# Attacking the Windows Kernel

Jonathan Lindsay (john-lindsay@ngssoftware.com)

An NGSSoftware Insight Security Research (NISR) Publication



http://www.ngssoftware.com

## Abstract

Most modern processors provide a supervisor mode that is intended to run privileged operating system services that provide resource management transparently or otherwise to non-privileged code. Although a lot of research has been conducted into exploiting bugs in user mode code for privilege escalation within the operating system defined boundaries as well as what can be done if one has arbitrary supervisor access (typically related to modern root kit work), not a great deal of research has been done on the interface between supervisor and non-supervisor, and potential routes from one to the other.

The biggest problem arises when trying to protect the kernel from itself - for example, under the IA32 architecture implementation of Windows, the distinction between user mode and kernel mode from the user mode perspective is easily enforced through hardware based protection. However, as the kernel is running as supervisor, how does the kernel make distinctions between what it should be accessing? This would be irrelevant if the supervisor was not exposed to interaction with supervisee; but that would defeat the purpose of having a kernel.

This paper is focused on Windows and the Intel Architecture, and will briefly outline the current supervisor boundaries provided. Different attack vectors, along with relevant examples, will be provided to demonstrate how to attack the supervisor from the perspective of the supervised, as well as an outline of what possible architectures could be used to mitigate such attacks, such as the research operating system Singularity.

**Keywords**: kernel security, windows nt, reverse engineering, windows driver framework, virtualization

# Contents



# Appendices

# 1 – Introduction

On any operating system that provides a distinction between user and supervisor mode functionality, the kernel will provide a powerful, yet restricted, set of services to manage hardware resources as well as potentially perform internal operating system management.

Modern kernels, for example Minix [17], Singularity [14], or Symbian [26], tend to only provide hardware management from supervisor mode, with the remaining services being provided by subsystems running in user mode. Such microkernels are written in order to improve security by reducing the attack surface provided by the supervisor code, as well as allow greater flexibility for software design by reducing the architectural constraints placed on developers.

This is in distinction to monolithic kernels such as UNIX [27] and Windows NT [19], though NT is often referred to as a 'hybrid kernel'. Monolithic kernels run entirely in supervisor mode, and provide all operating system management functionality at that level. Windows NT could be described as a 'hybrid kernel' as some components, though strongly tied to the kernel (especially the user mode subsystems such as CSRSS and LSASS), are run in user mode. Though this can be utilized to provide a broad framework for kernel development thus potentially speeding up kernel module development, it does greatly increase the potential attack surface available to a malicious user to abuse the code running in supervisor mode.

Given this, what can we hope to gain from either modifying kernel data, or running arbitrary code in supervisor mode? This is almost entirely beyond the scope of this paper, but not only has a great deal of research has been conducted into ([5], [28]) root kit technology, but such attack vectors could be leveraged to subvert digital rights management systems, perform kernel hacks such as replacing the Blue Screen of Death with a bitmap, or otherwise enhance the functionality of the kernel.

This paper is focused on the Windows NT architecture and the Intel architecture [7]; as such, the focus will be on what vectors there are for attacking the kernel, what tools and methods are available to investigate any potential attacks, and what mechanisms are in place, or could be put in place, to try and prevent them.

# 2 – Attack vectors

Given the architectural structure of Windows NT (outlined in Appendix A), what are the potential methods that could be employed to switch from user to supervisor mode?

## 2.1 – Direct from user mode

Although not an obvious attack, there are potential avenues from user mode to supervisor mode that do not involve any operating system APIs or interfaces.

Firstly, the operating system may have been designed in such a way that it is possible to do this by not utilizing the protection functionality provided by the hardware. The best example of this is Windows 95, the first operating system written by Microsoft to take advantage of the protected mode functionality provided by the contemporary x86 processors. It does not protect the descriptor tables, unlike NT, and they are writeable and located within the user portion of the address space. Thus, any user mode code that wishes to execute arbitrary code as supervisor just needs to add an entry to the LDT, GDT, or IDT and then execute the requisite interrupt or far call.

There are no publicly known issues like this, such as supervisor memory and control structures or hardware protection functionality being accessible to user mode, for Windows NT.

Aside from that, a theoretically possible approach could be provided by bugs in the CPU itself. Both Intel ([8] for example) and AMD ([1] for example) now publish lists of CPU errata for their processor lines, though there is no publicly known way of leveraging any of these bugs to allow escalation from user to supervisor. Even so, despite the vagueness of the descriptions for the errata, there are some interesting possibilities for certain processors:

**Erratum 64 for AMD Athlon 64 and Opteron processors - Real Mode RDPMC with Illegal ECX May Cause Unpredictable Operation**

> "However, if the RDPMC instruction is executed in real mode with a specific illegal value of ECX = 9, then the processor may incorrectly enter the GP fault handler as if it were in 32-bit mode."

The author has not currently had the resources to experiment with this erratum, though it does provide some interesting possibilities: is the processor just running in 32-bit real mode, or is it running in protected mode? If it is running in protected mode, which segment is it running in, thus what is the Current Privilege Level of the executing code?

**Erratum AK92 for Intel® Core™2 Extreme Quad-Core Processors - Invalid Instructions May Lead to Unexpected Behavior**

> "Invalid instructions due to undefined opcodes or instructions exceeding the maximum instruction length (due to redundant prefixes placed before the instruction) may lead, under complex circumstances, to unexpected behavior."

A typically vague description, though the author presumes that instructions are subsequently not decoded as expected due to the instruction pointer no longer pointing to the expected instruction boundaries, though this has not been verified

There are several other similar errata for both AMD and Intel processor lines, usually involving unexpected processor behaviour or unexpected memory access. There is obviously the potential for information disclosure in such cases, but the conditions necessary to trigger such situations in

the errata studied thus far are either undefined, or require the processor to be executing in supervisor mode.

## 2.2 – Public APIs

Beyond the ABI interfaces outlined in Appendix A, which are supposed to be reserved for internal operating system use, the NT OS must provide APIs that provide higher level programmatic access to the kernel, at the very least to load and unload kernel modules.

As such APIs allow a user to load arbitrary code to be executed in supervisor mode, such APIs should have some required privilege level necessary for them to be invoked – otherwise any unprivileged user can take full control of the machine. Under NT, driver loading functionality is restricted to users with *SeLoadDriverPrivilege* (administrators and system processes); a user with *SeDebugPrivilege* may then be able to access such functionality by injecting code into a system or administrator process.

The documented Win32 APIs involved are *CreateService*, *StartService*, and *StopService*; though these will not necessarily involve the loading of a kernel module. The documented NT APIs provided are *ZwLoadDriver* and *ZwUnloadDriver* (documented in the Driver Development Kit), and are used internally by the Service Control Manager to load and unload kernel modules for kernel mode services.

An additional mechanism introduced for 64-bit Vista is mandatory driver signing [13], so that unsigned kernel modules cannot be loaded into the kernel unless signing is disabled.

Beyond that, any API that results in information being passed over to supervisor code for processing creates a potential attack vector. Thus, device drivers may increase the attack surface through kernel mode callbacks, or interfaces exposed to user mode (for example through *DeviceIoControl*, *FilterSendMessage*, or *ExtEscape*). This will be discussed further below.

## 2.3 – Undocumented APIs

A large number of symbols exported by both user mode and kernel mode libraries are undocumented, and designed for internal operating system use. Consequently, there is always the possibility of an undocumented function or object providing greater access to kernel mode functionality and structures.

The three most prominent examples are *ZwSystemDebugControl*, *ZwSetSystemInformation* and the *PhysicalMemory* device interface; over recent service packs and versions of the NT OS, such as service pack 1 for Windows Server 2003 and Windows Vista, have attempted to restrict access to these APIs. Whereas previously usage of *ZwSystemDebugControl*, which can be used to read and write to arbitrary kernel memory amongst other things, was only restricted to users with *SeDebugPrivilege*, Vista checks that the OS has been booted with the kernel debugger enabled (checking *KdPitchDebugger*), before allowing access, with only the creation of a triage dump being available otherwise.

Debugging functionality provided by the internal kernel debugger has also been restricted, and is no longer directly available to user mode processes. The debugging functionality has been moved into *KdSystemDebugControl* which is exported by the NT kernel. This functionality is made available to WinDbg for local kernel debugging on Vista via a device driver that WinDbg drops and loads, otherwise it relies on the older *ZwSystemDebugControl*.

Calling *ZwSetSystemInformation* using a value of 26 for *SystemInformationClass*, which has most popularly been referred to as *SystemLoadAndCallImage*, *SystemLoadImage*, and *SystemLoadGdiDriverInformation*, allows a user with the *SeLoadDriverPrivilege* to load a driver through an alternate means than the *ZwLoadDriver* API. The final symbolic name suggests the use of this API to provide support for loading kernel components of display drivers.

Physical memory is accessible via the `'Device\PhysicalMemory'` file object; until Windows Server 2003 Service Pack 1 this was accessible from user mode, allowing a user mode application to map in physical memory, manually do the translation from virtual to physical address, and then modify arbitrary areas of kernel memory (such as descriptor tables).

Although such undocumented functionality is available more freely in older versions of NT, the coding best practices introduced under the Secure Windows Initiative at Microsoft suggest that the addition of such APIs in the future is unlikely without putting equivalent restrictions on their access. However, additional exported symbols and interfaces that are not documented in new NT OS releases are always worth investigation.

## 2.4 – Architectural flaws

A distinction will be made between architectural flaws, which will typically involve the breaking of defined security layers, and bugs in the code (discussed in the next section) due to the nature of the investigation used to find them, and the potential differences in exploitation.

A typical example of an architectural flaw would be an antivirus disinfection engine that contained functionality to modify the contents of arbitrary memory locations. This would be particularly useful for modifying malicious code to disable a watchdog function, or make alterations to data structures that have been modified by a kernel mode root kit. However, if no usage restrictions are placed on the access to this functionality, such as that it can only be used by a specific user, or requires *SeDebugPrivilege* or *SeTcbPrivilege*, then an unprivileged user can use it to modify arbitrary areas of kernel memory, and either modify kernel mode structures directly from user mode such as *PsLoadedModuleList*, or add an entry to a descriptor table and then run arbitrary code in kernel mode.

Architectural issues tend to be easier to recognize both in source code and in disassembled binaries, and exploitation tends to be easier than for bugs as this is intentional functionality provided by the kernel code, rather than a buffer overflow, for example; on the other hand, architectural flaws can be more difficult to fix – for the example above, either the AV disinfection must add suitable access restrictions (which is still fairly trivial), or the disinfection process itself must be redesigned and reimplemented.

## 2.5 – Bugs and their exploitation

Any situation where arbitrary user controlled input is processed by trusted code presents the possibility of the input being malformed in a specific way such that the code behaves in ways that were not intended (in distinction to architectural flaws). Any supervisor mode interface exposed to user mode must have robust input validation.

The NT kernel has only 24k reserved for the kernel stack, making exploiting buffer overflows more difficult due to the lack of space, and the decreased likelihood of arbitrary size buffers being used on the stack. Instead, it is more likely for pool memory to be allocated. This raises an interesting problem – pool memory is used to store a variety of object data, as well as IRPs and data buffers for the various kernel modules loaded and executing, thus providing a wide variety of potential pointers to overwrite, such as in a *DriverObject*.

However, the author is not aware of any work similar to [25] for kernel pool memory allocation to try and deterministically control where any allocated pool memory is placed and what data and objects are stored after it to allow a more controlled attempt at a function pointer overwrite; additionally, the modification of arbitrary kernel data within pool memory may leave the operating system in such an unstable state that the kernel will crash before the overwritten pointer is dereferenced. However, if such an attack triggering a bugcheck is available via a remote interface then it still allows a denial-of-service attack.

