

UEFI BIOS セキュリティ

@ SECURITY CAMP 2021-08-13



一般的なUEFIの認知度

覚えておくこと

- BIOSはただのソフトウェア
 - 人間が書いて出荷している
 - バグがある、脆弱性がある
 - 悪意あるコードが書ける
- 違うこと
 - 保存されているストレージ
 - 実行される環境
 - 理解している人、解析できる人がまだ少ない
 - セキュリティリスクを確認&緩和できるソフトウェアがゼロに近い

講師について

- 丹田賢 / Satoshi Tanda ([@SatoshiTanda](#) / [@standa_t](#))
 - バンクーバー在住9年目、[職業エンジニア10年以上](#)
- システムソフトウェアエンジニア
 - セキュリティソフトウェアの研究開発
 - @クラウドストライク(2016～)
 - @FFRI(2009～2012)
- ソフトウェアリバーースエンジニア
 - 産業システムの脆弱性解析、マルウェア解析
 - @Sophos(2015～2016)
 - @GE(2013～2014)
- トレーナー
 - [UEFI+Intelプラットフォームでのハイパーバイザー開発コース](#)
- スピーカー
 - CodeBlue、Recon、Bluehat、Nullcon、etc

クラウドストライクについて

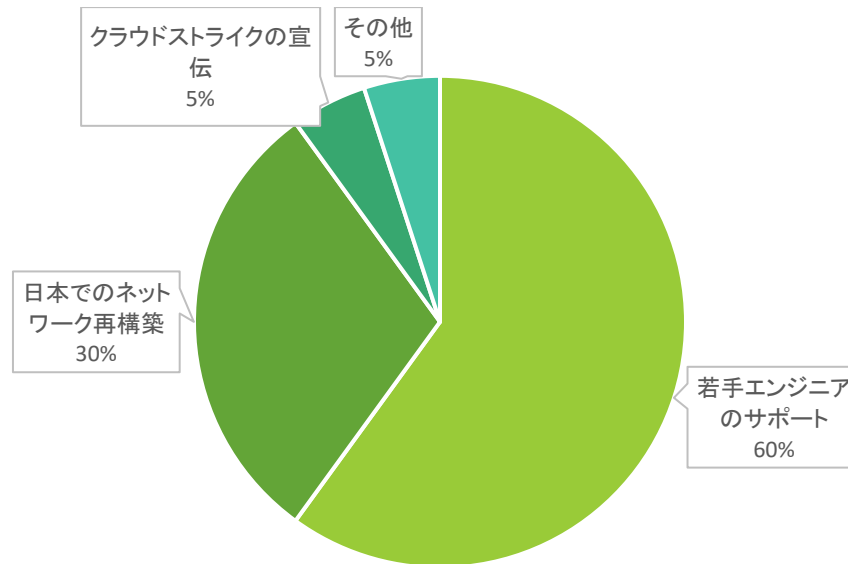
CrowdStrike® Inc. (Nasdaq:CRWD)は、サイバーセキュリティのグローバルリーダーであり、セキュリティ侵害を阻止するためにゼロから構築したエンドポイント・ワークロード保護プラットフォームにより、クラウド時代のセキュリティを再定義しています。

- クラウドを活用したセキュリティソフトウェア・サービスベンダー
- 日本在住のエンジニアも募集中！
 - 同僚エンジニアが非常にデキる、上司が理解がある
 - 仕事の内容＆規模が面白い
 - ワークライフバランスが良い
 - COVID以前から完全リモート
 - 給与もわるくない
 - アメリカ移住も支援(もし望むなら)
- インターン＆就職に興味のある人はDM

セキュリティキャンプについて

- 川古谷さん [@kwky](#) (プロデューサー) に声をかけていただきました！
 - 日本に居たとき勉強会、ブログあたりで知り合った
 - 古いゆるいつながり。最後にあったのも8年前(?)
 - 🏠 みなさんもアウトプットをしてください！
 - 🏠 みなさんも知り合いをつくってください！

■ 私の動機



どの業界で使える知識？

- セキュリティベンダー

- アンチウイルス (AV) や EDR

- ファームウェアに対する脅威を管理、軽減できる？

- インシデントレスポンス

- ファームウェアに対する攻撃が含まれる可能性を考慮、対応できる？

- 非セキュリティベンダー

- BIOSベンダー

- American Megatrends International

- OEM

- Dell, Microsoft, Lenovo

- X86ベースのゲームコンソール

- Sony – PlayStation
 - Microsoft - Xbox

UEFI BIOS概要

ファームウェアとは

- ソフトウェアの一種でデバイスの制御を行う
 - 組み込み機器では通常ROMに保存
- バグ、脆弱性、マルウェアがあり得る
 - ホストファームウェア: BIOS(本講義)
 - デバイスファームウェア
 - Thunderbolt Controller - <https://thunderspy.io/>
 - HDD Controller - <https://www.wired.com/2015/02/nsa-firmware-hacking/>
 - USB Controller - <https://shop.hak5.org/products/usb-rubber-ducky-deluxe>
 - Baseboard management controller (BMC) - <https://eclipsium.com/2019/01/26/the-missing-security-primer-for-bare-metal-cloud-services/>

BIOSとは

- 広義: ハードウェアの初期化とOSの起動を担当するソフトウェア
 - システム起動時にCPUによって最初に行われる
 - OSを読み込むためのデバイスドライバを実装する
 - HTTP, PXE, HDD, USB, DVE etc etc
 - OSローダーを実行、処理を引き渡す
- 狭義: レガシーBIOS (UEFI等と比較して)
- 他の“BIOS”:
 - iBoot - iPhone, macOS on Apple Silicon, macOS on T2
 - Coreboot – Chrome OS
 - Linux Boot – Facebook datacenters
 - Proprietary – Android phones

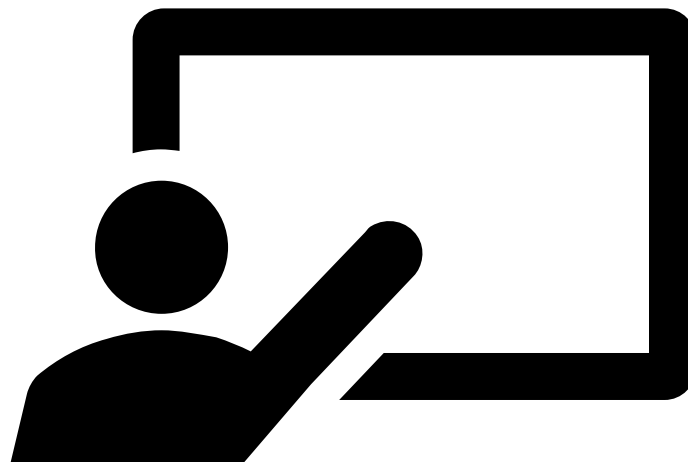
<https://support.apple.com/en-ca/guide/security/sec114e4db04/web>

<https://www.chromium.org/chromium-os/developer-information-for-chrome-os-devices/custom-firmware>

<https://www.linuxboot.org/>

UEFIとは

- BIOSの実装を標準化し、レガシーBIOSを置き換えるために策定された仕様群
- レガシーBIOSは
 - 他のCPUアーキテクチャーのサポートが困難
 - 開発が困難(16bit, real-mode, etc)
- 2005年: EFI(仕様v1.1)が策定
 - macOS on Intelは、厳密にはEFIベースのBIOSを使用
- 2006年: UEFI(仕様v2.0)が策定
 - 現在実質100%のOEM PCがUEFIベースのBIOSを使用
- 2021年: UEFI v2.9が最新
 - <https://uefi.org/specifications>

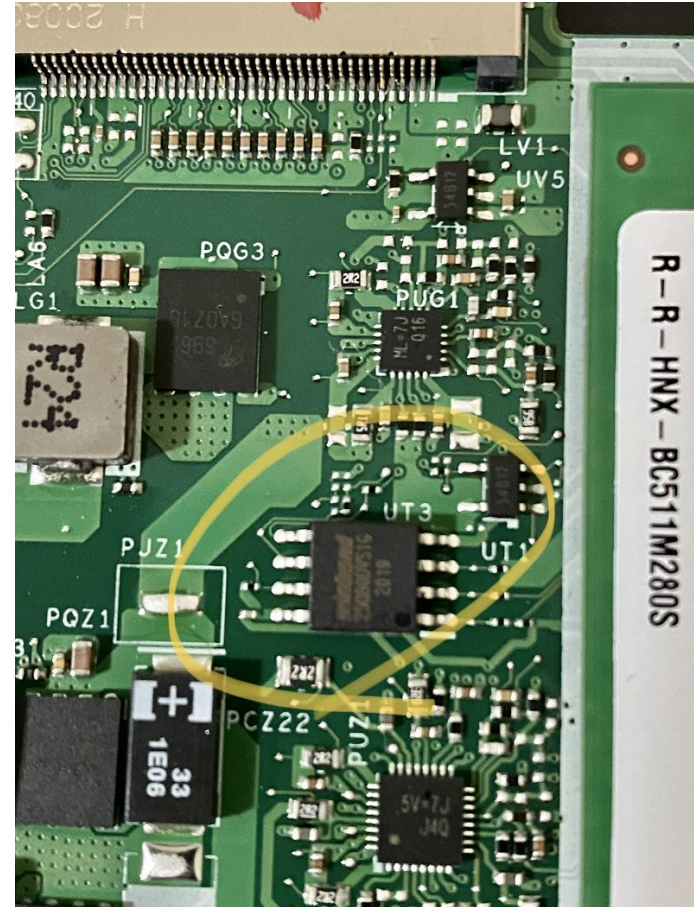
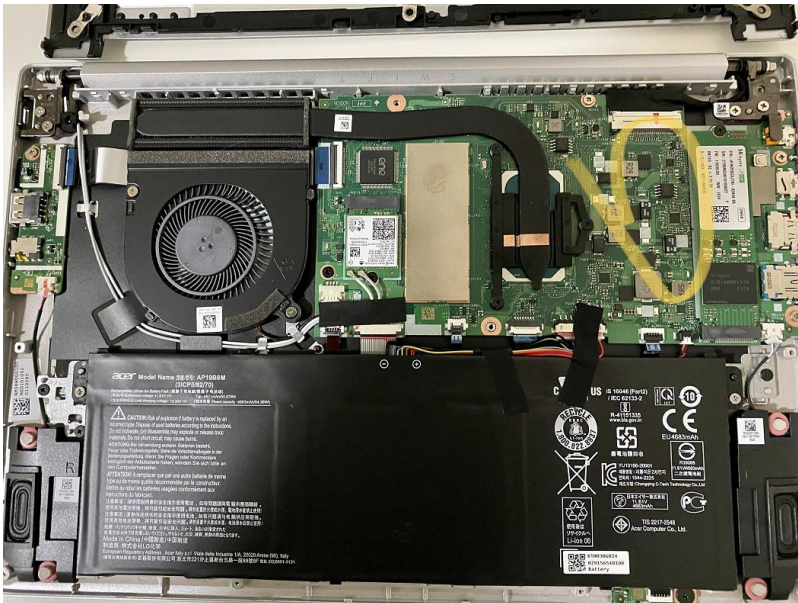


