The purpose of the `.reloc` section is to allow one program to load two DLLs that were compiled to use the same memory space.

Unlike Unix, the default behavior in Windows is to search for DLLs within the current working directory before it searches anywhere else. This provides certain abilities to escape Citrix or Terminal Server restrictions from a hacker's perspective, but from a developer's perspective it allows an application developer to distribute a version of a DLL that may be different from the one in the system root (`\winnt\system32`). This kind of versioning issue is sometimes called *DLL-hell*. Users will have to adjust their `PATH` environment variable and move DLLs around so that they don't conflict with each other when trying to load a broken program.

An important first thing to learn about PE-COFF is the *Relative Virtual Address* (RVA). RVAs are used to reduce the amount of work that the PE loader must accomplish. Functions can be relocated anywhere in the virtual address space; it would be extremely expensive if the PE loader had to fix every relocatable item. You'll notice as you learn Win32 that Microsoft tends to use acronyms (RVA, AV [Access Violation], AD [Active Directory], and so forth) rather than abbreviating the terms themselves as done in Unix (tmp, etc, vi, segfault). Each new Microsoft document introduces a few thousand additional terms and their associated acronyms.

NOTE Fun fact for conspiracy theorists: Near the Microsoft campus is a rather prominent Scientologist building that no one ever seems to go into or come out of.

RVA is just shorthand for saying "Each DLL gets loaded into memory at a base address, and then you add the RVA to the base address to find something." So, for example, the function `malloc()` is in the DLL `msvcrt.dll`. The header in `msvcrt.dll` contains a table of functions that `msvcrt.dll` provides, the export table. The export table contains a string with `malloc` and an RVA (for example, at `2000`); after the DLL is loaded into memory, perhaps at `0x80000000`, you can find the `malloc` function by going to `0x80002000`. The default Windows NT location into which an `.EXE` is loaded is `0x40000000`. This may change depending on language packs or compiler options, but is reasonably standard.

Symbols for PE-COFF files distributed by Microsoft are usually contained externally. You can download symbol packs for each version of its operating systems from Microsoft's MSDN Web site, or use its Symbol Server remotely with WinDbg. OllyDbg does not currently support the remote Symbol Server.

For more on PE-COFF, search Microsoft's Web site for "PE-COFF." As a final note, keep in mind that, like a few broken Unixes, Windows NT will not let you delete a file that is currently in use.

Heaps

When a DLL gets loaded, it calls an initialization function. This function often sets up its own heap using `HeapCreate()` and stores a global variable as a pointer to that heap so that future allocation operations can use it instead of the default heap. Most DLLs have a `.data` section in memory for storing global variables, and you will often find useful function pointers or data structures stored in that area. Because many DLLs are loaded, there are many heaps. With so many heaps to keep track of, heap corruption attacks can become quite confusing. In Linux, there is typically a single heap that can get corrupted, but in Windows, several heaps may get corrupted at once, which makes analyzing the situation much more complex. When a user calls `malloc()` in Win32, he or she is actually using a function exported by `msvcrt.dll`, which then calls `HeapAllocate()` with `msvcrt.dll`'s private heap. You may be tempted to try to use the `HeapValidate()` function to analyze a heap corruption situation, but this function does not do anything useful.

The confusion generally occurs when you have finished exploiting a heap overflow and you want to call some Win32 API functions with your shellcode. Some of your functions will work and some will cause access violations inside `RtlHeapFree()` or `RtlHeapAllocate`, which may terminate the process before you've had a chance to take control. `WinExec()` and the like are notorious for not working with a corrupted heap.

Each process has a default heap. The default heap can be found with `GetDefaultHeap()`, although that heap is unlikely to be the one that got corrupted. An important thing to note is that heaps can grow across segments. For example, if you send enough data to IIS, you will notice it allocating segments in high-order memory ranges and using that to store your data. Manipulating memory this way may be a useful trick if you have a limited set of characters with which to overwrite the return address, and if you need to get away from the low-memory address of default heaps. For this reason, memory leaks in target programs can become quite useful, because they let you fill all the program's memory with your shellcode.

Heap overflows on Windows are about as easy to write as they are on Unix. Use the same basic techniques to exploit them—if you're careful, you can even squeeze more than one write out of a heap overflow on Windows, which makes reliable exploitation much easier.

Threading

Threading allows one process to do multiple things, sharing a single memory space. The Windows kernel gives processor-time slices to threads, not processes. Linux does things with a "light-weight process" model, which is fairly weak; only when Linux Native Threads gets implemented will Linux be on stable thread footing with the rest of the modern OS world. Threads simply aren't as important a programming model under Linux for reasons that will become clear as the NT security structure is explained.

