HUNTING FOR SOTI

The Equation Group's advanced boot loader exposed

F-Secure.

CONTENTS

NB: This research is a follow-on from our previous piece, **Hunting for KillSuit.**

WHAT IS SOTI?

SolarTime (SOTI) is an advanced bootloader persistence mechanism used by The Equation Group as part of their frameworks, including within the Dandersprtiz framework that was exposed by The Shadow Brokers in 2017. The framework containing SOTI can be used in conjunction with the Killsuit (KiSu) post-exploitation modular component, allowing an attacker to persist their PeddleCheap (PC) agent across reboots. SOTI is the only persistence mechanism for this framework that still works on a modern version of the Windows OS; however, it is mitigated if the unified extensible firmware interface (UEFI) is used place of the standard basic input/output system (BIOS).

Other persistence mechanisms that are ineffective beyond Windows XP include driver installation persistence and JustVisiting (JUVI), which is XP specific. Driver persistence does not work beyond XP as driver signing became mandatory in future versions of the OS, thereby making the persistence mechanism fail. SOTI, however, uses firmware-level manipulation in order to create an advanced bootloader to the attacker's agent on the host that works at least up to Windows 7.

HOW SOTI WORKS

How a Windows 7 system boots

Part of understanding SOTI's persistence is refreshing ourselves on how Windows 7 boots. The figure below shows the general flow. We are going to review this legacy boot process and discuss how SolarTime (SOTI) affects the boot system of a Windows 7 x64 machine. We will not be exploring UEFI, for example, as it can obscure the underlying concepts we aim to explore.'

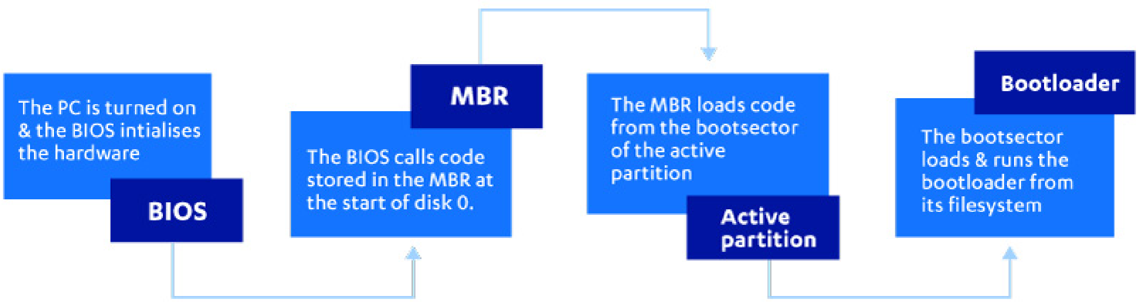

Figure 14: https://neosmart.net/wiki/mbr-boot-process/

When a computer is powered on, the BIOS performs some self-tests and hardware initialization before loading the Master Boot Record (MBR) into memory. The MBR is responsible for determining the active partition of the bootable hard drive. The structure of the MBR starts with 0x1BE bytes of boot code followed by four partition tables. The MBR then parses the partition tables to determine which Volume Boot Record (VBR) should be read into the system. It then overwrites itself in memory with the VBR.

The VBR contains further information about the partition and is responsible for loading the Initial Program Loader (IPL). It starts with 2 bytes of jmp instruction that jumps to the code that performs various checks. The bytes below the jump instruction contains the OEM ID 'NTFS' and the Bios Parameter Block (BPB), which contains information about the NTFS volume such as SectorsPerCluster and ClustersPerFileRecord.

```
seg000:7C00                 seg000          segment byte public 'CODE' use16
seg000:7C00                                 assume cs:seg000
seg000:7C00                                 ;org 7C00h
seg000:7C00                                 assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg000:7C00 EB 52                           jmp     short checkInt13Extensions
seg000:7C02                 ; ---------------------------------------------------------------------------
seg000:7C02 90                              nop
seg000:7C02                 ; ---------------------------------------------------------------------------
seg000:7C03 4E 54 46 53 20+aNtfs            db 'NTFS    ',0         ; DATA XREF: seg000:7C6A↓r
seg000:7C0C 02                              db    2
seg000:7C0D 08                              db    8
seg000:7C0E 00                              db 0                    ; DATA XREF: seg000:7C66↓w
seg000:7C0E                                                         ; seg000:7C95↓r
seg000:7C0F 00                              db    0
```