Demo: VMwareのUEFIバージョンの確認

UEFIシェルの「VER」コマンドを用いて実装された仕様のバージョンを確認する。

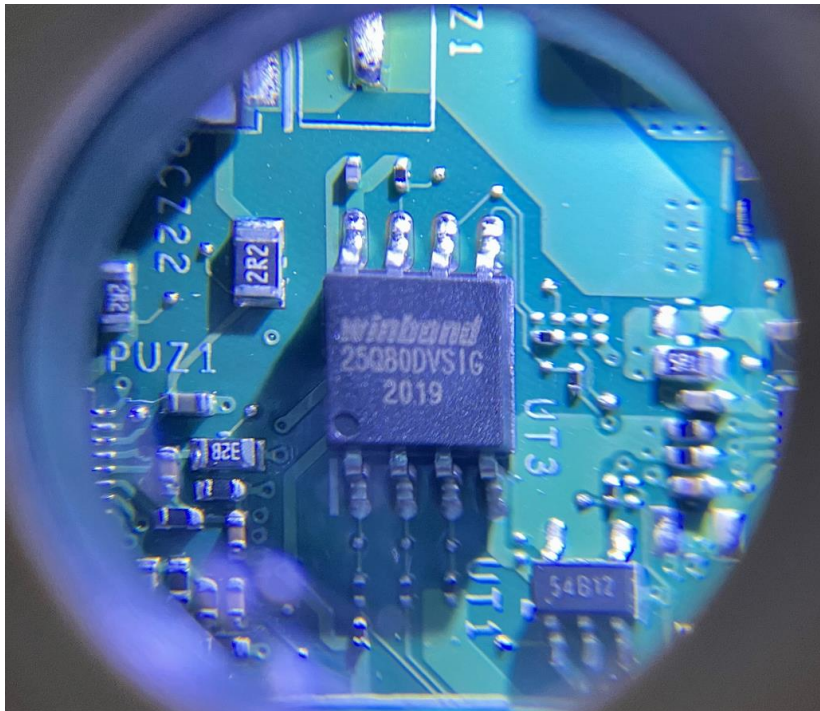
UEFIのストレージ

- SPIフラッシュに保存されている
 - ディスクとは別のストレージ



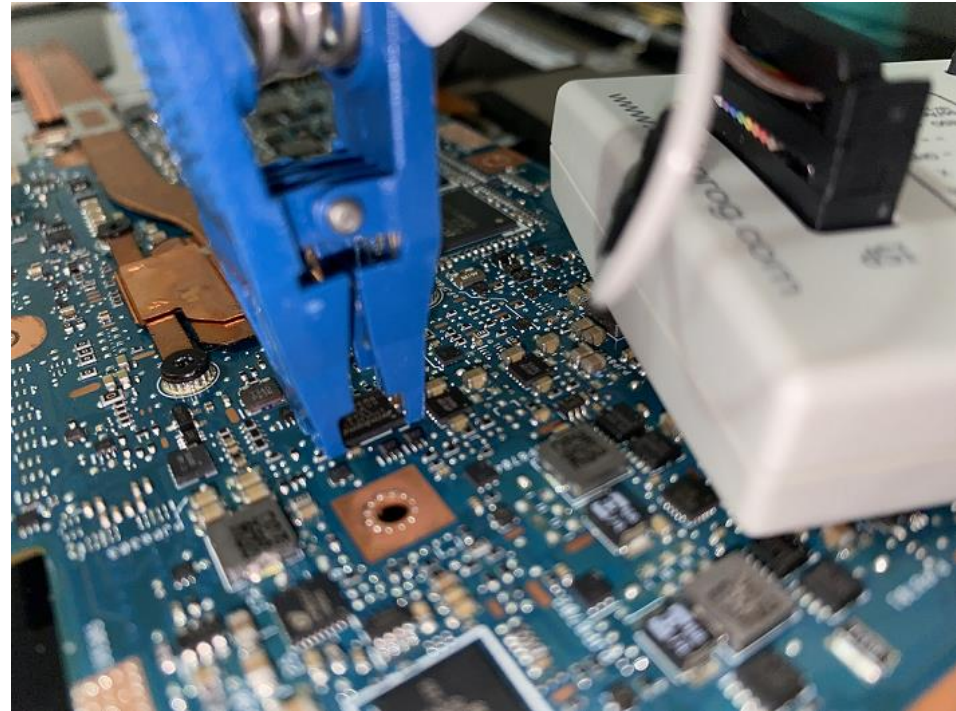
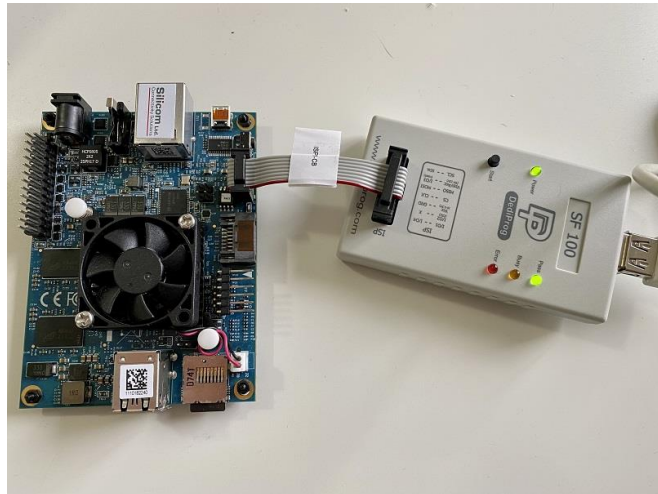
UEFIのストレージ

- Winbond製が一般的。通常8-32MB
- 写真の例は[W25Q80DV](#)




アクセス方法

- ソフトウェアから(後述)
 - UEFI変数はソフトウェアから読み書き可能
- ハードウェアから
 - SPIフラッシュプログラマー




ストレージの内容 (Intel)

- 複数のファームウェアが入っている
 - 例: Gigabit Ethernet, Intel Management Engine, and BIOS

 UEFITool NE alpha 58 (Nov 7 2020) - MBP141.rom

File Action View Help

Structure			
Name	Action	Type	Subtype
✓Intel image		Image	Intel
Descriptor region		Region	Descriptor
>PDR region		Region	PDR
>ME region		Region	ME
>BIOS region		Region	BIOS

 UEFITool NE alpha 58 (Nov 7 2020) - XPS_9360_2.10.0.bin

File Action View Help

Structure			
Name	Action	Type	Subtype
✓Intel image		Image	Intel
Descriptor region		Region	Descriptor
GbE region		Region	GbE
ME region		Region	ME
>BIOS region		Region	BIOS

BIOSイメージの内容

- Image -> Volumes
 - VolumesはFirmware File System (FFS)を持つ

Structure			
Name	Action	Type	Subtype
✓AMI Aptio capsule		Capsule	Aptio signed
✓UEFI image		Image	UEFI
Padding		Padding	Non-empty
> EfiFirmwareFileSystem2Guid		Volume	FFSv2
Padding		Padding	Empty (0xFF)
> 7DCCF422-45F6-4951-A557-CC421DA31599		Volume	FFSv2
> 4F1C52D3-D824-4D2A-A2F0-EC40C23C5916		Volume	FFSv2
> AFDD39F1-19D7-4501-A730-CE5A27E1154B		Volume	FFSv2
> 61C0F511-A691-4F54-974F-B9A42172CE53		Volume	FFSv2

https://edk2-docs.gitbook.io/edk-ii-build-specification/2_design_discussion/22_uefipi_firmware_images

ファイルの種類

- Fileは実行可能ファイルを含む
 - 実行可能ファイルのフォーマットはTEかPE

Structure			
Name	Action	Type	Subtype
▼AMI Aptio capsule		Capsule	Aptio signed
▼UEFI image		Image	UEFI
Padding		Padding	Non-empty
>EfiFirmwareFileSystem2Guid		Volume	FFSv2
Padding		Padding	Empty (0xFF)
>7DCCF422-45F6-4951-A557-CC421DA31599		Volume	FFSv2
▼4F1C52D3-D824-4D2A-A2F0-EC40C23C5916		Volume	FFSv2
>414D94AD-998D-47D2-BFCD-4E882241DE32		File	Freeform
>7B9A0A12-42F8-4D4C-82B6-32F0CA1953F4		File	Freeform
▼9E21FD93-9C72-4C15-8C4B-E77F1DB2D792		File	Volume image
▼LzmaCustomDecompressGuid		Section	GUID defined
Raw section		Section	Raw
▼Volume image section		Section	Volume image
▼5C60F367-A505-419A-859E-2A4FF6CA6FE5		Volume	FFSv2
>AprioriDxe		File	Freeform
>RomLayoutDxe		File	DXE driver
>DxeCore		File	DXE core
>Bds		File	DXE driver

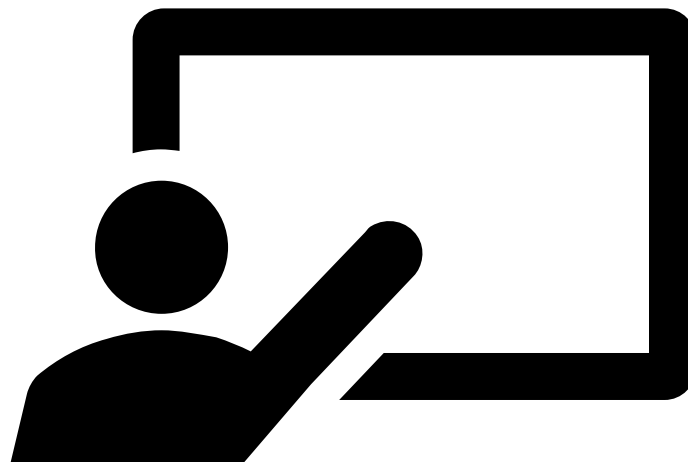
実行可能ファイルの種類

- 実行可能ファイル (TE/PE) はいくつかの種類に分けられる

- 例

タイプ	概要	例
Application	StartImage()で実行され、実行が終了するとアンロードされる	<ul style="list-style-type: none">• Shell.efi• SmmReset• SmmAccessSub
DXE Boot Driver	自動的、またはLoadコマンドで実行され、Runtimeフェーズまでメモリに居る	<ul style="list-style-type: none">• ほぼ全部のDXE Driver• SmmInterfaceBase• Ntfs
DXE Runtime Driver	自動的、またはLoadコマンドで実行され、OSが起動したあともメモリに居る	<ul style="list-style-type: none">• CRZEFI.efi• SecDxe
DXE SMM Driver	自動的に実行され、SMMとしてSMRAMにロードされ、OSが起動した後もメモリにいる	<ul style="list-style-type: none">• NvmeSmm

https://edk2-docs.gitbook.io/edk-ii-inf-specification/appendix_f_module_types

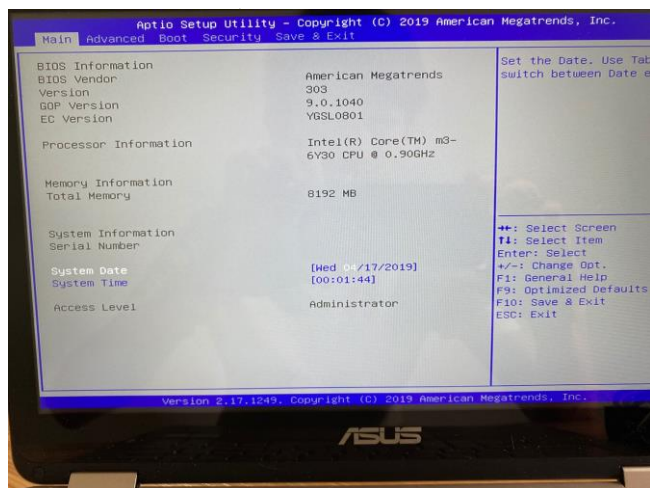


Demo: 実行ファイルタイプの確認

CFF Explorerを用いて演習で使った実行可能ファイルの種類をPEヘッダーから確認する

UEFIの実装例

- EDK2 – Reference Implementation & SDK
 - Open Virtual Machine Firmware (OVMF) – オープンソースUEFI BIOS。QEMUに使用される
- 各OEMのBIOS
 - BIOSベンダー(eg, AMI, Insyde)開発のコード+OEM独自モジュール



Item	Value
OS Name	Microsoft Windows 10 Enterprise
Version	10.0.18363 Build 18363
Other OS Description	Not Available
OS Manufacturer	Microsoft Corporation
System Name	DESKTOP-HLKR4S8
System Manufacturer	ASUSTeK COMPUTER INC.
System Model	UX360CA
System Type	x64-based PC
System SKU	ASUS-NotebookSKU
Processor	Intel(R) Core(TM) m3-6Y30 CPU @ 0.90GHz, 1512 Mhz, 2 C...
BIOS Version/Date	American Megatrends Inc. UX360CA.303, 4/17/2019

