UEFI threats moving to the ESP: Introducing ESPecter bootkit

welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/

October 5, 2021

ESET research discovers a previously undocumented UEFI bootkit with roots going back all the way to at least 2012

ESET researchers have analyzed a previously undocumented, real-world UEFI bootkit that persists on the EFI System Partition (ESP). The bootkit, which we've named ESPecter, can bypass Windows Driver Signature Enforcement to load its own unsigned driver, which facilitates its espionage activities. Alongside Kaspersky's recent discovery of the unrelated FinSpy bootkit, it is now safe to say that real-world UEFI threats are no longer limited to SPI flash implants, as used by Lojax.

The days of UEFI (Unified Extensible Firmware Interface) living in the shadows of the legacy BIOS are gone for good. As a leading technology embedded into chips of modern computers and devices, it plays a crucial role in securing the pre-OS environment and loading the operating system. And it's no surprise that such a widespread technology has also become a tempting target for threat actors in their search for ultimate persistence.

In the last few years, we have seen proof-of-concept examples of UEFI bootkits (DreamBoot, EfiGuard), leaked documents (DerStarke, QuarkMatter) and even leaked source code (Hacking Team Vector EDK), suggesting the existence of real UEFI malware either in the form of SPI flash implants or ESP implants. Despite all of the above, only three real-world cases of UEFI malware have been discovered so far (LoJax, discovered by our team in 2018, MosaicRegressor, discovered by Kaspersky in 2019, and most recently the FinSpy bootkit, whose analysis was just published by Kaspersky). While the first two fall in the category of SPI flash implants, the last falls in the ESP implants category, and surprisingly, it's not alone there.

Today, we describe our recent discovery of ESPecter, just the second real-world case of a UEFI bootkit persisting on the ESP in the form of a patched Windows Boot Manager to be analyzed. ESPecter was encountered on a compromised machine along with a user-mode client component with keylogging and document-stealing functionalities, which is why we believe ESPecter is mainly used for espionage. Interestingly, we traced the roots of this threat back to at least 2012, previously operating as a bootkit for systems with legacy BIOSes. Despite ESPecter's long existence, its operations and upgrade to UEFI went unnoticed and have not been documented until now. Note that the only similarity between ESPecter and the Kaspersky FinSpy find is that they share the UEFI boot manager compromise approach.

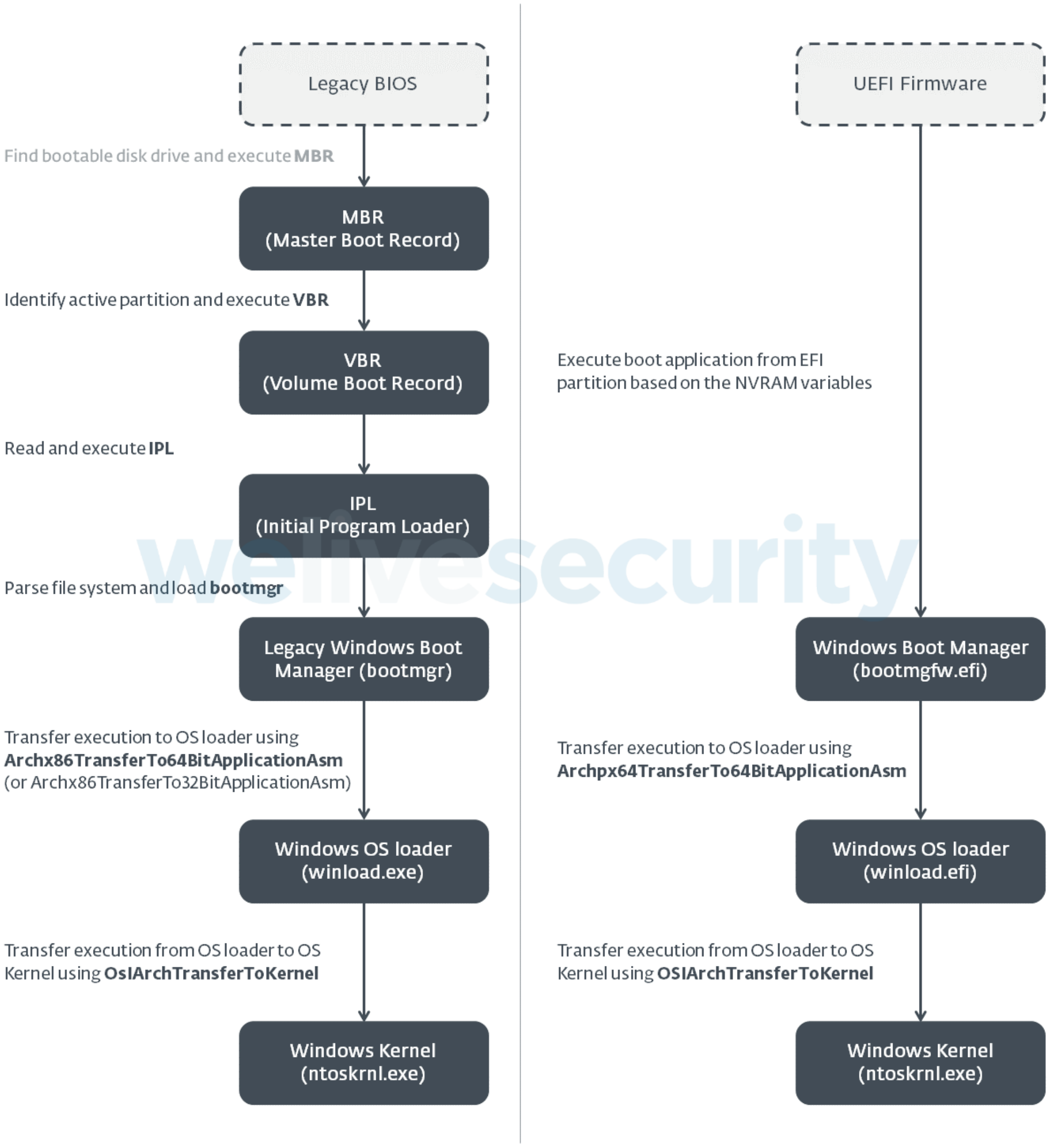

Legacy BIOS
Find bootable disk drive and execute MBR
MBR (Master Boot Record)
Identify active partition and execute VBR
VBR (Volume Boot Record)
Read and execute IPL
IPL (Initial Program Loader)
Parse file system and load bootmgr
Legacy Windows Boot Manager (bootmgr)
Transfer execution to OS loader using Archx86TransferTo64BitApplicationAsm (or Archx86TransferTo32BitApplicationAsm)
Windows OS loader (winload.exe)
Transfer execution from OS loader to OS Kernel using OslArchTransferToKernel
Windows Kernel (ntoskrnl.exe)
UEFI Firmware
Execute boot application from EFI partition based on the NVRAM variables
Windows Boot Manager (bootmgfw.efi)
Transfer execution to OS loader using Archpx64TransferTo64BitApplicationAsm
Windows OS loader (winload.efi)
Transfer execution from OS loader to OS Kernel using OslArchTransferToKernel
Windows Kernel (ntoskrnl.exe)

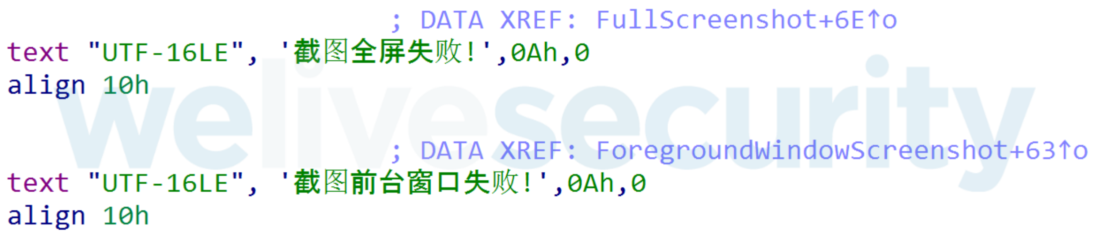
Figure 1. Comparison of the Legacy Boot flow (left) and UEFI boot flow (right) on Windows (Vista and newer) systems

By patching the Windows Boot Manager, attackers achieve execution in the early stages of the system boot process (see Figure 1), before the operating system is fully loaded. This allows ESPecter to bypass Windows Driver Signature Enforcement (DSE) in order to execute its own unsigned driver at system startup. This driver then injects other user-mode components into specific system processes to initiate communication with ESPecter's C&C server and to allow the attacker to take control of the compromised machine by downloading and running additional malware or executing C&C commands.

