Boot Security

Why the Hardware Root of Trust Matters

whoami

Nicholas Starke

Senior Security Researcher at Hewlett Packard Enterprise

Professional Reverse Engineer

https://starkeblog.com/

Agenda

- How do computers boot?
- Common Attacks against the boot process
- Protections against these attacks
- Useful tools for Analysing UEFI images

Why Does Boot Security Matter?

Advanced Adversaries are increasingly “moving down the stack” towards compromise of low-level components as security measures protecting operating systems and bootloaders have improved.

(*https://thehackernews.com/2023/03/blacklotus-becomes-first-uefi-bootkit.html*)

Scope

Today’s talk will focus on Intel-based PC firmware

Primarily focused on x86 / x64 Instruction Set Architectures

UEFI does exist for ARM-based PCs

Open Source vs Closed Source

Most UEFI PC Firmware is closed-source

CoreBoot is an Open Source PC Firmware that supports some Motherboards

https://www.coreboot.org/

How Do Computers Boot?

1) Firmware

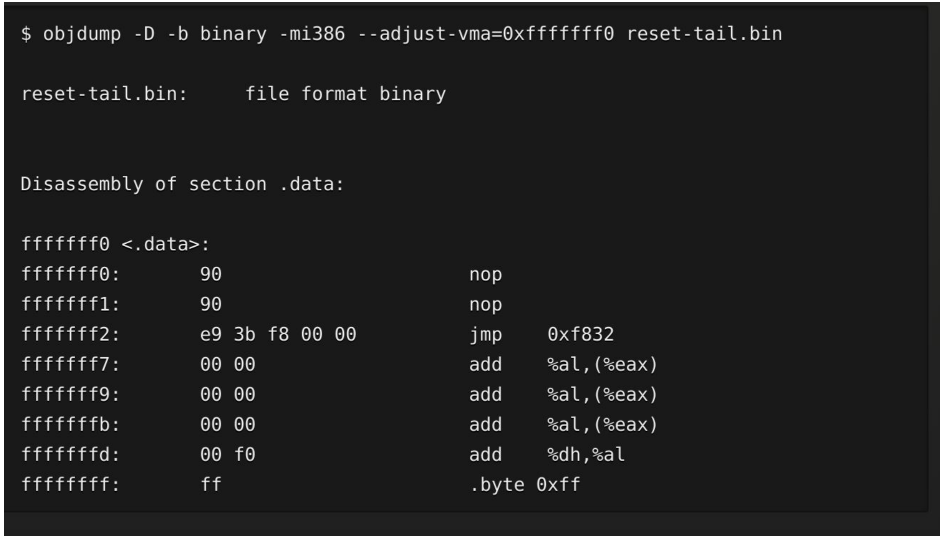
2) Bootloader

3) Operating System

Firmware - RESET Vector

Question: Where is the first instruction that a PC executes located in memory?

Firmware - RESET Vector



```
$ objdump -D -b binary -mi386 --adjust-vma=0xfffffff0 reset-tail.bin

reset-tail.bin:     file format binary


Disassembly of section .data:

fffffff0 <.data>:
fffffff0:       90                      nop
fffffff1:       90                      nop
fffffff2:       e9 3b f8 00 00          jmp    0xf832
fffffff7:       00 00                   add    %al,(%eax)
fffffff9:       00 00                   add    %al,(%eax)
fffffffb:       00 00                   add    %al,(%eax)
fffffffd:       00 f0                   add    %dh,%al
ffffffff:       ff                      .byte 0xff
```

https://starkeblog.com/uefi/binary/ghidra/2021/10/24/uefitool-board-init.html

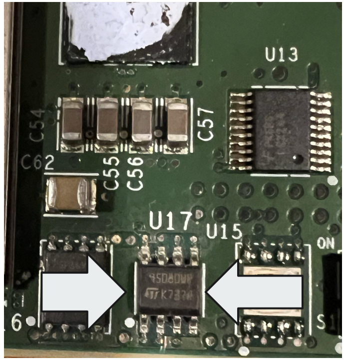
Reset Vector

Once the CPU is “Released from RESET”, the first instruction to be executed lives in memory at address **F:FFF0** - the last 16 bytes of the 32-bit address space.

These instructions are memory mapped to the last 512kb of the data on SPI ROM

What is SPI ROM?

- SPI ROM is a chip on the motherboard where the PC firmware code and corresponding EFI configuration values live.



What is PC Firmware?

- First x86 code to run after Power on.
- Responsible for constructing the memory map that the OS uses to access Hardware devices
- Lives on an SPI Chip on the motherboard

Legacy Boot

Traditionally, PC boot has been handled by BIOS

Basic Input Output System

- Every BIOS implementation was unique to the model of motherboard it ran on and implementations varied greatly from manufacturer to manufacturer
- Used Master Boot Record (MBR) format

Master Boot Record (MBR)

BIOS expects the Bootloader (GRUB/Winload) to exist in memory at **0:7c00** and transfers control of execution to the instructions starting at that address.

0:7c00 is mapped to the first sector of the first hard disk.

Sectors are 512 bytes in length

Limitations of MBR/BIOS

- Only Allows Four Partitions
- BIOS Runs in 16-Bit Real Mode
    - Debugging 16-bit Real Mode software is incredibly difficult
- BIOS had to be written in assembly

What is UEFI?

Universal Extensible Firmware Interface

A specification for PC firmware to handle the boot process.

