presented by

Attacking and Defending the Platform

Spring 2018 UEFI Seminar and Plugfest

March 26-30, 2018

Presented by Erik Bjorge and Maggie Jauregui (Intel)

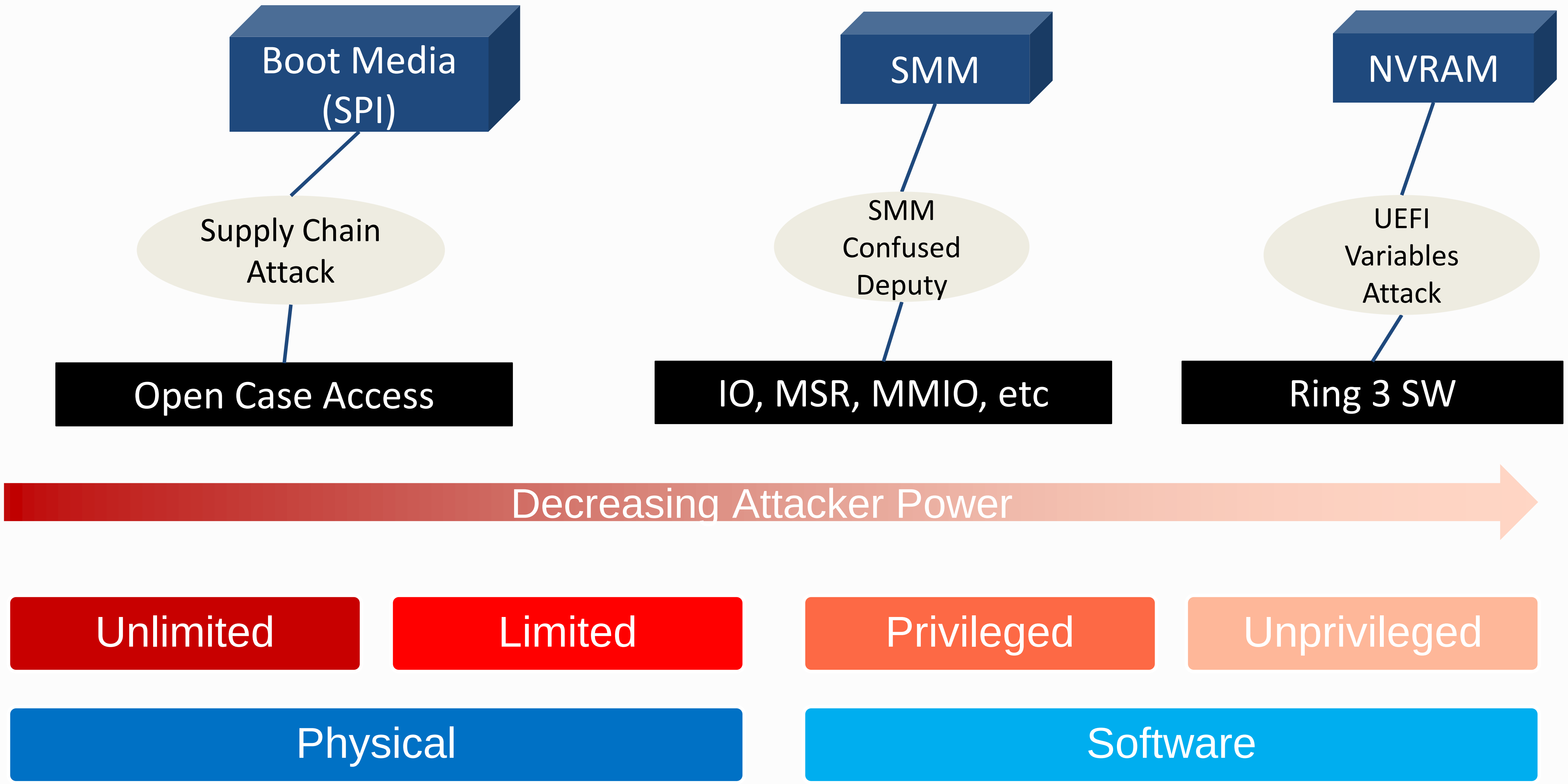
Legal Notice

No computer system can be absolutely secure.

