PatchGuard Reloaded

A Brief Analysis of PatchGuard Version 3

September, 2007

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

Abstract

Since the publication of previous bypass or circumvention techniques for Kernel Patch Protection (otherwise known as "PatchGuard"), Microsoft has continued to refine their patch protection system in an attempt to foil known bypass mechanisms. With the release of Windows Server 2008 Beta 3, and later a full-blown distribution of PatchGuard to Windows Vista and Windows Server 2003 via Windows Update, Microsoft has introduced the next generation of PatchGuard to the general public ("PatchGuard 3"). As with previous updates to PatchGuard, version three represents a set of incremental changes that are designed to address perceived weaknesses and known bypass vectors in earlier versions. Additionally, PatchGuard 3 expands the set of kernel variables that are protected from unauthorized modification, eliminating several mechanisms that might be used to circumvent PatchGuard while co-existing (as opposed to disabling) it. This article describes some of the changes that have been made in PatchGuard 3. This article also proposes several new techniques that can be used to circumvent PatchGuard's defenses. Countermeasures for these techniques are also discussed.

Contents

Chapter 1

Introduction

PatchGuard is a controversial feature of Windows x64 editions, starting with Windows Server 2003 x64 / Windows XP x64, and continuing on with Windows Vista x64 and Windows Server 2008 x64. The design goals behind PatchGuard are to prevent the kind of rampant hooking and modification of various kernel code and data structures that has been so common on x86 versions of Windows. Microsoft has stated that the vast majority of kernel crashes are caused by third party drivers, and the author's experiences with Windows firmly support this supposition. Because accessing internal kernel data structures and hooking kernel functions typically requires intricate synchronization with the rest of the system in order to be performed in a completely safe fashion, especially on multiprocessor machines, many third party drivers that perform these sorts of dangerous tasks have historically made egregious mistakes that have often lead to system stability or a compromise of system security. The latter is especially common in cases where third party programs hook functions, such as system calls, and subsequently fail to perform sufficient parameter validation.

Microsoft's solution to this problem is to attempt to forcibly prevent third party code from making unauthorized modifications to internal kernel data structures and code through technical means in addition to discouraging developers from performing such tasks. However, due to the nature of how the Windows kernel (and its supporting drivers) are designed, it is not feasible for kernel mode drivers to run at a lower effective privilege level than the kernel itself. This poses a problem with respect to Microsoft's goal of blocking unauthorized kernel patches due to the fact that there is no hardware-enforced separation between the kernel itself and third-party drivers. As such, said third party drivers have free reign to manipulate kernel code and data as desired.

Although emerging technologies such as TPM and hardware-assisted virtualization (hypervisors) may eventually provide a mechanism to deploy a hardware-

enforced boundary between certain key parts of the kernel and the third party drivers that interact with it, such an approach is not generally applicable to most computers sold today, given the current state of the technology involved (with respect to both hardware and software capabilities). Lacking a complete, hardware-enforced solution, Microsoft has turned to other approaches to dissuade third party software from making unauthorized kernel modifications. Specifically, the resulting kernel patch protection mechanism ("PatchGuard") is instead based on highly obfuscated code that, while running at the same effective privilege level as both the kernel itself and third party drivers, is designed to be resilient against detection and/or modification by third party drivers. This code is responsible for periodically checking the integrity of key kernel code and data structures and will bring down the system if such modifications are detected. By virtue of the fact that attempting to blithely patch the kernel as was once possible on Windows x86 editions, attempting to perform the same operations will result in a system crash on x64 versions of Windows. As such, third party drivers are effectively preventing from making such modifications on a large-scale basis with respect to code deployed on customer systems.

However, like all systems that are founded upon the principal of security through obscurity, PatchGuard has inherent weaknesses. These weaknesses can be exploited by third party drivers to either disable PatchGuard entirely or circumvent its checks altogether while peacefully co-existing with PatchGuard. Microsoft is fully aware of these deficiencies with respect to the fundamental approach taken by PatchGuard and has resorted to periodically updating PatchGuard in such a way as to block known public bypass techniques. The net result is that Microsoft gives the impression of a "moving target" to any ISV that would defy Microsoft's wishes with respect to circumventing PatchGuard. This helps to show that any code designed to stop or disable PatchGuard may become invalidated at some point in the future such as when Microsoft releases a new update for PatchGuard. This has resulted in a small arms race with code to circumvent PatchGuard being written by third parties, and Microsoft responding by developing and deploying countermeasures in the form of an updated version of PatchGuard that is not susceptible to these bypass techniques. This cycle has continued through several iterations already; in fact, PatchGuard is now being deployed to the general public in its third iteration.

Chapter 2

Protection Improvements

PatchGuard 3 implements several incremental improvements designed to protect PatchGuard from third party code attempting to disable it as compared to PatchGuard 2. The majority of the alterations to PatchGuard's self-defense logic appear to be direct responses to previously published, publicly-known bypass techniques, rather than general improvements meant to make PatchGuard 3 more resilient to analysis and attack. In this vein, while the alterations to PatchGuard 3 (over PatchGuard 2) are effective at disabling most previously-published bypass mechanisms that the author is aware of, it is not exceedingly difficult to alter many previous attack mechanisms to be effective against PatchGuard 3. Many of the protection systems that were implemented in PatchGuard 2 are still present in PatchGuard 3 in some form or another, though some of them have been altered to resist previously-published attacks.

This chapter will describe a number of specific improvements that have been made.

2.1 Multiple Concurrent PatchGuard Check Contexts

In previous PatchGuard releases, there existed a single PatchGuard check context that would periodically be used to verify the integrity of protected regions. Some bypass techniques relied on the fact that there existed only one PatchGuard context by virtue of disabling any invasive kernel patching that would be required to "catch PatchGuard in the act" after locating PatchGuard. PatchGuard 3 improves upon this by creating at least one PatchGuard context if PatchGuard is enabled, with a probability of a second context being initial-

ized at system boot time[1]. Both PatchGuard check contexts, which include all of the data used by PatchGuard to check system integrity (including the self-decrypting check routine in non-paged pool memory), operate completely independently if two contexts happen to be used for a particular system boot.

There are several advantages to randomly creating more than one check context. First of all, because the second context is not always created, an element of uncertainty is (theoretically) introduced into the testing and development process for PatchGuard bypass techniques, as it is possible that at first glance, an individual that is researching PatchGuard 3 might not notice that there is a chance to create more than one context. This may result in lost time during the debugging process, as some bypass techniques are affected by the number of active contexts. For example, the original bypass technique described by the author for PatchGuard 2 [1] effectively turned itself off after the first positive indication that PatchGuard was caught (although in this particular instance, the PatchGuard-catching hooks could have allowed to remain in place afterwards).

