DETECT ME IF YOU CAN - ANTI-FIRMWARE FORENSICS

TAKAHIRO HARUYAMA

VMWARE THREAT ANALYSIS UNIT

WHO AM I?

- Takahiro Haruyama (@cci_forensics)
  - Senior Threat Researcher at VMware Carbon Black TAU
- Past Research
  - Anti-Forensics (e.g., exploiting EnCase's Outside In)
  - RE (e.g., defeating compiler-level obfuscations)
  - Malware Analysis (e.g., Internet-wide C2 scanning)

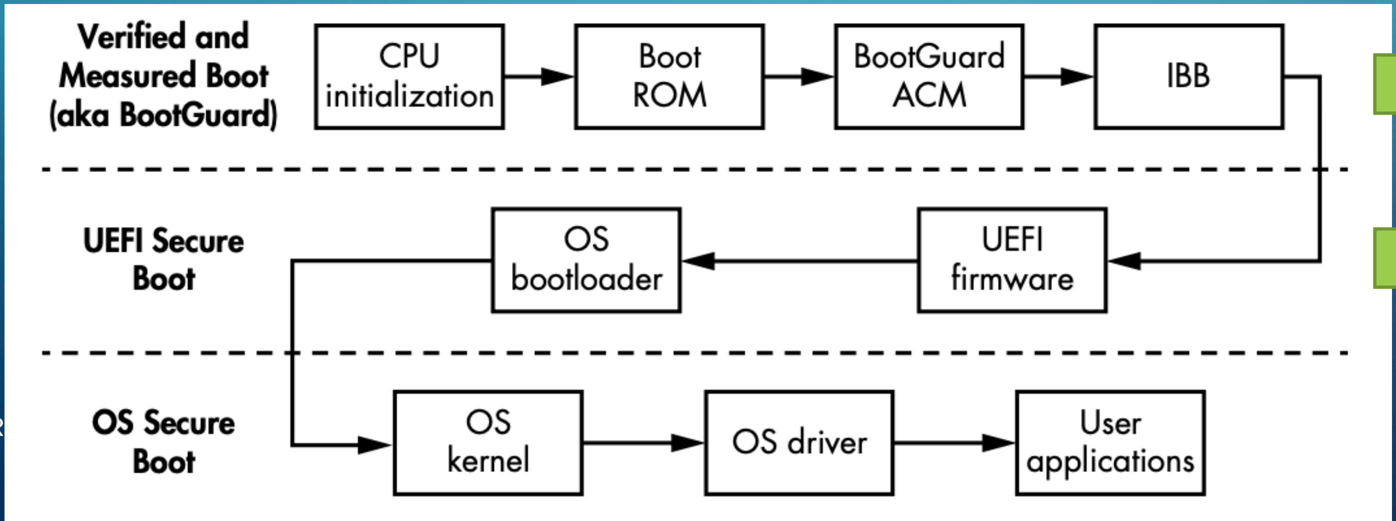
OVERVIEW

- Background and Motivation
- Test Environment Setup
- Implementation
- SpiMitm vs. Firmware Security Tools
- Countermeasures
- Wrap-up

BACKGROUND AND MOTIVATION

FIRMWARE THREATS

- Bootkits
  - longer persistence and lower observability than OS-level implants
- UEFI Secure Boot and Intel Boot Guard can detect bootkits
  - But they can be bypassed by vulnerability exploits
    - e.g., CVE-2021-0157 & CVE-2021-0158



Detect MoonBounce

Detect Vector-EDK, LoJax

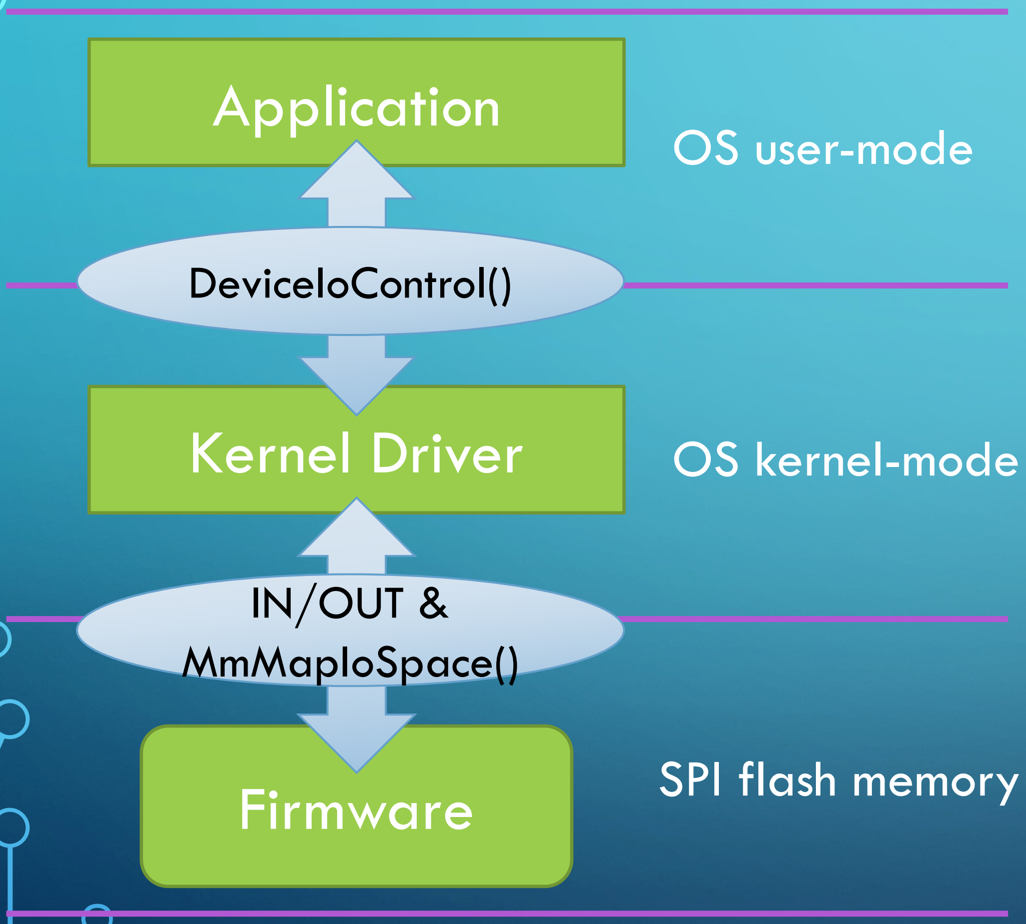
Source: Rootkits and Bootkits

FIRMWARE SCANNERS

- Several vendors provide an UEFI firmware scanner
  - AV/EDR: CrowdStrike, Microsoft, ESET, Kaspersky
  - firmware security: Eclypsium, Binarly
- The scanner behavior