- BIOSベンダーを使わない場合もある、例: Microsoft、Dell、Apple
 - この場合もEDK2をベースが多い

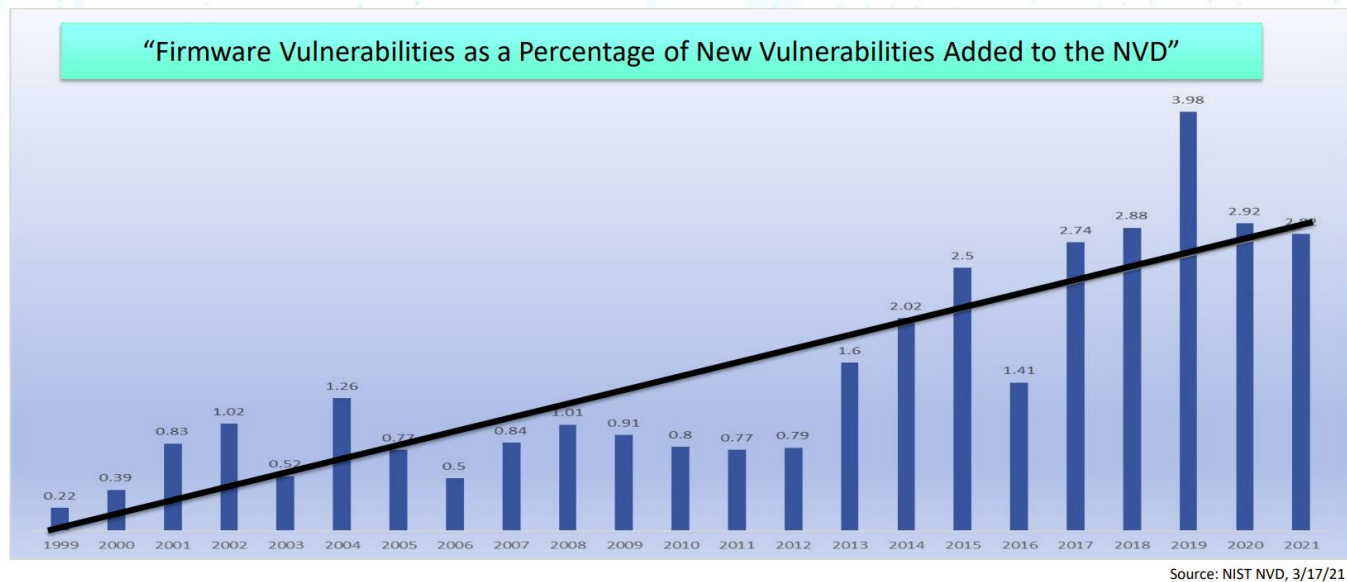
<https://support.apple.com/en-ca/guide/security/seced055bcf6/web>

UEFIの実装例 - VMware Workstation

- 多少EDK2に基づいた実装
 - C:\Program Files (x86)\VMware\VMware Workstation\x64\EFI64.ROM
 - /usr/lib/vmware/roms/EFI64.ROM
 - 最小限の実装
 - 2MB
 - No SMM
- VMXファイルで使用するUEFIファイルを指定できる
 - efi64.filename = "MY.ROM"
 - UEFIToolでUEFIファイルの内容を変更できる
 - 📁 UEFIマルウェアやUEFIアンチウイルスとかを書いて遊べる

ファームウェアセキュリティ

- 攻撃や問題が近年頻発



- 80%の組織が過去2年間にファームウェアに対する攻撃を確認
 - By Microsoft

<https://www.microsoft.com/security/blog/2021/03/30/new-security-signals-study-shows-firmware-attacks-on-the-rise-heres-how-microsoft-is-working-to-help-eliminate-this-entire-class-of-threats/>

https://static.rainfocus.com/rsac/us21/sess/1602603692582001zuMc/finalwebsite/2021_US21_Tech-W13_01_DHS-CISA-Strategy-to-Fix-Vulnerabilities-Below-the-OS-Among-Worst-Offenders_1620749389851001CH5E.pdf

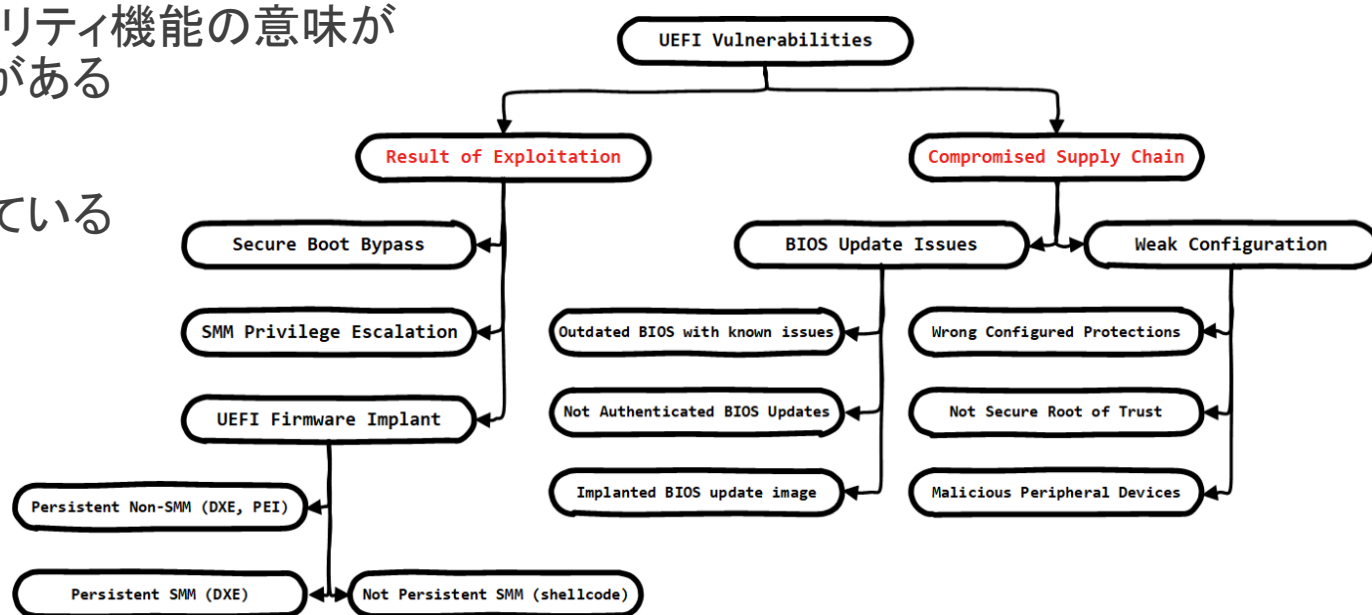
UEFI BIOSセキュリティ

- 広範で複雑 (UEFI BIOSに限定しても)

- 複数の技術が関連する
- 理解しているエンジニアが少ない

- 影響大

- ほかのセキュリティ機能の意味がなくなる場合がある
- 修正が困難
- 広く使用されている



<https://medium.com/firmware-threat-hunting/uefi-vulnerabilities-classification-4897596e60af>

UEFIのサプライチェーン

- 長い&遅い
- 例
 - OEM: 脆弱性確認 -> アップデート開発 -> テスト -> リリース(3か月)
 - IT管理者: 脆弱性情報の確認 -> アップデートの計画 -> アップデート適用
- 脆弱性がBIOSベンダーやEDK2など、複数OEMで共有されたコードの場合、さらに遅くなる

UEFI Hackz

UEFIの“悪用”

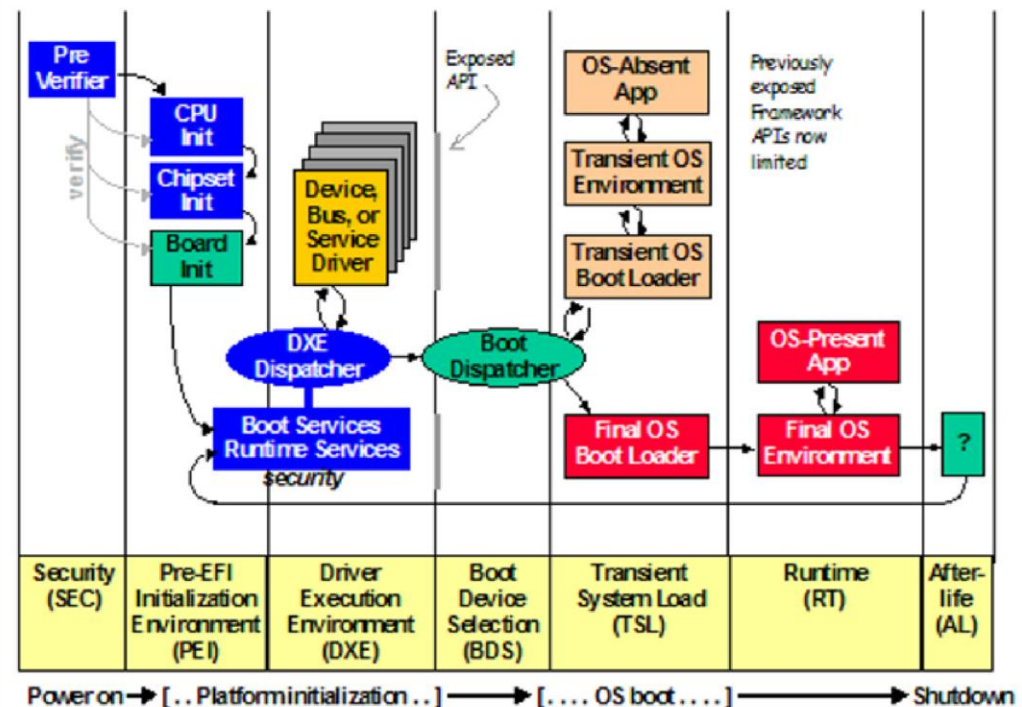
- UEFIモジュールを使用したゲームセキュリティの侵害
 - [CRZAIMBOT](#), ほかの例は[UnKnoWnCheaTs](#)で
- UEFIモジュールとして実装されたハッキングツールも多数
 - リバースエンジニアリングの支援ツール
 - [EfiGuard](#), [DmaBackdoorHv](#), [negativespoofer](#), [umap](#), [efi-memory](#)
- なぜ？

UEFIが悪用される理由

- カーネルモードで実行される
- OSレベルのセキュリティポリシーをバイパスできる
 - Code Signing on Windows
- 検知されにくい
 - アンチチートやアンチウイルスによるOSフェーズでのモニタリングが始まる前に実行可能
 - カーネルモジュールの場合のような、OSから見える管理用データ構造が存在しない(OSは基本的にUEFIモジュールの存在について知らない)
- 開発が比較的容易

ブートシーケンス(意味)

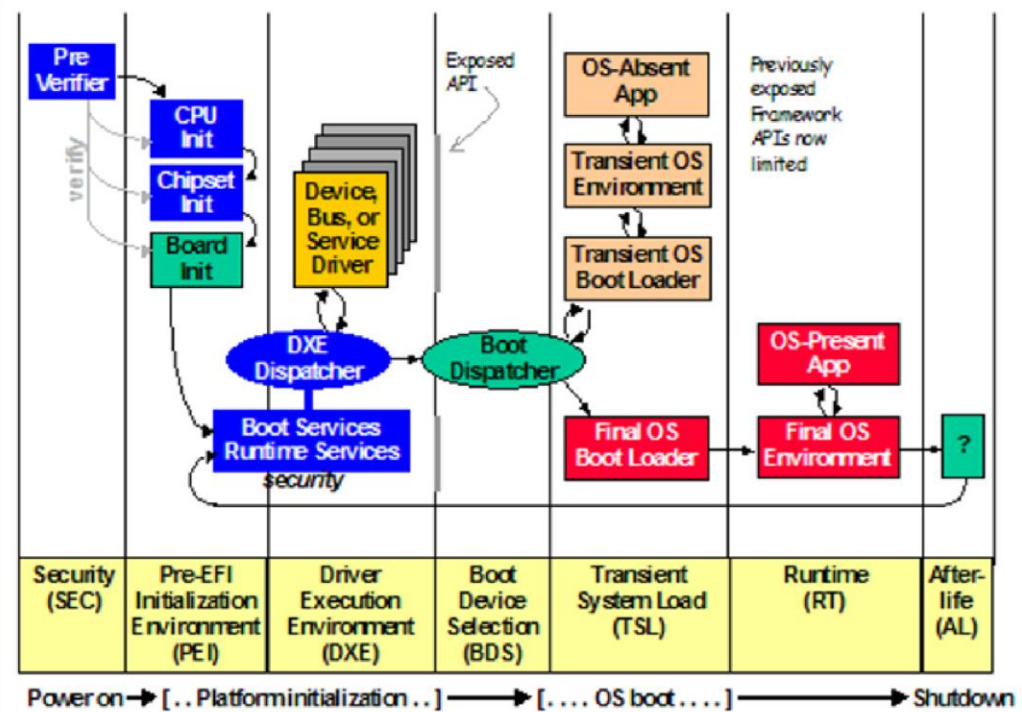
- PEI - ハードウェア初期化
- DXE - RTまでの実行環境の構築
- BDS - ブートメニューのリストと選択
- TSL - ブートメニューから選択された任意のUEFIアプリの実行。主にブートローダーが実行される
- RT - UEFI実行環境が破棄されOSに環境管理が移行した後



https://edk2-docs.gitbook.io/edk-ii-build-specification/2_design_discussion/23_boot_sequence

ブートシーケンス(セキュリティ)