Threading is the reason for `HRESULT`. `HRESULT`, basically an integer value, is returned by almost all Win32 API calls. `HRESULT` can be either an error value or an `OK` value. If it is an error value, you can get the specific error with `GetLastError()`, which retrieves a value from the thread's local storage. If you think about Unix's model, there's no way to differentiate one thread's `errno` from another. Win32 was designed from the ground up to be a threaded model.

Windows has no `fork()` (used to spawn a new process in Linux). Instead, `CreateProcess()` will spawn a new process that has its own memory space. This process can inherit any of the handles its parent has marked inheritable. However, the parent must then pass these handles to the child itself or have the child guess at their values (handles are typically small integers, like file handles).

Because almost all overflows occur in threads, the attacker never knows a valid stack address. This means the attacker almost always uses a return-into-libc-style trick (although using any DLL, not just libc or the equivalent) to gain control of execution.

The Genius and Idiocy of the Distributed Common Object Model and DCE-RPC

The Distributed Common Object Model (DCOM), DCE-RPC, NT's Threading and Process Architecture, and NT's Authentication Tokens are all interconnected. It helps to first understand the overall philosophy of COM in order to understand what sets COM apart from its Unix counterparts.

You should remember that Microsoft's position on software has always been to distribute binary packages for money and build an economy to support that. Therefore, every Microsoft software architecture supports this model. You can build a fairly complex application entirely by buying third-party COM modules from various vendors, throwing them into a directory structure, and then using Visual Basic script to tie them together.

COM objects can be written in any language COM supports and interoperate seamlessly. Most of COM's idiosyncrasies come forth as natural design decisions; for example, what is an integer to C++ may not be an integer to Visual Basic.

To dig deeper into COM, you should look at a typical Interface Description Language (IDL) file. We'll use a DCOM IDL file, which you will recognize later:

```
[ uuid(e33c0cc4-0482-101a-bc0c-02608c6ba218),
  version(1.0),
  implicit_handle(handle_t rpc_binding)
] interface ???
{
  typedef struct {
    TYPE_2 element_1;
    TYPE_3 element_2;
  } TYPE_1;
...
```

```
short Function_00(
      [in] long element_9,
      [in] [unique] [string] wchar_t *element_10,
      [in] [unique] TYPE_1 *element_11,
      [in] [unique] TYPE_1 *element_12,
      [in] [unique] TYPE_2 *element_13,
      [in] long element_14,
      [in] long element_15,
     [out] [context_handle] void *element_16
);
```

What we've defined here is similar to a C++ class's header file. It simply says that these are the arguments (and return values) for a particular function in a particular interface as defined by that UUID. Anything that must be unique—any name—is a GUID in COM. This 128-bit number is supposed to be *globally* unique; that is, there can be only one. Every time we see a reference to that particular UUID, we know we're talking about this exact interface.

Interface descriptions for COM objects can be arbitrarily complex. The compiler (and COM support) for the language is supposed to create a bit of code that can transform as *long* as the IDL specifies it into the format in which the language needs it to be represented. It is the same with characters, arrays, pointers stored with arrays, structures that have other arrays, and so on.

In practice, a number of shortcuts can be taken to maintain acceptable speed. By saying that a long will be 32 bits in little-endian order, transforming from C++ to another C++ COM object's representation is trivial.

A COM object can be called in two ways: It can be loaded directly into the process space as a DLL, or it can be launched as a service (by the Service Control Manager, a special process that runs as SYSTEM). Running a COM server in another process ensures that your process will be stable and more secure, though much slower. In-Process calls, which require no transformation of data types, are literally one thousand times faster than calling a COM interface on the same machine but in a different process. Going to the same machine is usually at least ten times faster than going to a machine on the same network.

The important thing to Microsoft was that programmers could make a simple registry change or change one parameter in a program, and then that program would use a different process, or a different machine to make the same call.

For example, look at the AT service on NT. If you were to write a program to interact with AT and schedule commands, you could look up the interface definition for the AT service, make a DCOM call to bind to that interface, and then call a particular procedure on that interface. Of course, you'd need the IDL file to know how to transform your arguments before you sent the data between your process and the AT service's process. This same procedure would work even if the process were on another computer entirely. In that case, your DCOM libraries would connect to the remote computer's endpoint mapper (TCP port 135) and then ask it where the `AT` service was listening. The endpoint mapper (itself a DCOM service, but one that is always at a known port) would respond "The `AT` service is listening on the following named pipe RPC services, which you can connect to over ports 445 or 139. It is also listening on TCP port 1025 and UDP port 1034 for DCE-RPC calls." All of this would be transparent to the developer.

