Stoned Bootkit

Your PC is now Stoned! ..again

The Rise of MBR Rootkits & Bootkits in the Wild

Peter Kleissner
Hacking at Random 2009

Abstract

Stoned Bootkit is a new bootkit attacking all Windows versions from XP up to 7. It is loaded by the BIOS before Windows has started and is memory resident up to the Windows kernel. Thus it has unrestricted access to the entire system.

It gives the user back the control to the system – which was taken away by Windows Vista with its signed driver policy. It allows executing any unsigned driver which can be useful both for device testers and malware developers.

There were previous bootkits in the wild, which have changed the battlefield to a lower level. In the past there were bootkits which have only been dedicated attacks, for example attacking only Vista or Windows 7 64-bit; my bootkit targets now to be the most sophisticated and most wide spread one in the wild in 2010.

Table of Contents

Bootkits

A bootkit is a rootkit that is able to load from a master boot record and persist in memory all the way through the transition to protected mode and the startup of the OS. It's a very interesting type of rootkit.

Robert Hensing about bootkits [2]

The term "bootkit" was originally used by Vipin & Nitin Kumar and can be expressed by bootkit = boot + rootkit. A bootkit is stored in the master boot record (the boot sector) and loaded before the main operating system, and its target is to pwn that operating system. Bootkits hook, patch and modify operating system functions in order to be both executed and at the same time being undetected.

There are code integrity verifications and signed code checks which must be patched that the OS will not detect any change. This was a big point of Windows Vista's security, securing the system by signing and verifying all system files.

Timeline

1987	Stoned, first master boot record (bootkit) virus
1990	Form, common boot sector virus in the early 1990s
August 1, 2005	eEye publishes BootRoot & SysRQ2, presented at BH USA 2005
March 29, 2007	Vbootkit was presented at Black Hat Europe 2007
October 30, 2007	Mebroot bootkit appears in the wild
November 2008	Hibernation File Attack was developed, attacking Windows files
April 2009	Mebroot is updated to work with Windows Vista (non-public info)
May 2009	Kon-Boot is released, a tool to bypass Windows logon
March 4, 2009	Vbootkit 2.0 attacks Windows 7 64-bit and goes open source

It first started with the Stoned virus, which was developed in 1987 in New Zealand. In the 1990s boot viruses were quite common because they were spread over floppy disks. One famous and wide spread boot sector virus was the Forms virus, which is still mentioned in Microsoft knowledge base article 122221.

In the last years bootkits were only dedicated attacks to Windows systems. Vbootkit only attacks Vista, Vbootkit 2.0 only Windows 7 64-bit, BootRoot only Windows XP and so on.

In contrast, Stoned Bootkit attacks all Windows operating systems from Windows XP on, including Vista and 7. Only one single master boot record is necessary for attacking all Windows versions. With its included file system drivers (for all FAT and NTFS file systems) it can load files from the file system without Windows, which makes it basically operating system independent.

Architecture of Stoned Bootkit

Stoned consists of two main parts, the master boot record and additional files stored in the file system. This is completely new – that a bootkit loads its files from hard disk. It removes previous limitations (for example only 63 sectors of code) and abstracts the attack to the pre-OS environment with the payload (executed virus).

Stoned consists of the following parts:

- Modules in the master boot record
- Boot applications stored in the file system or embedded in the MBR
- Plugins out sourced to the file system
- Payload drivers stored in the file system

Optionally the plugins and the payload can be out sourced to unpartitioned raw space on the hard disk, to eliminate any necessary file system access. This is for example required for the TrueCrypt attack where the file system is encrypted and not available at boot time.

An example module is the integrated file system module, which is responsible for handling file system access to FAT and NTFS. Other modules, plugins and boot applications can use the API provided by the modules.

Currently Stoned attacks the following operating systems:

- Windows XP
- Windows Server 2003
- Windows Vista
- Windows Server 2008
- Windows 7

There exists the possibility (and it is intended for the future) to attack other operating systems as well, including Linux. The whole magic is done by using signatures for the operating systems startup files.

Following Windows startup files have to be considered:

	Windows XP	Windows Vista	Environment
Stage 1	ntldr	bootmgr	Real Mode
Stage 2	OS Loader	OS Loader	Protected Mode
Stage 3	-	winload.exe	Protected Mode
Stage 4	Ntoskrnl.exe	Ntoskrnl.exe	Protected Mode + Paging

Windows XP (and Server 2003) uses the NT Loader and its embedded OS Loader for startup (= loaded by the bootloader). Ntldr contains a 16-bit stub and the 32-bit OS Loader.