- ~BDS
 - SPIフラッシュ内のモジュールのみ実行される
 - 例: DXE drivers, SMM modules
 - ☞ Trusted
- TSL~
 - 他のストレージ内のモジュールも実行される
 - 例: bootmgfw.efi, shell.efi, myhackz.efi
 - ☞ Untrusted



https://edk2-docs.gitbook.io/edk-ii-build-specification/2_design_discussion/23_boot_sequence

Driver Execution Environment (DXE)

- シングルスレッド
- すべてのモジュールを含む単一のアドレススペース
 - SMMを除く。SMMは独自のアドレススペースを持つ
- すべてのコードがカーネルモードで動作
 - SMMを除く。SMMは実質さらに高い特権レベルで動作
 - macOS on Intelはユーザーモードを使用

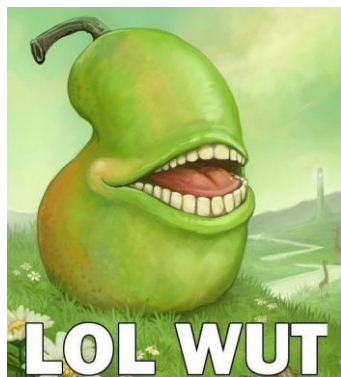
<https://support.apple.com/en-ca/guide/security/secc80b0cd0c/web>

Driver Execution Environment (DXE)

- Long-mode (64bitアドレッシング) がすでに有効
 - Not 16bit real-mode
- Identity mapped paging
 - 仮想アドレスが同じ物理アドレスにマップされている
 - 例: VA 0x123000 は PA 0x123000 に変換される
 - ページングなし。例外は基本的に即フリーズ
 - 基本すべてのページがRWE 😊

セキュリティ機能の使用状況

セキュリティ機能	状況
Control flow guard	Intel CET-SSは対応, Intel CET-IBTとコンパイラベースのCFI (/guard:cf, -fsanitize=cfi) は未対応
NULLポインターアクセスの禁止	EDK2は対応しているが、しばしば未使用
DEP / NX	EDK2は対応しているが、しばしば未使用
スタックカナリー	未対応
ASLR	未対応



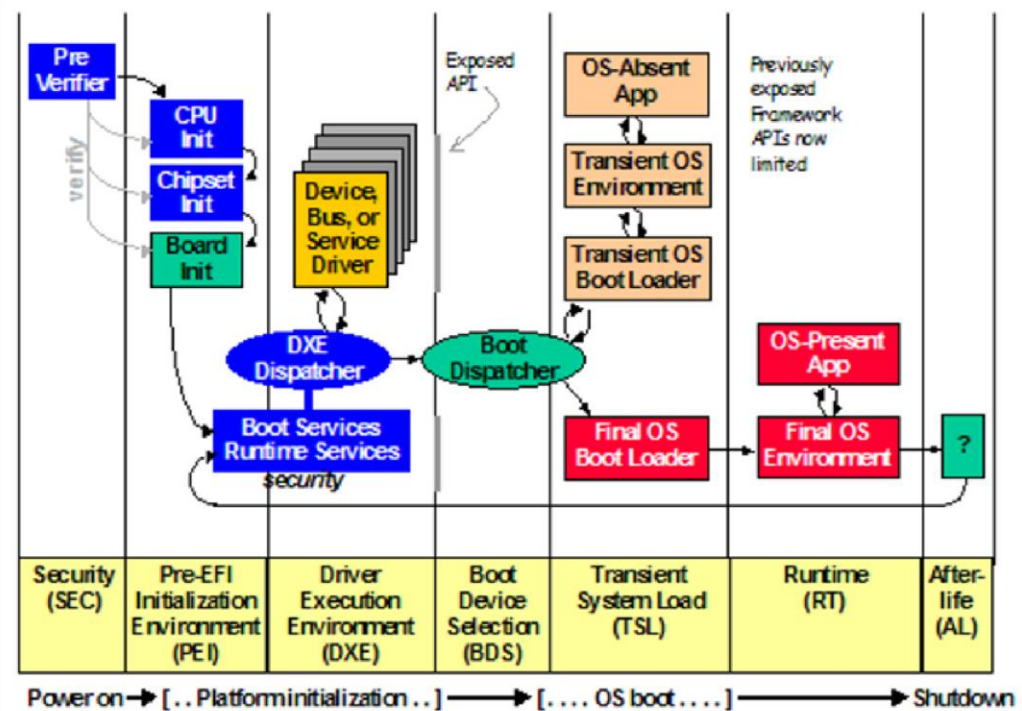
<https://github.com/jyao1/SecurityEx/blob/master/Summary.md>

UEFIモジュールとOS

- UEFIモジュールは基本:
 - OSの読み込みのために存在
 - OSが起動(RTに移行)する段階でアンロードされる
- ハック&マルウェアはOS段階で何かをしたい
 - 方法1: DXE Runtime Driverとして実装する(課題1)
 - 方法2: OSローダーとOSカーネルにパッチを当てて生き残る
 - [rainbow](#), [Voyager](#)など
 - 方法3: OSに呼び出してもらう(課題2,4)

Boot-time vs Run-time

- DXE ~ TLSがBoot-time
 - Boot DriverとRuntime Driverの両方が実行可能
 - すべてのUEFI APIが使用可能
 - すべてのアドレスが使用可能
- RT以降はRun-time
 - Runtime Driverだけ生き残る
 - Boot driverはアンロードされている
 - ごく一部のUEFI APIしか使用できない
 - ごく一部のアドレス (Runtime code and pool)のみ使用可能
 - あとはOSが管理



DXE Runtime Driver

- Runtime Driverとしてコンパイル、実行することでOS起動後も動作

タイプ	概要	例
Application	StartImage()で実行され、実行が終了するとアンロードされる	<ul style="list-style-type: none">• Shell.efi• SmmReset• SmmAccessSub
DXE Boot Driver	自動的、またはLoadコマンドで実行され、Runtimeフェーズまでメモリに居る	<ul style="list-style-type: none">• ほぼ全部のDXE Driver• SmmInterfaceBase• Ntfs
DXE Runtime Driver	自動的、またはLoadコマンドで実行され、OSが起動したあともメモリに居る	<ul style="list-style-type: none">• CRZEFI.efi• SecDxe
DXE SMM Driver	自動的に実行され、SMMとしてSMRAMにロードされ、OSが起動した後もメモリにいる	<ul style="list-style-type: none">• NvmeSmm

```
36  %efi: %.so
37  → objcopy -j .text -j .sdata -j .data -j .dynamic \
38  →      -j .dynsym -j .rel -j .rela -j .reloc \
39  →      --target=efi-rtdrv-$(ARCH) $^ $@
```

UEFIモジュールとOS #2

- ハッキングUEFIモジュールはOS実行環境で呼び出されたい
 - I.e, ただメモリーに居るだけでは意味がない
 - スレッドやタイマーのようなものを登録しておくこともできない
- Runtime Servicesをフックする
 - CRZEFIを含む多くのハッキングUEFIモジュールが実装する手段

Boot & Runtime Services

- UEFIプラットフォームは、2つの主要なAPIセットをモジュールに提供
 - Boot Services (BS)
 - DXE～TSLまで使用可能。メモリー確保などessentialなサービスを提供
 - Runtime Services (RT)
 - DXE～恒久的に使用可能。UEFI変数へのアクセス、BIOSアップデートのスケジュール、シャットダウンなど14APIのみ

```
///  
/// Cache pointer to the EFI Boot Services Table  
///  
extern EFI_BOOT_SERVICES *gBS;
```

```
///  
/// Cached copy of the EFI Runtime Services Table  
///  
extern EFI_RUNTIME_SERVICES *gRT;
```

Runtime Services

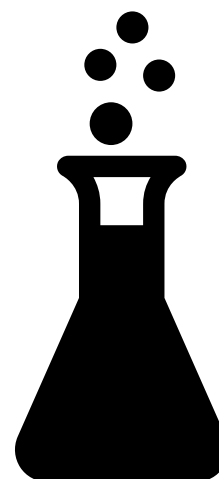
- Runtime ServicesはOSから呼び出される

```
1: kd> k
# Child-SP          RetAddr           Call Site
00 fffff9007`ee5fa0a8 ffffff801`570db27f 0xfffff801`5a45c000
01 fffff9007`ee5fa0b0 ffffff801`570c40ce nt!HalEfiGetEnvironmentVariable+0x53
02 fffff9007`ee5fa0f0 ffffff801`574b3203 nt!HalGetEnvironmentVariableEx+0xf074e
03 fffff9007`ee5fa1e0 ffffff801`574b26e0 nt!IopGetEnvironmentVariableHal+0x23
04 fffff9007`ee5fa220 ffffff801`57569071 nt!IoGetEnvironmentVariableEx+0x94
05 fffff9007`ee5fa340 ffffff801`57444b98 nt!ExpGetFirmwareEnvironmentVariable+0x8d
06 fffff9007`ee5fa390 ffffff801`570247b5 nt!NtQuerySystemEnvironmentValueEx+0x13d7a8
07 fffff9007`ee5fa450 00007fff`56d4f804 nt!KiSystemServiceCopyEnd+0x25
08 00000019`5967d648 00007fff`5670548f ntdll!NtQuerySystemEnvironmentValueEx+0x14
09 00000019`5967d650 00007fff`43912156 KERNEL32!GetFirmwareEnvironmentVariableExW+0x9f
```

- Runtime ServicesはBoot-timeでもアクセス可能 &&
- すべてのメモリーは書き込み可能 &&
- UEFIには実行時のIntegrity Checkが存在しない
- 📌 Runtime Servicesをフックして自分のモジュールが実行されるようにする

Lab: CRZEFI

- CRZEFIの実装を確認する
 - RTフックを実装している関数を見つける
 - どのRTがフックされているかを見つける
 - MakefileからRuntime Driverとしてコンパイルされるためのフラグを見つける
- すでに終わっているひとは
 - <https://github.com/tandasat/UefiVarMonitor>をチェック。Rust実装もあるよ！



CRZEFIの概要

■ 処理

- Efi_main()
 - SetServicePointer()
 - RT->SetVariableがHookedSetVariableに置き換えられる
- 誰かがSetVariable()を呼び出すと
 - HookedSetVariable()
 - mySetVariable()
 - RunCommand()
 - メモリー操作のバックドアコマンド

■ ビルド環境

- GNU-EFI – UEFIモジュール開発のための軽量なSDKを使用
 - EDK2がより標準的なビルド環境
- どちらを使ってもコードの見た目はだいたい同じ

UEFIモジュールの利点

- カーネルモードで実行される
- OSレベルのセキュリティポリシーをバイパスできる
 - ㊦ もしWindowsドライバだったら署名するか、ほかの脆弱性をつく必要がある
- 検知されにくい
 - アンチチートやアンチウイルスによるOSフェーズでのモニタリングが始まる前に実行可能
 - ㊦ もしWindowsドライバだったら、OS APIを使ってロードする必要があり、セキュリティソフトウェアが監視している可能性が高い
 - カーネルモジュールの場合のような、OSから見える管理用データ構造が存在しない(OSは基本的にUEFIモジュールの存在について知らない)
 - ㊦ もしWindowsドライバだったら、DRIVER_OBJECTが作られたり、OS管理内のメモリーにコードがロードされるのでスキャンしやすい
- 開発が比較的容易

潜在的な対策

- Secure bootが有効でないとゲームを起動しないようにする？
 - 署名されていないUEFIモジュールがロードできなくなる
 - Secure bootの脅威モデルはデバイス所有者を脅威に含めていない
 - 物理アクセス可能の場合、PK/KEK/DBを変更し、自己署名を許可できる
 - Secure bootのステータス報告を詐称できる
 - Secure boot無効で起動。GetVariable()をフック。以降、有効であると詐称する
- TPMを使用し、異常なブートシーケンスの場合ゲームを起動しないようにする？
 - TPMは3rd partyモジュールの実行をPCR[2]に記録する
 - たとえばゲームインストール時の値と違ったらBanなど
 - TPMを有効にしていないシステムも多い
- メモリスキャンニング & パターンマッチ？

https://twitter.com/d_olex/status/1377475132482969601?s=20

https://tianocore-docs.github.io/edk2-TrustedBootChain/release-1.00/3_TCG_Trusted_Boot_Chain_in_EDKII.html

