18/12/2011

Sinowal: the evolution of MBR Rootkit continues

Andrea Allievi
Advanced Malware Researcher
andrea.allievi@prevxresearch.com

Introduction

In these last weeks of year here at PrevX labs we found an interesting malware sample called Sinowal.knf. This is the last evolution of famous MBR rootkit that begun its spreading in the year 2008. Rootkit in this incarnation has evolved a lot. We start speaking about its way of starting up…

MBR Code – First stage of control

The MBR part of infection is different from the original MBR Rootkit because it uses another approach in Int13 hooking. The int13 hook part of infection is actually not in MBR.
Infected Mbr, after has read Volume Boot Record of Startup partition, it calculates rootkit sectors number with this Algorithm:

```
DWORD GetRootkitSector(PMASTER_BOOT_RECORD mbr) {
    PARTITION_ELEMENT * curPart = &mbr->PartitionTable.Primary1;
    DWORD rootkitSect = 0;
    for (int i = 0; i < 4; i++) {
      if (rootkitSect < curPart[i].FreeSectorsBefore) {
            rootkitSect = curPart[i].FreeSectorsBefore;
            rootkitSect += curPart[i].TotalSectors;
      }
    }

    return rootkitSect;
}
```

This is the first difference from the original MBR Rootkit. This time infected sector are placed immediately after the partitions space, not at the end of disk. Infected Mbr reads 9 sectors (at calculated offset of hard Disk) at real mode memory address 0000:7E00. It then looks if rootkit is correctly installed in system checking for byte pattern 75 2F F3 A4 starting at 12th bytes of second sector just read. This is the rootkit signature.
These 9 sectors are organized in this way: First sector is the original System Master Boot Record, and others 8 ones are rootkit startup code.
Rootkit startup code sectors are made by a mix of real mode 16 bit and 32 bit protected mode code.
Execution is then transferred at address 0000:8000 (second rootkit sector). Here is where infection starts. First of all, Mbr memory (at address 0000:7C00) is filled with original Master sector (that is in the first of 9 rootkit sectors). Then rootkit installs Int13 hook handler, relocating itself at memory address 9F00:0000.
At this stage execution is released to original Mbr.

```
                mov     cx, 9
                add     bx, 200h
                call    ReadFromDisk    ; Read first 9 Sectors after this entire bootable
                jb      short ReleaseControlToVbr                   partition at 0:0x7E00
                mov     si, bx
                add     si, 20Ch
                lea     dx, [si-0Ch]
                cmp     dword ptr [si], 0A4F32F75h ; Check if read sectors are infected
                jz      short loc_9E

ReleaseControlToVbr:                    ; CODE XREF: seg000:0079↑j
                                        ; seg000:0085↑j ...
                jmp     far ptr 0:7C00h
; ---------------------------------------------------------------------------

loc_9E:                                 ; CODE XREF: seg000:0097↑j
                mov     si, bx          ; SI = 0x7E00
                call    dx              ; Call Virus Entry point
                jmp     short ReleaseControlToVbr
```

Figure 1. Sinowal Mbr code that reads its sectors and checks if are correctly infected

Real Mode Rootkit Int 0x13 Handler

Rootkit Int13 handler does quite the same stuff as the previous version of MBR Rootkit. It intercepts BIOS "Read Sector" and "Extended Read Sector" functions to see if NTLDR (intended as NTLDR startup Code + OSLOADER.EXE) is loaded, and then it forges NTLDR in 2 points:

1. NTLDR Calls to integrity checks function are deleted (filled with many NOP instruction)
2. OSLOADER main startup function *BlOsLoader* is forged after the call to *BlLoadBootDrivers*. In this way rootkit 32 bit code is executed after all boot drivers are initialized (Disk port driver included, we will see why this is important next in analysis…)

After these 2 steps are performed successfully rootkit real mode code restores original Int13 Handler and return execution normally to Operation System Loader.

