Windows Vista Kernel-Mode: Functions, Security Enhancements and Flaws

Mohammed D. ABDULMALIK and Shafi'i M. ABDULHAMID

Mathematics/Computer Science Department, Federal University of Technology Minna, Nigeria.
E-mails: maliks26@hotmail.com, shafzon@yahoo.com

Abstract

Microsoft has made substantial enhancements to the kernel of the Microsoft Windows Vista operating system. Kernel improvements are significant because the kernel provides low-level operating system functions, including thread scheduling, interrupt and exception dispatching, multiprocessor synchronization, and a set of routines and basic objects.

This paper describes some of the kernel security enhancements for 64-bit edition of Windows Vista. We also point out some weakness areas (flaws) that can be attacked by malicious leading to compromising the kernel.

Keywords

Kernel, Kernel-Mode, Kernel Patching/Kernel hooking, PatchGuard.

Introduction

The kernel is the lowest-level, most central part of a computer operating system and one of the first pieces of code to load when the machine starts up. The kernel is what enables the software of the machine to talk to the hardware and is responsible for basic operating systems housekeeping tasks such as memory management, launching programs and processes, and managing the data on the disk. The performance, reliability, and security of the entire computer depend on the integrity of the kernel [1].

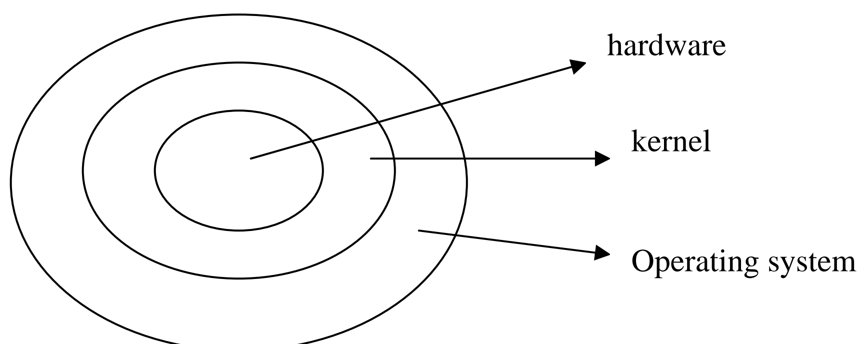

Figure 1. Structure of kernel-based operating system [2]

"*Kernel patching*" or "*kernel hooking*" is the practice of using unsupported mechanisms to modify or replace kernel code. Patching fundamentally violates the integrity of the Windows kernel and is undocumented, unsupported and has always been discouraged.

The new kernel-mode security features in Windows Vista include among them: *PatchGuard* and *Restricted user-mode access to \Device\Physical Memory.* These changes may secure the kernel of Windows Vista 64-bit Edition significantly.

Scope of this paper

The scope of this paper covers some of the new security features that have been incorporated to keep malicious code from compromising the kernel. Specifically, *PatchGuard* and *Restricted user-mode access to \Device\Physical Memory.* Since most of these features are only available in the 64-bit edition of Windows Vista, this paper will focus on the 64-bit edition.

Limitations of this paper

Only some attacks against PatchGuard will be discussed. This paper does not review the implementation of PatchGuard.

Windows Vista Booting Process

Microsoft has completely reengineered the boot environment for Windows Vista to address the increasing complexity and diversity of modern hardware and firmware. A key

aspect of this reengineering is a new firmware-independent data store called boot configuration data (BCD). BCD is designed to handle boot environment data for any type of system. It provides access to the information and applications that Windows Vista and later versions of Windows use to load the operating system or run boot applications such as memory diagnostics.

Some key Characteristics of boot configuration data (BCD)

- BCD provides clean and intuitive structured storage for boot settings.
- BCD abstracts the underlying data store, making BCD independent of the underlying firmware or processor architecture.
- BCD is available at run time as well as during the boot process.
- BCD provides improved security over *boot.ini.*
- BCD is designed to handle systems with multiple versions and configurations of Windows, including versions earlier than Windows Vista.
- BCD is the only boot data store that is required for Windows Vista and later versions of Windows [3].

Windows Vista Boot Manager

The process begins with Vista Boot Manager, located in the %SystemDrive%\bootmgr file (for PC/AT legacy BIOS) or %SystemDrive%\Boot\EFI\bootmgr.efi (for EFI BIOS). Though it can also be used to boot legacy versions of Windows, the Vista Boot Manager is required to boot Windows Vista.