A better example of bypass techniques that might be affected by this sort of scheme are those that rely on searching system pool memory for a sign of PatchGuard. For example, a theoretical bypass scheme that operates by proactively locating the PatchGuard context in non-paged pool and disabling it somehow (perhaps by rewriting the self-decrypting code stub to expand into a no-operation function) might run afoul of this approach randomly during testing if it were not designed to re-try a pool memory scan after a positive hit on PatchGuard. It also eliminates the degree of confidence that such memory scan approaches provide, as previously, if one had a way to locate the PatchGuard context in non-paged pool memory, one would either know for certain that PatchGuard had in fact been disabled by getting a single hit (which could be taken as an indication that it would now be safe to perform actions blockedg by PatchGuard). With multiple check contexts having a probability to run, it is no longer possible for a bypass technique to have logic along the lines of "if a PatchGuard context has been located and disabled, then it is safe to continue", because there may exist a non-constant number of contexts in the wild.

2.2 Filtering of Exception Codes Used to Trigger PatchGuard Execution

Like PatchGuard 2, and PatchGuard 1 before it, the third iteration of PatchGuard is primarily executed through an unhandled exception in a DPC routine which, through the use of a series of structured exception handlers, eventually results in the self-decrypting PatchGuard stub being called in non-paged pool memory (based off of the DPC arguments). This presented itself as a liability,

[1]This is randomized based on the processor time stamp counter, as all other PatchGuard randomization is done

as evidenced by the previous article [2] published on Uninformed on the subject of disabling PatchGuard (release 2). The problem with using SEH to trigger execution is that there are a number of points in the SEH dispatching mechanism that can easily be modified by an external caller in order to gain execution after an exception is raised, but before a registered exception handler itself might be called.

Previous techniques exploited this weakness by positioning themselves in after the access violation exception raised when a PatchGuard-repurposed DPC routine dereferenced a specially-crafted invalid pointer argument but before the SEH logic that invokes the PatchGuard check context in response to the access violation exception. Specifically, the operating system exported routine used by the Microsoft C/C++ compiler for all compiler-generated SEH frames, `_C_specific_handler`, was targeted by bypass attempts described in the aforementioned articles. As the SEH frame responsible for running PatchGuard appears to have been written in C for PatchGuard releases 1 and 2, `_C_specific_handler` would be called before the user-supplied SEH logic which would be responsible for executing the PatchGuard integrity check logic contained within the current PatchGuard context. At this point, a bypass technique need only abort the execution of the PatchGuard check routine and cleanly extricate itself from the call stack to a known-good location in order to disable PatchGuard.

However, in order for such a bypass mechanism to properly function, one would need to ensure that the particular exception being examined by `_C_specific_handler` is in fact PatchGuard and not a legitimate kernel mode exception. Applying a PatchGuard-style bypass to the latter case would be disastrous and almost certainly result in the system crashing or being corrupted immediately after the fact. Given this, positively identifying an exception from the exception dispatcher interception point is key to any bypass technique built upon exception dispatcher redirection. While the previous two PatchGuard releases made identifying PatchGuard a trivial task. In both cases, a special form of invalid address, a "non-canonical address", is dereferenced to trigger the access violation that ultimately results in PatchGuard's check context being executed[2].

The advantage of using a non-canonical address is clear when one examines the PatchGuard execution environment for a moment. In Windows kernel mode programming, it is not generally possible to blindly dereference a bogus kernel mode pointer. This often results in a sequence of events that bring down the system, depending on where the dereferenced location is. A non-canonical address is a special (undocumented) exception to this rule, as the processor reports the exception via a general protection fault and not the typical page fault mechanism. In this case, the operating system reports the exception as an access violation related to an access of the highest kernel address (0xFFFFFFFFFFFFFFFF).

[2] A non-canonical address is an address that does not fall within the subset of a 64-bit address space presented by modern x64 processors

This distinct signature can be used to locate and disable PatchGuard in a relatively safe fashion, as bogus kernel mode addresses should never make it to SEH dispatching (in kernel mode) unless the system is about to crash due to a fatal driver or kernel bug (PatchGuard being a special case). Thus, it was previously possible to positively identify PatchGuard by looking for an access violation that referenced 0xFFFFFFFFFFFFFFFF.

PatchGuard 3 improves the situation somewhat by performing some pre-filtering of the exception data through an exception handler written in assembly (which thus does not invoke `_C_specific_handler`) before the `_C_specific_handler` based logic that actually invokes the PatchGuard check routine is executed. Specifically, the pre-filtering exception handler, whose code is given below, alters the exception code to take on a random value which overlaps with many valid kernel mode exceptions. For example, some status codes that are applicable to the file system space are used, such as `STATUS_INSUFFICIENT_RESOURCES`, `STATUS_DISK_FULL`, `STATUS_CANT_WAIT`. Additionally, the exception address is altered as well (in some cases even set to be pointing into the middle of an instruction), and the dereferenced address (the second exception parameter for access violations) is also set to a randomized value. After these alterations are made, the assembly-language exception handler passes control on to the `_C_specific_handler` based exception handler, which invokes PatchGuard. Annotated disassembly for one of the assembly-language pre-filter exception handlers is provided below:

```
;
; EXCEPTION_DISPOSITION
; KiCustomAccessHandler8 (
; /* rcx */ IN PEXCEPTION_RECORD               ExceptionRecord,
; /* rdx */ IN ULONG64                         EstablisherFrame,
; /* r8  */ IN OUT PCONTEXT                    ContextRecord,
; /* r9  */ IN OUT struct _DISPATCHER_CONTEXT* DispatcherContext
; );
KiCustomAccessHandler8 proc near
  test    [rcx+_EXCEPTION_RECORD.ExceptionFlags], 66h
loc_14009B4C7:
  jnz     short retpoint
  rdtsc
; Randomize ExceptionInformation[ 1 ]
; ( This is the "referenced address" for
;   an access violation exception.)
;
; ( Note that rax is not set to any
;   specific defined value in this
;   context.  It depends upon the value
;   that RtlpExecututeHandlerForException
;   and by extension RtlDispatchException
;   last set rax to. )
  mov     [rcx+(_EXCEPTION_RECORD.ExceptionInformation+8)], rax
  xor     [rcx+(_EXCEPTION_RECORD.ExceptionInformation+8)], rdx
  shr     eax, 5
  and     eax, 70h
  sub     [r8+98h], rax
```

```
  and     edx, 7Fh
  or      edx, 0C0000000h
; Set ExceptionCode to a random value.  The code
; always has 0xC0000000 set, and the lowest byte
; is always masked with 7F.  This often results
; in an exception code that appears like a
; legitimate exception code.)
  mov     [rcx+_EXCEPTION_RECORD.ExceptionCode], edx
  lea     rax, loc_14009B4C7+1
; Set ExceptionAddress to a bogus value.  In this case,
; it is set to in the middle of an instruction.  This
; may interfere with attempts to unwind successfully from
 the exception.
  mov     [rcx+_EXCEPTION_RECORD.ExceptionAddress], rax
; Set Context->Rip to the same
; bogus exception address value.
  mov     [r8+0F8h], rax
  and     qword ptr [r8+88h], 0
retpoint:
  mov     eax, 1
  retn

KiCustomAccessHandler8 endp
```

As a direct result of scrubbing the exception and context records by the assembly-language exception routine, it is no longer possible to use the old mechanism of looking for an access violation referencing `0xFFFFFFFFFFFFFFFF` in order to differentiate a PatchGuard exception from the many legitimate kernel mode exceptions. In other words, PatchGuard attempts to hide in plain sight amongst the normal background noise of kernel mode exceptions, the vast majority of which exist inside filesystem-related code.

