

Subverting PatchGuard Version 2

12/2006

Skywing
skywing@valhallalegends.com
<http://www.nynaeve.net>

Contents

1	Foreword	2
2	Introduction	3
3	Notable Protection Mechanisms	5
3.1	Anti-Debug Code During Initialization	5
3.2	Expanded Set of DPC Routines	9
3.3	Self-Decrypting and Mutating System Integrity Check Routine	11
3.4	Obfuscation of System Integrity Check Calls via Structured Exception Handling	14
3.5	Disruption of Debug Register-Based Breakpoints	26
3.6	Misleading Symbol Names	27
3.7	Integrity Checks Performed During System Initialization	28
3.8	Overwriting PatchGuard Initialization Code Post-Boot	28
4	Bypass Techniques	30
4.1	Interception of <code>_C_specific_handler</code>	30
4.2	Interception of DPC Exception Registration	33
4.3	Interception of <code>PsInvertedFunctionTable</code>	35
4.4	Interception of <code>KiDebugTrapOrFault</code>	38
4.5	General Detect Bit Interception	41
4.6	Patching the Kernel Timer DPC Dispatcher	42
4.7	Searching for the PatchGuard DPC	43
4.8	TLB Desynchronization (Split TLB)	44
4.9	DPC Routine Patching	46
5	Subverting PatchGuard	48
6	Future Direction of PatchGuard and “Anti-Hack” Systems	55
7	Conclusion	58

Chapter 1

Foreword

Abstract: Windows Vista x64 and recently hotfixed versions of the Windows Server 2003 x64 kernel contain an updated version of Microsoft's kernel-mode patch prevention technology known as PatchGuard. This new version of PatchGuard improves on the previous version in several ways, primarily dealing with attempts to increase the difficulty of bypassing PatchGuard from the perspective of an independent software vendor (ISV) deploying a driver that patches the kernel. The feature-set of PatchGuard version 2 is otherwise quite similar to PatchGuard version 1; the SSDT, IDT/GDT, various MSRs, and several kernel global function pointer variables (as well as kernel code) are guarded against unauthorized modification. This paper proposes several methods that can be used to bypass PatchGuard version 2 completely. Potential solutions to these bypass techniques are also suggested. Additionally, this paper describes a mechanism by which PatchGuard version 2 can be subverted to run custom code in place of PatchGuard's system integrity checking code, all while leaving no traces of any kernel patching or custom kernel drivers loaded in the system after PatchGuard has been subverted. This is particularly interesting from the perspective of using PatchGuard's defenses to hide kernel mode code, a goal that is (in many respects) completely contrary to what PatchGuard is designed to do.

Thanks: The author would like to thank skape, bugcheck, and Alex Ionescu.

Disclaimer: This paper is presented in the interest of education and the furthering of general public knowledge. The author cannot be held responsible for any potential use (or misuse) of the information disclosed in this paper. While the author has attempted to be as vigilant as possible with respect to ensuring that this paper is accurate, it is possible that one or more mistakes might remain. If such an inaccuracy or mistake is located, the author would appreciate being notified so that the appropriate corrections can be made.

Chapter 2

Introduction

With x64 versions of the Windows kernel, Microsoft has attempted to take an aggressive stance [1] against the use of a certain class of techniques that have been frequently used to “extend” the kernel in potentially unsafe fashions on previous versions of Windows. This includes patching the kernel itself, hooking the kernel’s system service tables, redirecting interrupt handlers, and several other less common techniques for intercepting control of execution before the kernel is reached, such as the alternation of the system call target MSR.

The technology that Microsoft has deployed to prevent the unauthorized patching of the kernel that has been historically rampant on x86 is known as PatchGuard. This technology was initially released with Windows Server 2003 x64 Edition and Windows XP x64 Edition (known as PatchGuard version 1). The x64 editions of Windows Vista, and recently hotfixed versions of the Windows Server 2003 x64 kernel contain a newer version of the PatchGuard technology, known as PatchGuard version 2. The new version is designed to make it significantly more difficult for independent software vendors (ISVs) to deploy, in the field, solutions that involve patching the kernel after disabling the kernel patch protection mechanisms afforded by PatchGuard. The inner details of PatchGuard itself are much the same as they were in PatchGuard version 1 and thus will not be discussed in detail in this paper (excluding version 2’s improved anti-debugging and anti-patch technologies). A sufficiently interested reader wishing some more background information on the subject may find out more about how PatchGuard version 1 functions in Uninformed’s previous article [2] on the subject, “Bypassing PatchGuard on Windows x64”.

PatchGuard version 2 takes the original PatchGuard release and attempts to plug various holes in its implementation of an obfuscation-based anti-patching system. In this respect, it has met some mixed success and failure. Although the new PatchGuard version does, on the surface, appear to disable the major-

ity of the bypass techniques that had been proposed [2] as means to disable the original PatchGuard release, at least several of these techniques may be fairly trivially re-enabled through some minor alterations or additional new code. Furthermore, it is still possible to bypass PatchGuard version 2 without relying on dangerous (version-specific) constructs such as hard-coded offsets or code fingerprinting on frequently changing code. Additionally, aside from techniques that are based on disabling PatchGuard itself, there still exist several potential bypass mechanisms that have a strong potential to be “future-compatible” with new PatchGuard versions by virtue of preventing PatchGuard from even detecting that unauthorized alternations to the kernel have been made (and thus isolating themselves from any obfuscation-based changes to how PatchGuard’s system integrity check is invoked). To Microsoft’s credit, however, the resilience of PatchGuard to being debugged and analyzed has been significantly improved (at least with regard to certain key steps, such as initialization at boot time).

Chapter 3

Notable Protection Mechanisms

PatchGuard version 2 implements a variety of anti-debug, anti-analysis, and obfuscation mechanisms that are worth covering. Not all of PatchGuard's defenses are covered in detail in this paper, and those mechanisms (such as the obfuscation of PatchGuard's internal data structures) that are at least the same in principle as the previous PatchGuard release (and were already disclosed by Uninformed's previous article [2] on PatchGuard) are additionally not covered by this paper.

3.1 Anti-Debug Code During Initialization

That being said, there are still a number of interesting things to examine as far as PatchGuard's protection mechanisms go. Many of these techniques are on their own worthy of discussion, simply from the perspective of their worth as general debug/analysis protection mechanisms. PatchGuard version 2 begins as an appended addition to the `nt!SepAdtInitializePrivilegeAuditing` routine in the kernel (PatchGuard version 2 continues the tactic of misleading and/or bogus function names that PatchGuard version 1 introduced). This routine is responsible for performing the bulk of PatchGuard's initialization, including setting up the encrypted PatchGuard context data structures. Unlike PatchGuard version 1, the initialization routine is littered with statements that are intended to frustrate debugging, such as the following construct that enters an infinite loop if a debugger is connected (this particular construct is used in many places during PatchGuard initialization):

```
cli
```

```

cmp     cs:KdDebuggerNotPresent, r12b
jnz    short continue_initialization_1
infinite_loop_1:
jmp    short infinite_loop_1
sti

```

This particular approach is not all that robust as currently implemented in PatchGuard version 2 today. It remains relatively easy to detect these references to `nt!KdDebuggerNotPresent` ahead of time, and disable them. If Microsoft had elected to corrupt the execution context in a creative way on each occurrence (such as zeroing some registers, or otherwise arranging for a failure to occur much later on if a debugger was attached) before entering the forever loop, then these constructs might have been slightly effective as far as anti-debugging goes.

Other constructs include the highly obfuscated selection of a randomized set of bogus pool tags used to allocate PatchGuard data structures. Like PatchGuard version 1, PatchGuard version 2 uses a randomly chosen bogus pool tag and randomly adjusted allocation sizes in an attempt to frustrate easy detection of the PatchGuard context in-memory by scanning pool allocations. The following is an example of one of the sections of code used by PatchGuard to randomly pick a pool tag and random allocation delta from a list of possible pool tags. The actual allocation size is the random allocation delta plus the minimum size of the PatchGuard context structure, truncated at 2048 bytes. Here, the `rdtsc` instruction is used for random number generation purposes (readers that have examined the previous [2] PatchGuard paper may recognize this random number generation construct; it is used throughout PatchGuard anywhere a random quantity is required).

```

;
; Generate a random value, using rdtsc.
;
lea     ebx, [r14+r13+200h]
mov     dword ptr [rsp+0A28h+Timer], ebx
rdtsc
mov     r10, qword ptr [rsp+0A28h+arg_5F8]
shl     rdx, 20h
mov     r11, 7010008004002001h
or      rax, rdx
mov     rcx, r10
xor     rcx, rax
lea     rax, [rsp+0A28h+var_2C8]
xor     rcx, rax
mov     rax, rcx
ror     rax, 3
xor     rcx, rax
mov     rax, r11
mul     rcx
mov     [rsp+0A28h+var_2C8], rax
xor     eax, edx
mov     [rsp+0A28h+arg_1F0], rdx
;

```

```

; This is essentially a switch(eax & 7), where eax
; is a random value. Each case statement selects
; a unique obfuscated pooltag value. The magical
; 0x432E10h constant below is the offset used to
; jump to the switch case handler selected.
;
lea    rdx, cs:400000h
and    eax, 7
mov    ecx, [rdx+rax*4+432E10h]
add    rcx, rdx
jmp    rcx
-----
mov    dword ptr [rsp+0A28h+var_9D8], 0D098D0D8h
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
ror    r9d, 6
jmp    DoAllocation
-----
mov    dword ptr [rsp+0A28h+var_9D8], 0B2AD31A1h
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
rol    r9d, 1
jmp    DoAllocation
-----
mov    dword ptr [rsp+0A28h+var_9D8], 85B5910Dh
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
ror    r9d, 2
jmp    DoAllocation
-----
mov    dword ptr [rsp+0A28h+var_9D8], 0A8223938h
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
xor    r9d, 3
ror    r9d, 0Fh
jmp    DoAllocation
-----
mov    dword ptr [rsp+0A28h+var_9D8], 67076494h
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
rol    r9d, 4
jmp    DoAllocation
-----
mov    dword ptr [rsp+0A28h+var_9D8], 288C49EDh
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
ror    r9d, 5
jmp    DoAllocation
-----
mov    dword ptr [rsp+0A28h+var_9D8], 4E574672h
mov    r9d, dword ptr [rsp+0A28h+var_9D8]
xor    r9d, 6
ror    r9d, 18h
jmp    DoAllocation
-----
DoAllocation:
;
; Get another random value (for the allocation size),
; and deobfuscate the pooltag value that was selected.
;
; Eventually, the value ending up in "r9d" is used as
; the pooltag value.
;

```

```

rdtsc
shl    rdx, 20h
mov    rcx, r10
or     rax, rdx
xor    rcx, rax
lea    rax, [rsp+0A28h+var_858]
xor    rcx, rax
mov    rax, rcx
ror    rax, 3
xor    rcx, rax
mov    rax, r11
mul    rcx
mov    [rsp+0A28h+ValueName], rdx
mov    r9, rax
mov    [rsp+0A28h+var_858], rax
xor    r9d, edx
mov    eax, 4EC4EC4Fh
mov    ecx, r9d
mul    r9d
shr    edx, 3
shr    r9d, 5
mov    r8d, r9d
mov    eax, 4EC4EC4Fh
imul   edx, 1Ah
sub    ecx, edx
add    ecx, 61h
shl    ecx, 8
mul    r9d
shr    edx, 3
shr    r9d, 5
mov    eax, 4EC4EC4Fh
imul   edx, 1Ah
sub    r8d, edx
mul    r9d
add    r8d, 41h
mov    eax, 4EC4EC4Fh
or     r8d, ecx
shr    edx, 3
mov    ecx, r9d
shr    r9d, 5
shl    r8d, 8
imul   edx, 1Ah
sub    ecx, edx
add    ecx, 61h
or     ecx, r8d
shl    ecx, 8
mul    r9d
shr    edx, 3
imul   edx, 1Ah
sub    r9d, edx
add    r9d, 41h
or     r9d, ecx
rdtsc
shl    rdx, 20h
mov    rcx, r10
mov    r8d, r9d      ; Tag
or     rax, rdx

```

```

xor     rcx, rax
lea    rax, [rsp+0A28h+var_2E8]
xor     rcx, rax
mov     rax, rcx
ror     rax, 3
xor     rcx, rax
mov     rax, r11
mul     rcx
;
; Perform the actual allocation. We're requesting NonPagedPool,
; with the random pooltag selected by the deobfuscation and
; randomization code above. The actual size of the block being
; allocated here is given in ebx, with a random "fuzz factor" that
; is added to this minimum allocation size, then truncated to a
; maximum of 2047 bytes.
;
xor     ecx, ecx           ; PoolType
mov     [rsp+0A28h+var_310], rdx
xor     rdx, rax
mov     [rsp+0A28h+var_2E8], rax
and     edx, 7FFh
add     edx, ebx           ; NumberOfBytes
call    ExAllocatePoolWithTag

```

3.2 Expanded Set of DPC Routines

Other protection mechanisms used in PatchGuard version 2 include an expanded set of DPC routines used to arrange for the execution of the PatchGuard integrity check routine. Recall that in PatchGuard version 1, there existed a set of three possible DPC routines. In PatchGuard version 2, this set of potential DPC routines that can be repurposed for PatchGuard's use has been expanded to ten possibilities. One DPC routine is selected at boot time from this set of ten possibilities, and from that point is used for all further PatchGuard operations for the lifetime of the session. The fact that only one DPC routine is used in a particular Windows session is a weakness that is inherited from the previous PatchGuard version (as the reader will discover, eventually comes in handy if one is set on bypassing PatchGuard). The DPC routine to be used for the current boot session is selected in the `nt!SepAdtInitializePrivilegeAuditing` routine, much the same as how the bogus pooltag to be used for all PatchGuard allocations is selected:

```

INIT:0000000000832741:
PatchGuard_Pick_Random_DPC:
;
; Use the time stamp counter as a random seed.
;
rdtsc
shl     rdx, 20h
mov     rcx, r15
or      rax, rdx

```

```

xor     rcx, rax
lea    rax, [rsp+0A28h+var_360]
xor     rcx, rax
mov     rax, rcx
ror     rax, 3
xor     rcx, rax
mov     rax, 7010008004002001h
mul     rcx
mov     [rsp+0A28h+var_360], rax
mov     rcx, rdx
mov     qword ptr [rsp+0A28h+arg_260], rdx
xor     rcx, rax
mov     rax, 0CCCCCCCCCCCCCDh
mul     rcx
shr     rdx, 3
;
; The resulting value in 'rax' is the index into a switch jump table
; that is used to locate the DPC to be repurposed for initiating
; PatchGuard checks for this session.
;
lea     rax, [rdx+rdx*4]
add     rax, rax
sub     rcx, rax
jmp     PatchGuard_DPC_Switch

INIT:0000000000832317:
PatchGuard_DPC_Switch:
;
; The address of the case statement is formed by adding the image base (here,
; being loaded into 'rdx') and an RVA in the table indexed by rax.
;
lea     rdx, cs:400000h
mov     eax, ecx
;
; Locate the case statement RVA by indexing the jump offset table.
;
mov     ecx, [rdx+rax*4+432E60h]
;
; Add it to the image base to form a complete 64-bit address.
;
add     rcx, rdx
;
; Execute the case handler.
;
jmp     rcx

;
; The set of case statements are as follows:
;
; Each case statement block simply loads the full 64-bit address
; of the DPC routine to be repurposed for PatchGuard checks into
; the r8 register. This register is later stored into one of
; PatchGuard's internal data structures for future use.
;
lea     r8, CmpEnableLazyFlushDpcRoutine