Figure 15

At the end of the VBR, control is transferred to the IPL. The IPL occupies 15 sectors of 512 bytes each and is usually allocated right after the VBR. It parses the filesystem and loads the bootmgr into memory – hence, it is also sometimes called 'the bootmgr loader'. The following figure shows the first few bytes of the IPL.

```
00007E00  07 00 42 00 4F 00 4F 00  54 00 4D 00 47 00 52 00  ..B.O.O.T.M.G.R.
00007E10  04 00 24 00 49 00 33 00  30 00 00 E0 00 00 00 30  ..$.I.3.0..a...0
00007E20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007E30  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007E40  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
00007E50  00 00 00 00 00 00 EB 70  90 90 05 00 4E 00 54 00  ......dp....N.T.
00007E60  4C 00 44 00 52 00 00 00  00 00 00 00 00 00 00 00  L.D.R...........
00007E70  00 00 00 00 00 00 00 00  00 00 A0 6F 66 27 A3 38  ..........áof'ú8
00007E80  E9 11 88 F3 9C B6 D0 FD  A6 72 FF FF 00 00 00 00  T.ê=£¦-²ªr··....
00007E90  06 00 00 00 06 00 24 00  4F 00 62 00 6A 00 49 00  ......$.O.b.j.I.
00007EA0  64 00 02 00 24 00 4F 00  00 00 00 00 00 00 00 00  d...$.O.........
```

Figure 16

The bootmgr then manages the boot process and waits until a boot option is chosen before passing control to winload.exe to load the kernel and the boot start drivers.

How SOTI affects the early boot records in Windows 7

In 2015, Kaspersky published a report on a bootkit termed "GrayFish" that reflashes the hard drive firmware before infecting the VBR. It was later found out that "GrayFish" is actually SOTI.