Intel, the Intel logo are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others

Today’s Attack Scenarios



Example: UEFI Variable Attack from privileged ring 3 process

Unprivileged Software Attack

Possible Security Impacts

- **Overwrite early firmware code/data** if (physical address) pointers are stored in unprotected variables
- **Bypass UEFI and OS Secure Boot** if its configuration or keys are stored in unprotected variables
- **Bypass or disable hardware protections** if their policies are stored in unprotected variables
- **Make the system unable to boot (brick)** if boot-essential settings are stored in unprotected variables
- **Communication Channel** if malware uses variables for retrieval of data at a later time (e.g. after OS wipe)

Authenticated Variables

EDK II Variable Lock Protocol (Read-only Variables)

VarCheckLib

UEFI Variable Mitigation Options

Variables Protection Attributes

Boot Service (BS)

- Accessible to DXE drivers / Boot Loaders at boot time
- No longer accessible at run-time (after `ExitBootServices`)

Authenticated Write Access

- Digitally signed with *MonotonicCount* incrementing each successive variable update to protect from replay attacks
- List of signatures supported by the firmware is stored in `SignatureSupport` variable

Time Based Authenticated Write Access

- Signed with `TimeStamp` (time at signing) to protect from replay attacks
- `TimeStamp` should be greater than `TimeStamp` in existing variable
- Used by Secure Boot: PK verifies PK/KEK update, KEK verifies db/dbx update
- `certdb` variable stores certificates to verify non PK/KEK/db(x) variables

EDK II Read-Only Variables

- EDK II implements **`VARIABLE_LOCK_PROTOCOL`** which provides a mechanism to make some variables "**`Read-Only`**" during Run-time OS
- DXE drivers make UEFI variables **`Read-Only`** using **`RequestToLock()`** API before **`EndOfDxe`** event
- After **`EndOfDxe`** event (e.g. during OS runtime), all registered variables cannot be updated or removed (enforced by **`SetVariable`** API)
- Lock is transient, firmware has to request locking variables every boot. Before **`EndOfDxe`** variables are not locked

VarCheckLib

A single place to check for acceptable variable contents

- Each variable name/GUID is mapped to rules
- Return appropriate error when attempting to set invalid data to a given variable
- Begin checking at EndOfDxe (prior to execution of 3^{rd} party code)

https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Library/VarCheckLib/VarCheckLib.c `chipsec_util uefi var-write`