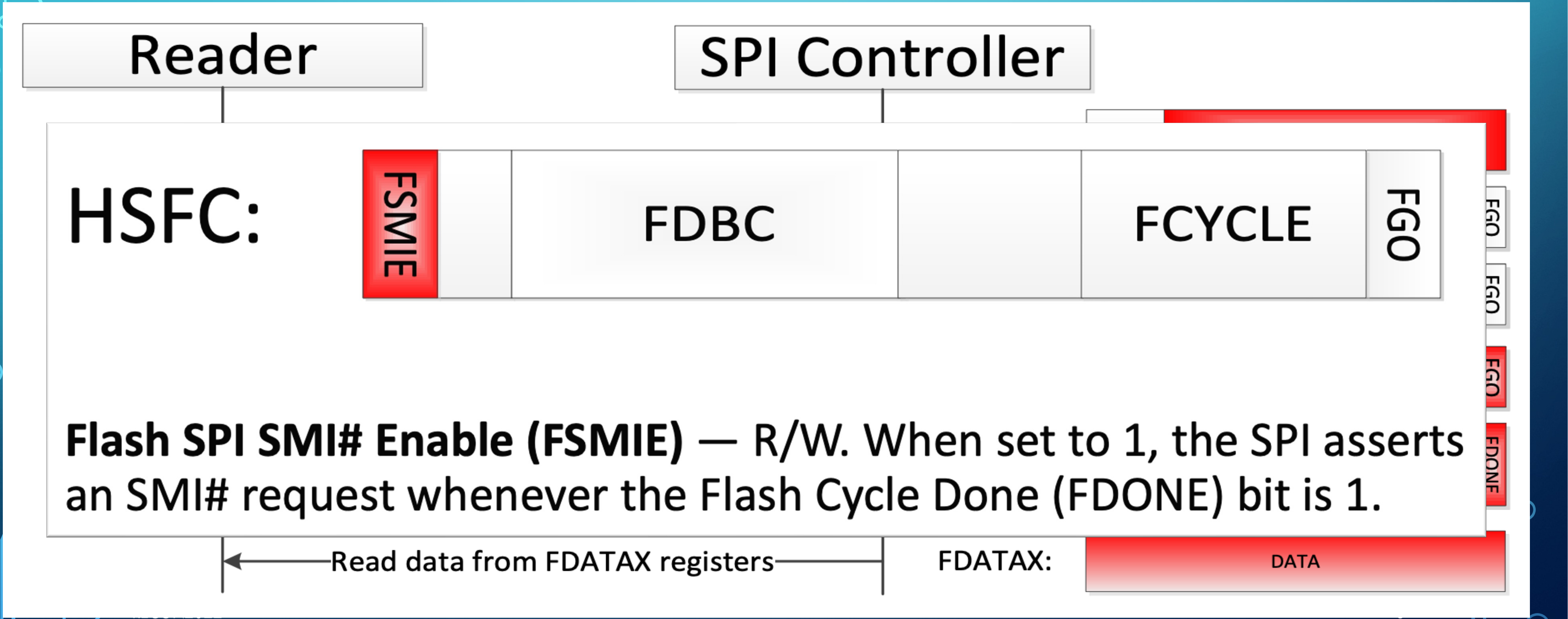
1. acquiring a firmware image inside a SPI flash memory
2. parsing and scanning the image with signatures

SOFTWARE-BASED APPROACH FOR FIRMWARE ACQUISITION



- I/O through the SPI flash interface
  - Port I/O (IN/OUT instructions)
  - Memory-Mapped I/O (MmMapIoSpace API)
- Steps for firmware acquisition
  1. Get SPI Base Address Register (SPIBAR)
  2. Read/write SPI registers

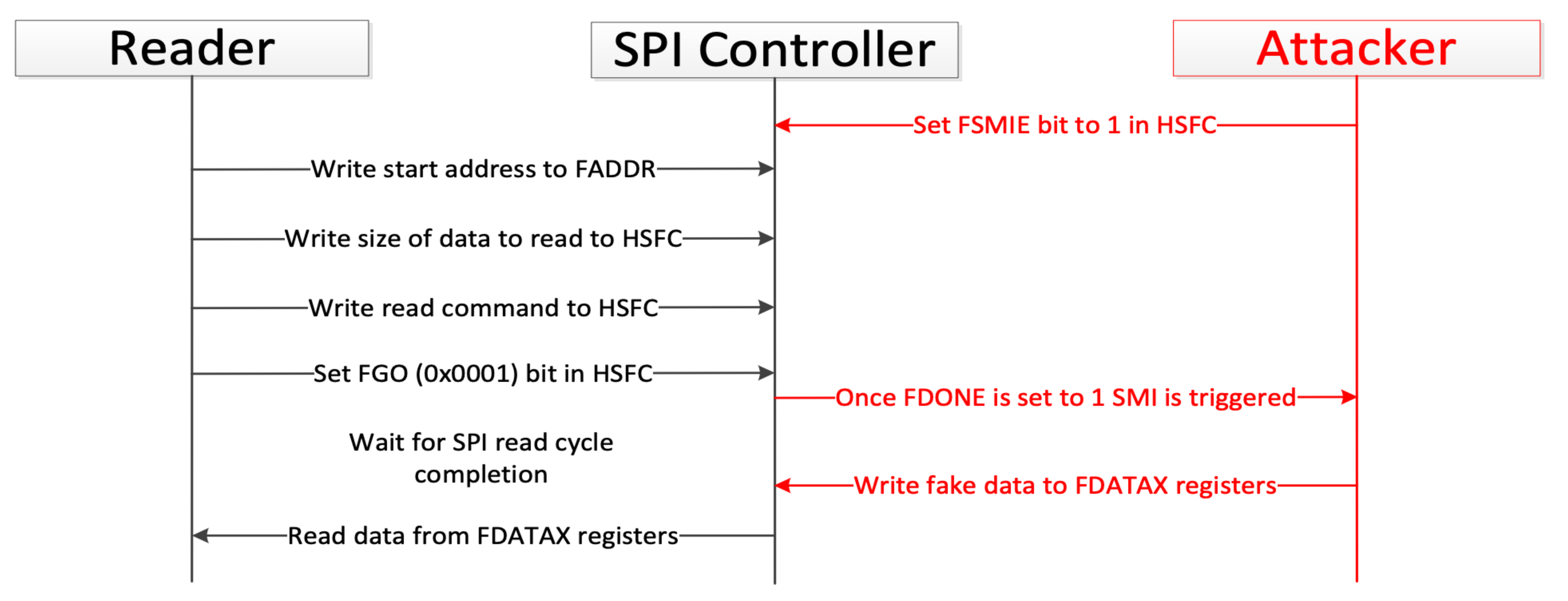
SPI REGISTER ACCESS FOR FIRMWARE ACQUISITION



Flash SPI SMI# Enable (FSMIE) — R/W. When set to 1, the SPI asserts an SMI# request whenever the Flash Cycle Done (FDONE) bit is 1.

