# INTRODUCING WINDOWS VISTA SECURITY

Hadeel T. Al-Rayes Diyala Univ. – Essential Col. Computer Science Dept.

## Abstract

Windows Vista introduces several additional barriers that aim to prevent malicious code from gaining access to the operating system kernel. This paper is intended to provide a technical review of their implementation. The kernel mode security enhancements in Windows Vista are quite substantial, resulting in a dramatic reduction of its overall attack surface. However, the researcher has identified certain weaknesses in the kernel enhancements that may be leveraged by malicious code to undermine these improvements.

## INTRODUCTION

Windows Vista introduces a number of security enhancements over prior versions of Microsoft Windows (including Windows XP SP2). The new kernel-mode security features in Windows Vista include among them:

- Driver signing [1]
- PatchGuard [2]
- Kernel-mode code integrity checks [3]
- Optional support for Secure Bootup using a TPM hardware chip [4]
- Restricted user-mode access to \Device\PhysicalMemory [5]

These changes may secure the kernel of Windows Vista 64-bit Edition significantly; even when compared to that of Linux or Mac OS X. The contributions of this paper are: (1) a thorough analysis of the kernel-mode security components through reverse engineering and (2) an assessment of potential kernel-mode attacks.

## A. What's Covered

This paper examines the new security features that have been incorporated to keep malicious code from compromising the kernel. 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.

## B. What's Not Covered

This paper does not review the implementation of PatchGuard. An analysis of PatchGuard was performed previously by Skape and Skywing [6], however it should be noted that its implementation has since changed. Only some attacks against PatchGuard will be discussed. An assessment of Vista user-mode security was previously covered in [7].

# VISTA BOOT PROCESS

## A. Windows Vista Boot Manager (Stage 1)

Windows Vista supports booting from a legacy PC/AT BIOS and Intel's new Extensible Firmware Initiative (EFI). The analysis was performed against the EFI version. For the remainder of this section, "it" refers to the instructions in bootmgr beginning at the entry point (DllMain for EFI, and instruction at offset 0 for PC/AT).

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.

The boot.ini configuration file has been replaced with Boot Configuration Data file in %SystemDrive%\Boot\BCD. This file is a registry hive (also mounted under HKEY_LOCAL_MACHINE\BCD00000000 on Windows Vista). Its contents can be viewed in a more human readable form using bcdedit.exe [10].

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

If there is only one boot application entry in the BCD, the Boot Manager will boot from that entry. If there is more than one entry, the Boot Manager will present the user a list of bootable choices and ask the user to choose. If boot status logging is enabled, the Boot Manager will write its status into the file %SystemDrive%\Boot\bootstat.dat (via BmpInitializeBootStatusDataLog). Next the Boot Manager will locate bootmgr.xsl in the resource section (of its own executable file) using BlResourceFindHtml and then pass it to BlXmiInitialize. The bootmgr.xsl file controls what the boot menu looks like and the options exposed through the boot menu.

Once the boot application is selected, it is loaded with BmpLaunchBootEntry followed by BmpTransferExecution.

BmpTransferExecution will retrieve the boot options (via BlGetBootOptionString) and pass them to BlImgLoadBootApplication. If Full Volume Encryption (FVE) is enabled, BlFveSecureBootUnlockBootDevice and BlFveSecureBootCheckpointBootApp will be called. This is necessary because the Windows system partition is encrypted and must be decrypted before control can be transferred to the Vista OS Loader.
Finally, the Boot Manager calls BlImgStartBootApplication to transfer control to the Windows Vista OS Loader.

## B. Windows Vista OS Loader (Stage 2)

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 options: No
Options editor: 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 reuses a lot of the same code bootmgr uses, so the InitializeLibrary described previously for bootmgr works the same way in WINLOAD.EXE. After InitializeLibrary, control is transferred to OslpMain.