Now you know the genius of DCE-RPC and DCOM. You can sell binary DCOM packages or simply put up a network-accessible machine with those DCOM interfaces installed and let developers connect to them from Visual Basic, C++, or any other DCOM-enabled language. For extra speed, you can load the interfaces directly into your client process as a DLL. This paradigm is the basis of almost all the features that make Windows NT a distinctive server platform. "Rich clients," "Remote manageability," and "Rapid Application Development" are all just the same thing—DCOM.

But of course, this is also the idiocy of DCE-RPC and DCOM. One man's remote manageability is another man's remote vulnerability. As a hacker, your goal is to know the target systems better than their administrators do. With DCOM as a complex, impossible-to-understand basis for every aspect of a system's security, this is not hard to do.

The next sections go over a few of the basics for exploiting DCE-RPC and DCOM.

Recon

Two useful tools for basic remote DCE-RPC recon are Dave Aitel's SPIKE (`www.immunitysec.com/`) and Todd Sabin's DCE-RPC tools (available from `http://www.bindview.com/Services/razor/Utilities/`).

In this example, we'll use SPIKE's dcedump utility to view the DCE-RPC services (also known as DCOM interfaces) available remotely that are registered with the endpoint mapper. This is roughly the same as calling `rpcdump -p` on a Unix system.

```
[dave@localhost dcedump]$ ./dcedump 192.168.1.108 | head -20
DCE-RPC tester.
TcpConnected
Entrynum=0

annotation=
uuid=4f82f460-0e21-11cf-909e-00805f48a135 , version=4
Executable on NT: inetinfo.exe
ncacn_np:\\WIN2KSRV[\PIPE\NNTPSVC]
Entrynum=1
```

```
annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncalrpc[LRPC000001f4.00000001]
Entrynum=2

annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncacn_ip_tcp:192.168.1.108[1025]
...
```

As you can see, here we have three different interfaces and three different ways to connect to them. We can further examine the interface that the endpoint mapper provides with SPIKE's interface ids (ifids) utility. Likewise, we can examine almost any other TCP-enabled interface (`msdtc.exe` is one exception).

```
[dave@localhost dcedump]$ ./ifids 192.168.1.108 135
DCE-RPC IFIDS by Dave Aitel.
Finds all the interfaces and versions listening on that TCP port
Tcp Connected
Found 11 entries
e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0
0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1
975201b0-59ca-11d0-a8d5-00a0c90d8051 v1.0
e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0
99fcfec4-5260-101b-bbcb-00aa0021347a v0.0
b9e79e60-3d52-11ce-aaa1-00006901293f v0.2
412f241e-c12a-11ce-abff-0020af6e7a17 v0.2
00000136-0000-0000-c000-000000000046 v0.0
c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0
4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0
000001a0-0000-0000-c000-000000000046 v0.0

Done
```

Now, these can be fed directly into SPIKE's msrpcfuzz program to attempt to find overflows in the endpoint mapper or in any other TCP service. If you had the IDL for these services (you can get some of them from open source projects such as Snort), you could guide your analysis of these functions. Otherwise you are reduced to doing automatic or manual binary analysis. One program that may help you is Muddle, by Matt Chapman. You can find this program at `www.cse.unsw.edu.au/~matthewc/muddle/`; it will automatically decode certain executables to tell you their arguments. Muddle generated the IDL fragment you saw earlier in this chapter, which we took from the file for the RPC locator service.

Microsoft has tunneled the DCE-RPC protocol across almost anything it can get its hands on. From SMB to SOAP, if you can tunnel DCE-RPC across it, you've enabled all Microsoft's tools. In the examples, you can see a DCE-RPC over named pipe interface (`ncacn_np`), a DCE-RPC over Local RPC interface, and a DCE-RPC over TCP interface. Named pipe, TCP, and UDP interfaces are all accessible remotely and should make your mouth water.

Exploitation

There are as many ways to exploit a remote DCOM service as there are to exploit a remote SunRPC service. You can do `popen()` or `system()` style attacks, try to access files on the filesystem, find buffer overflows or similar attacks, try to bypass authentication, or anything else you can think up that a remote server might be vulnerable to. The best tool currently publicly available for playing with RPC services is SPIKE. However, if you want to exploit remote DCE-RPC services, you will have to do a lot of work duplicating this protocol in the language of your choice. CANVAS (`www.immunitysec.com/CANVAS/`) duplicates DCE-RPC using Python.