32 Bit rootkit OSLOADER code

After NT Loader has initialized all system boot drivers, as we have just seen, execution is transferred to 32 bit rootkit code (at the end of second sector). This code performs the following steps:

1. Restore the modification done on *BlOsLoader*
2. Search in Osloader memory to find *BlLoaderBlock* symbol that is used by loader to maintain various System information and also loaded driver list.
3. With *BlLoaderBlock* symbol it gets Nt Kernel base address and it start searching and modifying memory of NT Kernel: it searches for *IoInitSystem* function call in *Phase1Initialization* kernel function and replace it with a call to relocated *NewIoInitFunction* rootkit procedure. Rootkit indeed relocate part of its code 1024 Bytes before the start of Nt Kernel address space (in our test systems this address is mostly 0x806d9c00) and finally zeroes out its original loader code.
4. At this stage execution is release to original Os Loader and then to Nt Kernel.

```
mov     al, 0A1h ; 'i'  ; Search  "mov eax,dword ptr [osloader!BlLoaderBlock]"
scasb
jnz     short loc_9F17A
mov     eax, [edi]      ; EAX = BlLoaderBlock pointer
mov     eax, [eax]      ; EAX = BlLoaderBlock->NextEntry
mov     eax, [eax]      ; EAX = curItem->NextEntry
mov     edi, [eax+18h]  ; EDI = curItem->BaseAddr
mov     ecx, [edi+3Ch]  ; ECX = curPe.e_lfanew
mov     ecx, [ecx+edi+50h] ; ECX = ntHdr->optHdr.SizeOfImage
call    GetIoInitSystemRelAddress
jnz     short Exit
```

Figure 2. Sinowal loader code that get NT Kernel Base address and IoInitSystem function address

Rootkit kernel mode code

NT Kernel begins its execution normally in function *KiSystemStartup*. *KiSystemStartup* orchestrate all system initialization process: executive, Hal, kernel tables like processors IDT and various kernel subsystems. Phase 0 of boot process proceeds normally (see Windows Internals book for details), Phase 1 too, but until initialization of I/O subsystem. System at this point calls rootkit *NewIoInitSystem* function. This implements a quite interesting obfuscation trick. First of all, as seen in the previous modifications, it restores original Kernel function modification. Then it places a special return address on the stack. In this way after original *IoInitSystem* function ends its life, execution will be diverted on another rootkit procedure. We will see afterward what this important function does.

Rootkit *NewIoInitSystem* now opens KeLoaderBlock internal variable, gets Boot Driver list, cycles between loaded driver and get system Disk port driver entry point. How does rootkit know which is real Disk Port Driver? It calculate a simple hash of its name and compares it with a hardcoded one (calculated by dropper at installation time). If the 2 hashes match then driver is the right one. After it has found right driver, it apply a deviation in target driver entry point function and returns execution at original *IoInitSystem* function.

IoInitSystem procedure has in its body a call to *IopInitializeBootDriver.* This procedure loads all 32 bit boot driver and calls for each one its EntryPoint function. As the reader has just guess, when the Disk port driver entry point is called, the execution is again diverted to rootkit **Disk Port driver new entry point**.

After *IoInitSystem* terminates its own work, the most important rootkit function is also called. ***AllocateAndInstallRootkitDrv***, as the name implies, has the main rule in startup procedure: it extracts and installs real protected Sinowal Driver.

Sinowal new Disk Port Driver Entry point

Rootkit new Port driver entry point function does nothing special. It restores original driver entry point modification and then it uses obfuscation similar to *NewIoInitSystem* one: original driver entry point return address is changed to point to a selected rootkit function. In this manner execution first return to original driver initialization procedure, and, at the end, when driver is completely operative, to "*InstallPortDrvDeviation*" rootkit function.

InstallPortDrvDeviation rootkit procedure has the responsibility to apply first Disk Port Driver IRP hook. Indeed it gets Driver IRP_MJ_SCSI Major Function pointer, save it, and replace with a forged one. This forged new Irp handler does nothing except call original, just saved, handler.
I would like to remember the reader that IRP_MJ_SCSI handler is responsible to manage SCSI Packets and is the lower procedure that administrates disk I/O, talking directly with real hardware (mostly with IN and OUT machine instruction).