Bugs that are easier to exploit are those that allow controllable data overwrites and pointer derefences. As memory can be allocated at a virtual address of 0 in Windows NT by using *ZwAllocateVirtualMemory*, either dereferencing a null pointer or overwriting a pointer with null can easily allow arbitrary code to be executed. Additional potential targets are entries in the descriptor tables, such as interrupt gate entries in the IDT:

```
typedef struct _KIDTENTRY {
  USHORT  Offset
  USHORT  Selector
  USHORT  Access
  USHORT  ExtendedOffset
} KIDTENTRY, *PKIDTENTRY;
```

*ExtendedOffset* is the high word of the address of the entry point for the given interrupt. Zeroing this will cause the interrupt to start executing somewhere within the first 64Kb of memory in the current process. Seeing as it is easy to determine where the IDT is located from user mode, as the `sidt` instruction is neither protected nor privileged, that such interrupts as 4 (interrupt on overflow) and 5 (bound error) being rarely if ever used under normal operating system usage, and that interrupt 4 has a DPL of 3 so can be accessed from user mode without generating an access violation, modifying the IDT presents a promising target under systems where the IDT is not protected.

Other pointers within kernel modules can be deterministically located by loading the target module into a user mode process with *LoadLibraryEx* and specifying a flag of DONT_RESOLVE_DLL_REFERENCES so none of the module code is executed. Entry points can then be heuristically scanned for references to target pointers. Obviously both this and the above technique require that code already be running on the target machine. A remote exploit allowing direct arbitrary code execution in the kernel would be difficult to make work, especially reliably, though not impossible.

It is also worth bearing in mind that even though the NT kernel itself has received a fairly thorough examination, it is not itself without bugs that may be directly exposed to user mode applications, or indirectly via some other kernel code. Four relevant examples have been provided in Appendix C.

## 2.6 – Subverting operating system initialization

There has been recent work on subverting the boot process of the NT OS from a variety of different vectors ([18], [9], [4]), mostly related to root kit research, so this topic will only be briefly discussed.

Although the attacks may be complicated, the essence of the attack is simple – during the boot process, before the operating system has at least partially initialized itself, if at any point the control flow can be changed to that of user controlled code then the user has arbitrary and unrestricted access to the operating system before it has had a chance to initialize and properly segregate itself from arbitrary user code. Consequently, as in eEye's BootRoot, arbitrary modifications can be made to the operating system, as also outlined in an attack against PatchGuard on 64-bit versions of NT [23].

## 2.7 – Modifying kernel modules on disk

The author is unaware of any malware, or malware related research, that has investigated either modifying kernel modules to allow kernel mode DLL injection or the infection of kernel modules.

Although there are measures to avoid the tampering with of kernel modules, such as driver signing, checksum verification for the module binary, and Windows File Protection and Windows Resource Protection, all these technologies can be circumvented in some manner (though

administrator privileges, and a reboot, may be required). So, presuming a kernel module has been modified, and this may include the display drivers in the Windows System folder, how can this be leveraged to run arbitrary user code?

Firstly, the standard viral infection methods are available. Though the driver's entry point is called when it is loaded, thus allowing entry point infection, it is much harder to programmatically determine what areas of the code will be executed and when, thus making mid-infecting techniques more difficult. The only real difference between this situation and the one for user mode viruses is in the APIs available to the viral code; though kernel mode viral code has a much wider array of potential targets to hook file system operations to infect files, or use root kit techniques.

Beyond that, kernel modules are standard PE files with in Import Directory (if they import any symbols). Although drivers typically only link with the NT kernel, or with win32k or videoprt for display drivers and their miniports, the NT kernel supports 'kernel mode DLLs' more formally referred to as export drivers. Normal drivers can be statically linked to export drivers, and the dependencies of any driver are recursively resolved during the image loading process. The entry point of an export driver is not executed, and as such it does not need to contain any code and is only needed as the build process requires that function to be present. However, if it exports *DllInitialize* then that will be called to allow whatever initialization the driver requires to be carried out. The unload routine is *DllUnload*.

Thus standard PE file modification techniques can be used to modify a device driver's Import Directory to add an entry for a user-created export driver, that will then be loaded and *DllInitialize* executed. However, the standard restrictions on kernel modules still apply for export drivers, so on 64-bit Windows Vista, it will need to be signed.

## 2.8 – Hardware

As the majority of kernel mode drivers are designed to actually allow the interfacing of user mode or other kernel mode code to actual hardware, then the hardware itself provides a potential attack vector. The Windows Driver Kit specifies the Windows Driver Framework as being suitable for the following device classes:

- IEEE 1394 client drivers
- ISA, PCI, PCMCIA, and secure digital (SD) devices
- NDIS protocol drivers
- NDIS WDM drivers
- SoftModem drivers
- Storage class drivers and filter drivers
- Transport driver interface (TDI) client drivers
- USB client drivers
- Winsock client drivers

The most interesting drivers are those that drive hardware that allows remote access, so particularly NDIS, modem, TDI and Winsock client drivers.

The primary methods for a driver to communicate with the underlying hardware are via port I/O and through memory mapped I/O. The Plug and Play manager enumerates devices and then calls the appropriate *AddDevice* or *EvtDriverDeviceAdd* routine in the driver that will drive the hardware; though the system libraries provided for video port, SCSI port, and NDIS drivers will perform the initialization. Depending upon the hardware resource to be driven, the resource can either be accessed by mapping the hardware on a given bus into the virtual address space by using *MmMapIoSpace* with the physical address of the resource provided by the PnP manager, or can be accessed programmatically with functions which are just wrappers for `in` and `out`.

Communications from hardware back to the given driver are carried out via interrupt, allowing the hardware to signal that it has entered a specific state (such as having processed a request). Interrupts are serviced by a driver provided Interrupt Service Routine. Windows NT abstracts from the hardware interrupt architecture provided with x86 and x64 processors and allows multiple ISRs to be chained to the same internal interrupt vector. Consequently, direct modification of the IDT is ill-advised for general purpose driver coding. ISRs are registered using *IoConnectInterrupt*, *IoConnectInterruptEx* or *WdfInterruptCreate*. Thus, anyone auditing either source code or a compiled binary can quickly locate the entry points into the binary from a hardware point of view, and then trace through to see how the data is processed.

In cases where hardware is being used as the attack vector, fuzzing can be of great use; however, the appropriate hardware needs to be present to provide the vector. Consequently, a virtual machine is of little use in such testing, and fuzzing on a non-virtual machine raises the issue of automating the process, particularly if bugs are found which will result in a bugcheck.

# 3 – Tools for the job

Given the above outline of the relevant kernel architecture, and the outlined potential attack vectors, the following section will describe tools useful in the assessment of kernel mode code. This is far from a complete list, and serves only to highlight useful and tools and techniques.

## 3.1 – Static analysis

Depending on what is available for scrutiny, there are at least one, possibly two essential resources for static analysis of kernel mode code. With access to source code then besides a reasonable IDE or source browser all that is needed is the current version of the Windows Driver Kit, not only for the documentation but the header files and the function prototypes, structures, and enumerations contained therein.

### 3.1.1 – Static Driver Verifier

An interesting tool from Microsoft that can also be used in this situation is the Static Driver Verifier [16] – with source and a WDM compliant driver, SDV can be used to execute the code and test for a set of pre-defined rules that can check code correctness with respect to seven different categories (such as IRP handling, IRQ levels, PnP functions). The SDV is state based, and rules it uses may include one or more of the following: state variables, actions that can change state, conditional expressions, and assignment statements.

This allows a quick assessment of certain aspects of the correctness of the driver, though there are limitations. Although some complex conditions are tested for, including certain race conditions, recursion, and synchronization issues, there is no rule-based testing for the more sophisticated issues related to data processing (presumably not least because of the intractable nature of such a task). Consequently, bugs leading to arbitrary code execution will require a more typical code review.

### 3.1.2 – PREFast

It is also worth considering PREFast [15], another static analysis tool provided by Microsoft for driver development. It also requires source code to be available, but detects different classes of errors than the Static Driver Verifier, such as memory leakage, dereferencing NULL pointers, the use of uninitialized variables, buffer overflows, excessive kernel stack use, type checking and type mismatches, as well as certain code correctness rules. Given the types of issues identified by PREFast, any coding errors picked up are more likely to lead to arbitrary code execution than those found by SDV. However, it is still not a fool-proof method to find all potential issues, and can be prone to false positives.

### 3.1.3 – Disassembler

It may be a case of black box research into an attackable driver exposed to the outside world via some piece of hardware, such as a Bluetooth driver. In these situations, there are two options – fuzz, or fire up a disassembler. Fuzzing will be dealt with in the following subsection. The author will make no recommendations as to which disassembler to use, and will leave that entirely up to the discretion of the reader.

Given the outline architectural information above, and some good reference material ([22], [11]), then it is not difficult to load up the binary and take a look. Depending on the version of the DDK/WDK the driver was built with (presuming it was) the entry point code will differ. For binaries compiled with older versions of the DDK, or with /GS turned off, the driver will start at user code; for the Windows 2003 DDK and the WDK /GS is turned on by default, so some stub code will be present at the entry point to set up stack protection before jumping to the user code. For the WDK, having built a KMDF driver, a library code wrapper is present that will initialize WDF and then call either the /GS stub (if enabled), or directly to the user code. Symbolically, these different entry points are referred to as DriverEntry, GsDriverEntry and FxDriverEntry.

As a quick example, the following is the start of the *EvtIoDeviceControl* routine from the RamDisk driver from the Kernel Mode Driver Framework examples from version 6000 of the WDK. This routine is added to an I/O queue created in the driver's *EvtDeviceAdd* routine.

```
RamDiskEvtIoDeviceControl proc near

 var_8         = dword ptr -8
 var_4         = dword ptr -4
 Queue         = dword ptr  8
 Request       = dword ptr  0Ch
 IoControlCode = dword ptr  18h

         mov     edi, edi
         push    ebp
         mov     ebp, esp
         push    ecx
         push    ecx
         mov     edx, [ebp+ Queue]
         mov     ecx, WdfDriverGlobals
         and     [ebp+var_4], 0
         push    ebx
         push    esi
         push    edi
         push    off_10AA0
         mov     edi, 0C0000010h
         call    WdfFunctions.WdfObjectGetTypedContextWorker
         mov     esi, [eax]
         mov     eax, [ebp+IoControlCode]
         sub     eax, 70000h                             ; Switch the IOCTL
         jz      loc_1055E
         sub     eax, 24h
         jz      loc_1058D
         sub     eax, 3FE0h
         jz      short loc_104DE                         ; Check for 0x704004
         sub     eax, 7FCh
         jz      loc_1058D
         jmp     loc_1058F                               ; IOCTL not handled, complete the request
 ; ---------------------------------------------------------------------------
 loc_104DE:
         mov     ebx, [esi]                              ; Process IOCTL 0x704004
         push    20h
         pop     eax
         lea     ecx, [ebp+var_8]
         push    ecx
         lea     ecx, [ebp+Queue]
         push    ecx
         push    eax
         push    [ebp+Request]
         mov     [ebp+var_4], eax
         push    WdfDriverGlobals
         call    WdfFunctions.WdfRequestRetrieveOutputBuffer ; Get the output buffer
         mov     edi, eax
```

A more detailed example is provided in Appendix B.

## 3.2 – Dynamic analysis

Dynamic analysis can be considerably less of an effort than reverse engineering a driver for bugs, and can easily help pick up issues that can be missed through static analysis. However, there is still a lot of work that could be done toward improving and automating dynamic code analysis.

### 3.2.1 - WinDbg

Whether just debugging kernel mode code, or analyzing the crash dump from a successful attempt at fuzzing some code, the most useful kernel debugger for Windows NT is Microsoft's WinDbg. There are other kernel mode debuggers available such as SoftICE, Syser, and RR0D, though they unsurprisingly suffer compatibility issues as well as having significantly less functionality when compared to WinDbg.

