UEFI BIOS Security

@ GLOBAL CYBERSECURITY CAMP 2022-01-16

Takeaways

- BIOS is just software
  - Written by humans
  - Can have bugs and vulnerabilities
  - Can run malicious code
- With differences of
  - Storage
  - Execution environment
  - Availability of security solutions for relevant threats
  - <u>A number of security engineers with a good understanding</u>
    - This is what this class helps for

Who am I

- Satoshi Tanda (@standa_t)
  - More than 10 years of cyber security experience (LinkedIn)
- A system software engineer
  - Developed end-point security software (AV/EDR)
    - @CrowdStrike (2016-)
    - @FFRI (2009-2012)
- A software reverse engineer
  - Discovered and weaponized vulnerabilities and analyzed malware
    - @Sophos (2015-2016)
    - @GE (2013-2014)
- A trainer
  - Teaches hypervisor development for security researchers (ref1, ref2)
- A speaker
  - CodeBlue、Recon、Bluehat、Nullcon、etc

UEFI BIOS Overview

What firmware is

- A type of software responsible for controlling hardware
  - Often on read-only memory (ROM) on embedded devices
- Bugs, vulnerabilities, malware are possible
- In x86_64 ecosystem,
  - Host firmware = BIOS (focus of this class)
  - Device firmware = eg,
    - Thunderbolt Controller - https://thunderspy.io/
    - HDD Controller - https://www.wired.com/2015/02/nsa-firmware-hacking/
    - USB Controller - https://shop.hak5.org/products/usb-rubber-ducky-deluxe
    - Baseboard management controller (BMC) - https://eclypsium.com/2019/01/26/the-missing-security-primer-for-bare-metal-cloud-services/

What BIOS is

- Broad sense: Software responsible for hardware init. and OS start up
  - Contains very first code that is ran by a CPU on system power up
  - Implements device drivers for loading OS
    - HTTP, PXE, HDD, USB, DVE etc etc
  - Executes the OS loader and hands over system up operations
- Strict sense: Legacy BIOS (vs. UEFI BIOS)
- Other “BIOS”:
    - iBoot - iPhone, macOS on Apple Silicon, macOS on T2 (Ref)
    - Coreboot – Chrome OS (Ref)
    - Linux Boot – Facebook datacenters (Ref)
    - Proprietary – Android phones

What UEFI is

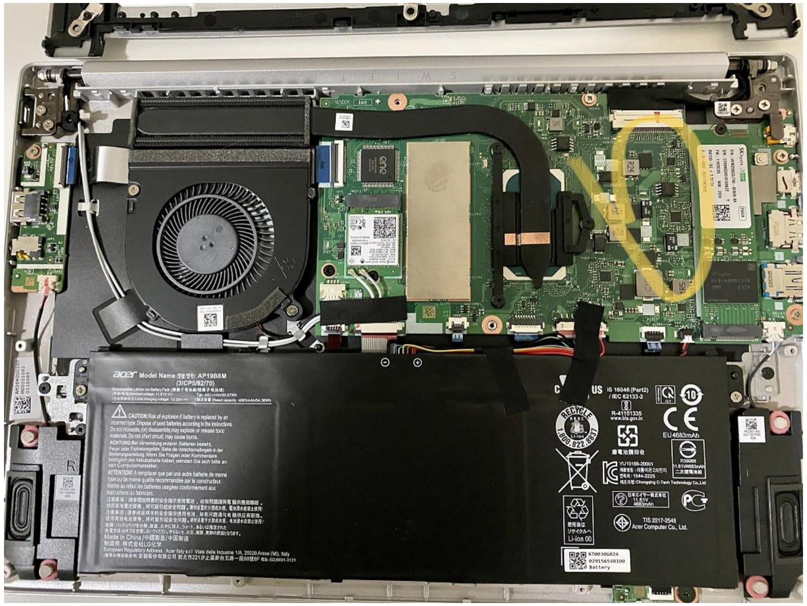
- The specification for BIOS to replace the legacy BIOS
- Legacy OS was difficult to
  - support other CPU architectures (eg, Itanium)
  - develop（16bit, real-mode, etc）
- 2005：EFI (v1.1)
- 2006：UEFI (v2.0)
  - All OEM PCs are now UEFI-based virtually
- As of this writing: UEFI v2.9 is the latest
  - https://uefi.org/specifications

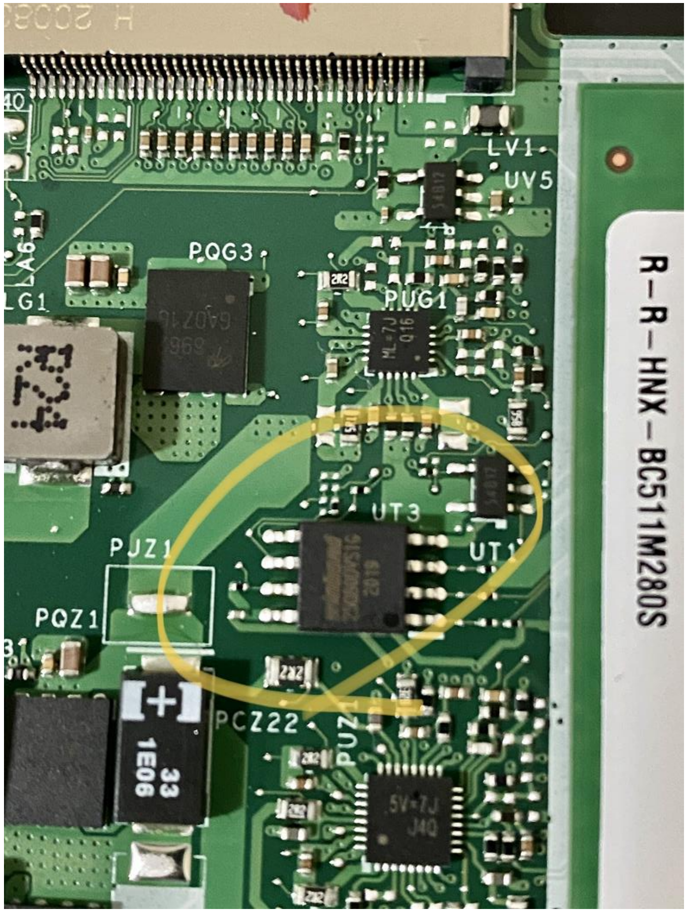
Demo: Checking UEFI spec version on VMware

CHECK OURPUT OF THE “VER” COMMAND ON THE UEFI SHELL

Storage for UEFI

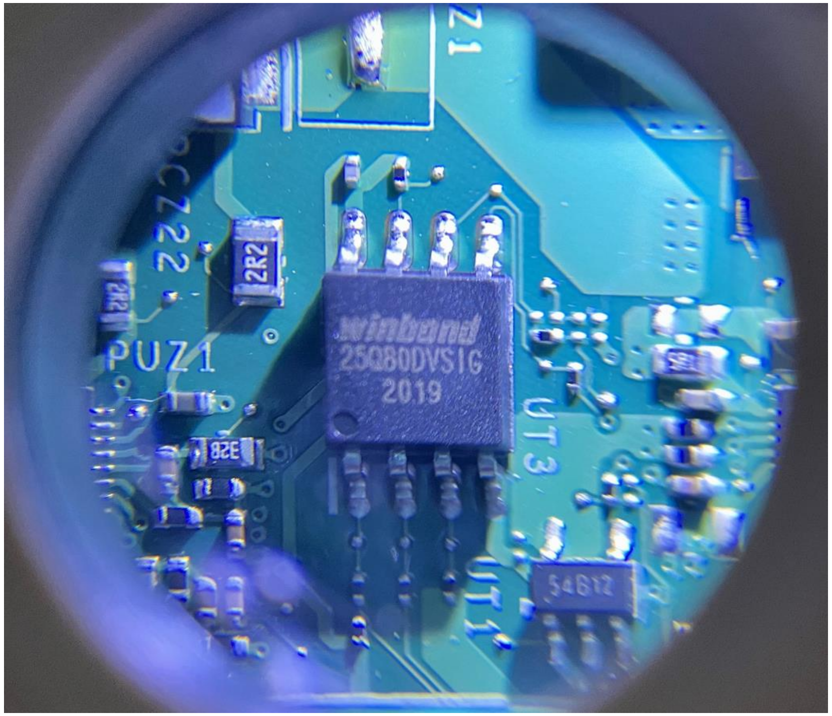
- Stored in a SPI flash
  - Separate storage from the HDD/SSD





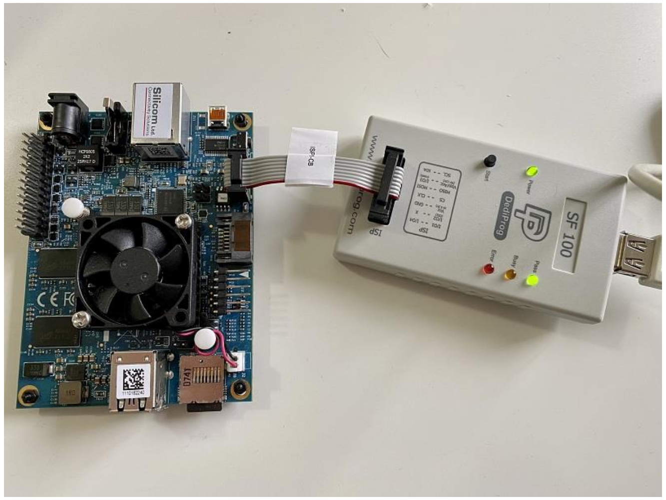
Storage for UEFI

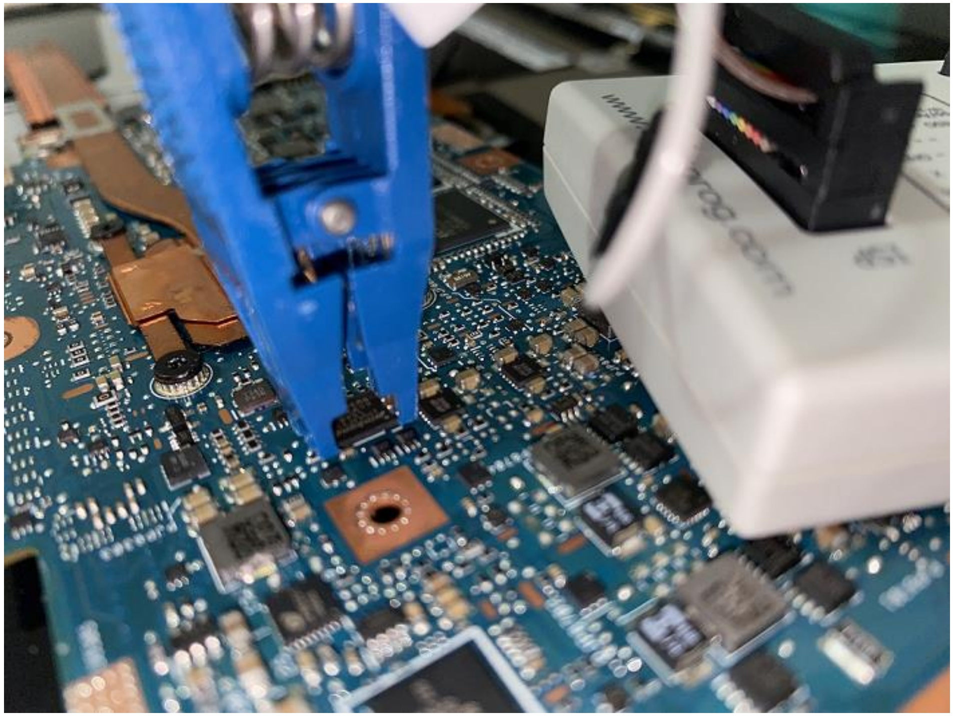
- Commonly manufactured by Winbond and is 8 or 16MB
- Example: [W25Q80DV]



How to access the SPI flash

- From software (later)
  - UEFI variables access and BIOS update are common scenarios
- With hardware
  - [SPI flash programmer]





Contents of the SPI flash（Intel）

- Multiple firmware
  - Eg, Gigabit Ethernet, Intel Management Engine, and BIOS

UEFITool NE alpha 58 (Nov 7 2020) - MBP141.rom

File Action View Help

Structure

Name	Action	Type	Subtype
Intel image		Image	Intel
Descriptor region		Region	Descriptor
PDR region		Region	PDR
ME region		Region	ME
BIOS region		Region	BIOS

UEFITool NE alpha 58 (Nov 7 2020) - XPS_9360_2.10.0.bin

File Action View Help

Structure

Name	Action	Type	Subtype
Intel image		Image	Intel
Descriptor region		Region	Descriptor
GbE region		Region	GbE
ME region		Region	ME
BIOS region		Region	BIOS

Contents of the BIOS image

- An Image contains one of more volumes
  - Volumes follows the Firmware File System (FFS) format (Ref)

Structure

Name	Action	Type	Subtype
AMI Aptio capsule		Capsule	Aptio signed
UEFI image		Image	UEFI
Padding		Padding	Non-empty
EfiFirmwareFileSystem2Guid		Volume	FFSv2
Padding		Padding	Empty (0xFF)
7DCCF422-45F6-4951-A557-CC421DA31599		Volume	FFSv2
4F1C52D3-D824-4D2A-A2F0-EC40C23C5916		Volume	FFSv2
AFDD39F1-19D7-4501-A730-CE5A27E1154B		Volume	FFSv2
61C0F511-A691-4F54-974F-B9A42172CE53		Volume	FFSv2

