Using the Boot emulator - Bootkit detection technology

Xcon2013 By nEINEI

Keywords: Bootkit; Bios; MBR; VBR; bootstrap code; polymorphic; Boot Emulator;

Abstraction:

Nowadays Bootkit technology based on Microsoft Windows had been greatly advanced. It had ceased to be some proof of concept (POC) but to be series of stable BootKit families on the wild (Gapz/Rovnix/TDL4/Phanta). And Bootkit's infecting component had expanded from hard disk MBR, to VBR, Bootstrap code and even BIOS chipset. Boot security and kernel-access security were deeply challenged by Bootkit.
Bootkit intercepts into system early in the Windows booting stage, security software can do very little things at that time; it makes Bootkit difficult to remove once infected. So we introduce a new method, by embedding boot emulator into antivirus engine and emulate system booting process to detect known and unknown Bootkit infection.

Introduction

Recent year Bootkit virus' attacking vector were being constantly improved, showing as 1. Hardware oriented infection 2. Bootloader code obfuscation, to make reverse engineering harder 3. Enhanced protection on malware data. These factors cause great difficulties to detection and repair.

BIOS infection is not a concept any longer since eEys' BootRoot project; more and more people are reaching that topic. Peter Kleissner demonstrated the method to bypass Win8 UAC by Bootkit technique on the MalCon, in Nov. 2011. This method is targeted for Win8 booted by BIOS, instead of secure boot by UEFI, indicating that traditional Bootkit thread will exist in the future, until PC BIOS completely replaced by UEFI.

At the same time, a lot of people keep keys on UEFI, not only security researchers but also hackers. Loukas presented an EFI Rootkit on Mac machine, on Black Hat USA 2012; and firmware attacking is also widely interested, e.g. SMM Rootkit; and flashing PCI/ISA module into BIOS may be outdated and gone. Last year security researcher Jonathan Brossard created Rakshasa, a conceptual UEFI Rootkit as hardware backdoor, using modified Coreboot and SeaBIOS open source tools to replace existing BIOS on PC, can harm the operating system on boot time seamlessly. All these research may help Bootkit malware writer in consequence; these techniques can be used by cybercriminals to turn into actual attack at any time.

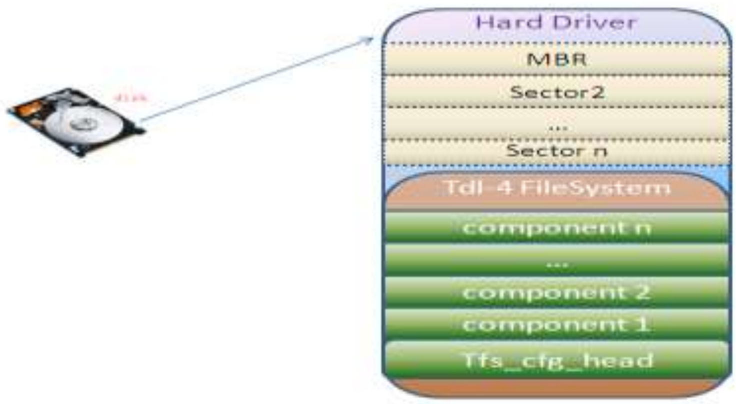
1.Current defense solution

Most defense solution against Bootkit available is to intercept disk operation on MBR/VBR/NT OS Loader/… in kernel mode by HIPS. But most viruses can bypass monitoring by injecting module into normal process, and write to disk in the host process.
Besides this, McAfee's Deep Defender can monitor kernel behavior with the help of hardware security feature; And Kaspersky released KUEFI this year, which could scan file system to detect malware before system boot. Certain detection signature is required for these two methods at the same time.
In this article we will discuss a new solution against Bootkit, which can be embedded into anti-virus engine and signature independent: by simulating Windows boot process to track malicious behavior.

2.The complexity of Bootkit detection

2.1 Self-created file system

Certain encrypted self-created file system is used by complex Bootkit (TDL4/Rovnix) to store its own data. It's very difficult to repair infected computer without decrypting the file system. Let's say TDL4: All component of TDL4 are stored in its own file system, locating in a few sectors at the end of hard disk drive.



(PICTURE 1)

Overall, there's a defined structure Tfs_cfg_head for configuration header, recoding information for all components. Component data is combined by certain sectors' data; every sector data take form in Tfs_base_data structure, which is the base data structure in TDL-4's file system.
Tfs_cfg_head recorded name, size, offset etc. for all components. As a result of our reverse engineering (TDL4 on Jun.2011), the definition of this structure is:

```
struct Tfs_cfg_head
 {
     unsigned char flag [2] ;           // "DC"-0x43 0x44 , this root directory
     unsigned char reserve[4];         //  4 bytes
     Tfs_cfg_section sections [10];    //  Tfs_cfg_section
     unsigned char buffer[512 - 2 - 4 - 10*sizeof(Tfs_cfg_section) ];
 }
```