Although WinDbg does not necessarily have the most user friendly interface, provided with symbols the local kernel debugging option is useful for gaining a quick look at the current kernel configuration and internal data structures; remote kernel debugging is fairly quick and convenient

over FireWire or USB without the stability issues raised by genuine local kernel debugging; debugging kernel code that does not require associated hardware is made considerably more convenient by using a named pipe for the debugger connection and debugging a virtual machine running in either Virtual PC or VMWare; and finally, the automated crash dump analysis created by entering `!analyze -v` can greatly speed up the assessment of kernel mode crashes and their exploitability.

There is a great deal of documentation on WinDbg, not just within the install package but also on the associated section of Microsoft's website [12].

**3.2.2 – The Driver Verifier**

The Driver Verifier is functionality built in to the NT kernel to enable stress testing of driver code in a live environment. It is similar in principle to the user mode Application Verifier, though whereas that is to some extent configurable with respect to what functions are detoured, and what the detours perform, the Driver Verifier is hard coded to detour specific functions to other predetermined functions that will allow the verifier to provide a hostile environment to the target driver.

Driver verification is managed through *ZwSetSystemInformation* with system information classes 40 (add driver) and 41 (remove driver) under Windows XP SP2. Internally, provided the user has *SeDebugPrivilege* then *MmAddVerifierEntry* or *MmRemoveVerifierEntry* are called as appropriate, whereby the name of the driver to be verified is added to or removed from a linked list named *MiSuspectDriverList* – drivers are only added to this list if they have not already been added and have not already been loaded.

When a driver is loaded, *MiApplyDriverVerifier* is called from *MmLoadSystemImage*; this checks if NT is configured to verify random drivers, all drivers, or only drivers on the suspect driver list. For a driver that is to be verified, the loader checks two lists of functions, *MiVerifierThunks* and *MiVerifierPoolThunks*, and if any of these functions are discovered whilst performing load time linking, then the alternate version that provides the spurious input is linked to instead. Currently functions related to event handling, the acquisition of mutexes and resources, memory mapping and locking, IRQ levels, synchronization, certain I/O manager functions, file access, and memory allocation and de-allocation, are targeted by the verifier. This allows the verifier to do things such as simulate low-memory availability conditions, check for deadlocks on synchronization objects, or whether code is being called at the appropriate IRQL.

Although this type of verification is useful for development, as with the Static Driver Verifier the potential results may not necessarily be helpful beyond finding potential denial-of-services.

**3.2.3 – Miscellaneous tools**

In addition to the more in-depth analysis tools outlined above, there are several small utilities targeted at providing technical information about operating system objects, and potentially manipulating or interacting with them.

WinObj [21] allows a user to browse the kernel object directory, which contains not only folders for device and driver objects, thus allowing the assessment of the Access Control Lists for given devices to determine who can open them, but also contains object information for session specific devices, symbolic links, and file system filters. The BaseNamedObjects subdirectory also lists named jobs, sections, events, semaphores, mutants, and symbolic links. Process Explorer, also from Sysinternals, can be used to provide a quick overview of how the user mode components of an application communicate with the kernel mode components by listing what open file handles, sections, tokens, and mutants, along with their associated ACLs.

NtDispatchPoints [3] is a quick tool to check what IRP functions some device, found for example with WinObj, supports. This can help narrow down areas to target in a binary, but will not help finding all the potential entry points into the kernel mode code. A similarly interesting tool that can

be useful for quickly finding potential attack vectors is the RootKit Hook Analyzer [20] (resplendence) that will determine of there are any system service table hooks, and can fuzz them if asked to. However, the author is unaware of any tools that can find deeper hooks within the kernel and fuzz them, such as hooks targeted at the dispatch functions of a *DriverObject*.

There is certainly a great deal of scope for other tools in this area to help automate the determination of potential attack vectors as well as their investigation.

# 4 – Defensive measures

Relying on the traditional monolithic architecture for an operating system kernel with the kernel mapped into every process's address space will always raise security issues; even though the parameter validation of the NT kernel itself has greatly improved since the time of NtCrash, and APIs have either been restricted in their use or their functionality, the greatest security concern is still raised by third party code that either exposes kernel mode functionality and data to an unprivileged user or opens up security holes by being poorly coded.

Applying an ACL to a device and running as a service under a different user certainly provides one method for restricting access, as does checking for privileges such as *SeDebugPrivilege* or *SeTcbPrivilege*; all of which uses the current NT architecture to enhance security. This does not solve the issue of parameter validation, which is a particularly thorny issue in the kernel – probably the best way to enhance parameter validation would be to be aware of the type of security issues involved, such as buffer overflows and arbitrary memory writes, and then design the code to be secure from the beginning; this should in principle minimize the appearance of pointers and buffer lengths being provided from user mode, and hopefully reduce the appearance of bugs due to unforeseen boundary conditions.

Microsoft has attempted to exert greater control over approved third party code by introducing Kernel Mode Code Signing in Windows Vista that mandates digitally signed kernel code, and has also introduced PatchGuard on 64-bit versions of Windows to limit the modification of kernel code and data structures. This does not prevent hooking of certain kernel data structures though, most importantly the objects stored in the object directory (such as device, driver, or port objects) which still allows kernel functionality to be hooked and file system or network requests and responses to be modified as desired.

Further than this, recent processor hardware developments have introduced virtualization technology to aid the execution of virtual machine technology. Running the most privileged part of the operating system as the hypervisor (above the supervisor) could allow the operating system to exert control over the behaviour of the supervisor; much has been said on this topic with respect to malware. Though perhaps tempting, a much simpler solution is provided by employing a microkernel (especially as a hypervisor may just be a microkernel beneath the supervisor).

In addition, the approach taken by the team developing the research OS Singularity could be taken – Singularity is written largely in an extension of C# (named Sing#) as a strongly typed microkernel (though some of the kernel code is not type-safe). All code runs in supervisor mode, removing the need to transfer buffers between user and kernel mode thus improving speed and security, and uses strong typing and code verification to try and ensure code correctness. Isolation between processes is maintained by software and not by hardware, such as having a different virtual address space or running in a different segment; communication is performed through contract-based channels; and all programs are manifest-based, thus allowing control of what code is executed within each process.

Although Singularity is currently only a research project within Microsoft, it and other microkernels being used or developed do indicate a desire to attempt to address some of the challenges in operating system design that have until recently been overlooked by the acceptance of the original OS development work carried out in the 1960's and 1970's.

# 5 – Further work

Manually auditing kernel code, especially in a black box situation, can be considerably time-consuming; there is a great deal of work that could be done to improve the automated assessment of kernel code.

## 5.1 – Fuzzing

As highlighted earlier, automated fuzzing of kernel code can be awkward due to a bugcheck halting the system, and then requiring further in-depth analysis to determine what occurred. Two fairly simple approaches could be used to help speed this process up – the use of a virtual machine, or the use of a hypervisor.

The essence of either of these methods is to restore the operating system to a reliable state before fuzzing (it is also possible to modify the kernel itself to do this upon a bugcheck and restore from a memory image stored on disk, though this would not be a very pleasant solution, especially as the operating system may have become so corrupt that it cannot perform this). Setting up something like VMWare to enable this should be fairly trivial, but raises the problem of assessing drivers for real hardware, which thus cannot be installed in the virtual machine. Writing a hypervisor would be an expensive and time consuming task.

Further than that, neither option helps with the location of entry points into the kernel, nor with any instrumentation to assess code coverage. Consequently, other solutions may prove more useful.

## 5.2 – Automated bug finding

Instead of relying on runtime analysis provided by fuzzing, it may be more productive to spend the time designing and implementing a static code analyzer. Either a source code analyzer (which would be language specific) or a binary analyzer (which would be architecture specific) could be useful.

The author has a partial implementation of an x86 emulator designed for static code analysis. Rather than provide full emulation as Bochs [2] does for example, the emulator models inputs and outputs and attempts to perform the inverse of the code correctness verification that can be attempted through the use of a formal language – assuming arbitrary input, what ‘classes’ of input are accepted by the target code, and what are the resulting code-paths that are executed? Dealing with this at a binary level can be simpler as no concept of type exists, and certain issues such as improper sign checking can be more easily discovered.

Schematically, any such tool along these lines could be though of as a ‘meta-fuzzer’ – all possible input is provided at the same time. Consequently, although the design and implementation would take longer than for a typical fuzzer, the results should be of a higher quality, with the added bonus of not needed to restart crashed process or analyze crash dumps.

The design of such a tool does raise important issues – if all potential input is provided at the same time, then in principle every conditional jump may or may not be taken (though code could be written whereby specific conditional jumps are never taken). To maximize code coverage, the state at every conditional jump would need to be stored and if necessary linked to a parent state (if it is in a subfunction, for example). For a moderately sized piece of code, the amount of space required to store these states would very quickly become very large.

This can be combated in two ways; firstly, a breadth-first based search could be used whereby subfunction calls are only followed to a certain depth, and their effect on the potential processor states available for processing assessed at a later time, thus allowing the narrowing down of areas of interest; secondly, a heuristic scan of the binary for areas that may be vulnerable, such as places in the code where unsafe string or memory operations are executed, and then a search

that does not require maintaining so much state information for code-paths to the specific areas in question.

This also raises the possibility of using artificial intelligence techniques, such as example-based training, to create software that can discover bugs via static analysis, though it is debatable how much the security industry would trust such a tool.

## 5.3 – Virtualization

One of the major issues with fuzzing kernel code is that causing bugchecks greatly slows down the entire process. Although reverse engineering has been highlighted above, this can be an extremely time consuming process. Bugchecks are generated when the operating system assesses that it has been corrupted or can no longer continue to function correctly, for whatever reason. The simple solution to this problem is to run kernel mode code in user mode instead of supervisor mode.

In principle there is no facet of the Windows NT architecture that prevents doing this, though it is not a trivial task. Essentially, the kernel execution environment must be ported over to user mode. The exported functions from the core kernel libraries (ntoskrnl, win32k, hal, and videoprt, depending on what is being assessed) need to be re-implemented where necessary in a manner that will function under user mode. The kernel exports a large number of runtime library functions that are also present in ntdll, and many of its exported functions can be called without any concern for whether the code is running in user or supervisor mode. There only remains to implement in a library those exported functions that are not compatible with user mode, and add any logging and tracing where necessary.

With this in place, any kernel module can then be linked to the appropriate libraries so it will work in user mode, and can then be fuzzed as normal. Logging functionality can be added to any replacement routines so the creation of callbacks, shared memory, interfaces, and so forth, can all be logged, and the given entry points then fuzzed. It would also be fairly simple to add the ability to determine IOCTLs supported by a driver in an initial scanning phase.

There still remain two problems with this solution – privileged and protected instructions, and drivers for real hardware. A generic exception handler, preferably a vectored exception handler as it is global to a process, could be used to trap all invalid instructions, as well as access violations, and process them as appropriate. Generally speaking, none of the protected or privileged instructions need to be executed when running kernel code in user mode, though if need be a driver could be written and used as a proxy for instructions such as `in` and `out`. Such a design would itself raise security concerns for flagrantly disregarding some of the coding principles discussed in this paper, but it would not be production code.

A similar approach can also be taken with hardware, whereby a driver acts as a proxy between the driver running in user mode and the real hardware. This would be a difficult to implement though, and less preferable perhaps than implementing a generic virtual device. As the relevant structures have either been described above, or are documented elsewhere, provided the relevant areas of the code are found and the type of hardware resource being made available is known then it would not be overly difficult to create virtual resources and then call the appropriate functions from the driver to register them, and then call the related functions for fuzzing. This description does underplay the design issues involved and the amount of work necessary to implement such a tool.

The author is currently working on both virtualization and automated static analysis.