2.3 Executing PatchGuard Without SEH

One recurring theme that has continued to remain a staple for PatchGuard since its inception is the use of structured exception handling to obfuscate the calls to PatchGuard. The intention here is to use the many differences of SEH between x64 and x86, and the lack of disassembler support for x64 SEH to make it difficult to understand what is happening when calls to PatchGuard are being made. Ironically, this use of x64 SEH as an obfuscation mechanism has been a catalyst for much of the author's research [2] into Windows x64 SEH. Today, it is the author's opinion that x64 exception handling is now publicly documented to an extent that is comparable (or even exceeds) that available for x86 SEH.

Although x64 SEH may have been useful as an obfuscation technology initially, it had clearly worked its way up to a major liability after PatchGuard 2 had been released. This is due to the fact that SEH-related aspects of PatchGuard had been successfully used to defeat PatchGuard on multiple occasions. With

the advent of PatchGuard 3, the authors of PatchGuard siezed the opportunity to extricate themselves in some respect from the liability that x64 SEH had become.

PatchGuard 3 introduces a special mode of operation that allows it to function without using SEH. This is a significant change (and improvement) with respect to how PatchGuard has traditionally operated. It eliminates a major class of single points of failure in that the exception dispatching path is particularly vulnerable to external interference in terms of third party drivers intercepting SEH dispatching before control is transferred to actual exception handlers. The SEH-less mode of PatchGuard 3 operates by copying a small section of code into non-paged pool memory (as part of a PatchGuard context block). This code is then referenced by a timer object's DeferredRoutine at the non-paged pool location in question. The code referred to by the timer object is essentially a stripped down version of what happens when any of the re-purposed DPC routines are invoked by PatchGuard: it sets up a call to the first stage self-decrypting stub that ultimately calls the system check routine.

By completely eliminating SEH as a launch vector for PatchGuard, many bypass techniques that hinged on being able to catch PatchGuard in the SEH dispatching code path are completely invalidated. In an example of defense in depth in terms of software protection systems, the old, SEH-based system is still retained (with the previously mentioned modifications), such that a would-be attacker now has multiple isolated launch vectors that he or she must deal with in order to block PatchGuard from executing. Annotated disassembly of the direct call routine that is copied to non-paged pool and invoked without SEH is presented below:

```
KiTimerDispatch proc near
  pushf
  sub     rsp, 20h
  mov     eax, [rsp+28h+var_8]
  xor     r9d, r9d
  xor     r8d, r8d
  mov     [rsp+28h+arg_0], rax
; [rcx+40] -> PatchGuard Decryption Key
  mov     rax, [rcx+40h]
  mov     rcx, 0FFFFF80000000000h
  xor     rax, rdx
; Form a valid address for the PatchGuard context block by
; xoring the decryption key with the DeferredContext
; argument.
  or      rax, rcx
; Set the initial code for the stage 1 self-decrypting stub.
  mov     rcx, 8513148113148F0h
  mov     rdx, [rax]
  mov     dword ptr [rax], 113148F0h
  xor     rdx, rcx
  mov     rcx, rax
; Call the stage 1 self-decrypting stub.
  call    rax
```

```
  add     rsp, 20h
  pop     rcx
  retn
KiTimerDispatch endp
```

2.4 Randomized Call Frames in Repurposed DPC Routine Exception Paths

One of the bypass vectors proposed for PatchGuard 2 was to intercept execution at `_C_specific_handler`, detect PatchGuard, and resume execution at the return point of the PatchGuard DPC (i.e. inside the timer or DPC dispatcher). This is trivially possible due to the extensive unwind metadata present on Windows x64 combined with the fact that a DPC that has been re-purposed by PatchGuard does no useful work (other than invoking PatchGuard) and has no meaningful effect on any out parameters or return value.

In order to counteract this weakness, PatchGuard 3 introduces a random number of function calls when a re-purposed DPC is called, but before any exception is triggered. The intent with this randomization of the call frame stack is to invalidate the approach of always unwinding one level deep in order to effect a return from the DPC routine in question. Because there are a random number of call frames between the point at which an exception is raised and the start of the PatchGuard DPC routine, and the fact that the PatchGuard DPC routines are not exported, it is more difficult to safely return out of a PatchGuard DPC routine from the anywhere in the SEH dispatching code path.

An example of the call frame randomization code is provided below (in this case, ecx is initialized to small, random number that denotes the number of calls to make). There are a number of routines in the form of `KiCustomRecurseRoutineN` where N is [0..9], each identical.

```
KiCustomRecurseRoutine4 proc near
  sub     rsp, 28h
  dec     ecx
  jz      short retpoint
  call    KiCustomRecurseRoutine5
retpoint:
  mov     eax, [rdx]
  add     rsp, 28h
  retn
KiCustomRecurseRoutine4 endp
```

Although unwinds can still be performed, an attacker would need to be able to locate the actual return address of the PatchGuard DPC routine which might involve differentiating between the bogus `KiCustomRecurseRoutine` calls and the actual call into the DPC routine itself.

2.5 Expanded Set of Protected Regions

With the release of PatchGuard 3, Microsoft has added to the list of kernel global variables that are protected from unauthorized modification. Most notably, PatchGuard now appears to take an interest in PsInvertedFunctionTable, which was proposed as a key way to patch the kernel "under PatchGuard's nose", as it were, by providing an un-protected mechanism to gain execution at any point in the kernel that is traversed by an exception.

Chapter 3

Additional Protection Mechanisms

PatchGuard 3 and PatchGuard 2 both share some additional protection mechanisms that have not been previously described. This chapter includes a description of these protection mechanisms.

