Report

Defeating PatchGuard

Bypassing Kernel Security Patch Protection in Microsoft Windows

By Deepak Gupta, McAfee Labs,
and Xiaoning Li, Intel Labs

Table of Contents

Summary

The kernel forms the core of any operating system. In conjunction with device drivers, the kernel abstracts interfaces for processes, memory management, file system, networking, and other services used by application developers. The kernel and other device drivers run at ring 0, the highest privilege, and form the bottom of the stack. Attacking the kernel and drivers puts an attacker in an advantageous position and helps hide footprints (rootkit activity). This stealth is required because most antimalware scanners update very frequently. If malware leaves behind footprints, then it can be traced, contained, and easily caught. Thus kernel-level malware with rootkit abilities are a very high-risk category.

To protect the 64-bit Windows kernel, Microsoft created Kernel Patch Protection, commonly called PatchGuard. We haven't seen many attacks on the 64-bit kernel barring some incidents of TDL4/Alureon and Xpaj. (These are actually "bootkit" attacks against the hard drive's master boot record that can be prevented or cleaned later.) We know of no attack in the wild that targets PatchGuard and then patches the kernel image or critical kernel data structures. However, independent research, including our own, has proved that it is possible to defeat PatchGuard. These "white hat" attacks were published with proof of concept code and are purely for educational purposes. However, just as we have seen with earlier versions of Windows, malware developers will eventually find a way to crack the operating system's defenses.

Unlike 32-bit x86 processors, 64-bit processors from Intel come with virtualization extensions that can be used to set memory and CPU-register protections at the hardware level. DeepSAFE technology is one such offering from the collaborative efforts of Intel and McAfee; it will be instrumental in staying one step ahead of malware authors.

Introduction

Given the popularity of Microsoft Windows with home and business users, it's no surprise that Windows is also popular with malware authors. The operating system has been under constant attack for many years, and the company is diligent in creating patches and countermeasures. However, most of these measures block attack vectors and aren't effective once the system has already been compromised.

One common way of compromising a system is by patching kernel memory, including function hooks or kernel object manipulations. The lack of documentation or support from Microsoft forced even legitimate software developers to use these techniques on 32-bit Windows. Microsoft couldn't effectively stop this type of compromise on its 32-bit platforms because it is very hard to differentiate between legitimate software and malware code patching kernel memory.

With the development of 64-bit Windows, Microsoft decided to guard the kernel and critical areas in kernel memory against patching. All versions of 64-bit Windows come with a component called Kernel Patch Protection or, for short, PatchGuard. PatchGuard runs at random times and checks the integrity of several kernel components in memory. If it finds an inconsistency in any of them, it bug-checks the system (with the "Blue Screen of Death") using bug-check code 0x109 to report that the system's integrity has been compromised.

PatchGuard is also another piece of code running in the Windows kernel with CPU privilege level 0—just as the other kernel drivers run. Thus it is possible for a third-party kernel module to disable PatchGuard or to fool it. To protect PatchGuard against those attacks, Microsoft has employed many antihacking tricks, including self-modified code, code obfuscation, self-integrity checking, and randomization to make PatchGuard difficult to locate and analyze. In spite of these measures, some publications have already discussed how early versions of PatchGuard operate and can be disabled.[1, 2, 3]

In this paper we will briefly examine major previously published attacks and their countermeasures from Microsoft. All of these attacks are difficult to implement and are usually ineffective after the release of subsequent hotfixes or service packs.

We will present an attack that can disable PatchGuard across different versions, including Windows 7 and its current service packs and hotfixes, as well as the Windows 8 preview we have seen. With our attack we will disable PatchGuard and hide a process by the inline hooking of the kernel's NtQuerySystemInformation system service.

The fundamental point we wish to make in this exercise is that it is impossible to fully secure a system against compromise solely by working within the system. Effective security solutions must detect system modifications at the moment when they are being modified—unlike PatchGuard, which determines system modifications at a later time and is vulnerable to being disabled in this interval.

Kernel Patching

The kernel is the piece of software that forms the core of the operating system; every other component uses services of the kernel. The kernel runs at CPU privilege level 0, also called Ring 0, the highest level, at which every privileged instruction execution is allowed (with the exception of 64-bit systems running in virtualization mode). These privileged instructions include loading the Global Descriptor Table Register with the correct table, loading the Interrupt Descriptor Table Register with the correct table, setting up the memory management unit, and preparing page tables for virtual addresses and process protection.

CPU instructions required to perform these operations are privileged and cannot be executed at a privilege higher than 0. Thus the CPU hardware helps create a boundary between user-mode applications, which run at privilege level 3, also called Ring 3, and the kernel. This protects the kernel from rogue and malicious applications.

The following schematic diagram of the Windows NT architecture shows user-mode applications (1, 2, and 3) in Ring 3. The applications make use of dynamic link libraries (DLLs) such as kernel32.dll, user32.dll, advapi32.dll, etc. (not shown). All DLLs eventually call into ntdll.dll, a system-provided DLL that forms the native subsystem of Windows. In simple terms a subsystem is an application programming interface. Windows supports Win32, POSIX, and OS2 interfaces; all call into the native subsystem, which abstracts the services provided by the kernel and provides the system-call interface. The native subsystem makes the transition from user mode to kernel mode using CPU-supported mechanisms: software interrupts or sysenter instructions.

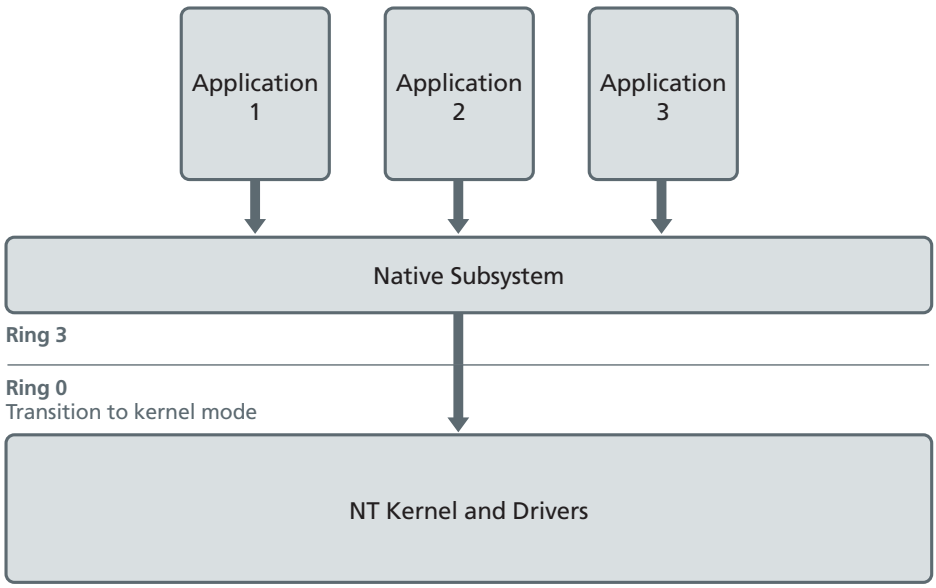

Figure 1. The Windows NT architecture.