Windows Vista has split up ntldr into bootmgr, winload.exe and winresume.exe. Winload.exe loads the kernel, and winresume.exe resumes from hibernation.

Everything starts with the interrupt 13h service handler. Interrupt 13h is responsible for handling disk I/O access (reading and writing sectors). The int 13h is hooked by overwriting the respective vector in the interrupt vector table (IVT) to point to our own handler. This custom handler forwards the call to the original interrupt 13h service handler and then checks the read contents for the stage 1 signatures.

There are two signatures required for stage 1:

1. Overwriting code to hook a function in order to get called in protected mode
2. Patching the code integrity verification, otherwise Windows would detect the checksum mismatch

For patching the code integrity there are different possible concepts:

- Re-calculating the checksum and storing it in the PE header in memory

  PE Optional Header +64, CheckSum

  The image file checksum. The algorithm for computing the checksum is incorporated into IMAGHELP.DLL. The following are checked for validation at load time: all drivers, any DLL loaded at boot time, and any DLL that is loaded into a critical Windows process.

  Modified RFC 1071 Implementation, [3]

- Patching the conditional jump instruction that decides about match or mismatch

```
mov eax,[ebp-24h]          ; get checksum of PE header
cmp ebx,eax                ; compare against calculated one
je Checksum_Valid
; invalid code handling comes here
[…]
Checksum_Valid:
```

- Patching the return code of the code integrity function to return STATUS_SUCCESS (0) instead of STATUS_IMAGE_CHECKSUM_MISMATCH (0C0000221h)

```
mov eax, 0C0000221h
cmp ebx,eax                ; compare against calculated one
je Checksum_Valid
; invalid code handling comes here
[…]
Checksum_Valid:
```

The third method is the best one, it prevents crashing, patching the wrong jumps, prevents overwriting other instructions and is most reliable. I have first seen it used in the Sinowal version from April 2009.

Stage 1 is necessary for executing the protected mode code, and stage 2 (and 3) is necessary for getting called in order to relocate the code to paged memory. This means the code must be executed and accessible in the running kernel when paging is enabled. This is done by copying the code to the end of the ntoskrnl image. The algorithm is to take (ntoskrnl.exe image base + SizeOfImage + 00000FFFh) & 0FFFFF000h – size of bootkit. The FFs are used for aligning to page boundaries (4 KB).

Also ntoskrnl.exe must be patched to execute the relocated kernel code (and furthermore the verification if ntoskrnl.exe is valid in winload.exe must be patched).

Further it is important to mention any payload. Because the bootkit is executed under the Windows kernel it can simply use the provided API to load any additional code from the file system (including plugins etc.). Mebroot (Sinowal) for example keeps its payload stored at unpartitioned space on the hard disk.

You can also out source a driver loader which does the PE relocation and resolving of the PE image, in order to be totally undetectable by anti-virus scanners. This is useful when using the Windows Driver Kit for high-level driver development.

Sinowal Bootkit

Since this is hacking at random I also want to cover the Sinowal bootkit. I once wrote an article back in 2008 [4], which let me to work together with high security industrials which I personally admire and respect in their work.

The old Sinowal bootkit only consists of the sectored master boot record:

Sector 0	Bootloader, Interrupt 13h Hook Code
Sector 60	Ntldr Hook Code
Sector 61	Kernel Code (executed directly after Ntoskrnl)
Sector 62	Original Master Boot Record Backup

All other sectors remain zero and unused. The total size of the master boot record is 63 sectors (7E00h bytes).

Additionally the kernel driver is stored at the end of the hard disk and a fixed reference (sector number) to the location of the kernel driver is patched in the bootkit when infecting the machine.

Sector 0: Master Boot Record

Sector 0 contains the malicious boot code which does some initialization and hooks the interrupt 13h. It relocates itself to the end of real mode memory (~ 1 MB, mostly 9F400h to be exact) and contains code to hook the ntldr and to read the sectors 60 and 61 into the memory. It loads the original bootloader from sector 62 and executes it.

Offset 1B5h	3 bytes for language message descriptions (unused)
Offset 440	Microsofts Disk Signature
Offset 1BEh	Partition Table
Offset 510	Boot Signature

Sector 60: Ntoskrnl hooking

The sector 60 contains code that is executed by the installed hook to ntldr, and does nothing more than copying sector 61 to the end of the ntoskrnl image and hooking ntoskrnl.