Even though Secure Boot stands in the way of executing untrusted UEFI binaries from the ESP, over the last few years we have been witness to various UEFI firmware vulnerabilities affecting thousands of devices that allow disabling or bypassing Secure Boot (e.g. VU#758382, VU#976132, VU#631788, …). This shows that securing UEFI firmware is a challenging task and that the way various vendors apply security policies and use UEFI services is not always ideal.

Previously, we have reported multiple malicious EFI samples in the form of simple, single-purpose UEFI applications without extensive functionality. These observations, along with the concurrent discovery of the ESPecter and FinFisher bootkits, both fully functional UEFI bootkits, show that threat actors are not relying only on UEFI firmware implants when it comes to pre-OS persistence, but also are trying to take advantage of disabled Secure Boot to execute their own ESP implants.

We were not able to attribute ESPecter to any known threat actor, but the Chinese debug messages in the associated user-mode client component (as seen in Figure 2) leads us to believe with a low confidence that an unknown Chinese-speaking threat actor is behind ESPecter. At this point, we don't know how it was distributed.



```
                                 ; DATA XREF: FullScreenshot+6E↑o
text "UTF-16LE", '截图全屏失败！',0Ah,0
align 10h

                                 ; DATA XREF: ForegroundWindowScreenshot+63↑o
text "UTF-16LE", '截图前台窗口失败！',0Ah,0
align 10h
```

Figure 2. Example of Chinese debug messages in the user-mode client component

Evolution of the ESPecter bootkit

When we looked at our telemetry, we were able to date the beginnings of this bootkit back to at least 2012. At its beginning, it used MBR (Master Boot Record) modification as its persistence method and its authors were continuously adding support for new Windows OS versions. What is interesting is that the malware's components have barely changed over all these years and the differences between 2012 and 2020 versions are not as significant as one would expect.

After all the years of insignificant changes, those behind ESPecter apparently decided to move their malware from legacy BIOS systems to modern UEFI systems. They decided to achieve this by modifying a legitimate Windows Boot Manager binary (bootmgfw.efi) located on the ESP while supporting multiple Windows versions spanning Windows 7 through Windows 10 inclusive. As we mentioned earlier, this method has one drawback – it requires that the Secure Boot feature be disabled in order to successfully boot with a modified boot manager. However, it's worth mentioning that the first Windows version supporting Secure Boot was Windows 8, meaning that all previous versions are vulnerable to this persistence method.

For Windows OS versions that support Secure Boot, the attacker would need to disable it. For now, it's unknown how the ESPecter operators achieved this, but there are a few possible scenarios:

- The attacker has physical access to the device (historically known as an "evil maid" attack) and manually disables Secure Boot in the BIOS setup menu (it is common for the firmware configuration menu to still be labeled and referred to as the "BIOS setup menu", even on UEFI systems).
- Secure Boot was already disabled on the compromised machine (e.g., user might dual-boot Windows and other OSes that do not support Secure Boot).
- Exploiting an unknown UEFI firmware vulnerability that allows disabling Secure Boot.
- Exploiting a known UEFI firmware vulnerability in the case of an outdated firmware version or a no-longer-supported product.

Technical analysis

During our investigation, we discovered several malicious components related to ESPecter:

- Installers, only for the older MBR versions of the bootkit, whose purpose was to set up persistence on the machine by rewriting the MBR of the boot device.
- Boot code in the form of either a modified Windows Boot Manager (bootmgfw.efi) on UEFI systems or a malicious MBR in the case of Legacy Boot systems.
- A kernel-mode driver used to prepare the environment for the user-mode payloads and to load them in the early stages of OS startup by injecting them into specific system processes.
- User-mode payloads responsible for communication with the C&C, updating the C&C configuration and executing C&C commands.

For the complete scheme of the ESPecter bootkit compromise, see Figure 3.

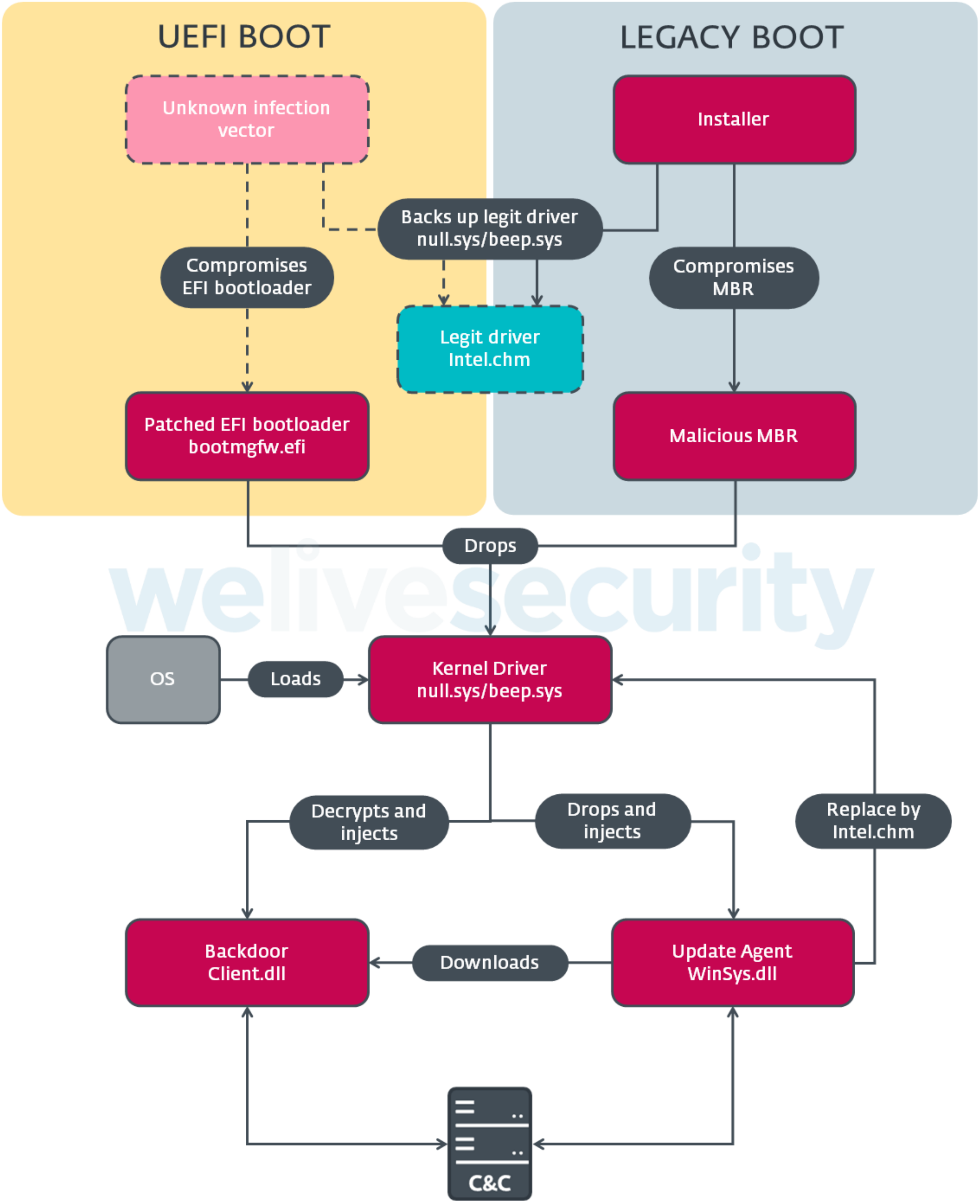
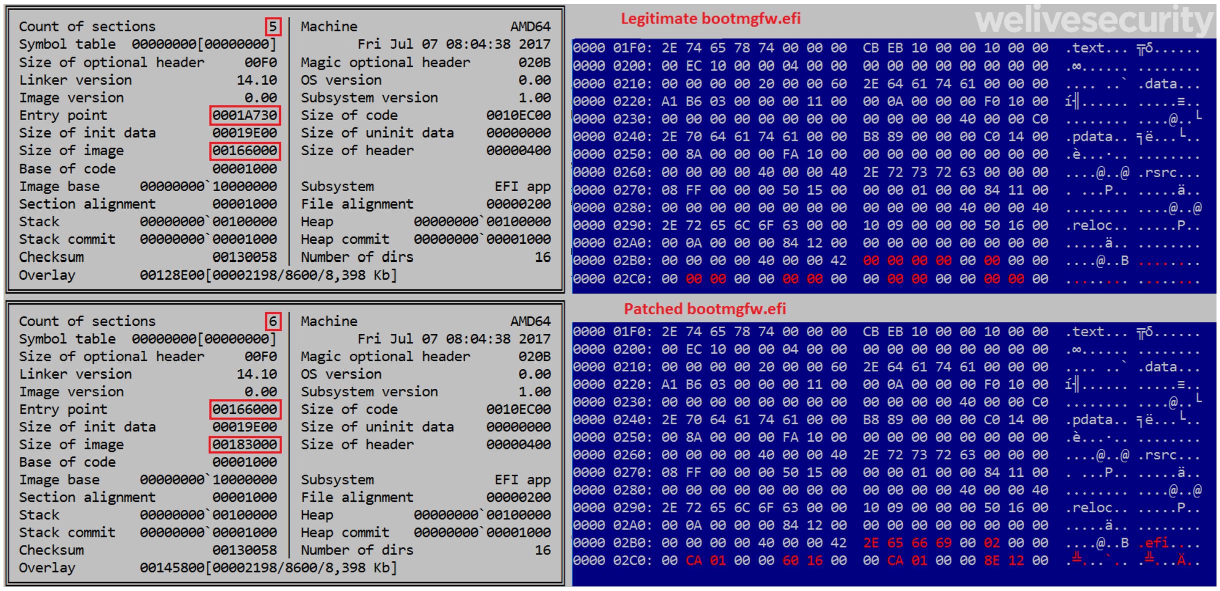