# 6 – Conclusion

The monolithic architecture for operating system kernels that has been inherited raises security issues that may be best addressed by trying different architectures. For the current, and by all appearances future, versions of the Windows NT operating system these issues will remain and will require the usual vigilance with respect to parameter validation and the exposure of supervisor functionality to user mode.

With that in mind, and the effort that has been put into making the core components of the NT kernel more secure, the best targets by far will be third party drivers where there may be less of a concern over security as the vendors will most likely have received much less criticism over security issues than Microsoft traditionally has; and although developments like the Windows Driver Framework and the verification tools provided by Microsoft certainly make driver development easier and help to improve coding practice, they in no way guarantee that any code developed will be secure.

Considering it is debatable how much effort is going into securing kernel code by third parties, there is certainly plenty of potential for further research and tool development in the area of kernel security, both from an attack and defense perspective.

# 7 – References


1. amd.com, *Revision Guide for AMD Athlon™ 64 and AMD Opteron™ Processors*, http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/25759.pdf

2. bochs.sourceforge.net, *Bochs Documentation*, http://bochs.sourceforge.net/doc/docbook/

3. Collake, J. (2001), *NtDispatchPoints*

4. Heasman, J. (2006), *Implementing and Detecting a PCI Rootkit*, http://www.ngssoftware.com/research/papers/Implementing_And_Detecting_A_PCI_Rootkit.pdf

5. Butler, J., Hoglund, G. (2005), *Rootkits: Subverting the Windows Kernel*, Addison-Wesley

6. info-pull.com, *Microsoft Windows kernel GDI local privilege escalation*, http://projects.info-pull.com/mokb/MOKB-06-11-2006.html

7. intel.com*, Intel® 64 and IA-32 Architectures Software Developer's Manuals*, http://www.intel.com/products/processor/manuals/index.htm

8. intel.com, *Intel® Core™2 Extreme Quad-Core Processor QX6000 Sequence and Intel® Core™2 Quad Processor Q6000 Sequence Specification Update*, http://www.intel.com/design/processor/specupdt/315593.htm

9. Kumar, N., Kumar, V. (2007), *Vbootkit: Compromising Windows Vista Security*, http://www.rootkit.com/newsread.php?newsid=671

10. linux-ntfs.org, *NTFS Documentation*, http://prdownloads.sourceforge.net/linux-ntfs/ntfsdoc-0.5.zip

11. microsoft.com, *About the Windows Driver Kit (WDK)*, http://www.microsoft.com/whdc/devtools/WDK/AboutWDK.mspx

12. microsoft.com, *Debugging Tools for Windows - Overview*, http://www.microsoft.com/whdc/devtools/debugging/default.mspx

13. microsoft.com, Kernel-*Mode Code Signing Policy (Windows Vista)*, http://msdn2.microsoft.com/En-US/library/aa906239.aspx

14. microsoft.com, *Microsoft Research Singularity Project*, http://research.microsoft.com/os/singularity/

15. microsoft.com, PREFast *step by step*, http://www.microsoft.com/whdc/DevTools/tools/PREfast_steps.mspx

16. microsoft.com, *Static Driver Verifier: Finding Bugs in Device Drivers at Compile-Time provided with the WDK*, www.microsoft.com/whdc/devtools/tools/sdv.mspx

17. minix3.org, *MINIX 3 - Documentation*, http://www.minix3.org/doc/

18. Permeh, R., Soeder, D., *eEye BootRoot*, http://www.eeye.com/html/resources/downloads/other/index.html

19. Probert, D. B., *Windows Kernel Internals*, http://www.i.u-tokyo.ac.jp/edu/training/ss/lecture/

20. resplendence.com, *RootKit Hook Analyzer*, http://www.resplendence.com/hookanalyzer

21. Russinovich, M. (2006), *WinObj*,
http://www.microsoft.com/technet/sysinternals/SystemInformation/WinObj.mspx

22. Russinovich, M., Solomon, D. (2005), *Windows Internals*, Redmond: Microsoft

23. skape, Skywing (2006), *Bypassing PatchGuard on Windows x64*,
http://www.uninformed.org/?v=3&a=3&t=sumry

24. Skywing (2006), Anti-*Virus Software Gone Wrong*,
http://www.uninformed.org/?v=4&a=4&t=sumry

25. Sotirov, A. (2007), *Heap Feng Shui in JavaScript*,
http://www.determina.com/security.research/presentations/bh-eu07/bh-eu07-sotirov-paper.html

26. symbian.com, *System documentation*,
http://developer.symbian.com/main/oslibrary/osdocs/index.jsp

27. unix.org, *The Single UNIX Specification*, Version 3, http://www.unix.org/version3/

28. Vieler, R. (2007), *Professional Rootkits*, Wiley

# A – Windows NT kernel architecture

This Appendix will provide an overview of the relevant areas of the NT kernel architecture from the point of view of an attacker as well as highlighting any protection systems that may provide barriers.

## A.1 – Terminology

For clarity, the meaning of several terms used will be outlined. In an effort to abstract away from specific processor architectures, terms such as 'ring 0' will not be used; having said that, the specifics of IA64 will not be dealt with. The meaning of standard terms such as 'virtual address' or 'page' is assumed to be known and understood by the reader.

- User mode: the normal execution mode implemented in processor hardware, whereby certain functionality and resources are restricted.
- Supervisor mode: a privileged execution mode implemented in processor hardware where additional functionality is available for hardware management.
- Kernel mode: in Windows kernel development documentation, this is used to refer to supervisor mode. In this paper, the terms 'supervisor mode' and 'kernel mode' are used interchangeably.
- NT kernel: this is a vague term used cover the module providing the core supervisor routines (ntoskrnl.exe, ntkrnlmp.exe, and so forth), the module providing the Hardware Abstraction Layer (typically hal.dll), and various kernel mode support libraries and drivers. This is used in an attempt to distinguish the operating system supervisor mode kernel code from the user mode kernel code, as well as other supervisor code.

## A.2 – Hardware based protection

Hardware based protection is architecture dependent, though there are several commonly used strategies, most of which are employed on most Windows operating systems.

The distinction between user and supervisor mode allows for protected and privileged instructions – instructions that can only be executed in supervisor mode, or instructions that may be executed in user mode depending upon the processor configuration (for example, the `in` and `out` instructions on x86 processors) [7]. Beyond that, pages can be marked as supervisor only, in which case they cannot be accessed by code running in user mode.

Memory is also typically protected with standard read/write settings, and some processor architectures may also support memory being marked as non-executable; and memory may have segment based protection where segments describe chunks of memory, and the privilege level needed to access them.

Modern versions of Windows NT support and use all these protection mechanisms, aside from memory segmentation. NT uses a flat memory model for x86, whereby all segments start at 0 and have a 4GB limit. Memory segmentation has largely been dropped for x64, with the `fs` register being retained and used to point to the Thread Information Block.

With this in mind, there needs to be one or more available methods for a thread of execution to transition between privilege levels. Descriptor tables can be, and are, used to define boundaries between different privilege levels – for example entries defining call and interrupt gates allow the privilege level required and the privilege level granted for interrupts and far calls or jumps to be specified, as well as the entry point for specific interrupts. More recent x86 processors introduced the `sysenter` and `sysexit` instructions which are programmable through Model Specific

Registers (accessible only from supervisor mode) that will quickly transition a thread from user mode to a specific location in supervisor mode and back.

## A.3 – Operating system memory layout and management

The NT kernel resides in the system address space of the operating system, and as such is mapped into all user mode processes. The session space of the kernel contains session specific kernel code and data mapped in per session (of relevance in particular for display drivers). Typically for 32-bit versions of NT, the split between user and supervisor memory is made at 0x80000000, though this can be configured to be otherwise where an application would gain from additional virtual address space. On 64-bit versions, the split is made at 0x80000000000. Pages above this boundary have the supervisor bit set so that they are inaccessible from user mode.

As this page based protection is only relevant for attempted user mode access to supervisor memory, the supervisor must also initialize internal variables that will allow it to distinguish between data in the user mode address space as opposed to the supervisor address space. Under 32-bit NT, the following 5 variables (the three MmXxx variables being exported from the kernel) are used to delimit the UM address space, provided with their typical initialization values from XP SP2:

| Symbol | Initialized value |
|---|---|
| *MmHighestUserAddress* | 0x7FFEFFFF |
| *MmUserProbeAddress* | 0x7FFF0000 |
| *MmSystemRangeStart* | 0x80000000 |
| *MiHighestUserPte* | 0xC01FFFBC |
| *MiHighestUserPde* | 0xC03007FC |

**Table A.1** – some of the variables used to define memory usage for x86 NT

Under 64-bit NT, these variables have the following values, though MmSystemRangeStart is no longer exported:

| Symbol | Initialized value |
|---|---|
| *MmHighestUserAddress* | 0x7FFFFFEFFFF |
| *MmUserProbeAddress* | 0x7FFFFF0000 |
| *MmSystemRangeStart* | 0xFFFF080000000000 |

**Table A.2** – some of the variables used to define memory usage for x64 NT

The page table data structure limit variables are necessarily expanded for 64-bit processors to allow for the additional translation layer for addresses 64-bits wide, and are also configured dynamically under Windows Vista.

If it is not at first obvious why these boundaries must be defined in this way, then it is fairly simple: the services provided by the NT kernel can operate on data provided from user mode. It may be necessary for supervisor code to copy this data into memory allocated within the supervisor, often due to the NT kernel being designed as an asynchronous architecture, thus allowing kernel services to run in an arbitrary process context, in which case pointers into user mode memory may not be trusted.

Communication with the NT kernel allows three different ways to access user mode memory: a Memory Descriptor List, which is a partially opaque structure used to describe chunks of memory that can be used with the Memory Manager in the NT kernel to lock pages from user mode into memory so that they will be available to the driver; via a system buffer, where the IO Manager allocates supervisor memory and copies the data to and from user mode; and finally via direct access to the user mode memory.

Both the second and third methods raise interesting issues: in the latter case, trusting pointers, structures and data directly from user mode; and in the former case – if the kernel has to copy data to and from user mode, then it must validate the buffers any user mode code provides for input and output. Typically this involves testing whether the memory is readable and/or writable, verifying the buffers are reasonably sized, and checking that they reside fully within the user mode address space by comparison with *MmUserProbeAddress*. This functionality is also provided to kernel modules via the *ProbeForRead* and *ProbeForWrite* NT kernel exports. The following code outlines how the I/O manager validates data from user mode, which is only executed if the function is not called directly from supervisor mode:

```
IopXxxControlFile(
    HANDLE FileHandle,
    HANDLE Event,
    PIO_APC_ROUTINE ApcRoutine,
    PVOID ApcContext,
    PIO_STATUS_BLOCK IoStatusBlock,
    ULONG IoControlCode,
    PVOID InputBuffer,
    ULONG InputBufferLength,
    PVOID OutputBuffer,
    ULONG OutputBufferLength,
    BOOLEAN DeviceIO
    )
{

    try
    {
        if ((IoControlCode & 3) == METHOD_BUFFERED)
        {
            if (OutputBuffer != NULL)
                ProbeForWrite(OutputBuffer, OutputBufferLength, 1);
            else
                OutputBufferLength = 0;
        }

        if (IoControlCode & 3 != METHOD_NEITHER)
        {
             if (InputBuffer != NULL)
                  ProbeForRead(InputBuffer, InputBufferLength, 1);
             else
                  InputBufferLength = 0;
        }
    }
    except(EXCEPTION_EXECUTE_HANDLER)
    {
          //handle the read or write exception
    }

    //dispatch the request appropriately
}
```

Thus, in principle, the user and supervisor address spaces are segregated. Further details of NT kernel memory management can be found in Windows Internals [22].