(*https://uefi.org/specifications*)

UEFI Reference Implementation

TianoCore / EDK2

https://github.com/tianocore/edk2

UEFI Stages

1) Security Phase
2) Pre-Initialization Phase
3) Driver Execution Environment
4) Boot Device Selection
5) Transient System Load

UEFI Stages - Security Phase

“Initializes a temporary memory (often CPU cache as RAM, or SoC on-chip SRAM, CAR) and serves as the system's software root of trust with the option of verifying PEI before hand-off.”

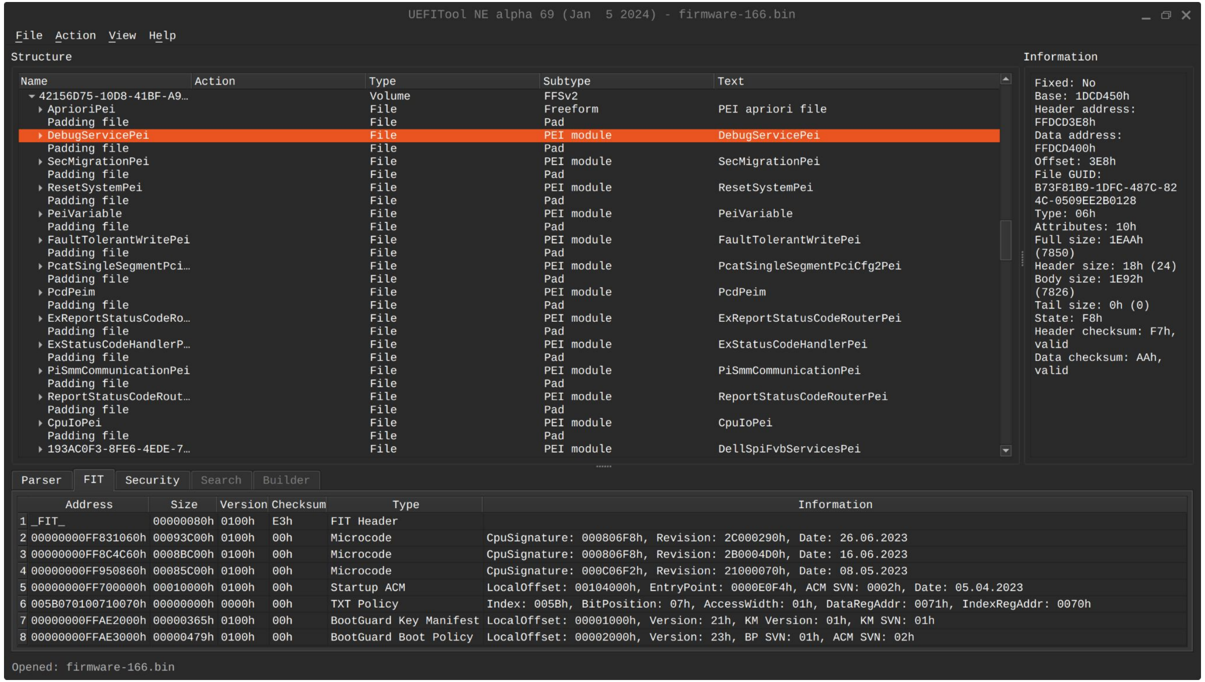
(*https://en.wikipedia.org/wiki/UEFI*)

UEFI Stages - PI

“The second stage of UEFI boot consists of a dependency-aware dispatcher that loads and runs PEI modules (PEIMs) to handle early hardware initialization tasks such as main memory initialization (initialize memory controller and DRAM) and firmware recovery operations.”

(*https://en.wikipedia.org/wiki/UEFI*)

UEFI Stages - PI



Name	Action	Type	Subtype	Text
42156D75-10D8-41BF-A9…		Volume	FFSv2	
AprioriPei		File	Freeform	PEI apriori file
Padding file		File	Pad	
DebugServicePei		File	PEI module	DebugServicePei
Padding file		File	Pad	
SecMigrationPei		File	PEI module	SecMigrationPei
Padding file		File	Pad	
ResetSystemPei		File	PEI module	ResetSystemPei
Padding file		File	Pad	
PeiVariable		File	PEI module	PeiVariable
Padding file		File	Pad	
FaultTolerantWritePei		File	PEI module	FaultTolerantWritePei
Padding file		File	Pad	
PcatSingleSegmentPci…		File	PEI module	PcatSingleSegmentPciCfg2Pei
Padding file		File	Pad	
PcdPeim		File	PEI module	PcdPeim
Padding file		File	Pad	
ExReportStatusCodeRo…		File	PEI module	ExReportStatusCodeRouterPei
Padding file		File	Pad	
ExStatusCodeHandlerP…		File	PEI module	ExStatusCodeHandlerPei
Padding file		File	Pad	
PiSmmCommunicationPei		File	PEI module	PiSmmCommunicationPei
Padding file		File	Pad	
ReportStatusCodeRout…		File	PEI module	ReportStatusCodeRouterPei
Padding file		File	Pad	
CpuIoPei		File	PEI module	CpuIoPei
Padding file		File	Pad	
193AC0F3-8FE6-4EDE-7…		File	PEI module	DellSpiFvbServicesPei

```
Fixed: No
Base: 1DCD450h
Header address:
FFDCD3E8h
Data address:
FFDCD400h
Offset: 3E8h
File GUID:
B73F81B9-1DFC-487C-82
4C-0509EE2B0128
Type: 06h
Attributes: 10h
Full size: 1EAAh
(7850)
Header size: 18h (24)
Body size: 1E92h
(7826)
Tail size: 0h (0)
State: F8h
Header checksum: F7h,
valid
Data checksum: AAh,
valid
```