- A volume contains one of more files
- A file contains more than one sections

File Types

- A section may be an executable file
  - Format is either: Portable Executable (PE) or Terce Executable (TE)

Structure

Name	Action	Type	Subtype
AMI Aptio capsule		Capsule	Aptio signed
UEFI image		Image	UEFI
Padding		Padding	Non-empty
EfiFirmwareFileSystem2Guid		Volume	FFSv2
Padding		Padding	Empty (0xFF)
7DCCF422-45F6-4951-A557-CC421DA31599		Volume	FFSv2
4F1C52D3-D824-4D2A-A2F0-EC40C23C5916		Volume	FFSv2
414D94AD-998D-47D2-BFCD-4E882241DE32		File	Freeform
7B9A0A12-42F8-4D4C-82B6-32F0CA1953F4		File	Freeform
9E21FD93-9C72-4C15-8C4B-E77F1DB2D792		File	Volume image
LzmaCustomDecompressGuid		Section	GUID defined
Raw section		Section	Raw
Volume image section		Section	Volume image
5C60F367-A505-419A-859E-2A4FF6CA6FE5		Volume	FFSv2
AprioriDxe		File	Freeform
RomLayoutDxe		File	DXE driver
DxeCore		File	DXE core
Bds		File	DXE driver

Executable File Types

- Executable files are classified into one of the few types (Ref)
- Notable types

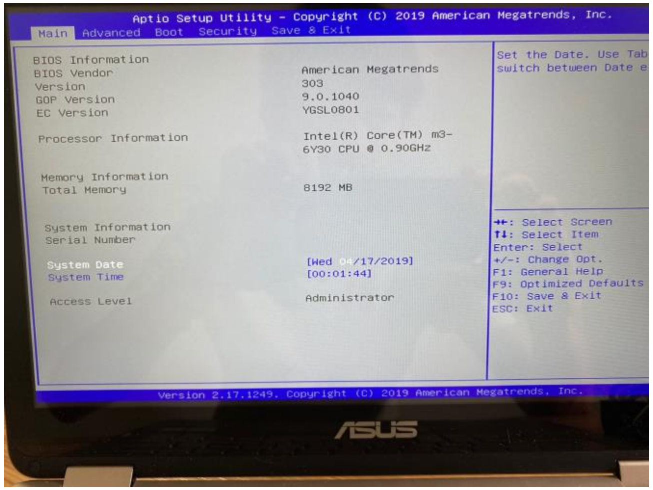
Type	Description	Example
Application	Gets started with StartImage() and unloaded when its execution finishes	• Shell.efi • SmmReset • SmmAccessSub
DXE Boot Driver	Gets started automatically or with the “load” command and remains on memory until the Runtime phase	• Almost all DXE drivers • SmmInterfaceBase • Ntfs
DXE Runtime Driver	Gets started automatically or with the “load” command and remains on memory indefinitely	• CRZEFI.efi • SecDxe
DXE SMM Driver	Get started automatically and remains on SMRAM indefinitely	• NvmeSmm

Demo: Checking executable file type

Using CFF Explorer, check the executable file type of the files used in exercises

Implementation of UEFI

- EDK2 – Reference implementation & SDK
  - Open Virtual Machine Firmware (OVMF) – Open source UEFI BIOS for QEMU
- OEM BIOS
  - is often: BIOS vendor（eg, AMI, Insyde）code + OEM original code
    - BIOS vendors use EDK2



Item	Value
OS Name	Microsoft Windows 10 Enterprise
Version	10.0.18363 Build 18363
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-HLKR4S8
System Manufacturer	ASUSTeK COMPUTER INC.
System Model	UX360CA
System Type	x64-based PC
System SKU	ASUS-NotebookSKU
Processor	Intel(R) Core(TM) m3-6Y30 CPU @ 0.90GHz, 1512 Mhz, 2 C...
BIOS Version/Date	American Megatrends Inc. UX360CA.303, 4/17/2019

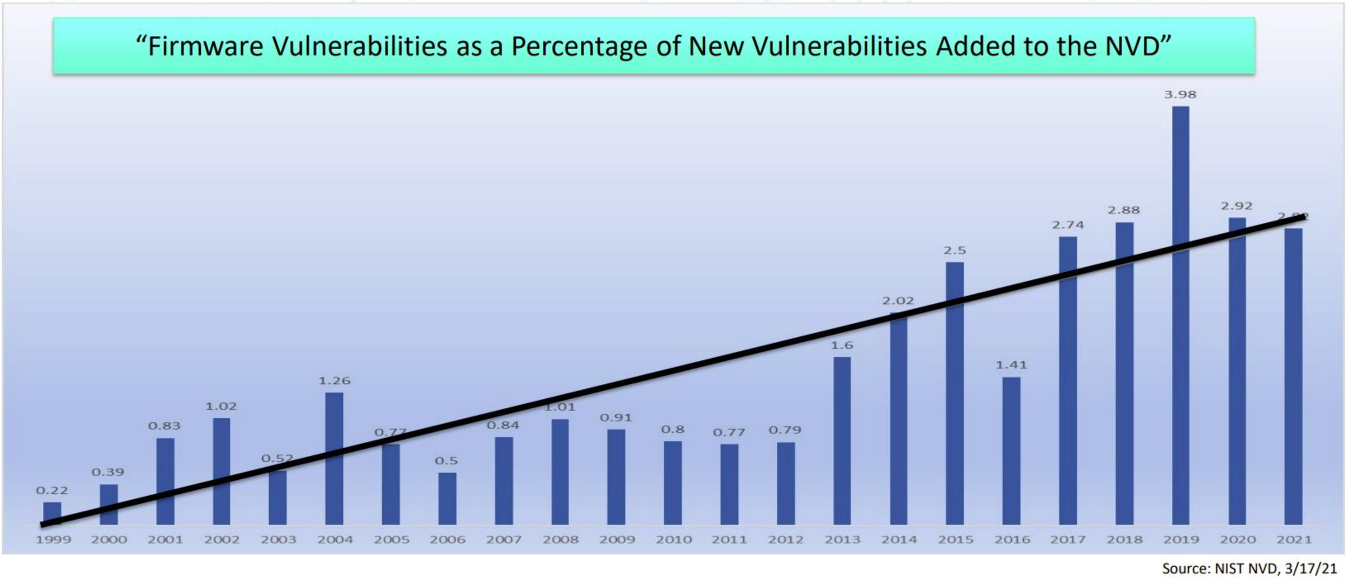
  - Some OEMs do not depend on BIOS vendors, eg, Microsoft, Apple (ref), Dell
    - Still based on EDK, eg, Mu(ref)

UEFI of VMware Workstation

- Somewhat based on EDK2
  - C:\Program Files (x86)\VMware\VMware Workstation\x64\EFI64.ROM
  - /usr/lib/vmware/roms/EFI64.ROM
  - Minimal implementation
    - 2MB (normally 8-16MB)
    - No SMM
- A custom BIOS file can be specified via the VMX file
  - efi64.filename = "MY.ROM“
  - Contents of a UEFI file can be changed with UEFITool
  - 👉 Could write and play with UEFI malware and UEFI antivirus

Firmware security

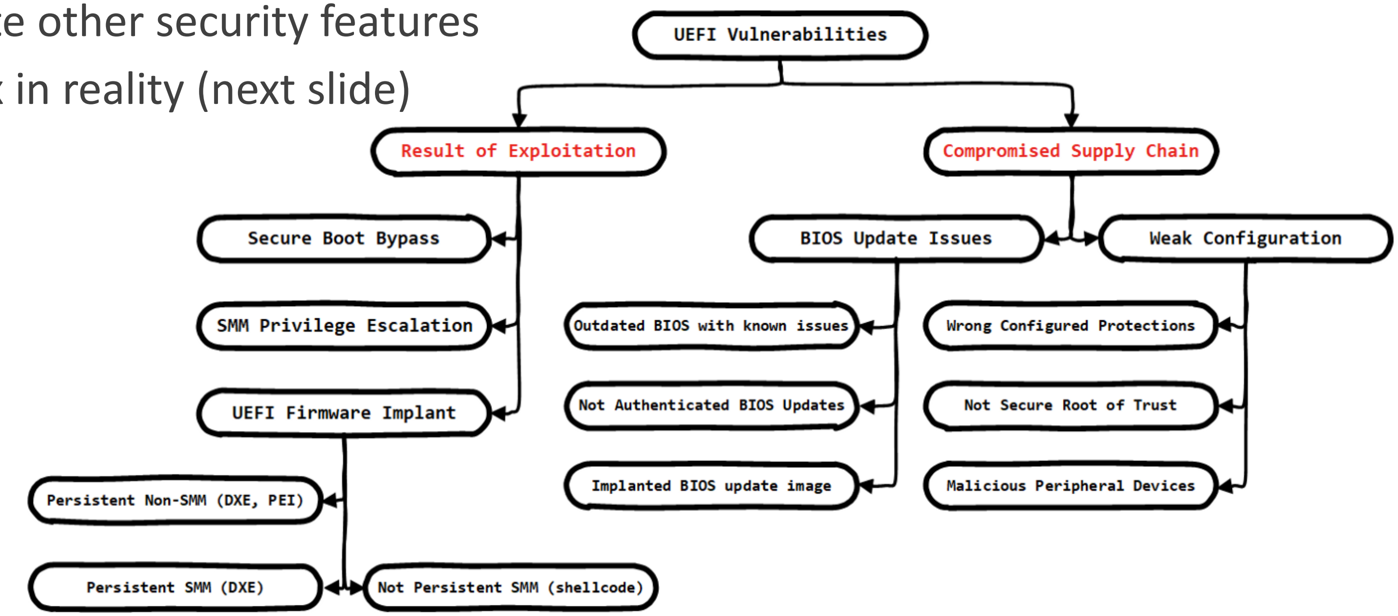
- Vulnerabilities and attacks became more common (ref)



- 80% of organizations confirmed attacks against firmware in the last 2 years, according with Microsoft (ref)

UEFI BIOS security

- Broad and complex（even limiting to “UEFI” BIOS）
  - Relates to multiple technology domains
  - Fewer engineers understand
- Severe impacts
  - May invalidate other security features
  - Difficult to fix in reality (next slide)
  - Widely used



https://medium.com/firmware-threat-hunting/uefi-vulnerabilities-classification-4897596e60af

UEFI supply chain problem

- Long and slow. *Usually* takes 6-9 months (ref)
- Example break down:
  - OEM：Confirms vuln. -> Develops a fix -> QA -> Release（3 months）
  - IT admins：Confirm published vuln. info. -> Plan for update -> Apply update
- Even slower if the vulnerability affects BIOS vendor’s or EDK2 code which is used by multiple OEMs

UEFI Hackz

Abusing UEFI

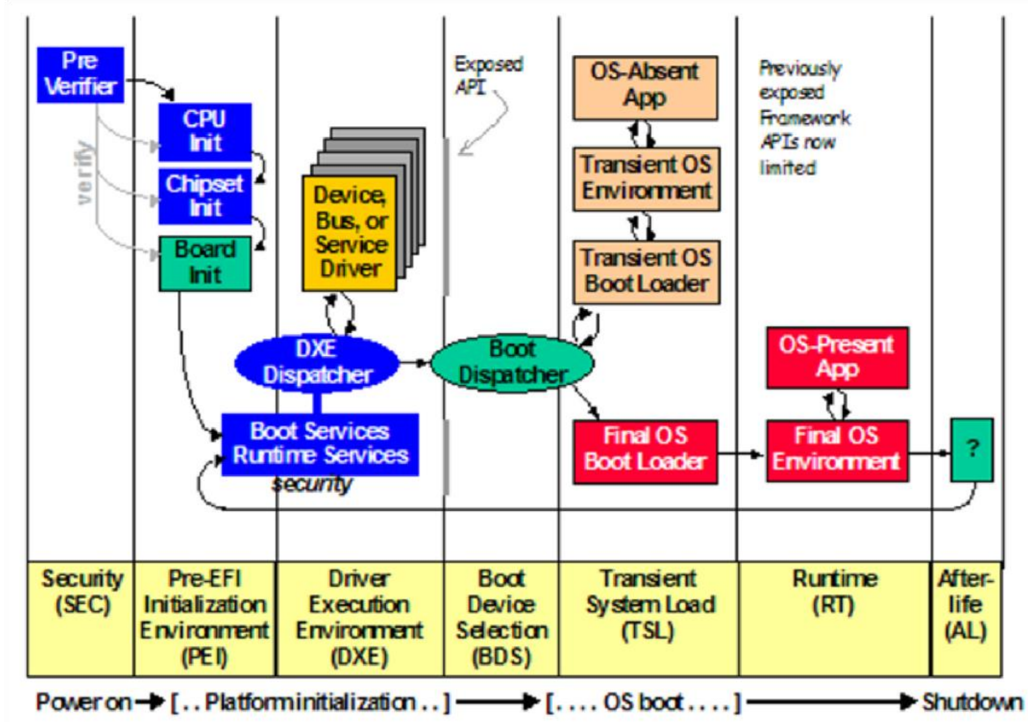
- Attack on game security with UEFI modules
  - CRZAIMBOT, and numerous examples in UnKnoWnCheaTs
- Hacking tools implemented as UEFI modules
  - Eg, assisting reverse engineering and other security research
  - EfiGuard, DmaBackdoorHv, negativespoofer, umap, efi-memory
- Why?