```

```

jmp     short PatchGuardSelectDpcRoutine
lea    r8, _CmpLazyFlushDpcRoutine
jmp    short PatchGuardSelectDpcRoutine
lea    r8, ExpTimeRefreshDpcRoutine
jmp    short PatchGuardSelectDpcRoutine
lea    r8, ExpTimeZoneDpcRoutine
jmp    short PatchGuardSelectDpcRoutine
lea    r8, ExpCenturyDpcRoutine
jmp    short PatchGuardSelectDpcRoutine
lea    r8, ExpTimerDpcRoutine
jmp    short PatchGuardSelectDpcRoutine
lea    r8, TopTimerDispatch
jmp    short PatchGuardSelectDpcRoutine
lea    r8, TopIrpStackProfilerTimer
jmp    short PatchGuardSelectDpcRoutine
lea    r8, KiScanReadyQueues
jmp    short PatchGuardSelectDpcRoutine
lea    r8, PopThermalZoneDpc
;
; (fallthrough from last case statement)
;
INIT:0000000000832800:
PatchGuardSelectDpcRoutine:
xor     ecx, ecx
;
; Store the DPC routine into r14+178. r14 points to one of
; the PatchGuard context structures in this particular instance.
;
mov     [r14+178h], r8

```

Much like PatchGuard version 1, each of the DPCs selected for use in launching the PatchGuard integrity checks has a legitimate function. Furthermore, the DPC routines are ones that are important for normal system operation, thus it is not possible for one to simply detect all DPCs that refer to these DPC routines and cancel them. Instead, much as with PatchGuard version 1, if one wanted to go the route of blocking PatchGuard's DPC, a mechanism to detect the particular PatchGuard DPC (as opposed to the legitimate system invocations thereof) must be developed. This aspect of PatchGuard's obfuscation mechanisms is relatively similar to version 1, other than the logical extension to ten DPCs instead of three DPCs.

3.3 Self-Decrypting and Mutating System Integrity Check Routine

PatchGuard version 2 also inherits the capability to encrypt its datastructures and executable code in-memory from version 1. This is a defensive mechanism that intends to make it difficult for an attacker to perform a classic *egghunt* style search, wherein the attacker has devised an identifiable signature for PatchGuard data structures that can be used to locate it in an exhaustive non-paged-

pool memory scan. From this perspective, the obfuscation and encryption of PatchGuard code and data structures that are dynamically allocated is still a reasonably strong defensive mechanism. Unfortunately for Microsoft, though, some of the data structures linking to PatchGuard are internal system structures (such as a KDPC and associated KTIMER used to kick off PatchGuard execution). This presents a weakness that could be potentially used to identify PatchGuard structures in memory (which will be explored in more detail later).

The encryption of PatchGuard’s internal context structures was covered by Uninformed’s original paper [2] on the subject. However, the mechanism by which PatchGuard obfuscates its system integrity checking and validation routines was not discussed. This mechanism is novel enough to warrant some explanation. The technique used to obfuscate PatchGuard’s executable code in-memory involves two layers of decryption/deobfuscation functions, each of which decrypts the next layer. After both layers have run their course, PatchGuard’s validation routines are plaintext in memory and are then directly executed.

The first decryption layer is the code block that is called from the repurposed DPC routine selected by PatchGuard at boot time. Its job is to decrypt itself (in 8 byte chunks, starting with the second instruction in the function). After the decryption of the this code block is complete, the decryption stub continues on to decrypt a second code block (the actual PatchGuard validation routine). When this second decryption/deobfuscation cycle is completed, the decryption stub then executes the actual PatchGuard system integrity check routine.

As noted above, the first task for the decryption stub is to decrypt itself. Except for the first instruction of the stub, the entire routine is encrypted when entered. The first instruction encrypts itself and decrypts the next instruction. The following instruction decrypts the next two instructions, and soforth. This is accomplished by a series of four byte long instructions that xor an eight byte quantity with a decryption key (initially starting at the current instruction pointer - here, `rcx` and `rip` always have the same value. An example of how this process works is illustrated below:

```

;
; rcx: Address of the decryption stub (same as rip)
; rdx: Decryption key
;
Breakpoint 5 hit
nt!ExpTimeRefreshDpcRoutine+0x20a:
fffff800'0112c98b ff5538      call    qword ptr [rbp+38h]
0: kd> u poi(rbp+38)
;
; Note that beyond the first instruction, the decryption stub is initially seemingly
; garbage data (though it has an apparent pattern to it, since it is merely obfuscated
; by xor).
;
fffffadf'f6e6d55d f0483111      lock xor qword ptr [rcx],rdx
fffffadf'f6e6d561 88644d68      mov     byte ptr [rbp+rcx*2+68h],ah
fffffadf'f6e6d565 62           ???

```

```

fffffadf'f6e6d566 d257df      rcl    byte ptr [rdi-21h],cl
fffffadf'f6e6d569 88644d78    mov    byte ptr [rbp+rcx*2+78h],ah
fffffadf'f6e6d56d 62          ???
fffffadf'f6e6d56e d257ef      rcl    byte ptr [rdi-11h],cl
fffffadf'f6e6d571 88644d48    mov    byte ptr [rbp+rcx*2+48h],ah
0: kd> t
fffffadf'f6e6d55d f0483111    lock xor qword ptr [rcx],rdx
0: kd> r
;
; Note the initial input arguments. rcx points to the decryption stub's first
; instruction (same as rip), and rdx is the decryption key.
;
rax=ffffadfff6e6d55d rbx=fffff8000116d894 rcx=ffffadfff6e6d55d
rdx=601c55c0cf06e32a rsi=fffff800003c7ad0 rdi=0000000000000003
rip=ffffadfff6e6d55d rsp=fffff800003c51f8 rbp=fffff800003c7ad0
r8=0000000000000000 r9=0000000000000000 r10=0000000001c7111e
r11=fffff800003c54c0 r12=fffff8000116d858 r13=fffff800003c5370
r14=fffff80001000000 r15=fffff800003c60a0
iop1=0          nv up ei pl zr na po nc
cs=0010  ss=0018  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
fffffadf'f6e6d55d f0483111    lock xor qword ptr [rcx],rdx ds:002b:ffffadff'f6e6d55d=684d6488113148f0
;
; After allowing the decryption of the stub to progress, we see the stub in its executable
; form. The first instruction is initially re-encrypted after executed, but a later
; instruction in the decryption stub returns the initial instruction to its executable,
; plaintext form.
;
0: kd> u FFFFFADFF6E6D55D
;
; The 'lock' prefix is used to create a four byte instruction when there
; is no immediate offset specified (a MASM limitation, as the assembler
; will convert a zero offset into the shorter form with no immediate
; offset operand).
;
fffffadf'f6e6d55d f0483111    lock xor qword ptr [rcx],rdx
fffffadf'f6e6d561 48315108    xor    qword ptr [rcx+8],rdx
fffffadf'f6e6d565 48315110    xor    qword ptr [rcx+10h],rdx
fffffadf'f6e6d569 48315118    xor    qword ptr [rcx+18h],rdx
fffffadf'f6e6d56d 48315120    xor    qword ptr [rcx+20h],rdx
fffffadf'f6e6d571 48315128    xor    qword ptr [rcx+28h],rdx
fffffadf'f6e6d575 48315130    xor    qword ptr [rcx+30h],rdx
fffffadf'f6e6d579 48315138    xor    qword ptr [rcx+38h],rdx
0: kd> u
fffffadf'f6e6d57d 48315140    xor    qword ptr [rcx+40h],rdx
fffffadf'f6e6d581 48315148    xor    qword ptr [rcx+48h],rdx
;
; Because the initial instruction was re-encrypted after it was executed,
; we need to decrypt it again.
;
fffffadf'f6e6d585 3111      xor    dword ptr [rcx],edx
fffffadf'f6e6d587 488bc2    mov    rax,rdx
fffffadf'f6e6d58a 488bd1    mov    rdx,rcx
fffffadf'f6e6d58d 8b4a4c    mov    ecx,dword ptr [rdx+4Ch]
;
; The following is the second stage decryption loop. It's purpose is to

```

```

; decrypt a code block following the current decryption stub in memory.
;
; This code block is then executed (it is responsible for performing the
; actual PatchGuard system verification checks).
;

fffffadb'f6e6d590 483144ca48    xor     qword ptr [rdx+rcx*8+48h],rax
fffffadb'f6e6d595 48d3c8      ror     rax,cl
0: kd> u
fffffadb'f6e6d598 e2f6      loop   fffffadb'f6e6d590
;
; After decryption of the second block is completed, we'll execute it
; by jumping to it. Doing so kicks off the system verification routine
; that verifies system integrity, arranging for a bug check if not,
; otherwise arranging for itself to be executed again several minutes
; later.
;
fffffadb'f6e6d59a 8b8288010000  mov    eax,dword ptr [rdx+188h]
fffffadb'f6e6d5a0 4803c2      add    rax,rdx
fffffadb'f6e6d5a3 ffe0      jmp    rax

```

Prior to returning control, the verification routine re-encrypts itself so that it does not remain in plaintext after the first invocation. In addition, PatchGuard also re-randomizes the key used to encrypt and decrypt the PatchGuard validation routine on each execution, such that a would-be attacker has a frequently mutating target. Due to this behavior, the PatchGuard validation routine changes appearance (in encrypted form) in-memory every few minutes, which is the period of PatchGuard's validation checks. While this is perhaps an admirable effort on Microsoft's part as far as interesting obfuscation techniques go, it turns out that there are much easier avenues of attack that can be used to disable PatchGuard without having to involve oneself in the search of a target that alters its appearance in-memory every few minutes.

3.4 Obfuscation of System Integrity Check Calls via Structured Exception Handling

Much like PatchGuard version 1, this version of PatchGuard utilizes structured exception handling (SEH) support as an integral part of the process used to kick off execution of the system integrity check routine. The means by which this is accomplished have changed somewhat since the last PatchGuard version. In particular, there are several layers of obfuscation in each PatchGuard DPC that are used to shroud the actual call to the integrity check routine. In an effort to make matters more difficult for would-be attackers, the exact details of the obfuscation used vary between each of the ten DPCs that may be repurposed for use with PatchGuard. They all exhibit a common pattern, however, which can be described at a high level.

The first step in invoking the PatchGuard system integrity checking routine is a KTIMER with an associated KDPC (indicating a DPC callback routine to be called when the timer lapses) associated with it. This timer is primed for single-shot execution in an interval on the order of several minutes (with a random fuzz factor delta applied to increase the difficulty of performing a classic egg hunt style attack to locate the KTIMER in non-paged pool). The DPC routine indicated with the KDPC that is associated with PatchGuard's KTIMER is one of the set of ten legitimate DPC routines that may be repurposed for use with PatchGuard. The means by which this particular invocation of the DPC routine is distinguished from a legitimate system invocation of the DPC routine in question is by the use of a deliberately invalid kernel pointer as one of the arguments to the DPC routine.

The prototype for a DPC routine is described by PKDEFERRED_ROUTINE:

```
typedef
VOID
(*PKDEFERRED_ROUTINE) (
    IN struct _KDPC *Dpc,          // pointer to parent DPC
    IN PVOID DeferredContext,    // arbitrary context - assigned at DPC initialization
    IN PVOID SystemArgument1,    // arbitrary context - assigned when DPC is queued
    IN PVOID SystemArgument2     // arbitrary context - assigned when DPC is queued
);
```

Essentially, a DPC is a callback routine with a set of user-defined context parameters whose interpretation is entirely up to the DPC routine itself. The standard use for context arguments in callback functions is to use them to point to a larger structure which contains information necessary for the callback routine to function, and this is exactly how the ten DPC routines that can be used by PatchGuard regard the DeferredContext argument during legitimate execution. It is this usage of the DeferredContext argument which allows PatchGuard to trigger its execution for each of the ten DPC routines via an exception; PatchGuard arranges for a bogus DeferredContext value to be passed to the DPC routine when it is called. The first time that the DPC routine tries to dereference the DPC-specific structure referred to by DeferredContext, an exception occurs (which transfers control to the exception dispatching system, and eventually to PatchGuard's integrity check routine). While this may seem simple at first, if the reader is familiar with kernel mode programming, then there should be a couple of red flags set off by this description; normally, it is not possible to catch bogus memory references at DISPATCH_LEVEL or above with SEH (usually, one of the PAGE_FAULT_IN_NONPAGED_AREA or IRQL_NOT_LESS_OR_EQUAL bugchecks will be raised, depending on whether the bogus reference was to a reserved non-paged region or a paged-out pagable memory region). As a result, one would expect that PatchGuard would be putting the system at risk of randomly bugchecking by passing bogus pointers that are referenced at DISPATCH_LEVEL, the IRQL at which DPC routines run. However, PatchGuard has a couple of tricks up its metaphorical sleeve. It takes advantage of an implementation-specific detail

of the current generation of x64 processors shipped by AMD in order to form kernel mode addresses that, while bogus, will not result in a page fault when referenced. Instead, these bogus addresses will result in a general protection fault, which eventually manifests itself as a `STATUS_ACCESS_VIOLATION` SEH exception. This path to raising a `STATUS_ACCESS_VIOLATION` exception does in fact work even at `DISPATCH_LEVEL`, thus allowing PatchGuard to provide safe bogus pointer values for the `DeferredContext` argument in order to trigger SEH dispatching without risking bringing the system down with a bugcheck.

Specifically, the implementation detail that PatchGuard relies upon relates to the 48-bit address space limitation in AMD's Hammer family of processors [4]. Current AMD processors only implement 48 bits of the 64-bit address space presented by the x64 architecture. This is accomplished by requiring that bits 63 through the most significant bit implemented by the processor (current AMD processors implement 48 bits) of any given address be set to either all ones or all zeros. An address of this form is defined to be a *canonical address*, or a well-formed address. Attempts to reference addresses that are not canonical as defined by this definition result in the processor immediately raising a general protection fault. This restriction on the address space essentially splits the usable address space into two halves; one region at the high end of the address space, and one region at the low end of the address space, with a no-mans-land in between the two. Windows utilizes this split to divide user mode from kernel mode, with the high end of the address space being reserved for kernel mode usage and the low end of the address space being reserved for user mode usage. PatchGuard takes advantage of this processor-mandated no-mans-land to create bogus pointer values that can be safely dereferenced and caught by SEH, even at high IRQLs.