	Address	Size	Version	Checksum	Type	Information
1	_FIT_	00000080h	0100h	E3h	FIT Header	
2	00000000FF831060h	00093C00h	0100h	00h	Microcode	CpuSignature: 000806F8h, Revision: 2C000290h, Date: 26.06.2023
3	00000000FF8C4C60h	0008BC00h	0100h	00h	Microcode	CpuSignature: 000806F8h, Revision: 2B0004D0h, Date: 16.06.2023
4	00000000FF950860h	00085C00h	0100h	00h	Microcode	CpuSignature: 000C06F2h, Revision: 21000070h, Date: 08.05.2023
5	00000000FF700000h	00010000h	0100h	00h	Startup ACM	LocalOffset: 00104000h, EntryPoint: 0000E0F4h, ACM SVN: 0002h, Date: 05.04.2023
6	005B070100710070h	00000000h	0000h	00h	TXT Policy	Index: 005Bh, BitPosition: 07h, AccessWidth: 01h, DataRegAddr: 0071h, IndexRegAddr: 0070h
7	00000000FFAE2000h	00000365h	0100h	00h	BootGuard Key Manifest	LocalOffset: 00001000h, Version: 21h, KM Version: 01h, KM SVN: 01h
8	00000000FFAE3000h	00000479h	0100h	00h	BootGuard Boot Policy	LocalOffset: 00002000h, Version: 23h, BP SVN: 01h, ACM SVN: 02h

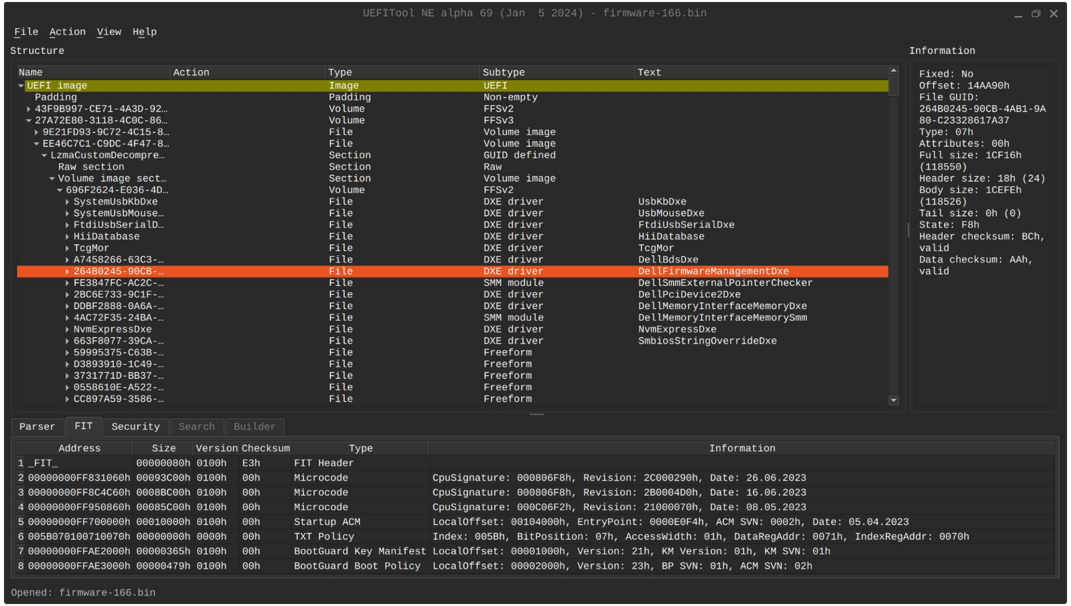
Opened: firmware-166.bin

UEFI Stages - DXE Phase

“This stage consist of C modules and a dependency-aware dispatcher. With main memory now available, CPU, chipset, mainboard and other I/O devices are initialized in DXE and BDS. Initialization at this stage involves assigning EFI device paths to the hardware connected to the motherboard, and transferring configuration data to the hardware.”

(*https://en.wikipedia.org/wiki/UEFI*)

UEFI Stages - DXE Phase



Name	Action	Type	Subtype	Text
UEFI image		Image	UEFI	
Padding		Padding	Non-empty	
43F9B997-CE71-4A3D-92…		Volume	FFSv2	
27A72E80-3118-4C0C-86…		Volume	FFSv3	
9E21FD93-9C72-4C15-8…		File	Volume image	
EE46C7C1-C9DC-4F47-8…		File	Volume image	
LzmaCustomDecompre…		Section	GUID defined	
Raw section		Section	Raw	
Volume image sect…		Section	Volume image	
696F2624-E036-4D…		Volume	FFSv2	
SystemUsbKbDxe		File	DXE driver	UsbKbDxe
SystemUsbMouse…		File	DXE driver	UsbMouseDxe
FtdiUsbSerialD…		File	DXE driver	FtdiUsbSerialDxe
HiiDatabase		File	DXE driver	HiiDatabase
TcgMor		File	DXE driver	TcgMor
A7458266-63C3-…		File	DXE driver	DellBdsDxe
264B0245-90CB-…		File	DXE driver	DellFirmwareManagementDxe
FE3847FC-AC2C-…		File	SMM module	DellSmmExternalPointerChecker
2BC6E733-9C1F-…		File	DXE driver	DellPciDevice2Dxe
DDBF2888-0A6A-…		File	DXE driver	DellMemoryInterfaceMemoryDxe
4AC72F35-24BA-…		File	SMM module	DellMemoryInterfaceMemorySmm
NvmExpressDxe		File	DXE driver	NvmExpressDxe
663F8077-39CA-…		File	DXE driver	SmbiosStringOverrideDxe
59995375-C63B-…		File	Freeform	
D3893910-1C49-…		File	Freeform	
3731771D-BB37-…		File	Freeform	
0558610E-A522-…		File	Freeform	
CC897A59-3586-…		File	Freeform	