If boot status logging is enabled, WINLOAD.EXE will write the results to %SystemDrive%\Boot\bootstat.dat (via OslpInitializeBootStatusDataLog and OslpSetBootStatusData). Next WINLOAD.EXE calls OslDisplayInitialize and locates osloader.xsl in the resource section using BlResourceFindHtml. Control is then passed to BlXmiInitialize. The osloader.xsl file controls the advanced (Vista-specific) boot options during the OS bootup. 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 Full Volume Encryption (_FvebDeviceFunctionTable) these are:

```
dd 0 ; FVE has no EnumerateDeviceClass callback
dd offset _FvebOpen@8 ; FvebOpen(x,x)
dd offset _FvebClose@4 ; FvebClose(x)
dd offset _FvebRead@16 ; FvebRead(x,x,x,x)
dd offset _FvebWrite@16 ; FvebWrite(x,x,x,x)
dd offset _FvebGetInformation@8 ; FvebGetInformation(x,x)
```

dd offset _FvebSetInformation@8 ; FvebSetInformation(x,x)
dd offset _FvebReset@4 ; FvebReset(x)
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)
For console (_ConsoleDeviceFunctionTable) these are:
dd offset _UdpEnumerateDeviceClass@12 ; UdpEnumerateDeviceClass(x,x,x)
dd offset _ConsoleOpen@8 ; ConsoleOpen(x,x)
dd offset _ConsoleClose@4 ; ConsoleClose(x)
dd offset _ConsoleRead@16 ; ConsoleRead(x,x,x,x)
dd offset _ConsoleWrite@16 ; ConsoleWrite(x,x,x,x)
dd offset _ConsoleGetInformation@8 ; ConsoleGetInformation(x,x)
dd offset _ConsoleSetInformation@8 ; ConsoleSetInformation(x,x)
dd offset _ConsoleReset@4 ; ConsoleReset(x)
For serial port (_SerialPortFunctionTable) these are:
dd offset _UdpEnumerateDeviceClass@12 ; UdpEnumerateDeviceClass(x,x,x)

dd offset _SpOpen@8 ; SpOpen(x,x)

dd offset _SpClose@4 ; SpClose(x)
dd offset _SpRead@16 ; SpRead(x,x,x,x)
dd offset _SpWrite@16 ; SpWrite(x,x,x,x)
dd offset _SpGetInformation@8 ; SpGetInformation(x,x)
dd offset _SpSetInformation@8 ; SpSetInformation(x,x)
dd offset _SpReset@4 ; SpReset(x)

For PXE (_UdpFunctionTable):

dd offset _UdpEnumerateDeviceClass@12 ; UdpEnumerateDeviceClass(x,x,x)
dd offset _UdpOpen@8 ; UdpOpen(x,x)
dd offset _SpClose@4 ; SpClose(x)
dd offset _UdpRead@16 ; UdpRead(x,x,x,x)
dd offset _UdpWrite@16 ; UdpWrite(x,x,x,x)
dd offset _UdpGetInformation@8 ; UdpGetInformation(x,x)
dd offset _UdpSetInformation@8 ; UdpSetInformation(x,x)
dd offset _UdpReset@4 ; UdpReset(x)

You'll notice that some of the function callbacks are shared between different classes (e.g., serial port and PXE).

Next 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. This is what LOADER_PARAMETER_BLOCK looks like on the Windows Vista 64-bit Edition:

+0x000 LoadOrderListHead : struct _LIST_ENTRY
+0x010 MemoryDescriptorListHead : struct _LIST_ENTRY
+0x020 BootDriverListHead : struct _LIST_ENTRY
+0x030 KernelStack : Uint8B
+0x038 Prcb : Uint8B
+0x040 Process : Uint8B
+0x048 Thread : Uint8B
+0x050 RegistryLength : Uint4B
+0x058 RegistryBase : Ptr64 to Void