Reasons for abusing UEFI

- Runs at the kernel-mode
- Can bypass OS-level security policies
  - eg, Code signing policy on Windows
- Less likely to be detected
  - Starts before anti-cheat and anti-virus start
  - Fewer footprints
    - Nearly zero management structures known to OS, unlike kernel-mode modules
    - OS has very little knowledge about installed UEFI modules
- Easy to develop

System boot sequence (ref)

- PEI – Initializes hardware
- DXE – Sets up execution environment used until RT
- BDS – Shows the boot menu
- TSL – Executes an UEFI application selected through the boot menu. Primarily a boot loader
- RT – Discards the execution environment and hands over ownership of resources to the OS



Driver Execution Environment (DXE)

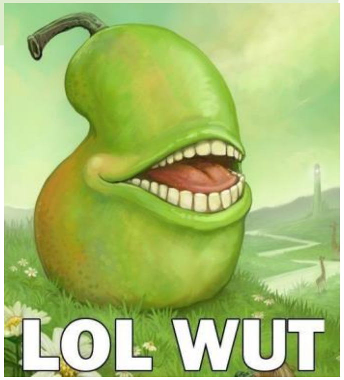
- Single threaded
- All modules run in the same address space
  - Except SMM modules, which run in a separate address space
- All code run as the kernel-mode
  - Except SMM modules, which run at even higher privilege
  - macOS on Intel processors uses user-mode (ref)

Driver Execution Environment (DXE)

- Long-mode（64bit addressing）is already enabled
  - Not 16bit real-mode
- Identity mapped paging
  - Virtual address is backed by the same physical address
  - eg：VA 0x123000 translates to PA 0x123000
    - No page-in/-out. Page faults causes the system to stop
  - Normally all pages are readable, writable and executable🥰

Adoption of security features (ref)

Security features	Adoption
Control flow guard	Intel CET-SS is supposed. Often unused anyway. Intel CET-IBT and compiler-based CFI（/guard:cf, -fsanitize=cfi）are unsupported.
NULL pointer access violation	Often unused
DEP / NX	Often unused
Stack canary	Unsupported
ASLR	Unsupported

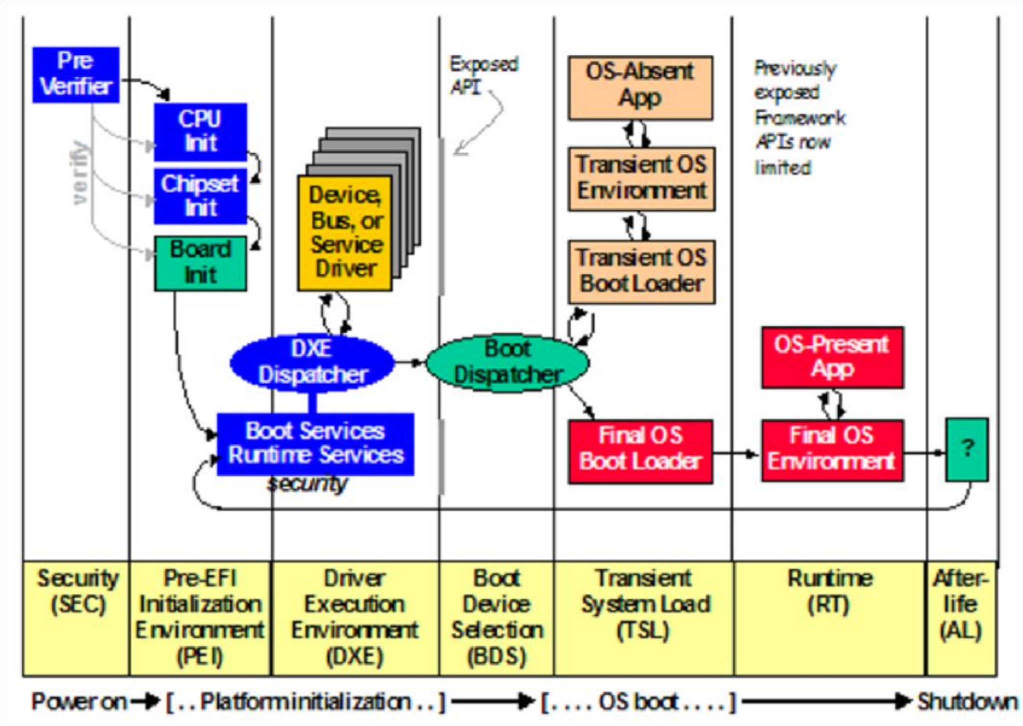


UEFI modules and OS interaction

- UEFI modules are normally:
  - For loading and starting OS
  - Get unloaded when OS starts
- UEFI hacking modules & malware need to interact with OS
  - Option 1：Be the DXE runtime drivers (exercise 1)
  - Option 2: Patch the boot loader and OS, and survive ("infect" to them)
    - eg: rainbow, Voyager
  - Option 3: Get called by OS (exercise 2 and 4)

Boot-time vs Run-time

- DXE - TLS = Boot-time
  - Starts both boot drivers and runtime drivers
  - All UEFI API are available
  - Whole address spaces is available
- RT = Run-time
  - Boot drivers are unloaded
  - Runtime drivers remain
  - Only minimal UEFI API is available
  - Only limited address space (runtime code and pool) is available
    - OS manages the rest



DXE runtime driver

- Runtime drivers keeps running even after OS starts

Type	Description	Example
Application	Gets started with StartImage() and unloaded when its execution finishes	• Shell.efi • SmmReset • SmmAccessSub
DXE Boot Driver	Gets started automatically or with the “load” command and remains on memory until the Runtime phase	• Almost all DXE drivers • SmmInterfaceBase • Ntfs
DXE Runtime Driver	Gets started automatically or with the “load” command and remains on memory indefinitely	• CRZEFI.efi • SecDxe
DXE SMM Driver	Get started automatically and remains on SMRAM indefinitely	• NvmeSmm

UEFI modules and OS interactions #2

- Hacking UEFI modules need to be called from the OS environment
  - ie, cannot just sit on memory
    - No timer or multi-threading API is available for this purpose
- Hooking the Runtime Services
  - Popular technique among those hacking UEFI modules

Boot & Runtime Services

- The UEFI core provides two sets of major API for UEFI modules
- Boot Services (BS)
  - Available between DXE - TSL
- Runtime Services (RT)
  - Available from DXE – indefinitely
  - eg: UEFI variable access, scheduling BIOS update, system shutdown

```
///
/// Cache pointer to the EFI Boot Services Table
///
extern EFI_BOOT_SERVICES  *gBS;
```

```
///
/// Cached copy of the EFI Runtime Services Table
///
extern EFI_RUNTIME_SERVICES  *gRT;
```

Runtime Services

- Runtime Services are called from the OS

```
1: kd> k
 # Child-SP          RetAddr               Call Site
00 ffff9007`ee5fa0a8 fffff801`570db27f     0xfffff801`5a45c000
01 ffff9007`ee5fa0b0 fffff801`570c40ce     nt!HalEfiGetEnvironmentVariable+0x53
02 ffff9007`ee5fa0f0 fffff801`574b3203     nt!HalGetEnvironmentVariableEx+0xf074e
03 ffff9007`ee5fa1e0 fffff801`574b26e0     nt!IopGetEnvironmentVariableHal+0x23
04 ffff9007`ee5fa220 fffff801`57569071     nt!IoGetEnvironmentVariableEx+0x94
05 ffff9007`ee5fa340 fffff801`57444b98     nt!ExpGetFirmwareEnvironmentVariable+0x8d
06 ffff9007`ee5fa390 fffff801`570247b5     nt!NtQuerySystemEnvironmentValueEx+0x13d7a8
07 ffff9007`ee5fa450 00007fff`56d4f804     nt!KiSystemServiceCopyEnd+0x25
08 00000019`5967d648 00007fff`5670548f     ntdll!NtQuerySystemEnvironmentValueEx+0x14
09 00000019`5967d650 00007fff`43912156     KERNEL32!GetFirmwareEnvironmentVariableExW+0x9f
```

- Runtime Services are accessible from boot-time &&
- All pages during boot-time are writable &&
- No runtime integrity check against runtime services
- 👉Runtime Services can be modified to execute hacking modules

Overview of CRZEFI

- Execution flow:
  - Efi_main()
    - SetServicePointer()
      - RT->SetVariable is replaced with HookedSetVariable
  - When SetVariable() is called:
    - HookedSetVariable()
      - mySetVariable()
        - RunCommand()
          - Back door commands

- Build environment
  - GNU-EFI – Lightweight SDK for building UEFI modules
    - Nonstandard. EDK2 is the standard environment

Advantages of UEFI modules

- Runs at the kernel-mode
- Can bypass OS-level security policies
    - 👉 If it were a Windows driver, it would have to be signed or exploit a vulnerability
- Less likely to be detected
    - Starts before anti-cheat and anti-virus start
        - 👉 If it were a Windows driver, it would have to use OS API to be loaded, which are more likely to be monitored by security software
    - Fewer footprints
        - 👉 If it were a Windows driver, it would create DRIVER_OBJECT and sit on OS-managed memory, which may be scanned by security software
- Easy to develop

Discussion: Potential mitigations

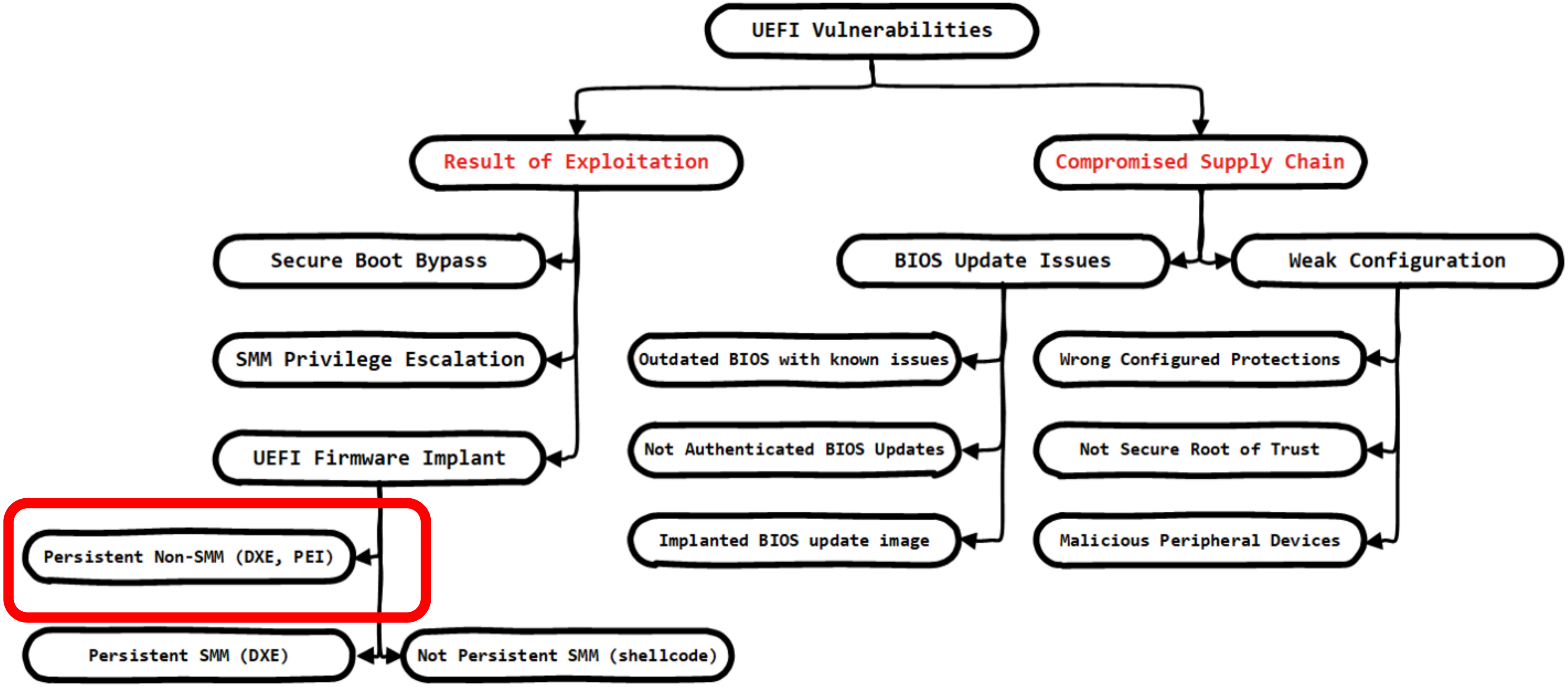
- Not starting the game unless secure boot is enabled?
  - Secure boot blocks loading of unsigned UEFI modules
  - Secure boot's threat model does not include a device owner
    - Often, secure boot is configurable to allow loading of arbitrary modules
    - Secure boot status may be falsely reported as enabled, while it is disabled (ref)
- Not starting the game unless boot events are known-good?
  - Trusted Platform Module (TPM) records loading of 3^{rd} party module into PCR[2] and TCG event logs (ref1, ref2)
  - Defining "known-good" is a challenge
- Memory scanning (ref) combined with byte-pattern match?