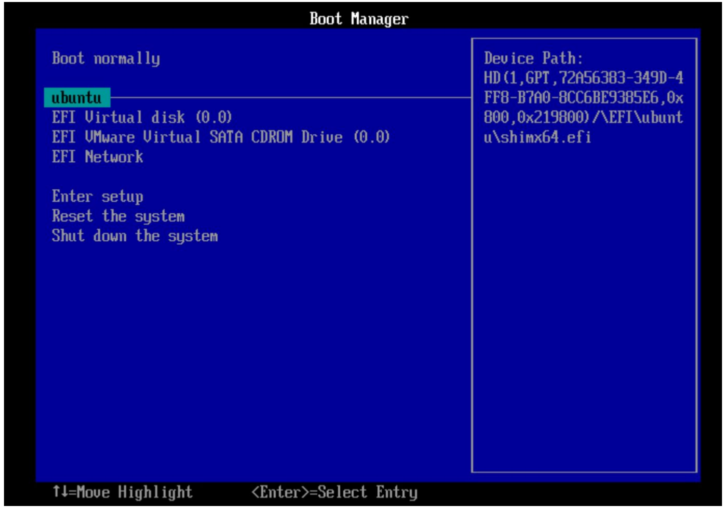
Fixed: No
Offset: 14AA90h
File GUID:
264B0245-90CB-4AB1-9A80-C23328617A37
Type: 07h
Attributes: 00h
Full size: 1CF16h (118550)
Header size: 18h (24)
Body size: 1CEFEh (118526)
Tail size: 0h (0)
State: F8h
Header checksum: BCh, valid
Data checksum: AAh, valid

	Address	Size	Version	Checksum	Type	Information
1	_FIT_	00000080h	0100h	E3h	FIT Header	
2	00000000FF831060h	00093C00h	0100h	00h	Microcode	CpuSignature: 000806F8h, Revision: 2C000290h, Date: 26.06.2023
3	00000000FF8C4C60h	0008BC00h	0100h	00h	Microcode	CpuSignature: 000806F8h, Revision: 2B0004D0h, Date: 16.06.2023
4	00000000FF950860h	00085C00h	0100h	00h	Microcode	CpuSignature: 000C06F2h, Revision: 21000070h, Date: 08.05.2023
5	00000000FF700000h	00010000h	0100h	00h	Startup ACM	LocalOffset: 00104000h, EntryPoint: 0000E0F4h, ACM SVN: 0002h, Date: 05.04.2023
6	005B070100710070h	00000000h	0000h	00h	TXT Policy	Index: 005Bh, BitPosition: 07h, AccessWidth: 01h, DataRegAddr: 0071h, IndexRegAddr: 0070h
7	00000000FFAE2000h	00000365h	0100h	00h	BootGuard Key Manifest	LocalOffset: 00001000h, Version: 21h, KM Version: 01h, KM SVN: 01h
8	00000000FFAE3000h	00000479h	0100h	00h	BootGuard Boot Policy	LocalOffset: 00002000h, Version: 23h, BP SVN: 01h, ACM SVN: 02h

Opened: firmware-166.bin

UEFI Phases - BDS

Boot device selected via EFI Configuration or manual user input



```
                          Boot Manager
Boot normally                                  Device Path:
                                               HD(1,GPT,72A56383-349D-4
ubuntu                                         FF8-B7A0-8CC6BE9385E6,0x
EFI Virtual disk (0.0)                         800,0x219800)/\EFI\ubunt
EFI VMware Virtual SATA CDROM Drive (0.0)      u\shimx64.efi
EFI Network

Enter setup
Reset the system
Shut down the system

↑↓=Move Highlight       <Enter>=Select Entry
```

UEFI - EFI System Partition (ESP)

DXE Phase contains a FAT32 driver that is capable of reading FAT32 partitions.

It finds the “ESP” and if configured to do so, executes an EFI program from that partition.

By Default, this is **/boot/efi/EFI/BOOT/BOOTx64.EFI**

UEFI - ESP Under Linux

```
nick@ubuntu-efi:~$ find /boot/efi -type f
/boot/efi/EFI/ubuntu/grubx64.efi
/boot/efi/EFI/ubuntu/shimx64.efi
/boot/efi/EFI/ubuntu/mmx64.efi
/boot/efi/EFI/ubuntu/BOOTX64.CSV
/boot/efi/EFI/ubuntu/grub.cfg
/boot/efi/EFI/BOOT/BOOTX64.EFI
/boot/efi/EFI/BOOT/fbx64.efi
/boot/efi/EFI/BOOT/mmx64.efi
nick@ubuntu-efi:~$
```

EFI Modules / EFI Applications

- EFI Modules and EFI Applications are PE32 Binaries - similar to Microsoft Windows Executables
- In contrast to Linux ELF files / Apple Mach-O files.