All of the DPC routines that are in the set which may be repurposed for use by PatchGuard dereference the `DeferredContext` argument as the first part of work that does not involve shuffling stack variables around. In other words, the first real work involved in any of the PatchGuard-enabled DPC routines is to touch a structure or variable pointed to by the `DeferredContext` argument. In the execution path of PatchGuard attempting to trigger a system integrity check, the `DeferredContext` argument is invalid, which eventually results in an access violation exception that is routed to the SEH registrations for the DPC routine. If one examines any of the PatchGuard DPC routines, it is clear that all of them have several overlapping SEH registrations (a construct that normally indicates several levels of nested try/except and try/finally constructs):

```
1: kd> !fnseh nt!ExpTimeRefreshDpcRoutine
nt!ExpTimeRefreshDpcRoutine Lc8 0A,02 [EU ] nt!_C_specific_handler (C)
> fffff8000100358a La (fffff8000112c830 -> fffff80001000000)
> fffff8000100358a Lc (fffff8000112c870 -> fffff80001003596)
> fffff8000100358a L16 (fffff8000112c8a0 -> fffff80001000000)
> fffff8000100358a L18 (fffff8000112c8f0 -> fffff800010035a2)
```

These SEH registrations are integral to the operation of PatchGuard's system integrity checks. The specifics of how each handler registration work differ for each DPC routine (in an attempt to frustrate attempts to reverse engineer them), but the general idea is that each registered handler performs a portion of the work necessary to set up a call to the PatchGuard integrity check routine. This work is divided up among four different exception/unwind handlers in an effort to make it difficult to understand what is going on, but ultimately the end result is the same for each of the DPC routines; one of the exception/unwind handlers ends up making a direct call to the system integrity check decryption stub in-memory. The decryption stub decrypts itself, and then decrypts the PatchGuard check routine, following with a transfer of control to the integrity check routine so that PatchGuard can inspect various protected registers, MSRs, and kernel images (such as the kernel itself) for unauthorized modification.

Additionally, all of the PatchGuard DPCs have been enhanced to obfuscate the DPC routine arguments in stack variables (whose exact stack displacement varies from DPC routine to DPC routine, and furthermore between kernel flavor to kernel flavor; for example, the multiprocessor and uniprocessor kernel builds have different stack frame layouts for many of the PatchGuard DPC routines). Recall that in the x64 calling convention, the first four arguments are passed via registers (`rcx`, `rdx`, `r8`, and `r9` respectively). Each PatchGuard DPC routine takes special care to save away significant register arguments onto the stack (in an obfuscated form). Several of the arguments remain obfuscated until just before the decryption stub for the system integrity check routine is called, in an effort to make it difficult for third parties to patch into the middle of a particular DPC routine and easily access the original arguments to the DPC. This is presumably designed in an attempt to make it more difficult to differentiate DPC invocations that perform the DPC routine's legitimate function from DPC invocations that will call PatchGuard. It also makes it difficult, though not impossible, for a third party to recover the original arguments to the DPC routine from the context of any of the exception handlers registered to the DPC routine in a generalized fashion.

This obfuscation of arguments can be clearly seen by disassembling any of the PatchGuard DPC routines. For example, when looking at `ExpTimeRefreshDpcRoutine`, one can see that the routine saves away the `Dpc` (`rcx`) and `DeferredContext` (`rdx`) arguments on the stack, rotates them by a magical constant (this constant differs for each DPC routine flavor and is used to further complicate the task of recovering the original DPC arguments in a generalized fashion), and then overwrites the original argument registers:

```
0: kd> uf nt!ExpTimeRefreshDpcRoutine
;
; On entry, we have the following:
;
; rcx -> Dpc
; rdx -> DeferredContext (if this is being called for
;                               PatchGuard, then DeferredContext
```

```

;                                     is a bogus kernel pointer).
; r8 -> SystemArgument1
; r9 -> SystemArgument2
;
nt!ExpTimeRefreshDpcRoutine:
;
; r11 is used as an ephemeral frame pointer here.
;
; Ephemeral frame pointers are an x64-specific compiler
; construct, wherein a volatile register is used as a
; frame pointer until the first function call is made.
;
fffff800'01003540 4c8bdc          mov     r11, rsp
fffff800'01003543 4881ecc8000000 sub     rsp, 0C8h
fffff800'0100354a 48896424460      mov     qword ptr [rsp+60h], rsp
;
; This DPC routine does not use SystemArgument1 or
; SystemArgument2. As a result, it is free to overwrite
; these argument registers immediately without preserving
; their value.
;
; r8 = Dpc
; rcx = Dpc
; rdx = DeferredContext
;
fffff800'0100354f 4c8bc1          mov     r8, rcx
fffff800'01003552 48895424448      mov     qword ptr [rsp+48h], rdx
;
; Set [rsp+20h] to zero. This is a state variable that is
; used by the exception/unwind scope handlers in order to
; coordinate the PatchGuard execution process across the
; set of four exception/unwind scope handlers associated
; with this section of code.
;
fffff800'01003557 4533c9          xor     r9d, r9d
fffff800'0100355a 44894c24240      mov     dword ptr [rsp+20h], r9d
;
; PatchGuard zeros out various key fields in the DPC.
; This is an attempt to make it difficult to locate the DPC
; in-memory from the context of an exception handler called
; when a PatchGuard DPC accesses the bogus DeferredContext
; argument. Specifically, PatchGuard zeros the Type and
; DeferredContext fields of the KDPC structure, shown below:
;
; 0: kd> dt nt!_KDPC
; +0x000 Type           : UChar
; +0x001 Importance     : UChar
; +0x002 Number         : UChar
; +0x003 Expedite       : UChar
; +0x008 DpcListEntry   : _LIST_ENTRY
; +0x018 DeferredRoutine : Ptr64
; +0x020 DeferredContext : Ptr64 Void
; +0x028 SystemArgument1 : Ptr64 Void
; +0x030 SystemArgument2 : Ptr64 Void
; +0x038 DpcData        : Ptr64 Void
;
; Dpc->Type = 0

```

```

;
fffff800'0100355f 448809      mov     byte ptr [rcx],r9b
;
; Dpc->DeferredContext = 0
;
fffff800'01003562 4c894920      mov     qword ptr [rcx+20h],r9
;
; Here, the DPC loads [r11-20h] with an obfuscated
; copy of the DeferredContext argument (rotated
; left by 0x34 bits).
;
; Recall that rsp == r11+0xc8, so this location
; can also be aliased by [rsp+0A8h].
;
; [rsp+0A8h] -> ROL(DeferredContext, 0x34)
;
fffff800'01003566 488bc2      mov     rax,rdx
fffff800'01003569 48c1c034      rol     rax,34h
fffff800'0100356d 498943e0      mov     qword ptr [r11-20h],rax
;
; Similarly, the DPC loads [r11-48h] with an
; obfuscated copy of the Dpc argument (rotated
; right by 0x48 bits).
;
; This location may be aliased as [rsp+80h].
;
; [rsp+80h] -> ROR(Dpc, 0x48)
;
fffff800'01003571 488bc1      mov     rax,rcx
fffff800'01003574 48c1c848      ror     rax,48h
fffff800'01003578 498943b8      mov     qword ptr [r11-48h],rax
;
; The following register context is now in place:
;
; r8          = Dpc
; rcx         = Dpc
; rdx         = DeferredContext
; rax         = ROR(Dpc, 0x48)
; [rsp+0A8h] = ROL(DeferredContext, 0x34)
; [rsp+80h]  = ROR(Dpc, 0x48)
;
; The DPC routine destroys the contents of rcx by
; zero extending it with a copy of the low byte of
; the DeferredContext value.
;
fffff800'0100357c 0fb6ca      movzx  ecx,dl
;
; The DPC routine destroys the contents of r8 with
; a right shift (unlike a rotate, the incoming left
; bits are simply zero filled instead of set to the
; rightmost bits being shifted off. The rightmost
; bits are thus lost forever, destroying the r8
; register as a useful source of the Dpc argument.
;
fffff800'0100357f 49d3e8      shr    r8,c1
;
; r8 is saved away on the stack, but it is no longer

```

```

; directly useful as a way to locate the Dpc argument
; due to the destructive right shift above.
;
fffff800'01003582 4c898424d8000000 mov     qword ptr [rsp+0D8h],r8
;
; r8      = Dpc >> (UCHAR)DeferredContext
; rcx     = (UCHAR)DeferredContext
; rdx     = DeferredContext
; rax     = ROR(Dpc, 0x48)
; [rsp+0A8h] = ROL(DeferredContext, 0x34)
; [rsp+80h] = ROR(Dpc, 0x48)
;
; Here, we temporarily deobfuscate the DeferredContext
; argument stored at [r11-20h] above.  In this particular
; instance, rdx also happens to contain the deobfuscated
; DeferredContext value, but not all instances of
; PatchGuard's DPC routines share this property of
; retaining a plaintext copy of DeferredContext in rdx.
;
fffff800'0100358a 498b43e0      mov     rax,qword ptr [r11-20h]
fffff800'0100358e 48c1c834      ror     rax,34h
;
; Now, we have the following context in place:
;
; r8      = Dpc >> (UCHAR)DeferredContext
; rcx     = (UCHAR)DeferredContext
; rdx     = DeferredContext (* But not valid for
;                               all DPC routines.)
; rax     = DeferredContext
; [rsp+0A8h] = ROL(DeferredContext, 0x34)
; [rsp+80h] = ROR(Dpc, 0x48)
;
; The next step is to dereference the DeferredContext value.
; For a legitimate DPC invocation, this operation is harmless;
; the DeferredContext value would point to valid kernel memory.
;
; For PatchGuard, however, this triggers an access violation
; that winds up with control being transferred to the exception
; handlers registered to the DPC routine.
;
fffff800'01003592 8b00      mov     eax,dword ptr [rax]

```

At this point, it is necessary to investigate the various exception/unwind handlers registered to the DPC routine in order to determine what happens next. Most of these handlers can be skipped as they are nothing more than minor layers of obfuscation that, while differing significantly between each DPC routine, have the same end result. One of the exception/unwind handlers, however, makes the call to PatchGuard's integrity check, and this handler is worthy of further discussion. Because the exception registrations for all of the PatchGuard DPC routines make use of `nt!_C_specific_handler`, the scope handlers conform to a standard prototype, defined below:

```

//
// Define the standard type used to describe a C-language exception handler,

```

```

// which is used with _C_specific_handler.
//
// The actual parameter values differ depending on whether the low byte of the
// first argument contains the value 0x1. If this is the case, then the call
// is to the unwind handler to the routine; otherwise, the call is to the
// exception handler for the routine. Each routine has fairly different
// interpretations for the two arguments, though the prototypes are as far as
// calling conventions go compatible.
//
typedef
LONG
(NTAPI * PC_LANGUAGE_EXCEPTION_HANDLER)(
    __in   PEXCEPTION_POINTERS   ExceptionPointers, // if low byte is 0x1, then we're an unwind
    __in   ULONG64               EstablisherFrame    // faulting routine stack pointer
);

```

In the case of `nt!ExpTimeRefreshDpcRoutine`, the fourth scope handler registration is the one that performs the call to PatchGuard's integrity check routine. Here, the routine only executes the integrity check if a state variable stored at `[rsp+20h]` in the DPC routine is set to a particular value. This state variable is modified as the access violation exception traverses each of the exception/unwind scope handlers until it reaches this handler, which eventually leads up to the execution of PatchGuard's system integrity check. For now, it is best to assume that this routine is being called with `[rsp+20h]` in the DPC routine having been set to a value other than `0x15`. This signifies that PatchGuard should be executed.

```

0: kd> uf fffff8000112c8f0
nt!ExpTimeRefreshDpcRoutine+0x17f:
;
; mov eax, eax is a hotpatch stub and can be ignored.
;
fffff800'0112c8f0 8bc0          mov     eax,eax
fffff800'0112c8f2 55          push   rbp
fffff800'0112c8f3 4883ec20    sub    rsp,20h
;
; rdx corresponds to the EstablisherFrame argument.
; This argument is the stack pointer (rsp) value for
; the routine that this exception/unwind handler is
; associated with. The typical use of this argument
; is to allow seamless access to local variables in
; the routine for which the try/except filter is
; associated with. This is what eventually ends up
; occurring here, with the rbp register being loaded
; with the stack pointer of the DPC routine at the
; point in time where the exception occurred.
;
;
fffff800'0112c8f7 488bea     mov    rbp,rdx
;
; We make the check against the state variable.
; Recall that when the DPC routine was first entered,
; [rsp+20h] in the DPC routine's context was set to

```

```

; zero. That location corresponds to [rbp+20h] in
; this context, as rbp has been loaded with the stack
; pointer that was in use in the DPC routine. This
; location is checked and altered by each of the
; registered exception/unwind handlers, and will
; eventually be set to 0x15 when this routine is called.
;
fffff800'0112c8fa 83452007      add     dword ptr [rbp+20h],7
fffff800'0112c8fe 8b4520      mov     eax,dword ptr [rbp+20h]
fffff800'0112c901 83f81c      cmp     eax,1Ch
;
; For the moment, consider the case where this jump is
; not taken. The jump is taken when PatchGuard is not
; being executed (which is not the interesting case).
;
fffff800'0112c904 0f858c000000 jne     nt!ExpTimeRefreshDpcRoutine+0x215 (fffff800'0112c996)

nt!ExpTimeRefreshDpcRoutine+0x189:
;
; To understand the following instructions, it is
; necessary to look back at the stack variable context
; that was set up by the DPC routine prior to the
; faulting instruction that caused the access
; violation exception. The following values were
; set on the stack at that time:
;
; [rsp+0A8h] = ROL(DeferredContext, 0x34)
; [rsp+80h] = ROR(Dpc, 0x48)
;
; The following set of instructions utilize these
; obfuscated copies of the original arguments to the
; DPC routine in order to make the call to PatchGuard's
; integrity check routine.
;
; The first step taken is to deobfuscate the Dpc value
; that was stored at [rsp+80h], or [rbp+80h] as seen from
; this context.
;
fffff800'0112c90a 488b8580000000 mov     rax,qword ptr [rbp+80h]
;
; rax = Dpc
;
fffff800'0112c911 48c1c048      rol     rax,48h
;
; [rbp+50h] -> Dpc
;
fffff800'0112c915 48894550      mov     qword ptr [rbp+50h],rax
;
; Next, the DeferredContext argument is deobfuscated and
; stored plaintext.
;
fffff800'0112c919 488b85a8000000 mov     rax,qword ptr [rbp+0A8h]
;
; rax = DeferredContext
;
fffff800'0112c920 48c1c834      ror     rax,34h
;