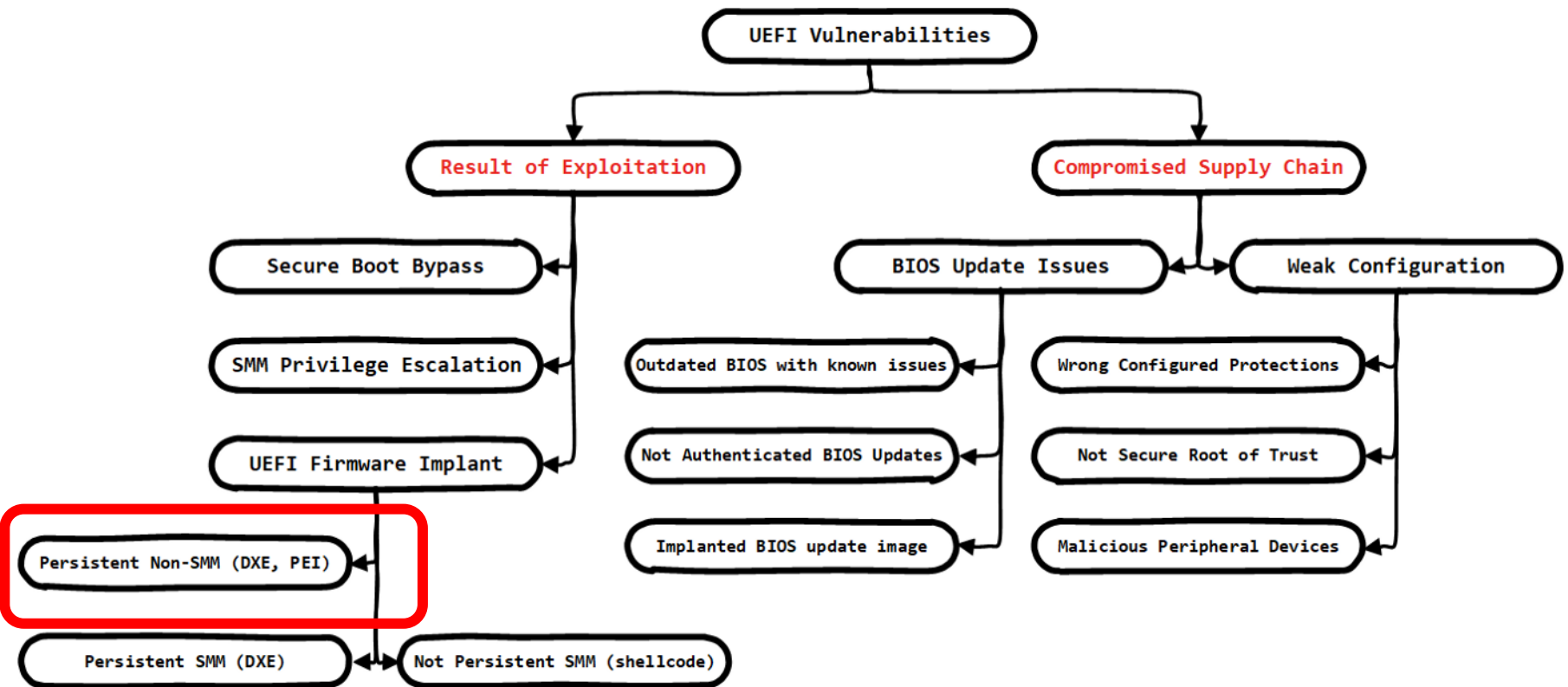
マルウェアによる UEFIモジュール利用の可能性

- インストールが障壁
 - ハッキングツールの場合、デバイス所有者が協力的
 - セキュアブートの無効化、USB経由での起動などなんでもアリ
 - マルウェアの場合、デバイス所有者が非協力的
- 攻撃者がリモートの場合
 - 脆弱性についてSPIフラッシュにインストール(課題4)
 - AVからの検知はやや困難
 - Secure bootが無効なら、OSローダーやブートエントリーの書き換え(EFILock)
 - AVはこの書き換えの処理を簡単に検知できる
- 攻撃者が物理アクセス可能な場合
 - SPIフラッシュプログラマーでSPIフラッシュにインストール

<https://twitter.com/ESETresearch/status/1275770256389222400>

UEFIマルウェアと リバースエンジニア リング

分類



ソフトウェア リバースエンジニアリング (RE) 101

- ソフトウェアの動作を解析し、その動作原理や仕様を調査する行為
- 目的例: 利用の安全性の調査、マルウェアの動作の理解
- 方法例: 動的解析、静的解析
 - 動的解析 – ソフトウェアを動作させて挙動を監視、記録、分析する
 - Process Monitor, API Monitor, Debuggers
 - 静的解析 – 逆アセンブルし実装を分析する
 - Ghidra, IDA Pro, Binary Ninja

Why: UEFIモジュールRE

- UEFIモジュールもただのソフトウェア
 - セキュリティ上の問題がある場合がある
 - UEFIモジュールとして実装されたマルウェアや、Potentially Unwanted Application (PUA) が存在する
 - 通常のソフトウェアと同じくREが必要となる
- 例
 - 課題2, 4 – UEFIモジュールとして実装されたマルウェア
 - 課題3 – 脆弱性を含むUEFIモジュール
 - Absolute Software (aka, Computrace, Lojack) はUEFIモジュールとして実装された盗難対策ソフトウェアだが、性質上Rootkitと評されることがある

<https://www.blackhat.com/presentations/bh-usa-09/ORTEGA/BHUSA09-Ortega-DeactivateRootkit-PAPER.pdf>

What: UEFIモジュールRE

■対象の入手

- SPIフラッシュからソフトウェア経由で
 - 例: CHIPSEC
- SPIフラッシュからSPIフラッシュプログラマー経由で
- BIOSアップデートファイルから
 - 直接UEFIToolで開けることがある(課題3)
 - BIOSUtilitiesで展開できる場合も多い
- VirtusTotalから
 - ハッシュ値がわかっている場合のみ(課題2, 4)

■脆弱性を探す場合、SMMモジュールを狙う

Structure			
Name	Action	Type	Subtype
> PiSmmCore		File	SMM core
> FlashDriverSmm		File	SMM module
> NvramSmm		File	SMM module
> NvramSmi		File	SMM module
> CpuIo2Smm		File	SMM module

<https://github.com/chipsec/chipsec>

<https://github.com/platomav/BIOSUtilities>

<https://standa-note.blogspot.com/2021/04/reverse-engineering-absolute-uefi.html>

UEFIファイルからモジュール抽出

- UEFIToolを使ってPE32 imageセクションを探し、抽出する
 - UEFIモジュールはGUIDで識別される
 - Human-friendlyな名前(DxeCoreなど)はオプション

UEFITool NE alpha 58 (Nov 7 2020) - UX360CA-AS.303

File Action View Help

Structure			
Name	Action	Type	Subtype
✓AMI Aptio capsule		Capsule	Aptio signed
✓UEFI image		Image	UEFI
Padding		Padding	Non-empty
>EfiFirmwareFileSystem2Guid		Volume	FFSv2
Padding		Padding	Empty (0xFF)
>7DCCF422-45F6-4951-A557-CC421DA31599		Volume	FFSv2
✓4F1C52D3-D824-4D2A-A2F0-EC40C23C5916		Volume	FFSv2
>414D94AD-998D-47D2-BFCD-4E882241DE32		File	Freeform
>7B9A0A12-42F8-4D4C-82B6-32F0CA1953F4		File	Freeform
✓9E21FD93-9C72-4C15-8C4B-E77F1DB2D792		File	Volume image
✓LzmaCustomDecompressGuid		Section	GUID defined
Raw section		Section	Raw
✓Volume image section		Section	Volume image
✓5C60F367-A505-419A-859E-2A4FF6CA6FE5		Volume	FFSv2
>AprioriDxe		File	Freeform
✓RomLayoutDxe		File	DXE driver
DXE dependency section		Section	DXE dependency
PE32 image section		Section	PE32 image
UI section		Section	UI
Version section		Section	Version
✓DxeCore		File	DXE core
✓LzmaCustomDecompressGuid		Section	GUID defined
PE32 image section		Section	PE32 image
UI section		Section	UI
Version section		Section	Version

Hex view...	Ctrl+D
Body hex view...	Ctrl+Shift+D
Extract as is...	Ctrl+E
Extract body...	Ctrl+Shift+E
Extract body uncompressed...	Ctrl+Alt+E

<https://github.com/LongSoft/UEFITool>

How: UEFIモジュールRE

- 静的解析のみ
 - 実用レベルの動的解析システム & ツールはない
 - VM内動作させ、VM付属のデバッガーを使える場合はある
 - 難読化ツールもまだ存在しない
- 既存の逆アセンブルツールをプラグインとともに使用
 - Ghidra + efiSeek, IDA Pro + efiXplorer, Binary Ninja + bn-uefi-helper

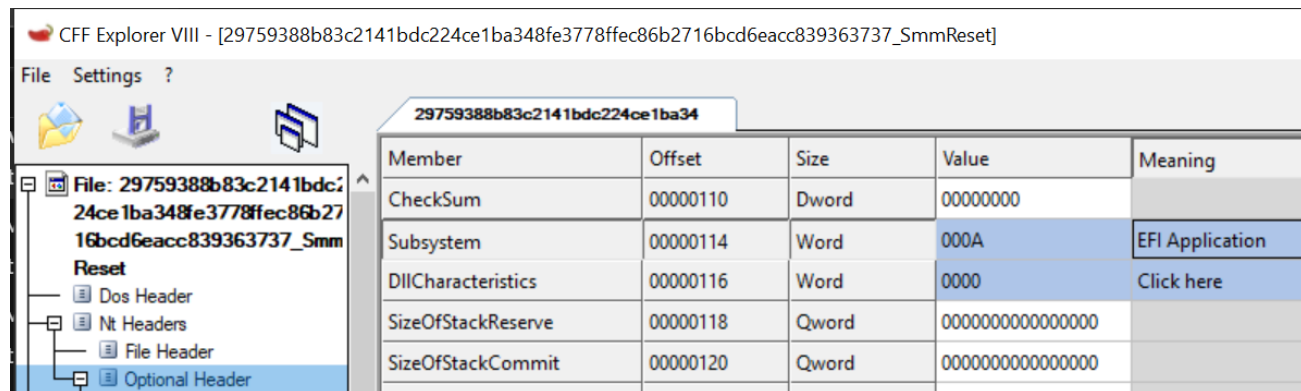
```
Decompile: entry - (c7c3e039700bc6072f84ff99ecb22557e460dcd2214539938a6a0ef73b9c...
1
2 // DISPLAY WARNING: Type casts are NOT being printed
3
4 undefined8 entry(undefined8 param_1,longlong param_2)
5
6 {
7     undefined local_18 [24];
8
9     FUN_000102b0(param_1,param_2);
10    DAT_00010668 = 0;
11    (*(DAT_00010680 + 0x170))(0x200,0x10,FUN_00010330,0,&DAT_00010260,local_18)
12    ;
13    return 0;
14 }
15
```

```
Decompile: ModuleEntryPoint - (c7c3e039700bc6072f84ff99ecb22557e460dcd22145399...
1
2 // DISPLAY WARNING: Type casts are NOT being printed
3
4 EFI_STATUS
5 ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)
6
7 {
8     EFI_EVENT local_18 [3];
9
10    FUN_800002b0(ImageHandle,SystemTable);
11    DAT_80000668 = 0;
12    (*gBS_12->CreateEventEx)
13        (0x200,0x10,FUN_80000330,NULL,&EFI_EVENT_READY_TO_BOOT_GUID,
14         local_18);
15    return 0;
16 }
17
```

UEFIモジュール表層解析

- DXE以降実行されるモジュールはすべてPE
- IMAGE_OPTIONAL_HEADER.SubSystemが特定の値

IMAGE_SUBSYSTEM_EFI_APPLICATION 10	Extensible Firmware Interface (EFI) application.
IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11	EFI driver with boot services.
IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12	EFI driver with run-time services.



https://docs.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-image_optional_header32

UEFIモジュール表層解析

- セクション情報やAddressOfEntryPointはValid
- インポートやエクスポートは存在しない
 - APIの解決はすべて動的に行う(後述)