Sector 61: Kernel Code

The code in sector 61 allocates memory using ExAllocatePool() and copies itself to the allocated region. It then loads the kernel driver from the hard disk and starts execution of Sinowal. After successful execution it deletes itself from memory by overwriting itself and the driver with zeroes, and by deallocating used memory back to the system. Thus it leaves no traces in memory.

Detailed description of the bootloader:

1. Initialize registers
2. Copy itself to the end of memory
3. Reading further virus data, sector 60 and sector 61
4. Hooking interrupt 13h
5. Loading original Microsoft MBR from the sector 62 to 7C00h and execute it
6. Interrupt 13h hook checks if Read Sectors or if Extended Read Sectors functions are requested and executes them
7. Check the read buffer for ntldr, to inject jump code to sector 60
8. Check read buffer for ntldr, to overwrite code integrity verification code with nops (another method to bypass code integrity verification)

Previously Windows Vista wasn't affected by the old Sinowal variant because of different startup dependencies (different startup files and mechanism). However there was an overhauled version in April 2009 which now attacks Windows Vista too (non-public, still not reported).

Following code patterns are used by Mebroot:

```
Signature:      8B F0 85 F6 74 21/22 80 3D
To be in:       ntldr
Is at offset:   +26B9Fh

Signature:      83 C4 02 E9 00 00 E9 FD FF
To be in:       ntldr
Is at offset:   +1C81h, +1C9Ch

Signature:      C7 46 34 00 40 ... A1
To be in:       ntldr
Is at offset:   +19A44h, and A1 located at +19A51h
```

```
Ntoskrnl is scanned for the following code patterns:

  + 6A 4B 6A 19 89/E8 ?? ?? ?? ?? ?? ?? E8/??
    Ntoskrnl +01CE87E0h
    Memory   +80683EC9h

  + E8 ?? ?? ?? ?? 84 C0
    Ntoskrnl +01CE87F3h          Ntoskrnl +1CE87F8h
    Memory   +80683ED8h          Memory   +80683EDDh
```

Yes, these signatures are the magic behind Mebroot. Take the knowledge, use it.

The source code of Mebroot is available to the public [5].

Bootkits in the Wild

There are a variety of usage methods of bootkits:

Sinowal	Stealing banking information
Kon-Boot	Bypassing Windows logon
Stoned.A	Keeping the user happy with text and sound messages :)
Vista Loader	Using AntiWPA

A very good example of using Stoned is the extraction of the unpacked driver of Sinowal. It traces memory operations and monitors the unpacking of Sinowal and writes out the unpacked driver. The Sinowal Extractor uses the Stoned Bootkit to get loaded and to install hooks to ntoskrnl by overwriting export RVAs.

Stoned Bootkit is interesting for:

- Security Researchers (AV industry)
- Law enforcement agencies – Stoned bypasses full volume encryption
- Black Hats, Stoned can be used for malware to get loaded (undetected!) to the Windows kernel for any Windows OS (gaining full rights, staying under the radar..)
- Home & personal use, bypassing Windows logon and Anti Windows Product Activation

A free development framework of Stoned is available on its project site [7].

Anti Windows Product Activation

In the wild there are some different Anti Windows Product Activation (AntiWPA) methods available; I will discuss the most interesting and sophisticated one here (others are time stopper, re-arm and some other techniques).

Microsoft has a secret arrangement with OEM hardware manufacturers to include a secret additional ACPI table to identify the system as OEM and activating it without any need of online activation. The background is that they can ship out PCs without

activating every single one online, which was previously handled by using corporate keys (which do not exist for Vista and 7 in the same way anymore).

The list of hardware manufacturers include: Acer, ASUS, Dell, Fujitsu Siemens, Gateway, HP, Lenovo, Medion, NEC, Sony, Sotec, Toshiba, MSI

Microsoft is using SLP 2.0 (System Locked Preinstallation) technology for activation process of OEM (Original Equipment Manufacturer) edition of Windows Vista on branded PC. One of the requirement to activate Windows Vista with OEM product key is the existence of SLP public key and SLP marker which stored in SLIC (Software Licensing Internal Code) table in ACPI.

Betalog.com, [8]

When activating Windows Vista with an OEM license it checks whether the SLIC table is present and uses it together with a digital certificate. There is no public description available of this SLIC table, not even any confirmation in any official ACPI document. However, there exists a source code file (`actbl1.h`) from Intel (which is also part of the arrangement) that reveals that table:

```
* Name: actbl1.h - Additional ACPI table definitions

 * Copyright (C) 2000 - 2008, Intel Corp.
 * All rights reserved.

#define ACPI_SIG_SLIC                   "SLIC"   /* Software Licensing
Description Table */
```

According to the Advanced Configuration and Power Interface specification (chapter "ACPI Software Programming Model") there is the Root System Description Pointer (RSDP) structure that is set up by the firmware (= BIOS) and contains the address of the Extended System Description Table (XSDT) which references other description tables. It is supposed that the XSDT also contains a pointer to the SLIC table, so this is the point where we have to look and inject the table at.