## A.4 – Public kernel interfaces

As the kernel is designed to provide resource and operating system management functionality to user mode code, it must have defined interfaces to allow communication between user and supervisor code.

### A.4.1 – System calls

The most well known and well investigated of these is the system call interface, whereby NT kernel functionality is exposed to user mode via particular instructions; for x86 NT, either `int 2e` handled by *KiSystemService* in the kernel, or `sysenter` handled by *KiFastCallEntry*, are used. Both of these functions provide initial validation, where necessary, and copy the arguments from the user mode to the kernel mode stack ready to be processed by the given function from either the specified function in the *KiServiceTable* table of pointers, or in the *W32pServiceTable* table from the supervisor portion of the GDI subsystem.

Neither of these instructions is intended for direct use by third party user mode code, and instead this functionality is exposed via symbols in the user mode DLLs ntdll.dll and gdi32.dll. In recent builds based of the NT architecture for x86, such as XP SP2 or Vista, any system calls go via stub code in the *SharedUserData* area of memory mapped into all processes. The appropriate instruction to transition to the kernel is in this stub depending upon whether the processor supports `sysenter` or not.

### A.4.2 – Device drivers

Device drivers are allowed to provide interfaces exposed to user mode to allow user mode code to interact by proxy with supervisor code, or hardware managed by supervisor code. When a driver is loaded and initialized, the entry point (typically some version of DriverEntry) is called and allows the driver to allocate and initialize resources and device specific data, register a callback for when devices are added for it to manage, and provide functions to the IO manager that will be called when the driver is requested to process some request. The kernel uses objects to describe kernel entities such as ports, devices, drivers, events, and so forth; function pointers, where needed, are stored within these objects.

Internally, the IO manager uses data structures called IO Request Packets (IRPs) to pass requests to a driver. The particular fields of interest are as follows:

- *SystemBuffer* – part of the AssociatedIrp union, used to point to the system buffer for buffered I/O
- *IoStatus* – used to report the status of the request after processing
- *RequestorMode* – indicates whether the IRP processing was requested from user or kernel mode
- *UserBuffer* – pointer to the user supplied buffer, if present
- *CurrentStackLocation* – part of the IRP Tail, and contains IRP specific information such as the major and minor IRP codes, as well as the IoControlCode if applicable

The IRP dispatching routines are stored within the *DriverObject* during driver initialization. IRPs of interest to those wishing to break into the kernel are detailed in the following table, along with the user mode APIs that are will cause their generation:

| IRP Major code | Win32 API | Native API |
|---|---|---|
| IRP IRP_MJ_CREATE | CreateFile | NtCreateFile |
| IRP_MJ_CLOSE | CloseHandle | NtClose |
| IRP_MJ_READ | ReadFile | NtReadFile |
| IRP_MJ_WRITE | WriteFile | NtWriteFile |
| IRP_MJ_DEVICE_CONTROL | DeviceIoControl | NtDeviceIoControlFile |

**Table A.3** – IRPs and their related user mode functions

When the driver is initialized, it can create a named device using *IoCreateDevice*, and then if need be create a symbolic link to this device for legacy DOS device support using *IoCreateSymbolicLink*. These can then be opened and operated upon as if they were files. *DeviceIoControl* is used for more sophisticated interaction with the device.

It is also worth paying extra attention to 64-bit drivers as the 64-bit NT kernel does not support 32-bit drivers, but the OS does support 32-but applications in a compatibility mode. Thus, any drivers must be able to distinguish between 32-bit and 64-bit processes interacting with it, by calling *IoIs32-bitProcess*, and transform the input from 32-bit to 64-bit before processing. This provides an additional area where bugs could be introduced into the code.

### A.4.3 – Display and display miniport drivers

The display driver architecture is somewhat different from that of standard device drivers. Communication from user mode code is directed toward a display driver that exists in the kernel's session space, and not system space. The driver therefore only links to other modules loaded in session space, typically just the exports from Win32k, though other modules can be and often are loaded for DirectX or OpenGL support. Requests are sent to the driver by calling the user mode APIs *ExtEscape* and *DrawEscape*.

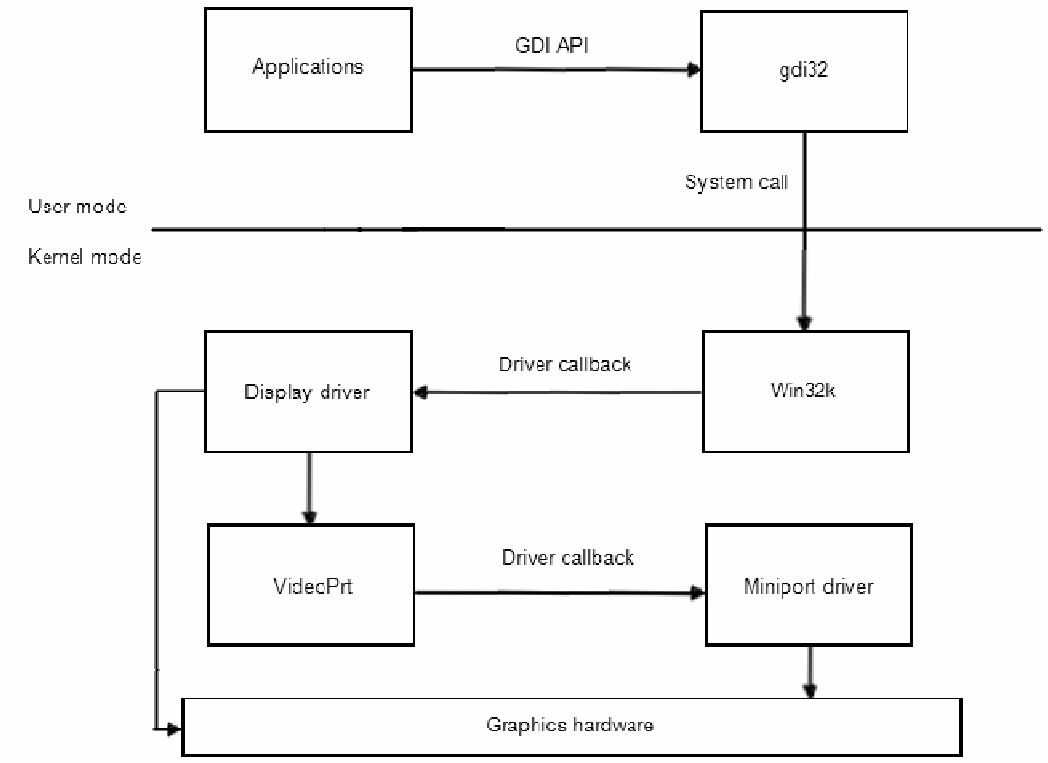


**Figure A.1 –** A schematic view of the GDI architecture

However, display drivers do not register IRP handling functions as for standard drivers; in fact their initialization is entirely different.

```
BOOL
  DrvEnableDriver(
    IN ULONG  iEngineVersion,
    IN ULONG  cj,
    OUT DRVENABLEDATA  *pded
    );
```

*iEngineVersion* and *cj* are used by the driver to identify and deal the version of the Graphics Device Interface (GDI) running on the system. `pded` points to a *DRVENABLEDATA* structure that is used to register function callbacks with GDI. Functions are indicated by index, and the functions exposed to user mode access are *DrvEscape* handling *ExtEscape* with an index of 24 and *DrvDrawEscape* handling *DrawEscape* with an index of 25.

```
int
  ExtEscape(
    HDC hdc,
    int nEscape,
    int cbInput,
    LPCSTR lpszInData,
    int cbOutput,
    LPSTR lpszOutData
  );
```

Of interest is the fact that data is dispatched differently to the display driver than in the case of standard drivers – if the input buffer is larger than 32 bytes in size then Win32k either allocates memory and copies the data across, or locks it in memory as PAGE_READONLY by calling *MmSecureVirtualMemory*; correspondingly, if the output buffer is larger than 32 bytes then it too is relocated to allocated pool memory. If either buffer is smaller than 32 bytes, then a buffer is allocated on the supervisor stack. This raises the possibility of a return pointer overwrite on the supervisor stack if there is poor input validation in the display driver.

Some requests may be handled by Win32k itself, the rest being passed on to the appropriate display driver. The display driver can then either handle the request itself, or forward it onto the miniport driver for the particular display device by calling *EngDeviceIoControl* (presumably having verified parameters where needed). The miniport driver does not reside in session space, but in system space like most other drivers, and the entry point has a similar prototype. However, it handles Video Request Packets which are to some extent a stripped down version of IRPs.

```
typedef struct _VIDEO_REQUEST_PACKET {
  ULONG  IoControlCode;
  PSTATUS_BLOCK  StatusBlock;
  PVOID  InputBuffer;
  ULONG  InputBufferLength;
  PVOID  OutputBuffer;
  ULONG  OutputBufferLength;
} VIDEO_REQUEST_PACKET, *PVIDEO_REQUEST_PACKET;
```

The miniport driver does not register VRP handling as standard drivers do by modifying their *DriverObject*, but instead by calling *VideoPortInitialize* exported by videoprt.sys. This registers the miniport driver with the framework provided mainly for handling hardware, initialization, and power events. The interface specified must also include *HwVidStartIO* which is the function responsible for processing any VRPs received. The miniport can then process these packets as appropriate, and then communicate with the underlying hardware if need be.

**A.4.4 – Shared memory sections**

Shared memory sections, created and mapped with *NtCreateSection*, *NtOpenSection* and *NtMapViewOfSection* are designed to allow (named) areas of memory to be shared between different contexts, and can be used to store shared information between processes or between user and kernel mode, or with some signaling mechanism (such as using an event and a semaphore) can be used to transfer data between processes or between user mode and kernel mode.

If a handle to an unnamed section is present in a process's handle table, then it is possible to brute-force the handle in an attempt to open the section by attempting *NtMapViewOfSection* on all possible handles.

**A.4.5 – Ports**

The filtering framework provided in the NT kernel contains APIs to create and manage ports for communication between a filter driver and a user mode application. *FltCreateCommunicationPort* is used to create a port from kernel mode and specify the callback function to process requests from a user mode application. *FltSendMessage* is then used to send data back to the user mode application. User mode applications can use *FilterConnectCommunicationPort*, *FilterSendMessage*, *FilterGetMessage*, and *FilterReplyMessage* to interact with the port.

```
NTSTATUS
  FltCreateCommunicationPort(
    IN PFLT_FILTER  Filter,
    OUT PFLT_PORT  *ServerPort,
    IN POBJECT_ATTRIBUTES  ObjectAttributes,
    IN PVOID  ServerPortCookie OPTIONAL,
    IN PFLT_CONNECT_NOTIFY  ConnectNotifyCallback,
    IN PFLT_DISCONNECT_NOTIFY  DisconnectNotifyCallback,
    IN PFLT_MESSAGE_NOTIFY  MessageNotifyCallback,
    IN LONG  MaxConnections
    );
```

An object to represent the port is created by the Object Manager in the kernel, and added to the object directory. Internally, *NtDeviceIoControlFile* is used by the user mode filter library (fltlib.dll) to perform requests with the library waiting for the filter driver to respond.

**A.4.6 – Windows Driver Framework**

The Windows Driver Framework was introduced to simplify driver development, particularly things like power management. This framework is essentially a provided by two kernel modules: a loader that is directly linked to by a driver using the framework, and the framework provider. As these may be version specific, drivers written using the WDF must be created with a coinstaller that installs the required WDF components.