UEFIモジュールRE 101

- エントリーポイントはいずれか
 - プログラマーが書いたコード
 - ライブラリー生成のコード
- ライブラリーコードの例
 - 左: GNU-EFI (unoptimized)
 - 右: EDK2 (unoptimized)

```
Decompile: ModuleEntryPoint - (memory.efi)
1
2 // DISPLAY WARNING: Type casts are NOT being printed
3
4 EFI_STATUS
5 ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)
6
7 {
8     EFI_STATUS EVar1;
9     undefined4 in_R8D;
10    undefined4 in_R9D;
11    EFI_HANDLE in_stack_00000018;
12
13    FUN_80009af0();
14    EVar1 = FUN_80003bd2(SystemTable,ImageHandle,in_R8D,in_R9D,in_stack_00000018
15    );
16    return EVar1;
17 }
18
```

```
Decompile: ModuleEntryPoint - (7ea33696c91761e95697549e0b0f84db2cf4033216cd16c...
1
2 // DISPLAY WARNING: Type casts are NOT being printed
3
4 EFI_STATUS
5 ModuleEntryPoint(EFI_HANDLE ImageHandle,EFI_SYSTEM_TABLE *SystemTable)
6
7 {
8     char cVar1;
9     EFI_STATUS EVar2;
10    EFI_STATUS extraout_RAX;
11    void *local_10 [2];
12
13    if ((DAT_80050460 == 0) || (DAT_80050460 <= (SystemTable->Hdr).Revision)) {
14        FUN_8000040c(ImageHandle,SystemTable);
15        if (DAT_8005045f != '\0') {
16            EVar2 = (*gBS_174->HandleProtocol)
17                (ImageHandle,&EFI_LOADED_IMAGE_PROTOCOL_GUID,
18                 local_10);
19            cVar1 = FUN_800019dc();
20            if ((cVar1 != '\0') && (EVar2 < 0)) {
21                cVar1 = FUN_800019dc();
22                if ((cVar1 != '\0') && (cVar1 = FUN_800019e0(), cVar1 != '\0'))
23                {
24                    FUN_800019b4();
25                }
26                FUN_800019cc();
27            }
28            *(local_10[0] + 0x58) = &LAB_800002a0;
29        }
30        FUN_800005bc(ImageHandle,SystemTable);
31        EVar2 = extraout_RAX;
32        if (extraout_RAX < 0) {
33            FUN_800005b0();
34        }
35    }
36    else {
37        EVar2 = 0x8000000000000019;
```

UEFIモジュールRE 101

- エントリーポイントはEFI_SYSTEM_TABLE*を受け取る
 - EFI_BOOT_SERVICES* とEFI_RUNTIME_SERVICES* を含む
- 以下のグローバル変数が初期化される

名前	内容
gST	EFI_SYSTEM_TABLE*のコピー
gBS	Boot Services
gRT	Runtime Services

```
Decompile: ModuleEntryPoint - (29759388b83c2141bdc224ce1ba348fe3778ff...)
1
2 // DISPLAY WARNING: Type casts are NOT being printed
3
4 EFI_STATUS
5 ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
6
7 {
8     undefined local_res8 [8];
9     undefined local_res10 [8];
10    UINTN local_res18 [2];
11
12    gBS_22 = SystemTable->BootServices;
13    gRS_23 = SystemTable->RuntimeServices;
14    local_res8[0] = 0;
15    local_res18[0] = 1;
16    gImageHandle_20 = ImageHandle;
17    gST_21 = SystemTable;
```

UEFIモジュールRE 101

- BSは基本サービスを提供
 - メモリー管理: `AllocatePages()`, `FreePages()` etc
 - コールバック登録: `CreateEvent()`, `SignalEvent()`, etc
 - イメージ実行: `LoadImage()`, `StartImage()`, etc
 - プロトコル解決: `OpenProtocol()`, `LocateProtocol()`, etc

<https://github.com/tianocore/edk2/blob/3c13938079886e0e49031ab29a4f1ed7a4ceab68/MdePkg/Include/Uefi/UefiSpec.h>

UEFIモジュールRE 101

- その他のAPIやデータは上記プロトコルAPIを使用して動的に解決
- GUIDが入力、対応する構造体(関数ポインターを含む)が出力
 - 例: 現在のモジュールに関する詳細情報の取得(CRZEFI)

```
// Get handle to this image
EFI_LOADED_IMAGE* LoadedImage = NULL;
EFI_STATUS status = BS->OpenProtocol(ImageHandle, &LoadedImageProtocol,
    (void**)&LoadedImage, ...);

// Set our image unload routine
LoadedImage->Unload = (EFI_IMAGE_UNLOAD)efi_unload;
```

UEFIモジュールRE 101

- その他のAPIやデータは上記プロトコルAPIを使用して動的に解決
 - 例: ファイルアクセス (SmmAccessSub) 1/2

```
57     efiStatus = (*gBS->HandleProtocol)
58               (*ppvVar2, &EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID,
59               &simpleFileSystemProtocol);
60
61     if ((-1 < efiStatus) &&
62         (efiStatus2 = (**)(simpleFileSystemProtocol + 8))
63         (simpleFileSystemProtocol, &fileProtocol), -1 < efiStatus2)) {
64         gFileProtocol = fileProtocol;
65         fileProtocol = (**)(fileProtocol + 8);
66     }
```

```
struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    ///
    /// The version of the EFI_SIMPLE_FILE_SYSTEM_PROTOCOL. The version
    /// specified by this specification is 0x00010000. All future revisions
    /// must be backwards compatible.
    ///
    UINT64                                     Revision;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
};
```

```
typedef
EFI_STATUS
(EFIAPI *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME)(
    IN EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *This,
    OUT EFI_FILE_PROTOCOL **Root
);
```

13.4 Simple File System Protocol

The Simple File System protocol allows code running in the EFI boot services environment to obtain file-based access to a device. `EFI_SIMPLE_FILE_SYSTEM_PROTOCOL` is used to open a device volume and return an `EFI_FILE_PROTOCOL` that provides interfaces to access files on a device volume.

EFI_SIMPLE_FILE_SYSTEM_PROTOCOL

Summary

Provides a minimal interface for file-type access to a device.

GUID

```
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID \
{0x0964e5b2, 0x6459, 0x11d2, \
{0x8e, 0x39, 0x00, 0xa0, 0xc9, 0x69, 0x72, 0x3b}}
```

Revision Number

```
#define EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_REVISION 0x00010000
```

Protocol Interface Structure

```
typedef struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
    UINT64                                     Revision;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
} EFI_SIMPLE_FILE_SYSTEM_PROTOCOL;
```

<https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdePkg/Include/Protocol/SimpleFileSystem.h#L530>

UEFIモジュールRE 101

- その他のAPIやデータは上記プロトコルAPIを使用して動的に解決
 - 例: ファイルアクセス (SmmAccessSub) 2/2

```
26     StrCat(Buffer, u_ \ProgramData\Microsoft\Windows\S_800036f0);
27     StrCat(filePath, u_ IntelUpdate.exe_80000290);
28     efiStatus = (*(gFileProtocol + 8))
29                 (gFileProtocol, &fileProtocol, filePath, 0x8000000000000003, 0);
30     if (-1 < efiStatus) {
31         efiStatus = (*(fileProtocol + 0x28))(fileProtocol, local_res18, &DAT_800002b0);
32         if (-1 < efiStatus) {
33             efiStatus = (*(fileProtocol + 0x10))(fileProtocol);
```

```
struct _EFI_FILE_PROTOCOL {
    ///
    /// The version of the EFI_FILE_PROTOCOL interface. The version specified
    /// by this specification is EFI_FILE_PROTOCOL_LATEST_REVISION.
    /// Future versions are required to be backward compatible to version 1.0.
    ///
    UINT64      Revision;
    EFI_FILE_OPEN      Open;
    EFI_FILE_CLOSE     Close;          // +0x10
    EFI_FILE_DELETE     Delete;
    EFI_FILE_READ       Read;          // +0x20
    EFI_FILE_WRITE      Write;
```

<https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdePkg/Include/Protocol/SimpleFileSystem.h#L530>

UEFIモジュールRE 101

- Eventに対応してコールバックを実行可能
- カスタムイベントに加え、UEFIがいくつかのイベントを登録、実行する
 - 例: ブートローダーが起動する直前にコード実行 (SmmInterfaceBase)

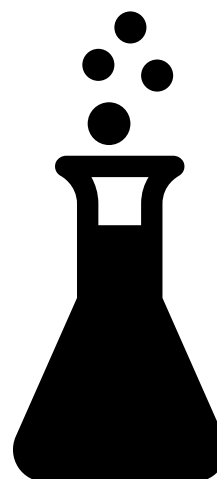
```
4  EFI_STATUS
5  ModuleEntryPoint(EFI_HANDLE ImageHandle, EFI_SYSTEM_TABLE *SystemTable)
6
7  {
8      EFI_EVENT local_18 [3];
9
10     FUN_800002b0(ImageHandle, SystemTable);
11     DAT_80000668 = 0;
12     (*gBS_12->CreateEventEx)
13         (0x200, 0x10, HandleReadyToBootEvent, NULL,
14          &EFI_EVENT_READY_TO_BOOT_GUID, local_18);
15     return 0;
16 }
```

EFI_EVENT_GROUP_READY_TO_BOOT

This event group is notified by the system right before notifying **EFI_EVENT_GROUP_AFTER_READY_TO_BOOT** event group when the Boot Manager is about to load and execute a boot option. The event group presents the last chance to modify device or system configuration prior to passing control to a boot option.

Lab: MosaicRegressor

- SmmAccessSubの実装のレビュー
 - FUN_80003aa8の実装を読みUEFIモジュールからファイルにアクセスするコードを確認する
 - すでにIntelUpdate.exeがあったらどうなるか確認する




AVによるMosaicRegressorの検出


- IntelUpdate.exeの書き込みは検知、ブロック不可
- IntelUpdate.exeの実行は検知、ブロック可能
 - 事実上の無効化
- UEFIモジュールの検出は可能、除去とブロックは不可
 - AVソフトウェアはSPIフラッシュを読み込みパースできる
 - CrowdStrike Falcon – Firmware Analysis
 - Microsoft Defender APT – UEFI Scanner
 - ESET, Kaspersky, etc
- Note: BIOSアップデートが走るとMosaicRegressorは上書きされてしまうように見える

Verified Boot - BIOS改竄に対する保護機能の一部

■ Secure boot

- DXE以降に実行されるモジュール(OEM Boot Block, OBB)の署名またはハッシュ値をチェック
- Allow Listされたモジュール以外実行できなくする
-  MosaicRegressorは恐らく完全に無効化される

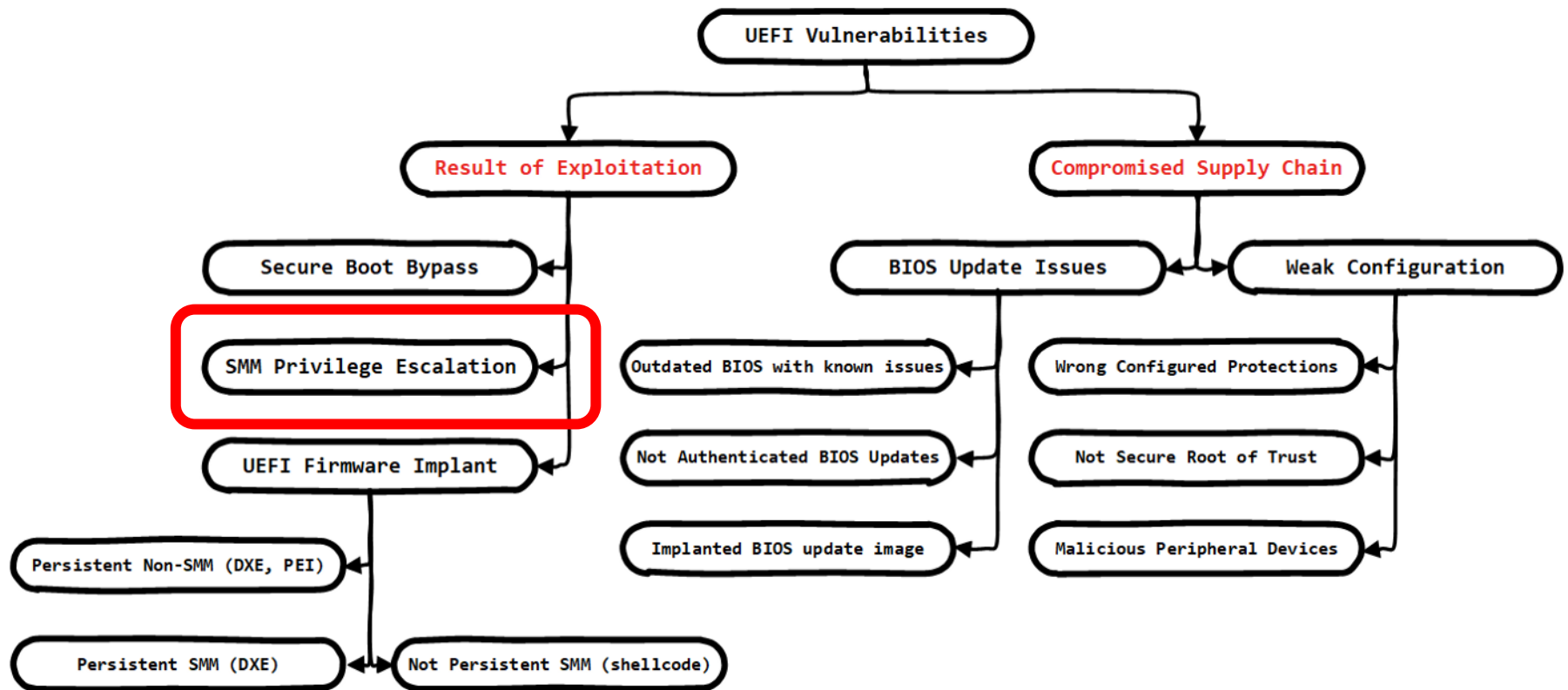
■ Intel Boot Guard

- Secure bootが機能するより前に実行されるコード(Initial Boot Block, IBB)の整合性をハードウェアからチェック
 - IBB = SEC & PEIモジュール
 - Secure bootの整合性を担保
- 改竄が検知された場合はブートを中止する
-  MosaicRegressorが検知されないように見える(DXEモジュールなので)