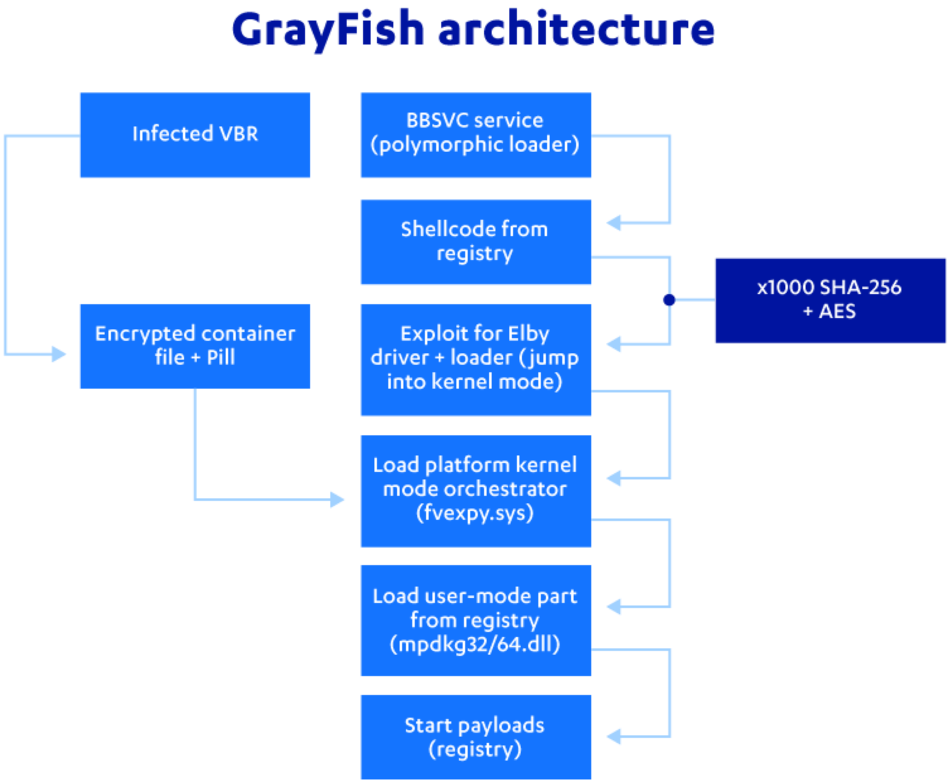

Figure 17

This particular bootkit is initially loaded from a modified VBR and IPL. It then waits for winload.exe to load, and patches the first legitimate driver with a malicious payload. We will now analyze the infected VBR in detail, using IDA Pro’s Remote GDB debugger to analyze an infected Windows 7 x64 SP1 Virtual Machine.

In a normal boot, the MBR is loaded at 0000:7C00, and proceeds to overwrite itself with the VBR. If you put a breakpoint at 7C00, the first run will present you with the MBR, and the second will be the VBR. The VBR starts with a jmp instruction.

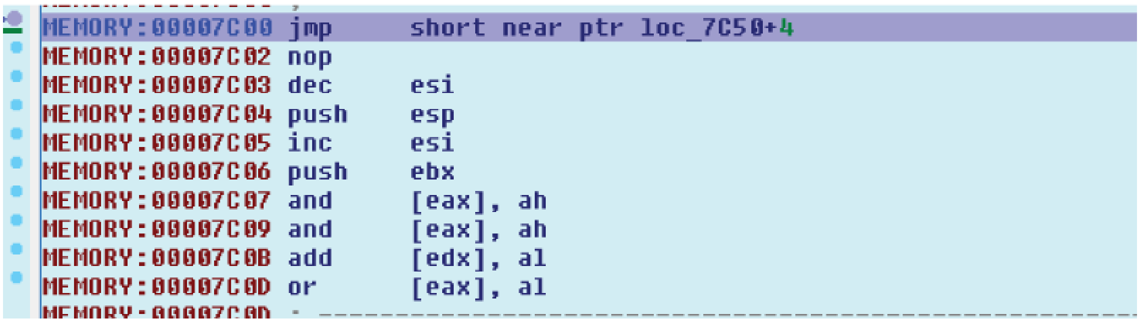

Figure 18

This jumps over the BPB to the address 0000:7C54, checks for INT 13 extensions, reads drive parameters, loads the 15 sectors of IPL into 0000:7E00, and finally checks for support for Trusted Computing Group (TCG) using BIOS interrupt 13 before passing control to the IPL.

The figure below shows a normal VBR on the left vs VBR infected by SOTI on the right. At the end of the VBR are multiple error strings used to inform the user if something goes wrong e.g bootmgr is missing. If an error is shown, the system will prevent execution via the hlt instruction. However, in SOTI, the hlt instruction is overwritten and therefore disables the disk error reporting.