EFI Applications

- Shell.efi - EFI Shell
- Winload.efi - Windows Bootloader
- Grub2.efi - Linux Bootloader
- Shim.efi - Linux Secure Boot Shim

UEFI Hand-off to Bootloader

When EFI is complete, it calls an EFI function called **ExitBootServices** that then transfers control of execution to a bootloader.

For windows that is **winload.efi** and for Linux it is **grub2.efi**.

Grub2

GRand Unified Bootloader 2

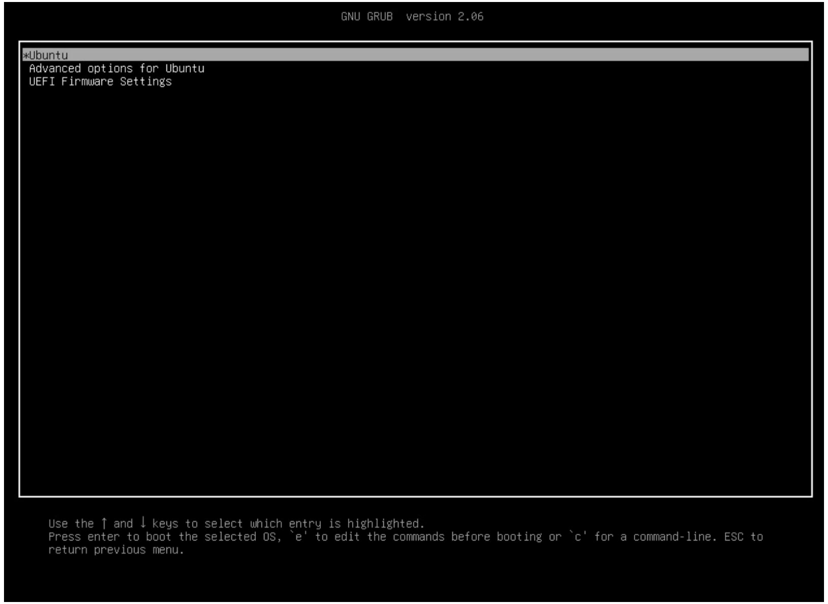
(*https://www.gnu.org/software/grub/index.html*)

- Standard Linux Bootloader (Equivalent of winload.efi in Windows)

Bootloader - What is it used for?

- Grub2 under EFI Allows the user to select an OS image to execute
- This allows for dual booting Windows/Linux on the same host
- Grub2 under BIOS handles the FAT32 partition parsing/loading.
- Responsible for starting kernel with proper command line parameters

Grub2



```
GNU GRUB  version 2.06

*Ubuntu
 Advanced options for Ubuntu
 UEFI Firmware Settings

   Use the ↑ and ↓ keys to select which entry is highlighted.
   Press enter to boot the selected OS, `e' to edit the commands before booting or `c' for a command-line. ESC to
   return previous menu.
```

Common Attacks Against the Boot Process

Typically, an attacker will want to subvert the boot process to accomplish two goals:

1) Achieve Persistence
2) Hide from Antivirus / EDR

Goal 1: Persistence

- Persistence across boot cycles (more prevalent in embedded products)
- Persistence across OS reinstallation

Goal 2: Hide from EDR / Antivirus

“He who executes first, executes Best” ~ Ancient OST2 Proverb (Xeno Kovah)

- Executing early in the boot process allows attackers to bypass or disable EDR solutions, as these solutions only have visibility into the primary operating system internals

Windows Platform Binary Table (WPBT)

The WPBT is a fixed Advanced Configuration and Power Interface (ACPI) table that enables boot firmware to provide Windows with a platform binary that the operating system can execute. The binary handoff medium is physical memory, allowing the boot firmware to provide the platform binary without modifying the Windows image on disk.

(*https://download.microsoft.com/download/8/a/2/8a2fb72d-9b96-4e2d-a559-4a27cf905a80/windows-platform-binary-table.docx*)

WPBT from an adversaries' Perspective

Allows DXE drivers to “embed” windows executables so that the executable is executed as SYSTEM upon OS initialization

Combine this with the ability to disable/bypass EDR and an adversary has everything they need to achieve persistence and stealth.

What about Linux?

Linux does not currently have an equivalent for WBPT, but there are still ways to accomplish the same goals.

- There are open source EFI Drivers for EXT2/3/4 filesystems that allow firmware to drop files onto a Linux filesystem.
- The firmware would need to parse/load the ext2 filesystem and write to a filesystem location that the chosen init system would pick up and execute.

Runtime Attacks

EFI exposes “Runtime Services” which may be executed after **ExitBootServices** is called and the operating system is loaded.

These sorts of attacks focus on executing code in **System Management Mode** (**SMM**)

System Management Mode (SMM)

Very privileged code execution level - provides the ability to modify the contents of SPI ROM.