<https://depletionmode.com/uefi-boot.html>

SMMと セキュリティ

分類



System Management Mode (SMM)とは

- クソの塊
- プロセッサの実行“モード”
 - 特権レベルとは別のコンセプト
 - Ring0 SMMやRing 3 SMMがあり得る
 - 現状Ring0以外の実装なし
 - SMMでのみ実行可能な操作がある
 - 例: SPIフラッシュへの書き込み保護のバイパス
 - 例: ハイパーバイザーによって管理されない(Ring -2と呼ばれることも)

30.1 SYSTEM MANAGEMENT MODE OVERVIEW

SMM is a special-purpose operating mode provided for handling system-wide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by applications software or general-purpose systems software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.

- SMRAMと呼ばれる物理メモリアドレスの範囲内で実行される

SMRAMとは

- メモリーコントローラーによってSMM外からのアクセスが禁止された領域
 - デバッガー、ハイパーバイザー、DMAすべてアクセス不可
 - Host Bridge and DRAM Controller (B0:D0:F0)の2レジスターによって指定される
 - TSEG Memory Base (TSEGMB)
 - Base of GTT stolen Memory (BGSM)

When the extended SMRAM space is enabled, processor accesses to the TSEG range without SMM attribute or without WB attribute are handled by the processor as invalid accesses.

2.5.3 TSEG

- SMMモジュールはこの領域にロードされ、実行される
 - SMRAM内のデータはTrusted
 - SMRAM外のデータはUntrusted
 - カーネルがユーザーモードのメモリを触るのと同じ状況

<https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/10th-gen-core-families-datasheet-vol-2-datasheet.pdf>

SMMモジュール

- ファイルフォーマットはDXE Boot Driverと同じ
 - UEFI “File”内にSMMを示すメタデータあり

➤ PcieSataController	File	DXE driver	File GUID: 0C375A90-4C4C-4428-81
➤ 2BA0D612-C3AD-4249-915D-AA0E8709485F	File	DXE driver	Type: 0Ah
➤ PiSmmCore	File	SMM core	Attributes: 00h
▼ FlashDriverSmm	File	SMM module	Full size: 58DAh (22746)
MM dependency section	Section	MM dependency	Header size: 18h (24)
PE32 image section	Section	PE32 image	Body size: 58C2h (22722)
UI section	Section	UI	Tail size: 0h (0)
Version section	Section	Version	State: F8h
			Header checksum: E8h, valid

- ひとつのBIOSに200以上のSMMモジュールがある場合も
- DXE Coreによってファームウェアボリューム(SPIフラッシュ)から自動的に実行される
 - USB等外部ストレージからのインストールは不可能

System Management Interrupt (SMI)

- ロードされたSMMモジュールはSMIハンドラーを登録する
 - 本当のSMIエントリーポイントはPiSmmCoreが実装
 - SMMモジュールのSMIハンドラーはそこから呼び出されるプラグイン形式
- SMIは以下の方法で発生する
 - ハードウェアから自動的に
 - 温度変化への対応など
 - ソフトウェアから OUT 0xb2, <Command_Number> によって
 - UEFI変数への書き込みなど

SMIハンドラーの登録

■2種類のAPI

- EFI_SMM_SW_DISPATCH2_PROTOCOL.Register()
 - Command_Numberをキーとして登録。一般的
- EFI_MM_SYSTEM_TABLE.MmiHandlerRegister()
 - GUIDをキーとして登録

■SMIハンドラーは同一プロトタイプ

```
typedef
EFI_STATUS
(EFI_API *EFI_MM_HANDLER_ENTRY_POINT)(
    IN EFI_HANDLE    DispatchHandle,
    IN CONST VOID    *Context          OPTIONAL,
    IN OUT VOID      *CommBuffer       OPTIONAL,
    IN OUT UINTN     *CommBufferSize  OPTIONAL
);
```

<https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdePkg/Include/Protocol/SmmSwDispatch2.h#L117>

<https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdePkg/Include/Pi/PiMmCis.h#L341>

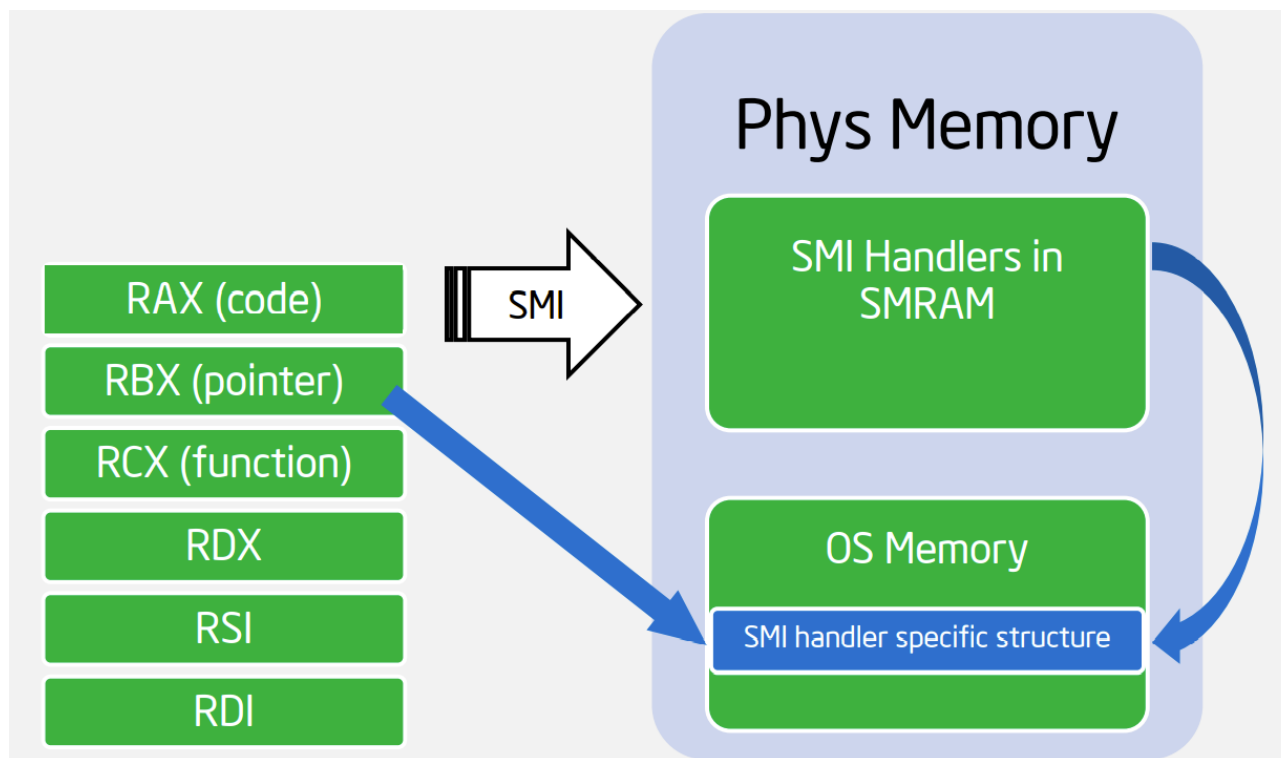
SMIの外部入力の扱い

- SMIハンドラーはSMRAM外からの入力が必要な場合がある
- SMRAM以外からの値は検証必須
- 典型的には2種類
 - SMM Communication Bufferから
 - レジスターから
- それ以外にも
 - 実装依存の物理メモリアドレスから(課題3)

<https://software.intel.com/content/dam/develop/external/us/en/documents/a-tour-beyond-bios-launching-stm-to-monitor-smm-in-efi-developer-kit-ii-819978.pdf>

安全でない検証による脆弱性

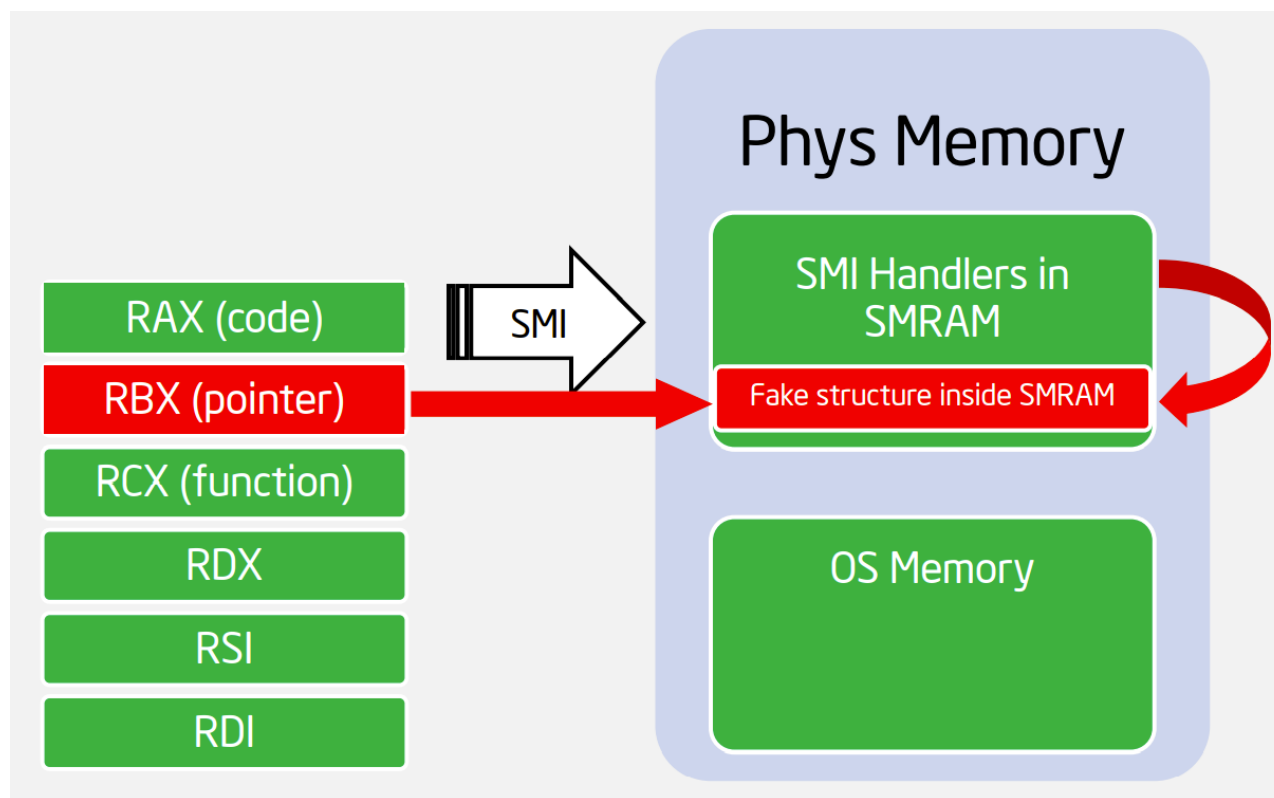
- SMIがレジスタRBXで指定されたアドレスに値を書き込む場合、



http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf

安全でない検証による脆弱性

- RBXがSMRAM内だとカーネルがSMRAMを破壊できる



http://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf

必要な検証

- 外部入力がSMRAM外を指定していることを確認する
 - AMIはAmiValidateMemoryBuffer()を実装
 - EDK2はSmmlsBufferOutsideSmmValid()を実装
- SMRAM内なら使わない