Tfs_cfg_section defines description information of every component, stored within Tfs_cfg_head, as:

```
struct Tfs_cfg_section
 {
    unsigned char name[16];     //  component name
    unsigned int size;         //  component file size
    unsigned int offset;       //  offset of decrypt file in filesystem
    FILETIME time;             //  time of creation
 }
```

Tfs_base_data stores offset data by sector, saving searching index, position offset data. Defined as:

```
struct Tfs_base_data
 {
    unsigned char header[2]; // base header flag,  "FC" -0x43 0x46 , block with
file data
    unsigned short int  next_offset;  //  Tfs_base_data
    unsigned short int  next_idx;     //  Tfs_base_data index
    unsigned char buffer[512-2-4];    //  decrypt data
 };
```

And there's another kind of header flag "NC", indicating unused disk data area.

The decryption process of TDL-4 can be described as pseudo-code below:

```
int  decrypt_tdl4_buf(char *module, char *encrypt_buff, char *out_buff,
                       int size, Tfs_cfg_head *fs_head)
{
       //Position to decrypt modules
       Tfs_cfg_section  *psection = find_module(fs_head,module);
       //Get decrypt date size,
        int size = psection->size;
        int offset = psecton-> offset;
        //Decrypt buffer
        int  max_section = get_max_section ();
        int  cur_section = max_section - offset;
        int  filesize = 0;
        while(filesize < size)
        {
            //Form disk read data to specified sector
            Tfs_base_data *base_data =  read_buff(cur_section);
```

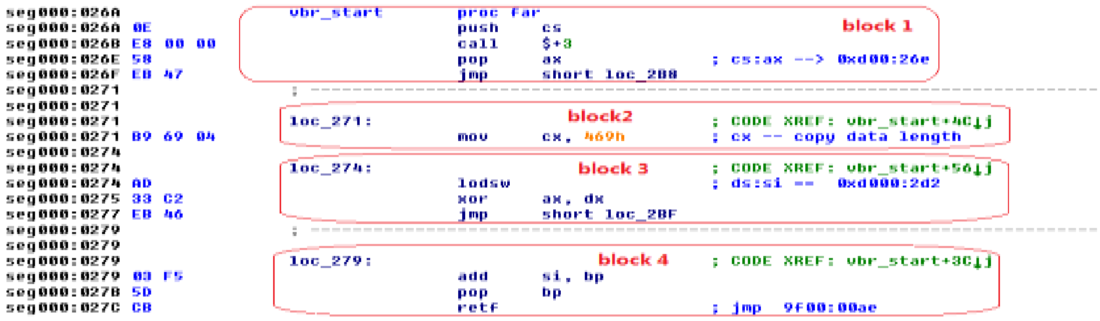
```
            decrypt_rc4(base_data->buffer, 506);
            memcpy(out_buff, base_data->buffer, 506);
            cur_section--;
            filesize += 506;
        }
    }
```

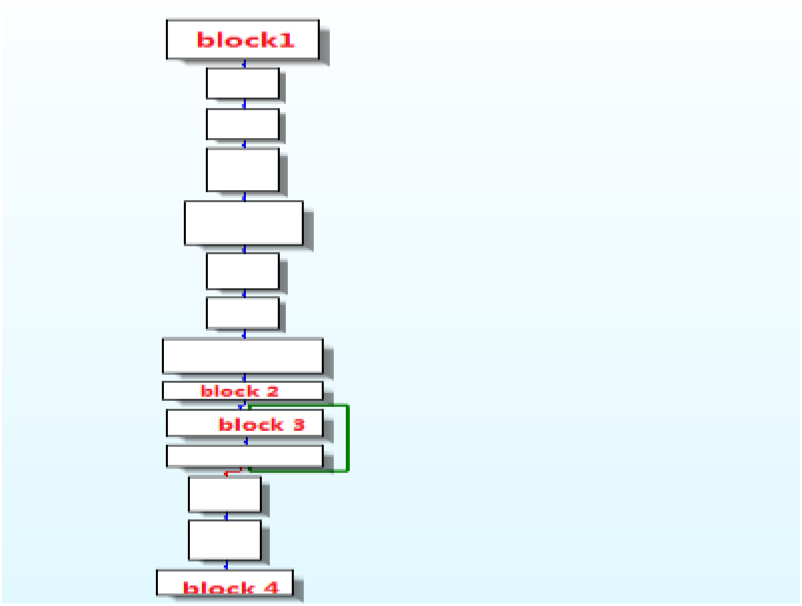
Phanta generation 1/2 all write virus data into first 63 sector of disk; the only difference is generation 2 makes data encrypted before writing. Phanta generation 3 no longer save data in first 2-64 sectors, but put it at the end of disk in encrypted form. Early variant of Phanta generation 6 still put virus data at end of disk but leave it unencrypted.

Chinese Bootkit writers try to protection data by driver module and pay less attention on data protection in designing new file system like TDL4.