The Vista Boot Manager calls InitializeLibrary, which in turn calls BlpArchInitialize (GDT, IDT, etc.), BlMmInitialize (memory management), BlpFwInitialize (firmware), BlpTpmInitialize (TPM), BlpIoInitialize (file systems), BlpPltInitialize (PCI configuration), BlBdInitialize (debugging), BlDisplayInitialize, BlpResourceInitialize (finds its own .rsrc section), and BlNetInitialize.

A typical BCD entry for the Boot Manager looks like this:

Windows Boot Manager

```
Identifier          {bootmgr}
Type                10100002
Device              partition=C:
Description         Windows Boot Manager
Locale              En-US
```

```
Inherit options              {globalsettings}
Boot debugger                No
Pre-boot EMS Enabled         No
Default                      {current}
Resume application           {3ced334e-a0a5-11da-8c2b-
                             cbb6baaeea6d}
Display order                {current}
Timeout                      30
```

Finally, the Boot Manager calls BlImgStartBootApplication to transfer control to the Windows Vista OS Loader [4].

Windows Vista OS Loader

The bootmgr calls the Windows Vista OS Loader, which is located under %SystemRoot%\System32\WINLOAD.EXE. WINLOAD.EXE replaces NTLDR (the legacy Windows NT OS loader). For the remainder of this section, "it" refers to the instructions in WINLOAD.EXE beginning at the entry point (OslMain).

A typical BCD entry for the Windows Vista OS Loader looks like this:

Windows Boot Loader

```
Identifier                {current}
Type                      10200003
Device                    Partition=C:
Path:                     \Windows\system32\WINLOAD.EXE
Description               Microsoft Windows
Locale                    en-US
Inherit options           {bootloadersettings}
Boot debugger             No
Pre-boot EMS Enabled      No
Advanced option           No
Options edito             No
Windows device            Partition=C:
Windows root              \Windows
Resume application        {3ced334e-a0a5-11da-8c2b-cbb6baaeea6d}
No Execute policy         OptIn
Detect HAL.               No
No integrity checks       No
Disable boot display      No
Boot processor only       No
Firmware PCI settings     No
Log initialization        No
OS boot information       No
Kernel debugger           No
HAL breakpoint            No
EMS enabled in OS         No
```

Execution begins at OslMain. It uses a lot of the same code bootmgr, so the InitializeLibrary works the same way in WINLOAD.EXE. After InitializeLibrary, control is transferred to OslpMain. The osloader.xsl file controls the advanced (Vista-specific) boot

options during the OSbootup. After handling the advanced boot options (in OslDisplayAdvancedOptionsProcess), WINLOAD.EXE is now ready to prepare for booting. Booting begins by first opening the boot device (using BlDeviceOpen). BlDeviceOpen will use a different set of device functions depending on the device type.

For example the function for block I/O (_BlockIoDeviceFunctionTable) these are:

```
dd offset _BlockIoEnumerateDeviceClass@12 ;
BlockIoEnumerateDeviceClass(x,x,x)
dd offset _BlockIoOpen@8 ; BlockIoOpen(x, x)
dd offset _BlockIoClose@4 ; BlockIoClose(x)
dd offset _BlockIoReadUsingCache@16 ; BlockIoReadUsingCache(x,x,x,x)
dd offset _BlockIoWrite@16 ; BlockIoWrite(x,x,x,x)
dd offset _BlockIoGetInformation@8 ; BlockIoGetInformation(x,x)
dd offset _BlockIoSetInformation@8 ; BlockIoSetInformation(x,x)
dd offset ?handleInputChar@OsxmlMeter@@UAEHG@Z ;
OsxmlMeter::handleInputChar(ushort)
dd offset _BlockIoCreate@12 ; BlockIoCreate(x,x,x)
```

Therefore, it is for console (_ConsoleDeviceFunctionTable), Full Volume Encryption (_FvebDeviceFunctionTable), serial port (_SerialPortFunctionTable) and PXE (_UdpFunctionTable). Some of the function callbacks are shared between different classes (e.g., serial port and PXE). After that, the LOADER_PARAMETER_BLOCK structure is initialized in OslInitializeLoaderBlock. The LOADER_PARAMETER_BLOCK contains information on the system state, such as boot device, ACPI and SMBios tables, etc. Next it discovers the system disks (OslEnumerateDisks) and loads the system hive KEY_LOCAL_MACHINE (OslpLoadSystemHive). After the system hive is loaded, we encounter the first code integrity check point in the Windows Vista boot process (OslInitializeCodeIntegrity). First it calls MincrypL_SelfTest, which validates the SHA1 hashing and PKCS1 signature verification algorithms are working (using a pre-defined test case). If the pre-defined test case fails, it returns error code 0xC0000428. Next, it checks if a debugger is enabled (BlBdDebuggerEnabled). If there is a debugger enabled, it calls KnownAnswerTest; otherwise, it skips the test and once all the integrity checks have passed (unless all integrity checks have been disabled), OslInitializeCodeIntegrity will return successfully, and execution continues in OslMain again.