Source: UEFI Firmware Rootkits: Myths and Reality

SPI FLASH READ MITM ATTACK



Reader

SPI Controller

Attacker

Set FSMIE bit to 1 in HSFC

Write start address to FADDR

Write size of data to read to HSFC

Write read command to HSFC

Set FGO (0x0001) bit in HSFC

Once FDONE is set to 1 SMI is triggered

Wait for SPI read cycle completion

Write fake data to FDATAX registers

Read data from FDATAX registers

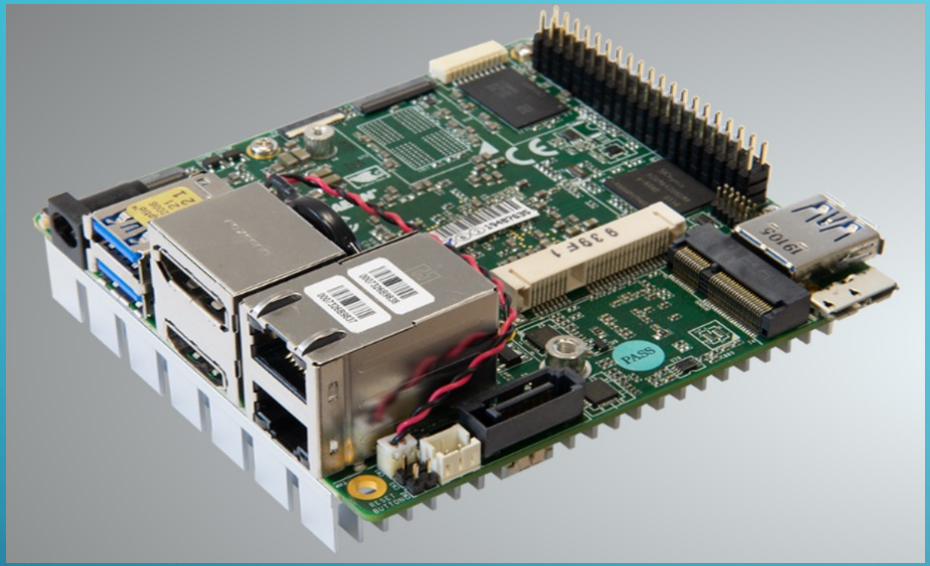
Source: UEFI Firmware Rootkits: Myths and Reality

MOTIVATION

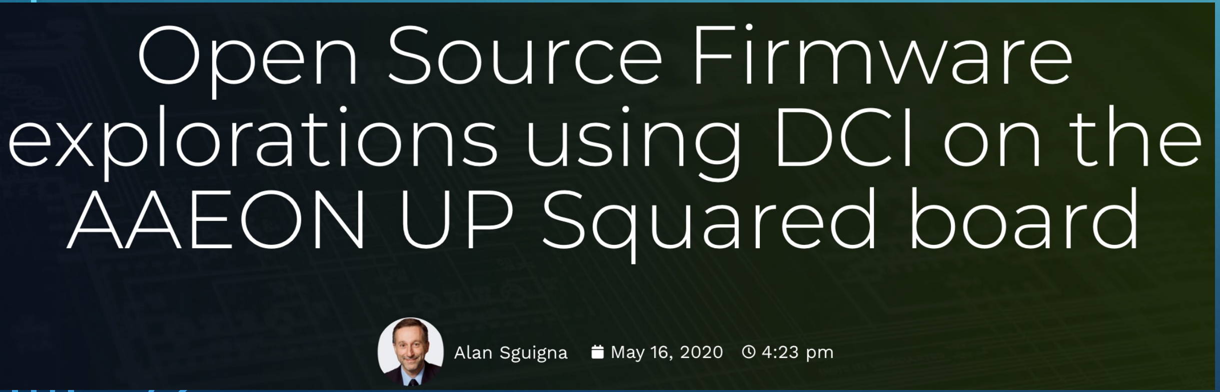
- This attack possibility was pointed out by researchers for years
  - Xeno Kovah et al. "Copernicus 2: SENTER the Dragon!" in 2014
- But there has been no publicly-available PoC
- Know our enemy!
  - Implement the attack PoC
  - Test firmware scanners against the PoC

TEST ENVIRONMENT SETUP

TESTED HARDWARE



- UP Squared
  - Intel Atom x7-E3950, Apollo Lake SoC
- Intel distributes the open source firmware debug image
- “The UP Squared Chronicles” by Alan Sguigna
  - How to flash the image
  - How to Build the image



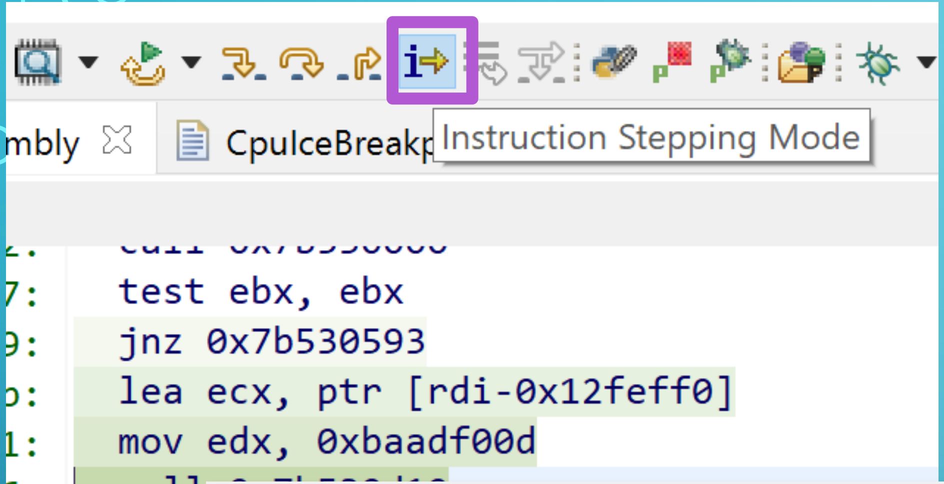
Open Source Firmware explorations using DCI on the AAEON UP Squared board

Alan Sguigna May 16, 2020 4:23 pm

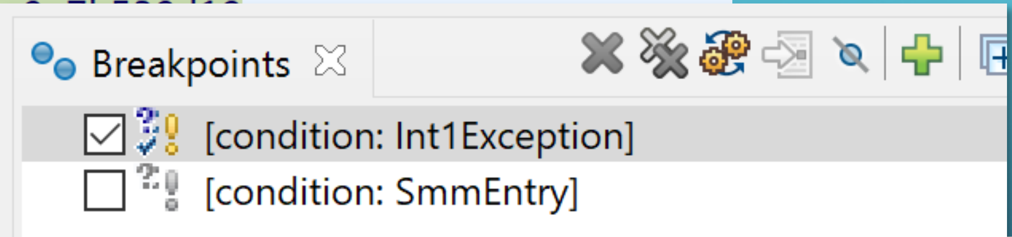
HARDWARE DEBUGGING

- Intel Direct Connect Interface (DCI)
  - DCI enables to JTAG-debug an Intel CPU over a USB port
- Intel System Studio (ISS) provides the debuggers
  - Intel System Debugger (embedded in ISS) recommended!
  - Intel System Debugger (legacy, stand-alone)
  - WinDbg extensions