Attack: Storing Data in UEFI Variables

Example: SMM Confused Deputy

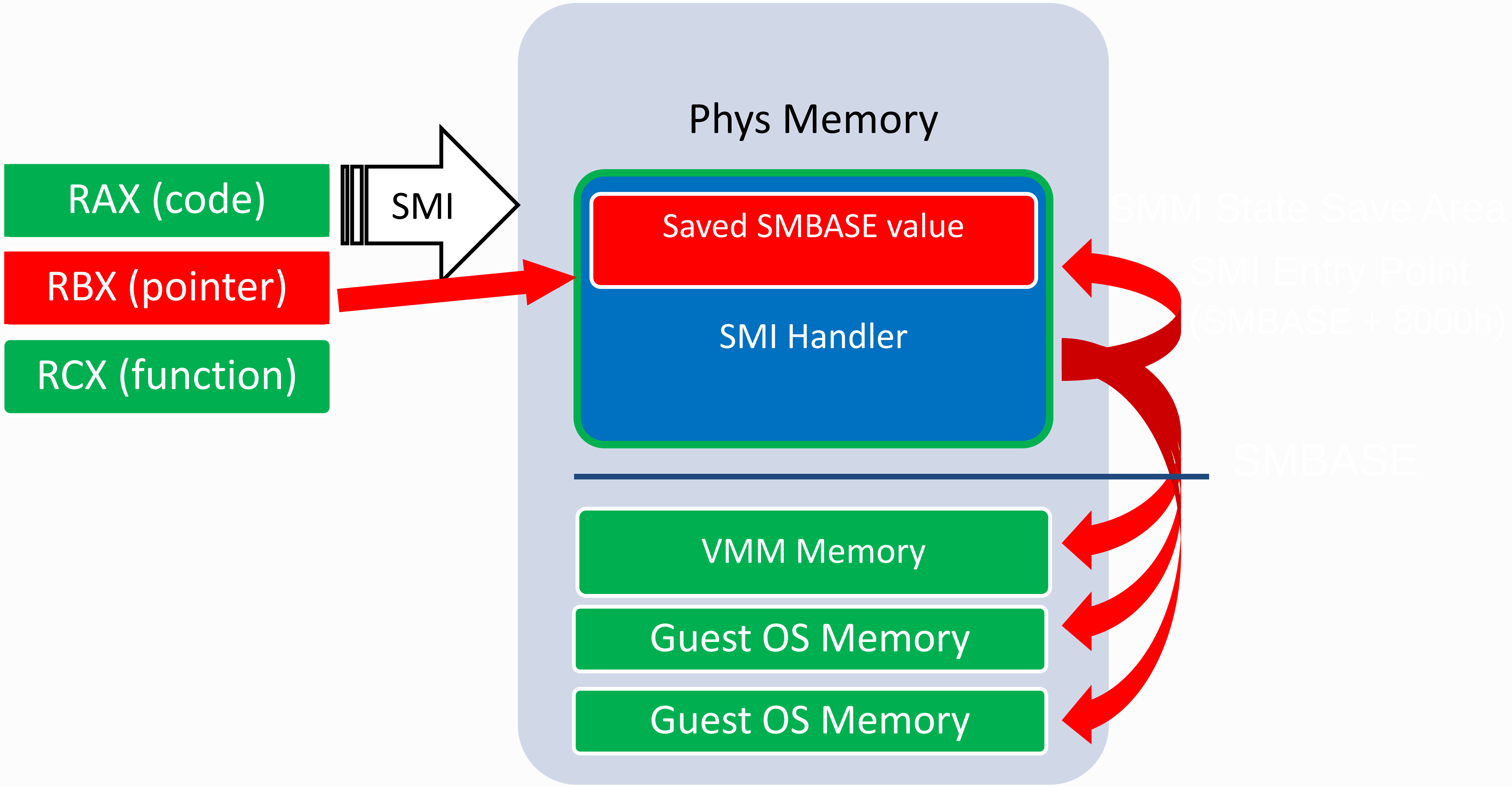
Privileged Software Attack

SMI Input Pointer Vulnerabilities

- When OS triggers SMI (e.g. SW SMI via I/O port 0xB2) it passes arguments to SMI handler via general purpose registers
- OS may also pass an address (pointer) to a structure through which an SMI handler can read arguments & returns result
- SMI handlers traditionally were not validating that such pointers are outside of SMRAM
- If an exploit passes an address which is inside SMRAM, SMI handler may write onto itself on behalf of the exploit

- **References:** A New Class of Vulnerability in SMI Handlers (2015)