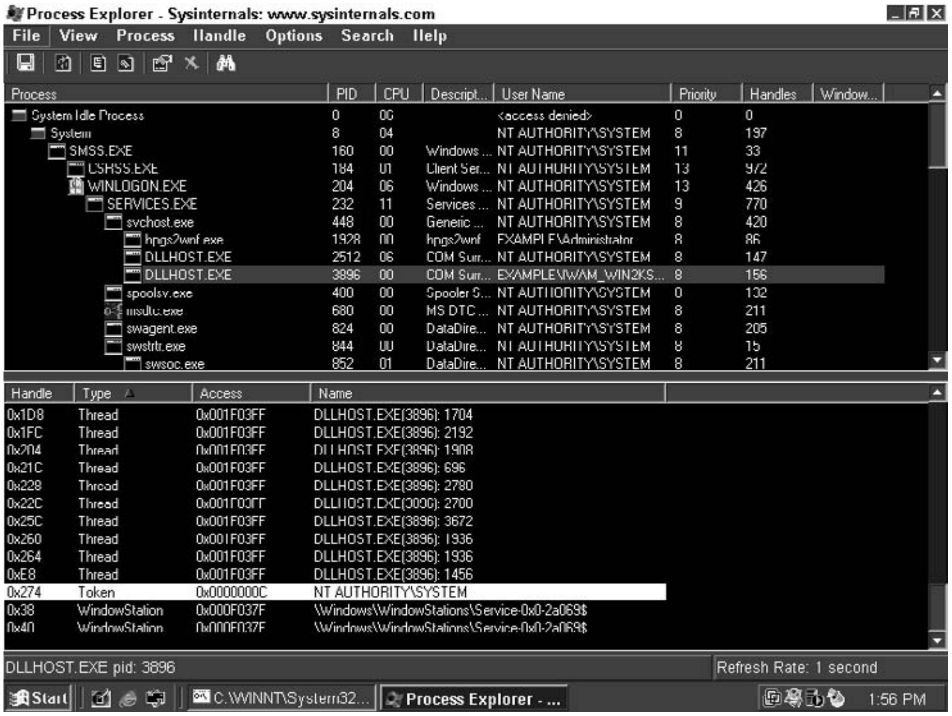
At first you may be tempted to use Microsoft's internal APIs to do DCE-RPC or DCOM exploitation work, but in the long run, your inability to directly control the APIs will lead to shoddy exploits. Definitely keep to using your own or an open source protocol implementation if possible.

Tokens and Impersonation

Tokens are exactly what they sound like—representations of access rights. In Windows, your access rights to things such as files or processes are not defined by a simple user/group/any permission set the way they are on Linux. Instead they use a flexible, and extremely poorly understood mechanism that relies on tokens. In the smallest sense, a token is simply a 32-bit integer, much like a file handle. The NT kernel maintains an internal structure per process that indicates what each token represents in terms of access rights. For example, when a process wants to spawn another process it must check to see if it can access the file it wants to spawn.

Now, here is where things get complicated, because there are several types of tokens, and two tokens can affect each operation: the primary token and the current thread token. The process was given the primary token when it started up. The current thread token can be obtained from another process or from the `LogonUser()` function. The `LogonUser()` function requires a username and password and returns a new token if it is successful. You can attach any given token to your current thread using `SetThreadToken(token_to_attach)` and remove it with `RevertToSelf()`, at which point the thread reverts to the primary token.

For fun, load the Sysinternals (`http://www.microsoft.com/technet/sysinternals/`) Process Explorer to a process and you'll see several things: The primary token is printed out as `ser Name` and you may see one or more tokens with varying levels of access listed in the bottom pane. Figure 6-2 shows the various tokens in a process.