The kernel on its own can't provide support for the various devices on a system's motherboard. Thus we have device drivers, which make use of services provided by the kernel and the hardware abstraction layer (HAL). Device drivers also run at CPU privilege level 0, the same level as the kernel. Any piece of code that runs at level 0 can be called kernel-mode code.

Since its initial days, the various interfaces of Windows NT were not well documented, and there were a lack of services from the operating system to support the filtering of the file system, disk input-output, network traffic, registry, and so on. This oversight forced third-party vendors to reverse-engineer certain kernel-mode components—including the kernel itself and the Network Driver Interface Specification (NDIS) library of APIs—and to use undocumented methods—including hooking system service calls by replacing the function pointers with their own implementation of the function—in effect making their own products. These mechanisms became so prevalent that Microsoft eventually documented various required interfaces, providing the filtering infrastructure.

You knew there was a catch to all this. If legitimate developers could reverse-engineer Windows NT to make their software work, then so could malicious developers. Malware authors also started implementing kernel hooks to hide their objects from process, file system, and registry views. This gave them a powerful defense against antimalware vendors. It raised the bar for the security industry, which now had to scan key kernel memory areas for modifications.

Microsoft long maintained that patching the kernel was not supported, but they never forced the issue because there were already many successful, legitimate products that did so. Cracking down on them would have disturbed Microsoft's ecosystem and business. However, when the company rolled out its 64-bit operating system, it decided to not allow modifications of key kernel-mode modules (NTOS, HAL, and NDIS), critical kernel-mode data structures (System Service Dispatch Table), and critical model-specific registers (MSRs).

PatchGuard

Here's where PatchGuard enters the picture. Kernel Patch Protection (PatchGuard) is the component of the 64-bit Windows kernel that is responsible for protecting these key areas.

PatchGuard protects following components:

- System modules (NTOS, NDIS, HAL)
- System Service Dispatch Table
- Global Descriptor Table
- Interrupt Descriptor Table

PatchGuard is probably the first kernel component in any operating system that validates kernel image, memory, and MSRs; and checks whether the system is compromised. To prevent third-party kernel modules disabling Kernel Patch Protection, its initialization and modes of operation are highly obfuscated and randomized; it's a frustrating job trying to analyze PatchGuard.

No doubt this one decision has significantly raised the bar for attackers and has reduced the number of kernel rootkits on 64-bit Windows. But for how long?

Initialization and operations

We will not go deeply into the details of how PatchGuard is initialized and validates the kernel images, data structures, and key MSRs. A number of very good articles, including those by skywing cited in endnotes 1 through 3, have already dissected PatchGuard's operations.

Initialization

PatchGuard's initialization is very obfuscated and uses a lot of misdirections to confuse the curious. It invokes the innocent-looking function KiDivide6432 at system startup.

```
if (KiDivide6432(KiTestDividend, 0xCB5FA3) != 0x5EE0B7E5)
{
 KeBugCheck(UNSUPPORTED_PROCESSOR);
}
```

KiTestDividend is hardcoded to 0x004B5FA3A053724C, as we learn with a remote kernel debugger. Analyzing with LiveKd, we find its value is 0x014B5FA3A053724C.

If we look carefully, we see that KiTestDividend's higher-order byte is "unioned" with KdDebuggerNotPresent. If the value of KdDebuggerNotPresent is 1, it will generate a divide error because the value of KiTestDividend will be high enough to cause a quotient overflow.

When the debugger is present, we get a value well under 32 bits, namely 0x004B5FA3A053724C/0xCB5FA3 = 0x5EE0B7E5. However, 0x014B5FA3A053724C/0xCB5FA3 = 1A11F49AE is a quotient overflow and hence a divide-error exception. Per code, it should give us KeBugCheck with the UNSUPPORTED_PROCESSOR bugcheck code. But this never happens.

The reason lies in KiDivideErrorFault, the error handler for this exception, which checks for the instruction pointer. If it comes from KiDivide6432, KiDivideErrorFault starts the process of initializing PatchGuard. At this point, the randomized pool tag and randomized size of bytes are selected and memory is allocated. PatchGuard code and context are then encrypted with a random key (used for validation of constructs it is protecting) and is copied into this memory.

Early versions of PatchGuard created only one context, but later versions can have more than one context independent of each other and executing in parallel to validate system integrity—making it harder to disable PatchGuard.

Operations

Once the encrypted code and context are copied into the kernel heap, PatchGuard decrypts itself, validates, and reports inconsistencies if found. PatchGuard is not invoked directly and involves multiple steps of obfuscation and redirection that finally lead to the validation routine. Here are the basic steps:

- Initialize NTOS Timer Construct (allocated in PatchGuard context) with KeInitializeTimer. Select a random time value, select a random kernel's Deferred Procedure Call (DPC) function from a list of the kernel's DPC functions, call KeSetTimer, and time the execution of this DPC with a DeferredContext value as a completely invalid pointer value (by XORing the Context Pointer value with a random key).
- DPC routines have nothing to do PatchGuard, but they will dereference DeferredContext and, being an invalid pointer, the routine will raise a General Protection Fault (one more redirection). From here the execution will pass to structured exception handlers (SEHs) of these selected DPC routines. (See footnote for exception handling on 64-bit Windows.[4])
- The exception handlers of these DPC routines will invoke the PatchGuard validation routine and report if any inconsistency is found. The stack is cleared before calling into its own KeBugCheckEx function copy to hide the stack from analysis.

Newer versions of PatchGuard have a variation in which a DPC routine can invoke a validation routine instead of first going through an SEH. Microsoft took this step to defeat earlier SEH-hooking attacks.

This short schematic diagram shows the initialization and operation of PatchGuard. The diagram describes the general architecture of PatchGuard and may vary between versions.