Figure 3. ESPecter bootkit components

Achieving persistence – UEFI boot

On systems using UEFI Boot mode, ESPecter persistence is established by modifying the Windows Boot Manager bootmgfw.efi and the fallback bootloader binary bootx64.efi, which are usually located in the ESP directories \EFI\Microsoft\Boot\ and \EFI\Boot\, respectively. Modification of the bootloader includes adding a new section called .efi to the PE, and changing the executable’s entry point address so program flow jumps to the beginning of the added section, as seen in Figure 4.



```
Count of sections                 5 | Machine                      AMD64
Symbol table  00000000[00000000]    |       Fri Jul 07 08:04:38 2017
Size of optional header        00F0 | Magic optional header         020B
Linker version                14.10 | OS version                    0.00
Image version                  0.00 | Subsystem version             1.00
Entry point                0001A730 | Size of code              0010EC00
Size of init data          00019E00 | Size of uninit data       00000000
Size of image              00166000 | Size of header            00000400
Base of code               00001000 |
Image base        00000000`10000000 | Subsystem                  EFI app
Section alignment          00001000 | File alignment            00000200
Stack             00000000`00100000 | Heap         00000000`00100000
Stack commit      00000000`00001000 | Heap commit  00000000`00001000
Checksum                   00130058 | Number of dirs                  16
Overlay       00128E00[00002198/8600/8,398 Kb]
```

Legitimate bootmgfw.efi

```
0000 01F0: 2E 74 65 78 74 00 00 00  CB EB 10 00 00 10 00 00  .text... ╦δ......
0000 0200: 00 EC 10 00 00 04 00 00  00 00 00 00 00 00 00 00  .∞...... ........
0000 0210: 00 00 00 00 20 00 00 60  2E 64 61 74 61 00 00 00  .... ..` .data...
0000 0220: A1 B6 03 00 00 00 11 00  00 0A 00 00 00 F0 10 00  í╢...... .....≡..
0000 0230: 00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 C0  ........ ....@..└
0000 0240: 2E 70 64 61 74 61 00 00  B8 89 00 00 00 C0 14 00  .pdata.. ╕ë...└..
0000 0250: 00 8A 00 00 00 FA 10 00  00 00 00 00 00 00 00 00  .è...·.. ........
0000 0260: 00 00 00 00 40 00 00 40  2E 72 73 72 63 00 00 00  ....@..@ .rsrc...
0000 0270: 08 FF 00 00 00 50 15 00  00 00 01 00 00 84 11 00  . ...P.. .....ä..
0000 0280: 00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 40  ........ ....@..@
0000 0290: 2E 72 65 6C 6F 63 00 00  10 09 00 00 00 50 16 00  .reloc.. .....P..
0000 02A0: 00 0A 00 00 00 84 12 00  00 00 00 00 00 00 00 00  .....ä.. ........
0000 02B0: 00 00 00 00 40 00 00 42  00 00 00 00 00 00 00 00  ....@..B ........
0000 02C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
```

```
Count of sections                 6 | Machine                      AMD64
Symbol table  00000000[00000000]    |       Fri Jul 07 08:04:38 2017
Size of optional header        00F0 | Magic optional header         020B
Linker version                14.10 | OS version                    0.00
Image version                  0.00 | Subsystem version             1.00
Entry point                00166000 | Size of code              0010EC00
Size of init data          00019E00 | Size of uninit data       00000000
Size of image              00183000 | Size of header            00000400
Base of code               00001000 |
Image base        00000000`10000000 | Subsystem                  EFI app
Section alignment          00001000 | File alignment            00000200
Stack             00000000`00100000 | Heap         00000000`00100000
Stack commit      00000000`00001000 | Heap commit  00000000`00001000
Checksum                   00130058 | Number of dirs                  16
Overlay       00145800[00002198/8600/8,398 Kb]
```

Patched bootmgfw.efi

```
0000 01F0: 2E 74 65 78 74 00 00 00  CB EB 10 00 00 10 00 00  .text... ╦δ......
0000 0200: 00 EC 10 00 00 04 00 00  00 00 00 00 00 00 00 00  .∞...... ........
0000 0210: 00 00 00 00 20 00 00 60  2E 64 61 74 61 00 00 00  .... ..` .data...
0000 0220: A1 B6 03 00 00 00 11 00  00 0A 00 00 00 F0 10 00  í╢...... .....≡..
0000 0230: 00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 C0  ........ ....@..└
0000 0240: 2E 70 64 61 74 61 00 00  B8 89 00 00 00 C0 14 00  .pdata.. ╕ë...└..
0000 0250: 00 8A 00 00 00 FA 10 00  00 00 00 00 00 00 00 00  .è...·.. ........
0000 0260: 00 00 00 00 40 00 00 40  2E 72 73 72 63 00 00 00  ....@..@ .rsrc...
0000 0270: 08 FF 00 00 00 50 15 00  00 00 01 00 00 84 11 00  . ...P.. .....ä..
0000 0280: 00 00 00 00 00 00 00 00  00 00 00 00 40 00 00 40  ........ ....@..@
0000 0290: 2E 72 65 6C 6F 63 00 00  10 09 00 00 00 50 16 00  .reloc.. .....P..
0000 02A0: 00 0A 00 00 00 84 12 00  00 00 00 00 00 00 00 00  .....ä.. ........
0000 02B0: 00 00 00 00 40 00 00 42  2E 65 66 69 00 02 00 00  ....@..B .efi....
0000 02C0: 00 CA 01 00 00 60 16 00  00 CA 01 00 00 8E 12 00  .╩...`.. .╩...Ä..
```

Figure 4. Comparison of original (top) and modified (bottom) Windows Boot Manager (bootmgfw.efi)

Simplified boot chain

As shown in the scheme on the left in Figure 5, the boot process on UEFI systems (ignoring the firmware part) starts with execution of the bootloader application located in the ESP. For the Windows OS, this is the Windows Boot Manager binary (bootmgfw.efi) and its purpose is to find an installed operating system and transfer execution to its OS kernel loader – winload.efi. Similar to the boot manager, the OS kernel loader is responsible for the loading and execution of the next component in the boot chain – the Windows kernel (ntoskrnl.exe*)*.