The XSDT contains an array with pointers to those tables. Each system description table is built up from a header `DESCRIPTION_HEADER` and of its data following. It is easy to find the RSDP and so on the XSDT, the RSDP is statically pointed to by a variable at 40Eh in memory (excerpt from the ACPI documentation):

```
The first 1 KB of the Extended BIOS Data Area (EBDA). For EISA or MCA
systems, the EBDA can be found in the two-byte location 40:0Eh on the
BIOS data area. The BIOS read-only memory space between 0E0000h and
0FFFFFh.
```

RSDP + 24 points to the XSDT, and XSDT +36 contains an array with pointers to the description tables. So all tables are revealed – and where to add the SLIC table. There is software available that lists the ACPI tables (and the contents of the `DESCRIPTION_HEADER`), the particular SLIC table reveals following information:

```
ACPI Signature SLIC
Table Description Software Licensing Description Table
```

```
Memory Address 7FEE8F30h
Table Length 374 bytes
OEM ID _ASUS_
OEM Table ID Notebook
OEM Revision 11000624h
Creator ID MSFT
Creator Revision 00000097h
```

Putting that information together, only the SLIC table has to be added, and all the other ACPI tables are well documented. The SLIC table contains the public key and the certificate must match to the SLIC table data. If both the SLIC table and the certificate are present, Windows Vista and 7 will accept the OEM key and no internet connection is necessary for activation =). It is as simple as that.

The `SLIC.bin` file contains the basic SLIC table (always 374 bytes). Following certificates and OEM IDs are necessary (must be patched to `SLIC table +CCh`):

Acer	`ACRSYSACRPRDCT`	acer.xml
ASUS	`_ASUS_Notebook`	asus.xrm-ms
DELL	`DELL  M07`	dell.xrm-ms
Fujitsu Siemens	`FSC  PC`	
Gateway	`GATEWASYSTEM`	gateway.xrm-ms
HP	`HPQOEMSLIC-MPC`	hp.xrm-ms
Lenovo	`LENOVOTC-2P`	lenovo.xrm-ms
Medion	`MEDIONMEDIONAG`	
NEC	`NEC ND000146`	nec.xrm-ms
Sony	`Sony  VAIO`	sony.xmr-ms
Sotec	`SOTEC SOTECDT`	
Toshiba	`TOSQCITOSQCI00`	

The OEM IDs are padded with white spaces (20h) and zeros (00h) to fit to 14 characters. These OEM tables are a very nice idea; however it has not been publicly documented. Security by obscurity is not a good idea, especially here where it reveals the possibility of modifying and adding the SLIC table at runtime.