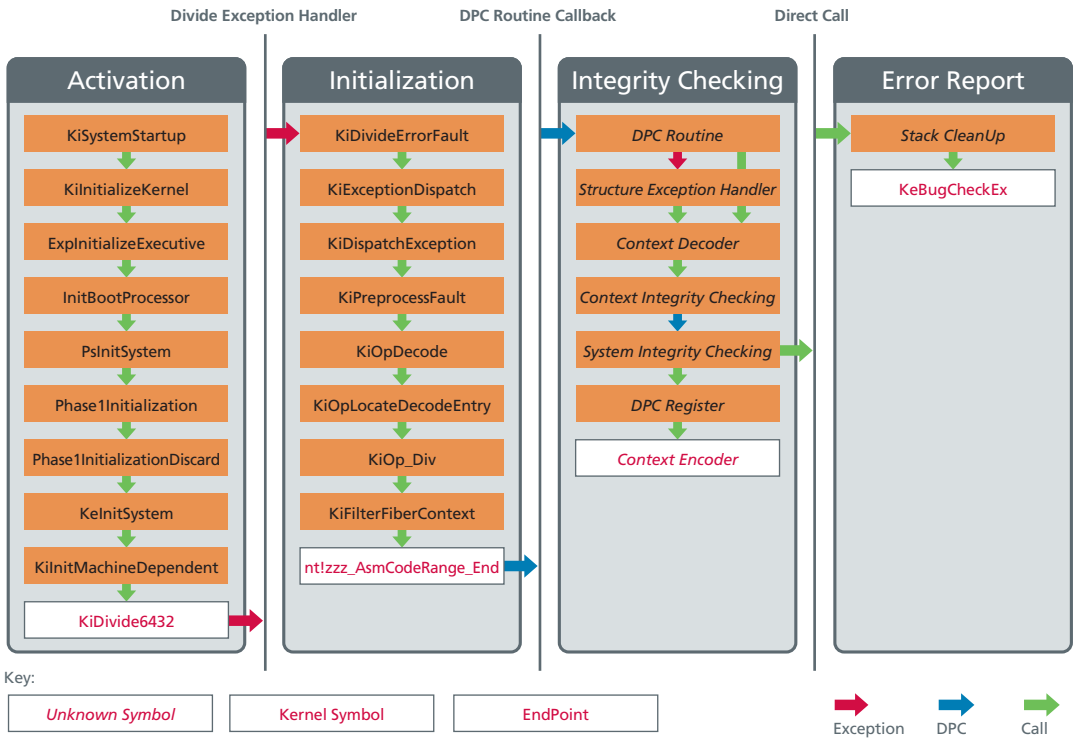

Figure 2. Schematic diagram for PatchGuard initialization and operations.

Attacks and countermeasures

PatchGuard is clearly a well-architected piece of code. Locating its footprints and presence in memory is a challenging and frustrating task, coupled with the fact PatchGuard is disabled if debugging is enabled.

However, because PatchGuard's and an attacker's code could run at the same privilege level, it is possible for malware to launch an attack and bypass PatchGuard's validation mechanisms. There are several places where an attack can be launched.

Next we'll list some of the notable bypasses to disable PatchGuard that have already been published. We will also discuss Microsoft's countermeasures in subsequent releases of the 64-bit kernel.

Exception handler hooking

PatchGuard's validation routine comes into action through exception handlers that are raised by certain DPC routines; this feature gives attackers an easy way to disable PatchGuard.

Exception handling on 64-bit Windows has been entirely redesigned: Exception handlers associated with routines are statically determined by compilers that keep the information in a separate section of the binary.

This approach requires building the list of DPC routines as each exception is raised, and locating their exception handlers from the separate section. Once the address of the exception handler for the corresponding DPC routine is located in the kernel image, an attacker can patch that address to jump on the attacker-implemented handler, which can return an "exception handled" status code. Thus PatchGuard's validation routine never runs and is disabled forever.

This method of attack has three primary drawbacks:

1. The addition of new DPC routines to invoke PatchGuard's validation. The list of DPC routines has already been increased from three (in the first version of PatchGuard) to 11 (in the latest version).

   DPC routines:

   - `KiTimerDispatch`
   - `PopThermalZoneDpc`
   - `KiScanReadyQueues`
   - `IopIrpStackProfilerTimer`
   - `IopTimerDispatch`
   - `ExpTimeZoneDpcRoutine`
   - `ExpTimeRefreshDpcRoutine`
   - `CmpLazyFlushDpcRoutine`
   - `CmpEnableLazyFlushDpcRoutine`
   - `ExpTimerDpcRoutine`
   - `ExpCenturyDpcRoutine`

2. Directly invoking the validation routine for the DPC routine. This does not involve an exception handler and has already been implemented in the latest versions of PatchGuard.

   - `KiTimerDispatch`: This new DPC routine was specially made to invoke the validation routine without using exception handling. It defeats attacks that depend on SEH.

3. Invoking the validation routine through some other mechanism than a DPC routine.

Hooking KeBugCheckEx

System compromises are reported through the KeBugCheckEx routine (BugCheck code 0x109); this is an exported function. PatchGuard clears the stack so there is no return point once one enters KeBugCheckEx, though there is a catch. One can easily resume the thread using the standard threadstartup function of the kernel.

This approach has been published and well explained by skywing.[1] Recent versions of PatchGuard countered this attack by making a copy of KeBugCheckEx at PatchGuard's initialization and overwriting memory with the saved code before calling it, thus defeating this hook attack.

Debug register attack with general detect bit on

This attack is based on hardware support for data breakpoints from x86- and 64-bit-processor architecture. The processor has eight debug registers—DR0, DR1, DR2, DR3, DR4, DR5, DR6, and DR7—which can be used to generate exceptions (#Debug Exception, INT1) whenever a particular linear address has been accessed for read, write, or execute access.[5] DR0 through DR3 can hold the linear addresses on which the data breakpoint has to be set up. DR4 and DR5 are reserved. DR6 determines which condition has generated this debug exception and can be examined by the exception handler. DR7, the configuration register, contains information about active data breakpoints (DR0–DR3) and accesses (read, write, or execute).

The debug register attack is based on the simple concept that data reads on compromised addresses (the attack locations) can be faked by setting up "read" data breakpoints on those addresses and returning the original values to the memory instruction that accessed them. But that's not enough: faking those addresses will still require an attacker to hook somewhere in the debug exception chain and fake the addresses. But hooking in the debug exception chain will require the hook to modify memory on kernel pages and thus PatchGuard will catch the attack. The solution for attackers is simple. They must also put a read breakpoint on this hooked address. Whenever read access occurs on a hooked address, it will generate a debug exception (#INT1) and will eventually invoke the handler (this time with execute access).

When Windows generates a debug exception (#INT1), it eventually passes control to a function whose address is stored at the KiDebugPointer symbol of the kernel. When the kernel debugger connects to system, KiDebugPointer contains either the address of the function KdpTrap or it contains the address of the function KdpStub. An attack can be implemented following these steps:

- Placing an inline hook on the function whose address is stored at the KiDebugPointer symbol
- Putting a data read breakpoint on inline hooked bytes
- Putting a data read breakpoint on the intended target addresses whose memory contents the attacker wants to fake
- Writing a hook handler that will determine if an exception handler has been called due to the read breakpoints the attacker has set. If yes, then just return the original value for those memory addresses.

However, there is a catch. Other kernel software (such as the kernel debugger) can modify the DR7 register, thus clearing all data breakpoints set by the attacker and disabling the attack. To overcome this defense, an attacker can use the general detect (GD) bit, which resides in the DR7 register. If GD is enabled, it generates a debug exception (#INT 1) whenever an instruction to access (read/write) the debug register is about to execute. The attacker can enable this bit and, using the hook handler implemented (in the last bullet above), increment the instruction pointer, reset the bit (which the processor clears during debug access), and resume the execution.