3.1 Timer List Obfuscation

PatchGuard 2 and PatchGuard 3 employ an obfuscation scheme that is used to obfuscate timer and DPC object pointers in the timer list. This obfuscation scheme hinges around two special kernel variables, `KiWaitAlways` and `KiWaitNever` that represent two random obfuscation keys that are calculated at boot time. These obfuscation keys are used to encode various pointers (such as links to DPC objects in a `KTIMER` object residing in the kernel timer list) that are intended to be protected from outside interference. For example, the following algorithm is used to decode the `KDPC` link in a `KTIMER` object when a timer DPC is going to be executed at expiration:

```
ULONGLONG Deobfuscated;
PKDPC     RealDpc;

Deobfuscated = Timer->Dpc ^ KiWaitNever;
Deobfuscated = _rotl64(Deobfuscated, (UCHAR)KiWaitNever);
Deobfuscated = Deobfuscated ^ Timer;
Deobfuscated = _byteswap_uint64(Deobfuscated);
Deobfuscated = Deobfuscated ^ KiWaitAlways;

RealDpc      = (PKDPC)Deobfuscated;
```

By virtue of being non-exported kernel variables, the original intention of such a scheme was to make it difficult for third party drivers to easily interfere with the timer list or certain other protected pointers. However, the algorithm itself is fairly easy to understand once one locates code that references it (such as most any timer-related code in the kernel), which simply leaves detecting the values of `KiWaitAlways` and `KiWaitNever` at runtime as the only remaining protection for the timer list to DPC object obfuscation.

Ironically, the kernel debugger extension `!kdexts.timer` implements the decoding algorithm (in `kdexts!KiDecodePointer`) so that a valid timer list can be presented to the user if the timer display command is invoked. Because the kernel debugger has access to PDB symbols for the kernel, it can trivially locate `KiWaitAlways` and `KiWaitNever`.

3.2 Anti-Debugging Code at PatchGuard Initialization Time

As with PatchGuard 2, PatchGuard 3 includes a sizable amount of anti-debugging code at runtime that is intended to frustrate attempts to step through the PatchGuard initialization routines with a debugger. Most of this code is based upon checking if a debugger is present while the PatchGuard initialization routines are executing (which should not typically occur as the PatchGuard initializtion routines are only called if a debugger is not attached), and if a debugger is so detected, disable interrupts and entering a spin loop so as to unrecoverably freeze the system.

Although this anti-debugging code may appear intimidating at first, disabling them is only a matter of locating all references to KdDebuggerNotPresent within the PatchGuard initialization routine and patching out the checks into the debugger. For example, the author used the following set of commands in the debugger at initialization time to disable the anti-debugging checks for Windows Vista x64 SP0, kernel version 6.0.6000.16514:

```
bp nt!KeInitAmd64SpecificState + 12 "r @edx = 1 ; r @eax = 1 ; g"
bp nt!KiFilterFiberContext
eb nt!KiFilterFiberContext+0x20 eb
eb nt!KiFilterFiberContext+0x19a eb


eb fffff800‘01c63d22 eb
eb fffff800‘01c64686 eb
eb fffff800‘01c652be eb
eb fffff800‘01c65334 eb
eb fffff800‘01c65880 eb
eb fffff800‘01c65a65 eb
eb fffff800‘01c67479 eb
eb fffff800‘01c68798 eb
```

```
eb fffff800‘01c6a940 eb
eb fffff800‘01c6b7a9 90 90
eb fffff800‘01c6b7dd eb
eb fffff800‘01c6bad9 eb
eb fffff800‘01c6d0e7 eb
eb fffff800‘01c6d2f6 eb
eb fffff800‘01c6d650 eb
eb fffff800‘01c65c3a 90 90 90 90 90 90
eb fffff800‘01c690b1 90 90 90 90 90 90
```

3.3 KeBugCheckEx Protection

One of the first bypass mechanisms proposed for PatchGuard 1 was to hook the code responsible for bugchecking the system[4]. From there, an attacker would simply resume normal system execution.

There are several defensive mechanisms in place to prevent this. In the the current version of PatchGuard, the entire contents of the thread stack are filled with zeros, making it difficult to resume execution of whichever thread was responsible for calling into PatchGuard. Furthermore, PatchGuard appears to make a copy of KeBugCheckEx at system initialization time, and copy this version over the actual code residing within the kernel at runtime just before bringing down the system in a bug check. This is clearly visible by making a modification to KeBugCheckEx in the debugger just as one enters the PatchGuard check context, and then setting a breakpoint on the internal function in the PatchGuard context to call KeBugCheckEx after clearing the stack and all registers. If one then examines KeBugCheckEx, any modifications that have been made will have vanished.

Additionally, PatchGuard appears to disable DbgPrint (patching it out with a "ret" opcode) before calling KeBugCheckEx. This may have been a (failed) attempt to prevent easy access to execution within KeBugCheckEx without actually patching KeBugCheckEx itself, which would circumvent the aforementioned protection on modifications to the bugcheck code itself. (KeBugCheckEx ordinarily utilizes DbgPrintEx to display a banner to the debugger when a bug check occurs. However, because PatchGuard only patches DbgPrint, there is no little to no effect in terms of what ends up happening when the bug check finally does happen.)

This code can be seen in the PatchGuard check routine, just before a call to the KeBugCheckEx wrapper is made. The pointer to DbgPrint is established during PatchGuard initialization at boot time.

```
mov     rax, [rbx+PATCHGUARD_CONTEXT.DbgPrint]
mov     byte ptr [rax], 0C3h ; '+' ; ret
```

3.4 Two-Stage Code Deobfuscation

One of the more interesting defensive features of PatchGuard 2 and PatchGuard 3 is the mechanism by which it obfuscates the PatchGuard check context, or the code and data necessary to verify system integrity. PatchGuard contexts are obfuscated such that they are completely randomized in-memory while inactive, and change their location and obfuscation keys (and thus contents) each time the context is invoked to check system integrity.

The decryption phase of PatchGuard is split into two stages. The first stage is essentially a small stub that remains completely obfuscated in-memory until just before it is called. The caller overwrites the first instruction in the stub that is called with a "lock xor qword ptr [rcx], rdx" instruction. The arguments to the stub are the address of the stub itself (in rcx), and the decryption key (in rdx). Thus, the first instruction now modifies itself (and more importantly the subsequent instruction, as each instruction is 4 bytes long but modifies 8 bytes of opcode bytes), which results in being another xor instruction. A small series of these xor instructions continues until the second stage of the decoding stub is completely decoded.

At this point, the second stage of the decoding stub is plaintext and may now execute. The second stage consists of a loop of xor operations starting at the end of the PatchGuard context and moving backward until the entire check routine is decoded. Additionally, the decryption key is shifted each xor round during the second stage decoding process.

After the second stage decoding loop is complete, control is transferred to the now-plaintext integrity check routine (all of the supporting data, such as critical function pointers into the kernel, will also have been translated into plaintext at this point by the second stage decoding loop).