SMM CODE DEBUGGING TIPS

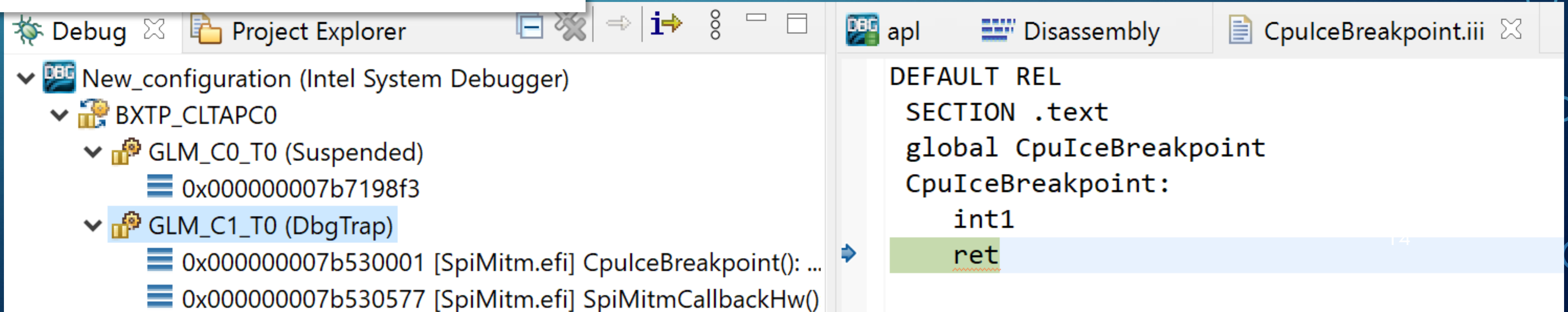


```
test ebx, ebx
jnz 0x7b530593
lea ecx, ptr [rdi-0x12feff0]
mov edx, 0xbaadf00d
call
add
```



- ☑ [condition: Int1Exception]
- □ [condition: SmmEntry]

- Instruction Stepping Mode
  - essential for step into/over
- How to break the SMM code
  - Break by SMMEntry then enable the hardware breakpoint manually
    - It's noisy if any periodic timer SMI
  - Insert CpuIceBreakpoint (INT1)

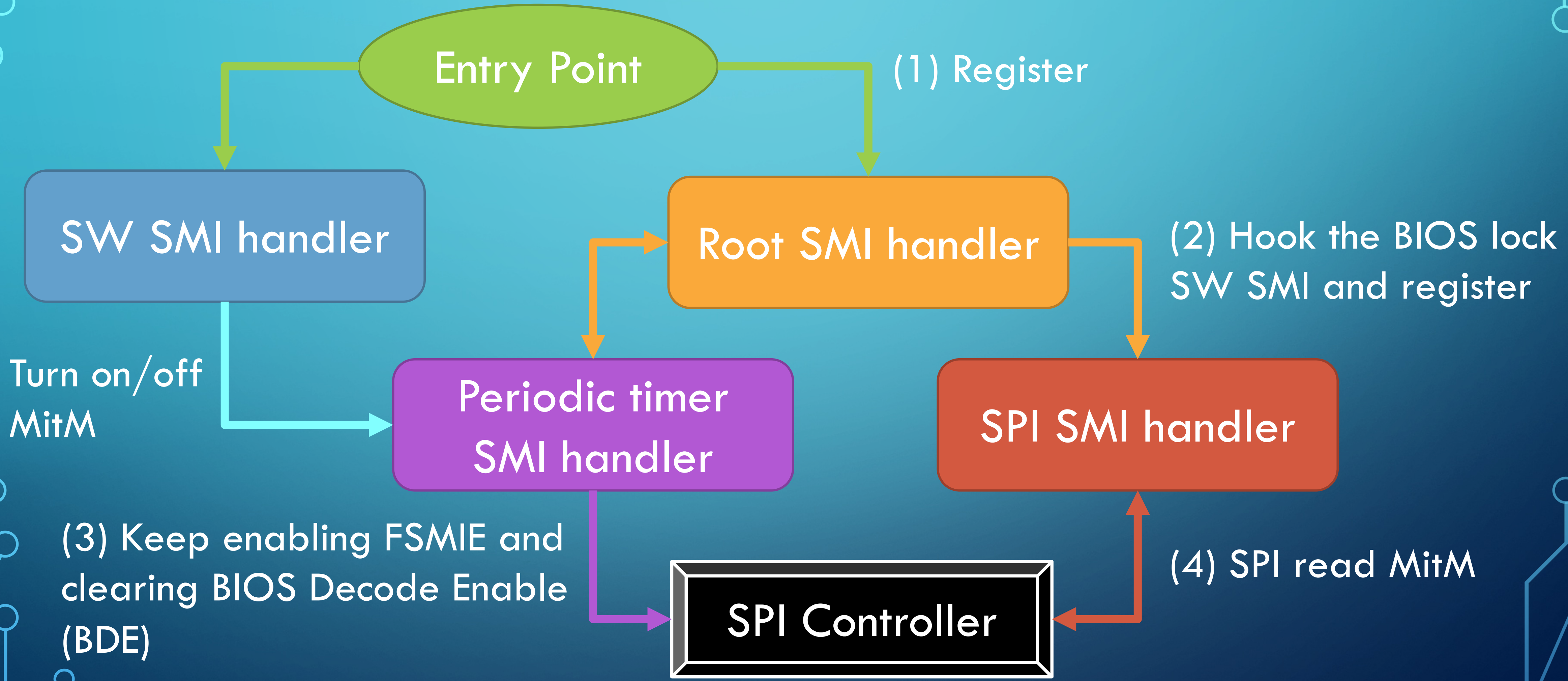


```
New_configuration (Intel System Debugger)
  BXTP_CLTAPC0
    GLM_C0_T0 (Suspended)
      0x000000007b7198f3
    GLM_C1_T0 (DbgTrap)
      0x000000007b530001 [SpiMitm.efi] CpuIceBreakpoint(): ...
      0x000000007b530577 [SpiMitm.efi] SpiMitmCallbackHw()
```

```
DEFAULT REL
 SECTION .text
 global CpuIceBreakpoint
 CpuIceBreakpoint:
    int1
    ret
```

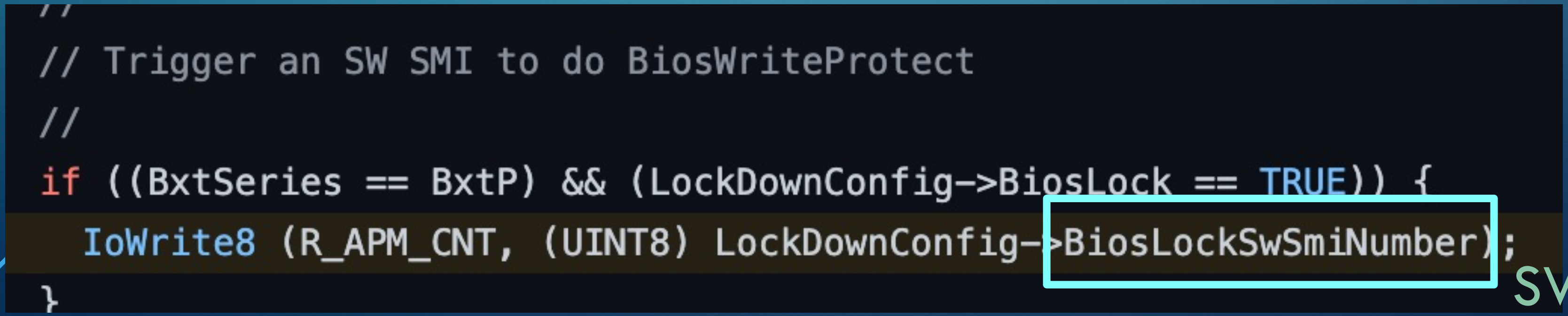
IMPLEMENTATION

SPIMITM SMM MODULE SUMMARY


Entry Point
(1) Register
SW SMI handler
Root SMI handler
(2) Hook the BIOS lock SW SMI and register
Turn on/off MitM
Periodic timer SMI handler
SPI SMI handler
(3) Keep enabling FSMIE and clearing BIOS Decode Enable (BDE)
SPI Controller
(4) SPI read MitM