We have implemented this attack and find it works on PatchGuard versions that came with the Windows Vista 64-bit kernel; however, Microsoft has already defeated this attack using a smart technique. It disables this threat by pointing the Interrupt Descriptor Table (IDT) Register to its own table and thus its own implementation of the debug exception handler, and clearing the DR7 register. If GD is enabled, it will generate a debug exception, and the processor will clear the GD bit and transfer control to PatchGuard's debug exception handler. PatchGuard restores the original IDT into the register just after this instruction.

The next image shows the code excerpt from PatchGuard's routine that performs the trick to clear the contents of the DR7 register. The sidt instruction stores the contents of the IDT Register and then loads it with a new table using the lidt instruction. The routine next moves the contents of Register r9 (which is 0, not shown in the image) to DR7 using mov dr7, r9. If the GD bit is not set, it will simply zero the DR7 register. If the GD bit is set, this instruction will generate a debug exception and call the debug exception handler of PatchGuard's IDT. The processor will clear the GD bit of DR7 and PatchGuard's debug exception handler will not set it again. Upon returning from this exception, the execution will resume and the processor will move the contents of Register r9 to DR7. The next step is to restore the IDT Register using the lidt instruction once again.

```
INITKDBG:000000014O1A44F0 FA                                        cli
INITKDBG:000000014O1A44F1 0F 01 8D E0 00 00 00                      sidt    fword ptr [rbp+0E0h]
INITKDBG:000000014O1A44F8 0F 01 9D 88 00 00 00                      lidt    fword ptr [rbp+88h]
INITKDBG:000000014O1A44FF 41 0F 23 F9                               mov     dr7, r9
INITKDBG:000000014O1A4503 0F 01 9D E0 00 00 00                      lidt    fword ptr [rbp+0E0h]
INITKDBG:000000014O1A450A FB                                        sti
```

Figure 3. PatchGuard instructions changing the Interrupt Descriptor Table Register.

The debug register attack no longer works against the latest versions of PatchGuard.

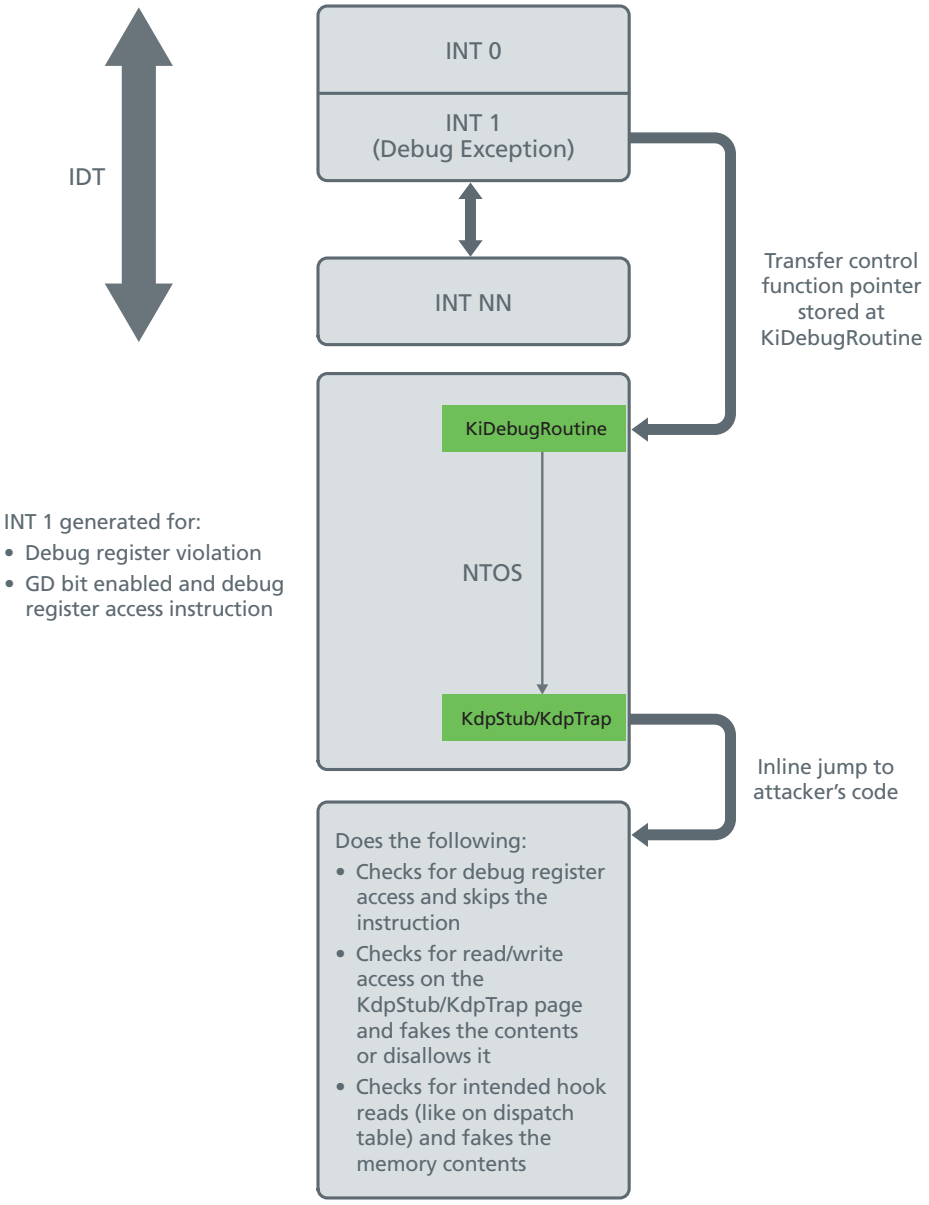

Figure 4. The implementation of the debug register hook handler.

Translation cache attack

Intel x86-based processors have translation lookaside buffers (TLBs), which help speed the translation of virtual addresses to physical addresses. TLBs associate page directory and page table entry values and other page attributes with their corresponding virtual address pages. TLBs speed translations because the processor doesn't have to go through the page-translation process for each byte it accesses in a physical page.

Almost all recent x86 processors have two kinds of caches for optimization: one for the instruction stream, and one for data accesses. An attacker can start an attack by providing different views for instruction-stream cache accesses and data accesses. The attack will need to hook into the page fault handler to always keep the modified code page entry in the TLB.[6] However, the attack cannot depend on hiding the hook on the page fault handler, thus exposing itself to being caught by PatchGuard's validation routine. Still, an attack can bypass this obstacle by putting a data read breakpoint on the first few bytes of the page fault handler and faking the memory contents during reads.

This attack suffers from same disadvantages we discussed in previous attacks and will not work against the latest versions of PatchGuard.

Patching the kernel timer DPC dispatcher

Let's look at another attack cited by skywing. By design, PatchGuard's validation routine relies on the kernel timer's dispatcher to kick in and dispatch the deferred procedure call (DPC) associated with the timer. Thus an obvious target for attackers is to patch the kernel timer's DPC dispatcher code to call their own code.