+0x060 ConfigurationRoot : Ptr64 to struct _CONFIGURATION_COMPONENT_DATA
+0x068 ArcBootDeviceName : Ptr64 to Char
+0x070 ArcHalDeviceName : Ptr64 to Char
+0x078 NtBootPathName : Ptr64 to Char
+0x080 NtHalPathName : Ptr64 to Char
+0x088 LoadOptions : Ptr64 to Char
+0x090 NlsData : Ptr64 to struct _NLS_DATA_BLOCK
+0x098 ArcDiskInformation : Ptr64 to struct _ARC_DISK_INFORMATION
+0x0a0 OemFontFile : Ptr64 to Void
+0x0a8 SetupLoaderBlock : Ptr64 to struct _SETUP_LOADER_BLOCK
+0x0b0 Extension : Ptr64 to struct _LOADER_PARAMETER_EXTENSION
+0x000 Size : Uint4B
+0x004 Profile : struct _PROFILE_PARAMETER_BLOCK
+0x014 MajorVersion : Uint4B
+0x018 MinorVersion : Uint4B
+0x020 EmInfFileImage : Ptr64 to Void
+0x028 EmInfFileSize : Uint4B
+0x030 TriageDumpBlock : Ptr64 to Void
+0x038 LoaderPagesSpanned : Uint4B
+0x040 HeadlessLoaderBlock : Ptr64 to struct _HEADLESS_LOADER_BLOCK
+0x048 SMBiosEPSHeader : Ptr64 to struct _SMBIOS_TABLE_HEADER
+0x050 DrvDBImage : Ptr64 to Void
+0x058 DrvDBSize : Uint4B
+0x060 NetworkLoaderBlock : Ptr64 to struct _NETWORK_LOADER_BLOCK bytes
+0x068 FirmwareDescriptorListHead : struct _LIST_ENTRY
+0x078 AcpiTable : Ptr64 to Void
+0x080 AcpiTableSize : Uint4B

+0x084 BootViaWinload : Bitfield Pos 0, 1 Bit
+0x084 BootViaEFI : Bitfield Pos 1, 1 Bit
+0x084 Reserved : Bitfield Pos 2, 30 Bits
+0x088 LoaderPerformanceData : Ptr64 to struct _LOADER_PERFORMANCE_DATA
+0x090 BootApplicationPersistentData : struct _LIST_ENTRY
+0x0a0 WmdTestResult : Ptr64 to Void
+0x0a8 BootIdentifier : struct _GUID
+0x0b8 u : union
+0x000 I386 : struct _I386_LOADER_BLOCK
+0x000 CommonDataArea : Ptr64 to Void
+0x008 MachineType : Uint4B
+0x00c VirtualBias : Uint4B

Next it discovers the system disks (OslEnumerateDisks) and loads the system hive HKEY_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.

Next it loads the OS signed catalog from %SystemRoot%\System32\CatRoot\{F750E6C3-38EE-11D1-85E5-00C04FC295EE}\nt5.cat in BlImgRegisterCodeIntegrityCatalogs (internally this calls the MinCrypL_AddCatalog API function).

After the signed OS catalog nt5.cat is loaded, WINLOAD.EXE verifies its own integrity using SelfIntegrityCheck. This does two things:

1. Computes the SHA1 hash of the PE image, and then compares it to the SHA1 hash in the PE's certificate table entry. These must match or it will return an error.

2. In addition, it also calls ImgpValidateImageHash to verify the image hash matches the one in nt5.cat. ImgpValidateImageHash calls the API function MinCrypL_CheckSignedFile to verify the signature and API function MinCrypL_CheckImageHash to find the matching image hash in the signed catalog.

If the signature doesn't match but a debugger is enabled (BlBdDebuggerEnabled returns TRUE) then it will print the message:

*** Windows is unable to verify the signature of the file %s.
It will be allowed to load because the boot debugger is enabled.
Use g to continue!!

If a debugger is present, it will be activated via a call to DbgBreakPoint; otherwise, ReportCodeIntegrityFailure is called instead to report a fatal error.