REGISTERING PERIODIC TIMER / SPI SMI HANDLERS

- I wanted to register the SMI handlers in the late stage
- Hook the BIOS Lock Software SMI before the OS boot
  - Triggered in the SC initialization routine ScOnReadyToBoot
    - 0xA9 (SW_SMI_BIOS_LOCK)
    - The SW SMI handler registers the TCO BIOSWR SMI handler disabling the BCR.BIOSWE bit

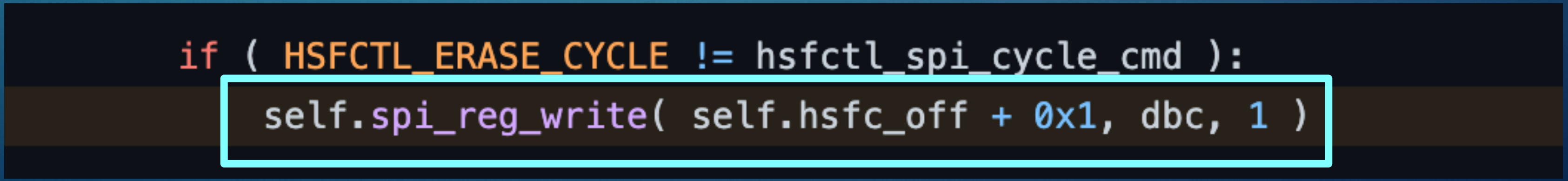


```
//
// Trigger an SW SMI to do BiosWriteProtect
//
if ((BxtSeries == BxtP) && (LockDownConfig->BiosLock == TRUE)) {
  IoWrite8 (R_APM_CNT, (UINT8) LockDownConfig->BiosLockSwSmiNumber);
}
```

SW_SMI_BIOS_LOCK

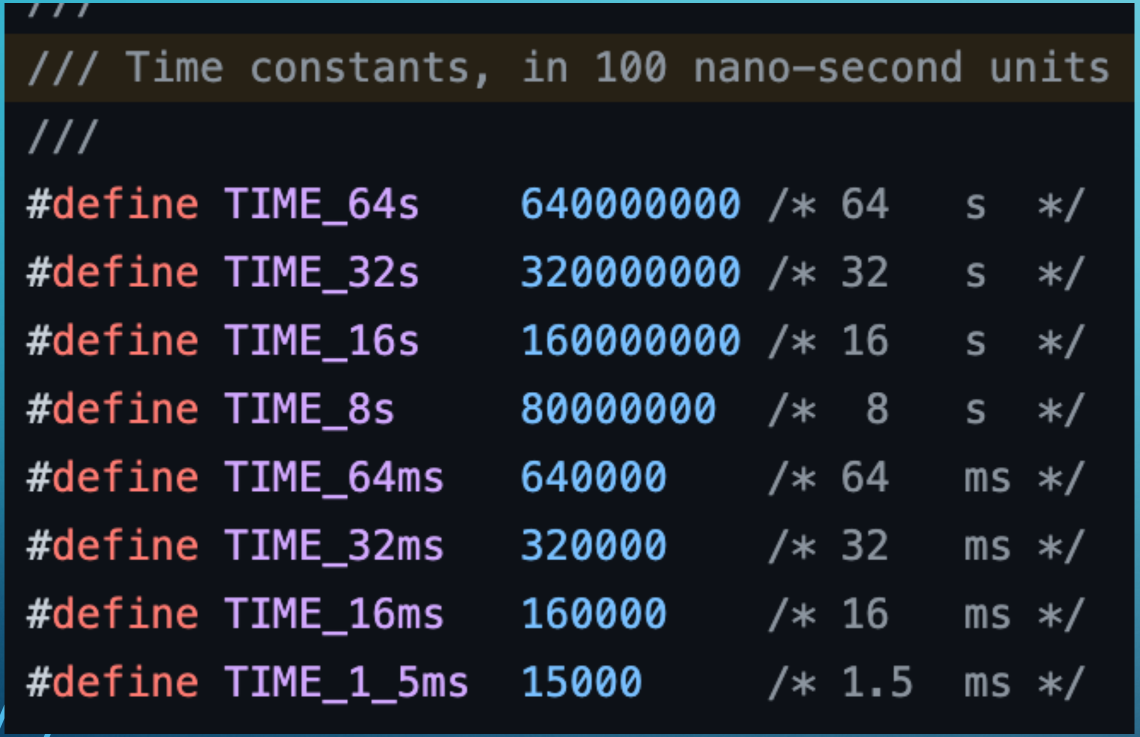
PERIODIC TIMER SMI HANDLER

- The Flash SPI SMI# Enable (HSFC.FSMIE) bit can be cleared by a kernel driver using MMIO
  - CHIPSEC clears the bit when setting the size (FDBC) per SPI command cycle
- The periodic timer SMI handler keeps enabling it



```
if ( HSFCTL_ERASE_CYCLE != hsfctl_spi_cycle_cmd ):
    self.spi_reg_write( self.hsfc_off + 0x1, dbc, 1 )
```

PERIODIC TIMER SMI HANDLER (CONT.)