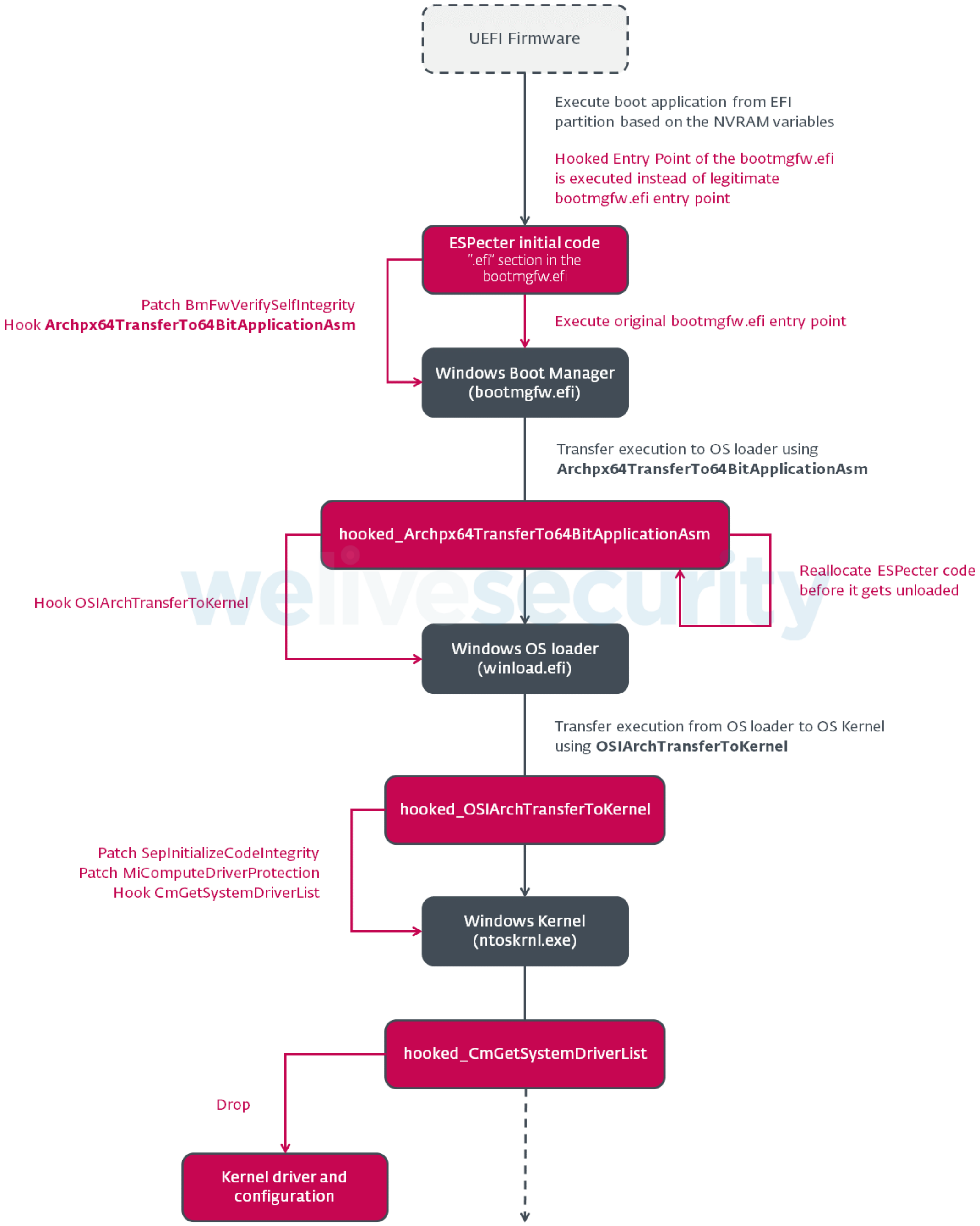

UEFI Firmware
Execute boot application from EFI partition based on the NVRAM variables
Hooked Entry Point of the bootmgfw.efi is executed instead of legitimate bootmgfw.efi entry point
ESPecter initial code
".efi" section in the bootmgfw.efi
Patch BmFwVerifySelfIntegrity
Hook **Archpx64TransferTo64BitApplicationAsm**
Execute original bootmgfw.efi entry point
Windows Boot Manager (bootmgfw.efi)
Transfer execution to OS loader using **Archpx64TransferTo64BitApplicationAsm**
hooked_Archpx64TransferTo64BitApplicationAsm
Reallocate ESPecter code before it gets unloaded
Hook OSIArchTransferToKernel
Windows OS loader (winload.efi)
Transfer execution from OS loader to OS Kernel using **OSIArchTransferToKernel**
hooked_OSIArchTransferToKernel
Patch SepInitializeCodeIntegrity
Patch MiComputeDriverProtection
Hook CmGetSystemDriverList
Windows Kernel (ntoskrnl.exe)
hooked_CmGetSystemDriverList
Drop
Kernel driver and configuration

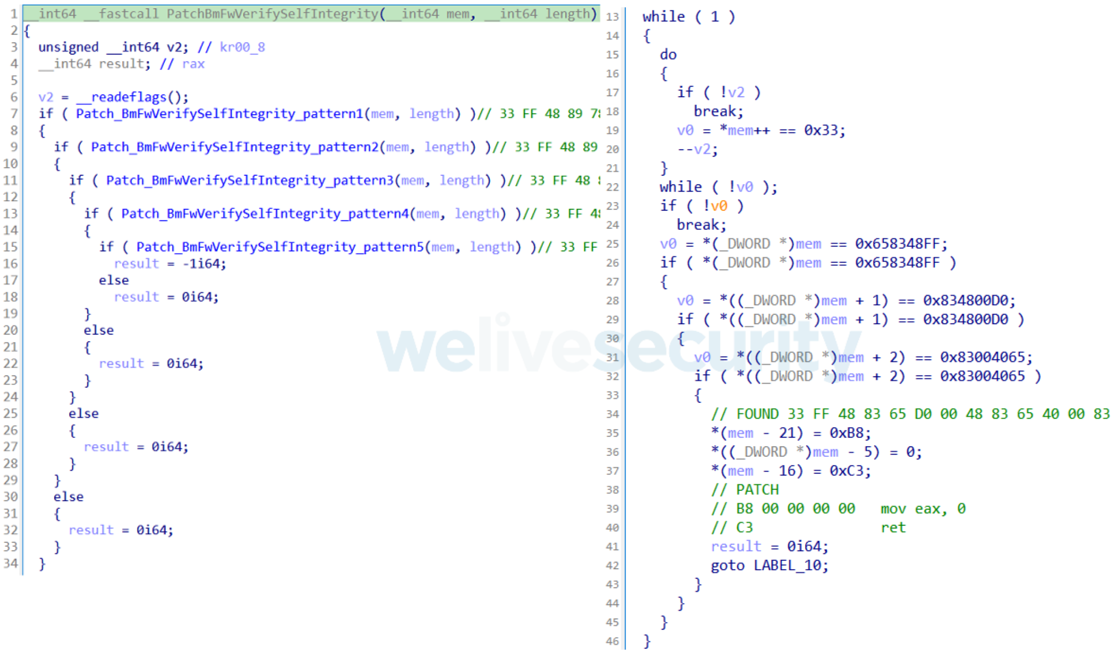
Figure 5. Boot flow modified by ESPecter

How does ESPecter modify the UEFI boot process?

In order to successfully drop its malicious payload, ESPecter needs to bypass integrity checks performed by the Windows Boot Manager and the Windows kernel during the boot process. To do this, it looks for byte patterns identifying the desired functions in memory and patches them accordingly.