Once all the integrity checks have passed (unless all integrity checks have been disabled), OslInitializeCodeIntegrity will return successfully, and execution continues in OslMain again. Now, OslpLoadAllModules is called to begin the loading of the system modules. First, OslLoadImage is called to load NTOSKRNL.EXE and HAL.DLL, but no imports are resolved at this time. Second, if kernel debugging is enabled, one of the debugging drivers is loaded depending on the boot debugging options (kdcom.dll for serial port, kd1394.dll for IEEE1394, or kdusb.dll for USB). Third, the imports of NTOSKRNL.EXE are loaded and initialized (using the LoadImports and BindImportRefences functions).

OslLoadImage calls GetImageValidationFlags to check the filename against a pre-defined list of boot drivers in LoadBootImagesTable. If integrity checks are enabled, then boot drivers must be signed by a trusted root authority and all the image hashes must match the signed catalog file unless a debugger is enabled. If a debugger is enabled,

WINLOAD.EXE does not enforce this requirement. Instead it will print an error message to the debugger, but will otherwise ignore the code integrity check failure. However, the following boot drivers (also listed in Appendix A) must 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
The process of loading the image and verifying its code integrity is done within BlImgLoadPEImageEx—it uses the same functions as SelfIntegrityCheck (described earlier in this section). Assuming all images passed the code integrity check, then NTOSKRNL.EXE and all of its imports are now loaded. This list (as of Build 5365) is:
\Windows\system32\NTOSKRNL.exe
\Windows\system32\HAL.dll
\Windows\system32\PSHED.dll
\Windows\system32\BOOTVID.dll
\Windows\system32\CLFS.SYS
\Windows\system32\CI.dll
Next 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, the remaining boot drivers are loaded and initialized. As of Build 5365 (64-bit Edition) this list in chronological order is:

1. \Windows\system32\drivers\Wdf01000.sys
2. \Windows\system32\drivers\WDFLDR.SYS
3. \Windows\system32\drivers\acpi.sys
4. \Windows\system32\drivers\WMILIB.SYS
5. \Windows\system32\drivers\msisadrv.sys
6. \Windows\system32\drivers\pci.sys
7. \Windows\system32\drivers\volmgr.sys
8. \Windows\system32\drivers\isapnp.sys
9. \Windows\system32\drivers\mpio.sys
10. \Windows\system32\drivers\compbatt.sys
11. \Windows\system32\drivers\BATTC.SYS
12. \Windows\System32\drivers\mountmgr.sys
13. \Windows\system32\drivers\intelide.sys
14. \Windows\system32\drivers\PCIIDEX.SYS
15. \Windows\system32\drivers\pcmcia.sys
16. \Windows\system32\drivers\aliide.sys
17. \Windows\system32\drivers\amdide.sys
18. \Windows\system32\drivers\cmdide.sys
19. \Windows\system32\drivers\msdsm.sys
20. \Windows\system32\drivers\pciide.sys
21. \Windows\system32\drivers\viaide.sys
22. \Windows\System32\drivers\volmgrx.sys
23. \Windows\system32\drivers\atapi.sys
24. \Windows\system32\drivers\ataport.SYS
25. \Windows\system32\drivers\hpcisss.sys
26. \Windows\system32\drivers\storport.sys
27. \Windows\system32\drivers\adp94xx.sys
28. \Windows\system32\drivers\adpu160m.sys