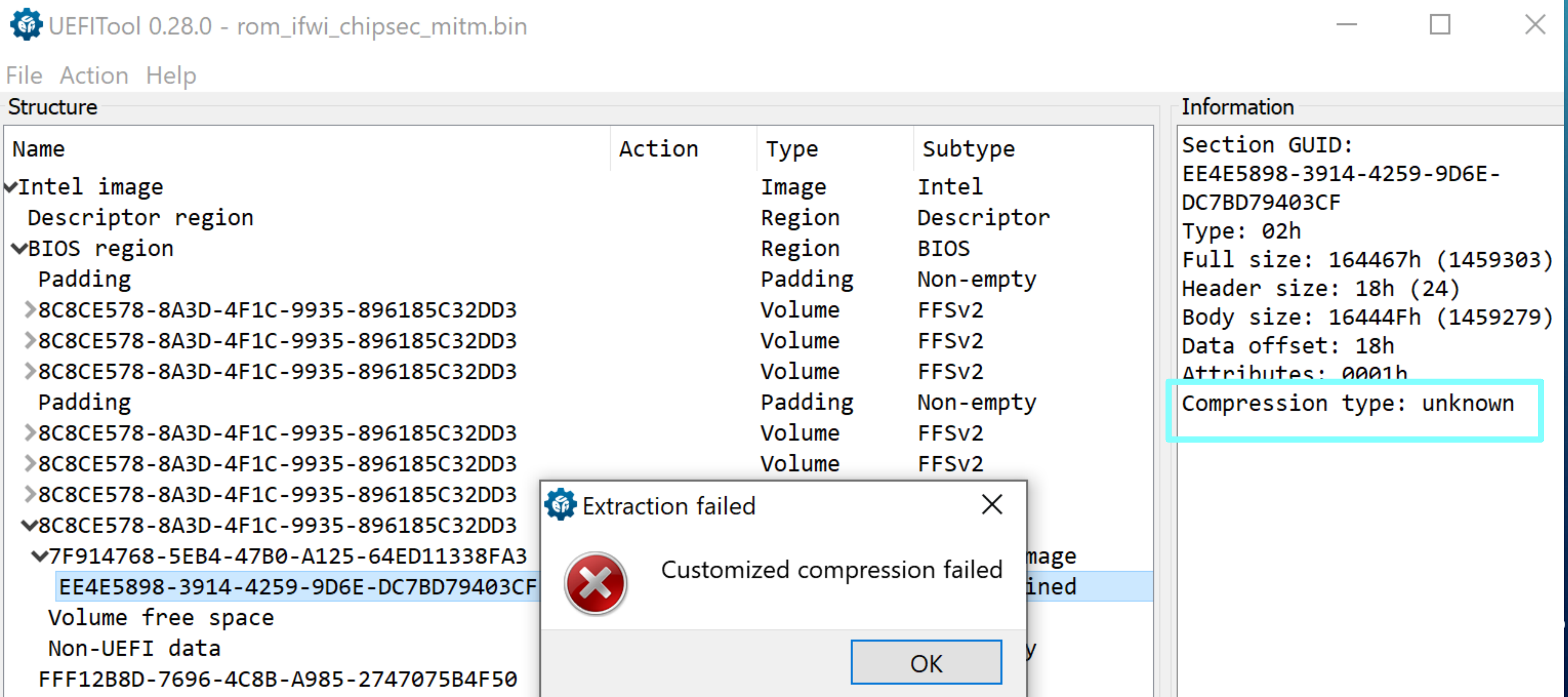


```
///
/// Time constants, in 100 nano-second units
///
#define TIME_64s    640000000 /* 64   s  */
#define TIME_32s    320000000 /* 32   s  */
#define TIME_16s    160000000 /* 16   s  */
#define TIME_8s     80000000  /*  8   s  */
#define TIME_64ms   640000    /* 64   ms */
#define TIME_32ms   320000    /* 32   ms */
#define TIME_16ms   160000    /* 16   ms */
#define TIME_1_5ms  15000     /* 1.5  ms */
```

- We can set the interval based on the definition
- 64ms or shorter required to generate the SPI SMI
  - The shorter the interval the more negative impact to system performance

PERIODIC TIMER SMI HANDLER (CONT.)

- The firmware acquisition performance in 64ms
  - Time overhead = 11%, Ratio of data overwritten by SPI SMI = 1.89%



UEFITool 0.28.0 - rom_ifwi_chipsec_mitm.bin

```
Name                                               Action   Type      Subtype
⌄Intel image                                                Image     Intel
  Descriptor region                                         Region    Descriptor
 ⌄BIOS region                                               Region    BIOS
   Padding                                                  Padding   Non-empty
  >8C8CE578-8A3D-4F1C-9935-896185C32DD3                     Volume    FFSv2
  >8C8CE578-8A3D-4F1C-9935-896185C32DD3                     Volume    FFSv2
  >8C8CE578-8A3D-4F1C-9935-896185C32DD3                     Volume    FFSv2
   Padding                                                  Padding   Non-empty
  >8C8CE578-8A3D-4F1C-9935-896185C32DD3                     Volume    FFSv2
  >8C8CE578-8A3D-4F1C-9935-896185C32DD3                     Volume    FFSv2
  >8C8CE578-8A3D-4F1C-9935-896185C32DD3
  ⌄8C8CE578-8A3D-4F1C-9935-896185C32DD3
   ⌄7F914768-5EB4-47B0-A125-64ED11338FA3                               mage
      EE4E5898-3914-4259-9D6E-DC7BD79403CF                             ined
    Volume free space
    Non-UEFI data                                                      y
   FFF12B8D-7696-4C8B-A985-2747075B4F50
```

```
Section GUID:
EE4E5898-3914-4259-9D6E-
DC7BD79403CF
Type: 02h
Full size: 164467h (1459303)
Header size: 18h (24)
Body size: 16444Fh (1459279)
Data offset: 18h
Attributes: 0001h
Compression type: unknown
```

Extraction failed

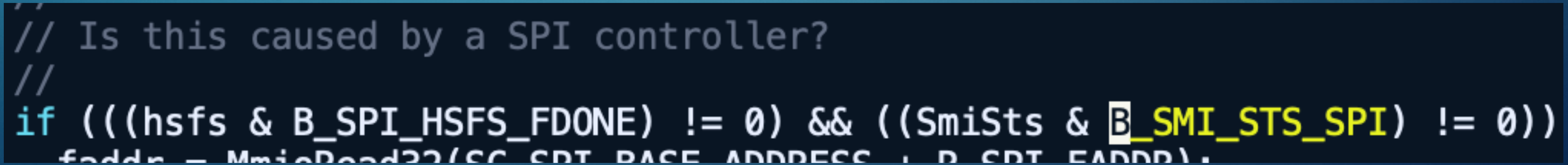
Customized compression failed

PERIODIC TIMER SMI HANDLER (CONT.)

- The msec interval SMIs prevent the OS boot?
- SpiMitm initally registers the 8sec handler then registers the 64 msec handler later after the boot

SPI SMI HANDLER

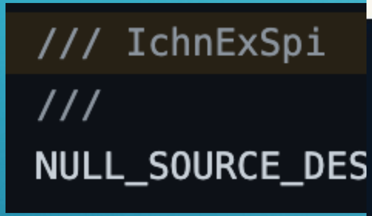
- Is this caused by FSMIE?
  - SPI SMI Status bit (SMI_STS.SPI_SMI_STS)
  - Flash Cycle Done bit (HSFS.FDONE)
- Overwrite Flash Data (FDATA0-15) registers
- Disable FSMIE to hide the MitM