This patch is easy to implement. Attackers must only queue their own timers and associate their DPCs through valid documented interfaces provided by the Windows kernel. In its own DPC, an attacker's code can easily walk the stack and determine the location from which a call has been made and hence locate where to patch in the malicious kernel timer's DPC dispatcher. An attacker's handler can then filter the DPC calls and allow them to go through or simply return. The code can filter based on the following heuristics:

- The targeted DPC routine lies within kernel range limits
- The DeferredContext argument is a noncanonical kernel address

A generic attack

PatchGuard can encrypt its data structures and code in memory and decrypt them on the fly. That should make it next to impossible to search for PatchGuard's context and validation routine in memory, right? But if we look at how the validation routine is executed and how it implements self-decryption, we realize that PatchGuard is not as guarded as it seems.

Let's look at how this two-phase self-decryption works in memory. The first phase of the decryption routine is called directly from the exception handler of the DPC routine or directly from the DPC routine. It decrypts itself in 8-byte chunks. (Each instruction is 4 bytes.)

The first instruction is "lock xor qword ptr [rcx], rdx" and this is the only plain text; all that follows is junk. This qword is present at the address pointed to by rcx and is xor-ed with the value in rdx. Before calling this routine, rcx is set to the address of this routine; thus it points to the instruction pointer when called. It converts the next instruction to "xor qword ptr [rcx+8], rdx," which is again a 4-byte instruction. This pattern repeats until [rcx+24]. After the first decryption loop, the code looks like the following:

```
Offset  Code
0       lock   xor qword ptr [rcx],rdx
4       xor    qword ptr [rcx+8],rdx
8       xor    qword ptr [rcx+10h],rdx
C       xor    qword ptr [rcx+18h],rdx
10      xor    qword ptr [rcx+20h],rdx
14      xor    qword ptr [rcx+28h],rdx
18      xor    qword ptr [rcx+30h],rdx
1C      xor    qword ptr [rcx+38h],rdx
20      xor    qword ptr [rcx+40h],rdx
24      xor    qword ptr [rcx+48h],rdx
28      xor    dword ptr [rcx], edx        [This is for decrypting back
                                           to the first 4 bytes]
```

In effect, this code decrypts 50 hex bytes starting from offset 0, 0x2C bytes of which is decryption code itself. The rest of the 0x24 code byte forms another decryption loop (the second phase) that decrypts the rest of the validation routine and then transfers control to it.

Our attack will concentrate on just the first decryption phase of PatchGuard. We can see now that the rdx value contains the XOR key to decrypt the code. Also because of the XORing pattern, code bytes in chunks of 4 bytes (except the last byte) starting at offset 8 will repeat themselves until the 20th byte.

XORing the first 8 bytes with next 8 bytes will generate the random key (present in rdx) for us. An attacker can XOR return opcode (RET, 0xC3) with the random key and overwrite this code blob. It will simply return and disable PatchGuard once and for all. We have implemented this attack successfully and have been able to place an inline hook on NtQuerySystemInformation to hide the processes.

Below are some screenshots of attack.

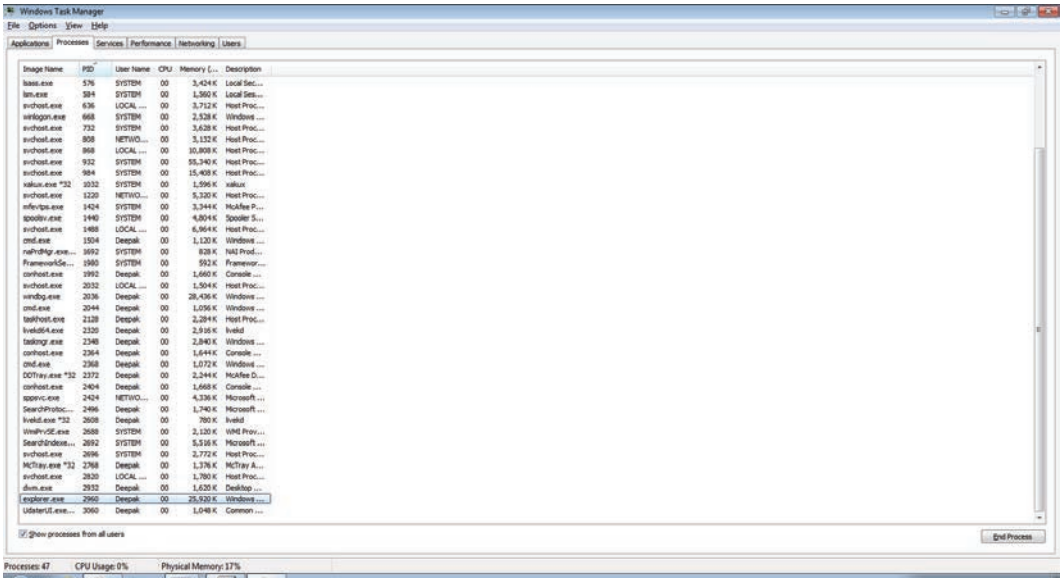

Figure 5. Sorted by Process ID, Task Manager shows explorer.exe (PID=2960).

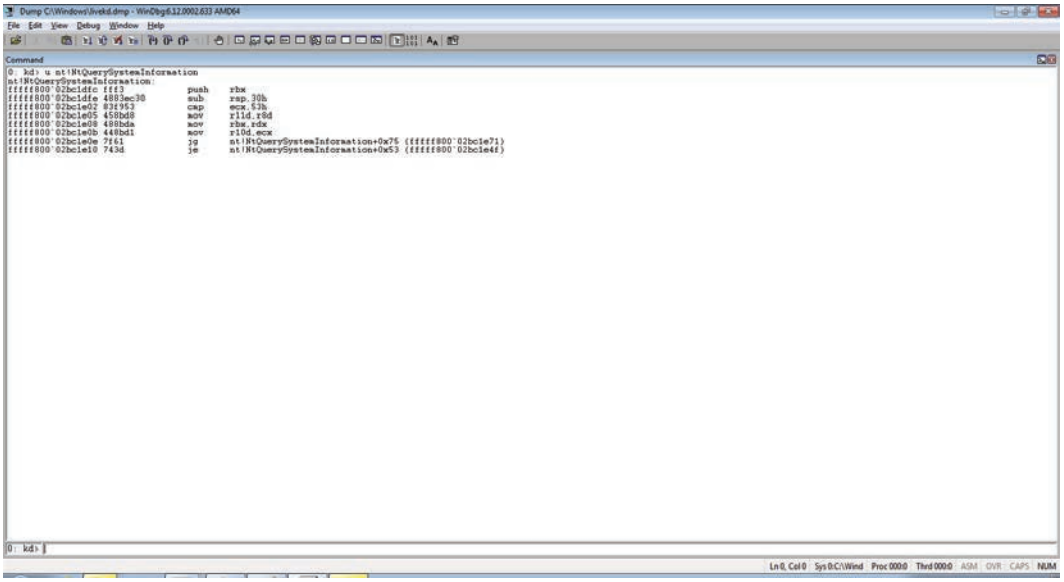

Figure 6. A view of nt!NtQuerySystemInformation before the attack.