With the aim to install new Irp Handler, rootkit relocates itself calling *ExAllocatePool* Nt function. Before continue in analysis we have to speak about Rootkit API resolver procedure. Indeed rootkit startup code doesn't know which address corresponds to each individual API exported by Ntoskrnl. To resolve them rootkit uses a clever method…

GetNtKrnlAndResolveFuncs rootkit procedure gets, from main processor IDT, Interrupt service routine number 0 address, does some mathematical operations and gets Nt kernel base address. Now it has to resolve 5 needed exported functions. To do so it starts parsing Nt kernel export address table: for each exported function name it calculates a simple hash (like *NewIoInitSystem* procedure) and compares it with a hardcoded one. If 2 hashes match then exported function address is copied from Nt Export Table to Rootkit import table.

```
dd 3707E062h            ; ExAllocatePool function hash
dd 9D489D1Fh            ; ExFreePool function hash
dd 58586D92h            ; KeDelayExecutionThread function hash
dd 0DCD44C5Fh           ; NtClose function hash
dd 3888F9Dh             ; NtCreateFile function hash
dd 84FCD516h            ; NtReadFile function hash
```

Figure 3. Sinowal startup code small Import Address Table

AllocateAndInstallRootkitDrv Rootkit function

This is the most important piece of rootkit startup code. Its purpose is to extract and run real Rootkit Driver and starts its execution after *IoInitSystem* has done its job...
First of all, if rootkit IAT hasn't already been resolved, it resolves Nt Functions address and builds its IAT.
It then builds data structures needed to open a handle to boot disk device, and gets a handle to it with *NtCreateFile*. Now it reads original system Mbr. From system partition table it calculates rootkit sectors number in the same manner as seen in infected Mbr, and adds 25. The resulting number is where real rootkit driver is stored. Loader code reads its first 16 sectors (8 Kbytes), analyzes its PE Header and then allocates 2 kernel buffers large as *SizeOfImage* PE attribute. Loader code reads entire driver from disk (512 sectors, 256 Kbytes) and places it in first buffer, in second buffer instead it stores relocate driver, correcting pointers,

rebasing each section, resolving driver IAT,… At the end, when all this stuff is done, the second buffer is freed with *ExFreePool*, and rootkit driver entry point is called. Rootkit driver begins its life now, and does a lot of stuff… but this is another story… (not covered in this analysis)

```
                add     edx, esi        ; EDX = ntHeaders
                mov     ebx, esi        ; EBX = drvBaseAddr
                mov     ecx, [edx+54h]  ; ECX = SizeOfHeaders
                mov     edi, [esp+4]    ; EDI = First Allocated Buffer
                pusha
                rep movsb               ; Copy headers to first buffer
                popa
                movzx   ecx, word ptr [edx+6] ; ECX = NumberOfSections
                push    edx
                add     edx, 0F8h ; '°' ; EDX = ImageFirstSection(ntHeaders)

CopySection:                            ; CODE XREF: AllocateAndStartRootkitDrv+1A1↓j
                pusha
                add     esi, [edx+14h]  ; ESI = thisSect.RawAddr Real Offset
                add     edi, [edx+0Ch]  ; EDI = thisSect.VirtualAddress
                mov     ecx, [edx+10h]  ; ECX = thisSect.RawSize
                jecxz   short NextSection
                rep movsb               ; Copy this section in right place

NextSection:                            ; CODE XREF: AllocateAndStartRootkitDrv+199↑j
                popa
                add     edx, 28h ; '('  ; Move to next Section
                loop    CopySection
```

Figure 4. Rootkit loader that relocate its main driver

After rootkit driver is initialized, loader code ends its life: it frees used resources and closes handle to system disk device. Its job is now **done**: rootkit is loaded and fully functional.