Starting with the bootloader, in our case Windows Boot Manager (bootmgfw.efi), the bootkit begins by patching the BmFwVerifySelfIntegrity function. This function is responsible for verification of the boot manager's own digital signature and is intended to prevent execution of a modified boot manager. In Figure 6 you can see how ESPecter searches memory for BmFwVerifySelfIntegrity using various byte patterns (to support many bootmgfw.efi versions) and modifies this function in a way that it always returns zero, indicating that verification was successful.

As mentioned earlier, the bootloader's main goal is to find an installed operating system and transfer execution to its OS kernel loader. For the Windows Boot Manager, this happens in the Archpx64TransferTo64BitApplicationAsm function; therefore, ESPecter looks for this function in order to catch the moment that the OS loader is loaded into memory, but still hasn't been executed. If found, ESPecter patches this function to insert its own detour function, which can easily modify the loaded OS loader in memory at the right moment.



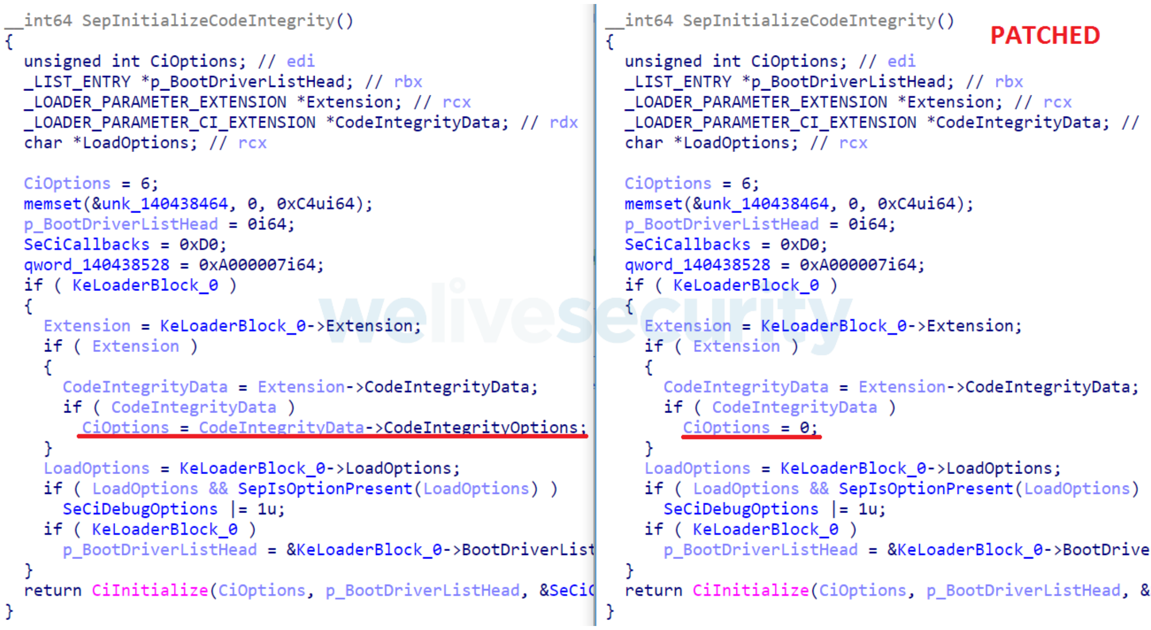
```
__int64 __fastcall PatchBmFwVerifySelfIntegrity(__int64 mem, __int64 length)
{
  unsigned __int64 v2; // kr00_8
  __int64 result; // rax

  v2 = __readeflags();
  if ( Patch_BmFwVerifySelfIntegrity_pattern1(mem, length) )// 33 FF 48 89 7:
  {
    if ( Patch_BmFwVerifySelfIntegrity_pattern2(mem, length) )// 33 FF 48 89
    {
      if ( Patch_BmFwVerifySelfIntegrity_pattern3(mem, length) )// 33 FF 48 :
      {
        if ( Patch_BmFwVerifySelfIntegrity_pattern4(mem, length) )// 33 FF 4:
        {
          if ( Patch_BmFwVerifySelfIntegrity_pattern5(mem, length) )// 33 FF
            result = -1i64;
          else
            result = 0i64;
        }
        else
        {
          result = 0i64;
        }
      }
      else
      {
        result = 0i64;
      }
    }
    else
    {
      result = 0i64;
    }
  }
```

```
  while ( 1 )
  {
    do
    {
      if ( !v2 )
        break;
      v0 = *mem++ == 0x33;
      --v2;
    }
    while ( !v0 );
    if ( !v0 )
      break;
    v0 = *(_DWORD *)mem == 0x658348FF;
    if ( *(_DWORD *)mem == 0x658348FF )
    {
      v0 = *((_DWORD *)mem + 1) == 0x834800D0;
      if ( *((_DWORD *)mem + 1) == 0x834800D0 )
      {
        v0 = *((_DWORD *)mem + 2) == 0x83004065;
        if ( *((_DWORD *)mem + 2) == 0x83004065 )
        {
          // FOUND 33 FF 48 83 65 D0 00 48 83 65 40 00 83
          *(mem - 21) = 0xB8;
          *((_DWORD *)mem - 5) = 0;
          *(mem - 16) = 0xC3;
          // PATCH
          // B8 00 00 00 00   mov eax, 0
          // C3               ret
          result = 0i64;
          goto LABEL_10;
        }
      }
    }
  }
```

Figure 6. Hex-Rays decompiled code – searching for and patching the BmFwVerifySelfIntegrity function

Modification of the OS loader does not include patching of any integrity checks or other functionality. At this stage it's important for the bootkit to reallocate its code, because as a UEFI Application it will be unloaded from memory after returning from its entry point function. For this purpose, it uses the BlImgAllocateImageBuffer or BlMmAllocateVirtualPages function (depending on the pattern found). After this reallocation, the bootkit inserts a detour (located in the previously allocated buffer) to the function responsible for transferring execution to the OS kernel – OslArchTransferToKernel – so it can patch the Windows kernel in memory, once it is loaded but before it is executed. The final stage of the bootkit's boot code is responsible for disabling DSE by patching the SepInitializeCodeIntegrity kernel function (see Figure 7 for details).



```
__int64 SepInitializeCodeIntegrity()
{
  unsigned int CiOptions; // edi
  _LIST_ENTRY *p_BootDriverListHead; // rbx
  _LOADER_PARAMETER_EXTENSION *Extension; // rcx
  _LOADER_PARAMETER_CI_EXTENSION *CodeIntegrityData; // rdx
  char *LoadOptions; // rcx

  CiOptions = 6;
  memset(&unk_140438464, 0, 0xC4ui64);
  p_BootDriverListHead = 0i64;
  SeCiCallbacks = 0xD0;
  qword_140438528 = 0xA000007i64;
  if ( KeLoaderBlock_0 )
  {
    Extension = KeLoaderBlock_0->Extension;
    if ( Extension )
    {
      CodeIntegrityData = Extension->CodeIntegrityData;
      if ( CodeIntegrityData )
        CiOptions = CodeIntegrityData->CodeIntegrityOptions;
    }
    LoadOptions = KeLoaderBlock_0->LoadOptions;
    if ( LoadOptions && SepIsOptionPresent(LoadOptions) )
      SeCiDebugOptions |= 1u;
    if ( KeLoaderBlock_0 )
      p_BootDriverListHead = &KeLoaderBlock_0->BootDriverList
  }
  return CiInitialize(CiOptions, p_BootDriverListHead, &SeCi
}
```

```
__int64 SepInitializeCodeIntegrity()
{
  unsigned int CiOptions; // edi
  _LIST_ENTRY *p_BootDriverListHead; // rbx
  _LOADER_PARAMETER_EXTENSION *Extension; // rcx
  _LOADER_PARAMETER_CI_EXTENSION *CodeIntegrityData; // r
  char *LoadOptions; // rcx

  CiOptions = 6;
  memset(&unk_140438464, 0, 0xC4ui64);
  p_BootDriverListHead = 0i64;
  SeCiCallbacks = 0xD0;
  qword_140438528 = 0xA000007i64;
  if ( KeLoaderBlock_0 )
  {
    Extension = KeLoaderBlock_0->Extension;
    if ( Extension )
    {
      CodeIntegrityData = Extension->CodeIntegrityData;
      if ( CodeIntegrityData )
        CiOptions = 0;
    }
    LoadOptions = KeLoaderBlock_0->LoadOptions;
    if ( LoadOptions && SepIsOptionPresent(LoadOptions) )
      SeCiDebugOptions |= 1u;
    if ( KeLoaderBlock_0 )
      p_BootDriverListHead = &KeLoaderBlock_0->BootDriver
  }
  return CiInitialize(CiOptions, p_BootDriverListHead, &S
}
```