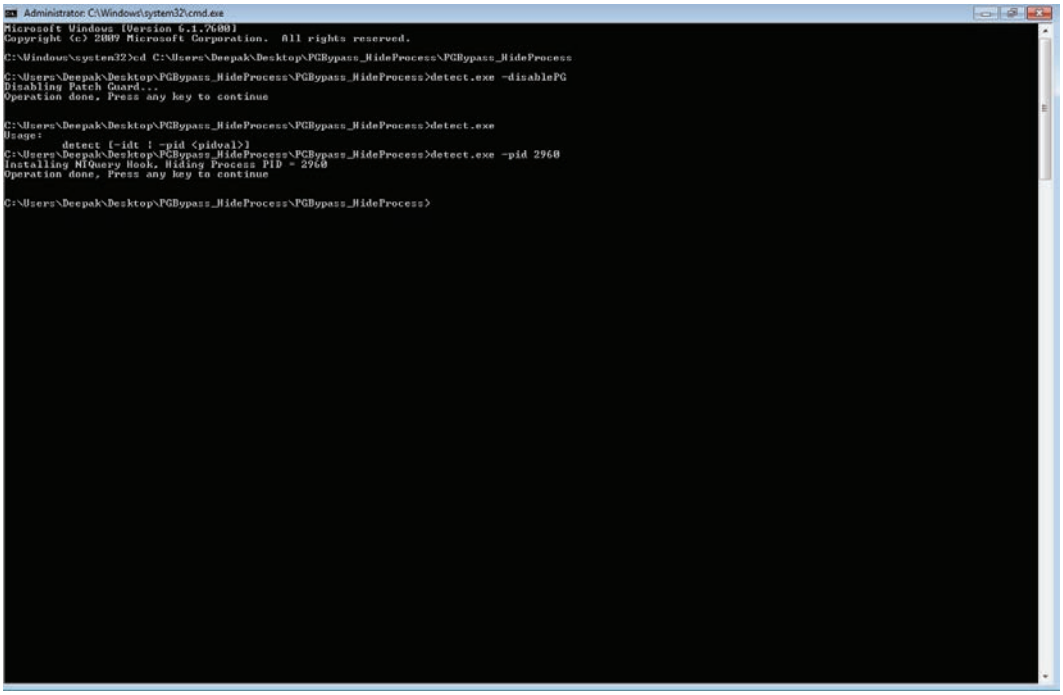

Figure 7. Disabling PatchGuard and hiding the explorer process by hooking the NtQuerySystemInformation routine.

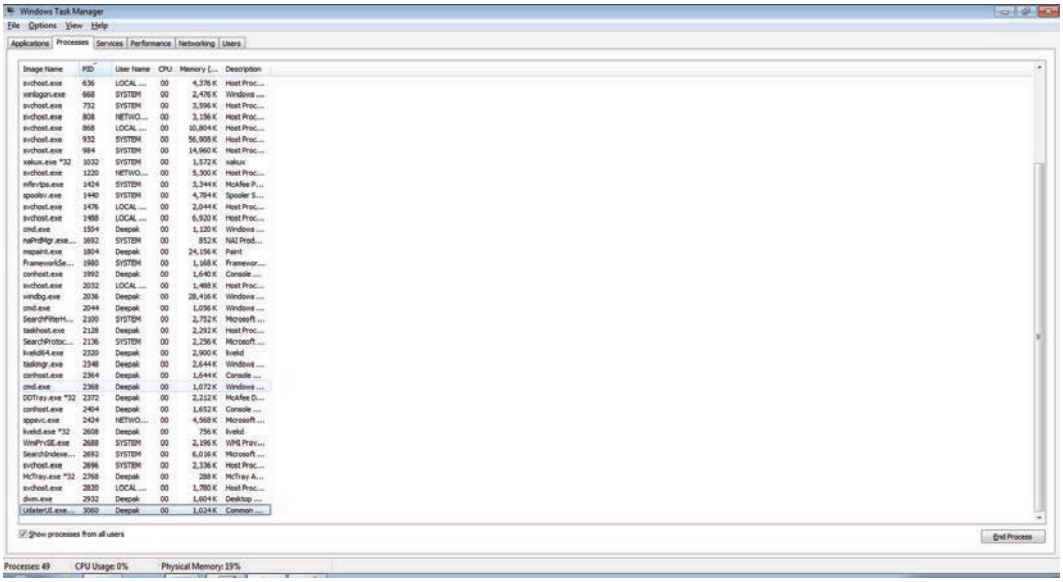

Figure 8. Sorted by Process ID, Task Manager no longer shows explorer.exe (PID=2960).

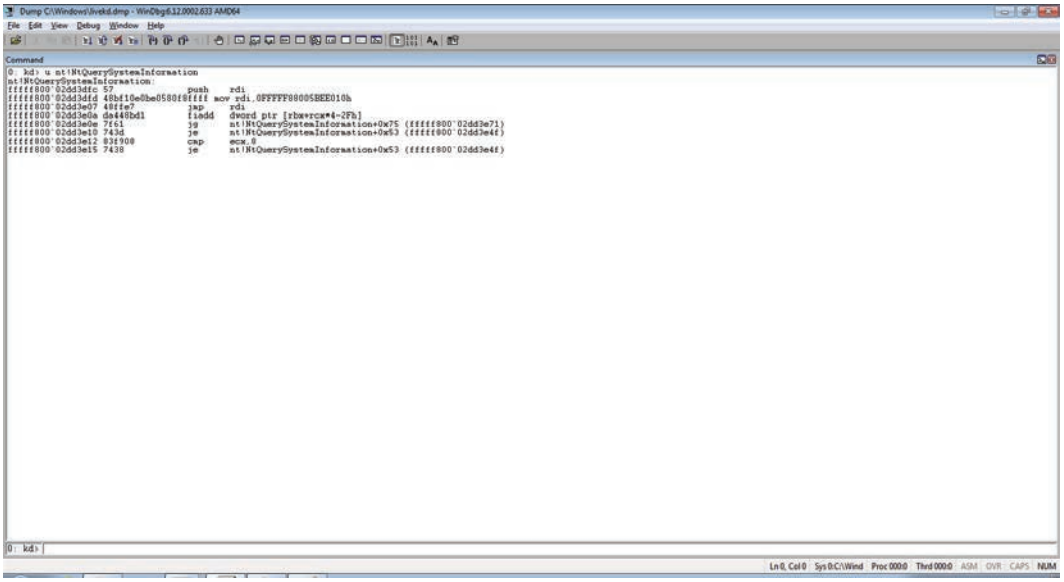

Figure 9. A view of the nt!NtQuerySystemInformation after the attack.