Abuse of UEFI modules by malware

- Installation would be the challenge
  - The device owner is the bad guy, in case of hacking tools
    - The owner would disable secure boot, boot the device from USB etc
  - The device owner is the good guy, in case of malware
- How would a remote attacker install UEFI malware?
  - Exploit vulnerabilities and write to the SPI flash (exercise 4)
    - Possible to detect and prevent, from the security software perspective
  - Overwrite a boot loader, eg, EFILock, ESPecter, FinFisher or the boot entry
    - Fairly easy to detect and prevent, from the security software perspective
- How an attacker with physical presence would install UEFI malware?
  - Attach the SPI flash programmer and install it onto the SPI flash

UEFI Malware Reverse Engineering

Category



https://medium.com/firmware-threat-hunting/uefi-vulnerabilities-classification-4897596e60af

Software reverse engineering 101

- An activity to investigate the implementation and internal of software
- Example purposes：Validating safety of using the software, understanding behaviour of malware
- Example methods：dynamic and static analysis
  - Dynamic analysis – Run the software and analyze its behaviour
    - Process Monitor, API Monitor, Debuggers
  - Static analysis – Disassemble the code and understand behaviour
    - Ghidra, IDA Pro, Binary Ninja

Why:
Reverse engineering UEFI modules

- UEFI modules are just software
  - Security problems are possible
  - Malware and Potentially Unwanted Application（PUA）are possible
  - 👉Same as the ordinary software, reverse engineering may be necessary
- Examples
  - Exercise 2, 4 – Malware implemented as UEFI modules
  - Exercise 3 – UEFI modules with vulnerabilities
  - Absolute Software（aka, Computrace, Lojack）is legitimate software, but
    - was criticized as rootkit due to its stealthy nature (ref). One might want to verify

What:
Reverse engineering UEFI modules

- Getting the target (ref)
  - From the SPI flash with software
    - eg : CHIPSEC
  - From the SPI flash with the SPI flash programmer
  - From BIOS update files
    - May be viewed with UEFITool (exercise 3)
    - May be extracted with BIOSUtilities
  - From VirtusTotal if a hash value is known (exercise 2 and 4)

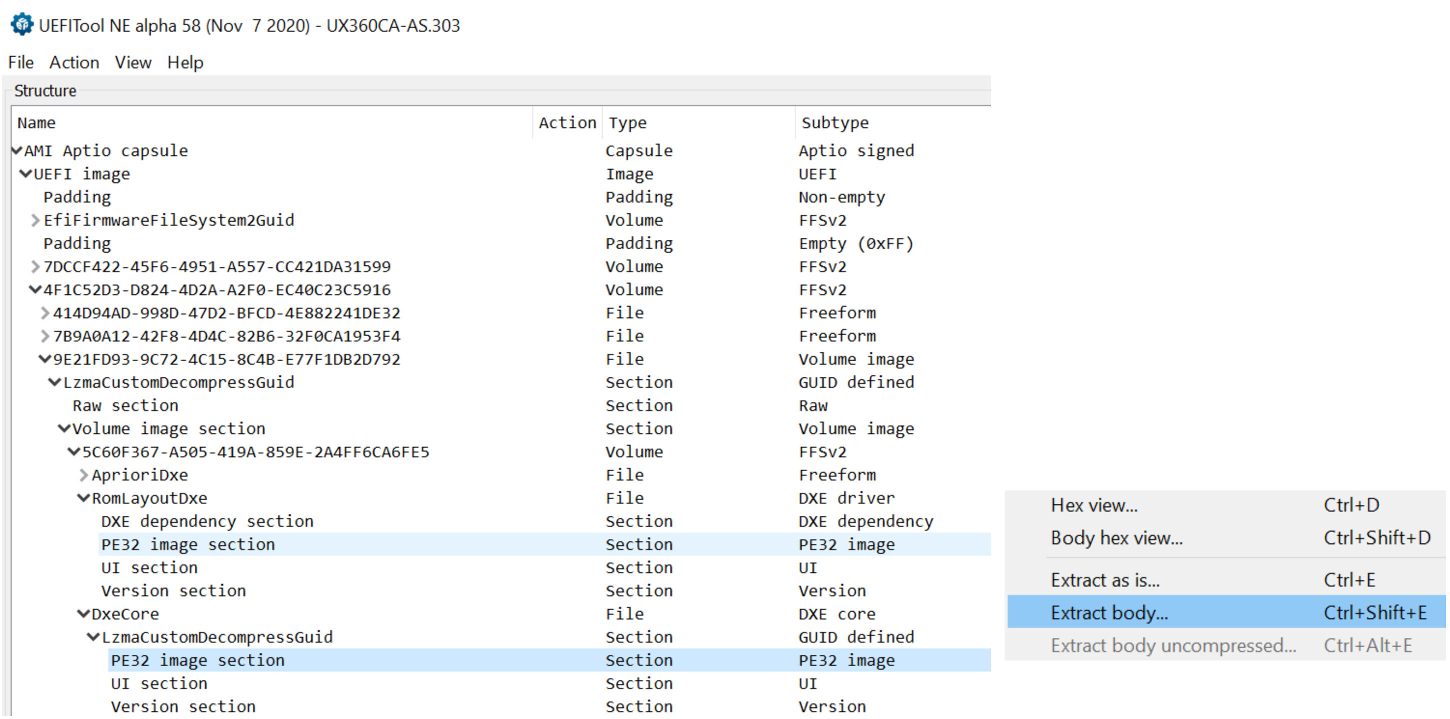
- SMM modules are easy targets for finding vulnerabilities

Structure

Name	Action	Type	Subtype
PiSmmCore		File	SMM core
FlashDriverSmm		File	SMM module
NvramSmm		File	SMM module
NvramSmi		File	SMM module
CpuIo2Smm		File	SMM module

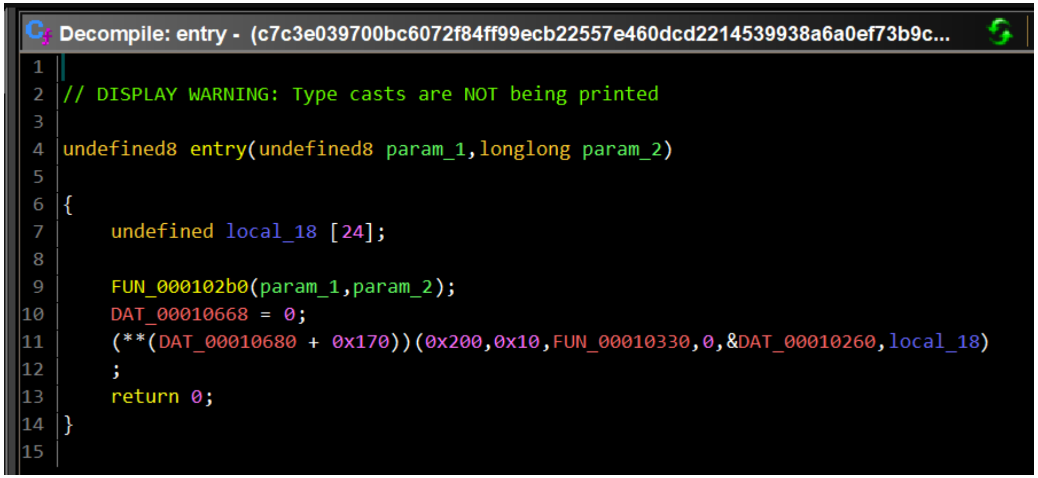
Extracting modules from UEFI blob

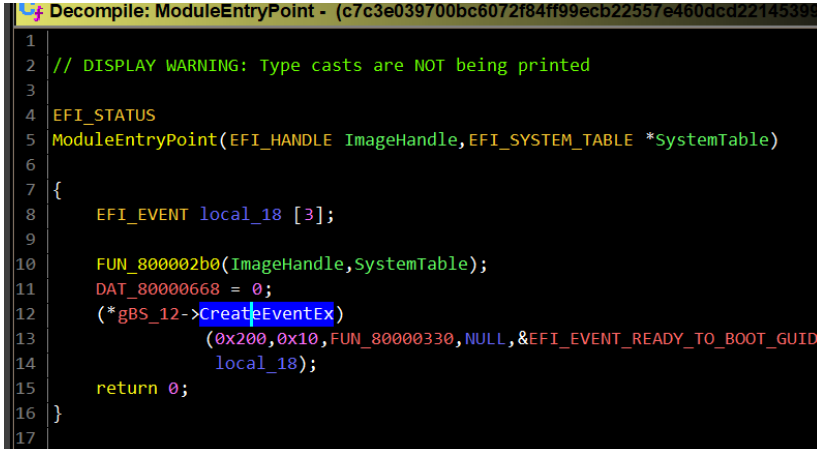
- Find the PE32 image sections and extract them with UEFITool
  - UEFI modules are identified by GUIDs
  - Human-friendly names (eg, DxeCore) is an optional attribute



How:
Reverse engineering UEFI modules

- Primarily static analysis
  - No dynamic analysis tool or framework
    - Possible to run in a VM and use VM debugger such as QEMU + GDB
  - No obfuscation tool
- Existing disassemblers + extensions work
  - Ghidra + efiSeek, IDA Pro + efiXplorer, Binary Ninja + bn-uefi-helper

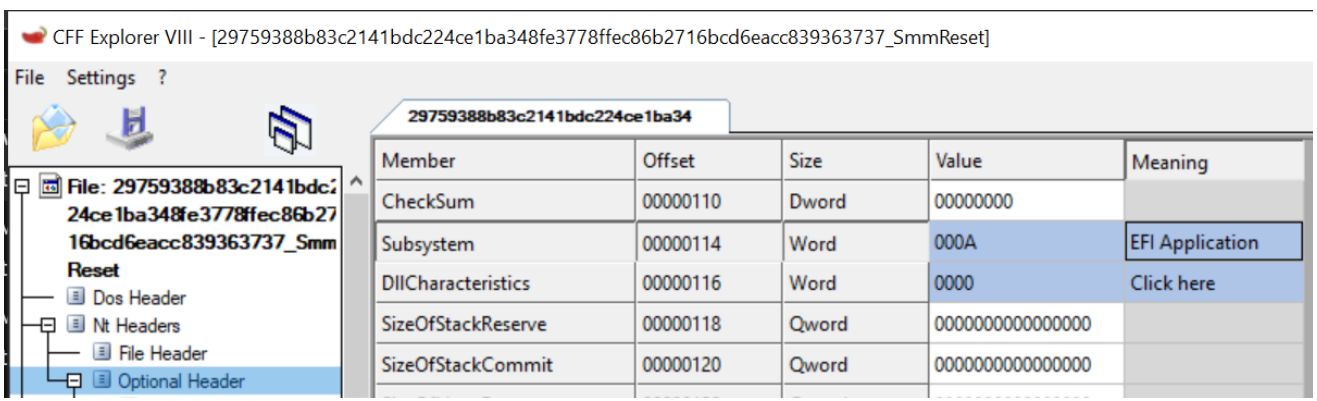




Surficial analysis of UEFI modules

- Modules that run after the DXE phase are PE files
- IMAGE_OPTIONAL_HEADER.SubSystem indicates the executable type (ref)

IMAGE_SUBSYSTEM_EFI_APPLICATION 10	Extensible Firmware Interface (EFI) application.
IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11	EFI driver with boot services.
IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12	EFI driver with run-time services.