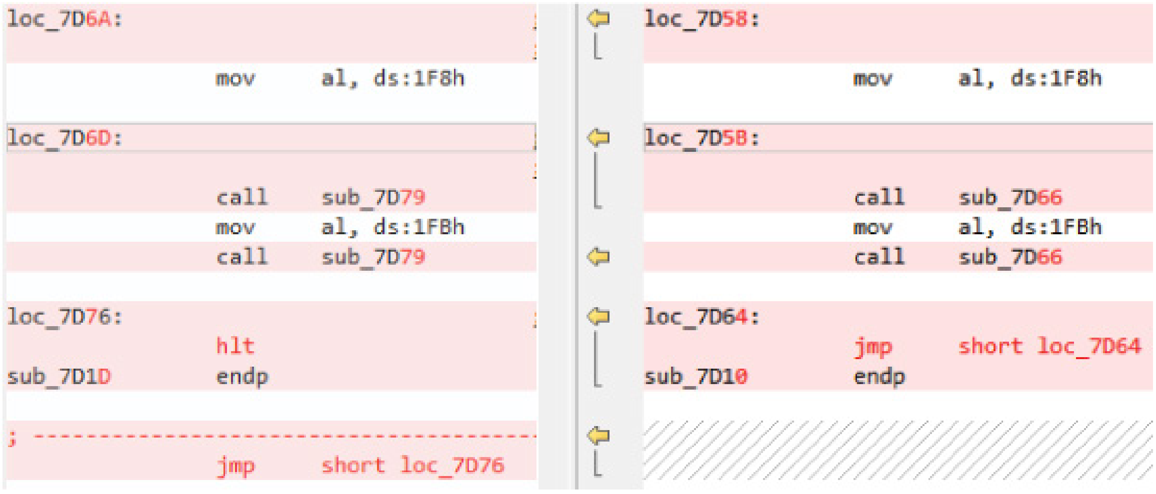

Figure 19

In a clean boot, the VBR passes control to the IPL code at address 0000:7E7A. SOTI overwrites 7E7A with malicious data that is used for decryption purposes later and so jumps to a different address to run the IPL.

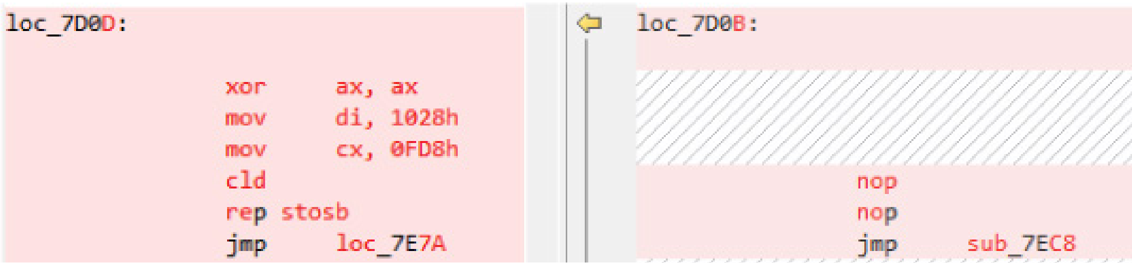

Figure 20

The IPL parses the NTFS filesystem and knows how to read MFT File Records as well as check their data integrity. It reads the $MFT File into memory to start finding out the location of the bootmgr. Some basic functions for reading and parsing the $MFT file are modified by SOTI to perform the loading of its malicious bootpack, so they do not need to be re-implemented by SOTI. An example is the function below, which performs the search for the first Index Node Header of the $INDEX_ROOT attribute of the MFT file. A clean boot stores the base address of $INDEX_ROOT in ds:232 and uses this function to search for the file with the filename "BOOTMGR". However, SOTI escapes the filename check by setting the register ecx as a flag. If ecx is 0, SOTI passes in its own $INDEX_ROOT located at ds:2B4 instead.

```
sub_87DF        proc near               ; CODE
                                        ; seg0(
                push    eax
                push    ecx
                mov     edx, eax
                mov     eax, ds:232h
                lea     ebx, [eax+10h]
                add     ax, [ebx+4]
                lea     eax, [eax+10h]
```

```
sub_889C        proc near
                push    eax
                push    ecx
                mov     edx, eax
                or      ecx, ecx
                jnz     short loc_88AE
                mov     eax, ds:2B4h
                jmp     short loc_88B2
; ---------------------------------------
loc_88AE:
                mov     eax, ds:232h
loc_88B2:
                lea     ebx, [eax+10h]
                add     ax, [ebx+4]
                lea     eax, [eax+10h]
```