The following boot drivers must also pass the code integrity checks *even if a debugger is enabled* (otherwise WINLOAD.EXE will refuse to boot Windows Vista):

```
\Windows\system32\bootvid.dll
\Windows\system32\ci.dll
\Windows\system32\clfs.sys
\Windows\system32\hal.dll
\Windows\system32\kdcom.dll (or kd1394.sys or kdusb.dll, depending on boot options)
\Windows\system32\ntoskrnl.exe
\Windows\system32\pshed.dll
\Windows\system32\WINLOAD.EXE
\Windows\system32\drivers\ksecdd.sys
\Windows\system32\drivers\spldr.sys
\Windows\system32\drivers\tpm.sys
```

Assuming all images passed the code integrity check, then NTOSKRNL.EXE and all of its imports are now loaded. After this, OslHiveFindDrivers is used to locate all the boot drivers and sort them based on the Group (which is ordered according to HKEY_LOCAL_MACHINE\CurrentControlSet\Control\GroupOrderList) and Tag (an integer which determines each driver's order within its respective group). This sorted list of boot drivers is then passed to OslLoadDrivers for loading. OslLoadDrivers calls LoadImageEx for each driver in the list. LoadImageEx will load each driver and all of its dependencies.

At this point, all the boot drivers are loaded. Next, OslpLoadNlsData is called to load native language locale information from HKEY_LOCAL_MACHINE\CurrentControlSet\Control\NLS. Finally, the last step of OslpLoadAllModules is to call OslpLoadMiscModules which does the following:

- Shows the progress bar seen during bootup
- Loads %SystemRoot%\AppPatch\drvmain.sdb (the application compatibility database)
- Loads %SystemRoot%\System32\acpitabl.dat
- Loads an INF file pointed to by HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Errata\InfName if present in the registry [4].

Windows Vistas Kernel

NTOSKRNL.EXE is responsible for the verification of system drivers (loaded after boot drivers) and drivers loaded at runtime (i.e., by the user or a device being inserted into the

system), in contrast, WINLOAD.EXE is responsible for checking the integrity of the signatures of boot drivers.. When integrity checks are enabled, the code integrity of the loaded image is checked SeValidateImageHeader (a wrapper to CiValidateImageHeader in CI.DLL) and SeValidateImageData (a wrapper to CiValidateImageData in CI.DLL). SeValidateImageHeader is called whenever an executable is mapped into kernel memory (via MmCreateSection). The code sections of kernel drivers are verified in SeValidateImageData which is called when a kernel module is being loaded. Runtime checks (e.g., continuously polling for modifications to the code sections of kernel drivers) are handled by PatchGuard and CI.DLL

PatchGuard

Functions of PatchGuard

PatchGuard's primary reason for existence is to prevent kernel-level rootkits. A rootkit is a set of software tools intended to conceal running processes, files or system data, thereby helping an intruder to gain and maintain access to a system while avoiding detection Rootkits often try to gain access to the kernel of the operating system. Kernel rootkits can be especially dangerous because they can be difficult to detect and are almost impossible to remove.

PatchGuard does not prevent all rootkits or other malware from attacking the operating system. However, it does mitigate one uniquely destructive way to attack the system, namely patching kernel structures and code to manipulate kernel functionality. Protecting the integrity of the kernel is a fundamental step in protecting the entire system from malicious attacks and the reliability problems that may result from even well-intentioned patching [4].