Process Explorer - Sysinternals: www.sysinternals.com
File View Process Handle Options Search Help
Start
C:\WINNT\System32...
Process Explorer - ...
1:56 PM

```
Process                      PID    CPU  Descript...  User Name                  Priority  Handles  Window...
System Idle Process          0      00                <access denied>            0         0
  System                     8      04                NT AUTHORITY\SYSTEM        8         197
    SMSS.EXE                 160    00   Windows ...  NT AUTHORITY\SYSTEM        11        33
      CSRSS.EXE              184    01   Client Ser...NT AUTHORITY\SYSTEM        13        972
      WINLOGON.EXE           204    06   Windows ...  NT AUTHORITY\SYSTEM        13        426
        SERVICES.EXE         232    11   Services ... NT AUTHORITY\SYSTEM        9         770
          svchost.exe        448    00   Generic ...  NT AUTHORITY\SYSTEM        8         420
            hpgs2wnf.exe     1928   00   hpgs2wnf     EXAMPLE\Administrator      8         86
            DLLHOST.EXE      2512   06   COM Surr...  NT AUTHORITY\SYSTEM        8         147
            DLLHOST.EXE      3896   00   COM Surr...  EXAMPLE\IWAM_WIN2KS...     8         156
          spoolsv.exe        400    00   Spooler S... NT AUTHORITY\SYSTEM        8         132
          msdtc.exe          680    00   MS DTC ...   NT AUTHORITY\SYSTEM        8         211
          swagent.exe        824    00   DataDire...  NT AUTHORITY\SYSTEM        8         205
          swstrtr.exe        844    00   DataDire...  NT AUTHORITY\SYSTEM        8         15
            swsoc.exe        852    01   DataDire...  NT AUTHORITY\SYSTEM        8         211

Handle  Type           Access      Name
0x1D8   Thread         0x001F03FF  DLLHOST.EXE(3896): 1704
0x1FC   Thread         0x001F03FF  DLLHOST.EXE(3896): 2192
0x204   Thread         0x001F03FF  DLLHOST.EXE(3896): 1908
0x21C   Thread         0x001F03FF  DLLHOST.EXE(3896): 696
0x228   Thread         0x001F03FF  DLLHOST.EXE(3896): 2780
0x22C   Thread         0x001F03FF  DLLHOST.EXE(3896): 2700
0x25C   Thread         0x001F03FF  DLLHOST.EXE(3896): 3672
0x260   Thread         0x001F03FF  DLLHOST.EXE(3896): 1936
0x264   Thread         0x001F03FF  DLLHOST.EXE(3896): 1936
0xE8    Thread         0x001F03FF  DLLHOST.EXE(3896): 1456
0x274   Token          0x0000000C  NT AUTHORITY\SYSTEM
0x38    WindowStation  0x000F037F  \Windows\WindowStations\Service-0x0-2a069$
0x40    WindowStation  0x000F037F  \Windows\WindowStations\Service-0x0-2a069$

DLLHOST.EXE pid: 3896                                  Refresh Rate: 1 second
```

Figure 6-2: Using Process Explorer to view tokens in a process. Note the different levels of access between the Administrator token and the user (primary token).

Getting a token from another process is simple: The kernel will give you the token of any process that is attached to a named pipe you created if you call `ImpersonateNamedPipeClient()`. Likewise you can impersonate remote DCE-RPC clients or any client that gives you a username and password.

For example, when a user connects to a Unix ftp server, that server is running as `root`, so it can use `setuid()` to change its user ID to whatever user the client authenticates as. With Windows, the user sends a username and password, and then the ftp server calls `LogonUser()`, which returns a new token. It then spawns a new thread and that thread calls `SetThreadToken(new_token)`. When that thread is finished serving the client, it calls `RevertToSelf()` and joins the threadpool or calls `ExitThread()` and disappears.

Think of this procedure as an opportunity for a hacker—in Unix when you've exploited an ftp server with a buffer overflow after authenticating, you cannot become root or any other user. In Windows, you will likely find tokens from all the users who have authenticated recently waiting in memory for you to grab them and use them. Of course, in many cases, the ftp server itself will be running as `SYSTEM`, and you can call `RevertToSelf()` to gain that privilege.

One common misunderstanding surrounds `CreateProcess()`. Unix hackers will often call `execve("/bin/sh")` as part of their shellcode, but under Windows, `CreateProcess()` uses the primary token as the token for the new process and uses the current thread token for all file access. This means that if the current primary token is of a lower access level than the token of the current thread, the new process may not be able to read or delete its own executable.

A good illustration of this quirk is what happens during an IIS attack. IIS's external components run inside processes whose primary tokens are `IUSR` or `IWAM` rather than `SYSTEM`. However, these processes often have threads that run inside them as `SYSTEM`. When an overflow gives hackers control of one of these threads and they download a file and `CreateProcess()` it they find themselves running as `IUSR` or `IWAM`, but the file is owned by `SYSTEM`.

If you ever find yourself in this situation you have two options: you can use `DuplicateTokenEx()` to generate a new primary token, which you can assign to a `CreateProcessAsUser()` call, or you can do all your work from within your current thread by loading a DLL directly into memory or by using a simple shellcode that does whatever you need from within the original process.