Figure 21

The main purpose of the IPL is to load the bootmgr code into memory, which starts with the signature E9D501EB0490000000528BC30E076633. It starts by searching through the $INDEX_ ALLOCATION attributes of the $MFT file, getting a list of all the subnodes of $INDEX_ROOT, and locating the bootmgr index record. The bootmgr index record indicates the logical sector number in the disk where bootmgr is located and loads the bootmgr into the address 0x20000. Immediately after the bootmgr is loaded, SOTI is seen altering the control flow and injecting a jump into a chunk of its own malicious code where it begins to load its bootpack from the drive.

```
seg000:80D0                   loc_80D0:                               ; CODE XREF: sub_7EC8+1C0↑j
seg000:80D0 67 66 0F B7 58+                   movzx   ebx, word ptr [eax+0Ch]
seg000:80D6 66 81 E3 FF 00+                   and     ebx, 0FFh
seg000:80DD 0F 85 59 0C                       jnz     loc_8D3A
seg000:80E1 66 8B D8                          mov     ebx, eax
seg000:80E4 68 00 20                          push    2000h
seg000:80E7 07                                pop     es
seg000:80E8                                   assume es:nothing
seg000:80E8 66 2B FF                          sub     edi, edi
seg000:80EB 66 A1 3E 02                       mov     eax, ds:23Eh
seg000:80EF E8 33 01                          call    readDriveAccordingToDataRun ; loads bootmgr to address 20000
seg000:80F2 68 00 20                          push    2000h
seg000:80F5 07                                pop     es
seg000:80F6 66 2B FF                          sub     edi, edi
seg000:80F9 66 A1 3E 02                       mov     eax, ds:23Eh
seg000:80FD E8 60 0B                          call    sub_8C60
seg000:8100 E9 AB 12                          jmp     loc_93AE        ; inserted malicious jump
seg000:8100                   sub_7EC8        endp ; sp-analysis failed
```

Figure 22

The first thing SOTI does after the jump is to use BIOS interrupt 15 to query the system address map and find out the type and length of memory available above 1MB. This information gathered is used to determine if the addresses are free to load its bootpack. To access memory above 1MB, SOTI also enables the A20 line. After all checks passes, SOTI begins the process of loading several MFT File Records to find the location of its bootpack.

One of the MFT File Records loaded into memory is the $ObjId file. This file contains all of the $OBJECT_ID Attributes in use in the volume. The $INDEX_ROOT of an $ObjId file has the filename "$O". As this file isn't loaded in the usual boot process, the string "$O" is not in the data section of the IPL (the strings are used when looking for the right file to load into memory). Therefore, SOTI injects its own set of data after the usual IPL data at 0000:7E7A.

```
seg000:89AD 66 33 D2                    xor     edx, edx
seg000:89B0 66 0F B7 0E 10+             movzx   ecx, word ptr ds:210h ; 4 = length of string "$I30"
seg000:89B6 66 BA 12 02 00+             mov     edx, 212h       ; offset of string "$I30"
seg000:89BC EB 0C                       jmp     short loc_89CA
seg000:89BE             ; ---------------------------------------------------------------------------
seg000:89BE
seg000:89BE             loc_89BE:                               ; CODE XREF: MFTDataToMemoryThenFindAttributeEbx+44↑j
seg000:89BE 66 0F B7 0E A2+             movzx   ecx, word ptr ds:2A2h ; Instead of string "$I30",
seg000:89BE 02                                                  ; malware changes parameters and makes the
seg000:89BE                                                     ; function compare with string "$O"
seg000:89BE                                                     ;
seg000:89BE                                                     ; this address stores length of "$O"
seg000:89BE                                                     ;
seg000:89BE                                                     ; malware makes sure this will be
seg000:89BE                                                     ; executed if initial value of edx is 0
seg000:89C4 66 BA A4 02 00+             mov     edx, 2A4h       ; offset of string "$O"
seg000:89CA
```

Figure 23

SOTI loops through the Index Entries of $ObjId file to find the index entry. The following figure shows the structure of an Index Entry from the NTFS documentation:

$O Index

Offset	Size	Value	Description
~	~	~	Standard Index Header
0x00	2	0x20	Offset to data
0x02	2	0x38	Size of data
0x04	4	0x00	Padding
0x08	2	0x58	Size of Index Entry
0x0A	2	0x10	Size of Index Key
0x0C	2		Flags
0x0E	2	0x00	Padding
0x10	16		Key | GUID Object Id
0x20	8		Data | MFT Reference
0x28	16		Data | GUID Birth Volume Id
0x38	16		Data | GUID Birth Object Id
0x48	16		Data | GUID Domain Id

Figure 24

At offset 0x10 is the GUID Object Id of the index. SOTI compares this value with the object ID stored at address 0000:7E7A. Should the values match, SOTI would load the File Record of the bootpack by referring to the MFT Reference at offset 0x20. In the system that we were testing, the file record belongs to a truetype font file named "davidbi.ttf".

```
80 01 00 00 00 04 00 00  00 00 00 00 00 00 00 00  Ç...............
05 00 00 00 3D 09 01 00  08 00 00 00 00 00 00 00  ....=...........
10 00 00 00 60 00 00 00  00 00 00 00 00 00 00 00  ....`...........
48 00 00 00 18 00 00 00  2B D9 04 AB 2B 04 CA 01  H.......++.½+.-.
DD 80 39 25 0C EA C9 01  3A E0 51 12 B4 CC D4 01  ¦Ç9%.O+.:aQ.¦¦+.
2B D9 04 AB 2B 04 CA 01  20 00 00 00 00 00 00 00  ++.½+.-.-.......
00 00 00 00 00 00 00 00  00 00 00 00 ED 01 00 00  ............f...
00 00 00 00 00 00 00 00  10 44 7C 01 00 00 00 00  .........D|.....
30 00 00 00 70 00 00 00  00 00 00 00 00 00 02 00  0...p...........
58 00 00 00 18 00 01 00  0B 07 00 00 00 00 01 00  X...............
58 F9 45 12 B4 CC D4 01  58 F9 45 12 B4 CC D4 01  X-E.¦¦+.X-E.¦¦+.
58 F9 45 12 B4 CC D4 01  58 F9 45 12 B4 CC D4 01  X-E.¦¦+.X-E.¦¦+.
00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ................
20 00 00 00 00 00 00 00  0B 03 64 00 61 00 76 00  -.........d.a.v.
69 00 64 00 62 00 69 00  2E 00 74 00 74 00 66 00  i.d.b.i...t.t.f.
40 00 00 00 28 00 00 00  00 00 00 00 00 00 03 00  @...(...........
10 00 00 00 18 00 00 00  A0 6F 66 27 A3 38 E9 11  ........áof'ú8T.
88 F3 9C B6 D0 FD A6 72  80 00 00 00 48 00 00 00  ê=£¦-²ªrÇ...H...
01 00 00 00 00 00 04 00  00 00 00 00 00 00 00 00  ................
0F 00 00 00 00 00 00 00  40 00 00 00 00 00 00 00  ........@.......
00 00 01 00 00 00 00 00  97 FF 00 00 00 00 00 00  ........ù-......
```

Figure 25

If you ran check_soti.py in your DdSz machine, the output shows that the SOTI Container for the kernel driver is "davidbi.ttf".

```
02:00:47>> python check_soti.py
[02:00:47] ID: 1426 'python' started [target: z0.0.0.53]
- soti_check:
-   Found 1 SOTI container(s):
-     C:\Windows\Fonts\davidbi.ttf (65431)
-   SOTI 1.3.3 or 1.3.4. doublefeature will confirm

    Command completed successfully
```

Figure 26

Looking at the source code of check_soti.py, the variable SOTIContainers defines the various names that the malicious kernel driver container could take and "davidbi.ttf" is in the list.

```
SOTIContainers =
        ['consolad.ttf',
        'davidbi.ttf',
        'georgiad.ttf',
        'palabd.ttf',
        'tahomabi.ttf',
        'timesbc.ttf',
        'trebucbc.ttf',
        'verdanad.ttf']
```

https://github.com/misterch0c/shadowbroker/blob/master/windows/Resources/Ops/PyScripts/check_soti.py