2.2 Boot-stage code obfuscation

Bootloader code of Win32/Rovnix is highly obfuscated and the 16bit Bootloader code after infection differs in every variant. This may suggest the code is generated by machine rather than hand made one. Every piece of code of Rovnix is divided to many fragments, combined by jmp instruction, or call to a meaningless function (We are not sure about the necessity of these function, or just to make code more complex?).



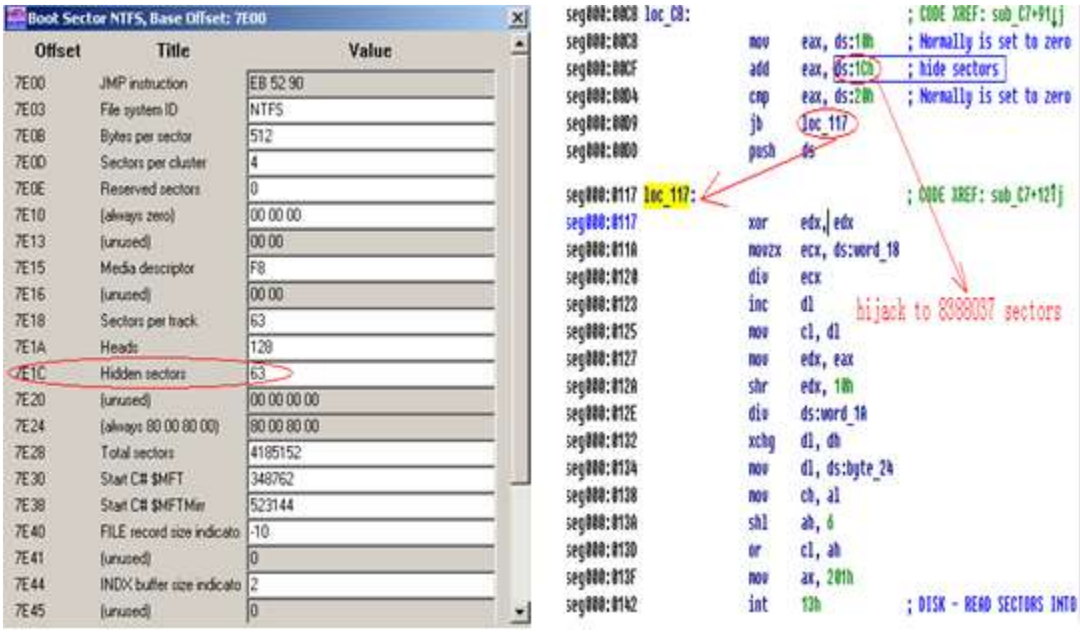


(PICTURE 2)

This is more likely to be polymorph method used in WIN32 virus, but obviously debugging polymorphic code in boot stage is much more difficult than that in Win32.

2.3 Boot-stage new code injection method

For the method of infected MBR/VBR is more and more common, Win32/Gapz changes target to Hideen sector field of BPB(Bios Parameter Block) in VBR. The infection is very hidden by just modifying 2-4 bytes and BPB structure is dependent with hardware device so that make signature for BPB will cause false positive. Normal MBR/VBR scanning will be bypassed.



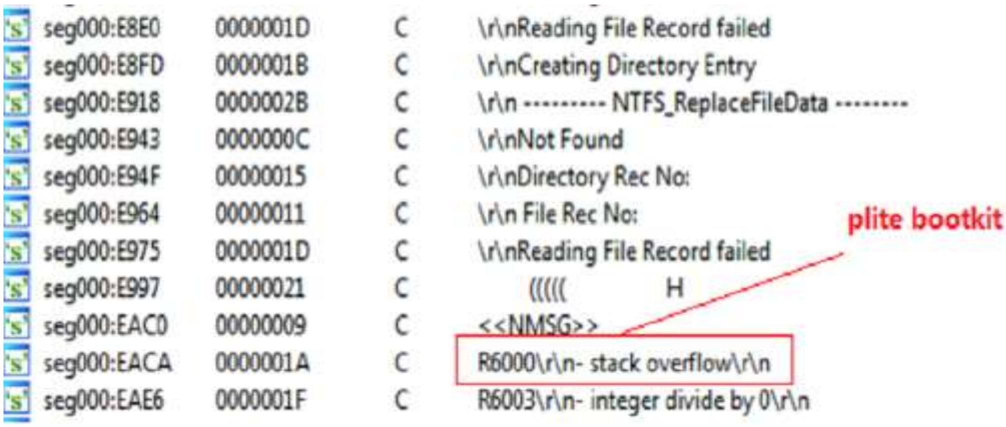
(PICTURE 3)

Similarly, as updated version of TDL4, TDL/MAXSS modified DPT table to bypass MBR scanning. In the same way, we can also move MBR-DPT table upper for 16 bytes, and modify MBR code for just 1 byte to make DPT offset 0x1ae position to be loaded and run. This method can bypass detection from all AVER.