Source code to a basic program to decrypt a PatchGuard memoy context is included with the article. The program expects to be supplied with a file containing "dq" logs from the kernel debugger that cover the entire memory context, along with the decryption key (at KDPC + 0x40) and KDPC-¿DeferredContext values.

3.5 Code Patching Support

Given PatchGuard's penchant for blocking attempts to patch the kernel, one would think that all kernel code is essentially expected to be fixed in stone at boot time. However, this is not really the case. There are a number of approved kernel patches that PatchGuard supports. For example, several functions (such as SwapContext) can be patched in approved ways if hypervisor support is

enabled. In the case of SwapContext, for instance, a runtime patch is made to redirect execution to EnlightenedSwapContext through a jump instruction being written to the start of the routine. PatchGuard appears to detect and permits patches to these functions through special exemptions (one can observe the address of functions such as SwapContext being stored in the PatchGuard context at initialization time, presumed to be for such a purpose).

The code responsible for checking the integrity of the SwapContext patch is provided below. Because the check ensures that a branch can only occur to EnlightenedSwapContext, it would be difficult to utilize this code to perform an arbitrary patch at SwapContext.

```
cmp     rdi, [rbx+PATCHGUARD_CONTEXT.SwapContext]
jnz     short NotSwapContextExemption
cmp     byte ptr [rdi], 0EBh ; 'd' ; backward jmps (short)
jnz     short NotSwapContextExemption
cmp     byte ptr [rdi+1], 0F9h ; ''
jnz     short NotSwapContextExemption
cmp     byte ptr [rdi-5], 0E9h ; 'T' ; jmp (long)
jnz     short NotSwapContextExemption
mov     rcx, [rbx+PATCHGUARD_CONTEXT.EnlightenedSwapContext]
movsxd  rax, dword ptr [rdi-4]
sub     rcx, rdi
cmp     rax, rcx
jz      short BadSwapContextHook
```

There also exists a second set of patches that PatchGuard must allow for compatibility with older processors. Very early releases of x64 processors by Intel did not implement the prefetch instruction, and so the kernel has support for detecting an illegal opcode fault on a prefetch instruction, and reacting by patching out the prefetch opcode on-the-fly. However, this sort of on-the-fly patching is not normally permitted by PatchGuard (for obvious reasons), at least not without special support. During initialization, PatchGuard generates some code that executes a prefetch operation, and then checks whether the the count of patched prefix instructions was incremented after executing the patch code. Assuming that the processor is an older model without prefetch support, then a special exemption (the "prefetch whitelist") is activated the exempts a list of RVAs from the image base from PatchGuard's checks. This list of RVAs is stored in a binary resource appended to ntoskrnl.exe (named "PREFETCHWLIST").

The code for detecting if the prefetch exemption should be enabled at boot time is as follows (the result of the check is, for Windows Server 2008 Beta 3, stored at offset 2B1 into the PatchGuard context):

```
call    KeGetPrcb
mov     ecx, 2
cmp     [rax+63Dh], cl  ; Prcb->CpuVendor
mov     [rsp+0EC8h+var_D48], rax
jnz     short SkipEnablePrefetchPatchExemption
```

```
lea     rdx, [rsi+214h] ; PrefetchRoutineCode
mov     dword ptr [rdx], 0C3090D0Fh ; prefetch [rcx] ; ret
mov     ebx, cs:KiOpPrefetchPatchCount
lea     rcx, [rsp+0EC8h+arg_18]
call    rdx
mov     ecx, cs:KiOpPrefetchPatchCount
cmp     ebx, ecx
jz      short SkipEnablePrefetchPatchExemption

mov     [rsi+2B1h], dil ; EnablePrefetchPatchExemption

SkipEnablePrefetchPatchExemption:
;
; Initialization continues ...
;
mov     eax, 100000h
```

Chapter 4

Bypass Mechanisms and Countermeasures

Like PatchGuard 2, it would be folly to state that PatchGuard 3 is invulnerable to assault by third party driver code intent on performing operations blocked by PatchGuard. There are many possible attacks for the new defenses in PatchGuard 3 (as well as several possible countermeasures that Microsoft could take in order to break the proposed bypass mechanisms in a future PatchGuard iteration). This article will describe specific attacks that are capable of defeating PatchGuard 3.

4.1 Hybrid Exception Interception and Memory Searching

As PatchGuard 3 utilizes completely randomized (self-decrypting) blocks of code and data for its constituent PatchGuard contexts in the SEH execution case, it is not generally possible to trivially locate and disable PatchGuard contexts through a non-paged pool scan. Additionally, due to PatchGuard 3's break on relying upon SEH to invoke PatchGuard in all cases, it is also not generally possible to disable PatchGuard 3 reliably via interception of the SEH dispatching code path.

While these defenses do complement one another, there still exists weaknesses that can be exploited by a third party. Specifically, when PatchGuard is running through a re-purposed DPC routine that is invoked via SEH, it is vulnerable in that the SEH dispatching code path can be intercepted to locate (and disable) PatchGuard just before it is executed. Furthermore, in the case where

PatchGuard runs without any SEH obfuscation, it is vulnerable to a memory search, as there is (necessarily) some static code placed in non-paged pool memory which makes the translation between the DPC function calling convention and the PatchGuard stage 1 decryption routine's calling convention.

By combining a memory search approach with the previously described SEH interception approach, it is possible to attack both launch vectors of PatchGuard simultaneously, with the effect of disabling it no matter which vector(s) are used in a particular boot.

However, there are still some sticking points that need to be resolved in the SEH interception case. As previously mentioned, the SEH-obfuscation-based launch vector was significantly improved over PatchGuard 2, with obfuscation of the exception information and randomization of the call stack from the point of view of the exception dispatcher logic itself. These obstacles must be overcome in order to successfully mount an attack using this approach against PatchGuard 3.

The first problem relating to the obfuscation and randomization of the exception information turns out to not be the roadblock that one might think at first glance. There are some weaknesses of the obfuscation logic that allow the true colors of the exception to show through if one is clever about examining the information available at the point of `_C_specific_handler`. Furthermore, it is also possible to hook at a lower level than `_C_specific_handler`, such as `KiGeneralProtectionFault` (easily located by examining the IDT), which would get one in before the assembly-language exception handler logic has a chance to fudge the exception information.