With the file record, SOTI then proceeds to read the encrypted file into memory.

```
600000  0F CD 3B 85 F3 27 39 5F  12 A0 87 78 F1 0C 11 38  .-;à='9_.áçx±..8
600010  0A 3D 94 88 37 66 9E 24  A7 41 62 F4 B4 29 9E 21  .=öê7fP$ºAb(¦)P!
600020  68 28 34 43 5C B4 14 E7  A9 64 A1 08 0B 86 64 D5  h(4C\¦.t¬dí..åd+
600030  1D ED 1F BD E6 CA F1 C6  13 02 D4 DC 38 9F 16 E1  .f.+µ-±¦..+_8■.ß
600040  61 DB 25 64 B2 CF 7D A9  7D B9 B5 EF 3F 5D C6 AE  a¦%d¦-}¬}¦¦n?]¦«
600050  76 64 86 75 B0 C7 54 97  57 45 B9 40 95 9A 1E 9A  vdåu¦¦TùWE¦@òÜ.Ü
600060  04 03 15 87 9A 46 96 38  50 84 E6 0A 7C F8 25 93  ...çÜFû8Päµ.|°%ô
600070  8B 02 CB C5 F1 8D 58 17  5B 4E 2E E8 8D 8F 96 A7  ï.-+±.X.[N.F..ûº
600080  9B B2 35 90 8D BF 30 0F  0B AA 70 F4 CC 11 1B 09  ¢¦5..+0..¬p(¦...
600090  BA B5 D2 73 87 85 11 45  68 07 56 99 12 D1 00 6F  ¦¦-sçà.Eh.VÖ.-.o
6000A0  80 EC 46 DA A6 D1 2C C0  22 AC B6 2F 3F 91 2F 72  Ç8F+ª-,+"¼¦/?æ/r
6000B0  EF 8C 01 B3 BD 1F 53 41  77 00 79 12 62 69 61 CD  nî.¦+.SAw.y.bia-
6000C0  FA 2F 9F 42 52 6E BE 97  5A 66 56 BC A5 84 0A B5  ·/■BRn+ùZfV+Ñä.¦
6000D0  FD 23 89 F2 38 31 B5 6F  CD 0D DF 70 09 E8 97 4B  ²#ë=81¦o-.¯p.FùK
6000E0  39 E6 1B 6D CC B9 8D D6  0E DD 8F 09 76 26 86 2A  9µ.m¦¦.+.¦..v&å*
6000F0  DD C0 FF BA 3E A0 F1 74  CE C6 50 09 4E 7E 3F 48  ¦+·¦>á±t+¦P.N~?H
600100  4A 42 24 CA 1E 19 F9 D4  60 6A 64 4E 68 66 B1 8D  JB$-..·+`jdNhf¦.
```

Figure 27

SOTI decrypts the malware loaded into memory using an encryption key that was stored at 0000:7E7A. The encryption key was generated by hashing the NTFS Object_ID 1000 times with SHA-256.

```
int polynomial = 0xEDB88320
int seed = 0xFFFFFFFF

function InitializeTable() {
    table = new int[256]
    for (int i=255; i>=0; i--) {

        int index = i
        for (int j=8; j>0; j--) {
            if (index & 1 == 1) {
                index = (index >>1) ^ polynomial
            } else {
                index >>=1
            }
        }
        table[i] = index
    }

    return table
}

function calculateHash(int start, int size, int[] table, byte[] message) {
    int hash = seed
    for(int i=start; i< size; i++) {
        hash = (hash >> 8) ^ table[message[i] ^ hash & 0xff]
    }
    return hash
}
```

After decryption of the malware, SOTI proceeds on to calculate the CRC32 hash for byte 5 to byte 0xEB1 of its bootpack. The figure below shows the pseudo code for the algorithm

Figure 28

SOTI then matches the hash with byte 1 to 4 of its bootpack, presumably to check the file integrity, before passing control to the bootpack.