Figure 7. Comparison of Hex-Rays decompiled SepInitializeCodeIntegrity function before (left) and after (right) it is patched in memory

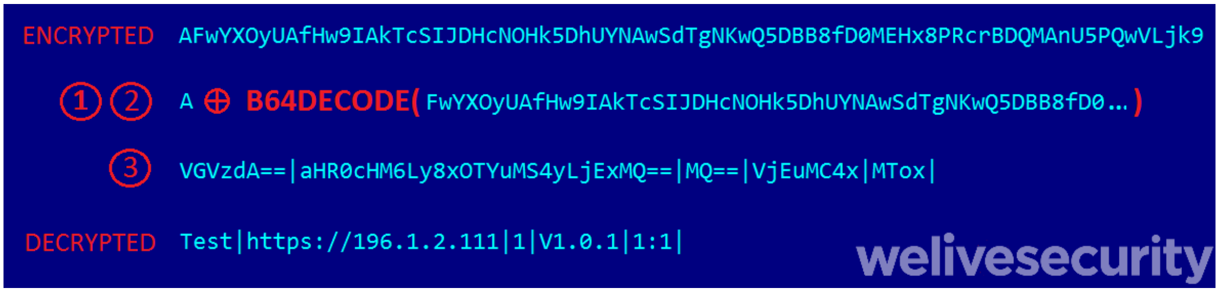
Interestingly, the boot code also patches the MiComputeDriverProtection kernel function. Even though this function does not directly affect successful loading of the malicious driver, the bootkit does not proceed to the driver dropping if it does not find and patch this function in kernel memory. We were not able to identify the purpose of this second patch, but we assume this modified function may be used by other, as yet unknown, ESPecter components.

- \SystemRoot\System32\null.sys (driver)
- \SystemRoot\Temp\syslog (encrypted configuration)

The configuration is used by the WinSys.dll user-mode component deployed by the kernel driver and consists of a one-byte XOR key followed by the encrypted configuration data. To decrypt the configuration, WinSys.dll:

1. Base64 decodes the configuration data
2. XORs the data with the XOR key
3. Base64 decodes each value delimited by "|" separately

An example of a configuration dropped by the EFI version of ESPecter is presented in Figure 8. A full list of IP addresses and domains from configurations embedded in the ESPecter bootkit samples that we have discovered (both Legacy Boot and UEFI versions) is included in the *IoCs* section.



```
ENCRYPTED  AFwYXOyUAfHw9IAkTcSIJDHcNOHk5DhUYNAwSdTgNKwQ5DBB8fD0MEHx8PRcrBDQMAnU5PQwVLjk9

   ① ②    A ⊕ B64DECODE( FwYXOyUAfHw9IAkTcSIJDHcNOHk5DhUYNAwSdTgNKwQ5DBB8fD0… )

     ③    VGVzdA==|aHR0cHM6Ly8xOTYuMS4yLjExMQ==|MQ==|VjEuMC4x|MTox|

DECRYPTED  Test|https://196.1.2.111|1|V1.0.1|1:1|
```

Figure 8. Decryption of configuration delivered by the EFI version of the ESPecter bootkit

Achieving persistence – Legacy Boot

As already mentioned, there are ESPecter versions supporting UEFI, and others supporting Legacy Boot, modes. In the case of Legacy Boot mode, persistence is achieved by the well-known technique of modifying the MBR code located in the first physical sector of the disk drive; therefore, we won't explain it in detail here, but will just summarize it.

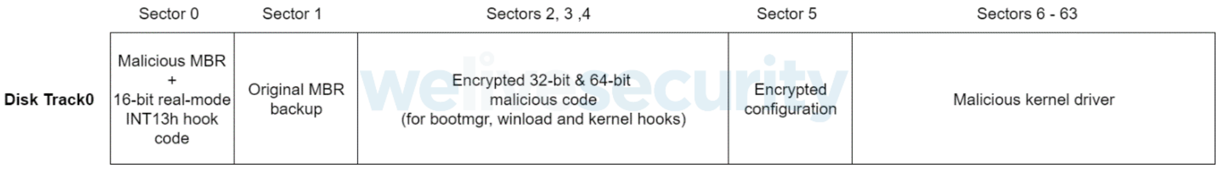
How does ESPecter modify the Legacy Boot process?

The malicious MBR first decrypts code previously copied to disk sectors 2, 3 and 4 by the installer, hooks the real-mode INT13h (BIOS sector read-write services) interrupt handler and then passes execution to the original MBR code, backed up to the second sector (sector 1) by the installer. Similar to other known MBR bootkits, when the INT13h interrupt handler is invoked, hook code (located in sector 0) checks for service 0x02 (Read sectors from drive) and 0x42 (Extended read sectors from drive) being handled in order to intercept loading of bootmgr – the legacy version of the Windows Boot Manager. Note that ESPecter legacy versions do not need to patch the BmFwVerifySelfIntegrity function in bootmgr, because the bootmgr binary wasn't modified in any way.

From this point, the functionality of the boot code is almost the same as in the UEFI version, resulting in dropping the malicious driver (located contiguously on Track 0, starting at sector 6) into one of the following locations, depending on the architecture:

- \SystemRoot\System32\drivers\beep.sys (x86)
- \SystemRoot\System32\drivers\null.sys (x64)

In this case, the encrypted configuration is not dropped to the syslog file but stays hidden in sector 5 of the compromised disk.



	Sector 0	Sector 1	Sectors 2, 3 ,4	Sector 5	Sectors 6 - 63
Disk Track0	Malicious MBR + 16-bit real-mode INT13h hook code	Original MBR backup	Encrypted 32-bit & 64-bit malicious code (for bootmgr, winload and kernel hooks)	Encrypted configuration	Malicious kernel driver

Figure 9. Modified disk scheme used by the legacy ESPecter version

Kernel-mode driver

The driver's main purpose is to load user-mode payloads, set up the keylogger and, in the end, delete itself. Setting up the keylogger is done in two steps:

- At first, it creates a device named \Device\WebBK that exposes a function handling IRP_MJ_DEVICE_CONTROL requests from the user-mode components. This function supports one IOCTL (Input/Output Control) code (0x22C004), which can be used to trigger registration of an asynchronous procedure call routine responsible for processing intercepted keystrokes.
- Interception of keystrokes is done by setting up CompletionRoutine for IRP_MJ_READ requests for the keyboard driver object \Device\KeyboardClass0.