Although the `KiGeneralProtectionFault` vector is easier to implement in that it completely bypasses one of the new defensive mechanisms with respect to the SEH-related PatchGuard execution code path, it is again still possible to attack PatchGuard using `_C_specific_handler` by relying upon information leakage when `_C_specific_handler` is called. Specifically, all exceptions altered by PatchGuard originate within the confines of the kernel itself, all of the exceptions have two parameters (most of the "legitimate" versions of exceptions like `STATUS_INSUFFICIENT_RESOURCES` always have zero parameters, because they originate from within `RtlRaiseStatus` which never stores any exception parameters in the exception record), and somewhere in the call stack the kernel routine responsible for dispatching DPCs or timer DPCs is going to be present.

By combining these facts, it is possible to make a highly accurate determination as to whether an exception is caused by PatchGuard. The latter piece of information (checking whether the routine responsible for calling the DPC or timer DPC is in the call stack) also proves valuable when one must later counteract the second defense added to the SEH code path, that is, the randomization of the call stack.

In order to determine whether the DPC or timer DPC dispatcher is in a given

call stack, it is first necessary to locate it in the kernel image. There are some complications here. First of all, the timer DPC dispatcher routine has three call instructions that can call a timer DPC, not all of which are readily triggerable. Additionally, neither the timer DPC dispatcher or the DPC dispatcher are exported.

However, while it is not possible to simply ask for the addresses of those two routines, it is possible to find them programmatically by requesting that a DPC and a timer DPC be executed through the documented APIs for DPCs and timer DPCs. From within the DPC or timer DPC routine, it is then possible to locate the return address via the use of the `_ReturnAddress()` compiler intrinsic. This works because the return address will be guaranteed to reside within the DPC or timer DPC dispatcher. Alternatively, an assembly language routine could be written that simply examines the current pointer at [rsp] at the time of the call.

This still leaves a problem in the timer DPC dispatcher case, as there are three call instructions, and it is not easy to observe calls from all three call sites within the timer DPC dispatcher on-demand, since it is necessary to programmatically find the return points at runtime. However, once again, the very same metadata that is critical to x64 SEH support dooms PatchGuard with respect to this approach, as it is possible to go from an arbitrary instruction in the middle of any function to the start of that function, by following chained unwind metadata until an unwind metadata block is reached that has no parent[3]. This top-level unwind metadata block has a reference to the first instruction in the function. Now that it is possible to locate the start of a function from any arbitrary valid instruction location within that function, it becomes trivial to determine if two addresses reside in the same function; to do this, one must only follow the unwind metadata chain for both addresses, and then check to see whether both top-level unwind metadata blocks refer to the same function. With this technique, combined with the ability to locate at least one call site within the timer DPC dispatcher, it again becomes possible to identify the timer DPC dispatcher, as no matter which call site is used, it will be guaranteed that the call site resides within the timer DPC dispatcher routine `KiTimerExpiration`. By comparing top-level unwind metadata blocks, it becomes possible to authoritatively discern whether any arbitrary instruction resides within the timer DPC dispatcher or not.

It is also possible to bypass the alterations to the exception (and instruction pointer) addresses that KiCustomAccessHandler (the assembly-language "first chance" exception handler routines for the repurposed DPC routines) makes by performing a stack trace from the `_C_specific_handler` itself instead of relying on the context record or exception handler information. This is because the call stack is conveyed as if the faulting instruction in the repurposed DPC call stack was the site of a call to KiGeneralProtectionFault. As a result, it is possible to substitute the current context for the context presented to `_C_specific_handler` for unwind purposes. This also provides a layer of defense against Microsoft altering other registers in the exception handler context in future PatchGuard

revisions, which could cause manual unwinds to return incorrect register values, resulting in system crashes after an unwind intended to effect a hard return out of the re-purposed DPC routine.

Furthermore, by clever usage of this mechanism for determining whether an address resides within a particular function, it is also now possible to determine the real return address for any given re-purposed DPC routine. Specifically, by checking whether each address in the call stack as of `_C_specific_handler` is within either the DPC dispatcher or the timer DPC dispatcher, one can determine whether a given call frame corresponds to the call site that called the re-purposed DPC routine or not, irrespective of any random amount of bogus function calls that may be layered on top of the re-purposed DPC. This in turn defeats the remaining improvement to the SEH PatchGuard code path, as it once again becomes possible to cleanly unwind from any arbitrary point in the PatchGuard exception callstack.

Through the combination of the ability to either circumvent entirely or "see through" the deception that KiCustomAccessHandler creates over the exception information passed to `_C_specific_handler`, and the ability to recover the correct return address of a repurposed DPC routine, it now becomes possible to disable the SEH control flow path of PatchGuard 3. This leaves the remaining problem of locating the non-SEH control flow path of PatchGuard in non-paged pool memory as the last piece of the puzzle with respect to this method of disabling PatchGuard. However, locating the trampoline routine that adapts a DPC routine call to a PatchGuard stage 1 decryption stub call is trivial, as the adapter trampoline is static and contains a very recognizable signature in terms of the constants written to the beginning of the decryption stub. In order to disable the trampoline routine, it is enough to simply patch it with a "ret" instruction (effectively the same thing as the SEH bypass technique, but as implemented in code instead of a virtual unwind).

The source code to a working implementation of the hybrid exception interception and memory searching bypass technique for PatchGuard 3 is included with the article.

Although this approach is successful in disabling the current iteration of PatchGuard 3, it is not without its weaknesses. Microsoft could, for instance, disable this technique via altering the SEH-less PatchGuard DPC-to-decryption-stub adapter to not be static (i.e. randomization of the code placed into non-paged pool at runtime). There are also a number of assumptions of the SEH-based approach that could be invalidated by Microsoft in a future PatchGuard release. However, in keeping with the fact that it is possible to gain control flow at a lower level than the exception dispatcher path itself (i.e. patching KiGeneralProtectionFault), the author feels that it would be better to focus on removing relevant information before any exception handlers (assembler or C-language) are called instead of after the defining moment (in other words, the exception) occurs, as it is the exception that presents the first easily-accessible interception

point to an outside attacker.

4.2 Timer DPC Dispatcher and DPC Dispatching

Although PatchGuard 3 eliminates SEH as a single point of failure with respect to executing the system integrity checks, the timer and DPC dispatchers continue to remain attractive targets. One simple bypass mechanism is to locate the call sites in both routines (such as by recording the addresses of both dispatcher routines as described in bypass technique 1, and then performing disassembly to locate and patch all call sites. At each call site, it is possible to detect that PatchGuard is being executed by looking for either a non-canonical DeferredContext parameter value or a DeferredRoutine that resides within the non-paged pool. (In PatchGuard 3, implementing the former check alone proves sufficient, as for the ease of the implementation of PatchGuard 3, both the repurposed DPC routines and the non-SEH-based control path use compatible calling conventions, which stipulate a non-canonical obfuscated pointer value as the DeferredContext parameter.)