Sinowal Rootkit main driver: a quick glance

Sinowal.knf rootkit driver is a large executable that does quite the same things just seen for the previous MBR Rootkit version. It uses a proprietary packer to hide its real code. The clever interesting thing to analyze is the method it uses to hide its sectors from disk. The technique has really evolved, this time is even more powerful than that one used in TDL rootkit. First inspection of an infected system doesn't reveal any kernel hook or internal deviation. Security utility like Gmer or Rootkit Unhooker doesn't find any kernel alteration, except for a strange driver and some user mode deviation, but no modification of disk class/port driver…

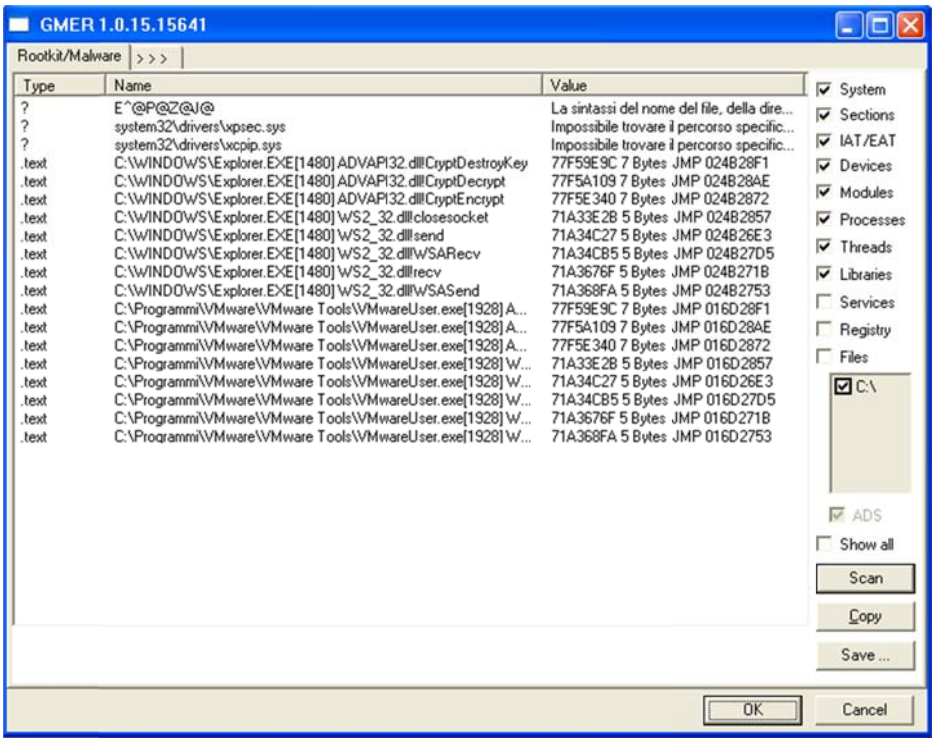

Figure 5. Gmer security utility running on an infected machine.

Noteworthy is the fact that no Disk driver modification is showed

To real show what happen in the infected system is necessary to run Windbg and have a great attention… Also with a debugger is actual very difficult to find out what is wrong on Disk stack. A deep analysis of system and rootkit code has revealed this interesting trick:

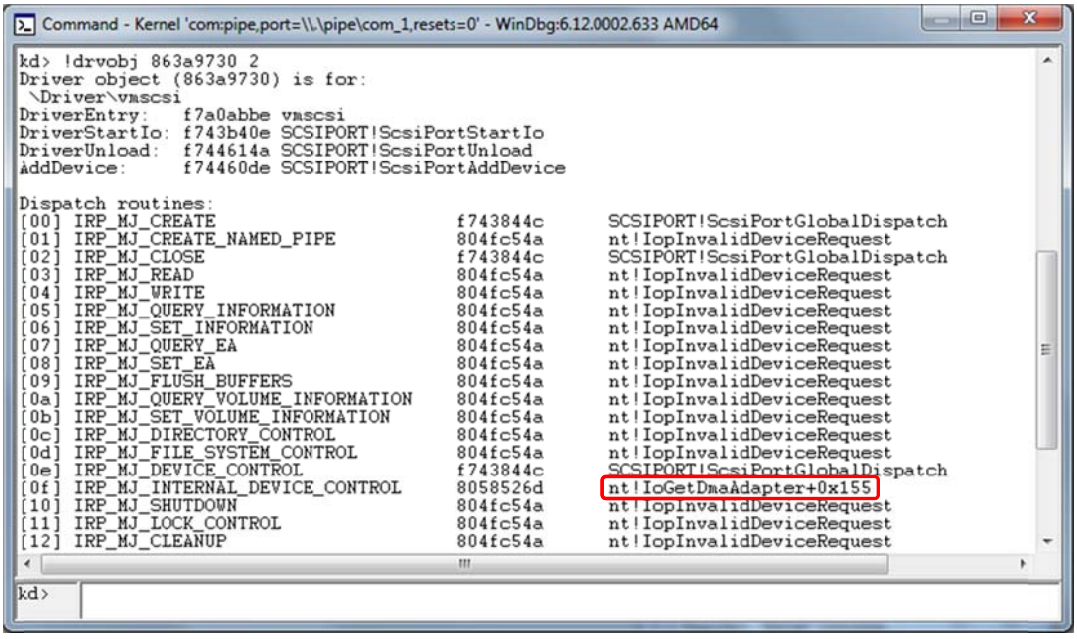

Figure 6. Windbg information gathered from system disk port driver

Disk port IRP_MJ_SCSI (same as IRP_MJ_INTERNAL_DEVICE_CONTROL) handler function point to a Nt kernel function that a further inspection reveals that is totally unrelated to disk driver. What's wrong with this?

```
kd> u 8058526d
nt!IoGetDmaAdapter+0x155:
8058526d ff1574e85480    call    dword ptr [nt!HalDispatchTable+0x3c (8054e874)]
80585273 8945fc          mov     dword ptr [ebp-4],eax
```

The memory pointed by Irp Handler contains a call to an entry in Hal Dispatch Table. Hal Dispatch Table is used by Nt Kernel to memorize address of some important Hal functions. Some entries in this table are not used. If we further inspect about infected system Hal Dispatch Table we encounter something like:

```
kd> dds nt!HalDispatchTable + 38 l3
8054e870  806f32cc hal!HaliInitPowerManagement
8054e874  8631d000
8054e878  806f2d50 hal!HalacpiGetInterruptTranslator
```

Address 0x8631D00 is in rootkit code address space and contains the real rootkit filter routine. Hook an Irp Handler function in this way is a very clever strategy. Indeed it prevents every security software to point out any kernel driver alteration, because the hook pointer is in Nt kernel address space and point to a documented symbol like for example *IopInvalidDeviceRequest*. To correct recognize this type of rootkit it's necessary to implement a really specific and particular strategy…

Conclusions

We have seen an interesting evolution of Mbr Rootkit that, even if uses a quit old strategy to load itself, is still powerful and very hard to find. The main weak point of its implementation is that a simple "`FIXMBR C:`" typed in a recovery console of an infected system can remove the entire rootkit (actually only mbr, but it's obviously that if loader code doesn't get executed, inactive rootkit driver become useless). However real mode Int13 hook has been moved in a different sector form Mbr, and the concept behind is a good choice because rootkit in this way can evade some old BIOS antivirus that recognize this behavior as malicious if done in MBR sector. Also the disk hooking method is still powerful and hard to find…

In this analysis we don't have covered rootkit main driver feature but we have focused only on Startup method. Our remover is able to correct identify and destroy this kind of infection.

WHO AM I

I am Andrea Allievi, nickname AaLl86, an Italian graduated security researcher currently working for PrevX Research team. You can contact me at aall86@altervista.org. If the reader found some language mistakes please contact me, indeed English, is not my native language, and I am secure that this paper is full of grammar errors.

REVISION HISTORY

19/12/2011 - First Release

Andrea Allievi
Advanced Malware Researcher