2.4 Server side polymorphism

1) Possible server side polymorphism has been seen in Rovnix virus family, whose boot stage code doesn't change after infection but completely different among variants, which is polymorphic obfuscated.
2) The Plite family infects MBR and continues to infect Explorer.exe by parsing NTFS/FAT32 itself. Interestingly, the virus uses multi programming language, even very old one.

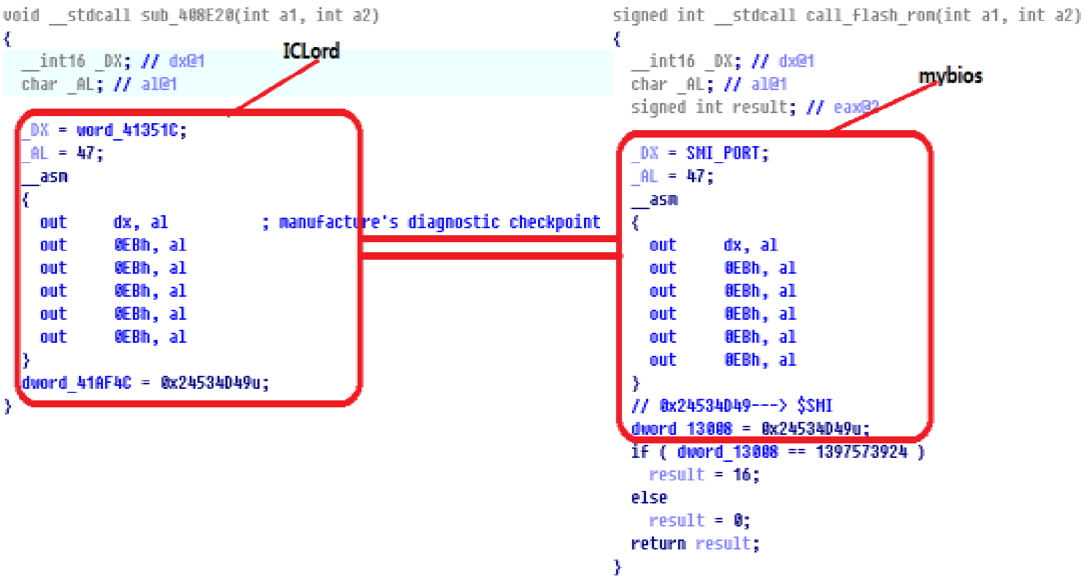
```
Dropper: C#
Droppee: Delphi
Boot loader: Fortran/Qbasic/VC 1.X
```



(PICTURE 4)

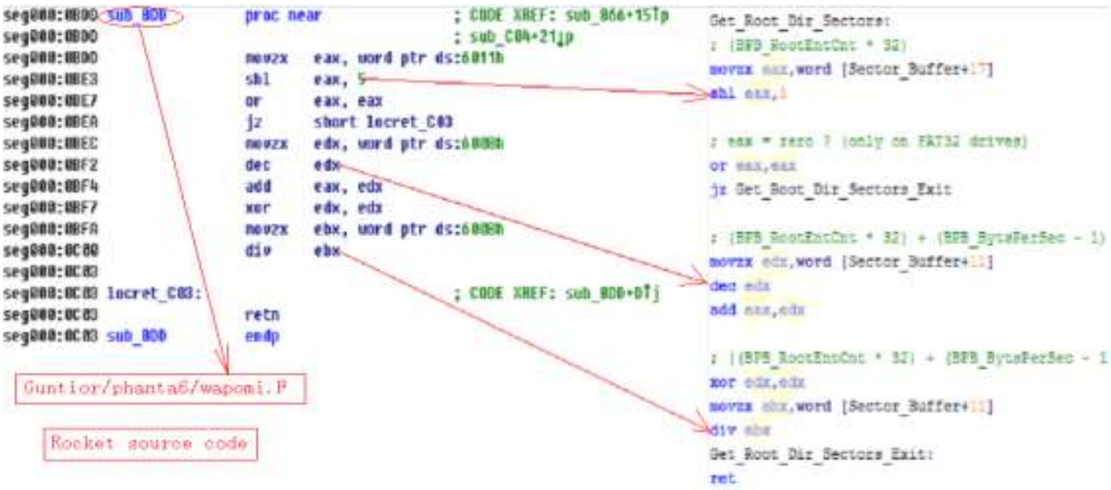
2.5 Bootkit trend in China

The most compelling Bioskit BMW actually imitates the code of Iclord in 2007, with the core component almost the same. And we also found that BMW Bioskit is evoluted from long-existing wapomi virus family. The instruction “out 0EBh, al” is to make delay, the two sample use the instruction even in same count; that’s less likely to be made by different writer.



(PICTURE 5)

Chinese Bootkit starts from imitating Mebroot and the method used to hook int13 and search for BILoaderBlock structure in OSloader are very similar, including Phantom generation 1/2/3. Chinese Bootkit is imitating and development fast; there has been sample can infect MBR+NTFS system and Bootkit source code has been spreading underground.