SMI “Confused Deputy” Attacks



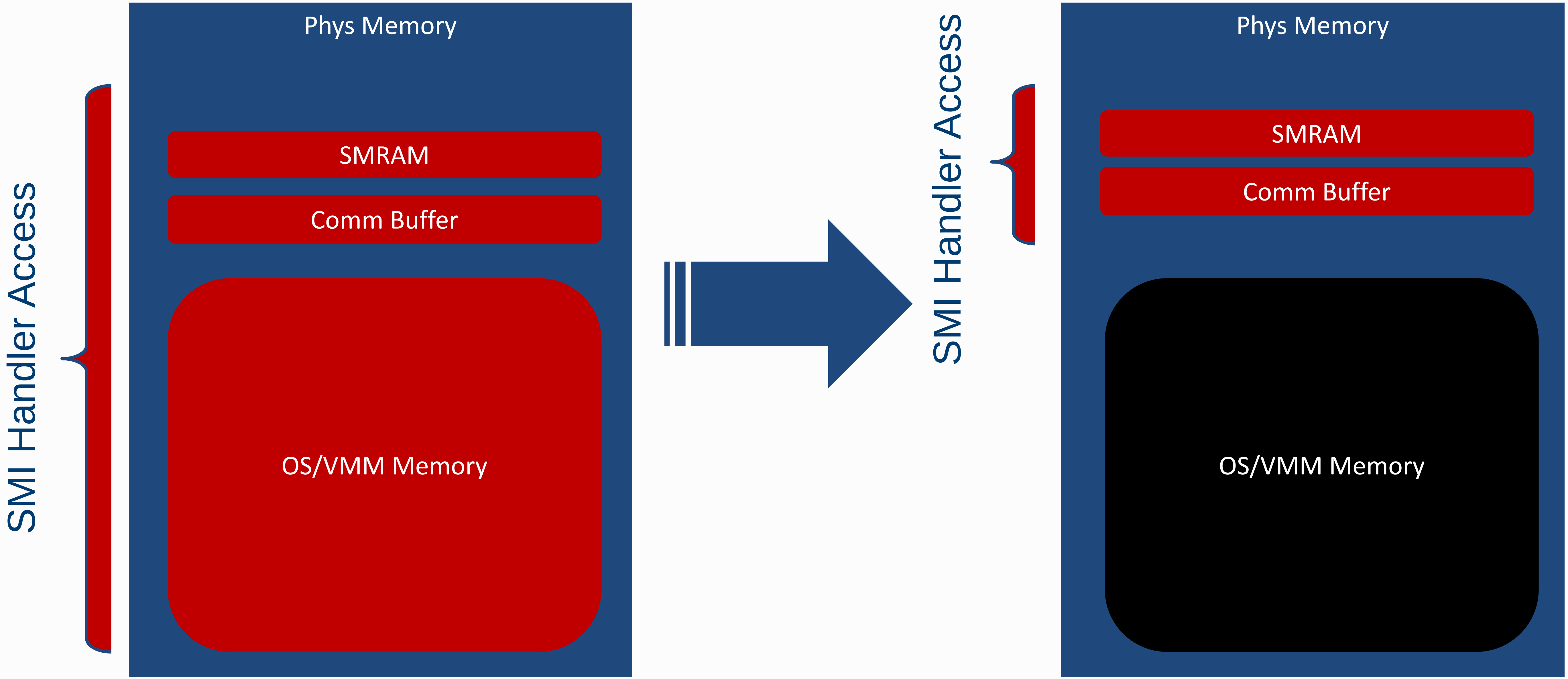
Attacker can target SMM itself or bypass VMM protections, writing to VMM or other Guest VM memory