Surficial analysis of UEFI modules

- Section and the AddressOfEntryPoint field are valid
- No import or export for UEFI modules
  - API resolutions are done at runtime (later)

UEFI module reverse engineering 101

- Entry point is either
  - Code written by a developer, or
  - Code generated by a library
- Examples of library code
  - Left : GNU-EFI（unoptimized）
  - Right : EDK2（unoptimized）

Decompile: ModuleEntryPoint - (memory.efi)

```
// DISPLAY WARNING: Type casts are NOT being printed

EFI_STATUS
ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)

{
    EFI_STATUS EVar1;
    undefined4 in_R8D;
    undefined4 in_R9D;
    EFI_HANDLE in_stack_00000018;

    FUN_80009af0();
    EVar1 = FUN_80003bd2(SystemTable,ImageHandle,in_R8D,in_R9D,in_stack_00000018
                        );
    return EVar1;
}
```

Decompile: ModuleEntryPoint - (7ea33696c91761e95697549e0b0f84db2cf4033216cd16c...

```
// DISPLAY WARNING: Type casts are NOT being printed

EFI_STATUS
ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)

{
    char cVar1;
    EFI_STATUS EVar2;
    EFI_STATUS extraout_RAX;
    void *local_10 [2];

    if ((DAT_80050460 == 0) || (DAT_80050460 <= (SystemTable->Hdr).Revision)) {
        FUN_8000040c(ImageHandle,SystemTable);
        if (DAT_8005045f != '\0') {
            EVar2 = (*gBS_174->HandleProtocol)
                              (ImageHandle,&EFI_LOADED_IMAGE_PROTOCOL_GUID,
                               local_10);
            cVar1 = FUN_800019dc();
            if ((cVar1 != '\0') && (EVar2 < 0)) {
                cVar1 = FUN_800019dc();
                if ((cVar1 != '\0') && (cVar1 = FUN_800019e0(), cVar1 != '\0'))
                {
                    FUN_800019b4();
                }
                FUN_800019cc();
            }
            *(local_10[0] + 0x58) = &LAB_800002a0;
        }
        FUN_800005bc(ImageHandle,SystemTable);
        EVar2 = extraout_RAX;
        if (extraout_RAX < 0) {
            FUN_800005b0();
        }
    }
    else {
        EVar2 = 0x8000000000000019;
```

UEFI module reverse engineering 101

- Entry point receives EFI_SYSTEM_TABLE*
  - Which contains EFI_BOOT_SERVICES* and EFI_RUNTIME_SERVICES*
- The following global variables are initialized

Name	Description
gST	A copy of EFI_SYSTEM_TABLE*
gBS	Boot Services
gRT	Runtime Services

Decompile: ModuleEntryPoint - (29759388b83c2141bdc224ce1ba348fe3778ff...

```
// DISPLAY WARNING: Type casts are NOT being printed

EFI_STATUS
ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)

{
    undefined local_res8 [8];
    undefined local_res10 [8];
    UINTN local_res18 [2];

    gBS_22 = SystemTable->BootServices;
    gRS_23 = SystemTable->RuntimeServices;
    local_res8[0] = 0;
    local_res18[0] = 1;
    gImageHandle_20 = ImageHandle;
    gST_21 = SystemTable;
```

UEFI module reverse engineering 101

- Boot Services offer basic functionalities (ref)
  - Memory management : AllocatePages(), FreePages() etc
  - Callback registration : CreateEvent(), SignalEvent(), etc
  - Image execution : LoadImage(), StartImage(), etc
  - Protocol resolution : OpenProtocol(), LocateProtocol(), etc

UEFI module reverse engineering 101

- Other API and data are accessed through the Protocol API
- Input: GUID
- Output: An associated data structure and function pointers
  - eg：getting details about the specified module（similar code seen in CRZEFI）

```
// Get handle to this image
EFI_LOADED_IMAGE* LoadedImage = NULL;
EFI_STATUS status = BS->OpenProtocol(ImageHandle, &LoadedImageProtocol,
    (void**)&LoadedImage, ...);

// Set our image unload routine
LoadedImage->Unload = (EFI_IMAGE_UNLOAD)efi_unload;
```

UEFI module reverse engineering 101

- Other API and data are accessed through the Protocol API
  - eg：Accessing a file（code seen in SmmAccessSub) 1/2

```
              efiStatus = (*gBS->HandleProtocol)
                                    (*ppvVar2,&EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID,
                                     &simpleFileSystemProtocol);
              if ((-1 < efiStatus) &&
                 (efiStatus2 = (**(simpleFileSystemProtocol + 8))
                                         (simpleFileSystemProtocol,&fileProcotol), -1 < efiStatus2)) {
                  gFileProcotol = fileProcotol;
```

```
struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
  ///
  /// The version of the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL. The version
  /// specified by this specification is 0x00010000. All future revisions
  /// must be backwards compatible.
  ///
  UINT64                                      Revision;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
};
```

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME)(
  IN EFI_SIMPLE_FILE_SYSTEM_PROTOCOL    *This,
  OUT EFI_FILE_PROTOCOL                 **Root
  );
```

13.4 Simple File System Protocol

The Simple File System protocol allows code running in the EFI boot services environment to obtain file based access to a device. EFI_SIMPLE_FILE_SYSTEM_PROTOCOL is used to open a device volume and return an EFI_FILE_PROTOCOL that provides interfaces to access files on a device volume.

EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

Summary

Provides a minimal interface for file-type access to a device.

GUID

```
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID \
 {0x0964e5b22,0x6459,0x11d2,\
  {0x8e,0x39,0x00,0xa0,0xc9,0x69,0x72,0x3b}}
```

Revision Number

```
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_REVISION  0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
 UINT64                                      Revision;
 EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdePkg/Include/Protocol/SimpleFileSystem.h#L530

UEFI module reverse engineering 101

- Other API and data are accessed through the Protocol API
  - eg：Accessing a file（code seen in SmmAccessSub) 2/2

```
            StrCat(Buffer,u_\ProgramData\Microsoft\Windows\S_800036f0);
            StrCat(filePath,u_IntelUpdate.exe_80000290);
            efiStatus = (**(gFileProcotol + 8))
                                  (gFileProcotol,&fileProtocol,filePath,0x8000000000000003,0);
            if (-1 < efiStatus) {
                efiStatus = (**(fileProtocol + 0x28))(fileProtocol,local_res18,&DAT_800002b0);
                if (-1 < efiStatus) {
                    efiStatus = (**(fileProtocol + 0x10))(fileProtocol);
```

```
struct _EFI_FILE_PROTOCOL {
  ///
  /// The version of the EFI_FILE_PROTOCOL interface. The version specified
  /// by this specification is EFI_FILE_PROTOCOL_LATEST_REVISION.
  /// Future versions are required to be backward compatible to version 1.0.
  ///
  UINT64                Revision;
  EFI_FILE_OPEN         Open;
  EFI_FILE_CLOSE        Close;      // +0x10
  EFI_FILE_DELETE       Delete;
  EFI_FILE_READ         Read;       // +0x20
  EFI_FILE_WRITE        Write;
```

UEFI module reverse engineering 101

- Callback API allows the module to be called on certain events
- UEFI core signals multiple pre-defined events
  - eg：Running code right before a boot loader starts（SmmInterfaceBase）

```
EFI_STATUS
ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)

{
    EFI_EVENT local_18 [3];

    FUN_800002b0(ImageHandle,SystemTable);
    DAT_80000668 = 0;
    (*gBS_12->CreateEventEx)
              (0x200,0x10,HandleReadyToBootEvent,NULL,
               &EFI_EVENT_READY_TO_BOOT_GUID,local_18);
    return 0;
}
```

EFI_EVENT_GROUP_READY_TO_BOOT

This event group is notified by the system right before notifying **EFI_EVENT_GROUP_AFTER_READY_TO_BOOT** event group when the Boot Manager is about to load and execute a boot option. The event group presents the last chance to modify device or system configuration prior to passing control to a boot option.

Discussion: MosaicRegressor detection by anti-virus

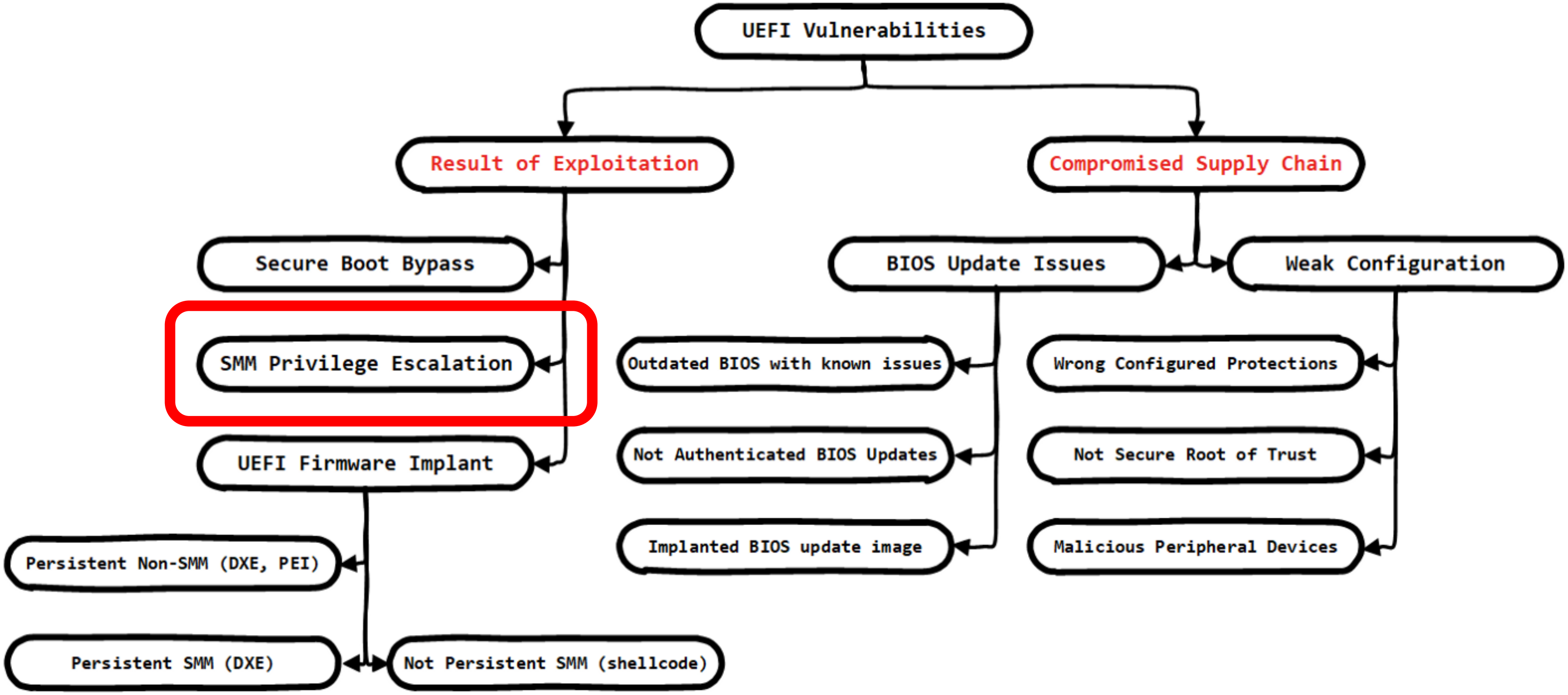
- Not possible to detect/prevent IntelUpdate.exe being written to disk
- Possible to detect/prevent execution of IntelUpdate.exe
  - 👉Practically disabling the threat
- Possible to detect the UEFI module but not possible to remove
  - Anti-virus software may read the SPI flash and detect malicious UEFI modules
    - Microsoft Defender APT UEFI Scanner, CrowdStrike Falcon, ESET, Kaspersky, etc
- Note: MosaicRegressor will most likely be erased by BIOS update

Discussion: Verified Boot – Mitigation against BIOS modification

- Secure boot
  - Checks a hash or digital signature of any module that ran after the DXE phase（ie, OEM Boot Block, OBB）
  - Prevents execution of modules that not allowed per configration
  - 👉MosaicRegressor will be disabled
- Intel Boot Guard
  - Checks integrity of code that ran before secure boot starts（ie, Initial Boot Block, IBB）with hardware
  - May prevent the boot process when tampering is detected
  - 👉MosaicRegressor will NOT be detected as it is a DXE driver
- Ref

SMM and Security

Category



https://medium.com/firmware-threat-hunting/uefi-vulnerabilities-classification-4897596e60af

What System Management Mode (SMM) is

- ~~Sack of crap 💩~~
- One of processor execution “modes”
  - Orthogonal to the privilege levels
    - Both SMM with ring0 and ring 3 privileges are possible
    - In reality, runs at ring 0, except on Secured-Core PCs (ref)
  - Several operations are allowed only in SMM, eg,
    - Bypass of some SPI flash write protection mechanism
    - Bypass of management/restriction by a hypervisor (hence, called ring -2)

30.1 SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

- Executed in the special physical memory region called SMRAM

What is SMRAM

- The regions where all access outside SMM is blocked by the memory controller
  - eg, any debuggers, hypervisor, DMA access
  - Specified by two registers in the Host Bridge and DRAM Controller (ref)
    - TSEG Memory Base (TSEGMB)
    - Base of GTT stolen Memory (BGSM)

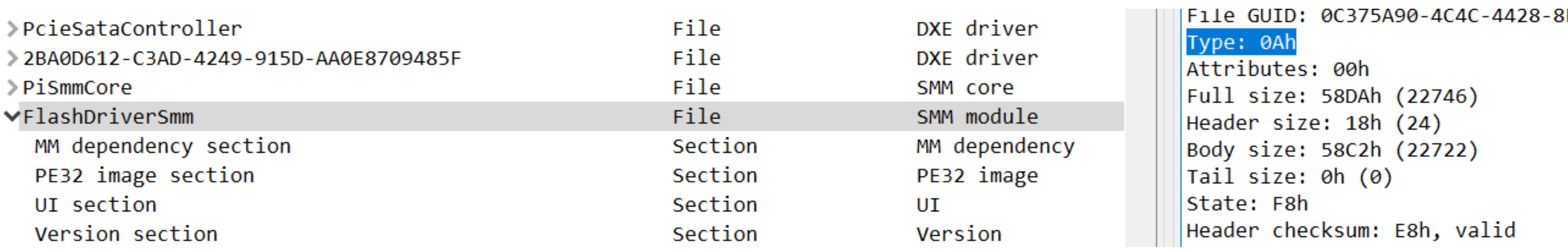
    When the extended SMRAM space is enabled, processor accesses to the TSEG range without SMM attribute or without WB attribute are handled by the processor as invalid accesses.