```

```

; [rbp+58h] -> DeferredContext
;
fffff800'0112c924 48894558      mov     qword ptr [rbp+58h],rax
;
; rax = Dpc
;
fffff800'0112c928 488b4550      mov     rax,qword ptr [rbp+50h]
;
; The next instruction accesses memory after the KDPC
; object in memory. Recall that a KDPC object is 0x40
; bytes in length on x64, so [Dpc+40h] is the first
; value beyond the DPC in memory. In reality, the KDPC
; is a member of a larger structure, which is defined
; as follows:
;
; struct PATCHGUARD_DPC_CONTEXT {
;   KDPC      Dpc;           // +0x00
;   ULONGLONG DecryptionKey; // +0x40
; };
;
; As a result, this instruction is equivalent to casting
; the Dpc argument to a PATCHGUARD_DPC_CONTEXT*, and then
; accessing the DecryptionKey member
;
;
; rcx = Dpc->DecryptionKey
;
fffff800'0112c92c 488b4840      mov     rcx,qword ptr [rax+40h]
;
; [rbp+40h] -> DecryptionKey
;
fffff800'0112c930 48894d40      mov     qword ptr [rbp+40h],rcx
;
; rax = DecryptionKey
;
fffff800'0112c934 488b4540      mov     rax,qword ptr [rbp+40h]
;
; The DeferredContext value is then xor'd with the
; decryption key stored in the PATCHGUARD_DPC_CONTEXT
; structure. This yields the significant bits of the
; pointer to the PatchGuard decryption stub. Recall
; that due to the "no-mans-land" region in between the
; kernel mode and user mode address space boundaries
; on current AMD64 processors, the rest of the bits
; are required to be either all ones or all zeros in
; order to form a valid address. Because we are
; dealing with a kernel mode address, it can be safely
; assumed that all of the bits must be ones.
;
fffff800'0112c938 48334558      xor     rax,qword ptr [rbp+58h]
;
; [rbp+30h] -> DeferredContext ^ DecryptionKey
;
fffff800'0112c93c 48894530      mov     qword ptr [rbp+30h],rax
;
; Set the required bits to ones in the decrypted
; pointer, as required to form a canonical address on

```

```

; current AMD64 systems.
;
fffff800'0112c940 48b8000000000f8ffff mov rax,0FFFFFF8000000000h
;
; [rbp+30h] -> [rbp+30h] | 0xFFFFF80000000000
;
; Now, [rbp+30h] is the pointer to the decryption stub.
;
fffff800'0112c94a 48094530          or      qword ptr [rbp+30h],rax
;
; The following instructions make extra copies of the decryption
; stub on the stack of the DPC routine. There is no real purpose
; to this, other than a half-hearted attempt to confuse anyone
; attempting to reverse engineer this section of PatchGuard.
;
; [rbp+38h] -> [rbp+30h] (Decryption stub)
;
fffff800'0112c94e 488b4530          mov     rax,qword ptr [rbp+30h]
fffff800'0112c952 48894538          mov     qword ptr [rbp+38h],rax
;
; [rbp+28h] -> [rbp+38h] (Decryption stub)
;
fffff800'0112c956 488b4538          mov     rax,qword ptr [rbp+38h]
fffff800'0112c95a 48894528          mov     qword ptr [rbp+28h],rax
;
; The next set of instructions rewrite the first
; four bytes of the initial opcode in the decryption
; stub. This opcode must be set to the following
; instruction:
;
; f0483111          lock xor qword ptr [rcx],rdx
;
; The individual opcode bytes for the instruction are
; written to the decryption stub one byte at a time.
;
; *(PULONG)DecryptionStub = 0x113148f0
;
fffff800'0112c95e 488b4528          mov     rax,qword ptr [rbp+28h]
fffff800'0112c962 c600f0          mov     byte ptr [rax],0F0h
fffff800'0112c965 488b4528          mov     rax,qword ptr [rbp+28h]
fffff800'0112c969 c6400148          mov     byte ptr [rax+1],48h
fffff800'0112c96d 488b4528          mov     rax,qword ptr [rbp+28h]
fffff800'0112c971 c6400231          mov     byte ptr [rax+2],31h
fffff800'0112c975 488b4528          mov     rax,qword ptr [rbp+28h]
fffff800'0112c979 c6400311          mov     byte ptr [rax+3],11h
;
; Finally, a call to the decryption stub is made. The
; decryption stub has a prototype that conforms to the
; following definition:
;
; VOID
; NTAPI
; PgDecryptionStub(
;     __in PVOID PatchGuardRoutine,
;     __in ULONG64 DecryptionKey,
;     __in ULONG Reserved0,
;     __in ULONG Reserved1

```

```

;      );
;
; The two 'reserved' ULONG values are always set to zero.
;
; rcx is loaded with the address of the decryption stub,
; and rdx is loaded with the DecryptionKey value.
;
fffff800'0112c97d 4533c9      xor     r9d,r9d
fffff800'0112c980 4533c0      xor     r8d,r8d
fffff800'0112c983 488b5540     mov     rdx,qword ptr [rbp+40h]
fffff800'0112c987 488b4d38     mov     rcx,qword ptr [rbp+38h]
;
; At this point, control is transferred to the decryption
; stub, as described previously. The decryption stub will
; decrypt itself, decrypt the PatchGuard integrity check
; routine, and then transfer control to the PatchGuard
; integrity check routine. The integrity check routine is
; responsible for ensuring that the DPC is returned to a
; usable state (recall that parts of it were zeroed out
; by the DPC routine earlier), and that it is re-queued
; for execution. It is also responsible for re-encrypting
; the decryption stub as desired.
;
fffff800'0112c98b ff5538      call   qword ptr [rbp+38h]
;
; After the call is made, the exception filter returns
; the EXCEPTION_EXECUTE_HANDLER manifest constant. This
; causes one of the registered handlers to be invoked
; in order to handle the exception. The handler will
; transfer control to the return point of the DPC routine,
; thus skipping the body of the DPC (since the call to
; the DPC was not a request for the legitimate function of
; the DPC to be performed).
;
fffff800'0112c98e 41b901000000     mov     r9d,1
fffff800'0112c994 eb03             jmp     nt!ExpTimeRefreshDpcRoutine+0x218 (fffff800'0112c999)

nt!ExpTimeRefreshDpcRoutine+0x215:
fffff800'0112c996 4533c9      xor     r9d,r9d

nt!ExpTimeRefreshDpcRoutine+0x218:
fffff800'0112c999 418bc1      mov     eax,r9d
fffff800'0112c99c 4883c420     add     rsp,20h
fffff800'0112c9a0 5d          pop     rbp
fffff800'0112c9a1 c3          ret

```

This does represent a significant level of obfuscation, but it is not impenetrable, and there are various simple ways through which an attacker could bypass all of these layers of obfuscation entirely.

3.5 Disruption of Debug Register-Based Breakpoints

PatchGuard version 2 attempts to protect itself from breakpoints that are set using the hardware debug registers. These breakpoints operate by setting up to four designated memory locations that are of interest. Each memory location can be configured to cause a debug exception when it is read, written, or executed. Because breakpoints of this flavor are not visible to PatchGuard's code integrity checks (unlike conventional breakpoints, these breakpoints do not involve `int 3 (0xcc)` opcodes being substituted for target instructions), debug register-based breakpoints (sometimes known as "memory breakpoints" or "hardware breakpoints") pose a threat to PatchGuard. PatchGuard attempts to counter this threat by disabling all such debug register-based breakpoints as a first step after the system integrity checking routine has been decrypted in-memory:

```
;
; Here, the second stage decryption sequence is
; set to run to decrypt the system integrity
; check routine. We step over the second stage
; decryption and examine the integrity check
; routine in its plaintext state...
;
fffffadf'f6edc043 8b4a4c      mov     ecx,dword ptr [rdx+4Ch]
fffffadf'f6edc046 483144ca48    xor     qword ptr [rdx+rcx*8+48h],rax
fffffadf'f6edc04b 48d3c8        ror     rax,cl
fffffadf'f6edc04e e2f6         loop   fffffadf'f6edc046
fffffadf'f6edc050 8b8288010000  mov     eax,dword ptr [rdx+188h]
fffffadf'f6edc056 4803c2        add     rax,rdx
fffffadf'f6edc059 ffe0         jmp     rax
fffffadf'f6edc05b 90           nop
;
; We set a breakpoint on the 'jmp rax' instruction
; above. This instruction is what transfers control
; to the system integrity check routine.
;
0: kd> ba e1 fffffadf'f6edc059
0: kd> g
Breakpoint 2 hit
fffffadf'f6edc059 ffe0          jmp     rax
;
; rax now points to the decrypted system
; integrity check routine in-memory. The
; first call it makes is to a routine whose
; purpose is to disable all debug register-based
; breakpoints by clearing the debug control
; register (dr7). Doing so effectively turns
; off all of the debug register breakpoints.
;
0: kd> u @rax
fffffadf'f6edd8de 4883ec78     sub     rsp,78h
```

```

fffff6e138e2 48895c2470      mov     qword ptr [rsp+70h],rbx
fffff6e138e7 48896c2468      mov     qword ptr [rsp+68h],rbp
fffff6e138ec 4889742460      mov     qword ptr [rsp+60h],rsi
fffff6e138f1 48897c2458      mov     qword ptr [rsp+58h],rdi
fffff6e138f6 4c89642450      mov     qword ptr [rsp+50h],r12
fffff6e138fb 488bda         mov     rbx,rdx
fffff6e138fe 4c896c2448      mov     qword ptr [rsp+48h],r13
0: kd> u
fffff6e13903 e8863a0000      call   fffff6e138e
;
; The routine simply writes all zeros to dr7.
;
0: kd> u fffff6e138e
fffff6e138e 33c0           xor     eax,eax
fffff6e1390 0f23f8         mov     dr7,rax
fffff6e1393 c3             ret

```

3.6 Misleading Symbol Names

One of the things that Microsoft needed to consider when implementing PatchGuard is that would-be attackers would have access to the operating system symbols. As a debugging aid, Microsoft makes symbols for the entire operating system publicly available. It is not feasible to remove the operating system symbols from public access (doing so would severely hinder ISVs in the process of debugging their own drivers). As a result, Microsoft took the route of using misleading function names to shroud PatchGuard routines from casual inspection. Many of the internal PatchGuard routines have names that are seemingly legitimate-sounding at first glance, such that without a detailed knowledge of the kernel or actually inspecting these routines, it would be difficult to simply look at a list of all symbols in the kernel and locate the routines responsible for setting up PatchGuard.

The following is a listing of some of the misleading symbols that are used during PatchGuard initialization:

1. RtlpDeleteFunctionTable
2. FsRtlMdlReadCompleteDevEx
3. RtlLookupFunctionEntryEx
4. SdbpCheckDll
5. FsRtlUninitializeSmallMcb
6. KiNoDebugRoutine
7. SepAdtInitializePrivilegeAuditing
8. KiFilterFiberContext

3.7 Integrity Checks Performed During System Initialization

During system initialization, PatchGuard performs integrity checks on several of the anti-debug mechanisms it has in place. If these mechanisms are altered on-disk, PatchGuard will detect the changes. For example, PatchGuard validates that the routine responsible for clearing debug register-based breakpoints contains the correct opcode bytes corresponding to the instructions used to actually zero out Dr7:

```
;
; Here, we are in SepAdtInitializePrivilegeAuditing, or the
; initialization routine for PatchGuard during system startup.
;
; This code fragment is designed to validate that the
; KiNoDebugRoutine routine contains the expected opcodes that
; are used to zero out debug register breakpoints. If the
; routine does not contain the correct opcodes, PatchGuard
; makes an early exit from SepAdtInitializePrivilegeAuditing.
;
INIT:0000000000832A6D lea    rax, KiNoDebugRoutine
INIT:0000000000832A74 cmp    dword ptr [rax], 230FC033h
INIT:0000000000832A7A jnz    abort_initialization
INIT:0000000000832A80 add    rax, 4
INIT:0000000000832A84 cmp    word ptr [rax], 0C3F8h
INIT:0000000000832A89 jnz    abort_initialization
```

3.8 Overwriting PatchGuard Initialization Code Post-Boot

After PatchGuard has initialized itself, it intentionally zeros out much of the code responsible for setting up PatchGuard. It is assumed that this is done in an attempt to prevent third party drivers from analyzing kernel code in-memory in order to detect or defeat PatchGuard. This approach is obviously trivially bypassed by opening the kernel image on disk, however.

After boot, many PatchGuard-related routines contain all zeros:

```
0: kd> u nt!KiNoDebugRoutine
nt!KiNoDebugRoutine:
fffff800'011a4b20 0000          add     byte ptr [rax],al

nt!FsRtlUninitializeSmallMcb:
fffff800'011a4aa2 0000          add     byte ptr [rax],al

0: kd> u nt!KiGetGdtIdt
nt!KiGetGdtIdt:
```

```
fffff800'011a4a20 0000      add    byte ptr [rax],al

0: kd> u nt!RtlpDeleteFunctionTable
nt!RtlpDeleteFunctionTable:
fffff800'011a1010 0000      add    byte ptr [rax],al
```

Most of the PatchGuard initialization code resides in the INITKDBG section of ntoskrnl. Portions of this section are zeroed out during initialization.

Chapter 4

Bypass Techniques

Despite the myriad anti-reverse-engineering and anti-debug techniques employed by PatchGuard version 2, it is hardly invincible to being bypassed by third party code. Contrary to one might expect, given the descriptions in the initial section of this article, there are a number of holes in PatchGuard's armor that can be exploited by third party software. Several potential techniques for bypassing PatchGuard version 2 are outlined below, including one technique that includes functional proof of concept code. These techniques are applicable to the version of PatchGuard currently shipping with Windows XP x64 Edition with all hotfixes, Windows Server 2003 x64 Edition with all hotfixes, and Windows Vista x64 with all hotfixes at the time that this article was written. The author has only written a complete implementation of the first proposed bypass technique, although the remaining proposed bypass approaches are expected to be viable in principle.

4.1 Interception of `_C_specific_handler`

The simplest course of action for disabling PatchGuard version 2 is, in the author's opinion, to intercept execution at `_C_specific_handler`. The `_C_specific_handler` routine is responsible for dispatching exceptions for routines compiled with the Microsoft C/C++ compiler (and using `try/except`, `try/finally`, or `try/catch` clauses). This set of functions includes all ten of the PatchGuard DPC routines and most other C/C++ functions in the kernel. It also includes many third party driver routines as well; `_C_specific_handler` is exported, and the compiler references this function for all C/C++ images that utilize SEH in some form (imported from `ntoskrnl`). Due to this, Microsoft is forced to export `_C_specific_handler` from the kernel perpetually, making it difficult for Microsoft to deny access to the routine's address from the perspective of third

party drivers. Furthermore, because `_C_specific_handler` is exported from the kernel, it is trivial to retrieve its address across all kernel versions from the context of a third party driver. This approach capitalizes on the fact that PatchGuard utilizes SEH in order to obfuscate the call to the system integrity checking routine, in effect turning this obfuscation mechanism into a convenient way to hijack execution control before the system integrity check is actually performed.