The SLIC table for my DELL Notebook:

```
00000000  53 4C 49 43 76 01 00 00 01 47 44 45 4C 4C 20 20  SLICv....GDELL
00000010  4D 30 37 20 20 20 20 00 12 0C D6 27 41 53 4C 20  M07    ...Ö'ASL
00000020  61 00 00 00 00 00 00 00 9C 00 00 00 06 02 00 00  a.......œ.......
00000030  00 24 00 00 52 53 41 31 00 04 00 00 01 00 01 00  .$..RSA1........
00000040  7F F6 C1 05 BE 5C 57 63 A5 8A 68 F3 6E 8F 06 FA  .öÁ.¾\Wc¥Šhón..ú
00000050  AF B4 9F 68 82 23 EC 50 40 5A 73 7F EC E4 07 CB  ¯´Ÿh‚#ìP@Zs.ìä.Ë
00000060  DC 25 1A 9C E3 E3 66 11 E0 A5 98 06 C5 80 0A FA  Ü%.œããf.à¥˜.Å€.ú
00000070  42 93 86 98 E7 D5 1B D4 D7 3A A4 0B EE E2 7D BE  B"†˜çÕ.Ô×:¤.îâ}¾
00000080  5F 5B 15 0C AB D0 21 DE BF E9 B5 6E A4 57 B9 8C  _[..«Ð!Þ¿éµn¤W¹Œ
00000090  0C D2 BA 3A 69 30 76 94 71 A2 64 D7 4C D8 85 BF  .Òº:i0v"q¢d×LØ…¿
000000A0  DF A5 6A C8 DC 45 D5 4D 8C B8 8C 05 2F FC 2E 23  ß¥jÈÜEÕMŒ¸Œ./ü.#
000000B0  C4 29 C5 6F 3F 29 6C 6D 57 79 0E B6 75 ED 21 95  Ä)Åo?)lmWy.¶uí!•
000000C0  01 00 00 00 B6 00 00 00 00 00 02 00 44 45 4C 4C  ....¶.......DELL
000000D0  20 20 4D 30 37 20 20 20 20 00 57 49 4E 44 4F 57    M07    .WINDOW
000000E0  53 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00  S ..............
000000F0  00 00 00 00 00 00 51 E9 A5 CD 35 30 91 B0 9B C0  ......Qé¥Í50'°›À
00000100  CE 05 FA 26 B5 43 29 40 1C 13 16 EF E3 BF 17 2F  Î.ú&µC)@...ïã¿./
```

```
00000110  BD 3B 99 B5 6E 23 49 F7 97 BC ED FF C9 4A 95 F4  ½;™µn#I÷—¼íÿÉJ•ô
00000120  A5 CD 33 0B 40 2E C8 E1 8B E6 8F B6 74 8E 94 43  ¥Í3.@.Èá‹æ.¶tŽ"C
00000130  E0 2F B6 CE 53 F0 09 3D B4 18 0F 44 23 10 64 F3  à/¶ÎSð.=´..D#.dó
00000140  74 06 2E 1D 00 71 13 6A C7 C9 9E 82 CB 71 09 B1  t....q.jÇÉž‚Ëq.±
00000150  9E 42 5A 7D F3 F8 CC D1 FD 22 90 BF 37 3E 2C 68  žBZ}óøÌÑý".¿7>,h
00000160  BB 30 FF 84 0F B5 2B B3 C0 7A 71 44 C5 EB 13 15  »0ÿ„.µ+³ÀzqDÅë..
00000170  C3 CA 66 1B 80 2E                                 ÃÊf.€.
```

You see there clearly the "RSA1" keyword, which means that the RSA algorithm is used with the public key following from 40h up to C0h; so the size of the key = 80h = 128 bytes = 1024 bit. The OEM identifier is stored at CCh up to D9h.

A secure way would be using the TPM to verify the startup, which implies that the startup was not hooked and thus no modification of any ACPI table occurred. There is the program Vista Loader in the wild which can be considered as a bootkit using this AntiWPA technique. It adds add runtime the SLIC tale to the BIOS (the BIOS is writable at runtime in memory), and spoofs an OEM BIOS which allows using the OEM key together with the certificate.

Conclusion

Bootkits are currently getting more and more attention. The security flaw is by design, there is no fix and it cannot be fixed. However the computer's startup can be verified by using the Trusted Platform Module in connection with full volume encryption, which would break down bootkits. If the bootkit was detected (using TPM) and the key for full volume decryption cannot be unlocked then the OS will not be able to start (and thus the system useless). There is quite much potential left in bootkits, so I am sure that we will see a lot of movement in this field in the future.

The next target of Stoned is to add Linux kernel support and for me to do research work with the Trusted Platform Module. There was once TPMkit by Nitin & Vipin Kumar which should be presented at Black Hat USA 2007, however, which has been withdrawn from there by the authors for unknown purposes.

References

[1] Your Computer is Now Stoned (...Again!): The Rise of MBR Rootkits
Elia Florio (Symantec) and Kimmo Kasslin (F-Secure)
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/your_computer_is_now_stoned.pdf

[2] VBootkit vs. Bitlocker in TPM mode
Robert Hensing's Blog
http://blogs.technet.com/robert_hensing/archive/2007/04/05/vbootkit-vs-bitlocker-in-tpm-mode.aspx

[3] An Analysis of the Windows PE Checksum Algorithm
Jeffrey Walton
http://www.codeproject.com/KB/cpp/PEChecksum.aspx

[4] Analysis of Sinowal
Paul Kleissner
http://web17.webbpro.de/index.php?page=analysis-of-sinowal

[5] Mebroot Source Code
http://web17.webbpro.de/downloads/Sinowal%20Article/Sinowal%20Source%20Code.zip

[6] Anti-Sinowal strategies and Sinowal Bootkit Extractor
www.bootkitanalytics.com

[7] Stoned Bootkit Project Site
www.stoned-vienna.com

[8] Improved Way to Add SLIC (SLP 2.0) Table into BIOS ACPI to Activate Windows Vista OEM
http://www.betalog.com/read.php/152.htm