29. \Windows\system32\drivers\SCSIPORT.SYS
30. \Windows\system32\drivers\adpu320.sys
31. \Windows\system32\drivers\djsvs.sys
32. \Windows\system32\drivers\arc.sys
33. \Windows\system32\drivers\arcsas.sys
34. \Windows\system32\drivers\elxstor.sys
35. \Windows\system32\drivers\i2omp.sys
36. \Windows\system32\drivers\iirsp.sys
37. \Windows\system32\drivers\iteraid.sys
38. \Windows\system32\drivers\lsi_fc.sys
39. \Windows\system32\drivers\lsi_sas.sys
40. \Windows\system32\drivers\lsi_scsi.sys
41. \Windows\system32\drivers\megasas.sys
42. \Windows\system32\drivers\mraid35x.sys
43. \Windows\system32\drivers\msahci.sys
44. \Windows\system32\drivers\nfrd960.sys
45. \Windows\system32\drivers\ql2300.sys
46. \Windows\system32\drivers\ql40xx.sys
47. \Windows\system32\drivers\sisraid2.sys
48. \Windows\system32\drivers\sisraid4.sys
49. \Windows\system32\drivers\symc8xx.sys
50. \Windows\system32\drivers\sym_hi.sys
51. \Windows\system32\drivers\sym_u3.sys
52. \Windows\system32\drivers\vsmraid.sys
53. \Windows\system32\drivers\fltmgr.sys
54. \Windows\system32\drivers\fileinfo.sys
55. \Windows\system32\drivers\ndis.sys
56. \Windows\system32\drivers\msrpc.sys
57. \Windows\system32\drivers\NETIO.SYS
58. \Windows\System32\Drivers\Ntfs.sys

At this point all the boot drivers are loaded (see Appendix B for the full list of drivers loaded in chronological order). 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:

1. Displays the progress bar seen during bootup
2. Loads %SystemRoot%\AppPatch\drvmain.sdb (the application compatability database)
3. Loads %SystemRoot%\System32\acpitabl.dat

4. Loads an INF file pointed to by HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Errata\InfName if present in the registry.

After OslpLoadAllModules has finished, OslMain saves the boot log (OslpLogSaveInformation), finishes the Full Volume Encryption loading if enabled (BlFveSecureBootRestrictToOne and BlTpmShutdown), and finally calls OslArchTransferToKernel to transfer control to NTOSKRNL.EXE.

## C. Vista Windows OS Kernel (Stage 3)

Windows Vista uses the same naming convention as previous versions of Windows. For 64-bit Windows Vista there is a single version of NTOSKRNL.EXE located under %SystemRoot%\System32\ntoskrnl.exe. For the remainder of this section, “it” refers to the instructions in ntoskrnl.exe beginning at the entry point (KiSystemStartup).

Execution begins in KiSystemStartup. A significant portion of NTOSKRNL.EXE is unchanged from Windows 2003 SP1, so we focus on the changes that are specific to Windows Vista. NTOSKRNL.EXE adds a new function SepInitializeCodeIntegrity which is just a wrapper to the CiInitialize function in CI.DLL (CiInitialize is discussed later in Section VI). If code integrity checks are enabled, then SepInitializeCodeIntegrity calls CiInitialize; otherwise, it doesn’t do anything.

WINLOAD.EXE was responsible for checking the integrity of the signatures of boot drivers. NTOSKRNL.EXE, in contrast, 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). 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—discussed later in this paper.

## DRIVER SIGNING

Observing past exploits, the most common mechanism used by malicious code to enter the Windows XP kernel is through a driver. A Windows XP user would browse the Internet using Internet Explorer, and then a malicious banner ad or website would exploit an Internet Explorer vulnerability to install malware on the victim's machine. Since the de facto user account usually had administrative privileges, the malware could then install a kernel-mode rootkit with a few simple API calls, such as ZwLoadDriver.

In the Windows Vista 64-bit edition, all drivers must be signed by a Class 3 code signing certificate. By requiring all drivers to be signed by a trusted certificate authority (i.e., VeriSign or Microsoft), kernel-mode drivers will no longer pose a threat unless a rogue vendor convinces a trusted certificate authority to issue them a publishing certificate. The onus, however, is upon these trusted certificate authorities to not give code signing certificates to companies that install malicious or

questionable applications. If they did, this enhancement will be undermined. An unscrupulous entrepreneur could register a legitimate business, obtain a publishing certificate from a trusted certificate authority, and then sign drivers on behalf of malicious vendors for a profit. It is incumbent upon Microsoft and all software vendors to keep their signing certificates secure, and only sign software that meets the users' expectations for acceptable behavior.