This approach can be implemented in several different ways, but the basic idea is to intercept execution somewhere between the faulting instruction in the PatchGuard DPC (whichever is selected at boot time), and the exception handlers associated with the DPC routine which invoke the PatchGuard system integrity check routine. With this in mind, `_C_specific_handler` is exactly what one could hope for; `_C_specific_handler` is invoked when the benign access violation triggered by the bogus `DeferredContext` value to the PatchGuard DPC routine is called. Furthermore, being exported, there are no concerns with compatibility with future kernel versions, or different flavors of the kernel (PAE vs non-PAE, MP vs UP, and so forth).

Although hooking `_C_specific_handler` provides a convenient way to gain control of execution in the execution path for the PatchGuard check routine, there remains the problem of how to safely defuse the check routine and resume execution at a safe point such that DPCs continue to be processed by the system in a timely fashion. On x86, this would pose a serious problem, as in this context, we (as an attacker attempting to bypass PatchGuard) would gain control at an exception handler with a context record describing the context at middle of the PatchGuard DPC routine, with no good way to unwind the context back up to the DPC routine's caller (the kernel timer DPC dispatcher).

Ironically, by virtue of being only on x64 and not x86, this problem is made trivial where it might have been difficult to solve in a generalized fashion on x86. Specifically, there is extensive unwind support baked into the core of the x64 calling convention on Windows, such that there exists metadata describing how to unwind any function that manipulates the stack at any point in its execution lifetime. This metadata is used to implement unwind semantics that allow functions to be cleanly unwound without having to call exception/unwind handlers implemented in code that depend on the execution context of the routine they are associated with. This extensive unwind metadata can be used to our advantage here, as it provides a clean mechanism to unwind past the DPC routine (to the DPC dispatcher) in a completely compatible and kernel-version-independent manner. Furthermore, there is no good way for Microsoft to disable this unwind metadata, given how deeply involved it is with the x64 calling convention.

The process of using the unwind metadata of a function to unwind an execution context is known as a virtual unwind, and there is a documented, exported routine [5] to implement this mechanism: `RtlVirtualUnwind`. Using `RtlVirtualUnwind`

alUnwind, it is possible to alter the execution context that is provided as an argument to `_C_specific_handler` (and thus the hook on `_C_specific_handler`). This execution context describes the machine state at the time of the access violation in the PatchGuard DPC routine. After performing a virtual unwind on this execution context, all that remains is to return the manifest `ExceptionContinueExecution` constant to the kernel mode exception dispatcher in order to realize the altered context. This completely bypasses the PatchGuard system integrity check. As an added bonus, the hook on `_C_specific_handler` is only needed until the first time PatchGuard is called. This is due to the fact that the PatchGuard timer is a one-shot timer, and as the code to re-queue the timer is skipped by the virtual unwind, PatchGuard is effectively permanently disabled for the remainder of the Windows boot session.

The last remaining obstacle with this bypass technique is filtering out the specific PatchGuard access violation exceptions from legitimate access violations that kernel mode code may produce. This is important, as access violations in kernel mode are a normal part of parameter validation (the probe and lock model used to validate user mode pointers) for drivers and system services. Fortunately, it is easy to make this determination, as it is generally only legal to use a try/except to catch an access violation relating to a user mode address from kernel mode (as previously described). PatchGuard is a rare exception to this rule, in that it has a well-defined no-mans-land region where accesses can be attempted without fear of a bugcheck occurring. As a result, it is a safe assumption that any access violation relating to a kernel mode address is either PatchGuard trigger its own execution, or a very badly behaved third party driver that is grossly breaking the rules relating to Windows kernel mode drivers. It is the author's opinion that the latter case is not worth considering as a blocker, especially since if such a completely broken driver were to exist, it would already be randomly bringing the system down with bugchecks. It is worth noting, as an addendum, that the referenced address in the exception information block passed to the exception handler will always be `0xFFFFFFFFFFFFFFFF` due to how violations on non-canonical addresses are reported by the processor. This does not impact the viability of this technique as a valid way to bypass PatchGuard in a version-independant manner, however.

It is worth noting that the fact that this technique involves modifying the kernel is not a problem (aside from the inherent race conditions involved in safely patching a running binary). The hook will disable PatchGuard before PatchGuard has a chance to notice the hook from the context of the system integrity check routine.

This proposed approach has several advantages over the previously suggested approach by Uninformed's original paper on PatchGuard [2]. Specifically, it does not involve locating each individual DPC routine (and does not even rely on any sort of code fingerprinting; only exported symbols are used). This improves both the reliability of the proposed approach (as code fingerprinting always introduces an additional margin of error as far as false positives go) and its resiliency to

attack by Microsoft. Because this technique relies solely on exported functions, and does not carry any sort of dependency on how many possible DPCs are available to PatchGuard for use (or any sort of dependency on locating them at runtime), blocking this approach would be significantly more involved than simply adding another possible DPC routine or changing the attributes of an existing DPC routine in an effort to third-party drivers that were taking a signature-based approach to locating DPC routines for patching.

Although this technique is quite resilient to kernel changes that do not directly involve the underlying mechanisms by which PatchGuard itself functions (the fact that it can operate unmodified on both Windows Server 2003 x64 and Windows Vista x64 is testament to this fact), there are a number of different ways by which Microsoft could block this attack in a future update to PatchGuard. The most obvious solution is to entirely abandon SEH as a core mechanism involved in arranging for the PatchGuard system integrity check. Abandoning SEH removes the convenient mechanism (hooking `_C_specific_handler`) that is presented here as a version-independent way to hook in to the execution path involved in PatchGuard's system integrity check. If Microsoft were to go this route, a would-be attacker would need to devise another mechanism to achieve control of execution before the system integrity check runs. Assuming that Microsoft played their hand correctly, a future PatchGuard revision would not have such an easily-accessible mechanism to hook into the execution process in a generic manner, largely counteracting this proposed approach. Microsoft could also employ some sort of pre-validation of the exception handler path before the DPC triggers an exception, although given that this is not the easiest and most elegant way to counter such a technique, the author feels that it is an unlikely solution.

4.2 Interception of DPC Exception Registration

Presently, all execution paths leading to the execution of PatchGuard DPC routines involve an exception/unwind handler. This is another single point of failure weakness that can be exploited by third parties attempting to disable PatchGuard. An approach involving the detection of all of the PatchGuard DPC routines, followed by interception of the exception handler registrations for each DPC is proposed as another means of defeating PatchGuard.

Though this technique is not as clean or clear-cut as the technique proposed in 4.1, this approach is considered by the author as a viable bypass mechanism for PatchGuard version 2. This technique essentially involves patching the exception registrations for each possible DPC routine that could be used by PatchGuard, such that each exception registration points to a routine that employs a virtual unwind to safely exit out of the PatchGuard DPC without invoking the system integrity check. Any such approach faces several obstacles,

however.

The first major difficulty for this technique is locating each PatchGuard DPC. Since none of the PatchGuard DPC routines are exported, a little bit more creative thinking is involved in finding the locations to patch. The author feels that a combination of pattern matching and code fingerprinting would best serve this goal; there are a number of commonalities between the different PatchGuard DPC routines that could be used to locate them with a relatively high degree of confidence in PatchGuard version 2. Specifically, the author feels that the following criteria are acceptable for use in detecting the PatchGuard DPC routines:

1. Each DPC routine has one exception/unwind-marked registration with `_C_specific_handler`.
2. Each DPC routine has exactly four `_C_specific_handler` scopes.
3. Each DPC routine is referenced in raw address form (64-bit pointer) in the executable code sections comprising `ntoskrnl` at least twice.
4. Each DPC routine has at least two `_C_specific_handler` scopes with an associated unwind/exception handler.
5. Each DPC routine has exactly one `_C_specific_handler` scope with a call to a common subfunction that references `RtlUnwindEx` (an exported routine).
6. Each DPC routine has several sets of distinctive, normally rare instructions (`ror/rol` instructions).

Given several (or even all) of these criteria, it should be possible to accurately locate all ten DPC routines via scanning non-pagable code in the kernel. It is possible to locate the exception registration information for the DPC routines through processing of the exception directory for the kernel (and indeed, most of the criteria require doing this as a prerequisite). Locating the kernel image base is fairly trivial as well; the address of an exported routine can be taken, and truncated to a 64K region. From there, one need only perform downward searches in 64K increments for the DOS header signature (followed by a check for a PE32+ header).

Another hurdle that must be solved for this approach is the placement of the replacement exception handler routines. These routines are required to be within 4GB of the kernel image base (there is only a 32-bit RVA in the unwind metadata), meaning that in general, it is not practical to simply store them in a driver binary or pool allocation (by default, these addresses are usually far more than 4GB away from the kernel image base). There are no documented and exported routines to allocate kernel mode virtual memory at a specific virtual address to the author's knowledge. However, other, less savory approaches

could theoretically be taken (such as allocating physical memory and altering paging structures directly to create a valid memory region within 4GB of the kernel image base).

After one has solved these difficulties, the rest of this approach is fairly trivial (and similar to portions of the technique described in 4.1.). Specifically, the replaced exception handlers need to invoke `RtlVirtualUnwind` to unwind back to the kernel DPC dispatcher, and then request that execution be resumed at the unwound context.

This mechanism is not nearly as robust as the first in the author's point of view, though both approaches could be disabled by abandoning SEH entirely as a critical path in the execution of the PatchGuard system integrity check routine. Specifically, Microsoft could change the characteristics of the DPC routines in an attempt to frustrate fingerprinting and detection of them at runtime. Pre-validation of unwind metadata (or additional checks in the exception dispatcher itself to ensure that all SEH routines registered as part of an image are within the confines of the image in-memory) could also be used to defeat this technique. There are other security benefits to validating that SEH routines on x64 that are registered as part of an image really exist within an image, as will be discussed below. As such, the author would expect this to appear in a future Windows version.

4.3 Interception of `PsInvertedFunctionTable`

Another variation on the theme of intercepting PatchGuard within the SEH code path critical to the system integrity check routine involves taking advantage of an optimization that exists in the x64 exception dispatcher. Specifically, it is possible to utilize the fact that the exception dispatcher on x64 uses a cache to improve the performance of exception handling. By taking advantage of this cache, it may be possible to intercept control of execution when the PatchGuard DPC routine deliberately creates an access violation exception in order to trigger the system integrity check. This proposed technique uses the `nt!PsInvertedFunctionTable` global variable in the kernel, which represents a cache used to perform a fast translation of RIP values to an associated image base and exception directory pointer, without having to do a (slow) search through the linked list of loaded kernel modules.

This technique is fairly similar to the one described in technique 4.2. Instead of altering the actual exception directory entries corresponding to each PatchGuard DPC routine in the kernel's image in-memory, this technique alters the cached exception directory pointer stored within `PsInvertedFunctionTable`. `PsInvertedFunctionTable` is consulted by `RtlLookupFunctionTableEntry`, in order to translate a RIP value to an associated image (and unwind metadata block). The logic within `RtlLookupFunctionTable` is essentially to search through

the cached entries resident in PsInvertedFunctionTable for an image that corresponds to a given RIP value. If a hit is found, then the exception directory pointer is loaded directly from the PsInvertedFunctionTable cache, instead of through the (slower) process of parsing the PE header of the given image. If no hit is found, then the loaded module linked list is searched. Assuming a hit is made in the loaded module list, then the PE header for the associated module is processed in order to locate the exception directory for the module. From there, the exception directory is searched to locate the unwind metadata block corresponding to the function containing the specified RIP value.

The structure backing PsInvertedFunctionTable (RTL_INVERTED_FUNCTION_TABLE) can be described as so in C:

```
typedef struct _RTL_INVERTED_FUNCTION_TABLE_ENTRY
{
    PIMAGE_RUNTIME_FUNCTION_ENTRY ExceptionDirectory;
    PVOID ImageBase;
    ULONG ImageSize;
    ULONG ExceptionDirectorySize;
} RTL_INVERTED_FUNCTION_TABLE_ENTRY, * PRTL_INVERTED_FUNCTION_TABLE_ENTRY;

typedef struct _RTL_INVERTED_FUNCTION_TABLE
{
    ULONG Count;
    ULONG MaxCount; // always 160 in Windows Server 2003
    ULONG Pad[ 0x2 ];
    RTL_INVERTED_FUNCTION_TABLE_ENTRY Entries[ ANYSIZE_ARRAY ];
} RTL_INVERTED_FUNCTION_TABLE, * PRTL_INVERTED_FUNCTION_TABLE;
```

In Windows Server 2003, there is space reserved for up to 160 loaded modules in the array contained within PsInvertedFunctionTable. In Windows Vista, this number has been expanded to 512 module entries. The array of loaded modules is maintained by the system module loader such that when a module is loaded or unloaded, a corresponding entry within PsInvertedFunctionTable is created or deleted, respectively. It is not a fatal error for the module array within PsInvertedFunctionTable to be exhausted; in this case, performance for exception dispatching relating to additional modules will be slower, but the system will still function.

Because the RIP-to-exception-directory cache described by PsInvertedFunctionTable maintains a full 64-bit pointer to the exception directory of the associated module, it is possible to disassociate the cached exception directory pointer from its corresponding image. In other words, it is possible to modify the ExceptionDirectory member of a particular cached RTL_INVERTED_FUNCTION_TABLE_ENTRY to point to an arbitrary location instead of the exception directory of that module. There are no security or integrity checks that validate that the ExceptionDirectory member points to within the given image. This could be exploited by a third-party driver in order to take control of exception dispatching for any of the first 160 (or 512, in the case of Windows Vista) kernel modules. This loaded

module list includes critical images such as the HAL (typically the first entry in the cache) and the kernel itself (typically the second entry in the cache). With respect to bypassing PatchGuard, this makes it possible for a third party driver to copy the exception directory data of the kernel to dynamically allocated memory and adjust it such that exception handlers for the PatchGuard DPC routines point to a stub function that invokes a virtual unwind as described in technique 4.2. After setting up its altered shadow copy of the exception directory for the kernel, all that a third party driver would need to do is swap the ExceptionDirectory pointer within the PsInvertedFunctionTable cache entry for the kernel with the pointer to the shadow copy. Following that, this approach is essentially the same as the proposed approach described in 4.2. It has the added advantage of being more difficult to detect from the perspective of validating the integrity of the exception dispatching path, as the exception directory associated with the kernel image in-memory is not actually altered; only a pointer to the exception directory in a cache is changed.

This approach does require a reliable mechanism to detect PsInvertedFunctionTable (which is not exported) at run-time, however. The author feels that this is not a particularly difficult task, as the first few members of PsInvertedFunctionTable (specifically, the maximum entry count and the entries for the HAL and kernel) will have predictable values that can be used in a classic egghunt style search of kernel global variable space. Additional heuristics, such as requiring several data references to the suspected PsInvertedFunctionTable location within kernel code could be applied as well, in the interest of improving accuracy.