    2.5.3 TSEG

- SMM modules are loaded into and executed from this region
  - Data inside SMRAM is trusted
  - Data outside SMRAM is untrusted
    - Ie, same as the kernel touching user-mode memory

SMM Modules

- The same format as the DXE Boot Driver
  - UEFI “File” contains metadata indicating if SMM or not



- One BIOS can contain 200+ SMM modules
- DXE core automatically executes them from the firmware volume
  - Not possible to execute them from external storage such as a USB thumb drive

System Management Interrupt (SMI)

- SMM modules register SMI handlers
  - Think of this as registering command/event/message handlers
  - PiSmmCore implements the entry point of SMI handler
  - Registered handlers get called from there (like plug-in modules)
- SMI may be triggered by
  - Hardware (chipset) automatically
    - eg: thermo management
  - Software on execution of the “OUT” instruction against a certain port
    - eg: access to UEFI variables

Registration of SMI handlers

- Two APIs
  - EFI_SMM_SW_DISPATCH2_PROTOCOL.Register()
    - Registers SMI for a specified command number (ref)
  - EFI_MM_SYSTEM_TABLE.MmiHandlerRegister()
    - Registers SMI for a GUID (ref)
- Both SMI handlers have the same prototype

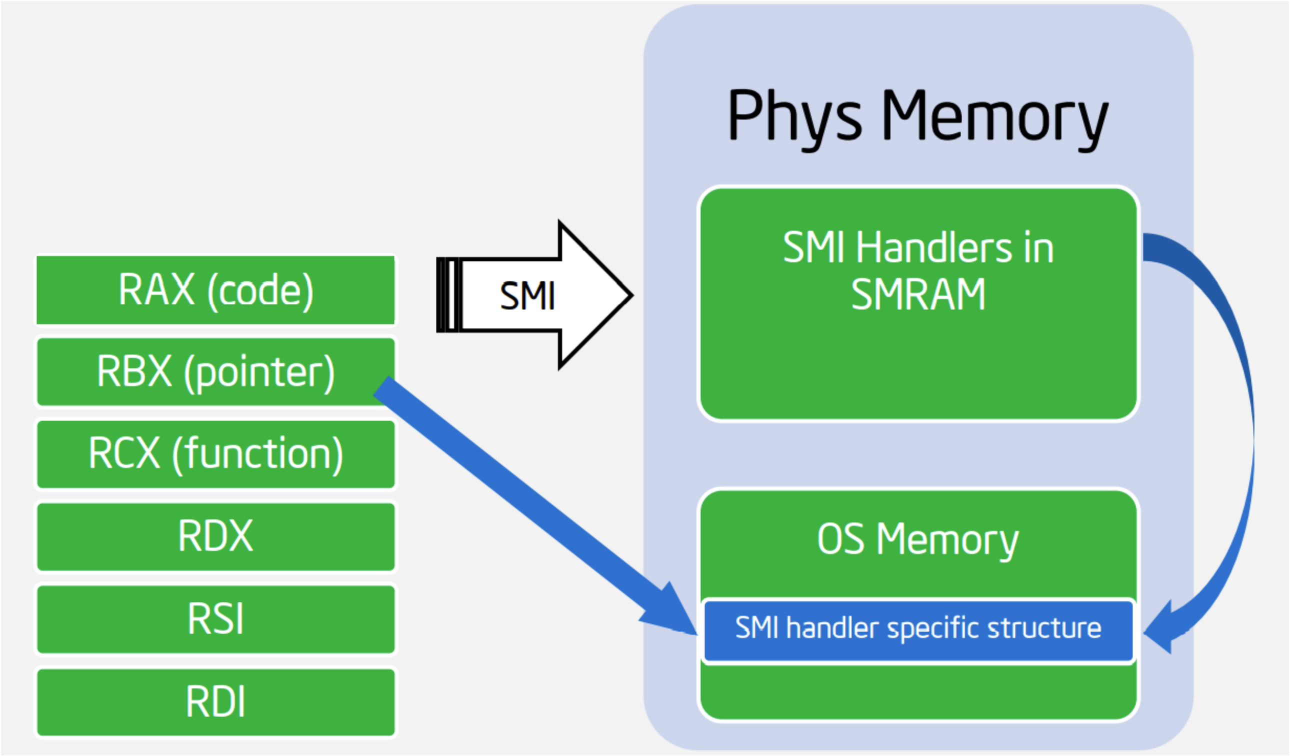
```
typedef
EFI_STATUS
(EFIAPI *EFI_MM_HANDLER_ENTRY_POINT)(
  IN EFI_HANDLE  DispatchHandle,
  IN CONST VOID  *Context        OPTIONAL,
  IN OUT VOID    *CommBuffer     OPTIONAL,
  IN OUT UINTN   *CommBufferSize OPTIONAL
  );
```

Handling of external input to SMI

- SMI handlers may need to receive input from outside SMRAM
- Values outside SMRAM must be validated
- Two common ways to receive such input (ref)
  - SMM Communication Buffer
  - General purpose registers
- Additionally, from an implementation specific physical address (exercise 3)

Vulnerability due to the lack of input validation

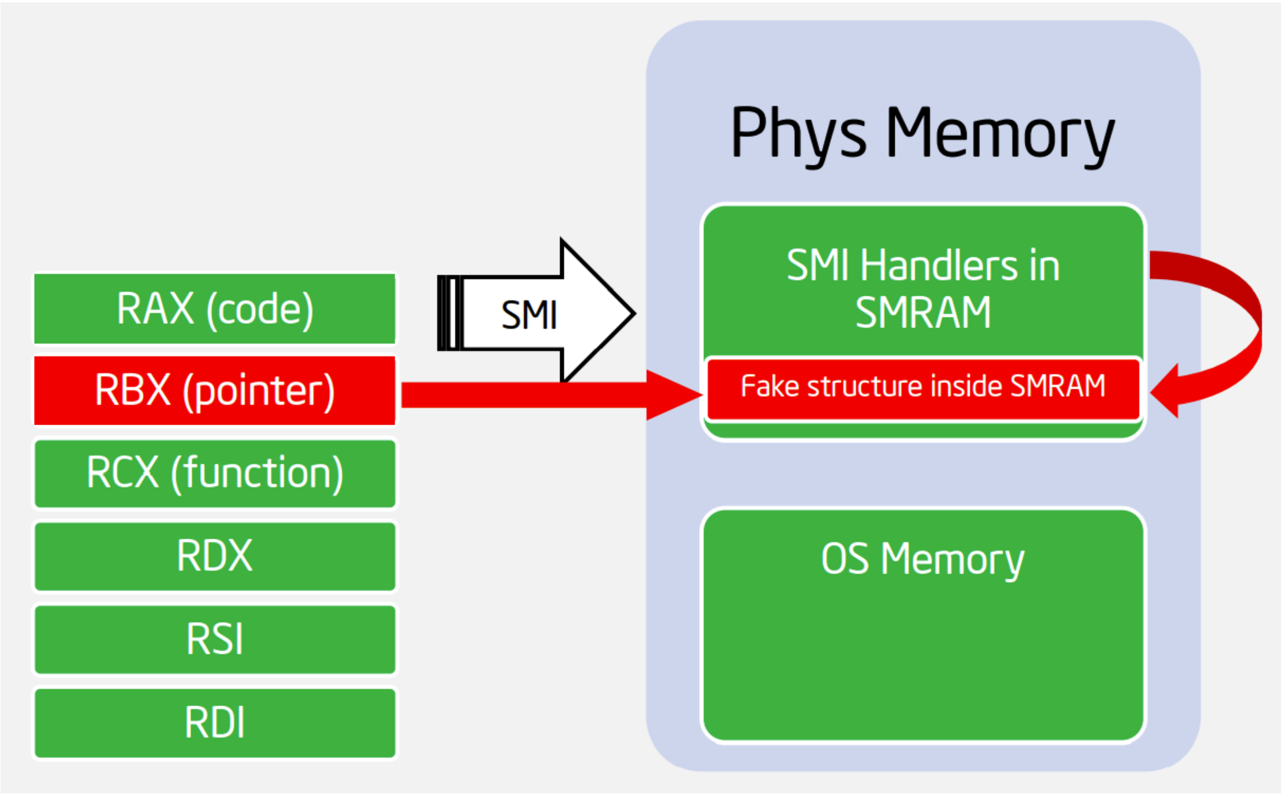
- When SMI writes a value to the location specified by the RBX register



http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf

Vulnerability due to the lack of input validation

- SMRAM may be corrupted by the kernel if RBX points inside SMRAM



http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf

Required validation

- Must validate that externally specified addresses point to outside SMRAM
  - EDK2: SmmIsBufferOutsideSmmValid() (ref)
  - AMI: AmiValidateMemoryBuffer() (ref)
- AND, do not use it if it points within SMRAM

Exercise 3 : CVE-2021-26943

- Vulnerable SMM modules: UsbRt, SdioSmm, NvmeSmm
- SMI registration
  - ModuleEntryPoint() -> FUN_80000ec8()

```
smiNumber = 0x42;
efiStatus = (*EFI_SMM_SW_DISPATCH2_PROTOCOL6->Register)
                      (EFI_SMM_SW_DISPATCH2_PROTOCOL6,swSmiHandler7,&smiNumber,&local_res20)
```

- SMI handler : swSmiHandler7()

Exercise 3 ：CVE-2021-26943

- Taking external input
  - Refers to physical memory address outside SMRAM
  - usedAsAddress = ((*(uint16_t)0x40e) * 0x10 + 0x104)
- Vulnerability:
  - Using the value even if SMRAM range validation fails

```
    if (addressFromOutsideSmram == NULL) {
        addressFromOutsideSmram = *(uRam000000000000040e * 0x10 + 0x104);
    }
    else {
        *CommBuffer = NULL;
    }
    efiStatus = IsOutsideSmram(addressFromOutsideSmram);
    efiStatus2 = 0;
    if (((efiStatus < 0) || (efiStatus2 = efiStatus, *addressFromOutsideSmram != '\'')) ||
       (0xb < addressFromOutsideSmram[1])) {
        efiStatus = efiStatus2;
        addressFromOutsideSmram[2] = '\a';
    }
    else {
```

Exploitation

- Steps
  1. Set 0 to the physical address 0x40e
  2. Write X to the physical address 0x104, where X is inside SMRAM
  3. Execute SMI 0x42
  4. '\a' (0x07) is written to the physical address X + 2

```
    if (addressFromOutsideSmram == NULL) {
        addressFromOutsideSmram = *(uRam000000000000040e * 0x10 + 0x104);
    }
    else {
        *CommBuffer = NULL;
    }
    efiStatus = IsOutsideSmram(addressFromOutsideSmram);
    efiStatus2 = 0;
    if (((efiStatus < 0) || (efiStatus2 = efiStatus, *addressFromOutsideSmram != '\'')) ||
       (0xb < addressFromOutsideSmram[1])) {
        efiStatus = efiStatus2;
        addressFromOutsideSmram[2] = '\a';
    }
    else {
```

Impacts

- Local kernel-to-SMM privilege escalation
  - Prerequisites：Must be able to run kernel-mode code
  - Example impacts: Breaking hypervisor (ref), overwriting the SPI flash
- Exact same modules are used by other vendors
  - Thousands of devices still contains the vulnerable modules
  - Challenging for OEMs to know and fix ALL impacted BIOSes for a given vulnerability
  - Challenging for IT admins to ensure ALL endpoints get updated for a given fix

Discussion: Detection of those vulnerable modules

- SMM developers
  - More comprehensive BIOS inventory management
  - More vulnerability information sharing across OEMs
    - It was the same issue as INTEL-SA-00057 (ref) but ASUS failed to apply the fix to their BIOS
- Security researchers
  - Detecting the same code pattern（efiXplorer, Brick）
  - Scanning module hashes from BIOS update files
- IT administrators
  - Inventory for the BIOS versions in the org
  - Inventory for UEFI modules (hash, name, GUID) in the org
  - Patch management