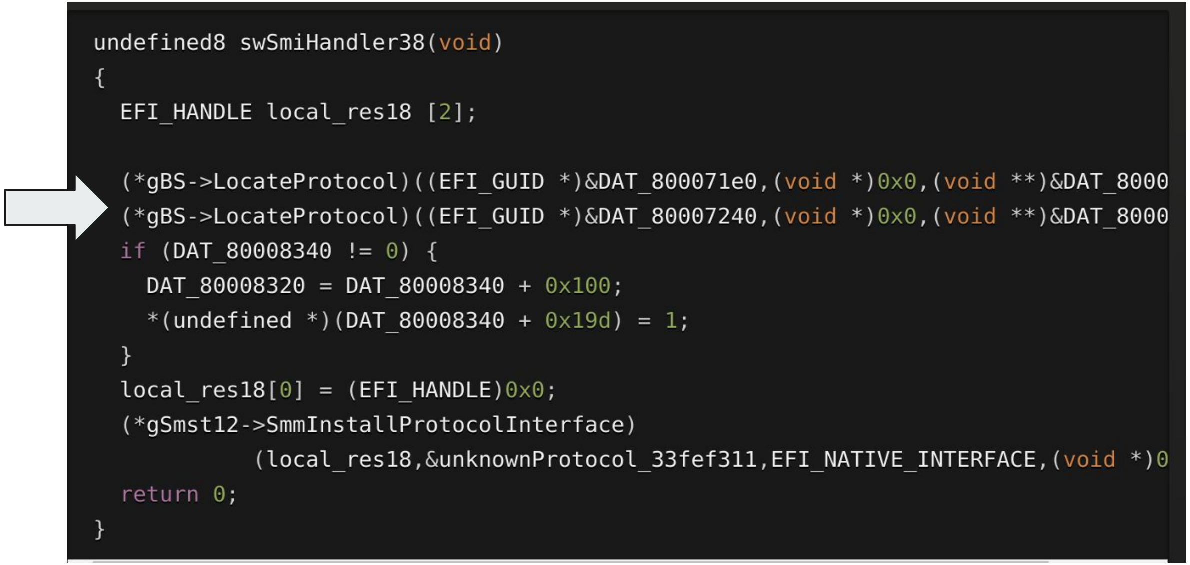
Modifying SPI ROM, where the firmware is located, can allow an attacker to potentially disable security features that are meant to protect the host from firmware attacks.

(*https://en.wikipedia.org/wiki/System_Management_Mode*)

SMM - Callout Attacks

- Occur when a System Management Mode Interrupt Handler attempts to call EFI Runtime Services or EFI Boot Services functions.
- These services are references by a global EFI struct that contains function pointers.
- The memory locations these function pointers point to can be overwritten in physical memory when executing in the context of the kernel (Kernel Drivers/Modules)

SMM - Callout Attacks



```
undefined8 swSmiHandler38(void)
{
  EFI_HANDLE local_res18 [2];

  (*gBS->LocateProtocol)((EFI_GUID *)&DAT_800071e0,(void *)0x0,(void **)&DAT_8000
  (*gBS->LocateProtocol)((EFI_GUID *)&DAT_80007240,(void *)0x0,(void **)&DAT_8000
  if (DAT_80008340 != 0) {
    DAT_80008320 = DAT_80008340 + 0x100;
    *(undefined *)(DAT_80008340 + 0x19d) = 1;
  }
  local_res18[0] = (EFI_HANDLE)0x0;
  (*gSmst12->SmmInstallProtocolInterface)
            (local_res18,&unknownProtocol_33fef311,EFI_NATIVE_INTERFACE,(void *)0
  return 0;
}
```

https://starkeblog.com/uefi/smm/2022/05/10/smm-callout-in-hp-products.html

Evil Maid Attacks

Occurs when an attacker has physical access to a device and can modify some aspect of the device so that the system owner does not know the device has been modified maliciously.

(*https://en.wikipedia.org/wiki/Evil_maid_attack*)

More SMM Attacks

- https://jjensn.com/at-home-in-your-firmware/
- https://github.com/tandasat/SmmExploit

Protections against Boot Attacks

Secure Boot is a protocol defined within UEFI

- Secure Boot allows the UEFI firmware to cryptographically validate each of the boot phases before control of execution is passed along to each boot phase.
- Upon BDS, Secure boot allows the UEFI firmware to validate the bootloader image to ensure it has not been modified from its original form.

How does Secure Boot Work?

- Uses the **Trusted Platform Module** (**TPM**) to verify the cryptographic hash of the code for the next stage in the boot process before that stage is executed
    - If the hash matches, the code has not been modified from its originally intended form and can be safely executed.
    - If the hash does not match, the code has been corrupted or maliciously modified and the firmware will then refuse to execute the code.

What is a TPM?

A TPM is a hardened component that is meant to provide protection from physical tampering.

The TPM has its own built-in cryptographic functions (encrypting/decrypting + hashing). This way, an attacker cannot modify, for example, the SHA256 hash function to return a hard coded hash value as a means to bypass certain integrity checks.

Secure Boot Chain

This chain of verifications form what is referred to as the **Hardware Root of Trust**.