The main disadvantage of this approach involves inherent difficulties in performing arbitrary code patching in x64 (specifically, the large size of any code patch and the large number of now relatively common instruction-pointer-relative instructions). However, given that this is a difficulty that impacts any code patching on x64, the author feels that it should not be considered a significant problem for a determined attacker. In fact, Microsoft Research's very own Detours implements a code patching system for x64[5], illustrating that code patching on x64 in general is not a task that should be considered insurmountable by any means.

Because the timer and DPC dispatchers remain relatively unprotected targets that have not been involved in public bypass source code that has been released to date, the author would recommend bolstering the defenses of the timer and DPC dispatcher for the next PatchGuard release, as the two routines continue to represent an attractive single point of failure. Adding a third PatchGuard execution mechanism that does not involve traditional DPCs at all would be an example of one approach to eliminate the DPC dispatcher related logic as a single point of failure. It may also be possible to increase the difficulty of locating all the call sites within the DPC dispatching related code through a combination of differing static call stack differences for each of the three call sites of the timer DPC dispatcher (i.e. adding dummy function calls) combined with call stack randomization on top of static call stack differences between each of the three timer DPC dispatcher calll sites. Randomized call stacks alone would not suffice as by examining the call stacks of many iterations of timer DPC requests, it would become easy to eliminate the randomized entries

(which would not be common to all recorded call stacks) with a relatively high degree of accuracy given a large sample size. A disadvantage to taking such an approach is that it would essentially result in adding deliberately-difficult-to-maintain "spaghetti code" into yet another critical area of the operating system (timer DPC dispatcher logic). The author suspects that the maintainer of the timer DPC dispatcher code would likely not appreciate having to deal with such things.

4.3 Canceling the PatchGuard Timer(s)

As PatchGuard continues to rely upon timer DPCs for the execution of its check routines, the kernel timer DPC list itself continues to remain a relatively attractive target for attack. The timer DPC list is common to all control paths leading to PatchGuard, as timers are always used for the delayed execution component that periodically calls the check routine.

There are presently two obstacles in the way of the timer DPC list. The first of which is that altering it relies upon locating non-exported kernel variables. Although it may be possible to do so via fingerprinting, this does make the approach slightly less desirable than it might initially appear. However, fingerprinting can work if done carefully, and there are many short functions that reference the timer list in a fairly predictable fashion (e.g. KeCancelTimer). One other possible way to find the DPC list would be to create and set a timer (thus inserting it into the timer list), and then scan every 8-byte-aligned value in a non-paged uninitialized data section in ntoskrnl, treating each valid address as a linked list and searching the first several entries for the timer that was just linked into the list. While a rather ugly and bruteforce-based approach (and not entirely safe either as one would need to be relying heavily on MmIsAddressValid), scanning the ntoskrnl data sections is one alternative to fingerprinting in terms of finding the timer list.

The secondary problem with this approach is that starting with PatchGuard 2, the timer list itself is obfuscated such that the link between a KTIMER object and its corresponding KDPC is obfuscated. This obfuscation mechanism, as previously describedbackref to 1, hinges upon two additional non-exported kernel variables (KiWaitAlways, KiWaitNever) that act as obfuscation keys. Locating these variables would be likely entail code analysis or fingerprinting of (possibly exported) routines that need to insert a timer into the timer list, such as KeSetTimerEx.

Another alternative approach that dispenses with fingerprinting and/or bruteforce-based approaches altogether, at the expense of requriring added complexity (a user mode component), would be to postpone the activation of any driver code that would run afoul of PatchGuard until after Win32 in user mode has been started. A user mode service could then be created that would download the

symbols for the kernel binary in use, retrieve the addresses of KiTimerTableListHead (the timer list), KiWaitNever and KiWaitAlways, and pass these addresses on to the driver via any standard user mode to kernel mode communication mechanism (such as DeviceIoControl). Because the kernel debugger relies on the ability to retrieve these variables by name via the PDB symbols for the !kdexts.timer extension, Microsoft would not be able to block this approach by removing or renaming the obfuscation key variables without imparing the functionality of existing debugger binaries.

Once one has located the KiTimerTableListHead, KiWaitAlways, and KiWaitNever, it is a fairly simple (if perhaps unsafe without synchronization, though one could always take the "sledgehammer" approach and stop all but one CPU and raise IRQL to `HIGH_LEVEL`) to traverse the timer list, deobfuscate the DPC link on each corresponding timer object, and from there check each timer to see whether it bears the characteristics of being a PatchGuard timer (which may include attributes like a timer interval several minutes into the future, a non-canonical DeferredContext value, and possibly a DPC routine pointer into non-paged pool). After one has located the timer in question, it can be easily disabled (either removing it from the list entirely, such as via KeCancelTimer, or by rewriting the DPC routine to point to an empty function that simply returns without performing any operation.

Because Microsoft has functionality in the debugger that depends on the ability to use these variables to access the timer list, they have unfortunately backed themselves into something of a corner with respect to current operating system versions, as it is generally Microsoft's policy that existing debugger binaries continue to function properly after hotfixes or service pack to a particular already-released operating system version. The best ways to counteract this approach would be to make it more difficult to pick out the PatchGuard DPC in-memory with respect to all of the other timer DPC objects that are in the list at any given time for a typical system, and to create additional launch vectors for PatchGuard that do not depend so heavily on the timer list. There exist a number of other ways to execute code without drawing the attention of someone that does not know what they are looking, many of which are less obvious than a timer.

4.4 Page-Table Swap

Like all memory accesses in the Windows kernel, PatchGuard's system integrity check routine operates in protected mode with paging enabled. It may theoretically be possible to take advantage of this fact to hide kernel patches from PatchGuard.

The proposed bypass technique would involve patching the first instruction in the timer and DPC dispatchers to branch to third party code. When a DPCs

and timer DPCs are about to be considered for execution, as signaled by a call to one of the two dispatcher routines, a shadow copy of the page tables is created. This shadow copy is configured to be identical to the normal page table for the current process, except that the page table entries for any kernel code pages that have been patched are altered to refer to physical pages that are representative of the original state. The return address of the DPC or timer DPC dispatcher on the stack is swapped with a pointer into driver-supplied code, and cr3 is reconfigured to point to the shadow page table. Then, execution is transferred back to the timer or DPC dispatcher entrypoint (which no longer shows any signs of patching due to the page table swap), and DPCs are dispatched. When the dispatcher is finished with its work, which would include invoking PatchGuard if PatchGuard is to be executed in any batched timer DPCs, then control is returned to driver-supplied code, which then mirrors any page table modifications since the shadow copy was made back to the actual page table for the process, and cr3 is returned to its original value. Control is then transferred to the normal return point of the dispatcher.