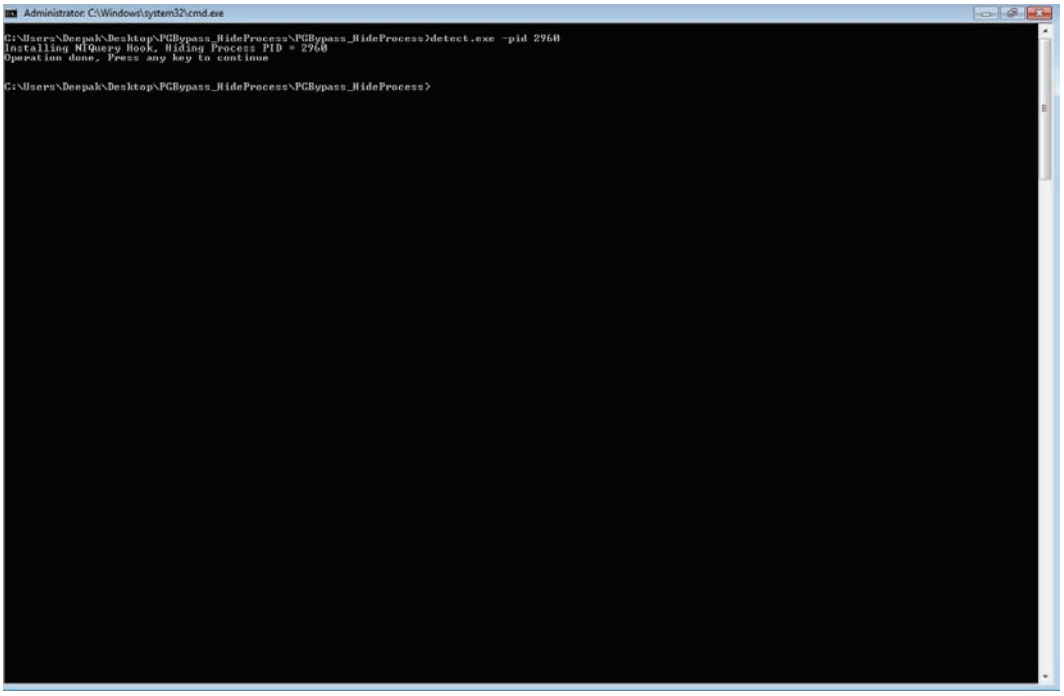

Figure 10. We attempt our attack again, only this time with McAfee-Intel DeepSAFE technology and McAfee Deep Defender enabled on the system. We create our hook and try to hide explorer.exe.

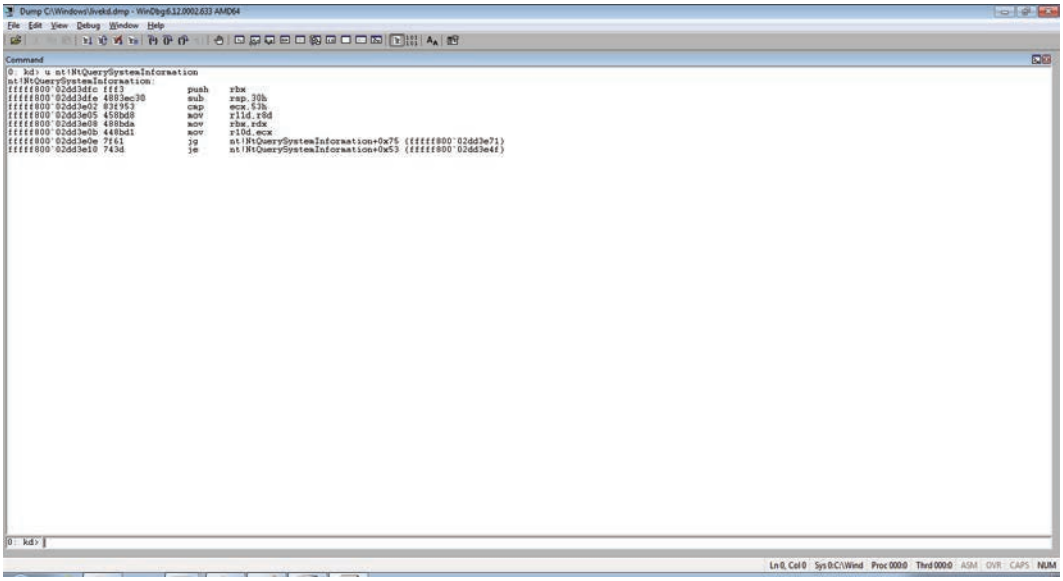

Figure 11. With DeepSAFE enabled, our view of nt!NtQuerySystemInformation after the attack shows that no bytes have been modified. The attack has been thwarted by DeepSAFE.

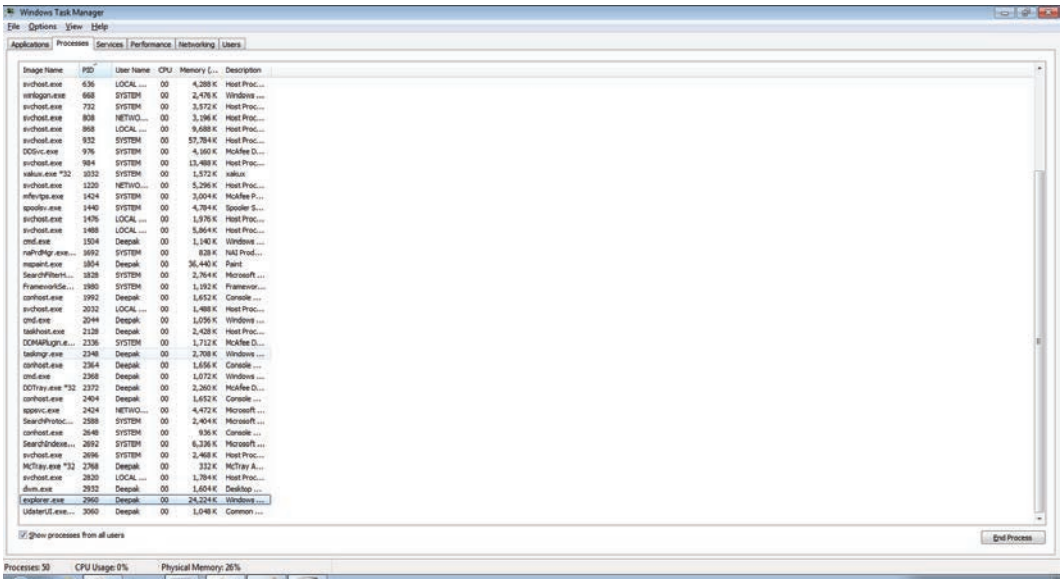

Figure 12. With DeepSAFE enabled, explorer.exe is still visible in Task Manager after the attack.