Limiting SMI Handler Memory Map to Addresses Reserved for Firmware

CHIPSEC Testing

Mitigation Options

SMI Handler Memory Map Restriction



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
```

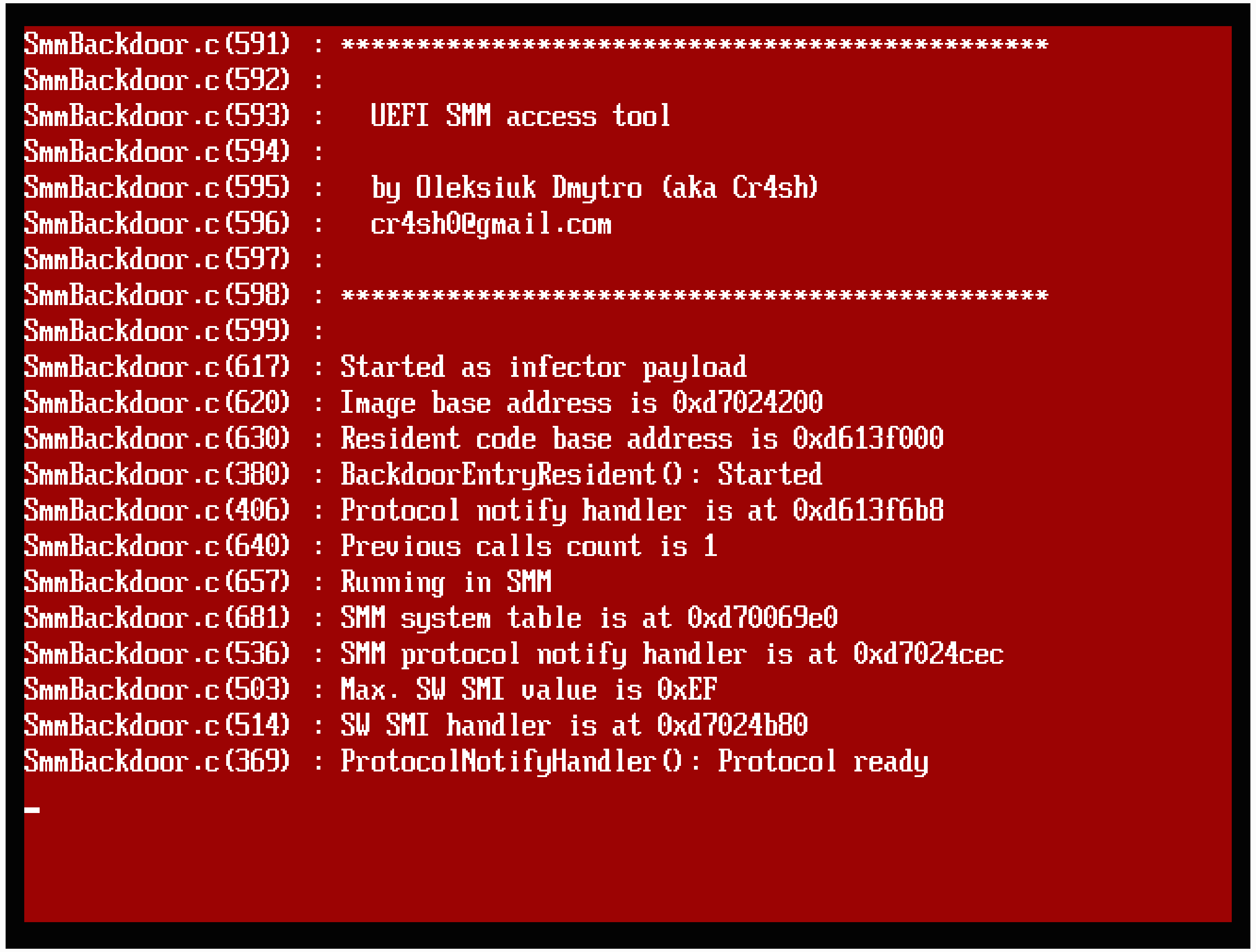
https://www.youtube.com/watch?v=z2Qf45nUeaA

Example: Supply Chain Attack

Limited Physical Access Attack

PoC SmmBackdoor by Dmytro Oleksiuk

- Installed by adding additional sections to existing SMM driver
- Provides SMI interfaces for OS level caller
- Provides read/write memory access. Easily extensible



[Building reliable SMM backdoor for UEFI based platforms]