Exception Handling under Win32

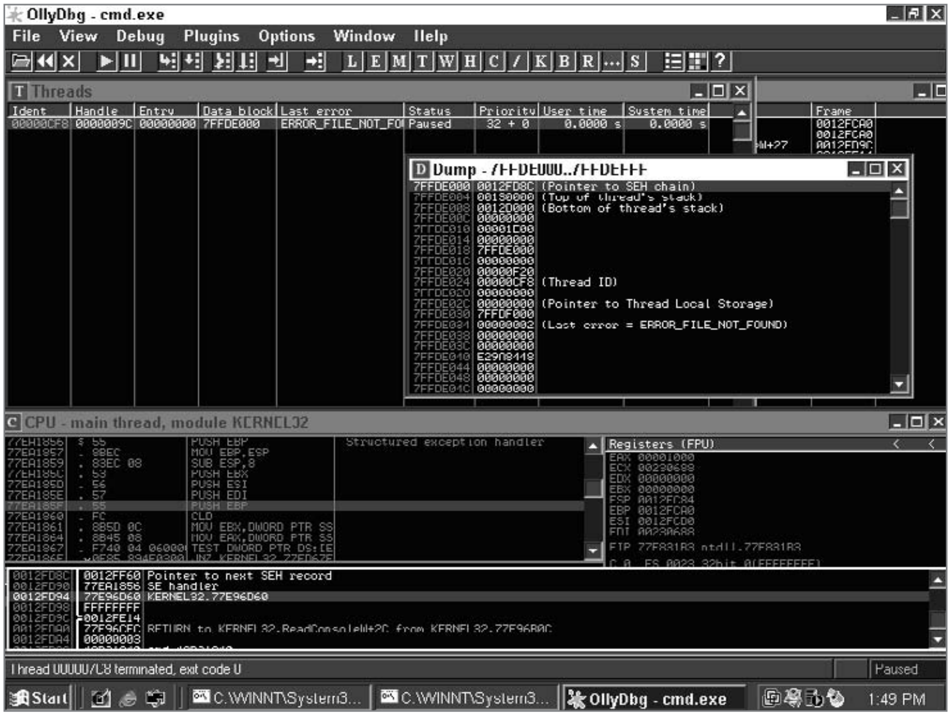
In Linux, exception handlers are typically global; in other words, per-process. You set an exception handler with the `signal()` system call, which gets called whenever an exception such as a segfault (or in Windows terminology, an AV) occurs. In Windows, that global handler (in `ntdll.dll`) catches any and all exceptions and then performs a fairly complex routine in order to determine to where it gives control. Because the programming model under Windows NT is thread-focused, the exception-handling model is also thread-focused.

Figure 6-3 may help explain exception handling under Windows NT.

As you can see in the figure, the `cmd.exe` process has two threads. The second thread's data block (which will be at `fs:[0]` while it is executing) has a pointer to a linked list (chain) of exception structures. The first element of that structure is the pointer to the next handler. The second element of that structure (Structured Exception Handler [SEH]) is a function pointer. As shown in Figure 6-3, the pointer to the next handler is set to –1, indicating no more handlers. However, if the first handler should choose not to handle a given exception, then the next handler (if there is one) would do it, and so on. If no handler wants to accept the exception, the default exception handler for the process handles it. Usually this results in the termination of the process.

As a hacker you should now see several ways to take control of this system via heap overflows or similar attacks that let you write a word into memory. You could certainly overwrite the pointer to the SEH chain. Every process in a Win32 application has an operating system supplied SEH. The SEH is responsible for displaying the error box that tells the user that the application has terminated. If you happen to have a debugger running, the SEH gives you an option to debug the application. Another possibility is to overwrite the function pointer for the handler on the stack, or you could overwrite the default exception handler.

On Windows XP you have another option: Vectored Exception Handling. Basically, it's just another linked list that the exception handling code in `ntdll.dll` checks first. So now you have a global variable that gets called on every exception—perfect for overwriting.