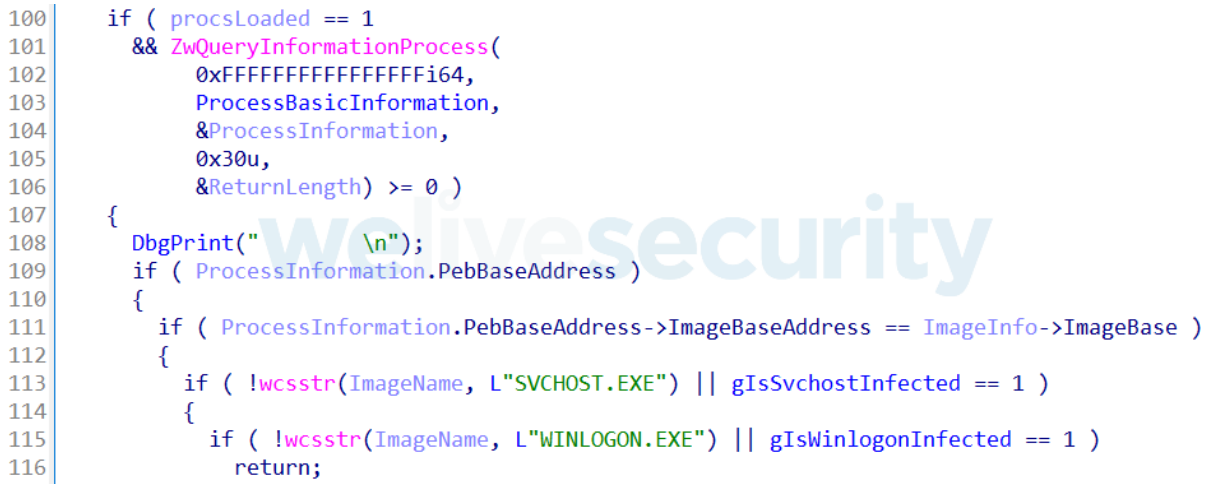
When done, any process can start logging intercepted keystrokes by defining its own routine and passing it to the created device object using the custom IOCTL 0x22C004.

By default, the driver tries to load two base payloads – WinSys.dll and Client.dll – which have the ability to download and execute additional payloads. The first one, WinSys.dll, is an MPRESS-packed DLL embedded in the driver's binary in an encrypted form. The second one, Client.dll, is downloaded by WinSys.dll to the file \SystemRoot\Temp\memlog, also in an encrypted form, using the same encryption method – a simple one-byte XOR with subtraction – but not the same keys. Both libraries are decrypted and dropped to the system directory \SystemRoot\System32\ by the driver.

Execution of both WinSys.dll and Client.dll libraries is achieved by injecting them into svchost.exe and winlogon.exe, respectively. To do this, the driver registers the image load callback routine NotifyRoutine using PsSetLoadImageNotifyRoutine, which is used to execute:

- The MainThread export from Client.dll, in context of the winlogon.exe process
- The MainThread export from WinSys.dll, in context of the svchost.exe process

NotifyRoutine hooks the entry point of the winlogon.exe and svchost.exe process images in memory before being executed; this hook is then responsible for loading and executing the appropriate payload DLL. As shown in Figure 10, only the first svchost.exe or winlogon.exe image being loaded is processed by the routine.



```
  if ( procsLoaded == 1
    && ZwQueryInformationProcess(
         0xFFFFFFFFFFFFFFFFi64,
         ProcessBasicInformation,
         &ProcessInformation,
         0x30u,
         &ReturnLength) >= 0 )
  {
    DbgPrint("        \n");
    if ( ProcessInformation.PebBaseAddress )
    {
      if ( ProcessInformation.PebBaseAddress->ImageBaseAddress == ImageInfo->ImageBase )
      {
        if ( !wcsstr(ImageName, L"SVCHOST.EXE") || gIsSvchostInfected == 1 )
        {
          if ( !wcsstr(ImageName, L"WINLOGON.EXE") || gIsWinlogonInfected == 1 )
            return;
```

Figure 10. Hex-Rays decompiled NotifyRoutine checking if svchost.exe or winlogon.exe is being loaded

User-mode components – WinSys.dll

WinSys.dll acts as a base update agent, which periodically contacts its C&C server in order to download or execute additional payloads or execute simple commands. The C&C address, along with other values like campaign ID, bootkit version, time between C&C communication attempts and active hours range, are located in the configuration, which can be loaded from:

- DefaultConfig value in HKLM\SYSTEM\CurrentControlSet\Control registry
- \SystemRoot\Temp\syslog file
- or directly from the specific disk sector (in the Legacy Boot version)

If both registry- and disk-stored configurations exist, the one from the registry is used.

C&C communication

WinSys.dll communicates with its C&C using HTTPS and the communication is initiated by sending an HTTP GET request using the following URL format:

https://<ip>/Heart.aspx?ti=<drive_ID>&tn=<campaign_ID>&tg=<number>&tv=<malware_version>

where drive_ID is the MD5 hash of the serial number of the main system volume and the other parameters are further information identifying this instance of the malware.

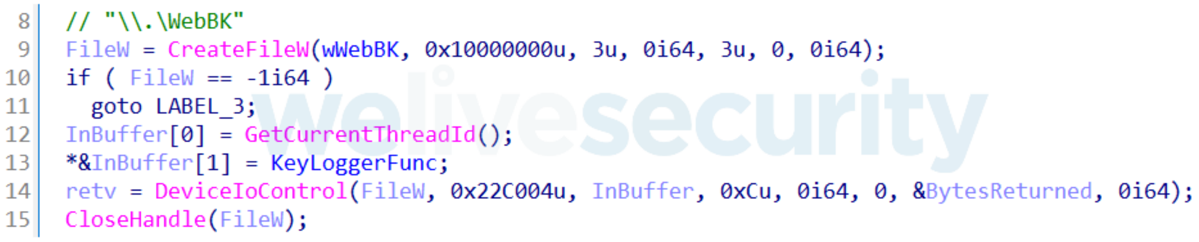
As a result, the C&C can respond with the command ID represented as a string, optionally followed by command parameters. The full list of commands is available in Table 1.

Table 1. WinSys component C&C commands

Command ID	Description	URL
1 or 4	Exit.	-
2	Upload various system info (CPU name, OS version, memory size, ethernet MAC address, list of installed software, etc.) to the predefined URL using the HTTP POST.	https://<ip>/GetSysteminfo.aspx
3	Download or download and execute file into one of the predefined locations (listed in *IoCs*) from the predefined URL using HTTP GET.	https://<ip>/UpLoad.aspx?ti=<drive_ID>
5	Restart PC via ExitProcess (for Windows Vista only).	N/A
6	Download new configuration from the predefined URL using HTTP GET and save it in the registry.	https://<ip>/ModifyIpaddr.aspx?ti=<drive_ID>

User-mode components – Client.dll

The second payload deployed by the malicious driver, if available, is Client.dll. It's a backdoor that supports a rich set of commands (Table 2) and contains various automatic data exfiltration capabilities including document stealing, keylogging, and monitoring of the victim's screen by periodically taking screenshots. All of the collected data is stored in a hidden directory, with separate subdirectories for each data source – the full list of directory paths used is available from our GitHub repository. Also note that interception of the keyboard is handled by the driver and the client just needs to register its logging function by sending IOCTL 0x22C004 to the driver's device in order to save intercepted keystrokes to the file (Figure 11).



```
// "\\.\WebBK"
FileW = CreateFileW(wWebBK, 0x10000000u, 3u, 0i64, 3u, 0, 0i64);
if ( FileW == -1i64 )
  goto LABEL_3;
InBuffer[0] = GetCurrentThreadId();
*&InBuffer[1] = KeyLoggerFunc;
retv = DeviceIoControl(FileW, 0x22C004u, InBuffer, 0xCu, 0i64, 0, &BytesReturned, 0i64);
CloseHandle(FileW);
```

Figure 11. Client payload setting up keylogger function by sending IOCTL to the bootkit's device driver

Configuration for the Client component should be located in an encrypted form in the file's overlay. It contains information such as the C&C address and port, flags indicating what data should be collected (keystrokes, screenshots, files with specific extensions), time period for the screenshotting thread, maximum file size for exfiltrated data and a list of file extensions.

C&C communication

The client sets its own communication channel with the C&C. For communication with the C&C, it uses the TCP protocol with single-byte XOR encryption applied to non-null message bytes that are different from the key, which was 0x66 in the campaign analyzed here. Communication is initiated by sending beacon messages to the IP:PORT pair specified in the configuration. This message contains the drive_ID value (the MD5 hash of the serial number of the main system volume) along with a value specifying the type of message – that is, a command request or the uploading of collected data.

After execution of the C&C command, the result is reported back to the C&C specifying the result code of the executed operation, command ID and, interestingly, each such resulting report message contains a watermark/tag representing the wide string WBKP located at offset 0x04, which makes it easier to identify this malicious communication at the network level.