First Commercial UEFI Rootkit from HackingTeam

intel Security

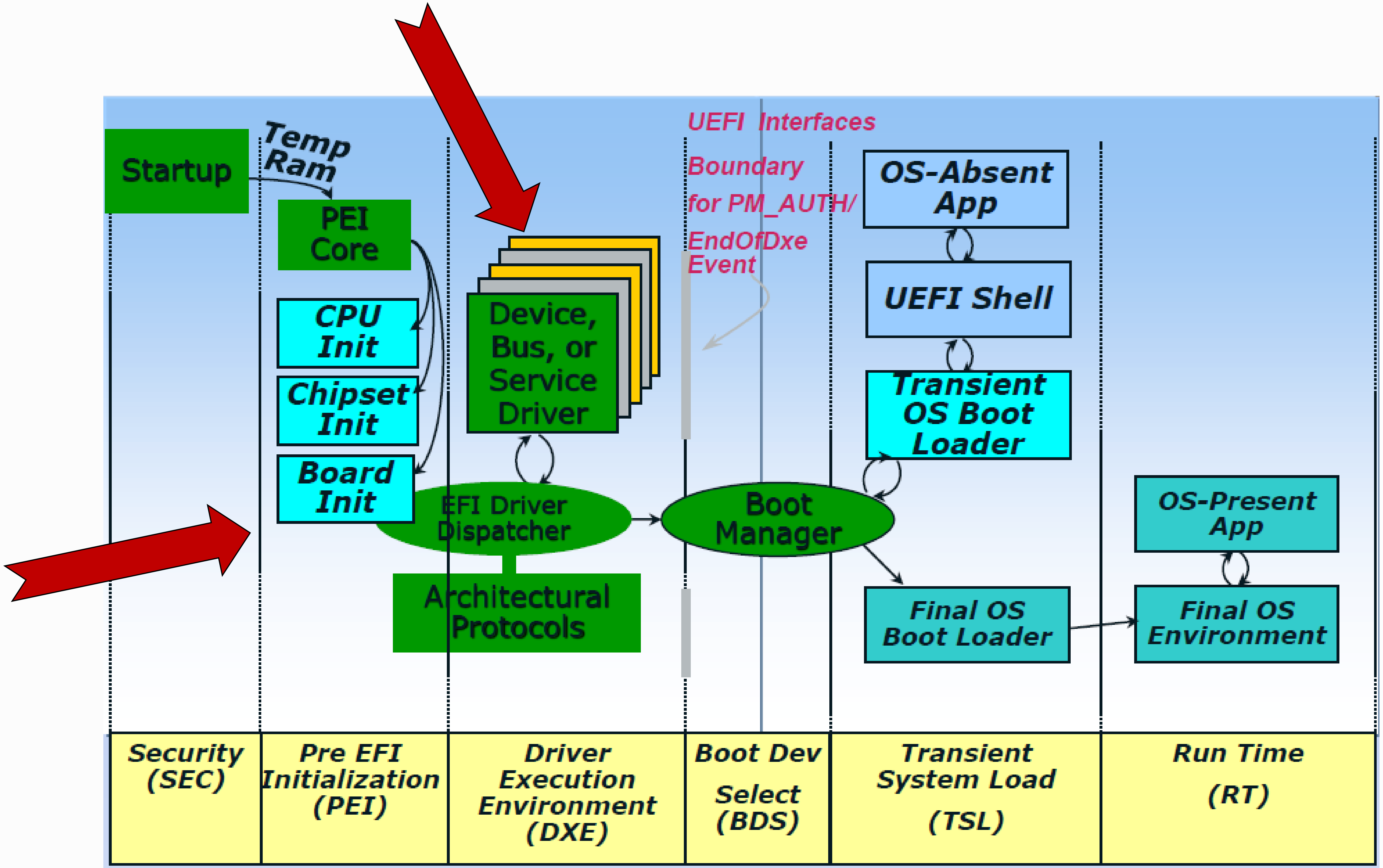
Products What's New

Hacking Team's "Bad BIOS": A Commercial Rootkit for UEFI Firmware?

Tuesday, July 14, 2015

Attack Details

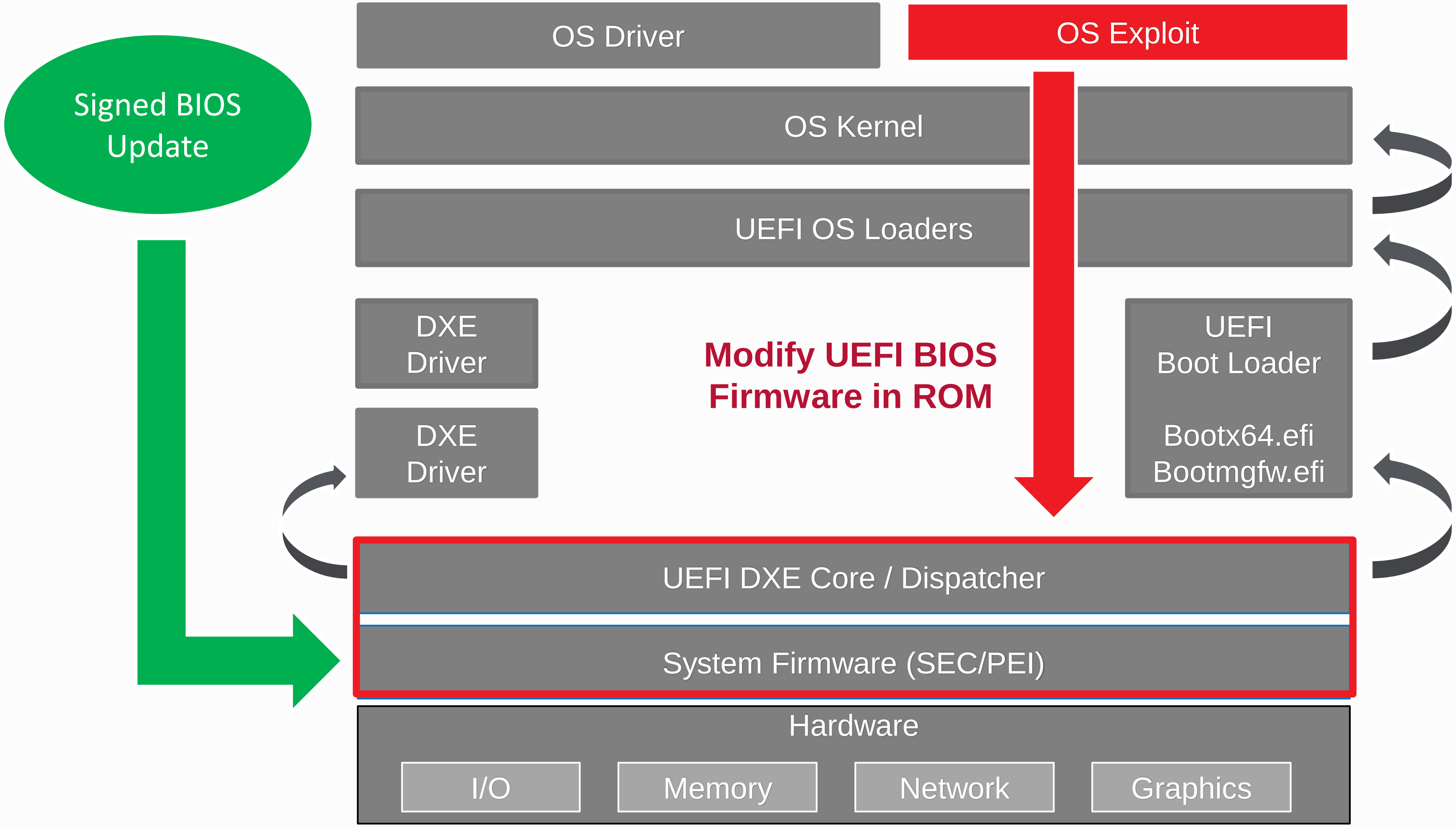
The examination of commercial malware developed by Hacking Team has revealed much to the security community. Of particular interest to platform security researchers at Intel's Advanced Threat Research team (ATR) is the presence of what appears to be a UEFI-based persistent infection mechanism. ATR has been researching vulnerabilities related to system firmware and working with a community of firmware developers and platform manufacturers to mitigate these threats. Others have also posted good information about this issue. Here, we will provide some preliminary analysis of the firmware threat.



From [Secure Boot, Network Boot, Verified Boot, oh my]() and almost every publication on UEFI

Attacking without Physical Access

(targeting vulnerable firmware)



CHIPSEC Vulnerability testing

CHIPSEC Whitelist testing

Hardware Root of Trust

Protection and Mitigation Options

Checking for BIOS Write Protection

 # chipsec_main.py --module common.bios_wp