PatchGuard is hidden within NTOSKRNL.EXE (obscured, but not encrypted) and checks the critical system structures at random intervals, usually around 5-10 minutes. When a modification is detected, the system will blue screen with the following bug check (which will obviously cause the user to lose all unsaved data):

```
CRITICAL_STRUCTURE_CORRUPTION (109)
Bug check is generated when the kernel detects that critical
kernel code or data have been corrupted. There are generally
three causes for a corruption:
```

```
1) A driver has inadvertently or deliberately modified
critical kernel code or data. See
http://www.microsoft.com/whdc/driver/kernel/64bitPatching.mspx
2) A developer attempted to set a normal kernel breakpoint
using a kernel debugger that was not attached when the system
was booted. Normal breakpoints, "bp", can only be set if the
debugger is attached at boot time. Hardware breakpoints, "ba",
can be set at any time.
3) A hardware corruption occurred, e.g. failing RAM holding
kernel code or data.
Type of corrupted region, can be
0 : A generic data region
1 : Modification of a function or .pdata
2 : A processor IDT
3 : A processor GDT
4 : Type 1 process list corruption
5 : Type 2 process list corruption
6 : Debug routine modification
7 : Critical MSR modification
```

PatchGuard *cannot* be disabled. Even when integrity checks are disabled, PatchGuard is still active [5].

PatchGuard Detection

The two methods proposed to locate PatchGuard are:

1. Walk the KiTimerListHead

The background is that PatchGuard must register a timer event that will trigger the next scan (PatchGuard scans for changes in random intervals). These entries are represented using the KTIMER structure. The PatchGuard entry is easy to detect because all other entries will have a valid DeferredContext pointer in the KTIMER structure.

The problem is that this list, pointed to by the KiTimerListHead variable is exported. We propose a reliable method to find the KiTimerListHead variable. After locating the KiTimerListHead variable, we traverse the linked list and locate all time entries that do not have a valid DeferredContext pointer. We cannot find the location of PatchGuard code in memory from the DeferredContext pointer because it is encoded (using unknown random numbers). Instead, we can remove the entries to disable PatchGuard. This is a partial implementation in C utilizing this technique:

```
LIST_ENTRY *GetKiTimerListHead()
{
      KTIMER Timer;
      LARGE_INTEGER DueTime;
      KIRQL OldIrql;
      LIST_ENTRY *ListHead;
      KeInitializeTimer(&Timer);
      // If KeSetTimer returns TRUE, this is guaranteed to be index 0
      because
```

Issue 12, January-June 2008
p. 99-110

```
        // we used the smallest possible time.
        // Likewise, we will be at the head of the list because there can't
        be anything smaller.
        // If KeSetTimer returns FALSE, then the timer already expired
        // So just use the smallest unit possible and we be at
        KiTimerListHead[0].Flink
        DueTime.QuadTime = -1;
        // Negative times are relative to current time--that's what we're
        interested in
        // If the timer object was already in the timer queue,
        // it is implicitly canceled before being set to the new expiration
        time.
        KeRaiseIrql(DISPATCH_LEVEL, &OldIrql);
        while (!KeSetTimer(&Timer, DueTime)) DueTime.QuadTime--;
        ListHead = Timer.TimerListEntry.Blink;
        KeCancelTimer(&Timer);
        KeLowerIrql(OldIrql);
        return ListHead;
}
void DisablePatchGuard()
{
        LIST_ENTRY *TimerTable = GetKiTimerListHead();
        ASSERTMSG("Couldn't find KiTimerTableListHead", TimerTable);
        if (TimerTable)
        {
        do
                {
                ListHead = &TimerTable[Index];
                NextEntry = ListHead->Flink;
                while (NextEntry != ListHead)
                {
                        Timer = CONTAINING_RECORD(NextEntry, KTIMER,
                        TimerListEntry);
                        NextEntry = NextEntry->Flink;
                        ASSERT(Timer->Dpc && Timer->Dpc->DeferredRoutine);
                        // Current DeferredRoutine will be either
                        KiScanReadyQueues,
                        // ExpTimeRefreshDpcRoutine, or ExpTimeZoneDpcRoutine
                        if (IS_IN_NTOSKRNL(Timer->Dpc->DeferredRoutine) &&
                        !MmIsValidAddress(Timer->Dpc->DeferredContext))
                                {
                                RemoveEntryList(&Timer->TimerListEntry);
                                return;
                                }
                        }
                        Index += 1;
                } while(Index < MAX_INDEX);
                ASSERTMSG("Couldn't find PatchGuard timer", 0);
        }
}
```

2. Utilize a memory read breakpoint [7].

- Add a memory read breakpoint (using the Intel debug registers) on IDT entry 1.
- Add an interrupt 3 (breakpoint exception) handler. PatchGuard will scan the IDT sequentially from the first entry to the last, so PatchGuard thread will trigger the

memory read breakpoint. A custom interrupt handler will be installed to handle breakpoint exceptions.