Table 2. List of Client C&C commands

Command ID	Description
0x0000	Stop backdoor.
0x0064	Execute command line received from C&C and capture output using pipes.
0x00C8	Execute power commands logoff, power off, reboot, or shutdown, depending on the value of this C&C command's parameter.
0x012C	Take screenshot of foreground window, full screenshot, or change automatic screenshotting parameters, depending on the value of the parameter.
0x0190	Execute various file system operations.
0x01F4	Upload collected data and files.
0x0258	Execute various service-related commands.
0x02BC	Execute various process-related commands.
0x0320	Modify configuration values.
0x0384	Stop/start keylogger, depending on the value of the parameter.

Conclusion

ESPecter shows that threat actors are relying not only on UEFI firmware implants when it comes to pre-OS persistence and, despite the existing security mechanisms like UEFI Secure Boot, invest their time into creating malware that would be easily blocked by such mechanisms, if enabled and configured correctly.

To keep safe against threats similar to the ESPecter bootkit, make sure that:

- You always use the latest firmware version.
- Your system is properly configured and Secure Boot is enabled.
- You apply proper Privileged Account Management to help prevent adversaries from accessing privileged accounts necessary for bootkit installation.

Indicators of Compromise (IoCs)

A comprehensive list of IoCs and samples can be found in our GitHub repository.

ESET detections

EFI/Rootkit.ESPecter
Win32/Rootkit.ESPecter
Win64/Rootkit.ESPecter

C&C IP addresses and domains from configurations

196.1.2[.]111
103.212.69[.]175
183.90.187[.]65
61.178.79[.]69

swj02.gicp[.]net
server.microsoftassistant[.]com
yspark.justdied[.]com
crystalnba[.]com

Legacy version installers

ABC03A234233C63330C744FDA784385273AF395B
DCD42B04705B784AD62BB36E17305B6E6414F033
656C263FA004BB3E6F3EE6EF6767D101869C7F7C
A8B4FE8A421C86EAE060BB8BF525EF1E1FC133B2
3AC6F9458A4A1A16390379621FDD230C656FC444
9F6DF0A011748160B0C18FB2B44EBE9FA9D517E9
2C22AE243FDC08B84B38D9580900A9A9E3823ACF
08077D940F2B385FBD287D84EDB58493136C8391
1D75BFB18FFC0B820CB36ACF8707343FA6679863
37E49DBCEB1354D508319548A7EFBD149BFA0E8D
7F501AEB51CE3232A979CCF0E11278346F746D1F

Compromised Windows Boot Manager

27AD0A8A88EAB01E2B48BA19D2AAABF360ECE5B8
8AB33E432C8BEE54AE759DFB5346D21387F26902

MITRE ATT&CK techniques

This table was built using version 9 of the MITRE ATT&CK framework.

Tactic	ID	Name	Description
Execution	T1106	Native API	ESPecter leverages several Windows APIs: VirtualAlloc , WriteProcessMemory, and CreateRemoteThread for process injection.
Persistence	T1542.003	Pre-OS Boot: Bootkit	ESPecter achieves persistence by compromising Windows Boot Manager (bootmgfw.efi) located on the ESP, or by modifying the MBR on Legacy Boot systems.
T1547	Boot or Logon Autostart Execution	ESPecter replaces the legitimate null.sys or beep.sys driver with its own malicious one in order to be executed on system startup.	
Defense Evasion	T1055.001	Process Injection: Dynamic-link Library Injection	ESPecter’s driver injects its main user-mode components into svchost.exe and winlogon.exe processes.
T1564.001	Hide Artifacts: Hidden Files and Directories	ESPecter’s Client.dll component creates hidden directories to store collected data.	

Tactic	ID	Name	Description
T1564.005	Hide Artifacts: Hidden File System	ESPecter bootkit installers for Legacy Boot versions use unallocated disk space located right after the MBR to store its code, configuration and malicious driver.	
T1140	Deobfuscate/Decode Files or Information	ESPecter uses single-byte XOR with subtraction to decrypt user-mode payloads.	
T1562	Impair Defenses	ESPecter patches Windows kernel function directly in memory to disable Driver Signature Enforcement (DSE).	
T1036.003	Masquerading: Rename System Utilities	ESPecter bootkit installers for Legacy Boot versions copy cmd.exe to con1866.exe to evade detection.	
T1112	Modify Registry	ESPecter can use DefaultConfig value under HKLM\SYSTEM\CurrentControlSet\Control to store configuration.	
T1601.001	Modify System Image: Patch System Image	ESPecter patches various functions in Windows Boot Manager, Windows OS loader and OS kernel directly in memory during the boot process.	
T1027.002	Obfuscated Files or Information: Software Packing	ESPecter's WinSys.dll component is packed using the MPRESS packer.	
T1542.003	Pre-OS Boot: Bootkit	ESPecter achieves persistence by modifying Windows Boot Manager (bootmgfw.efi) located on the ESP or by modifying the MBR on Legacy Boot systems.	
T1553.006	Subvert Trust Controls: Code Signing Policy Modification	ESPecter patches Windows kernel function SepInitializeCodeIntegrity directly in memory to disable Driver Signature Enforcement (DSE).	
T1497.003	Virtualization/Sandbox Evasion: Time Based Evasion	ESPecter's WinSys.dll component can be configured to postpone C&C communication after execution or to communicate with the C&C only in a specified time range.	
Credential Access	T1056.001	Input Capture: Keylogging	ESPecter has a keylogging capability.
Discovery	T1010	Application Window Discovery	ESPecter's Client.dll component reports foreground window names along with keylogger information to provide application context.
T1083	File and Directory Discovery	ESPecter's Client.dll component can list file information for specific directories.	
T1120	Peripheral Device Discovery	ESPecter's Client.dll component detects the insertion of new devices by listening for the WM_DEVICECHANGE window message.	
T1057	Process Discovery	ESPecter's Client.dll component can list running processes and their loaded modules.	
T1012	Query Registry	ESPecter's WinSys.dll component can check for installed software under the Registry key HKLM\Software\Microsoft\Windows\CurrentVersion\Uninstall.	
T1082	System Information Discovery	ESPecter user-mode payloads can collect system information from the victim's machine.	

Tactic	ID	Name	Description
T1124	System Time Discovery	ESPecter’s WinSys.dll component can use GetLocalTime for time discovery.	
Collection	T1119	Automated Collection	ESPecter’s Client.dll component can automatically collect screenshots, intercepted keystrokes and various files.
T1025	Data from Removable Media	ESPecter’s Client.dll component can collect files with specified extension from removable drives.	
T1074.001	Data Staged: Local Data Staging	ESPecter’s Client.dll component stores automatically collected data into a hidden local directory.	
T1056.001	Input Capture: Keylogging	ESPecter has keylogging functionality.	
T1113	Screen Capture	ESPecter’s Client.dll component has screen capture functionality.	
Command and Control	T1071.001	Application Layer Protocol: Web Protocols	ESPecter’s WinSys.dll component communicates with its C&C server over HTTPS.
T1573.001	Encrypted Channel: Symmetric Cryptography	ESPecter’s Client.dll component encrypts C&C traffic using single-byte XOR.	
T1105	Ingress Tool Transfer	ESPecter’s user-mode components can download additional payloads from C&C.	
T1104	Multi-Stage Channels	ESPecter’s user-mode components use separate C&C channels.	
T1095	Non-Application Layer Protocol	ESPecter’s Client.dll component uses TCP for C&C communication.	
Exfiltration	T1020	Automated Exfiltration	ESPecter’s Client.dll component creates a thread to automatically upload collected data to the C&C.
T1041	Exfiltration Over C2 Channel	ESPecter exfiltrates data over the same channel used for C&C.	
T1029	Scheduled Transfer	ESPecter’s Client.dll component is set to upload collected data to the C&C every five seconds.	

5 Oct 2021 - 11:30AM

Sign up to receive an email update whenever a new article is published in our Ukraine Crisis – Digital Security Resource Center

Newsletter

Discussion