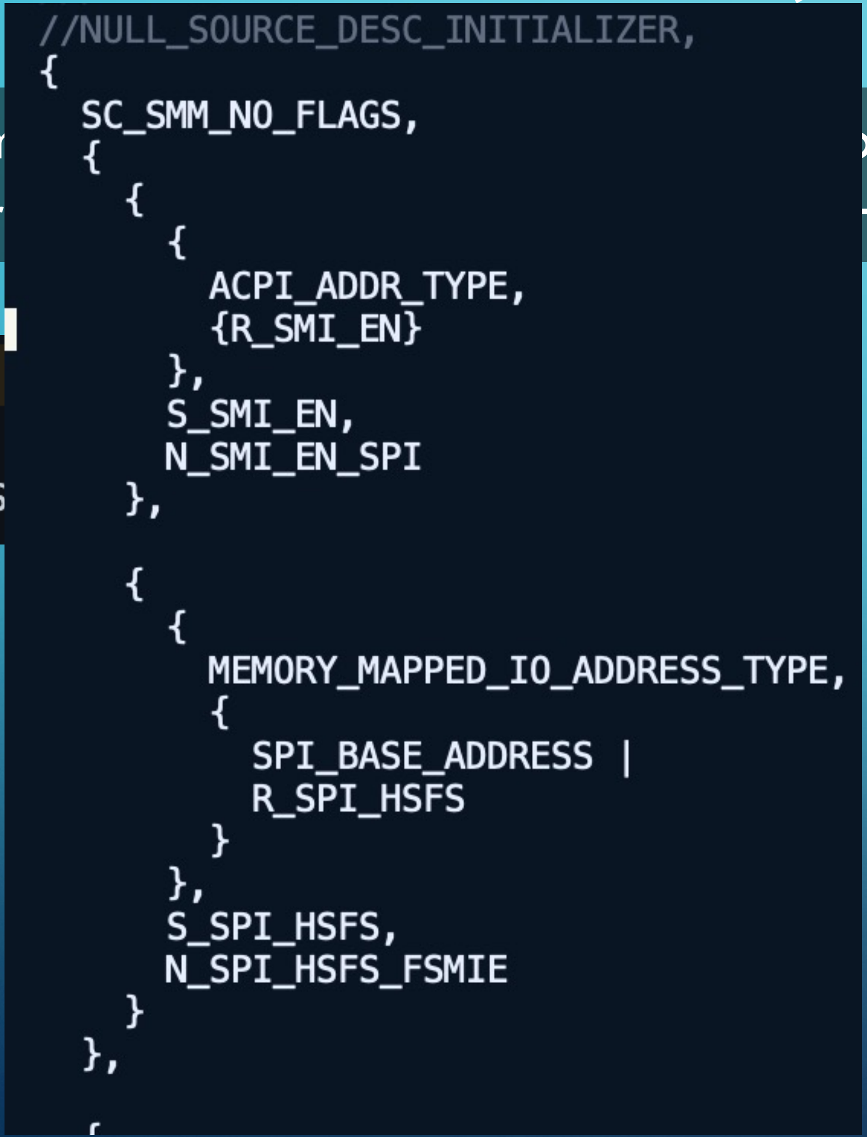
```
//
// Is this caused by a SPI controller?
//
if (((hsfs & B_SPI_HSFS_FDONE) != 0) && ((SmiSts & B_SMI_STS_SPI) != 0))
  faddr = MmioRead32(SC_SPI_BASE_ADDRESS + R_SPI_FADDR);
```

SPI SMI HANDLER (CONT.)

ASSERT [ScSm rms\Silicon\BroxtonSoC\BroxtonSiPkg
\SouthCluster Helpers.c(573): ((BOOLEAN)(0==1))



```
/// IchnExSpi
///
NULL_SOURCE_DES
```



```
//NULL_SOURCE_DESC_INITIALIZER,
{
  SC_SMM_NO_FLAGS,
  {
    {
      {
        ACPI_ADDR_TYPE,
        {R_SMI_EN}
      },
      S_SMI_EN,
      N_SMI_EN_SPI
    },

    {
      {
        MEMORY_MAPPED_IO_ADDRESS_TYPE,
        {
          SPI_BASE_ADDRESS |
          R_SPI_HSFS
        }
      },
      S_SPI_HSFS,
      N_SPI_HSFS_FSMIE
    }
  },
```

- No SPI logic definition in the firmware ☹
- I added the logic for the SMI

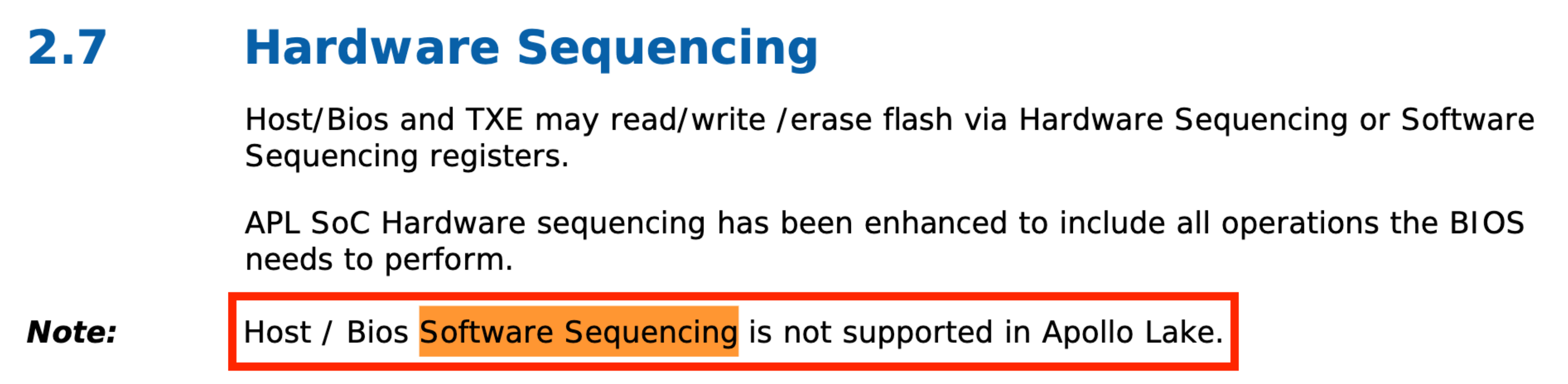
SEQUENCING

- Two types of SPI register access methods
  - "Hardware Sequencing" means the hardware picks the actual SPI commands that get sent for read/write
    - hides the details of SPI flash opcodes
  - "Software Sequencing" means we pick the actual SPI commands
    - offers a little more fine-grain control
- I've referred to only Hardware Sequencing so far

Source: Advanced x86: BIOS and System Management Mode Internals
SPI Flash Programming

SEQUENCING (CONT.)

- I also implemented the SPI SMI handler for SW Sequencing
  - Enable the SPI SMI# Enable (SSFC.SME) bit
  - Define the SPI logic for SW Sequecing
- But SW Sequencing is usually disabled after POST using the FLOCKDN bit
  - I checked HSFS.FLOCKDN was enabled by the CHIPSEC spi_lock module
- It's not supported in Apollo Lake SoC?



2.7 Hardware Sequencing

Host/Bios and TXE may read/write /erase flash via Hardware Sequencing or Software Sequencing registers.

APL SoC Hardware sequencing has been enhanced to include all operations the BIOS needs to perform.

Note: Host / Bios Software Sequencing is not supported in Apollo Lake.

SPIMITM VS. FIRMWARE SECURITY TOOLS

TEST STEPS

1. Build the firmware image with SpiMitm
2. Embed Hacking Team’s Vector-EDK with debug messages
   - rkloader and fsbg modules (no NTFS driver)
3. Acquire or scan the firmware using the security tools
   - Can the tools detect the Vector-EDK modules?