This proposed approach may be countered by many of the proposed counters to techniques 4.1 and 4.2. Additionally, this technique could also be countered by validating exception directory pointers within PsInvertedFunctionTable, such as by ensuring that such exception directory pointers are within the confines of the purported associated image. Although this validation is not perfect since it might still be possible for one to reposition the exception directory pointer to a different location within the image that could be safely modified at runtime, such as overlapping a large global variable array or the like, it would certainly increase the difficulty of subverting the exception dispatcher's RIP translation cache. Additional validation techniques, such as requiring that the exception directory point to read-only memory, could be similarly adopted to reduce the chance that a third party driver could meaningfully subvert the cache (with results leading to something other than a system crash).

It should be noted that in the current implementation, PsInvertedFunctionTable presents a relatively inviting target for potentially malicious software to hijack parts of the kernel without being detected. Indeed, through careful planned subversion of PsInvertedFunctionTable, third party software could take control of exception dispatchers throughout the kernel in order to gain control of execution. Though this technique would be much more limited than outright kernel patching, it has the advantage of being completely undetected by current Patch-

Guard versions (which cannot validate global variables that may change without notice at runtime, for obvious reasons). It also has the advantage of being undetected by current rootkit detection systems, which are presently (to the author's knowledge) blissfully unaware of `PsInvertedFunctionTable`. Although it would require administrative permissions (or an exploit granting such permissions) for an attacker to modify `PsInvertedFunctionTable` in the first place, Microsoft has at late focused a great deal of effort on protecting the kernel even from users with administrator permissions. For example, one could conceive of a rootkit-style program that intercepts exception dispatchers for system services, and passes invalid user mode pointers to system services in order to surreptitiously execute kernel mode code without detection when the standard pointer probe throws an exception indicating that the given usermode pointer parameter is invalid. Given this sort of threat (from the rootkit perspective), the author feels that it would be in Microsoft's best interests to put into place additional validation of `PsInvertedFunctionTable`'s cached exception directory pointers (assuming that Microsoft wishes to continue down the path of strengthening the kernel against malicious administratively-privileged code).

4.4 Interception of `KiDebugTrapOrFault`

Although many of the proposed techniques for blocking PatchGuard have so far relied on the fact that PatchGuard utilizes SEH to kick off execution of the system integrity check, there are different approaches that can be taken which do not rely on this specific PatchGuard implementation detail. One such alternative technique for bypassing PatchGuard involves subverting the kernel debug fault handler: `KiDebugTrapOrFault`. This handler represents the entry point for all debug exceptions (such as so-called hardware breakpoints), and as such presents an attractive target for bypassing PatchGuard.

The basis of this proposed technique is to utilize a set of hardware breakpoints to intercept execution at a convenient critical location within PatchGuard's execution path leading up to the system integrity check. This technique has a greater degree of flexibility than many of the previously described techniques, though this flexibility comes at cost of a significantly more involved (and difficult) implementation. Specifically, one could use this proposed technique to intercept control at any point critical to the execution of PatchGuard's system integrity check (for example, the kernel DPC dispatcher, one of the PatchGuard DPC routines, or a convenient location in the exception dispatching code path, such as `_C_specific_handler`).

The means by which this interception of execution could be accomplished is by assuming control of debug exception handling. This could be done in several different ways; for example, one could hook `KiDebugTrapOrFault` or alter the IDT directory to simply repoint the debug exception to driver-supplied code, bypass-

ing `KiDebugTrapOrFault` entirely. There are even ways that this interception could be done in a way that is transparent to the current PatchGuard implementation, such as by intercepting `PsInvertedFunctionTable` as described in technique 4.3. A driver could then alter the unwind metadata for `KiDebugTrapOrFault` and create an exception handler for this routine. This step would allow transparent, first-chance access to all debug faults (because `KiDebugTrapOrFault` internally constructs and dispatches a `STATUS_SINGLE_STEP` exception describing the debug fault; normally, this would present the `STATUS_SINGLE_STEP` exception to a debugger, but there is no technical reason why a standard SEH-style exception handler could not catch the exception). Regardless of how control of execution at the debug trap handler is gained, the next step in this proposed approach is to alter execution at the requested point of interest (whether it be the kernel timer DPC dispatcher, which could be easily found by queuing a DPC and executing a virtual unwind, or a PatchGuard DPC routine, or `_C_specific_handler` or any other place of interest in the critical PatchGuard execution path) to prevent PatchGuard's system integrity check from executing.

After the implementor has established control over the debug trap handler (through whichever means desired), all that remains is to set debug-register-based breakpoints on target locations. When these breakpoints are hit, control is transferred to the debug trap handler, and from there to the implementor's driver code which can act as necessary, such as by altering the execution context of the processor at the time of the exception before resuming execution.

The advantages of this approach over directly patching into kernel code (i.e. opcode replacement) are threefold. First, it is more flexible in that there are no difficulties with placing an absolute 64-bit jump in an arbitrary location (in x64, this typically takes around 12 opcode bytes to do from any arbitrary location in memory). For example, one does not have to worry about whether a the opcode space overwritten by the jump might overlap a whole instruction boundary that is a jump target, which might lead to invalid code being executed. Secondly, this approach can be used to get out of having to implement a disassembler (or other similar forms of code analysis) in kernel mode, as hardware breakpoints allow one to gain control of execution at a precise location without having to worry about creating enough space for a jump patch, and then placing the original instructions back into a jump stub to allow execution to resume at the original effective instruction stream (if desired). Finally, if done correctly, this technique could be implemented in a truly race-condition free manner (as the only patching that would need to be done is an interlocked 8-byte swap of a pointer-aligned value in `PsInvertedFunctionTable`, if one took that approach).

This approach does require that the implementor pick a location (or multiple locations) in the kernel that are to have breakpoints set over in order to gain execution control. There are many possibilities, such as the DPC dispatcher (where one could filter out the PatchGuard DPC by detecting, say, invalid kernel pointers in `DeferredContext`), the execution dispatcher path (where one could unwind past a PatchGuard DPC's access violation exception), a Patch-

Guard DPC itself (where one could again unwind past with `RtlVirtualUnwind`, bypassing PatchGuard if the DPC is being invoked by PatchGuard), or any other choice area. One of the advantages of this approach is that it is comparatively easy to intercept execution anywhere in the kernel that can be reliably located across kernel versions, making it potentially a great deal more flexible to being easily adapted to defeat future PatchGuard implementations than some of the previously discussed bypass techniques.

Normally, the kernel has logic in place that prevents stray kernel addresses from being placed in debug registers by user mode code via `NtSetContextThread`. It may be necessary to make additional alterations to ensure that the custom values in the debug registers are persisted across context switches, via the same mechanisms used by the kernel debugger to persist debug registers.

In the author's opinion, this technique would be difficult for Microsoft to defeat in principle, barring hardware support (like virtualization). Although Microsoft could move around critical code paths for PatchGuard, this technique presents a general mechanism by which any location in the kernel could be surreptitiously intercepted, thus lending itself to relatively easy adaptation to future PatchGuard revisions. One approach that could be taken is to perform increased validation of the debug trap handler in an attempt to make it more difficult to intercept without being detected by PatchGuard or some other validation mechanism. Other counters to this sort of tactic (in general) would be to make it difficult to reliably locate all of the critical code paths in a consistent and reliable manner across all kernel versions, from the perspective of a third party driver. This is likely to prove difficult, as a great deal of the internal workings of the kernel are exposed in some way to drivers (i.e. exported functions), or are otherwise indirectly exposed to drivers (i.e. trap labels via the IDT, exception handlers via unwind metadata and exports used in the process of dispatching exceptions to SEH registrations). Completely insulating PatchGuard from all such externally visible locations (that could be comparatively easily compromised by a third party driver) would, as a result, likely be an arduous task.

The debug trap handler can be used to do more than simply evade PatchGuard for purposes of allowing conventional kernel code patches via opcode replacement. It can also be utilized in order to completely eliminate the need to perform opcode-replacement-based kernel patches in order to gain execution control. In this vein, via assuming control of the debug trap handler in a way that is transparent to PatchGuard (such as via the proposed `PsInvertedFunctionTable`-based approach), it would then be possible to set debug-register-based breakpoints at every address of interest (assuming that there enough debug registers to patch all of the locations of interest). From the debug trap handler, it is possible to completely alter the execution context at the point of the debug exception, which is exactly the same as what one could do via traditional opcode-replacement-based patching for a given location. This sort of transparent patching would be extremely difficult for Microsoft to detect, because the debug registers must remain available for use by the kernel debugger. Without completely crippling

the ability of the kernel debugger to set breakpoints without being attached before PatchGuard is initialized, the author does not see a particularly viable (i.e. without a trivial workaround) way for Microsoft to prevent the use of debug registers to alter execution context at select points in the kernel (from a third party driver). Because such an approach would capitalize on the fact that Microsoft must, from a business case perspective, make it possible for IHVs and ISVs to debug their code on Windows, the author believes that it would be unlikely to be successfully disabled by Microsoft. Furthermore, because such techniques can be implemented without even having the basic requirement of disabling PatchGuard, they would be inherently much more likely to work with future PatchGuard revisions. After all, if PatchGuard can't even detect changes to the kernel (because kernel code isn't being patched), then there is no reason to even bother with trying to disable it, which gets one out of the comparatively messy business of playing catch-up with Microsoft with each new PatchGuard revision.

4.5 General Detect Bit Interception

One of PatchGuard's anti-debug mechanisms relates to debug registers. Specifically, PatchGuard attempts to clear `Dr7` (the debug control register) in an attempt to disable all debug-register-based breakpoints, as one of the first tasks upon entering the system integrity check routine. This presents an inherent weakness within PatchGuard, as there is support built-in to the processor that allows one to detect (and intercept) direct accesses to any of the debug registers. This support is primarily legacy, intended for so-called *in-circuit emulators* (ICEs), which were special hardware components that acted as a true hardware-based debugger by allowing one to control a processor from outside the context of the system entirely, in essence truly isolating the debugger from the operating system and any programs running under it. This support is embodied in the General Detect bit in `Dr7`, which when set, causes a debug trap to be generated on any successful access to a debug register. This is significant in that it provides a way for an attacker to trap PatchGuard's access to `Dr7` (zeroing it), which in effect provides a means to pinpoint the exact location of PatchGuard's system integrity routine in-memory, in-plaintext. Furthermore, it gives an attacker the possibility of making any alterations desired to the execution context at the very start of the system integrity check, which could be trivially used in order to simply implement an immediate return out of the system integrity check logic without actually verifying the system's integrity (as `dr7` is zeroed before any integrity checks are performed). This approach effectively turns another one of PatchGuard's protection mechanisms against it, utilizing the anti-debug-register behavior to detect (and block) PatchGuard.

The general idea behind this approach is similar to that described in technique 4.4. In the same fashion as in technique 4.4, an implementor of this approach

is required to gain control of the debug trap handler. For this task, any of the proposed approaches in technique 4.4 may be used. After control of the debug trap handler is established, an attacker must then set the general detect bit in `Dr7` and wait for PatchGuard to access the debug registers. It should be noted that during the legitimate course of execution, the kernel itself will often directly access debug registers, such as during context switches or if `NtSetContextThread/NtGetContextThread` are invoked. Any such implementation of this technique must be able to differentiate between PatchGuard's accesses of the debug registers and legitimate accesses. This could be trivially implemented by checking if the RIP value at the time of the trap was within a valid kernel image or not, as the PatchGuard system integrity check routine resides in dynamically allocated non-paged pool and not within the confines of the kernel images in-memory.

When the debug trap handler is invoked as a result of PatchGuard zeroing `Dr7`, then the appropriate action (which could be as trivial as simply executing a hard return out of the system integrity check routine) can be taken by the third-party driver wishing to disable PatchGuard.

Like the techniques that capitalize on PatchGuard's use of SEH to obfuscate the call to the system integrity check routine, this approach relies on using one of PatchGuard's defensive mechanisms against it. The most obvious counter would be to thus remove the behavior of zeroing debug registers. However, disabling this behavior may not be very desirable, as it would then be very easy to detect PatchGuard by, say, setting a read breakpoint on kernel code and waiting for PatchGuard to perform a read. Since reads of kernel code (as opposed to execute fetches) are fairly atypical, this would open up another easy mechanism by which PatchGuard could be bypassed.

The best course of action by Microsoft here would be to make it as difficult as possible to differentiate between legitimate accesses to debug registers and PatchGuard's own accesses, although this is likely to not be very doable. Strengthening the debug trap path against interception by placing additional validation checks over that code path might also be useful in countering this technique, although likely to only a limited, easily-bypassable extent.

4.6 Patching the Kernel Timer DPC Dispatcher

Currently, PatchGuard utilizes a timer with an associated DPC to transfer control to a preselected one of ten possible legitimate DPC routines that have been slightly modified for use with PatchGuard. Because third party kernel drivers are given a documented and exported interface to create timers with associated DPC routines, this represents a weakness in PatchGuard, in that it presents an easily-detectable location in the critical execution path for PatchGuard's system integrity check routine that could be relatively easily compromised by a

third-party driver. This technique focuses on gaining control of the timer DPC dispatcher, with the goal of detecting when the PatchGuard DPC is about to be dispatched. When the PatchGuard DPC is detected, then the third-party driver could skip over the PatchGuard DPC routine entirely, thus disabling PatchGuard.

In order to accomplish this, a third party driver would need to locate the exact instruction within the kernel timer DPC dispatcher that is responsible for making calls to timer DPC routines. Fortunately, this is a fairly easy task for a driver, as the interfaces for creating timers with associated DPCs and DPC routines are documented and exported. Specifically, a third party driver could queue a timer DPC, and then record address of the DPC dispatcher routine via inspection of the return address of the timer DPC routine when it is called. From there, the driver can derive the address of the call instruction responsible for making the call to the DPC routine associated with a DPC object that is associated with a timer.

At this point, all a third party driver needs to do is patch the call instruction in the DPC dispatcher to transfer execution control to the driver's code. From there, the driver can filter all timer DPCs for the PatchGuard DPC routine (perhaps by looking for a bogus kernel address in `DeferredContext`, paired with a DPC routine that is within the confines of the kernel image in-memory). When the PatchGuard DPC is detected, then the driver can decline to call the DPC routine and instead simply return control to the kernel DPC dispatcher after the call instruction in the logical original instruction stream. This effectively prevents PatchGuard from ever running the system integrity check, which again gives the driver free reign to patch the kernel without fear of intervention by PatchGuard.

In the author's opinion, the best way to prevent this approach is to use a multitude of different mechanisms to kick off execution of the PatchGuard check routine. For example, a dedicated thread waiting on a timer could also be used, or a frequently-called system routine could be modified to periodically make calls to PatchGuard. As long as calls to PatchGuard are funneled through one location, such as the timer DPC dispatcher, the entire PatchGuard integrity check system is at risk of being trivially bypassed in one fell swoop by third party drivers.

4.7 Searching for the PatchGuard DPC

PatchGuard currently uses a `KTIMER` object with an associated `KDPC` object, both allocated within non-paged pool memory, as a periodic trigger used to start PatchGuard's system integrity check routine. It should be possible to locate this timer object in memory and cancel it, preventing PatchGuard from executing.

