HUNTING FOR SOTI

The Equation Group’s advanced boot loader exposed
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.

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.

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.

**GrayFish architecture**

![GrayFish architecture diagram]

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 SPI 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.

![Memory view of 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:284 instead.
The main purpose of the IPL is to load the bootmgr code into memory, which starts with the signature E9D501EB049000000528BC30E076633. 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.

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.
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

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>~</td>
<td>~</td>
<td>Standard Index Header</td>
</tr>
<tr>
<td>0x00</td>
<td>2</td>
<td>0x20</td>
<td>Offset to data</td>
</tr>
<tr>
<td>0x02</td>
<td>2</td>
<td>0x38</td>
<td>Size of data</td>
</tr>
<tr>
<td>0x04</td>
<td>4</td>
<td>0x00</td>
<td>Padding</td>
</tr>
<tr>
<td>0x08</td>
<td>2</td>
<td>0x58</td>
<td>Size of Index Entry</td>
</tr>
<tr>
<td>0x0A</td>
<td>2</td>
<td>0x10</td>
<td>Size of Index Key</td>
</tr>
<tr>
<td>0x0C</td>
<td>2</td>
<td>0x00</td>
<td>Padding</td>
</tr>
<tr>
<td>0x10</td>
<td>16</td>
<td>Key</td>
<td>GUID Object Id</td>
</tr>
<tr>
<td>0x20</td>
<td>8</td>
<td>Data</td>
<td>MFT Reference</td>
</tr>
<tr>
<td>0x28</td>
<td>16</td>
<td>Data</td>
<td>GUID Birth Volume Id</td>
</tr>
<tr>
<td>0x38</td>
<td>16</td>
<td>Data</td>
<td>GUID Birth Object Id</td>
</tr>
<tr>
<td>0x48</td>
<td>16</td>
<td>Data</td>
<td>GUID Domain Id</td>
</tr>
</tbody>
</table>

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

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.

Figure 27

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

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.

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 the 267th and 268th byte of the VBR (0x7D0B – 0x7C00). If these bytes are equal to 0x90, the host will be marked as infected.
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.youtube.com/watch?v=R5mgAsd2VBM

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


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