VS. OPEN-SOURCE TOOL (CHIPSEC)

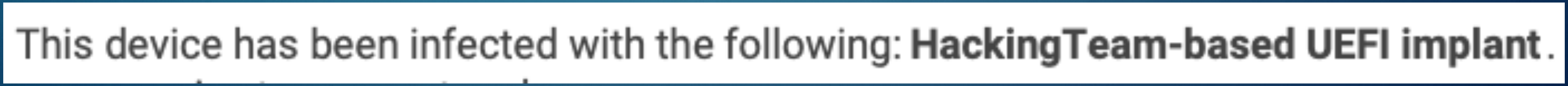
- Demo

VS. CLOSED-SOURCE TOOLS

- 4 firmware scanners including commercial products
- I don’t disclose the tested scanner names :-)
  - The purpose of this research is not to blame any specific product, but to check the actual efficacy

RESULT

- The 3 scanners couldn’t discover Vector-EDK even if the MitM was disabled
  - They don’t support the Atom platform
  - Or simply the detection capabilities are poor
- The last one detected Vector-EDK with the MitM!



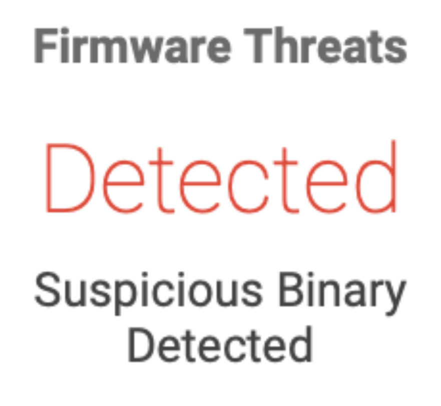
This device has been infected with the following: **HackingTeam-based UEFI implant**.

RESULT (CONT.)

- I reversed the scanner then identified this had 2 methods for the firmware acquisition
  - Hardware Sequencing that programs a SPI flash
  - MMIO of the BIOS region based on the BIOS Decode Enable (BDE) register value
- The latter one was not covered by SpiMitm initially

RESULT (CONT.)

- I added a code clearing BDE to SpiMitm
- The improved SpiMitm could prevent the tool from detecting Vector-EDK :-)

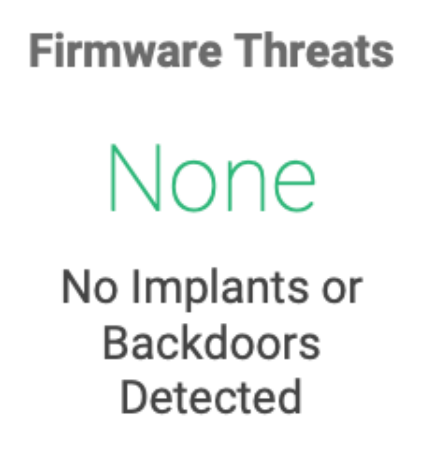


Firmware Threats

Detected

Suspicious Binary Detected

SpiMitm improved



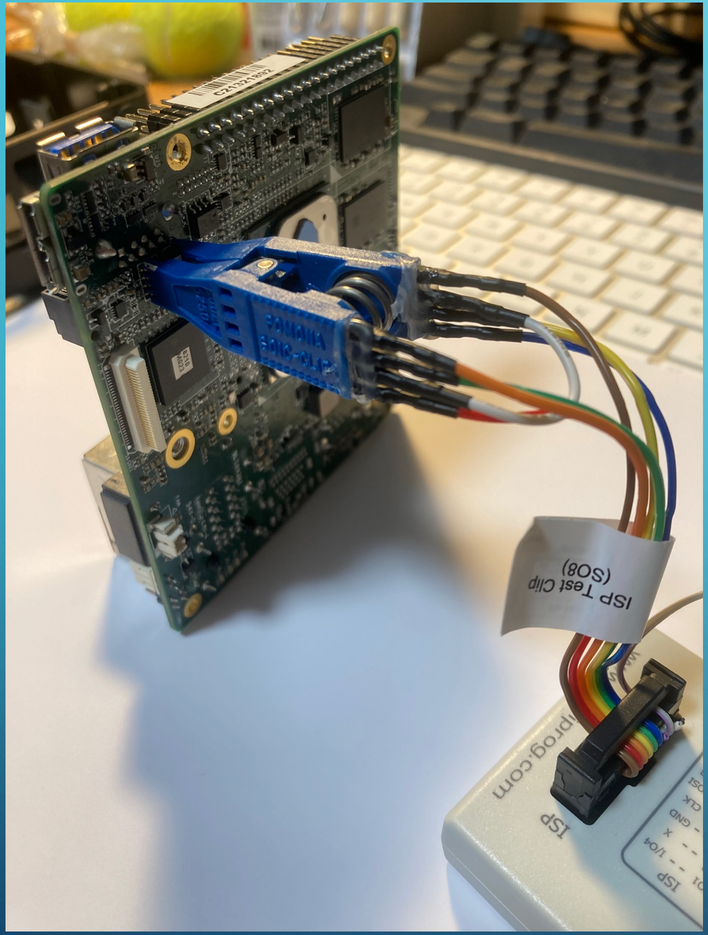
Firmware Threats

None

No Implants or Backdoors Detected

COUNTERMEASURES

HARDWARE-BASED ACQUISITION



- Use a SPI programmer
  - not affected by SMM rootkits
  - but not scalable ☹

SMRAM FORENSICS

- Dump SMRAM using hardware debugger
  - It’s hard to enable the Intel DCI on normal platforms :-(
  - The dump takes long time (8MB SMRAM in a few hours)
- Parse the SMRAM then detect malicious SMI handlers
  - smram_parse.py by Dmytro Oleksiuk
  - The SMM structures are different for different firmware ☹

```
0x7b4e0c18: periodic timer SMI 0x7b530640 with Period 1000000 and SmiTickInterval
640000 (image = SpiMitm, link error = False)
…
0x7b4ebd18: Ichn/IchnEx SMI 0x7b5304c8 with context type 0x2e (image = SpiMitm, link
error = False)
```

OTHER SOFTWARE-BASED DETECTIONS

- Notice the MitM attack possibility
  - Detect the SMM code modification using Measured Boot
    - compare hash values of the OEM code (TPM PCR[0])
  - Periodically check the FSMIE bit
  - Detect FV decompress/parse errrors after the acquisition
- We can’t identify the malicious implants but we can recognize “something is wrong” at least

WRAP-UP

WRAP-UP

- The reality of the firmware security tools
  - Only one scanner could detect VEDK without the MitM
  - SpiMitm could hide VEDK from the scanner
- Every firmware doesn't always implement the SPI logic for the SMI
  - Attackers have to not only bypass BootGuard but also append the logic by the RE
- Once the MitM module is installed, it's hard to detect the threat explicitly using software-based approaches

ACKNOWLEDGMENT

- Satoshi Tanda
- Alex Matrosov
- Brian Baskin

ANY QUESTIONS?

- https://github.com/TakahiroHaruyama/SpiMitm