Discussion: Detection of modules with similar 0-day vulnerabilities

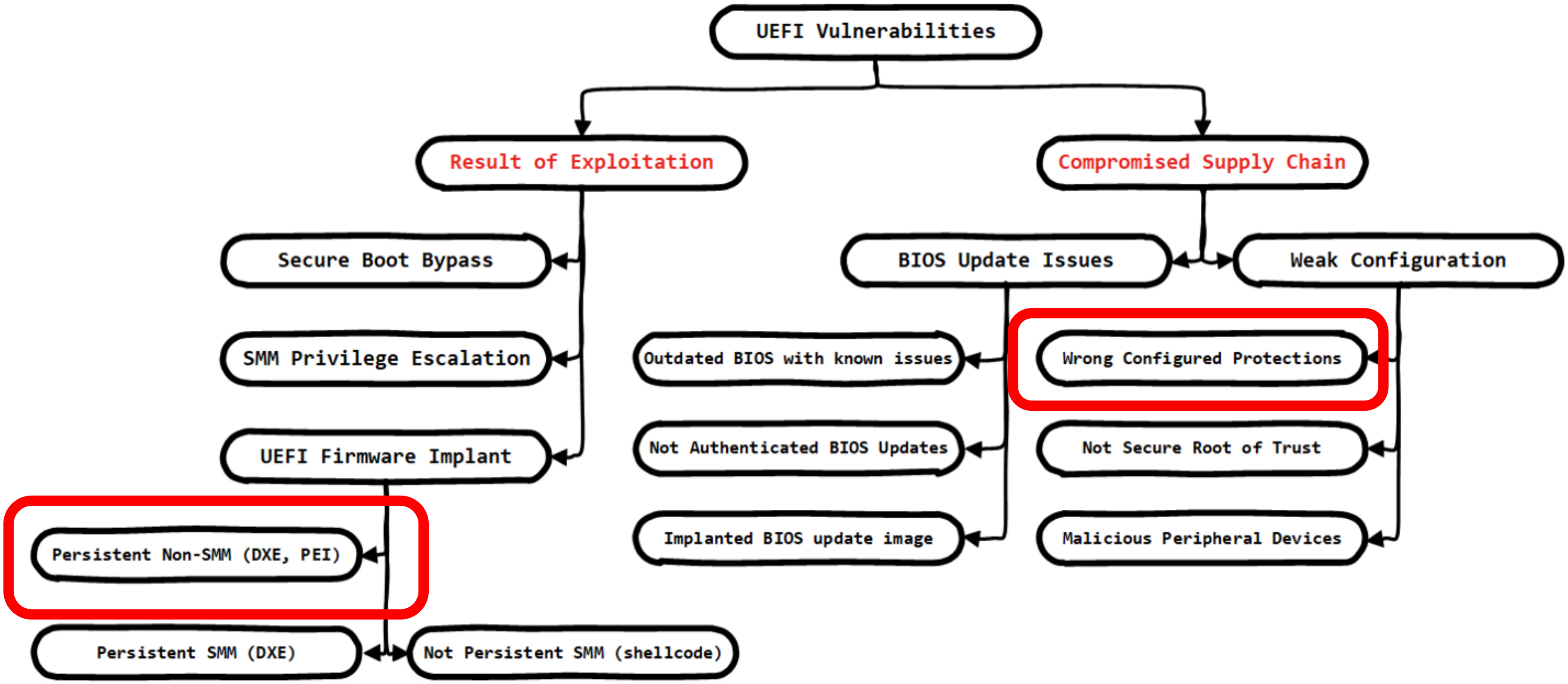
- SMM developers
  - Security design review
  - Using the processor security features（SMM_Code_Chk_En, SMM page protection, etc）
  - Using newer EDK2
  - Using a safer programming language (ref)
- Security Researchers
  - Scraping BIOS Update + Vulnerable code pattern identification
  - UEFI fuzzing
- IT admins
  - Kernel-module management
  - BIOS hash abnormality check (post-exploitation detection)?

MSR_SMM_FEATURE_CONTROL	Package	Enhanced SMM Feature Control (SMM-RW) Reports SMM capability Enhancement. Accessible only while in SMM.
0		Lock (SMM-RWO) When set to '1' locks this register from further changes.
1		Reserved
2		SMM_Code_Chk_En (SMM-RW) This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.

Malware that infects UEFI using vulnerabilities

(INTEL SPECIFIC)

Category



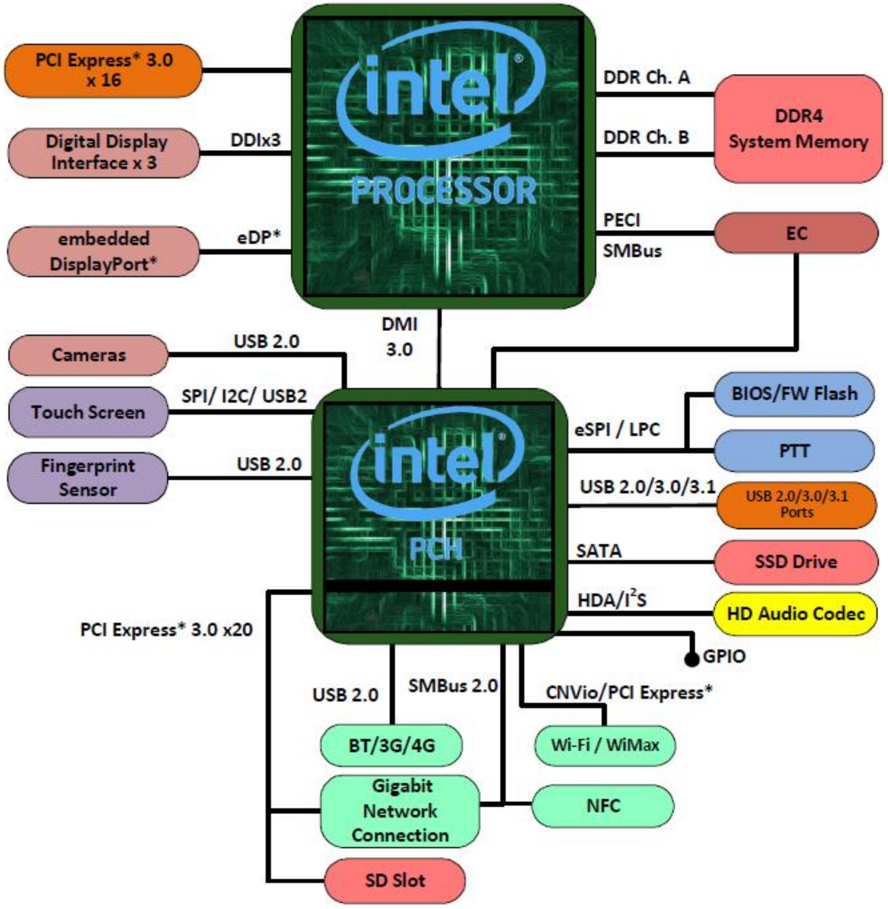
https://medium.com/firmware-threat-hunting/uefi-vulnerabilities-classification-4897596e60af

Protection of the SPI flash

- BIOS, ie, the SPI flash, must be updatable
- BIOS must be write-protected from the OS
  - 👉 kernel to SMM privilege escalation if BIOS were writable from OS
- Hardware implements registers that control write-protection
  - 👉 BIOS configures the registers
  - 👉 BIOS is software written by humans
  - 👉 Misconfiguration is possible
    - And in fact, it was very (VERY) common

Intel chipset architecture

- Upper : Processor die (latest is 12th gen)
- Lower : Platform Controller Hub (PCH), a.k.a. chipset (latest is 600 series)



Accessing the SPI flash from software

- Hardware sequencing
- By configuring certain registers, up to 64bytes may be read or written at a time
  - Steps
    1. In the Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL) register,
       1. Confirm that the Flash Descriptor Valid (FDV) bit == 1
       2. Write 0 to the Flash Cycle (FCYCLE) bit for read（1 for write）
       3. Write the size of I/O to the Flash Data Byte Count (FDBC) bit
    2. Write the I/O offset to the Flash Address (BIOS_FADDR) register
    3. In the Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL) register
       1. Write 1 to the Flash Cycle Go (FGO) bit
       2. (Hardware starts I/O)
       3. Wait until the SPI Cycle In Progress (H_SCIP) bit == 0
       4. Read Flash Data 0 .. 15 (BIOS_FDATA0 .. 15) registers containing data read from the SPI flash
    4. Repeat (2) and (3)

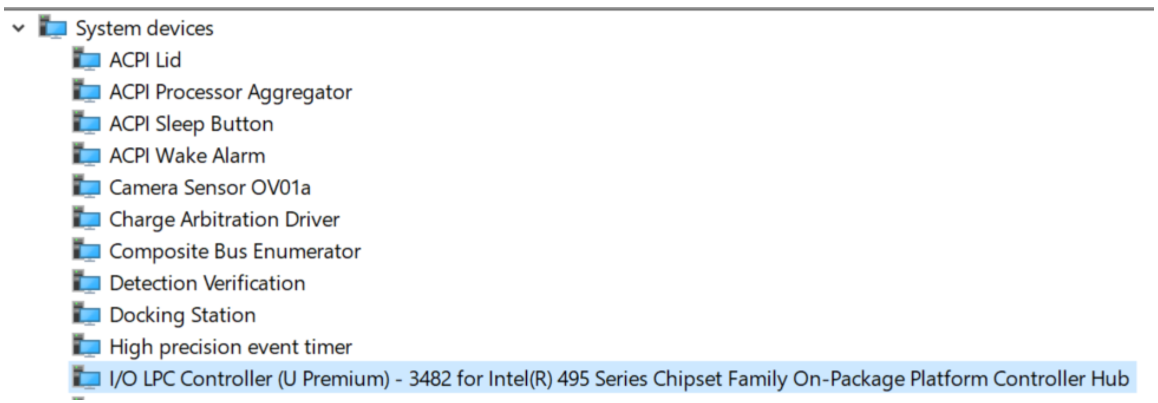
7.2.2 Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL)—Offset 4h

7.2.3 Flash Address (BIOS_FADDR)—Offset 8h

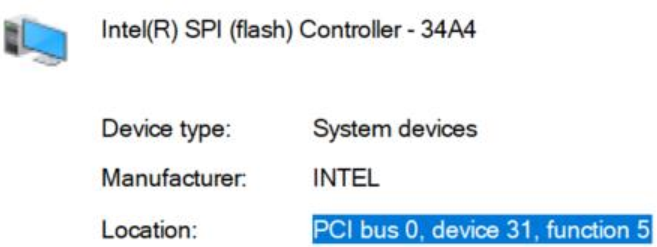
7.2.5 Flash Data 0 (BIOS_FDATA0)—Offset 10h

Locating the registers

- Identify the PCH version from the device information（eg: 495 On-Package）



- Get the corresponding specification (ref)
- Confirm B0:D31:F5 is the SPI controller according with the spec



Locating the registers

- As BIOS_HSFSTS_CTL is at the offset from SPI_BAR0, locate SPI_BAR0 firest

7.2 SPI Memory Mapped Registers Summary

The SPI memory mapped registers are accessed based upon offsets from SPI_BAR0 (in PCI config SPI_BAR0 register).

Table 7-2. Summary of SPI Memory Mapped Registers

Offset Start	Offset End	Register Name (ID)—Offset
0h	3h	BIOS Flash Primary Region (BIOS_BFPREG)—Offset 0h
4h	7h	Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL)—Offset 4h
8h	Bh	Flash Address (BIOS_FADDR)—Offset 8h
Ch	Fh	Discrete Lock Bits (BIOS_DLOCK)—Offset Ch
10h	13h	Flash Data 0 (BIOS_FDATA0)—Offset 10h

- SPI_BAR0 is at the offset 0x10 in the PCI Config Space

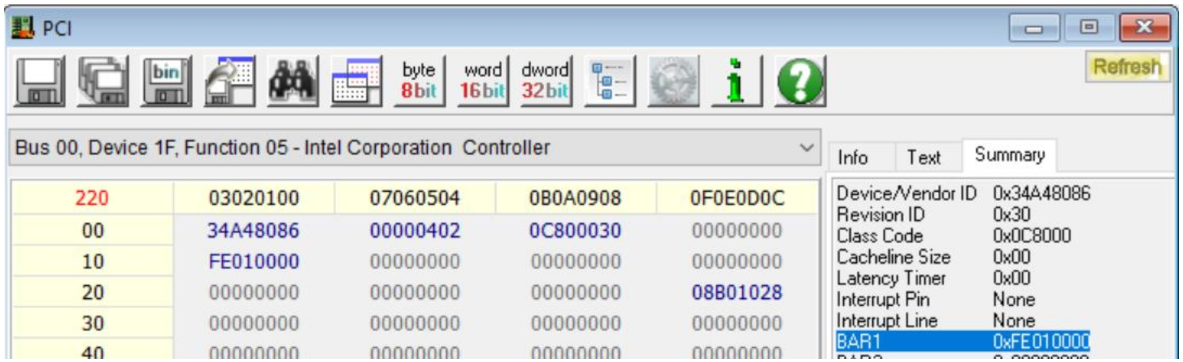
7.1 SPI Configuration Registers Summary

Table 7-1. Summary of SPI Configuration Registers

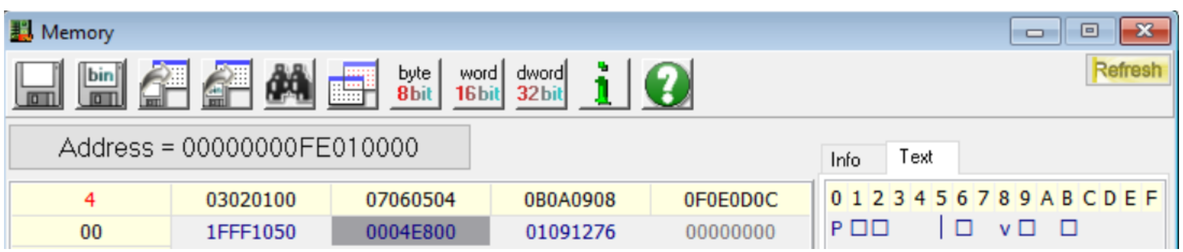
Offset Start	Offset End	Register Name (ID)—Offset
0h	3h	Device ID and Vendor ID (BIOS_SPI_DID_VID)—Offset 0h
4h	7h	Status and Command (BIOS_SPI_STS_CMD)—Offset 4h
8h	Bh	Revision ID and Class Code (BIOS_SPI_CC_RID)—Offset 8h
Ch	Fh	BIST, Header Type, Latency Timer, Cache Line Size (BIOS_SPI_BIST_HTYPE_LT_CLS)—Offset Ch
10h	13h	SPI BAR0 MMIO (BIOS_SPI_BAR0)—Offset 10h