```
[*] running module: chipsec.modules.common.bios_wp
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BIOS Control = 0x02
    [05] SMM_BWP = 0 (SMM BIOS Write Protection)
    [04] TSS     = 0 (Top Swap Status)
    [01] BLE     = 1 (BIOS Lock Enable)
    [00] BIOSWE  = 0 (BIOS Write Enable)

[!] Enhanced SMM BIOS region write protection has not been enabled (SMM_BWP is not used)

[*] BIOS Region: Base = 0x00500000, Limit = 0x007FFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 87FF0780 | 00780000 | 007FF000 | 1   | 0
PR1 (78)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR2 (7C)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR3 (80)     | 00000000 | 00000000 | 00000000 | 0   | 0
PR4 (84)     | 00000000 | 00000000 | 00000000 | 0   | 0
[!] SPI protected ranges write-protect parts of BIOS region (other parts of BIOS can be modified)
[!] BIOS should enable all available SMM based write protection mechanisms or configure SPI protected
ranges to protect the entire BIOS region
[-] FAILED: BIOS is NOT protected completely
```

CHIPSEC: Detecting Firmware Modification

- Use CHIPSEC to generate and check hashes of firmware modules
  - Use whitelists to detect changes from the original firmware
  - Whitelist can be generated by user or manufacturer
  - Whitelists can be signed to verify source of information

- More info including full module source and blog:
  - https://github.com/chipsec/chipsec/blob/master/chipsec/modules/tools/uefi/whitelist.py
  - https://software.intel.com/en-us/blogs/2017/12/05/using-whitelists-to-improve-firmware-security

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 is a way to generate clean (uninfected) list of EFI executables. For example, from the update image downloaded from the vendor web-site

Checking 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      256)
    d359a9546b277f16bc495fe7b2e8839b5d0389a8 (sha1)
    {EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B}
    <unknown>
[!] found EFI executable not in the list:
    dd2b99df1f10459d3a9d173240e909de28eb895614a6b3b7720eebf470a98
    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

Firmware Forensic Artifacts to Consider

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
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: Embedded Controller, HDD/SSD, NIC, Baseboard Management Controller (BMC) etc.
15. MBR/VBR or UEFI GUID Partition Table (GPT)
16. Files on EFI system partition (boot loaders)

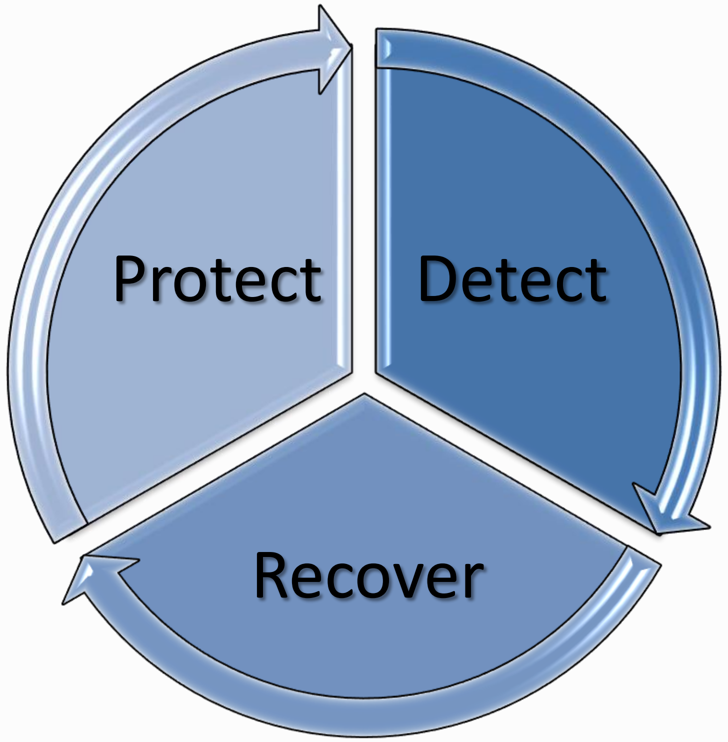
Conclusions

Resilient Defense

Boot Media

Runtime Firmware
(eg. SMM)

HW Configuration



Decreasing Attacker Power

Unlimited	Limited	Privileged	Unprivileged
Physical		Software	

Thanks for attending the Spring 2018 UEFI Plugfest

For more information on the UEFI Forum and UEFI Specifications, visit http://www.uefi.org

presented by