Drivers written using WDF have their DriverEntry wrapped in WDF specific code that is responsible for binding to the correct version of the WDF library, and then calling the actual DriverEntry. This is achieved by calling *WdfVersionBind* in wdfldr.sys. This allows different version of the WDF to be present on the same system. The *WdfVersionBind* prototype is provided below:

```
NTSTATUS
  WdfVersionBind(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath,
    IN PWDF_VERSION_INFO  WdfVersion,
    IN P WdfDriverGlobals
    );
```

Unbinding on driver unloading is also handled internally by the framework. *WdfDriverGlobals* is used as a hidden parameter to all WDF functions (namely, a driver developer does not have to specify it, it is automatically added during compilation) and allows the WDF library to determine which particular driver it is dealing with; internally it is a pointer to an object that is inserted into a linked list that contains internal support routines and a pointer to the original *DriverObject*. The *WDF_VERSION_INFO* structure is outlined below, and contains a pointer to a table that will be filled out by the WDF loader with the requisite functions from the requisite WDF library.

```
typedef struct _WDF_VERSION_INFO {
  ULONG  Size
  PWSTR  LibraryName
  ULONG  WdfMajorVersion
  ULONG  WdfMinorVersion
  ULONG  WdfBuildNumber
  ULONG  NumWdfFunctions
  PVOID  WdfFunctions
} WDF_VERSION_INFO, *PWDF_VERSION_INFO;
```

The *WdfFunctions* pointer is to a table (of at least size *NumWdfFunctions*) that is filled out by the specified WDF library with all the WDF routines. These are the typical WDF routines that are then called by the code written by the driver author.

Beyond this architectural change to the original WDM, Microsoft has greatly simplified the model used to write drivers. The IRP handling functions are filled out by the framework library itself, and dealt with as appropriate. So, instead of modifying the *DriverObject* directly as previously, the driver registers unload and device addition functions by calling *WdfDriverCreate*:

```
NTSTATUS
  WdfDriverCreate(
    IN PDRIVER_OBJECT  DriverObject,
    IN PUNICODE_STRING  RegistryPath,
    IN OPTIONAL PWDF_OBJECT_ATTRIBUTES  DriverAttributes,
    IN PWDF_DRIVER_CONFIG  DriverConfig,
    OUT OPTIONAL WDFDRIVER*  Driver
    );
```

Here, *DriverConfig* contains the pointers to the requisite functions:

```
typedef struct _WDF_DRIVER_CONFIG {
  ULONG  Size;
  PFN_WDF_DRIVER_DEVICE_ADD  DriverDeviceAdd;
  PFN_WDF_DRIVER_UNLOAD  DriverUnload;
  ULONG  DriverInitFlags;
  ULONG  DriverPoolTag;
} WDF_DRIVER_CONFIG, *PWDF_DRIVER_CONFIG;
```

Create and close request callbacks can be created by calling *WdfDeviceInitSetFileObjectConfig*. Other I/O operations are managed via I/O queues for specific devices. These are created by calling *WdfIoQueueCreate*:

```
NTSTATUS
  WdfIoQueueCreate(
   IN WDFDEVICE  Device,
   IN PWDF_IO_QUEUE_CONFIG  Config,
   IN OPTIONAL PWDF_OBJECT_ATTRIBUTES  QueueAttributes,
   OUT WDFQUEUE*  Queue
   );
```

In this case, the optional *WDF_IO_QUEUE_CONFIG* structure will contain the interesting function pointers:

```
typedef struct _WDF_IO_QUEUE_CONFIG {
  ULONG  Size;
  WDF_IO_QUEUE_DISPATCH_TYPE  DispatchType;
  WDF_TRI_STATE  PowerManaged;
  BOOLEAN  AllowZeroLengthRequests;
```

```
  BOOLEAN  DefaultQueue;
  PFN_WDF_IO_QUEUE_IO_START  IoDefault;
  PFN_WDF_IO_QUEUE_IO_READ  IoRead;
  PFN_WDF_IO_QUEUE_IO_WRITE  IoWrite;
  PFN_WDF_IO_QUEUE_IO_DEVICE_CONTROL  EvtIoDeviceControl;
  PFN_WDF_IO_QUEUE_IO_INTERNAL_DEVICE_CONTROL IoInternalDeviceControl;
  PFN_WDF_IO_QUEUE_IO_STOP  IoStop;
  PFN_WDF_IO_QUEUE_IO_RESUME  IoResume;
  PFN_WDF_IO_QUEUE_IO_CANCELED_ON_QUEUE  IoCanceledOnQueue;
} WDF_IO_QUEUE_CONFIG, *PWDF_IO_QUEUE_CONFIG;
```

If implemented, the *EvtIoDeviceControl* routine will process *DeviceIoControl* calls from user mode code. Individual I/O queues can be configured with *WdfDeviceConfigureRequestDispatching* to allow processing of different requests; this method can be called multiple times for the same queue to configure it to receive one or more of the following request types, corresponding to the functions provided in the *WDF_IO_QUEUE_CONFIG* structure:

*WdfRequestTypeCreate*
*WdfRequestTypeRead*
*WdfRequestTypeWrite*
*WdfRequestTypeDeviceControl*
*WdfRequestTypeDeviceControlInternal*

So, though the WDF provides a simpler model for development, the layers added do make analyzing WDF drivers slightly more time consuming, though it would not be particularly difficult to create tools to aid in the automated analysis of such drivers.

**A.4.7 – Kernel mode callbacks**
Recent iterations of the NT architecture, particularly from XP onwards have introduced specific callbacks (though a general callback interface is available via *ExCreateCallback* and *ExRegisterCallback*). Process and executable image related callbacks can be registered through the Process manager (*PsSetCreateProcessNotifyRoutine, PsSetCreateThreadNotifyRoutine, and PsSetLoadImageNotifyRoutine*), registry related callbacks can be registered through the configuration manager (*CmRegisterCallback and CmRegisterCallbackEx*), and file system filtering through the runtime file system support (*FsRtlRegisterFileSystemFilterCallbacks*).

All of these callbacks provide an opportunity for an unprivileged user to provide arbitrary data to some supervisor code (within reason).

# B – CDFS driver disassembly

The following example code is taken from the CDFS driver provided in the current release of the WDK (version 6000), thus allowing the reader to correlate the disassembly with the related source code should they so desire. The first section is from the DriverEntry routine:

```
; int __stdcall DriverEntry(PDRIVER_OBJECT DriverObject, PUNICODE_STRING RegistryPath)

DeviceName= LSA_UNICODE_STRING ptr -0Ch
DeviceObject= dword ptr -4
DriverObject= dword ptr  8

        mov     edi, edi
        push    ebp
        mov     ebp, esp
        sub     esp, 0Ch
        push    esi
        push    offset Cdfs                                     ; SourceString
        lea     eax, [ebp+DeviceName]
        push    eax                                             ; DestinationString
        call    ds:__imp__RtlInitUnicodeString@8                ; RtlInitUnicodeString
        mov     esi, [ebp+DriverObject]
        lea     eax, [ebp+DeviceObject]
        push    eax                                             ; DeviceObject
        push    0                                               ; Exclusive
        push    0                                               ; DeviceCharacteristics
        push    3                                               ; DeviceType
        lea     eax, [ebp+DeviceName]
        push    eax                                             ; DeviceName
        push    0                                               ; DeviceExtensionSize
        push    esi                                             ; DriverObject
        call    ds:__imp__IoCreateDevice@28                     ; IoCreateDevice
        test    eax, eax
        jl      loc_1DFD1                                       ; Continue if we haven't failed
        mov     eax, offset CdFsdDispatch                       ; to create the associated
        push    edi                                             ; device object.
        push    [ebp+DeviceObject]
        mov     dword ptr [esi+34h], offset CdUnload
        mov     [esi+0A4h], eax                                 ; Fill out the supported
        mov     [esi+_DRIVER_OBJECT.IrpCleanupDispatch], eax    ; functions in the driver's
        mov     [esi+_DRIVER_OBJECT.IrpLockControlDispatch], eax ; dispatch table.
        mov     [esi+_DRIVER_OBJECT.IrpDeviceControlDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpFileSystemControlDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpDirectoryControlDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpQueryVolumeInformationDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpSetInformationDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpQueryInformationDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpReadDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpCloseDispatch], eax
        mov     [esi+_DRIVER_OBJECT.IrpCreateDispatch], eax
        mov     dword ptr [esi+78h], offset CdShutdown
        mov     dword ptr [esi+28h], offset CdFastIoDispatch
        call    ds:__imp__IoRegisterShutdownNotification@4      ;IoRegisterShutdownNotification
        push    [ebp+DeviceObject]                              ; DeviceObject
        mov     edi, eax
        test    edi, edi
        jge     short loc_1DF99
        call    ds:__imp__IoDeleteDevice@4                      ; IoDeleteDevice
        mov     eax, edi
        jmp     short loc_1DFD0
; ---------------------------------------------------------------------------
loc_1DF99:
        push    esi
        call    CdInitializeGlobalData                          ; Check the global data was
        mov     esi, eax                                        ; initialized ok, otherwise
        test    esi, esi                                        ; delete the device and return
        jge     short loc_1DFB2                                 ; the error code.
```

With the correct structures available, it is a very quick and easy task to spot any potential ways to access this code from user mode: it registers several dispatch routines, including one to handle *DeviceIoControl* requests, it registers as a file system driver, and the function *CdInitializeGlobalData(DriverObject)* fills out the fast I/O dispatch routines.

Dispatch routines receive a pointer to the associated *DEVICE_OBJECT* and a pointer to the IRP that requires processing. For the CDFS example, all supported IRP functions are handled by a generic dispatch handler, as can be seen from the disassembly above. Internally to this generic dispatch function, *CdCreateIrpContext* is called passing in the IRP as a parameter. Memory is allocated and useful information extracted from the IRP:

```
CdCreateIrpContext proc near

 IRP            = dword ptr  8
 bSynchronous   = byte ptr  0Ch

         mov     edi, edi
         push    ebp
         mov     ebp, esp
         mov     eax, [ebp+IRP]                          ; Get the DriverObject
         push    esi
         push    edi
         mov     edi, [eax+60h]                          ; Get the CurrentStackLocation
         mov     eax, [edi+14h]                          ; from the IRP, and then
         cmp     eax, DeviceObject                       ; the associated DeviceObject
         jnz     short loc_1C531
         cmp     dword ptr [edi+18h], 0                  ; If they're not the same,
         jz      short loc_1C531                         ; check there's no FileObject
         mov     al, [edi]
         test    al, al
         jz      short loc_1C531                         ; If there is, check it's an
         cmp     al, 12h                                 ; IRP for IRP_MJ_CLEANUP,
         jz      short loc_1C531                         ; IRP_MJ_CREATE, IRP_MJ_CLOSE
         cmp     al, 2
         jz      short loc_1C531
         push    0C0000010h                             ; Status
         call    ds:__imp__ExRaiseStatus@4              ; ExRaiseStatus
 loc_1C531:
```