The implementation of this technique is essentially a classical egghunt style search through non-paged pool, with some specially defined restrictions as to how to find the target. Specifically, one is looking for a region of memory matching the following criteria:

1. The memory is a valid KTIMER object. This means that the linked list entries should be valid, and point to other seemingly valid KTIMER objects (or the list head), and that the type field of the KTIMER is consistent with a timer object.
2. The timer should have a timer interval in the range of several minutes. PatchGuard applies a randomized fuzz factor to the timer interval (within a small range), but verifying that the range of the timer is no more than several minutes (say 7 or 8) should be an ample sanity check.
3. The timer should have a KDPC associated with it (and the pointer should be valid non-paged pool).
4. The associated KDPC should have the appropriate type field.
5. The associated KDPC should have a DPC routine that is within the confines of the kernel image in-memory.
6. The associated KDPC should have a DeferredContext value that is a non-canonical kernel address.

After the DPC is found, it can simply be canceled in order to disable PatchGuard. Similar approaches as could be used to prevent technique 4.6 would be applicable here. Specifically, a diversity in the set of paths leading up to PatchGuard's execution would make a technique like this, which is targeted at one path (such as the timer DPC dispatcher) less effective at blocking PatchGuard.

4.8 TLB Desynchronization (Split TLB)

All x86 processors supporting protected mode and paging employ a caching scheme to speed up the translation of virtual addresses to physical addresses. This scheme is implemented via a set of Translation Lookaside Buffers, or TLBs, which cache the contents of the page attributes (and associated physical address) for a given virtual address. Recent x86 processors (Pentium II-class or later) utilize several sets of TLBs, such as one set of TLBs for data accesses and one set of TLBs for instruction accesses. In normal system operation, both TLBs (if a processor supports multiple TLBs) maintain consistent views for the attributes of a particular page; however, it is possible to deliberately desynchronize the contents of these TLBs, thereby maintaining the illusion that a single page has different attributes depending on whether it is referenced as data

or as executable code. This deliberate desynchronization of TLBs has many uses, from the implementation of no-execute support (utilized by PaX/GRsec on GNU/Linux [6]) to “memory cloaking”, a technique often used by rootkits to provide one view of memory when memory is referenced as data by a read operation, and a different view of memory if memory is referenced by an instruction fetch. This same memory cloaking technique that has appealed to rootkit developers for the purpose of hiding rootkits from detection can also be used to hide kernel patching from PatchGuard’s integrity check. Strictly speaking, this proposed technique is not a bypass mechanism for PatchGuard; rather, it is a mechanism to hide kernel patching from PatchGuard, thus making PatchGuard harmless to third parties that are patching the kernel.

The details of this approach are essentially similar in many respects to that of any program implementing a split-TLB approach to altering page attributes or contents based on execute or read fetches. The exact details behind how this can be accomplished are beyond the scope of this paper, and are discussed elsewhere, by the PaX team (in the context of implementing no-execute on legacy platforms) [6], and by Sherri Sparks and Jamie Butler (in the context of implementing a Windows rootkit that utilizes split-TLBs to implement so-called “memory cloaking”) [7]. Interested readers are encouraged to review these references for the raw details on how the general split-TLB concept is implemented. Although the referenced articles directly apply to x86, the concepts apply in principle to x64 as well, and can likely be made to work on x64 with minimal modification.

After one has established a mechanism for desynchronizing TLBs (such as by hooking the page fault handler), the recommended approach for this technique is to desynchronize the TLBs for any regions in the kernel where one is performing traditional opcode-replacement-based patching or hooking. Specifically, when kernel code is read for execute on a page where an opcode-replacement-based patch is in place, then the patched page should be returned. If kernel code is read for a data reference (such as PatchGuard making a read of kernel code to validate its integrity), then the original data should be returned. This technique effectively hides all modifications to kernel code to any access other than direct execution, which prevents PatchGuard from detecting that kernel code has been altered by a third party.

Note that in order for this approach to succeed, the hook on the page fault handler itself must be hidden from PatchGuard. This cannot be directly accomplished by the same TLB desynchronization tactic, as the page fault handler must remain resident. A combined approach, such as utilizing a debug breakpoint on the page fault handler (when coupled with a subverted debug trap handler, perhaps via PsInvertedFunctionTable as described previously in technique 3) along with a scheme to prevent PatchGuard from disabling debug-register-based breakpoints (such as described in technique 5) might be needed in order to hook the page fault handler in a manner truly transparent to PatchGuard.

The most logical defense for this approach is to attempt to detect a compromise in the page fault dispatching path. Because TLB desynchronization cannot in general be used to hide the page fault handler itself (the page fault handler must remain marked present in memory), it would be difficult for a third party to conceal the alteration to the page fault handler from the kernel. This difficulty would be expressed in a limited number of ways in which alterations to the page fault handler could be hidden, such as by clever utilization of debug registers. As a result, the key to preventing this technique from remaining viable is to develop a way for PatchGuard to detect the page fault hook. If, for example, the debug trap handler and a debug breakpoint on the page fault handler were used to gain control on a page fault, then Microsoft might be able to prevent this technique by blocking or detecting the interception of the debug trap handler. One such approach might be to better secure PsInvertedFunctionTable, which represents an easy way for a third party to subvert the debug trap handler without PatchGuard's knowledge. Such counters will vary based on the mechanism used to hide the page fault handler hook, however.

4.9 DPC Routine Patching

A variation on technique 4.2, a very simple-minded approach to disabling PatchGuard would be to simply hook every possible DPC routine, check if the DPC is probably being called in order to execute PatchGuard's system integrity check, and if so, simply returning from the DPC to the kernel timer DPC dispatcher. In order to implement this approach, one first needs to locate each possible DPC routine. Technique 4.2 lists a number of viable algorithms for fingerprinting (and locating) each DPC routine; any (preferably multiple) of the suggested algorithms in that technique would be directly applicable to this proposed approach.

After one has identified all the possible DPC routines, all that is left is to patch each one to branch to driver controlled code. From there, the driver could make the decision as to whether the DPC is being invoked legitimately, or whether it is being invoked as part of PatchGuard's system integrity check process (easily identified by a non-canonical kernel address being passed as DeferredContext). If the DPC is PatchGuard-related, then all the driver need do to block PatchGuard is to immediately return to the DPC dispatcher.

This approach is fairly trivial to prevent (from Microsoft's point of view). Because it is signature-based, one possible counter-approach Microsoft could implement would be determining which signatures third party drivers use to detect PatchGuard DPCs, and altering the PatchGuard DPC routines to not match those signatures in the next PatchGuard version. Microsoft could also change the number of DPC routines to throw off drivers that assume PatchGuard will use exactly ten DPCs, or Microsoft could switch to an alternative delivery mech-

anism other than DPCs in order to prevent existing code that detects and hooks specific DPC routines from blocking PatchGuard.

Chapter 5

Subverting PatchGuard

PatchGuard currently possesses a formidable array of defensive mechanisms that are aimed at making it difficult to reverse engineer and debug. Given that Microsoft does not currently have in place the infrastructure to make PatchGuard enforced by hardware, this is arguably the best that Microsoft will ever really be able to do in the short term. They're only able to build a system that is based on obfuscation and anti-debugging techniques in an attempt to make it difficult for third parties to detect, disable, or bypass it.

There are other classes of software that seek to create defenses similar to those of PatchGuard's. However, these other classes usually have far more nefarious purposes than preventing third parties from patching the kernel. Specifically, anti-debugging, anti-reverse-engineering, and self-decrypting code have often been used to hide viruses, rootkits, and other malicious software on compromised systems.

Although Microsoft may have intended the defensive mechanisms employed by PatchGuard for an (arguably) good cause, these same anti-debugging, anti-detection, and anti-reverse-engineering techniques that protect PatchGuard from attack by third party drivers can also be subverted to protect custom code from detection or analysis by anti-virus or anti-rootkit software. With this respect, Microsoft has created a double-bladed-sword, as the same elaborate obfuscation and anti-debugging schemes that guard PatchGuard against third party software can also be used to guard malicious software from system security software. It is in fact quite possible to subvert PatchGuard version 2's myriad defenses to execute custom code instead of PatchGuard's system integrity check routine. While doing so might not be exactly called trivial, it is far from impossible.

In order to subvert PatchGuard to do one's bidding, one must first catch PatchGuard in the act, so to speak. To accomplish this, the author recommends turning to one of the proposed bypass techniques as a starting place. For example,

consider the first proposed bypass technique, wherein the author recommends hooking `_C_specific_handler` to intercept control of execution at the exception generated by the PatchGuard DPC routine in order to trigger execution of the system integrity check. An implementation of this bypass technique provides direct access to the machine context inside the PatchGuard DPC routine, and this machine context contains the information necessary to locate the PatchGuard system integrity check routine.

Since the objective is to repurpose the system integrity check routine to execute custom code, this is a good starting point. However, determining the location of the system integrity check routine is much more involved than simply skipping over PatchGuard's checks entirely; the pointer to the routine in question is encrypted based off of the original arguments to the DPC (the `Dpc` and `DeferredContext` arguments). Additionally, the original arguments to the PatchGuard DPC have at this point already been moved from registers to the stack and obfuscated (rotated left or right by a magical constant). As the original contents of the argument registers are deliberately overwritten by the DPC routine before the access violation is triggered, there is no choice other than to somehow fish the DPC arguments out of the caller's stack. This is actually somewhat of a challenge, given that such an approach must work for all kernel versions, and must also work for all of the different DPC permutations. Since this set of possibilities represents an unmaintainably large number of routines to reverse engineer in order to determine rotate obfuscation values and stack offsets, a more generalized approach to locating the original arguments on the stack must be taken. In order to create such a generic approach, one must take a closer look at the first few instructions of each DPC routine (leading up to the intentional access violation). Although PatchGuard has put into place several barriers to prevent easy retrieval of the original arguments from this context, there might be a pattern or weakness that could be exploited in order to recover the arguments in question.

The basic things common to each DPC routine, when it comes to the machine context at the time of the access violation, are:

1. The original arguments have been stored on the stack in an obfuscated form (rotated left or right by an arbitrary magical constant).
2. The access violation always occurs by dereferencing `rax`. Here, `rax` is always the deobfuscated form of the `DeferredContext` argument. This gives us one of the arguments for free, as `rax` in the register context at the time of the access violation is always the plaintext `DeferredContext` value.
3. The stack location where the `Dpc` argument is stored at varies greatly between DPC version to DPC version. Furthermore, it also varies between different kernel flavors within an operating system family, and between

operating system families. As a result, it is not practical to hardcode stack displacements for this argument.

4. The instruction immediately prior to the faulting instruction is always an instruction in the form of `ror rax, <magical constant>`. Here, the magical constant is an immediate value, which means that it is encoded as a part of the opcode for this instruction itself. Each DPC has its own unique magical constant, and the magical constants used do not change for a particular DPC flavor across all kernel flavors and operating system families. This gives us a nice way to quickly identify which of the ten PatchGuard DPCs is in use from the context of the `.C_specific_handler` hook (without having to do ugly code fingerprinting or analysis). Unfortunately, we still don't have a way to determine the stack displacement of the Dpc argument.
5. The `r8` register is always equal to the original Dpc argument, shifted right by the low byte of the DeferredContext argument. Although this may seem tantalizingly close to what we're looking for, it can't actually be used as a substitute for the original Dpc argument, even though the DeferredContext argument is known here (due to the value of `rax`). This is because the right shift operation is destructive, in that information is permanently lost as bits are shifted right off of the register into oblivion. As a result, depending on the low byte of the DeferredContext argument, important bits in the Dpc argument have already been permanently lost in the pseudo-copy residing in `r8`.

Although the situation may initially appear grim, it is in fact still possible to locate the Dpc argument given the above information; all that is needed is a bit of work (and getting one's hands dirty with some ugly tricks). Specifically, it is possible to search the stack frame of the DPC routine for the Dpc argument with a brute-force attack. This isn't exactly elegant, but it gets the job done. There are a number of hints that can be used to increase the chance of successfully finding the real Dpc argument on the stack:

1. The stack is 8-byte aligned (at least) due to x64 calling convention requirements, and the Microsoft C/C++ compiler will always place pointer-sized values on the stack in 8-byte-aligned locations. As a result, the search can be narrowed down to 8-byte-aligned locations on the stack, instead of a bitwise search.
2. Because the identity of the current DPC routine is known (due to analyzing the `ror` instruction immediately preceding the faulting `mov eax, [rax]` instruction), the rotate constant used to obfuscate the Dpc argument is known. Each DPC routine has its own unique magical rotate constant, and as the current DPC routine has been positively identified, the rotate constant used to obfuscate the Dpc argument on the stack is thus also known.

3. A quick check as to whether a value on the stack could possibly be the Dpc argument can be made by rotating the value on the stack by the known obfuscation constant, then shifting the value right by the low byte in the DeferredContext argument and comparing the result to the r8 value at the time of the exception. If there is a mismatch, then the current stack location can be eliminated from the search. This does not provide a positive match, but it does provide a way to positively eliminate possibilities. This step is also optional, in that it is still possible to locate the Dpc argument without relying on r8; the check against r8 is simply an optimization.
4. The Dpc argument should point to a valid non-paged pool address, given that it must represent a valid kernel pointer. In order to check that this is the case, MmIsAddressValid can be used to test whether the deobfuscated value in question is a valid pointer or not. (Yes, MmIsAddressValid is a bit of a race condition and certainly a hack. The author would like to note that this approach was described as requiring that the implementor get his or her “hands dirty with some ugly tricks”, in an attempt to forstall the inevitable complaints about how this approach might be decried as an un stomachable ugly hack by some.)
5. The Dpc argument should point to a valid non-paged pool address whose length is great enough to contain a KDPC object, plus at least one pointer-sized additional field. A secondary MmIsAddressValid test can be used to verify that the pointer describes a valid region large enough to contain the KDPC object, plus the additional pointer-sized field following it (the PatchGuard decryption key).
6. The Dpc argument should point to a DPC whose Type and DeferredContext arguments have been zeroed. (The DPC routine intentionally zeros these values in the DPC before intentionally triggering an access violation.) If the suspected Dpc argument, when treated as a PKDPC, does not have these properties then it can be eliminated as a possibility.

By repeatedly applying these rules to every applicable location within a reasonable distance upward from the rsp value at the time of the exception (say, 256 bytes, although the exact size can be greater; the only requirement is that the entire local variable space of the DPC routine with the largest local variable space is completely contained within the search region), it is possible to recover the Dpc argument with virtual certainty. In the author’s experience, this technique works quite reliably, despite that one might intuit that a search of an unknown stack frame might be prone to failing or turning up false positives.