SEC Verifies PEI

PEI Verifies DXE

DXE Verifies BDS

BDS Verifies Bootloader

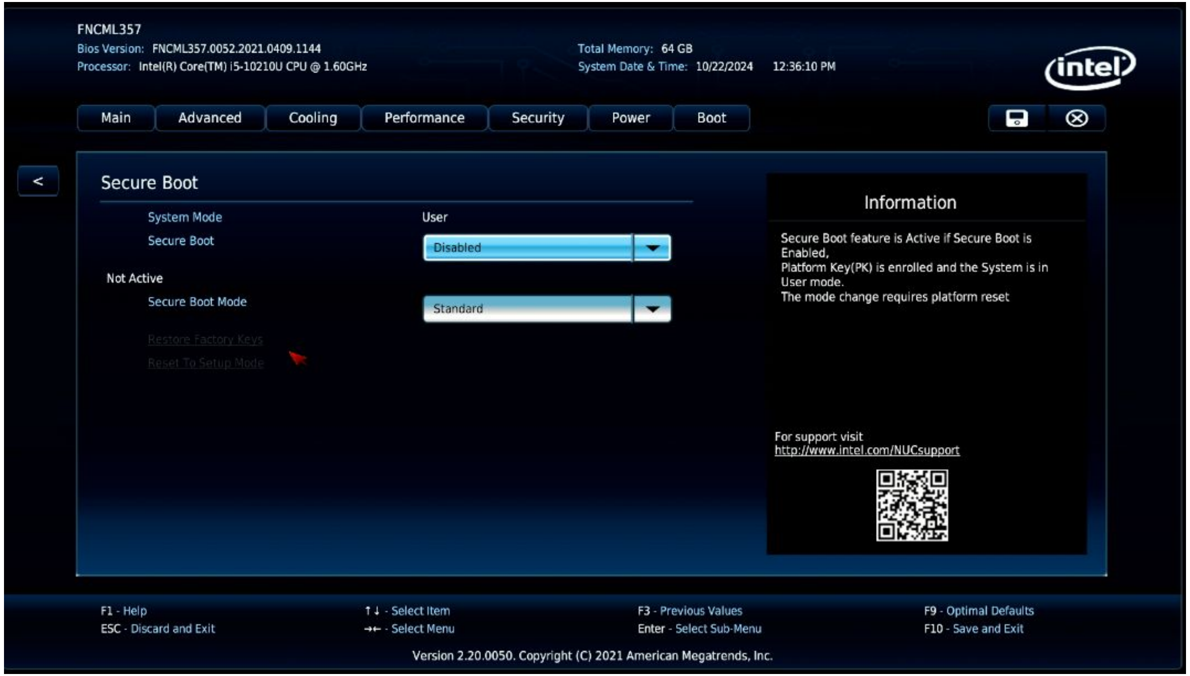
Bootloader Verifies OS

OS Verifies Kernel Drivers

But what verifies SEC?

In systems with Intel Bootguard, the SEC phase is verified by an Authenticated Code Module (ACM) that uses keys burned via One Time Programmable fuse in the Platform Configuration Hub (PCH)

In Systems without Intel Bootguard, the SEC phase is NOT protected.



FNCML357
Bios Version: FNCML357.0052.2021.0409.1144
Processor: Intel(R) Core(TM) i5-10210U CPU @ 1.60GHz
Total Memory: 64 GB
System Date & Time: 10/22/2024 12:36:10 PM

Secure Boot

System Mode: User
Secure Boot: Disabled

Not Active

Secure Boot Mode: Standard

Restore Factory Keys
Reset To Setup Mode

Information

Secure Boot feature is Active if Secure Boot is Enabled,
Platform Key(PK) is enrolled and the System is in User mode.
The mode change requires platform reset

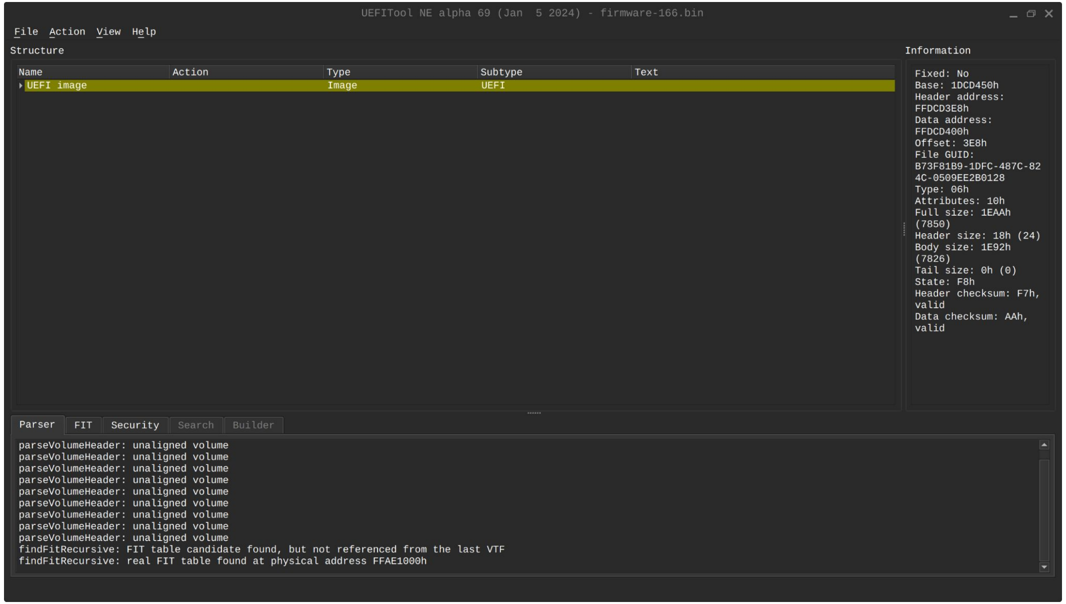
For support visit
http://www.intel.com/NUCsupport

Version 2.20.0050. Copyright (C) 2021 American Megatrends, Inc.