Locating the registers

- PCI Config Space can be checked with tools like [RWEverything]



- The register is at 0xfe01000(SPI_BAR0) + offset as per the spec



Write-protection features

- The BIOS Control register implements some write-protection mechanisms

Old Name	Recent Name	Description
BIOS Write Enable (BIOSWE)	Write Protect Disable (WPD)	1: Allows write to BIOS Causes #SMI when updated from 0 to1, and the LE bit == 1
BIOS Lock Enable (BLE)	Lock Enable (LE)	1: Write-protects EISS and causes #SMI when the WPD bit is updated from 0 -> 1
SMM BIOS Write Protect Disable (SMM_BWP)	Enable InSMM.STS (EISS)	1: Write-protects BIOS unless all processors are in SMM

7.1.8 BIOS Control (BIOS_SPI_BC)—Offset DCh

Vulnerability checks by Lojax

- Lojax checks the registers with RWEverything (which is a signed driver)
  - and attempts to infect to BISO by modifying the SPI flash if a vulnerability is detected (ref1, ref2)

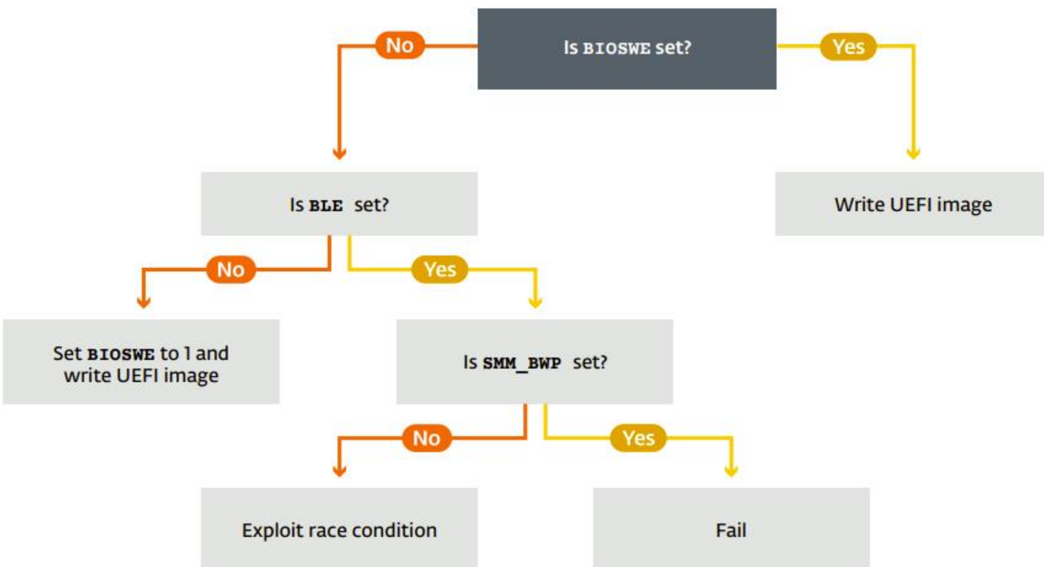


Figure 12 // Decision tree of the writing process

https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf

Old Name	Description
BIOSWE	1: Allows write to BIOS Causes #SMI when updated from 0 to1, and the LE bit == 1
BLE	1: Write-protects EISS and causes #SMI when the WPD bit is updated from 0 -> 1
SMM_BWP	1: Write-protects BIOS unless all processors are in SMM

Speed Racer

- The vulnerability when SMM_BWP/EISS == 0 or do not exist (ref)
- Attack flow
  1. Assume BIOSWE/WPD == 0, BLE/LE == 1
  2. CPU1 writes 1 to BIOSWE/WPD
  3. CPU1 receives SMI
  4. CPU2 performs hardware sequencing and modifies BIOS
  5. CPU1 resets 0 to BIOSWE/WPD in SMM (but it is too late)
- (4) would fail if SMM_BWP/EISS == 1

Discussion: Identification of vulnerable devices

- Any devices older than PCH5 (2008）
  - No SMM_BWP/EISS
- Run software that performs the vulnerability check, eg,
  - CHIPSEC
  - CrowdStrike Falcon

Discussion: Detection of new malware that exploits the same vulnerability

- At installation and runtime
  - Detecting installation of suspicious/bad kernel-mode drivers
  - Detecting use of suspicious/bad IOCTL
  - Detecting access to the registers with a hypervisor?
- Post infection
  - Enabling Secure Boot
  - Detecting the malicious UEFI modules in BIOS
    - GUID: 832d9b4d-d8d5-425f-bd52-5c5afb2c85dc
    - SHA256: 7ea33696c91761e95697549e0b0f84db2cf4033216cd16c3264b10daa31f598c
  - Detecting the change of BIOS image hash?
  - Detecting the malicious Windows components such as modified autochk.exe

Wrap up

What may the next step be?

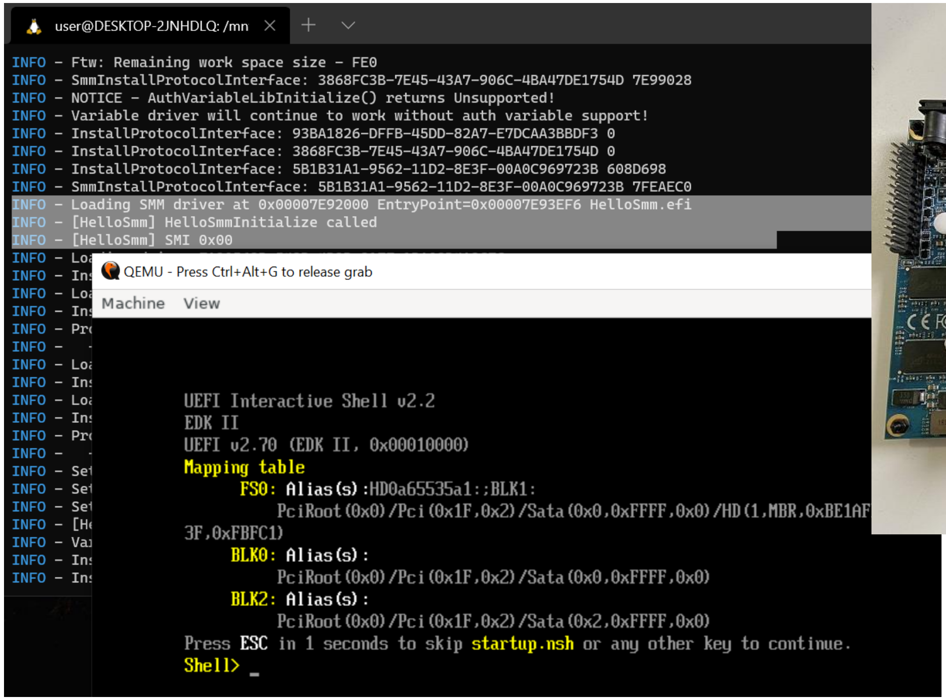
- Output ideas
  - Reimplement the UEFI malware
  - Discovering firmware 0-day
  - Researching scalable vulnerability discovery
    - Emulation, fuzzing etc
- Input ideas
  - Reading books
    - Excellent for understanding both high- and low-levels of the domain
  - Follow experts on Twitter
  - Watch conference talks and slides

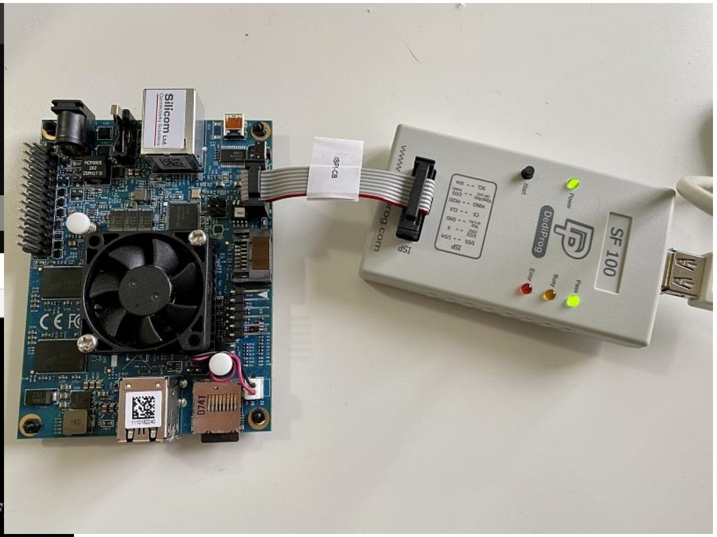
Enjoy!

Questions?

Note : D.I.Y. SMM development

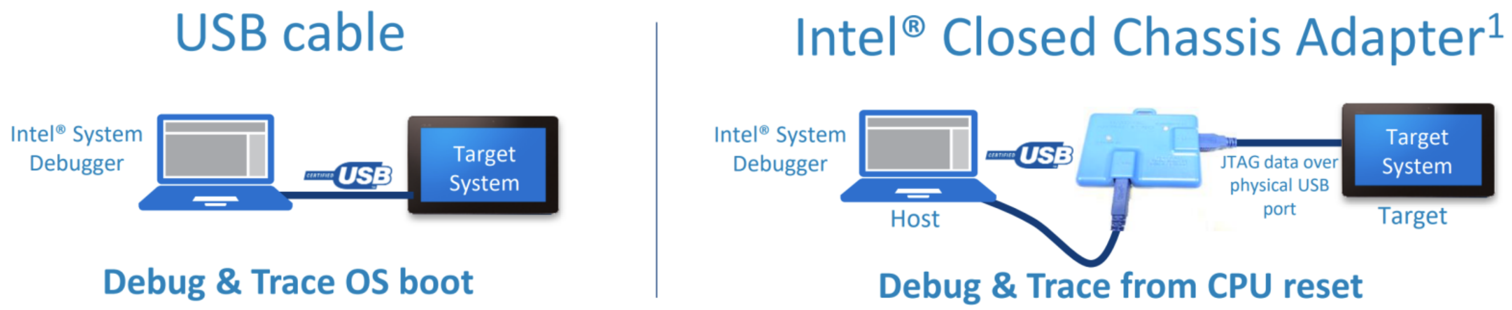
- Like DXE drivers, SMM module can be developed too ([ref]())
  - QEMU can emulate SMM to some extent but not perfect
  - VMware does not have SMM at all
  - For physical devices, it must be flashed with the SPI flash programmer





Note : debugging SMM

- Requires a hardware debugger against a physical device
- Intel systems can be debugged with a USB cable through Direct Connect Interface（DCI）(rer1, ref2)



- DCI could also be used for stealthier kernel-debugging
  - eg: for debugging Windows Kernel Patch Protection

Note：SMRAM forensic

- SMRAM may be obtained when a hardware debugger is attached
- SMRAM contains many data structures starting with 4 bytes magic values
  - ‘SMST’, ‘smmc’, ‘smih’ etc
  - Their layouts can be found in EDK2
- SMRAM can be parsed to discover SMI handlers
  - smram_parse.py
    - Authored by Dmytro Oleksiuk @d_olex (ref)
    - Updated by myself for Python3 (ref)

Note : SMM entry points and code flow

- [_SmiEntryPoint (SmiEntry.nasm)]
  - [SmiRendezvous (MpService.c)]
    - [BSPHandler (MpService.c)]
      - [SmmEntryPoint (PiSmmCore.c)]
        - [SmiManage (Smi.c)]
          - SmiHandler->Handler()

- SMM entry points starts on 16bit real-mode
  - Then, transitions to the long-mode quickly
  - SmiRendezvous makes sure all processors are in SMM by issuing SMIs to all other processors（prevents race condition）

Note : other recent SMM vulnerabilities

- SMM callout
  - https://github.com/binarly-io/Vulnerability-REsearch/blob/main/Lenovo/BRLY-2021-001.md
- SMM callout via EFI_BOOT_SERVICES
  - https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there.html
- Write-what-where through SMM
  - https://dannyodler.medium.com/attacking-the-golden-ring-on-amd-mini-pc-b7bfb217b437

Note: conference talks

- Link Collections
  - [Low Level PC/Server Attack & Defense Research Timeline]
  - [InfoSec Reference - Low Level Attacks/Firmware/BIOS/UEFI]
- Talks
  - [Automated vulnerability hunting in SMM using Brick]
  - [Summary of Attacks Against BIOS and Secure Boot]
  - [Safeguarding UEFI Ecosystem: Firmware Supply Chain is Hard(coded)]
  - [UEFI Firmware Rootkits: Myths and Reality]
  - [MODERN SECURE BOOT ATTACKS: BYPASSING HARDWARE ROOT OF TRUST FROM SOFTWARE]
  - [BETRAYING THE BIOS: WHERE THE GUARDIANS OF THE BIOS ARE FAILING]

Note: Books and free online course(s)

- Books
  - Rootkits and Bootkits
  - Building Secure Firmware
  - Intel Safer Computing Initiative Building Blocks for Trusted Computing (Free)
  - Platform Embedded Security Technology Revealed (Free)
- Course(s)
  - Architecture 4001: x86-64 Intel Firmware Attack & Defense (Free)