(PICTURE 6)

Above code is obtained from Phanta6, which uses FAT/NTFS parsing code in StonedBootkit.

2.6 Evolution of Anti-Bootkit technique

1) Standalone reparation tool (kernel level reparation, disk reparation). It is used to resolve particular issue like Phanta, tdss… by recovering MBR etc. in kernel hook.
   Advantage: Highly targeted
   Disadvantage: Can't remove unknown or not targeted one
2) Protection in boot level. It uses Bootkit-like technique to provide common boot stage protection/recovery.
   Advantage: Can defend from unknown threat
   Disadvantage: Challenge in stability
3) Unknown threat protection module embedded in anti-virus engineer
   Advantage: Can be used in any scenario and can provide analyzing and reparation information for Bootkit.
   Disadvantage: Can't remove virus directly

3. Boot stage simulator design

3.1 Current detecting solutions

Most available defending solutions currently is by standalone reparation tool, such as Tdsskiller, bitdefender, avast Anti-bootkit, AVG Anti-bootkit, 360safe, kingsoft etc. But these tools are difficult to embed into anti-virus engine for its high risk on system stability, while signature based detection doesn't work for unknown Bootkit threat. So we are thinking about the method to analyze the behavior of booting stage by simulating Windows booting process, could be a good solution.

3.2 Main component

1. Target code to be simulated (MBR/VBR/BootStrap code)
2. Dissembler, support both 16bit and 32bit
3. Instruction simulator
4. Hardware simulator, (BIOS/Memory(1M),Hard Disk, DMA/Interruptions)

3.3 Simulating BIOS

For we haven't work out to read real BIOS data to against Bioskit on video card/network card firmware, our BIOS simulator is mainly for preparations before simulation process starts. Of course we can directly start from MBR as well. BIOS simulator do these things:

1. IVT table loading configuration
2. Switch to simulate BIOS or initialize IVT table by simulator
3. System memory configuration 0x000:0x413
4. Booting media selection (floppy or HD)

3.4 Simulating real mode memory

We need just 1MB memory to run simulating code; its data layout is complete the same as real physical memory range 0~1MB. Some memory range is not necessary to support.

1. 0~0x400 for IVT table
2. 0x7C00~0x9FC00 for disk boot code
3. Reserve memory for extended BIOS data, VGA buffer and BIOS Routine
4. 0xFE000~0xFFF0 for BIOS BootBlock, which is code simulated by our BIOS simulator.

3.5 Simulating hard disk

The simulated hard disk is read from real physical hard disk and for some Bootkit we have to read data from read data according to simulating conditions.

1. Loading from Cylinder 0 head 0 sector 1
2. MBR(1 sector), VBR(1 sector),bootstrap code (16 sector)
3. If Bootkit hide data at the end of disk, we need to load these data in run time
4. Hard disk configuration (63 sector, 16 head, a big cylinder)

3.6 Simulating Boot CPU

CPU simulator consists of simulated registers, instruction identifier, addressing system, code parser and exception handler. There are 8 regular registers, 6 segment register, flag register, eip register, FPU register, MMX register, floating point register and 6 debugging registers to be simulated; instruction identifier is the disassembler; the addressing system is to calculate memory address by addressing mode. The details are listed below:

1) Simulated CPU
   Common registers, segment registers, flag register, EIP, XMM registers…
2) Dissembler
   Code identifier for 16bit and 32bit instructions
3) Addressing system
   Memory addressor, to map operator to memory address
4) Instruction parser/simulator
   Execute instruction virtually

```
    //CPU context
typedef struct tag Emu_cpu
{
    uint32_t                eip;            //eip ,16bit/32bit
    uint32_t                addr_ip;        //cs:ip
    CommonRegister          r[8];           //common 32bit regsiter
    FlagsRegister           f;              //flags register
    uint8_t                 flagOF;
    uint8_t                 flagDF;
    CommonRegister          s[6];           //segment regsiter
    uint16_t                nFpuCWD;        //fpu control register
    uint16_t                nFpuSWD;        //fpu control register
    uint16_t                nFpuTWD;        //fpu control register
    //cpu control global information
    Memory          *emu_mem;           //point to global memory
    uint16_t        **ppInsTable;       //point to global parse instruction table
    uint8_t         opcode;             //current parse opcode
    VOID            *op1;               //temp variable
    VOID            *op2;               //temp variable
} Emu_cpu, *PEmu_cpu;
```