OllyDbg - cmd.exe
File View Debug Plugins Options Window Help
L E M T W H C / K B R ... S ?
Threads
Dump - 7FFDE000..7FFDEFFF
CPU - main thread, module KERNEL32
Registers (FPU)
Start
C:\WINNT\System3...
C:\WINNT\System3...
OllyDbg - cmd.exe
1:49 PM

```
Ident    Handle   Entry    Data block Last error         Status  Priority User time  System time
00000CF8 0000009C 00000000 7FFDE000   ERROR_FILE_NOT_FO  Paused   32 + 0   0.0000 s   0.0000 s
```

```
7FFDE000 0012FD8C (Pointer to SEH chain)
7FFDE004 00130000 (Top of thread's stack)
7FFDE008 0012D000 (Bottom of thread's stack)
7FFDE00C 00000000
7FFDE010 00001E00
7FFDE014 00000000
7FFDE018 7FFDE000
7FFDE01C 00000000
7FFDE020 00000F20
7FFDE024 00000CF8 (Thread ID)
7FFDE028 00000000
7FFDE02C 00000000 (Pointer to Thread Local Storage)
7FFDE030 7FFDF000
7FFDE034 00000002 (Last error = ERROR_FILE_NOT_FOUND)
7FFDE038 00000000
7FFDE03C 00000000
7FFDE040 E29D8418
7FFDE044 00000000
7FFDE048 00000000
7FFDE04C 00000000
```

```
77EA1856  $ 55           PUSH EBP                Structured exception handler
77EA1857  . 8BEC         MOV EBP,ESP
77EA1859  . 83EC 08      SUB ESP,8
77EA185C  . 53           PUSH EBX
77EA185D  . 56           PUSH ESI
77EA185E  . 57           PUSH EDI
77EA185F  . 55           PUSH EBP
77EA1860  . FC           CLD
77EA1861  . 8B5D 0C      MOV EBX,DWORD PTR SS
77EA1864  . 8B45 08      MOV EAX,DWORD PTR SS
77EA1867  . F740 04 06000 TEST DWORD PTR DS:[E
```

```
EAX 00001000
ECX 00230688
EDX 00000000
EBX 00000000
ESP 0012FC84
EBP 0012FCA0
ESI 0012FCD0
EDI 00230688
EIP 77F831B3 ntdll.77F831B3
```

```
0012FD8C  0012FF60 Pointer to next SEH record
0012FD90  77EA1856 SE handler
0012FD94  77E96D60 KERNEL32.77E96D60
0012FD98  FFFFFFFF
0012FD9C  0012FE14
0012FDA0  77E96CFC RETURN to KERNEL32.ReadConsoleW+2C from KERNEL32.77E96B0C
0012FDA4  00000003
```

Thread 00000CF8 terminated, exit code 0 Paused

Figure 6-3: OllyDbg nicely shows you how exception handling works in Windows NT.

Debugging Windows

You have basically three options for debugging Windows: the Microsoft tool chain, WinDbg; a kernel debugger, SoftICE; or OllyDbg. You can also use Visual Studio if you're so inclined.

Of these options, SoftICE is perhaps one of the oldest and most powerful. SoftICE features a macro language and can debug kernelspace. The downside of SoftICE is that it can be nearly impossible to install, and the GUI is somewhat old-school. Its main use is for debugging new device drivers. For a long time it was the only choice for a hacker, and so several good texts are available on how to use it. While debugging the kernel, SoftICE sets all the pages to writable; be aware of this fact if a kernel overflow you are working with seems to work only while SoftICE is enabled.

WinDbg can be set up to debug a kernel—although it requires a serial cable and another computer—but it can also be extremely good for debugging an overflow in user space. WinDbg has a primitive language, but the user interface is terrible—almost impossible to use quickly and accurately. Nevertheless, because Microsoft uses this debugger, it does have a few nifty advanced features, like automatic access to Microsoft's Symbol Server. CDB, the command-line equivalent of WinDbg, is extremely flexible and might be preferable for those addicted to the command line.

Just as SPIKE is the best fuzzer ever created, OllyDbg is the best debugger ever created. It supports amazing features such as run-traces (which allow you to execute backward) memory searching, memory breakpoints (you can tell it to, for example, set a break every time someone accesses anything in `MSVCRT.DLL`'s global data space), smart data windows (such as the ones in Figure 6-3 displaying the thread structure), an assembler, a file patcher—basically everything you need. If OllyDbg doesn't support something you need, you can email the author and the next version probably will. Spend some time attaching to processes with OllyDbg, then fuzzing them with SPIKE and analyzing their exceptions. This will get you quickly familiar with OllyDbg's excellent GUI.

Bugs in Win32

There are many bugs in Win32, and many of these are undocumented and painfully discovered by people writing shellcode. For example, `LoadLibraryA()`, which loads a DLL into memory, will fail if a period is in the `PATH` and the machine has not been patched for this particular bug. The WinSock routines will fail if the stack is not word aligned. Various other APIs are poorly documented on MSDN, if at all.

The bottom line is: When your shellcode is not working, the reason could quite possibly be a bug in Windows, and you might have to simply work around it.

Writing Windows Shellcode

Writing reliable Windows shellcode was for a long time a somewhat secret affair. The problem is that, unlike in Unix shellcode, you don't have system calls with a known API. Instead, the process has loaded function pointers to external functions such as `CreateProcess()` or `ReadFile()` into various places in memory. But you, the attacker, don't know where in memory these happen to be. Early shellcode just assumed they were in a certain place or guessed that they were in one of a few places. But this means that every time you create an exploit, you must version it across several different service packs or executables.