Some initial preprocessing is done to validate the request. Of particular interest is the *IO_STACK_LOCATION* structure retrieved from `IRP.Tail.CurrentStackLocation` – this contains some IRP specific request information, such as the IRP major and minor codes, though the contents are specific to the different *IRP_MJ_X* requests. For example, the CurrentStackLocation for *IRP_MJ_DEVICE_CONTROL* is defined as follows:

```
typedef struct _IO_STACK_LOCATION {
  UCHAR  MajorFunction;
  UCHAR  MinorFunction;
  UCHAR  Flags;
  UCHAR  Control;

  struct  {
    ULONG  OutputBufferLength;
    ULONG  POINTER_ALIGNMENT  InputBufferLength;
    ULONG  POINTER_ALIGNMENT  IoControlCode;
    PVOID  Type3InputBuffer;
  } DeviceIoControl;

  PDEVICE_OBJECT  DeviceObject;
  PFILE_OBJECT  FileObject;
  PIO_COMPLETION_ROUTINE  CompletionRoutine;
  PVOID  Context;
} IO_STACK_LOCATION, *PIO_STACK_LOCATION;
```

```
loc_1C57D:
        push    'cidC'                                  ; Tag
        push    38h                                     ; NumberOfBytes
        push    10h                                     ; PoolType
        call    ds:__imp__ExAllocatePoolWithTag@12      ; ExAllocatePoolWithTag
        mov     esi, eax
 loc_1C58E:
        push    38h                                     ; size_t
        push    0                                       ; int
        push    esi                                     ; void *
        call    _memset                                 ; Zero the allocated memory
        mov     eax, [ebp+IRP]
        mov     word ptr [esi], 308h                    ; Initialize the context
        mov     word ptr [esi+2], 38h                   ; structure
        mov     [esi+4], eax                            ; Store the IRP pointer
        mov     eax, [edi+18h]
        add     esp, 0Ch
        test    eax, eax                                ; If there is an associated
        jz      short loc_1C5B9                         ; FileObject, then store its
        mov     eax, [eax+4]                            ; DeviceObject
        mov     [esi+14h], eax
 loc_1C5B9:
        mov     eax, [edi+14h]
        cmp     eax, DeviceObject
        jz      short loc_1C5CC
        add     eax, 0D0h
        mov     [esi+8], eax
 loc_1C5CC:
        cmp     [ebp+bSynchronous], 0
        mov     al, [edi]                               ; Get the IRP major code
        mov     [esi+20h], al
        mov     al, [edi+1]                             ; And the minor code
        mov     [esi+21h], al
        jz      short loc_1C5E3
        or      dword ptr [esi+10h], 4                  ; Set whether it's synchronous
        jmp     short loc_1C5E7                         ; or not
 ; ---------------------------------------------------------------------------
 loc_1C5E3:
        or      dword ptr [esi+10h], 8
 loc_1C5E7:
        pop     edi
        mov     eax, esi
        pop     esi
        pop     ebp
        retn    8
 CdCreateIrpContext endp
```

Of note is that the memory allocation above is not checked as having succeeded before the memory is zeroed. This could lead to a bugcheck in severely low-memory situations, where a 56 byte allocation may fail. Having processed the IRP in this manner, the major code stored in the context is then used in a switch statement to determine which internal function should process the IRP; all IRP processing functions take the IRP and the constructed context as parameters, thus allowing the I/O operations to be asynchronous. In this case, of interest is the function to handle *IRP_MJ_DEVICE_CONTROL*:

```
CdCommonDevControl proc near

 var_4   = dword ptr -4
 Context = dword ptr  8
 IRP     = dword ptr  0Ch

        mov     edi, edi
        push    ebp
        mov     ebp, esp
        push    ecx
        push    ebx
        mov     ebx, [ebp+IRP]
        push    esi
        mov     esi, [ebx+60h]
        push    edi
        mov     edi, [ebp+Context]
        lea     eax, [ebp+var_4]
        push    eax
        lea     eax, [ebp+IRP]
        push    eax
        push    dword ptr [esi+18h]                     ; Get and decode the FileObject
        push    edi
```

```
        call    CdDecodeFileObject
        cmp     eax, 2
        jz      short loc_15745
        mov     esi, 0C000000Dh                         ; Check it's a valid request
loc_15739:
        push    esi
        push    ebx
        push    edi
        call    CdCompleteRequest                       ; Complete if invalid
        mov     eax, esi
        jmp     short loc_15799
; ---------------------------------------------------------------------------
loc_15745:                                              ; Get the IoControlCode from
        mov     eax, [esi+0Ch]                          ; IRP.Tail.CurrentStackLocation
        cmp     eax, 24000h                             ; and check if it is 0x24000
        jnz     short loc_157A0
        mov     eax, [ebp+IRP]
        push    dword ptr [eax+40h]
        push    edi
        call    CdVerifyVcb                             ; Verify the Volume Control
loc_1575B:                                              ; and proceed with the request
        mov     edi, [ebx+60h]
        push    9
        sub     edi, 24h
        pop     ecx
        rep movsd
        mov     eax, [ebx+60h]
        mov     esi, [ebp+Context]
        sub     eax, 24h
        mov     dword ptr [eax+1Ch], offset CdDevCtrlCompletionRoutine
        xor     edi, edi
        mov     [eax+20h], edi
        mov     byte ptr [eax+3], 0E0h
        mov     eax, [esi+8]
        mov     ecx, [eax+8]
        mov     edx, ebx
        call    ds:__imp_@IofCallDriver@8               ; Forward the request down the
        push    edi                                     ; device stack
        push    edi
        push    esi
        mov     ebx, eax
        call    CdCompleteRequest
        mov     eax, ebx
loc_15799:
        pop     edi
        pop     esi
        pop     ebx
        leave
        retn    8
; ---------------------------------------------------------------------------
loc_157A0:
        cmp     eax, 20040h                             ; Check if the IoControlCode is
        jnz     short loc_1575B                         ; 0x20040, and if it isn't then
        mov     eax, [ebp+IRP]                          ; forward the request
        push    dword ptr [eax+40h]
        push    edi
        call    CdVerifyVcb                             ; Verify and proceed with the
        push    4                                       ; request
        pop     eax
        cmp     [esi+4], eax
        jnb     short loc_157C5
        mov     esi, 0C0000023h
        jmp     loc_15739
; ---------------------------------------------------------------------------
loc_157C5:
        mov     ecx, [ebp+IRP]                          ; For this IOCTL, the request
        mov     ecx, [ecx+40h]                          ; is not forwarded
        mov     ecx, [ecx+13Ch]
        mov     edx, [ebx+0Ch]
        push    0
        push    ebx
        mov     [edx], ecx
        push    edi
        mov     [ebx+1Ch], eax
        call    CdCompleteRequest
        xor     eax, eax
        jmp     short loc_15799
CdCommonDevControl endp
```

As can be seen from the above disassembly, the CDFS driver only supports two IOCTLs itself, and processing of those IOCTLs is not complicated. However, the principle is the same for drivers irrespective of their complexity: if IRPs are handled in a generic dispatch routine then most likely a switch statement will be used to call the appropriate dispatch handler. Useful parameters are retrieved from the IRPs `CurrentStackLocation` structure. Within the *IRP_MJ_DEVICE_CONTROL* function, a switch statement can be used to process the `IoControlCode` and hand off the IRP to the relevant control function, provided there are enough IOCTLs supported.

# C – Real world examples

The Uninformed assessment of Kaspersky Internet Security Suite 5 and McAfee Internet Security Suite 2006 [24] provides an excellent example of assessing vulnerabilities in the kernel components of two anti-virus products, and the 'Microsoft Windows kernel GDI local privilege escalation' [6] found during the 'Month of Kernel Bugs' is also an interesting case in point. This section will provide several other real world examples of either architectural flaws or coding bugs, along with a quick overview of how they could be exploited; the three final examples have been modified to protect the indigent.

## C.1 – The NT kernel compression library

The runtime library provided by the NT operating system contains support for compression in both user mode and kernel mode. The functions exported from the NT kernel are as follows, with an example function prototype:

*RtlCompressBuffer*
*RtlCompressChunks*
*RtlDecompressBuffer*
*RtlDescribeChunk*
*RtlDecompressChunks*
*RtlDecompressFragment*
*RtlGetCompressionWorkSpaceSize*
*RtlReserveChunk*

```
NTSTATUS
RtlCompressBuffer (
    IN USHORT   CompressionFormatAndEngine,
    IN PUCHAR   UncompressedBuffer,
    IN ULONG    UncompressedBufferSize,
    OUT PUCHAR  CompressedBuffer,
    IN ULONG    CompressedBufferSize,
    IN ULONG    UncompressedChunkSize,
    OUT PULONG  FinalCompressedSize,
    IN PVOID    WorkSpace
);
```

For all these functions, the parameter of interest is `CompressionFormat` or `CompressionFormatAndEngine`. Internally, this is used as an index into a table of pointers to the actual compression, decompression, or support routine. Currently in all versions of the NT kernel the first two pointers are NULL, and are never supposed to be used; the third pointer is to functions providing LZ compression support; the following five functions all return *STATUS_UNSUPPORTED_COMPRESSION*. Thus there are in principle eight pointers in the table.

However, the code that uses `CompressionFormat` or `CompressionFormatAndEngine` as an index checks that the index is neither zero nor one, thus avoiding a NULL pointer dereference, but only checks that the index is between 0 and 15 beyond that. Therefore it is possible to treat some of the code or data immediately after the pointer table as function pointers. If the code or data after the table can be interpreted as a pointer into the user mode portion of the address space, then this bug in principle allows user mode code to be called directly from kernel mode. The following disassembly is taken from the Windows Vista x64 RTM ntoskrnl.exe:

```
RtlGetCompressionWorkSpaceSize proc near
        sub     rsp, 28h
        test    cl, cl
        movzx   r9d, cl
        jz      short loc_140200E76    ; Check the index is not zero
        cmp     r9w, 1
        jz      short loc_140200E76    ; Check the index is not one
        test    r9b, 0F0h
        jz      short loc_140200E60    ; Check the index is less than 0x10
        mov     eax, 0C000025Fh
        jmp     short loc_140200E7B
; ---------------------------------------------------------------------------
loc_140200E60:
        movzx   eax, r9w
        lea     r9, RtlWorkSpaceProcs
        and     cx, 0FF00h             ; Mask off the format, and leave only the compression level
        call    qword ptr [r9+rax*8]   ; Call the relevant function from the table
        jmp     short loc_140200E7B
; ---------------------------------------------------------------------------
loc_140200E76:
        mov     eax, 0C000000Dh
loc_140200E7B:
        add     rsp, 28h
        retn
RtlGetCompressionWorkSpaceSize endp


RtlWorkSpaceProcs dq 0
        dq 0
        dq offset RtlCompressWorkSpaceSizeLZNT1
        dq offset RtlReserveChunkNS
        dq offset RtlReserveChunkNS
        dq offset RtlReserveChunkNS
        dq offset RtlReserveChunkNS
        dq offset RtlReserveChunkNS
LZNT1Formats dq 0F00000FFFh             ; With the above code, all the following quadwords
        dq 1000001002h                  ; can be treated as function pointers
        dq 7FF0000000Ch
        dq 8020000001Fh
        dq 0B00000020h
        dq 3F000003FFh
        dq 4000000402h
        dq 1FF0000000Ah
```

The question then raised is can this code path be executed whereby the input is controlled by an unprivileged user. The compression APIs are only used by two drivers on a default installation of an NT based system – the SMB driver and the NTFS driver. The SMB driver strictly controls the `CompressionFormat` and it always uses a value of 2 (*COMPRESSION_FORMAT_LZNT1*). The NTFS driver uses these APIs to support file and folder compression; whether a file is compressed or not is stored in what is referred to as the Flags element of the attribute header of any non-resident NTFS file record attribute by the Linux NTFS project documentation [10].

During compression or decompression by the driver, the Flag value is incremented and passed in as the `CompressionFormat`. It is not verified as being within the acceptable range, so there is now a potential code path to allow the exploitation of this bug. Seeing as the specific version of the core NT kernel binary loaded can easily be determined (whether ntoskrnl, ntkrnlmp, and so forth), and the correct image can be mapped into a user mode process thus allowing the pointers that could be dereferenced to be discovered by searching through the APIs that could be hit (in this case *RtlGetCompressionWorkSpaceSize, RtlCompressBuffer*, and *RtlDecompressBuffer*).

Thus it is possible to reliably determine if there are pointers into the user mode address space, and exactly what they are. There is an issue raised by which process context the pointer will be dereferenced in; as the NT kernel is asynchronous (even though it exposes synchronous functionality to user mode) lower level IRP processing may occur in arbitrary process contexts, in which case it cannot be guaranteed that the pointer will be dereferenced within the context of a process that the user controls. Two ways around this present themselves – either raise the priority of the target process as high as possible, or inject code at the required address into all the processes present on the system. Both of these would require elevated privileges.