With respect to the integrity of boot drivers (those handled by WINLOAD.EXE), there is no mechanism to revoke a driver signing certificate (as of Build 5365). It seems that the problem with this approach is that if even a single software publishing certificate is stolen and published on the Internet (thus permitting anyone to sign their own drivers) then driver signing checks will become ineffective. Since even revoked certificates can be used to sign boot drivers, each certificate owner must ensure that employees do not have direct access to the publishing certificate and prevent an employee from taking a copy of the certificate in the event that he separates from the company. Microsoft has stated that they now plan to incorporate certificate revocation into Windows Vista RC1.

## Implementation

Windows Vista not only requires the driver be signed, but it also requires it be signed by one of eight (as of Build 5365) trusted root authority certificates. The driver signing checks for loading boot drivers is handled by WINLOAD.EXE, whereas signing checks for all other drivers is handled by NTOSKRNL.EXE (which incidentally uses CI.DLL to do the actual checks).

Windows Vista relies on a small set of API functions in the MinCrypt library for driver signing verification:

1. MinCrypL_CheckSignedFile (a.k.a. MinCrypK_CheckSignedFile) verifies that the driver is signed by a trusted certificate authority. It is used by both WINLOAD.EXE and CI.DLL.

2. MinCrypL_CheckImageHash verifies that the driver matches image hashes in the signed catalog. It is used by WINLOAD.EXE.

3. MinCryptK_FindPageHashesInCatalog checks the code pages of a process or driver at runtime. Used by CI.DLL.

1) MinCrypL_CheckSignedFile/MinCrypK_CheckSignedFile

MinCrypL_CheckSignedFile eventually calls MinCryptVerifySignedDataLMode which parses and verifies the certificate. MinCryptVerifySignedDataLMode calls MinCryptVerifyCertificateWithRootInfo to extend the chain-of-trust to a root certificate authority. This function will use:

```
ROOT_INFO
*I_MinCryptFindRootByKey(CRYPTOAPI_BLOB *)
ROOT_INFO *I_MinCryptFindRootByName(CRYPTOAPI_BLOB *)
```

Both I_MinCryptFindRootByKey and I_MinCryptFindRootByName iterate across an array of trusted roots until a match is found.

```
CheckNextRootTableEntry:
cmp esi, [ebx-4]
jnz short GetNextRootTableEntry
mov esi, [edi]
push esi
push dword ptr [ebx]
push dword ptr [edi+4]
call _RtlCompareMemory@12 ;
RtlCompareMemory(x,x,x)
cmp eax, esi
jz short FoundTrustedRoot
GetNextRootTableEntry
inc [ebp+var_4]
mov eax, [ebp+var_4]
add ebx, 14h
cmp eax, ds:?ulRoots@@3KB ; ulong const ulRoots
jb short CheckNextRootTableEntry
```

FoundTrustedRoot:
mov eax, [ebp+var_4]
imul eax, 14h
add eax, offset RootTable ; ROOT_INFO const *RootTable
return eax ; return RooTable[Index] (the matched trusted root)
Microsoft Authenticode(tm) Root Authority
Microsoft Root Authority
Microsoft Root Certificate Authority
Microsoft Code Verification Root
Microsoft Test Root Authority
VeriSign Commercial Software Publishers CA
MS Protected Media Test Root
Microsoft Digital Media Authority 2005
Since the trusted root certificate authorities are fixed in the code, attack vectors that require expanding the set of the trusted root certificates, ala Internet Explorer attacks, do not work. Adding a new certificate authority under HKEY_LOCAL_MACHINE\Policies\Microsoft\SystemCertificates or HKEY_LOCAL_MACHINE\Software\Microsoft\EnterpriseCertificates has no effect on driver signing.
If it is signed by a trusted root certificate authority, then the final step is to verify the actual signature—this is done by MinCryptVerifySignedHash. MinCryptVerifySignedHash performs the following sequence:

1. MinAsn1ParsePublicKeyInfo
2. MinAsn1ParseRSAPublicKey
3. I_ConvertParsedRSAPubKeyToBSafePubKey

4. I_VerifyPKCS1SigningFormat (which does the actual memory comparison)

If the comparison matches, it returns success, otherwise it returns an error.
2) MinCrypL_CheckImageHash

MinCryptL_CheckImageHash is quite simple. It walks a linked list of signed catalogs pointed to by g_CatalogList (which is a LIST_ENTRY structure) and calls I_CheckImageHashInCatalog to try to match the image hash in the signed catalog. If the image hash is found in one of the signed catalogs, it returns success, otherwise it returns an error.

3) MinCrypK_FindPageHashesInCatalog

MinCryptL_CheckImageHash is also quite simple. It does a binary search for a matching page hash in ntpe.cat, nt5.cat, or ntph.cat.

## PATCHGUARD

The researcher only provides a brief introduction to PatchGuard here, however that research is no longer current as of the more recent Windows Vista builds. PatchGuard was designed to prevent kernel patching on Windows Vista 64-bit edition. It protects the kernel by periodically checking and validating certain important data structures and core OS images. 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 bugcheck (which will obviously cause the user to lose all unsaved data): . [6]

> CRITICAL_STRUCTURE_CORRUPTION (109)
> This bugcheck 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.msp x
> 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

Although integrity checks and driver signing requirements can be disabled, PatchGuard cannot. Even when integrity checks are disabled, PatchGuard is still active.

## PatchGuard Detection

There are many options for detecting the PatchGuard thread as described in [6]. Two other methods we propose to locate PatchGuard (which may likewise be addressed before the public release) are:

1. Walk the KiTimerListHead

This is an enhanced version of an idea proposed in [6]. 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. In [6], the authors proposed walking the list of registered timer events to find an entry with an invalid DeferredContext. 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. The authors of [6] did not provide any good mechanism to discover this variable in memory and instead proposed

some rudimentary heuristics. We propose a more 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
// 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

First, add a memory read breakpoint (using the Intel debug registers) on IDT entry 1. Second, 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.

Third, 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.

## V. DISABLING \DEVICE\PHYSICALMEMORY

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. In the Phrack article [11], Crazylord demonstrated how to use \Device\PhysicalMemory to access to system memory and manipulate large parts of the kernel that are resident in non-paged memory. 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.

## VECTORS OF ATTACK

### A. Kernel-Mode Network Drivers

Windows Vista handles several network protocols in the kernel through the use of kernel mode drivers. If vulnerability is discovered in one of these signed network drivers, the vulnerability could allow full machine compromise from a remote attacker. Such an attack is not precluded by Vista's driver signing. Some examples of signed drivers handling network protocols in Windows Vista include:

- NETIO.SYS handles the new Vista integrated IPv4/IPv6 network stack
- HTTP.SYS handles HTTP requests
- MRXSMB10.SYS handles SMB version 1 (used prior to Windows Vista)
- MRXSMB20.SYS handles SMB version 2 (new for Windows Vista)
- MRXDAV.sys handles WebDAV
- MSRPC.sys handles MS RPC

For the purposes of this paper, I did not perform an exhaustive analysis of the Windows Vista protocol drivers in order to identify security vulnerabilities. This exercise may be beneficial in the future in order to identify alternate mechanisms that may be used to execute within the Windows Vista kernel.

### B. Disabling Driver Signing and Code Integrity

The most straight forward way to evade driver signing restrictions is to simply patch the on-disk executable files and disable the checks entirely. To load unsigned drivers at runtime, NTOSKRNL.EXE needs to be patched. However, patching NTOSKRNL.EXE will invalidate its digital signature, so that WINLOAD.EXE will refuse to load it. Therefore, WINLOAD.EXE will also need to be patched.