- Wait for the memory read breakpoint exception to call the special interrupt handler we've installed. If the interrupt is not a memory read breakpoint on the first IDT entry, then this exception will be passed to the original interrupt 3 handler. Otherwise, the faulting instruction pointer is the PatchGuard thread and steps can be taken to disable PatchGuard (such as overwriting the PatchGuard code page with NOPs). This approach is likely to be effective at detecting PatchGuard since it detects a basic behavior of any integrity-checking memory scanning algorithm.

Restricted User-Mode Access To \Device\Physical Memory

Disabling user-mode access to \Device\PhysicalMemory is also a significant step in reducing the possibility of malicious code entering the kernel. It was first disabled in Windows 2003 SP1 and is still disabled in Windows Vista [6]. This can be done either by (1) scanning physical memory for a known signature of the area an attacker wishes to modify, or (2) calculating the physical address from a virtual address. One very convenient attack is to find the location of the Global Descriptor Table (GDT) and add a ring 0 call gate. Malicious code can then utilize the call gate using the CALL FAR instruction to jump into the kernel. Another technique is to find the Interrupt Descriptor Table (IDT) and install an interrupt gate. Malicious code can then use the INT instruction to utilize it and jump into the kernel. The author previously created proof-of-concept tool that utilized \Device\PhysicalMemory to detect BIOS rootkits.

Conclusions

Windows Vista's out-of-the-box kernel-mode security is a significant improvement over previous versions of Windows. It is likely that the security community will aggressively probe and seek to undermine Vista's kernel security improvements. The Windows Vista kernel enhancements are aimed at preventing unsigned code from being injected into the

kernel and to establish a chain-of-trust from the time that Vista boots until applications are run.

In this paper, we have identified some limits to the effectiveness of Windows Vista's new kernel-mode security capabilities. The malicious driver could be hooked to NtQueryInformationFile and NtCreateFile (after disabling PatchGuard) to redirect attempts to load the NTOSKRNL.EXE or WINLOAD.EXE to the original, unmodified copy. This is to prevent any user-mode tools from detecting that the binaries have been patched. The only way to detect that the files have been patched would be to inspect them "on-disk" at a lower level.

Because this research work started when Windows Vista was still in beta, some of the behaviors described have changed prior to Windows Vista's public release. Therefore, it is advised the reader continues to follow up on Microsoft Vista blogs such as *http://blogs.msdn.com/uac* and *http://blogs.msdn.com/ie*.

Reference

1. IBM, *IBM Internet Security Systems Supports Microsoft Vista's Kernel-Locking or Improved Customer Security,* http://www.iss.net/ iss_vista_kernel_lock_whitepaper.pdf, 2007
2. Dang Van Duc, et al., *Operating Systems*, *chap. 1. (Fig. 1.2).* Institute of Information Technology, Hoang Quoc Viet Road, Cau Giay District, Hanoi, Vietnam. E-mail: dvduc@ioit.ncst.ac.vn. 2004.
3. Microsoft, *Windows Vista and Windows server Longhorn,* 2006. http://www.microsoft.com/whdc/system/vista/kernel-en.mspx
4. Conover M., *Assessment of Windows Vista Kernel-Mode Security,* 2006, March, http://www.symantec.com/avcenter/reference/Windows_Vista_Kernel_mode_Security.pdf
5. Skape, Skywing, *Bypassing PatchGuard on Windows x64 Uninformed* ,2005, December, 1,Volume 3, http://www.uninformed.org/?v=3&a=3&t=txt
6. Crazylord, *Playing with Windows /dev/(k)mem,* Phrack 11(59), http://www.phrack.org/phrack/59/p59-0x10.txt
7. Silberschatz et al, *Operating Systems Concepts*; Seventh Edition (chapter 9, pg. 353), 2004.

8. Microsoft, *Benefits of Microsoft Windows x64 Editions,* 2005, April. http://download.microsoft.com/download/D/A/A/DAA7245D-E01D-46A4-AB70-3A95ED3F6934/Windowsx64BenefitsWP.doc
9. Microsoft, *Boot Configuration Data Editor Frequently Asked Questions*, *TechNet*, http://www.microsoft.com/technet/windowsvista/library/85cd5efe-c349-427c-b035-c2719d4af778.mspx
10. Microsoft, *"Device\PhysicalMemory Object,"* *TechNet*, http://technet2.microsoft.com/WindowsServer/en/Library/e0f862a3-cf16-4a48-bea5-f2004d12ce351033.mspx?mfr=true
11. Shafi'i M. A., M. D. Abdulmalik, *Analysis of the Windows Vista security model and the implications in the Nigerian market,* June, 2007, volume 5, p. 111 - 116.