The compression state of a file can be controlled by using *DeviceIoControl* on a handle to the file, specifying *FSCTL_SET_COMPRESSION* as the control code. However, the NTFS driver validates that the compression type supplied is one that is currently supported, so the Flag cannot be set to a bad value through this method. Since the Master File Table used to describe the file and folder layout under NTFS is itself just a file, it can be located easily on disk either by checking the Master Boot Record or by opening a handle to '$MFT' with no access rights and using *DeviceIoControl* with a control code of *FSCTL_GET_RETRIEVAL_POINTERS*.

The MFT can then be parsed, and a suitable file that is already compressed can be located, and its Flag element can be updated directly on disk. This obviously requires administrator privileges to perform the raw write to disk, and some effort must be made to deal with issues raised with respect to the Cache Manager. As disk access is cached by the Cache Manager, any raw writes to the MFT will leave the on disk data inconsistent with the cached data in memory. This will be a problem if the file has just been created, in which case it's entry in the MFT will most likely be in the cache. There are also two methods that can be used to deal with this – either attempt to force the cache to empty by allocating and using large amounts of memory, or by dismounting the target volume.

This could be used by removable media, whereby it is considerably easier to create a malformed file record with the potential to trigger the bug upon insertion of the media. This still requires physical access (at least of a sort), and administrator privileges are required to perform raw disk writes, with the consequence that this is not considered to be a security issue by Microsoft.

This bug is of interest because it is in code that is as old as the NT kernel itself, and raises the question of what other legacy kernel code in Windows NT has not had sufficient scrutiny. Another case in point is *MmGetSystemRoutineAddress*, whereby the search algorithm can cause an access violation with specific input; this only raises a security issue if it is directly exposed to user mode, for example by an anti-root kit scanner driver that exposed such kernel mode functionality to a user mode process, and the issue has been fixed in builds of the NT OS from Windows Server 2003 Service Pack 1 onwards.

## C.2 – Unvalidated structure initialization

The following code is used by a device driver to initialize a data structure on behalf of either another driver or a user mode process, being processed by the driver's *DispatchDeviceControl* routine. The address of the structure to be initialized and the size of it are controlled by the caller, so can be controlled from user mode. The parameters are not validated before some are used, so it is possible to zero an arbitrary area of memory and then have the function fail and exit gracefully.

```
pdataBuffer= dword ptr  8
Structure= dword ptr  0Ch
Count   = dword ptr  10h

         push    ebp
         mov     ebp, esp
         sub     esp, 444h
         mov     edx, [ebp+Count]
         mov     eax, [ebp+pdataBuffer]
         push    ebx
         mov     ecx, edx
         mov     ebx, ecx
         shr     ecx, 2
         push    esi
         mov     esi, [ebp+Structure]    ; Get the address of the structure to initialize
         push    edi
         mov     [ebp+var_400], eax
         xor     eax, eax
         mov     edi, esi
         rep stosd                       ; Zero out the contents
         mov     ecx, ebx
         and     ecx, 3
         rep stosb
         lea     eax, [edx-2C0h]
```

```
xor     edx, edx
mov     ecx, 2B8h
div     ecx                     ; Work out how many entries we can have
xor     ecx, ecx
cmp     esi, ecx
mov     [ebp+var_404], esi
lea     ebx, [esi+8]
mov     [ebp+var_408], eax
jz      InvalidParameter
cmp     [ebp+Count], ecx
jz      InvalidParameter
cmp     eax, ecx
jz      InvalidParameter
mov     eax, [ebp+var_400]
cmp     eax, ecx
jz      InvalidParameter
cmp     word ptr [eax], 4       ; Seeing as we control this pointer, we can leave this
jnz     InvalidParameter        ; function now, with some arbitrary data buffer having
mov     eax, [eax+4]            ; been zeroed
cmp     eax, ecx
mov     [ebp+var_3F0], eax
jz      SizeOk
mov     [ebp+var_3F8], ebx
jmp     UseMaxSize
```

The parameters used are taken without sufficient validation from the input buffer provided to *DeviceIoControl*. This particular issue could be used to overwrite a function pointer, provided a suitable pointer can be located, bearing in mind that if that pointer is then dereferenced in a different context then the kernel will bugcheck. It would also be possible to overwrite entries in the GDT, LDT, or IDT.

Pointers that are supposed to be in the user mode portion of the virtual address space can be validated as such by using *ProbeForRead* or *ProbeForWrite*. However, if the pointer can legitimately be to a virtual address in the kernel mode portion of the address space, for example if it is to process data on behalf of another driver, then an awkward problem is raised as there is no programmatic way to verify if the address is valid – the address must be trusted. Driver developers have sometimes resorted to using *MmIsAddressValid* to try and validate a pointer; however, the name of the routine is somewhat of a misnomer and it is only intended for internal use by the Memory Manager as it only returns a Boolean indicated whether access of the specified address will cause a page fault or not. As such, in general use, a page may have been swapped in or out by the Memory Manager between this function being called and any subsequent use of the address.

At any rate, if the driver is *only* supposed to process data from another driver, then the value of *RequestorMode* in the IRP to be processed is *KernelMode* (0).

## C.3 – An architectural flaw

Where the previous two examples highlighted poor input validation and a coding error, this example will highlight what appears to be an architectural flaw. The following code exposed by this particular driver's *DispatchDeviceControl* routine allows the memory at an arbitrary address to be overwritten with an arbitrary buffer, thus allowing an unprivileged user to overwrite kernel code or data structures.

The code that ends up calling the following code verifies that the buffer passed in from user mode is writeable, but performs no further validation of the data passed in. The buffer is then passed down to a routine that then processes the buffer passed in. The first DWORD in the buffer provides the function code, which is then used as an index into a jump table to select the appropriate function to further process the buffer; a snippet from one of the functions is provided:

```
SourceDescriptor   = dword ptr  8
Function           = dword ptr  0Ch
Destination        = dword ptr  10h
DestinationSize    = dword ptr  14h

        push    ebx
        mov     ebx, [esp+Function]
        cmp     ebx, MEMORY_OPERATION         ; Check if it is a memory operation
        push    ebp
        mov     ebp, [esp+4+SourceDescriptor] ; Get a pointer to the source buffer descriptor
        jnz     short NoAddress
        mov     ebx, [ebp+4]                  ; Get the source start address
NoAddress:
        mov     eax, [ebp+8]
        mov     edx, [eax]
        test    edx, edx                      ; Check that the buffer offset is non-zero
        jz      short InvalidParameter
        test    ebx, ebx                      ; Check the source buffer is a user mode address
        jl      short InvalidParameter
        mov     eax, [eax+4]                  ; Get the source end address
        cmp     eax, ebx
        jb      short InvalidParameter        ; Check the end is after the start
        mov     ecx, [esp+4+DestinationSize]
        sub     eax, ebx
        cmp     eax, ecx                      ; Make sure that the copy will not overflow the buffer
        jb      short SizeOk
        mov     eax, ecx                      ; Set the copy size to the size of the destination
SizeOk:
        test    eax, eax
        jz      short RequestProcessed        ; Make sure we are copying some bytes
        push    esi
        push    edi
        mov     edi, [esp+0Ch+Destination]    ; Destination address is an arbitrary address passed in
        mov     ecx, eax                      ; from the user supplied buffer
        lea     esi, [edx+ebx]                ; Address the relevant part of the target buffer
        shr     ecx, 2
        rep movsd                             ; DWORD aligned copy
        mov     ecx, eax
        and     ecx, 3
        rep movsb                             ; Copy the remaining bytes
        pop     edi
        pop     esi
        jmp     short RequestProcessed        ; And we're done
```

In this instance a somewhat complicated data structure is passed in, used to describe a particular subfunction to execute, and the parameters of the source buffer to be used. As there is no verification beyond the fact that the two buffers exist (source and destination), and that the source buffer has an end address higher than the start address, it is trivial for an unprivileged user to pass in a suitably crafted structure to overwrite an arbitrary area of memory with whatever they want.

This presents a more serious issue than the previous examples as it is completely reliable, and raises no issues with respect to being run in an arbitrary context, or only being able to write zeroes. As such, this can be used to overwrite a function pointer within the kernel so that it points to a user mode address, which would allow successful exploitation of this type of issue even on 64-bit versions of Windows.

## C.4 – Trusting user input

The following is a code snippet from a large and complex driver that can perform complicated data processing, and as such the bug took quite a lot of time reverse engineering the binary to find. It is another example of trusting a pointer that is controllable by a user, and presents somewhat more interesting challenge in terms of exploitation than the previous example.

```
SubFunction:
         test    esi, esi                        ; Check it is a valid handle
         jz      InvalidParameter
         test    ebp, ebp                        ; Check we have a non-NULL input buffer pointer
         jz      InvalidParameter
         mov     edi, [esp+9Ch+OutBuffer]
         test    edi, edi                        ; Check we have a non-NULL output buffer pointer
         jz      InvalidParameter
         cmp     edx, 20h                        ; Check the size of the input buffer is 0x20
         jnz     InvalidParameter
         cmp     edx, ecx                        ; Check the output buffer is the same size
         jnz     InvalidParameter
         mov     eax, [ebp+0Ch]
         test    eax, eax                        ; Verify the user controlled function index
         jz      short DefaultOp
         cmp     eax, 7Fh
         jbe     short ValidOp
         cmp     eax, 87h
         ja      short ValidOp
         mov     ecx, [ebp+10h]                  ; Get a user controlled pointer from the input buffer
         lea     eax, [esp+9Ch+var_80]           ; Address part of the thread's kernel mode stack
         cdq                                     ; This will set edx to 0xffffffff
         mov     dword ptr [ebp+0Ch], 0FFh
         mov     [ecx], eax                      ; Write the sign-extended stack address to the user
         mov     [ecx+4], edx                    ; specified buffer
         jmp     short ValidOp
 ; ---------------------------------------------------------------------------
 DefaultOp:
         mov     dword ptr [ebp+0Ch], 41h
 ValidOp:
         mov     edx, [ebp+10h]
         mov     eax, [ebp+0Ch]
```

The input and output buffers for all functions that use them are carefully validated before their use by the driver. However, in this case it is taking a user-controlled pointer and trusting that it is valid. This provides an eight byte overwrite but the target data will be over-written by a sign extended kernel stack address. As the value written to that address on the stack is not controllable by the user and attempting to execute at 0xffffffff will cause a bugcheck there is no point in trying to use this to overwrite the kernel stack pointer (as information about the kernel stack is disclosed by the bug). Overwriting a function pointer would be useless for exactly the same reason.

This leaves two options – either overwrite a pointer such as *MmUserProbeAddress* thus breaking the user-mode buffer verification provided by *ProbeForRead* and so forth, or overwrite an object pointer in pool memory with 0xffffffff with the result that any function pointers within that object will be dereferenced via a base address of 0xffffffff, with the result that they will be read from somewhere at the bottom of the user mode address space, and will be controllable by the user.

The difficulty with the second route is that it would require further information disclosure via reading arbitrary kernel memory addresses to find a suitable function pointer in pool memory; this would most likely rely on different bug within the same driver, or another vulnerability within the system. Thus, overwriting something like *MmUserProbeAddress* would be the easiest option, though the author has experienced stability issues when doing this under the 32-bit version of Windows Vista.

This particular case highlights the difficulty faced by using an approach purely based on fuzzing – for a fuzzer to hit this bug in a reasonable amount of time it would require a solid grasp of the data structures being used and what they are being used for to avoid situations where enormous amounts of test cases are generated yet the vast majority will be automatically rejected by the validation code within the driver. On the other hand, this particular driver highlights the usefulness of a smart fuzzer. Provided the data structures are described in suitable detail, and are fuzzed appropriately, fuzzing the code would be considerably quicker than manual examination due to the vast amount of functionality exposed by the driver.