Most of instructions are simulated by software and a few instructions are replaced by real asm instruction directly.

```
   void Emu_Xchg(Emu_cpu * e_cpu, OPT_SIZE opType,uint32_t t_Tmp)
   {
    void *DesOp1, void *SrcOp2;
    DesOp1 = e_cpu->op1;
```

```
    SrcOp2 = e_cpu->op2;
    If(!DesOp1 || !SrcOp2 ) return;
    switch(opType)
    {
    case Bit_8:
        t_Tmp = *(uint8_t *)DesOp1;
        *(uint8_t *)DesOp1 = *(uint8_t *)SrcOp2;
        *(uint8_t *)SrcOp2 = t_Tmp;
        break;
    case Bit_16:
        t_Tmp = *(uint16_t *)DesOp1;
        *(uint16_t *)DesOp1 = *(uint16_t *)SrcOp2;
        *(uint16_t *)Src = t_Tmp;
        break;
    case Bit_32:
        t_Tmp = *(uint32_t *)DesOp1;
        *(uint32_t *)DesOp1  = *(uint32_t *)SrcOp2;
        *(uint32_t *)Src = t_Tmp;
        break;
    }
   }
```

For example it's too complex to simulate the DAA instruction, but rather use real ASM DAA instruction to get result.

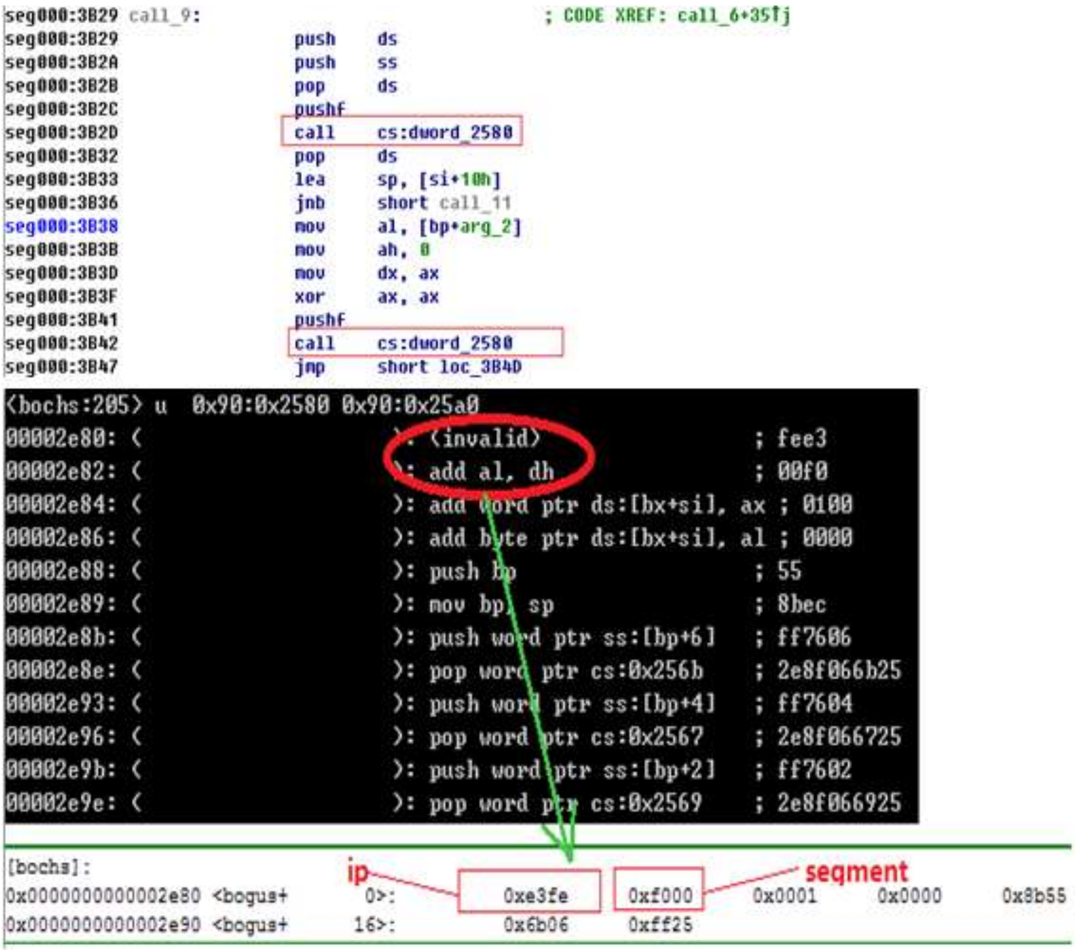
```
void DAA(Emu_cpu *e_cpu)
{
   If(!Cpu) return;
    __asm
    {
        mov al, e_cpu.r[EAX]._8._l
        push e_cpu.f
        popf
        DAA
        pushf
        pop e_cpu.f
        mov e_cpu.r[EAX]_8._l,al
    }
   }
```

We have to be aware that real mode can use 32bit registers, so instruction like pushad/popad which is not in 16bit instruction set should be supported as well, in spite of 32bit addressing. And interruption routine need to be simulated too.