After both the Dpc and DeferredContext arguments to the PatchGuard DPC routine have been recovered, it is a simple matter of analyzing how PatchGuard invokes the system integrity check in order to determine how to locate it in memory. This has been discussed previously, and it amounts to the following set of statements:

```

ULONG64 DecryptionKey, PatchGuardCheckFunction;

DecryptionKey      = *(PULONG64)(Dpc + 0x40);
PatchGuardCheckFunction = DecryptionKey ^ DeferredContext;
PatchGuardCheckFunction |= 0xFFFFF80000000000;

```

At this point, it's almost possible to replace the system integrity check routine with custom code. However, there is still the matter of the pesky self-decrypting stub that runs before the check function. Because the DPC routine's exception handler rewrites the first instruction of the stub before it is executed, one doesn't have a whole lot of choice but to implement at least a very basic version of the decryption stub for the system integrity check routine.

Recall that the first instruction in the stub is set to the following:

```
lock xor qword ptr [rcx],rdx
```

Looking at the prototype for the decryption stub, `rcx` corresponds to the address of the decryption stub itself, and `rdx` corresponds to the decryption key. Since this instruction modifies both itself and the next instruction (the instruction is four bytes long and the xor alters eight bytes), the replacement code for the system integrity check routine must allow the first instruction to be the above xor instruction, and the must allow for the second instruction (at a minimum) to be initially xor-obfuscated. For simplicity's sake, the author has chosen to implement the simplest possible solution to this conundrum, which is to make the second instruction in the replacement code a duplicate of the first instruction. In other words, the replacement code would read as follows:

```

;
; This instruction is forced on us by PatchGuard,
; and cannot be altered; it is rewritten at runtime.
;
lock xor qword ptr [rcx],rdx

;
; The next instruction, conveniently four bytes
; long, re-encrypts itself by xoring the first
; eight bytes of the decryption stub (which includes
; the second instruction) by the decryption key a
; second time;
;
lock xor qword ptr [rcx],rdx

;
; (... any custom code may follow here ...)
;

```

As noted previously, after specially constructing the replacement code, it is necessary to initially encrypt the second instruction (as it will be immediately

decrypted by the first instruction). This must be done before control is returned to PatchGuard.

After the custom code is configured and the second instruction is encrypted, all that remains is to copy the custom code over the PatchGuard decryption stub. When this is accomplished, the PatchGuard DPC's exception handler will invoke the supplied custom code instead of the system integrity check routine.

However, this is not really all that interesting due to the fact that PatchGuard utilizes a one-shot timer. The custom code that was substituted for the decryption stub will never be run again. To account for this fact, it would be prudent to place a call to queue a timer with an associated DPC routine (pointing to the DPC routine that PatchGuard selected at boot) within the custom code block.

At this point, it is possible to simply allow the normal exception dispatching process to continue (i.e. to resume `_C_specific_handler`), after which the custom code will be invoked instead of PatchGuard. In essence, PatchGuard has been not only disabled, but completely subverted to call customized code under the control of a third party driver instead of the system integrity check.

Still, the situation is less than optimal. Presently, there is still a hook in `_C_specific_handler` that is there for anyone to see (and recognize that someone has tampered with the kernel). Additionally, the driver that was used to subvert PatchGuard in the first place is still loaded, which may also be a tell-tale giveaway sign that someone may have done something unsavory to the kernel.

These problems are also solvable, however. It turns out that after PatchGuard has been subverted, it is safe to unhook from `_C_specific_handler`, and then simply call back into `_C_specific_handler` after the hook is removed. Furthermore, everything necessary to run the subverted system integrity check routine could even reside within PatchGuard's own internal data structures; for example, one could simply utilize extra space after the custom code, where the decryption stub and PatchGuard check routine would normally reside as a parameter block. This is especially convenient, as the custom code block is given a pointer to itself in `rcx` (the first argument), and it is easy to add a known constant value to that pointer in order to retrieve the parameter block for the custom code. At this point, all of the code and data necessary for the custom code that the driver has subverted PatchGuard with is located in dynamically allocated memory. Given this, the original driver is no longer needed and can even be unloaded (so as to further disguise the fact that any alterations to the kernel have taken place). After the driver has been unloaded, the only traces of the alterations that have taken place would be the unloaded module list (easily modified), and the re-written PatchGuard system integrity routine itself (which could easily be bolstered to be self-decrypting (with a differing encryption key in order to make for an extremely difficult to locate target in-memory)).

The end result is that PatchGuard has been disabled, and in its place, arbitrary custom code is periodically executed. Furthermore, no modifications or

patches to kernel code or global data are present and no suspicious drivers (or even suspicious extraneous memory allocations) remain present in memory. In essence, the only traces of the fact that PatchGuard has been subverted would be visible only to someone (or something) that knows how to locate and disable PatchGuard.

The supplied example program for subverting PatchGuard is fairly simple, and it does not utilize all of the defensive technologies employed by PatchGuard. For instance, it does not change the decryption key on every execution, nor does it follow through with keeping the entire code block encrypted except just before execution. These features could be easily added, however, and would greatly increase the difficulty of locating the subverted PatchGuard code in memory.

Chapter 6

Future Direction of PatchGuard and “Anti-Hack” Systems

In the future, there are a couple of generalized approaches that Microsoft could take to significantly strengthen PatchGuard against attack. Specifically, these involve adding redundancy and removing single points of failure from PatchGuard. It is often helpful to look at an anti-hack system like PatchGuard as a critical system that one would like to keep running at all times with minimal downtime (i.e. a network or service with high-availability). The logical way to accomplish that goal is to locate and eliminate single points of failure, such as by adding redundancy. In a high availability network, one would accomplish this by adding redundant cables, switches, and the like, such that if one component were to fail, the system as a whole would continue to operate instead of failing entirely. With an anti-hack system such as PatchGuard, it is helpful to add redundancy to all critical code paths such that there is no single point where an attacker can simply change an opcode or hook in with the end result of disabling the entire anti-hack system.

Removing these single points of failure is critical to the longevity of an anti-hack system. The main concept to grasp in such cases is that the attacker will always try to seek out the easiest way to break the defenses of the target system. All the obfuscation and encryption in the world does little good if an attacker can simply change a jmp to a nop and prevent elaborate encryption and anti-debugging facilities from ever getting the chance to run. In this respect, PatchGuard is flawed in its current implementation. There are many different single points of failure where an attacker could inject themselves at a single place and completely disrupt PatchGuard.

One possible solution to this problem might be to ensure that there are multiple different code paths that can lead to every point in the PatchGuard system integrity check. The nature of the battle between anti-hack systems and attackers relates to how easy it is to bypass the weakest link in the anti-hack system. Until all of the weak links in the system are shored up simultaneously, the system remains much more vulnerable to easy attack or bypass. With this respect, PatchGuard version 2 does little to improve on the weakest links of the system and as such there are still a vast number of ways to bypass it. Even worse, each bypass technique is often only required to attack one specific aspect of PatchGuard in order to disable it as a whole.

As far as PatchGuard itself is concerned, one approach that Microsoft could take to significantly increase the resiliency and robustness of the system to outside interference would be to merge some sort of critical system functionality with the PatchGuard system integrity check. Such an approach would make it difficult for a would-be attacker to simply bypass a call to PatchGuard, as doing so would also bypass some sort of critical system functionality that would (ideally) be required for the system to operate in any usable capacity. At this point, the challenge for attackers then turns into either replicating the critical system functionality that is contained within PatchGuard, finding a way to split the critical system functionality away from the system integrity check portions of PatchGuard, or finding a way to evade PatchGuard's detection of kernel patching entirely. Microsoft can make the first two points arbitrarily difficult, especially since the knowledge of Windows internals is presumably greater inside Microsoft than outside Microsoft. The incorporation of critical system functionality would be theoretically easier for Microsoft to do than it would be for would-be attackers to reliably reverse engineer and re-implement such functionality on their own, forcing would-be attackers to take the hard route of trying to separate PatchGuard from critical system functionality. This is where clever use of obfuscation and anti-debug techniques would really see maximum effectiveness, as an attacker would (optimally) have no choice other than to step through and understand PatchGuard entirely before being able to replicate the critical functionality contained within PatchGuard (or selectively activate the critical functionality without activating the system integrity check).

The latter problem (evading PatchGuard detection entirely) is likely to be a much more difficult one to tackle, however. Techniques such as the clever use of debug registers, TLB desynchronization, and other related attacks are extremely difficult to detect (and typically very easy to alter to avoid detection after a known detection scheme for such attacks is developed). In this particular respect, Microsoft is presently at a great disadvantage. Improving PatchGuard to avoid such evasion tactics is likely to prove both difficult and a poor investment of time relative to how quickly attackers can adapt and compensate for Microsoft's efforts at bolstering PatchGuard's capabilities.

Looking towards the future, it can be expected that PatchGuard will ultimately see the obfuscation-based defensive mechanisms currently in place substituted

with hardware-based defensive mechanisms. In particular, the author expects that Microsoft will eventually deploy a PatchGuard version that is augmented by the hardware-based virtualization (also known as hypervisor) support present in recent processors (and being developed for Windows Server “Longhorn”, code-named “Viridian”). An implementation of PatchGuard that is guarded by a hypervisor would be immune to being simply patched out of existence (which eliminates some of the most significant flaws in current versions of PatchGuard), at least as long as the hypervisor itself remains secure and free from exploitable bugs. In a hypervisor-based system with PatchGuard, third party drivers would not be permitted to execute with hypervisor privileges, thus completely preventing runtime patching of PatchGuard itself (which would be a part of the privileged hypervisor layer). A hypervisor-based system might also be able to implement concepts such as write-once memory that could be adapted to prevent the kernel from being patched in the first place once it is initially loaded into memory (as opposed to detecting patching after the fact, and bringing down the system in response to third party drivers performing underhanded deeds).

Even with hypervisor support in-place, however, it is anticipated that there will still be ways for third parties to alter the behavior of the kernel in ways not completely authorized by Microsoft. For instance, as long as support for debug registers must be retained in order for the kernel debugger to function, it may be difficult to prevent an approach that utilizes debug registers to modify execution context at arbitrary locations within the kernel (at least, not without making the hypervisor completely responsible for managing all activities relating to the processor’s complement of debug registers).

Chapter 7

Conclusion

Although PatchGuard version 2 introduces significant improvements in some areas, it still remains vulnerable to a wide variety of potential attacks. Additionally, it is possible (though involved) to subvert PatchGuard entirely, with the purpose of running arbitrary custom code in a difficult-to-detect manner in the place of PatchGuard.

With these points in mind, it is perhaps time to re-evaluate whether PatchGuard, in its current incarnation, is really worth all the trouble that Microsoft has put into it. Although forcing the IHV and ISV world to clean house with their kernel mode code is certainly a reasonable goal (and one which ultimately benefits all Windows customers, no matter how certain companies with poorly written kernel mode code [8] may care to spin the facts), as badly written kernel mode code results in the chronic instability that Windows is often associated with (at best), and privilege escalation and arbitrary code execution exploits in the worst case. However, there are still significant counterpoints to what PatchGuard represents; the fact that it may provide a convenient way for malicious kernel mode code to hide in a very difficult to detect manner, and that there is real innovation that is stifled by the restrictions that PatchGuard places on the system. As an example of the latter, consider that Microsoft's very own Virtual Server 2005 R2 SP1 (Beta) runs afoul of PatchGuard and requires a special kernel hotfix to alter what, exactly, PatchGuard protects in order to run without bugchecking the system with the infamous `CRITICAL_STRUCTURE_CORRUPTION` bugcheck made famous by PatchGuard [3]. This alone should be taken as an indicator that there *are* in fact legitimate uses for some of the techniques that PatchGuard prevents, despite Microsoft's insistence to the contrary. It should also be noted that despite Microsoft's statements that no exceptions would be made for PatchGuard [1], they have had to make adjustments at least once for their own code to run on PatchGuard. The conspiracy theorists among you might wonder whether Microsoft would be so gracious as to make such exemp-

tions for legitimate uses of techniques blocked by PatchGuard for third party software with similar needs as Virtual Server 2005 R2 SP1, given their pointed statements to the contrary.

As a final note relating to the objectives of PatchGuard, even with hypervisor technology deployed (and furthermore, even with so-called immutable memory as implemented by a hypervisor), there is little that can be done to protect drivers from each other, as even in a hypervisor based system (where the kernel itself is protected from drivers), interdependent drivers will still be able to interfere with each other so long as they co-exist in the same domain. This is particularly problematic in Windows, given the concepts of device stacks and device interfaces that allow drivers to directly interact with each other in a variety of ways. It will be difficult to ensure that drivers do not resort to patching each other (or modifying pool allocations instead of patching code, in the case where immutable memory on code regions is being enforced by a hypervisor). Depending on what the objectives of a third party ISV attempting to bypass PatchGuard are, it may be possible to simply patch drivers (such as Ntfs.sys or Tcpip.sys) in lieu of patching the kernel. From this perspective, it is unlikely that Windows will ever become an environment where kernel mode drivers are completely isolated and unable to interfere with each other, despite the efforts of technologies such as PatchGuard.

Microsoft has already started down a path that may eventually lead to a system where buggy drivers will be unable to crash the system (or patch each other), with the advent of the User Mode Driver Framework (UMDF). It remains to be seen whether isolated user-mode based drivers will become a viable alternative for high performance devices (such as PCI/PCI Express as opposed to USB devices), however, instead of simply being confined to a small subset of the devices that ship with a typical computer. The author expects that wherever possible, Microsoft will attempt to move third party code outside of sensitive areas (like the kernel) and into more contained locations (such as a user-mode process). This is in-line with the purported goals of PatchGuard; increasing system stability by preventing third party drivers from performing questionable actions (or at least, questionable actions in such a way that might bring down the system).

Bibliography

- [1] Microsoft Corporation. *Patching Policy for x64-Based Systems*.
<http://www.microsoft.com/whdc/driver/kernel/64bitpatching.aspx>; accessed December 10, 2006.
- [2] skape, Skywing. *Bypassing PatchGuard on Windows x64*.
<http://uninformed.org/index.cgi?v=3&a=3&t=sumry>; accessed December 10, 2006.
- [3] Microsoft Corporation. *Connect: Virtual Server 2005 R2 SP1 Beta*.
<https://connect.microsoft.com/site/sitehome.aspx?SiteID=151>;
accessed December 28, 2006.
- [4] Advanced Micro Devices, Inc. *AMD 64-Bit Technology*
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/x86-64_overview.pdf; accessed December 28, 2006.
- [5] Microsoft Corporation. *RtlVirtualUnwind*.
<http://msdn2.microsoft.com/en-us/library/ms680617.aspx>; accessed December 28, 2006.
- [6] The PaX Team. *Paging Based Non-Executable Pages*.
<http://pax.grsecurity.net/docs/pageexec.txt>; accessed December 30, 2006.
- [7] Sherri Sparks and Jamie Butler. *"SHADOW WALKER" Raising the Bar for Rootkit Detection*.
<http://www.blackhat.com/presentations/bh-jp-05/bh-jp-05-sparks-butler.pdf>; accessed December 30, 2006.
- [8] Skywing. *Anti-Virus Software Gone Wrong*.
<http://www.uninformed.org/?v=4&a=4&t=sumry>; accessed December 31, 2006.