The trick to writing reliable and reusable shellcode is that Windows stores a pointer to the process environment block at a known location: `FS:[0x30]`. That plus `0xc` is the load order module list pointer. Now, you have a linked list of modules you can traverse to look for `kernel32.dll`. From that you can find `LoadLibraryA()` and `GetProcAddress()`, which will allow you to load any needed DLLs and find the addresses of any other needed functions. You'll want to go back and reread the PE-COFF document from Microsoft's shellcode to do this.

This technique tends to result in large shellcode because of its complexity. That said, in recent years several techniques have evolved to make it smaller, including innovative hashing methods. In a paper published in 2005, Dafydd Stuttard of NGS documented a 191-byte shell-binding shellcode—with no null bytes—that uses several cunning tricks to make the code smaller including using an 8-bit hash of the required function names.

There is, of course, another way. Various Chinese hackers have been writing shellcode that hunts through memory for `kernel32` by setting an exception handler. See various NSFOCUS exploits for this technique put into practice against IIS.

Even this shellcode can be fairly large. Therefore, CANVAS uses a separate shellcode, which is 150 bytes encoded using CANVAS's chunked additive encoder (similar to an XOR encoder/decoder but using `addl` instead of `xorl`), which simply uses exception handling to hunt through all the process memory for another set of shellcode prefixed with 8 bytes of tag value. This shellcode has proven to be highly reliable, and because you can put your main payload anywhere in memory, you don't have to worry about space restrictions.

A Hacker's Guide to the Win32 API

`VirtualProtect()`—Sets the access control to a page of memory. Useful for changing `.text` segments to `+w` so that you can modify functions.

`SetDefaultExceptionHandler`—Disassemble this to find the global exception handler location for a given service pack.

`TlsSetValue()/TlsGetValue()`—Thread Local Storage is a space that each thread can use to store thread-specific variables (other than the stack or heap). Sometimes valuable pointers that your shellcode may want to ravage are located here.

`WSASocket()`—Calling `WSASocket()` instead of `socket()` sets up a socket you can use directly as standard in or standard out. This technique can be used to make smaller shellcode if you're using shellcode that spawns a `cmd.exe`. (The problem in socket handles created with `socket()` is in the `SO_OPENTYPE` attribute.)

A Windows Family Tree from the Hacker's Perspective

Win9X/ME

- No user or security infrastructure (largely obsolete).

WinNT

- Hugely buggy RPC libraries make owning RPC services easy—RPC data structures are not verified by default the way they are in Win2K, so almost any bad data will make them crash.
- Doesn't support some NTLMv2 and other authentication options, making sniffing nicer.
- IIS 4.0 runs entirely as system and doesn't restart after it crashes.

Win2K

- NTLMv2 makes headway among entirely Win2K installation bases.
- RPC libraries much less buggy than NT 4.0 (which isn't saying much).
- SP4—Exception registers are cleared.
- IIS 5.0 runs as system, but most URL handlers don't run as system (with the exception of FrontPage, WebDav, and the like).

Win XP

- Addition of Vectored Exception Handling makes things easier for heap overflows.
- SP1—Exception registers are cleared.
- IIS 5.1—URLs are limited to a reasonable size.
- SP2 introduces firewall, heavily modifies RPC, introduces Data Execution Prevention (DEP), SafeSEH makes exploiting exception handlers harder, various other miscellaneous security improvements.

Windows 2003 Server

- Entire OS compiled with stack canary, including kernel.
- Parts of IIS moved into the kernel.
- IIS 6.0 still written in C++, now runs under an entirely different setup with a management process and a bunch of managed processes, each of which can serve port 80/443 from particular URLs and virtual hosts.
- Can finally detach from a process without it crashing. In previous versions of Win32, if you attached to a process with the debugger, detaching would forcefully kill it. This was useful sometimes, but mostly just annoying.

Windows Vista

- Everything compiled with a modified, better version of the /GS stack canary.
- ASLR (Address Space Layout Randomization) makes most exploits slightly harder; can be a serious difficulty when combined with DEP.
- Firewall now filters outbound traffic.

Conclusion

In this chapter, you learned the basic differences between exploitation on Linux/Unix and Windows. The same high-level concepts such as syscalls and process memory are present on Windows, but from a hacker's point of view, the implementation is grossly different. Armed with your knowledge of exploitation on Windows, you will be able to proceed to the next chapters, which cover Windows hacking in detail.