3.7 Behavior judgment in boot stage

Bootkit behaves differently with normal boot code, so we can make detections according to these malicious behaviors such as:

1. Interruption vector table (IVT) hook
2. Run into illegal memory range
3. Code obfuscation in MBR/VBR
4. Code decompressing in boot stage
5. Delay loading MBR (After bootstrap code)
6. Execute invalid instruction (Demange only like DarkSeoul)
7. Execute int 16 to wait for keyboard input (MBRLock)



(PICTURE 7)

There are several tricks used by Bootkit to escape detection, as the picture above; the malware read segment address and offset of int 13h and save them into its own memory space, and then make up to a new disk read/write function.

There are some boot codes created by thread party or manufactor, such as encrypted MBR with PGP Diskcryptor, Dell computer etc. have to be excluded to avoid false positive.

Only a few interruption routine need to simulated, like int 1h, int 10h, int 13h, int 16h etc. The solution to simulate int 13h can be:

1) Simulate IO port operation on BIOS simulation
2) To simulate DMA, read from HD to memory directly

When the simulated BIOS initialize:

1) Define interruption address table, copy to 0~0x3ff memory range on BIOS initialization

```
db   13h                              ; INT13
DW OFFSET DGROUP:Emu_Bios_13h         ; INT13 offset
Emu_Bios_13h:
cmp al,0
je _DiskReset
out 3,al   // call DMA funtion
iret
_DiskRest:
cmp dl,0x80  ;80h = drive
je _SetDisk
iret; // ret
_SetDisk
LAHF
and al,0xfe  //cf set 0
sahf
mov al,0
iret
```

2) Following code run in simulated CPU when simulating DMA

```
Emu_IO_Out_3(emu_cpu *e_cpu)
{
  Get_Select_road();
  Get_mem_address_for_start(e_cpu->r[EAX]._8) ; //cpu.ax
  Get_count_bytes(e_cpu->r[EAX]._8._l); //cpu.al
   if(2 == e_cpu->r[EAX]._8._h)        //cpu.ah
     emu_read_disk(e_cpu);
   if else (3 == e_cpu->r[EAX]._8._h) //cpu.ah
     emu_write_disk(e_cpu);
   ...
}
```

The details of the implementation can refer to design of EasyVM..

The interruptions can also be simulated by pure software: opcode for interruption is 0xCD; we can map interruption implementation in instruction parsing table to simulate interruptions.

```
uint32_t X86_Instruction_Parse[MAX_X86_INS_FUNC];

 X86_Instruction_Parse[0xCC]=(uint32_t)Emu_Int3;
```

```
X86_Instruction_Parse[0xCD]=(uint32_t)Emu_Int_imm8;
X86_Instruction_Parse[0xCE]=(uint32_t)Emu_Into ;
Emu_Int_imm8(emu_cpu *e_cpu)
{
   uint_8 num = *(( uint8_t *)e_cpu->opcode+1);
   if(2 == e_cpu->r[EAX]._8._h && 13h == num) //ah == 2
        emu_read_disk(e_cpu);
   else if(3 == e_cpu->r[EAX]._8._h && 13h == num)ah == 3
        emu_write_disk(e_cpu);
   ...
}
```

DEMO:

We could detection MebRoot used Boot-Emulator.

You can see that IVT hooked by Boot-Emulator and malicious MBR run to original MBR position at 0x7C00 address.

```
1) From disk to start(0x80)
2) System Memory size 0x27F KB,segment value:0x9FC0
[ 1]Bios is running...

[ 2]malicious code is running...

<Warning!!!>
Boot Emulator: Found IVT-hook on system load to MBR.

IVT interrupt          :0x005b:0xf000 [0x0000:0x004C | 0x0000:0x004E]

Hooked IVT interupt    :0x0066:0x9f40 [0x0000:0x004C | 0x0000:0x004E]


<Warning!!!>
Boot Emulator: Detect current code runing illegal code area.
[ 3]MBR is running...

[ 4]VBR is running...


[ 5]BootStrap is running...
```

4. Customized bootstrap code to reload boot files

Researcher form Kaspersky release a new Bootkit report on 2012 <<Cybercriminals switch from MBR to NTFS>>, analyzed the new hiding method. This reminds us that Bootkit writer is looking for more hiding place. But wherever Bootkit hide itself, if we can make sure the booting process is using clean file/data, Bootkit will not get change to take control.

For Windows XP/7, only DPT, BPB is hardware dependent, other components is same on different machine. So we can save first 63+17 sector of HD as a clean backup. The differences between Windows XP and Windows 7 are:
Windows XP (x32/x64) -- mbr,vbr, bootstrap code (6 sectors)
Windows 7 (x32/x64)   -- mbr,vbr, bootstrap code (8 sectors)

Windows XP just load MBR and VBR to 0x7c00, then bootstrap code will be loaded into corresponding memory segment. While Windows 7 modified its own data, boot loading will fail if there's no real run VBR. So we have to set register context to be consistent with real machine.