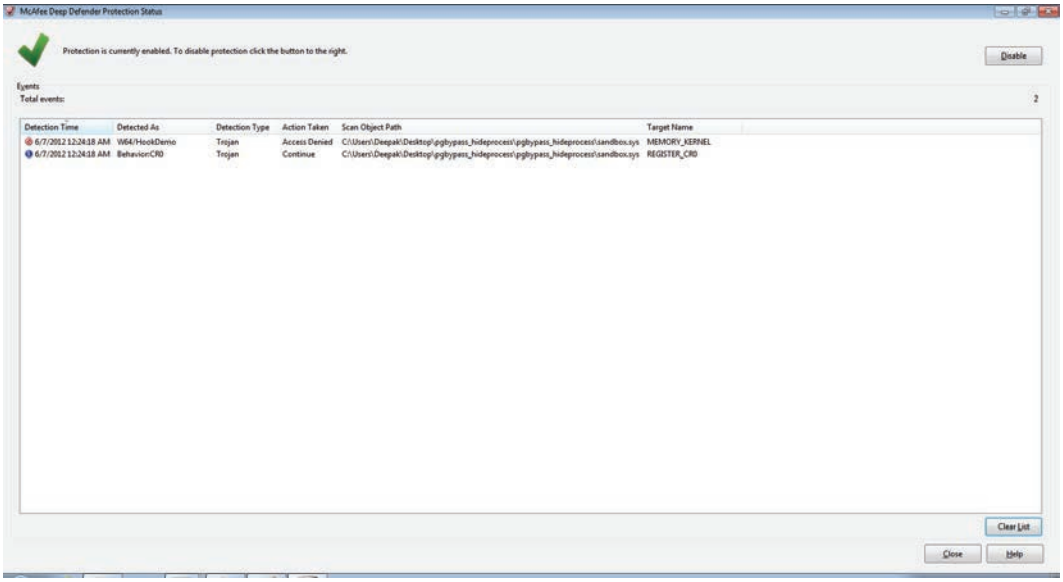

Figure 13. The report by DeepSAFE and McAfee Deep Defender after the attack was executed.

This attack is generic—but successful—because of the decryption engine; the same code works on all PatchGuard versions from Windows Vista through Windows 8 Preview. This attack also defeats PatchGuard's multiple parallel contexts, which appear in recent versions, because all contexts use the same decryption design. We needed only to search the whole kernel heap for these signatures and patch them with the encrypted return instructions.

A New Level of Security

It is impossible to protect a system from an attack that runs at the same privilege level at which its protection code runs. Both sets of code can pretty much run the same operations on the processor and there are no restrictions on either of them. PatchGuard has certainly raised the security bar to a new level, restricting the number of kernel-mode rootkits on 64-bit platforms. Nonetheless, we have already seen a number of published techniques for attacking PatchGuard. Up to now, these attacks have been stopped by subsequent versions of PatchGuard. But this success may not continue.

We have presented an attack on the basic code encryption design of PatchGuard that works on the three latest versions of Windows. PatchGuard has used several methods of obfuscation, misdirection, randomization, and encryption to initialize, operate, and hide its footprints. But PatchGuard cannot hide from the fact that it runs at the same privilege level as some attacks.

We wrote this paper to emphasize that it is not possible to provide software-only security against kernel-level attacks; there will always be a way to fool or circumvent software defenses. Security developers can go only so far with their implementations; they can't afford to destabilize systems to protect them. For example, the current implementation of PatchGuard changes the IDT for a short period to guard against debug register attacks. That's the same table that PatchGuard is trying to defend.

To remove the vulnerability that we have discovered, Microsoft might have to move the PatchGuard validation routines to a memory area that doesn't come under heap ranges and hide that area from the rest of the system, or they could change their encryption strategy. Regardless of any changes, how long will it take for a dedicated researcher, or a malware author, to successfully attack the new implementation? There are no hardware restrictions on attacks related to execution control.

There is, however, a way to stop our PatchGuard attack. (See Figures 10–13.) This is where security enforcement through hardware features enters the picture. Intel and McAfee have jointly developed McAfee DeepSAFE® technology, which leverages the virtualization feature of the latest 64-bit processors to run a secure layer of code at a new highest privilege level (VMX root) while running the operating system's kernel code at its intended level of privilege. This secure layer configures key areas—including CPU registers and memory areas—for protection. If any code running at kernel level tries to modify these key areas, hardware enforces an exit to the DeepSAFE security layer, which decides whether to allow the modification. Because code running at kernel level is no longer the highest level, DeepSAFE protects the system against any kernel-level attacks. Our attack against systems running only software defenses succeeded, but our attacks against systems with DeepSAFE—running McAfee application Deep Defender—failed.

About the Authors

Deepak Gupta (deepak_gupta4@mcafee.com) is a Senior Antimalware Researcher at McAfee Labs. He works on future-generation security product research. Gupta has been a kernel developer and has enjoyed reverse engineering and malware analysis as hobby for the last six years.

Xiaoning Li (xiaoning.li@intel.com) is a Security Researcher and Architect at Intel Labs. He works on proactive defense research with new processor and platform approaches to mitigate advanced cyberattacks, especially for zero-day threats. Li has worked on reverse engineering, vulnerability research, malware analysis, and operating system internals for more than 14 years. He has discovered and reported many vulnerabilities, from silicon to applications.

About McAfee Labs

McAfee Labs is the global research team of McAfee. With the only research organization devoted to all threat vectors—malware, web, email, network, and vulnerabilities—McAfee Labs gathers intelligence from its millions of sensors and its cloud-based service McAfee Global Threat Intelligence™. The McAfee Labs team of 350 multidisciplinary researchers in 30 countries follows the complete range of threats in real time, identifying application vulnerabilities, analyzing and correlating risks, and enabling instant remediation to protect enterprises and the public.

About McAfee

McAfee, a wholly owned subsidiary of Intel Corporation (NASDAQ:INTC), is the world's largest dedicated security technology company. McAfee delivers proactive and proven solutions and services that help secure systems, networks, and mobile devices around the world, allowing users to safely connect to the Internet, browse, and shop the web more securely. Backed by its unrivaled Global Threat Intelligence, McAfee creates innovative products that empower home users, businesses, the public sector, and service providers by enabling them to prove compliance with regulations, protect data, prevent disruptions, identify vulnerabilities, and continuously monitor and improve their security. McAfee is relentlessly focused on finding new ways to keep our customers safe. www.mcafee.com

[1] www.uninformed.org/?v=3&a=3&t=txt
[2] www.uninformed.org/?v=6&a=1&t=txt
[3] www.uninformed.org/index.cgi?v=8&a=5&t=txt
[4] http://msdn.microsoft.com/en-us/library/1eyas8tf.aspx
[5] http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html
[6] http://www.cs.uiuc.edu/homes/kingst/spring2007/cs598stk/papers/p63-0x08_Raising_The_Bar_For_Windows_Rootkit_Detection.txt

2821 Mission College Boulevard
Santa Clara, CA 95054
888 847 8766
www.mcafee.com

47902rpt_patchguard_0712_fnl_ETMG