Tools for Analyzing UEFI Images

- UEFITool - https://github.com/LongSoft/UEFITool
- Ghidra - https://github.com/NationalSecurityAgency/ghidra
  - Ghidra-firmware-utils - https://github.com/al3xtjames/ghidra-firmware-utils
  - efiSeek - https://github.com/DSecurity/efiSeek
- IDAPro - https://hex-rays.com/ida-pro
  - efiXplorer - https://github.com/binarly-io/efiXplorer

UEFITool



Name	Action	Type	Subtype	Text
UEFI image		Image	UEFI	

```
Fixed: No
Base: 1DCD450h
Header address:
FFDCD3E8h
Data address:
FFDCD400h
Offset: 3E8h
File GUID:
B73F81B9-1DFC-487C-82
4C-0509EE2B0128
Type: 06h
Attributes: 10h
Full size: 1EAAh
(7850)
Header size: 18h (24)
Body size: 1E92h
(7826)
Tail size: 0h (0)
State: F8h
Header checksum: F7h,
valid
Data checksum: AAh,
valid
```

```
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
parseVolumeHeader: unaligned volume
findFitRecursive: FIT table candidate found, but not referenced from the last VTF
findFitRecursive: real FIT table found at physical address FFAE1000h
```

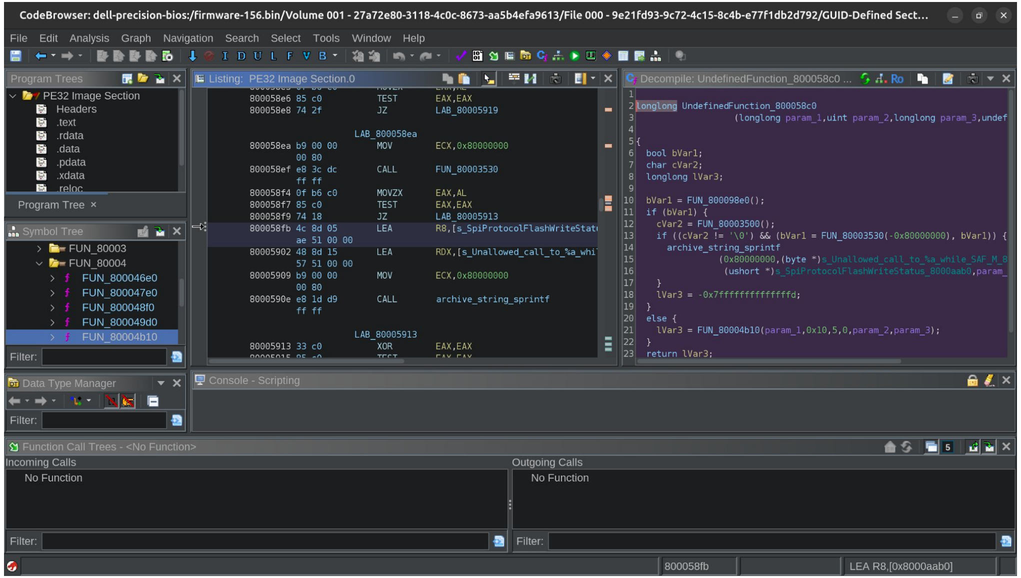
UEFITool

Useful for analyzing the contents of an EFI Image in terms of SEC/PI/DXE modules

Useful for extracting individual modules from a large EFI blob

Useful for searching for GUIDs and ASCII/Unicode strings in a monolithic EFI blob

Ghidra



```
800058e6 85 c0          TEST       EAX,EAX
800058e8 74 2f          JZ         LAB_80005919

                    LAB_800058ea
800058ea b9 00 00       MOV        ECX,0x80000000
         00 80
800058ef e8 3c dc       CALL       FUN_80003530
         ff ff
800058f4 0f b6 c0       MOVZX      EAX,AL
800058f7 85 c0          TEST       EAX,EAX
800058f9 74 18          JZ         LAB_80005913
800058fb 4c 8d 05       LEA        R8,[s_SpiProtocolFlashWriteStat
         ae 51 00 00
80005902 48 8d 15       LEA        RDX,[s_Unallowed_call_to_%a_whi
         57 51 00 00
80005909 b9 00 00       MOV        ECX,0x80000000
         00 80
8000590e e8 1d d9       CALL       archive_string_sprintf
         ff ff

                    LAB_80005913
80005913 33 c0          XOR        EAX,EAX
```

```
longlong UndefinedFunction_800058c0
                  (longlong param_1,uint param_2,longlong param_3,undef

{
  bool bVar1;
  char cVar2;
  longlong lVar3;

  bVar1 = FUN_800098e0();
  if (bVar1) {
    cVar2 = FUN_80003500();
    if ((cVar2 != '\0') && (bVar1 = FUN_80003530(-0x80000000), bVar1)) {
      archive_string_sprintf
                (0x80000000,(byte *)s_Unallowed_call_to_%a_while_SAF_M_8
                 (ushort *)s_SpiProtocolFlashWriteStatus_8000aab0,param_
    }
    lVar3 = -0x7ffffffffffffffd;
  }
  else {
    lVar3 = FUN_80004b10(param_1,0x10,5,0,param_2,param_3);
  }
  return lVar3;
```

Ghidra

Useful for analyzing the actual extracted modules

Provides disassembly and decompiler output (an approximation at a higher level, “Pseudo C” syntax

Additional Resources

Architecture 4001: x86-64 Intel Firmware Attack & Defense

Trusted Computing 1101: Introductory Trusted Platform Module (TPM) usage

Summary

The PC Boot Process is a complex endeavor

Adversaries are targeting this process to compromise hosts

Robust, cryptographically guaranteed countermeasures have come to market to mitigate the risks posed by attacks at such a low level

Thank you! Questions?

https://starkeblog.com/