For Windows XP, register context after VBR should be:
ax = 0x0d00   cx = 0x1b8
dx = 0x0   sp = 0x7bfc
bp = 0x7be   si   = 0x7be
di = 0x7c00   ip   = 0x7c7a
Jump to 0xd00:0x26a to continue running.
push 0d00h
push 26ah
retf ;

Set following registers' value for Win7:
cs - 0x7c0 ,ss-0x000,ds-0x7c0 ,es-0x09a0,fs-0xea30 ,gs - 0xf000

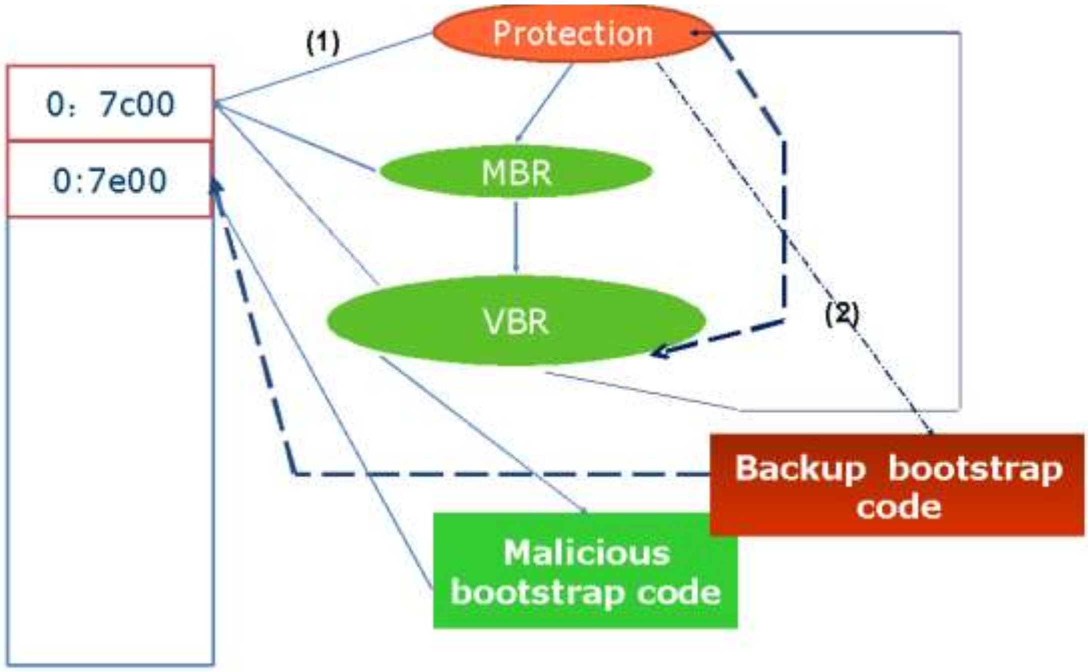
For bootstrap code could also be infected, loading bootstrap code by default VBR will still vulnerable. So we need to hook VBR, to load our clean bootstrap code.

```
;hook win7 vbr ,and then we loader custom backup bootstrap code.
;seg000:010D 33 C0                      xor     ax, ax          ;
;seg000:010F BF 28 10                   mov     di, 1028h
;seg000:0112 B9 D8 0F                   mov     cx, 0FD8h
;seg000:0115 FC                         cld
;seg000:0116 F3 AA                      rep stosb
;seg000:0118 E9 5F 01                   jmp     near ptr 27Ah   ;  hook here
;seg000:0118            reloc_vbr_mian  endp
;seg000:0118
;seg000:0118            ; ---------------------------------------------------------
;seg000:011B 90                         db  90h ;
;seg000:011C 90                         db  90h ;

;fix it
;seg000:05CC 68 22 11        push    1122h
;seg000:05CF 68 44 33        push    3344h
;seg000:05D2 CB              retf
```

(PICTURE 8)

A clean boot file reloading process by our customized bootstrap code is shown below:



(PICTURE 9)

This method can defend against almost all Bootkit except MBRLock, including Gapz, Rovnix, TDL4, Plite, phata, fishp, sinowal, Stoned etc.

5. Bootkit review

There's a question about how to identify a Bootkit technique features from complex Bootkit threat.

We think that there are 3 basic components for Bootkit technique.

1) Boot component. To get early change to take control of system, which can use UEFI, BIOS, MBR, VBR, Bootstrap code, ntldr, bootmgr… any of them. Current Bootkit's existence is rooted on traditional real mode BIOS booting without security mechanism. TPM or Secure Boot based booting process will be better protected.
2) Patch kernel code is just like 0day; will be used in boot stage once disclosed. We can see
3) *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.

6. 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]《Advanced Evasion Techniques by Win32/Gapz》CARO2013

[2]《DeepBoot》CanSecWest2012

[3]《Boot Emulator:anti-bootkit technology》
Kaspersky_Lab_Whitepaper_Boot_Emulator_ENG_final.pdf

[4] http://bochs.sourceforge.net/

[5] http://bbs.pediy.com/showthread.php?t=96970&highlight=easyvm

[6]《Bootkit's development overview and trend》AVAR2012