DETECTING SOTI

If you have DanderSpritz installed in your machine, connect to a victim and run its internal check_soti.py script. If SOTI is present, the script would inform you the exact SOTI container present in the victim. Although precise, this method is rather inconvenient as it requires you to compromise the victim through fuzzbunch before you can execute the script. A handier way is to scan the VBR for abnormalities. As previously mentioned, SOTI modifies the section of VBR just before control is transferred to the IPL. In a clean VBR (Windows 7 and above), the code zero-fills all the linear memory locations from AA28 through B9FF. This is overwritten by SOTI and replaced with 2 nop instruction before jumping to the new IPL location.

```
loc_7D0D:

                xor     ax, ax
                mov     di, 1028h
                mov     cx, 0FD8h
                cld
                rep stosb
                jmp     loc_7E7A
```

```
loc_7D0B:

                nop
                nop
                jmp     sub_7EC8
```

One could simply scan the VBR for the two nop instruction in the VBR using Python. Here I have a modified version of pyMBR written by hamptus. pyMBR parses the MBR and looks for the active partition table. In the partition table structure, there is an element lbaStart, which contains the first sector of the partition relative to the start of disk. This first sector is the start of VBR:

```
def get_vbr(open_disk, sector):
    with open(open_disk, 'rb') as disk:
        disk.seek(0)
        disk.seek(sector * SECTOR_SIZE)
        vbr = Vbr(disk.read(512))
        return vbr
```

Since the loaded position of the nop instructions during boot is 0x7D0B, they will occupy be the 267th and 268th byte of the VBR (0x7D0B – 0x7C00). If these bytes are equal to 0x90, the host will be marked as infected.

```
class Vbr(object):
    """
    Parses the Volume Boot Record
    """

    def __init__(self, data):
        self.first = struct.unpack("<BB", data[0:2])
        self.soti_bytes = struct.unpack("<BB", data[267:269])

    def is_soti_infected(self):
        if (self.soti_bytes[0] == 144 and self.soti_bytes[1] == 144):
            return True

        return False
```

```
C:\Python27>python C:\Users\lacie\Desktop\checksoti.py.txt \\.\PhysicalDrive0
SOTI Bootkit is present
```

CONCLUSION

With the above, we have shown how SOTI hides itself in the boot section and how to detect it. This method is used in every Windows 7 KillSuit installation due to its ability to bypass driver signature enforcement. Its low-level persistence that started in the firmware shows that The Equation Group has access to the hard disk drive (HDD) manufacturer's proprietary information. The high level of encryption also escapes detection from tools and anti-virus software. Prevention would require manufacturers to sign the firmware, where verification of the firmware would fail should anyone tamper with it.

SOURCES

https://www.cs.bu.edu/~goldbe/teaching/HW55815/presos/eqngroup.pdf

https://www.youtube.com/watch?v=R5mgAsd2VBM

https://github.com/misterch0c/shadowbroker/blob/master/windows/Resources/Ops/PyScripts/check_soti.py

http://www.irongeek.com/i.php?page=videos/derbycon8/track-3-17-killsuit-the-equation-groups-swiss-army-knife-for-persistence-evasion-and-data-exfil-francisco-donoso

https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2018/03/08064459/Equation_group_questions_and_answers.pdf

Nobody has better visibility into real-life cyber attacks than F-Secure. We're closing the gap between detection and response, utilizing the unmatched threat intelligence of hundreds of our industry's best technical consultants, millions of devices running our award-winning software, and ceaseless innovations in artificial intelligence. Top banks, airlines, and enterprises trust our commitment to beating the world's most potent threats.

Together with our network of the top channel partners and over 200 service providers, we're on a mission to make sure everyone has the enterprise-grade cyber security we all need. Founded in 1988, F-Secure is listed on the NASDAQ OMX Helsinki Ltd.

f-secure.com | twitter.com/fsecure | linkedin.com/ f-secure

F-Secure