One barrier that immediately becomes apparent when an attempt is made to patch these images is Windows Resource

Protection. Windows Resource Protection (WRP) sets ACLs on system files so that they are not writable by Administrator or LocalSystem—only the TrustedInstaller has write access. However, because Administrators and LocalSystem both have sufficient privilege to take ownership of securable objects, the steps to evade WRP are to first enable the SeTakeOwnership privilege, secondly take ownership of the WRP-protected file or registry key and finally grant Administrators full access. These steps can be done using the AdjustTokenPrivileges and SetNamedSecurityInfo APIs. After that, the on disk binaries can be patched without inhibition.

Using this technique, we are now able to patch both NTOSKRNL.EXE and WINLOAD.EXE to successfully disable driver signing and code integrity within the Windows Vista kernel with a simple one byte modification.

## CONCLUSION

Researcher has discussed in this paper the numerous security enhancements Windows Vista has added to kernel-mode security. 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 security capabilities:

1. It is possible to disable the driver signing and code integrity capabilities by using binary patches on WINLOAD.EXE and CI.DLL. Patching these files at runtime is quite straightforward; each file requires patching at just a single location. Though these files are protected by Windows Resource Protection (WRP), this can easily be evaded as we have demonstrated in [7].

2. The lack of certificate revocation support in WINLOAD.EXE can easily undermine the benefits of driver signing if the software publishing certificate of a company (signed by one of the trusted root

certificate authorities—currently Microsoft or VeriSign) is stolen, published, or misused by a current or former employee. It should be noted, Microsoft has stated that certificate revocation will be implemented in Windows Vista RC1.

Finally, once the driver signing checks have been disabled, a malicious unsigned driver can now be loaded. This malicious driver could then hook 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—for example, by analyzing the NTFS structures.

## REFERENCES


[1] Microsoft. (2006, January). "Digital Signatures for Kernel Modules on x64-based Systems Running Windows Vista," http://download.microsoft.com/download/9/c/5/9c5b2167-8017-4bae-9fde-d599bac8184a/x64KMSigning.doc

[2] Microsoft. (2005, April). "Benefits of Microsoft Windows x64 Editions," http://download.microsoft.com/download/D/A/A/DAA7245D-E01D-46A4-AB70-3A95ED3F6934/Windowsx64BenefitsWP.doc

[3] Microsoft. "First Look: New Security Features in Windows Vista," TechNet, http://www.microsoft.com/technet/technetmag/issues/2006/05/FirstLook/default.aspx

[4] Microsoft. (2005, April). "Secure Startup—Full Volume Encryption: Technical Overview," WinHEC 2005, http://download.microsoft.com/download/5/D/6/5D6EAF2B-7DDF-476B-93DC-7CF0072878E6/secure-start_tech.doc

[5] Microsoft. “Device\PhysicalMemory Object,” TechNet, http://technet2.microsoft.com/WindowsServer/en/Library/e0f862a3-cf16-4a48-bea5-f2004d12ce351033.mspx?mfr=true
[6] Skape, Skywing. (2005, December, 1). “Bypassing PatchGuard on Windows x64,” Uninformed Volume 3, http://www.uninformed.org/?v=3&a=3&t=txt
[7] M. Conover (2006, March). “Analysis of the Windows Vista Security Model,” http://www.symantec.com/avcenter/reference/Windows_Vista_Security_Model_Analysis.pdf
[8] R. Hyde. “The Art of Assembly Language,” http://webster.cs.ucr.edu/AoA/Windows/HTML/AoATOC.html
[9] M. Russinovich, D. Solomon. Microsoft Windows Internals, Fourth Edition: Microsoft Windows™ 2003, Windows XP, and Windows 2000. Redmond, WA: Microsoft Press, 2005.
[10] Microsoft. Boot Configuration Data Editor Frequently Asked Questions,” TechNet, http://www.microsoft.com/technet/windowsvista/library/85cd5efe-c349-427c-b035-c2719d4af778.mspx
[11] Crazylord. “Playing with Windows /dev/(k)mem,” Phrack Volume 11, Issue 59, http://www.phrack.org/phrack/59/p59-0x10.txt