This approach does not involve disabling PatchGuard at all. Instead, it describes a potential way to "peacefully coexist" with it, so long as only kernel code patches are being done. (Data pages, which could be expected to be modified by a DPC, are considered by the author to be much less practical to protect from PatchGuard in this fashion.) Because the DPC and timer DPC dispatcher logic executes at IRQL `DISPATCH_LEVEL`, thread context switching is disabled for the current thread, making the cr3 swap approach relatively feasible.

Because this approach does not involve attacking PatchGuard directly, it automatically circumvents all of the myriad defensive mechanisms built into PatchGuard in current releases, making it a fairly attractive potential avenue of attack. However, there are some downsides. Among other things, the synchronization required to pull a page tabpe swap off in a multiprocessor environment are likely to be complex and difficult to safely duplicate if one allows DPC routines to perform operations that alter PTEs. Additionally, there would be a performance impact incurred by this approach as it would need to run continuously in a relatively high-impact path (DPC dispatching) throughout system lifetime. The performance implications of invalidating TLBs on every DPC batch may be problematic in some circumstances (swapping cr3 automatically clears out TLBs).

Another disadvantage of this approach is that by virtue of the fact that all DPCs (and potentially all device hardware interrupts) may run with the shadow copy of the page table, most hardware-related events will not be subject to kernel code patches hidden by this mechanism. This may or may not be a problem depending on what the goal of the desired kernel patching is.

Microsoft could counteract this approach by making a copy of all PTEs that describe the kernel at PatchGuard initialization time and then validating all kernel code PTEs from within the PatchGuard check routine. Additionally, if

Microsoft could make the assumption that PatchGuard always executes in the system process, another approach could be to require that cr3 take on a known value.

4.5 DPC Exception Handler Patching

One of the changes introduced in PatchGuard 3 over PatchGuard 2 was a slight change to the protocol used to invoke the first stage of the decryption process. Specifically, all callers of an encrypted PatchGuard context now include a static 8-byte string (of instruction opcodes) that is xor'd with a value at the start of the PatchGuard context to form the initial decryption key.

The reasons for making this change over the original behavior are unclear to the author, but it unfortunately represents an easy target for disabling PatchGuard, as the string itself (0x8513148113148F0) is fairly unique and unlikely to appear outside of PatchGuard in terms of kernel code. Furthermore, all PatchGuard callers, including all ten of the repurposed DPC routine exception handlers and the non-paged pool memory DPC adapter (if used) reference the string with no obfuscation to speak of. This presents an extremely easy, fingerprint-based approach to disabling PatchGuard. By scanning non-paged pool space for this string, as well as kernel code regions, it is trivially easy to locate an instruction in the middle of the every single code path responsible for invoking PatchGuard's check context.

After the instructions referencing the 8-byte string have been located, it is trivial to patch them to execute an unwind out of the exception handler logic (or in the case of the non-paged pool memory code, simply return directly). Such an attack prevents PatchGuard from ever starting, and furthermore has the advantage of a minimum of additional supporting logic required (when compared to many of the other bypass techniques outlined in this article).

It would be trivial for Microsoft to disable this technique. The recommendation of the author would be to get rid of the static 8-byte string referenced in every PatchGuard caller. Ironically, PatchGuard 2 necessarily has a similar 4-byte string (which is also still used in PatchGuard 3), representing the initial instruction of the first stage decryption stub. Unlike with PatchGuard 3, however, PatchGuard 2 takes care to obfuscate the process of writing the opcode string out to the PatchGuard context, so that one cannot simply use a single blanket fingerprint to cover all cases. The change made in PatchGuard 3 completely blows this work out of the water, so to speak, and it has the added advantage of being twice as large as a value to fingerprint as well.

4.6 System Call MSR Swap

A variation on the technique described in backref:4, it should theoretically be possible to swap the system call MSRs (or in fact several other processor control registers that are protected by PatchGuard) for the duration of DPC or timer DPC dispatching online, with the "tainted" values being restored after the dispatcher returns. The system call MSRs are responsible for designating the address of the system call dispatcher, and are thus an attractive target for third parties that would like to perform system call hooking.

The same basic concepts would be applied to this technique as previously described in the cr3 swap technique. If system calls are the only desired targets to hook, then the cr3 swap can be eliminated as unnecessary for single processor systems (as it would be safe to make and restore changes to the actual underlying physical pages before and after a DPC dispatcher call, using the return address on the stack as a way to return to the altered location without leaving opcodes patched in the kernel across dispatcher invocations). For multiprocessor systems, some mechanism would need to be developed to allow the MSR swap to be made across DPC dispatchers while preventing code patches from becoming visible to a second processor. This is necessary because there could be more than one PatchGuard context executing simultaneously with the PatchGuard 3 addition of a probability to initialize a second check context at system boot time.

In order to block such a technique, Microsoft would likely be best served by making it difficult to locate all the regions necessary to patch in order to maintain the deception of an unpatched system across PatchGuard checks. The principal way to do this would be to create other, alternative launch vectors for PatchGuard that are unrelated to DPCs and, preferably, do not involve exported APIs that are easy to intercept from a third party perspective.

Chapter 5

Conclusion

Although PatchGuard 3 does bring some pointed counter-attacks to many previously disclosed bypass techniques, version 3, like its predecessors, is hardly immune to being either disabled completely or simply co-existed with. It is likely that future revisions to PatchGuard will continue to be vulnerable to a variety of bypass techniques, though it is certain within Microsoft's reach to counter many of the publicly disclosed bypass vectors. It is anticipated by the author that until PatchGuard can be implemented with hardware support, such as via a combination of trusted boot (TPM) and a permanent hypervisor, future revisions will continue to be vulnerable to attack from determined individuals.

On the other hand, Microsoft's efforts with PatchGuard appear to have paid off so far in terms of preventing a mass-uptake of PatchGuard-violating drivers on Windows x64. In other words, a case could be made that Microsoft doesn't need to be perfect with PatchGuard, only "good enough" to give vendors cold feet about trying to ship products that bypass it. Only time will tell if this continues to remain the case into the future, however.

Bibliography

[1] Skywing. *Subverting PatchGuard version 2*. http://www.uninformed.org/?v=6&a=1&t=sumry; accessed September 16, 2007

[2] Skywing. *Programming against the x64 exception handling support, part 7: Putting it all together, or building a stack walking routine*. http://www.nynaeve.net/?p=113; accessed September 16, 2007

[3] skape. *Improved Automated Analysis of Windows x64 Binaries*. http://uninformed.org/index.cgi?v=4&a=1&t=sumry; accessed September 16, 2007

[4] skape, Skywing. *Bypassing PatchGuard on Windows x64*. http://uninformed.org/index.cgi?v=3&a=3&t=sumry; accessed September 16, 2007

[5] Microsoft. *Detours*. http://research.microsoft.com/sn/detours/; accessed September 16, 2007