<https://github.com/killvxk/LoginDemo/blob/859b3d842c61b130b6ec115373447650759aee9a/AmiModulePkg/Library/SmmAmiBufferValidationLib/SmmAmiBufferValidationLib.c#L21>

<https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdePkg/Library/SmmMemLib/SmmMemLib.c#L104>

課題3: CVE-2021-26943

- 脆弱性のあるSMMモジュール: UsbRt, SdioSmm, NvmeSmm
- SMIの登録
 - ModuleEntryPoint() -> FUN_80000ec8()

```
smiNumber = 0x42;  
efiStatus = (*EFI_SMM_SW_DISPATCH2_PROTOCOL6->Register)  
            (EFI_SMM_SW_DISPATCH2_PROTOCOL6, swSmiHandler7, &smiNumber, &local_res20)
```

- SMIハンドラー: swSmiHandler7()

課題3: CVE-2021-26943

- SMRAM外の物理メモリアドレスを参照している
 - `usedAsAddress = ((* (uint16_t) 0x40e) * 0x10 + 0x104)`
- 検証はしているが成否にかかわらず値を使っている



```
13     if (addressFromOutsideSmram == NULL) {
14         addressFromOutsideSmram = *((JRam0000000000000040e * 0x10 + 0x104);
15     }
16     else {
17         *CommBuffer = NULL;
18     }
19     efiStatus = IsOutsideSmram(addressFromOutsideSmram);
20     efiStatus2 = 0;
21     if (((efiStatus < 0) || (efiStatus2 = efiStatus, *addressFromOutsideSmram != '\')) ||
22         (0xb < addressFromOutsideSmram[1])) {
23         efiStatus = efiStatus2;
24         addressFromOutsideSmram[2] = '\a';
25     }
26     else {
```

Exploitation - Basic

■ 攻撃手順

- 物理メモリーアドレス0x40eをゼロにする
- 物理メモリーアドレス0x104にSMRAMのアドレスXを書き込む
- SMI 0x42を実行する
- $X + 2$ に '\a' (0x07) が書き込まれる

```
13     if (addressFromOutsideSmram == NULL) {
14         addressFromOutsideSmram = *(&URam0000000000000040e * 0x10 + 0x104);
15     }
16     else {
17         *CommBuffer = NULL;
18     }
19     efiStatus = IsOutsideSmram(addressFromOutsideSmram);
20     efiStatus2 = 0;
21     if (((efiStatus < 0) || (efiStatus2 = efiStatus, *addressFromOutsideSmram != '\')) ||
22         (0xb < addressFromOutsideSmram[1])) {
23         efiStatus = efiStatus2;
24         addressFromOutsideSmram[2] = '\a';
25     }
26     else {
```

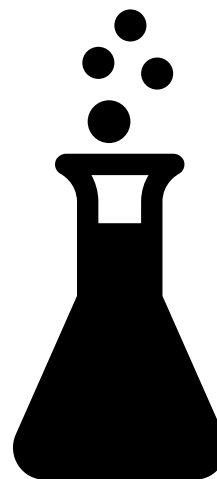
Exploitation – In Practice

- どのSMRAMを書き換える？
 - SMRAM内の簡単に実行できる関数ポインター
 - gSmst->SmmLocateProtocol
 - カーネルモードからアドレスを取得可能
 - 他のSMI経由で実行可能
- カーネルモードで物理メモリーアドレス0x07070707にシェルコードを置く
 - 脆弱性を4度ついて、gSmst->SmmLocateProtocolを0x07070707にする
 - 他のSMI経由でgSmst->SmmLocateProtocol呼び出す
 - シェルコードがSMMで実行される
 - Ring 0 -> SMMローカル特権昇格

<https://github.com/tandasat/SmmExploit>

Lab: 脆弱性とエクスプロイトの確認

- swSmiHandler7()とエクスプロイトの実装を確認する
<https://github.com/tandasat/SmmExploit>
- エクスプロイトがSMM特権昇格のデモとして行っていることはなにか？（ただのカーネルモードではできないことをしている）



脆弱性の影響

- ローカル特権昇格
 - 要件: 初めにカーネルモードでコード実行が必要
 - が、SMMの脆弱性に対しては多くのセキュリティ機能が意味をなさない
 - VBS/HVCI、CFG、kASLR、NX/DEP
 - 影響: ハイパーバイザーの破壊(デモ)、SPIフラッシュの書き換え
- ASUS以外のBIOSにも同モジュールが使用されている
- 50000以上のデバイスで前述脆弱なモジュールの存在を確認
 - 未修正バージョンを使用し続けている
 - 他OEMのBIOSで、脆弱性が検証、報告されていない(Known Zero Day)

本脆弱性のあるモジュールの検知

- SMM開発者
 - OEM間での脆弱性情報の共有
 - 本脆弱性は5年前に報告されたINTEL-SA-00057と同一。ASUSは情報を得られていなかった
- セキュリティ研究者
 - コードパターンの認識(efiXplorerなど)
 - BIOSアップデートなどから当該SMMモジュールの発見
- 組織のIT管理者
 - BIOSバージョンのインベントリーの作成
 - SMMモジュールのインベントリーの作成(ハッシュ、名前、GUID)
 - パッチ管理

<https://github.com/Cr4sh/Aptiocalypsis>
<https://github.com/binarly-io/efiXplorer>

類似脆弱性のあるモジュールの検知

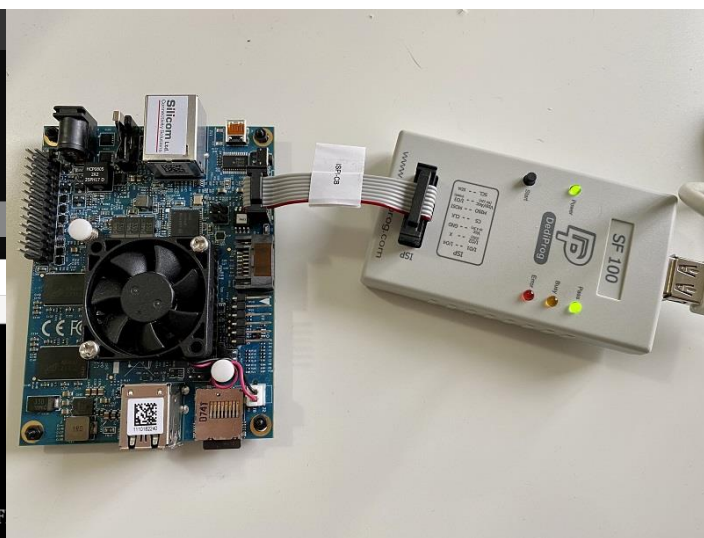
- SMM開発者
 - セキュリティデザインレビュー
 - プロセッサのセキュリティ機能 (SMM_Code_Chk_En、ページ保護など) を有効にする
 - 新しいEDK2を使う
- セキュリティ研究者
 - スクレイピング + BISOアップデート展開 + コードパターンの認識？
 - UEFIエミュレーション？
 - ファジング？
- 組織のIT管理者
 - BIOSハッシュの定期的なチェック (Post-exploitation detection) ？
 - カーネルモジュールの実行管理

MSR_SMM_FEATURE_CONTROL	Package	Enhanced SMM Feature Control (SMM-Rw) Reports SMM capability Enhancement. Accessible only while in SMM.
0		Lock (SMM-RwO) When set to '1' locks this register from further changes.
1		Reserved
2		SMM_Code_Chk_En (SMM-Rw) This control bit is available only if MSR_SMM_MCA_CAP[58] == 1. When set to '0' (default) none of the logical processors are prevented from executing SMM code outside the ranges defined by the SMRR. When set to '1' any logical processor in the package that attempts to execute SMM code not within the ranges defined by the SMRR will assert an unrecoverable MCE.

参考:SMM開発と仮想マシン

- DXEドライバー同様SMMモジュールも開発可能
 - QEMUはSMMをエミュレートできる(ただし不完全)
 - 実機ではSPIフラッシュプログラマーでBIOSをに書き込む必要がある
 - VMwareはSMMを持たない

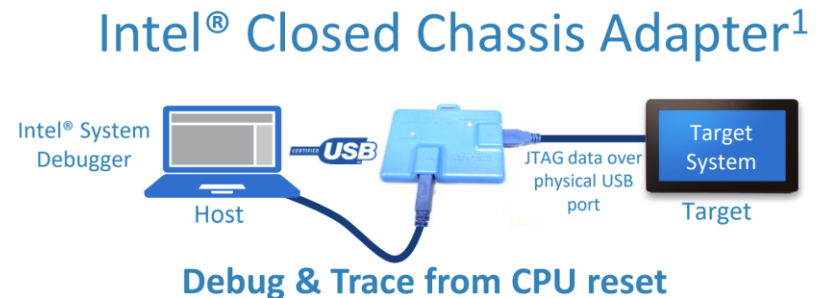
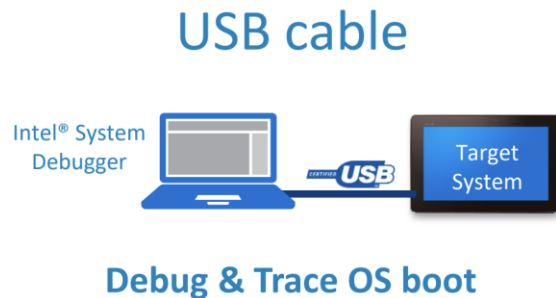
```
user@DESKTOP-2JNHDLQ: /mn × + ▾
INFO - Ftw: Remaining work space size - FE0
INFO - SmmInstallProtocolInterface: 3868FC3B-7E45-43A7-906C-4BA47DE1754D 7E99028
INFO - NOTICE - AuthVariableLibInitialize() returns Unsupported!
INFO - Variable driver will continue to work without auth variable support!
INFO - InstallProtocolInterface: 93BA1826-DFFB-45D0-82A7-E7DCAA3B8DF3 0
INFO - InstallProtocolInterface: 3868FC3B-7E45-43A7-906C-4BA47DE1754D 0
INFO - InstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 608D698
INFO - SmmInstallProtocolInterface: 5B1B31A1-9562-11D2-8E3F-00A0C969723B 7FEAEC0
INFO - Loading SMM driver at 0x00007E92000 EntryPoint=0x00007E93EF6 HelloSmm.efi
INFO - [HelloSmm] HelloSmmInitialize called
INFO - [HelloSmm] SMI 0x00
INFO - QEMU - Press Ctrl+Alt+G to release grab
Machine View
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
FS0: Alias(s): HD0a65535a1::BLK1:
PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x0,0xFFFF,0x0)/HD(1,MBR,0xBE1AF
3F,0xFBFC1)
BLK0: Alias(s):
PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x0,0xFFFF,0x0)
BLK2: Alias(s):
PciRoot(0x0)/Pci(0x1F,0x2)/Sata(0x2,0xFFFF,0x0)
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Shell> _
```



<https://github.com/tandasat/HelloSmm>

参考：SMMデバッグング

- 実機でのSMMデバッグはハードウェアデバッガーが必要
- IntelシステムはUSBを通してDirect Connect Interface (DCI) でデバッグ可



- アンチデバッグに検知されにくいカーネルデバッグ技術としても使用可能
 - Kernel Patch Protection (Windows) の解析等

<https://standa-note.blogspot.com/2021/03/debugging-system-with-dci-and-windbg.html>

参考: SMRAMフォレンジック

- ハードウェアデバッガーを接続した場合、SMRAMを取得可能
- SMRAM内の多くの構造体は4bytesのmagic valueから始まる
 - 'SMST', 'smmc', 'smih' など
 - 構造体のレイアウトはEDK2から取得可能
- SMRAMをパースしてSMIハンドラーなどを特定可能
 - smram_parse.py
 - Authored by Dmytro Oleksiuk @d_olex
 - Updated by myself for Python3

https://github.com/tandasat/smram_parse
https://github.com/Cr4sh/smram_parse

参考： SMMエントリーポイントの実装

- [_SmiEntryPoint \(SmiEntry.nasm\)](#)
 - [SmiRendezvous \(MpService.c\)](#)
 - [BSPHandler \(MpService.c\)](#)
 - [SmmEntryPoint \(PiSmmCore.c\)](#)
 - [SmiManage \(Smi.c\)](#)
 - `SmiHandler->Handler()`
- SMMエントリーポイントは16bit real-modeで始まる
 - すぐに64bitに移行する
 - `SmiRendezvous`は全プロセッサにSMIを発行し、SMIが処理されている間、SMM内にロックする(race conditionの防止)

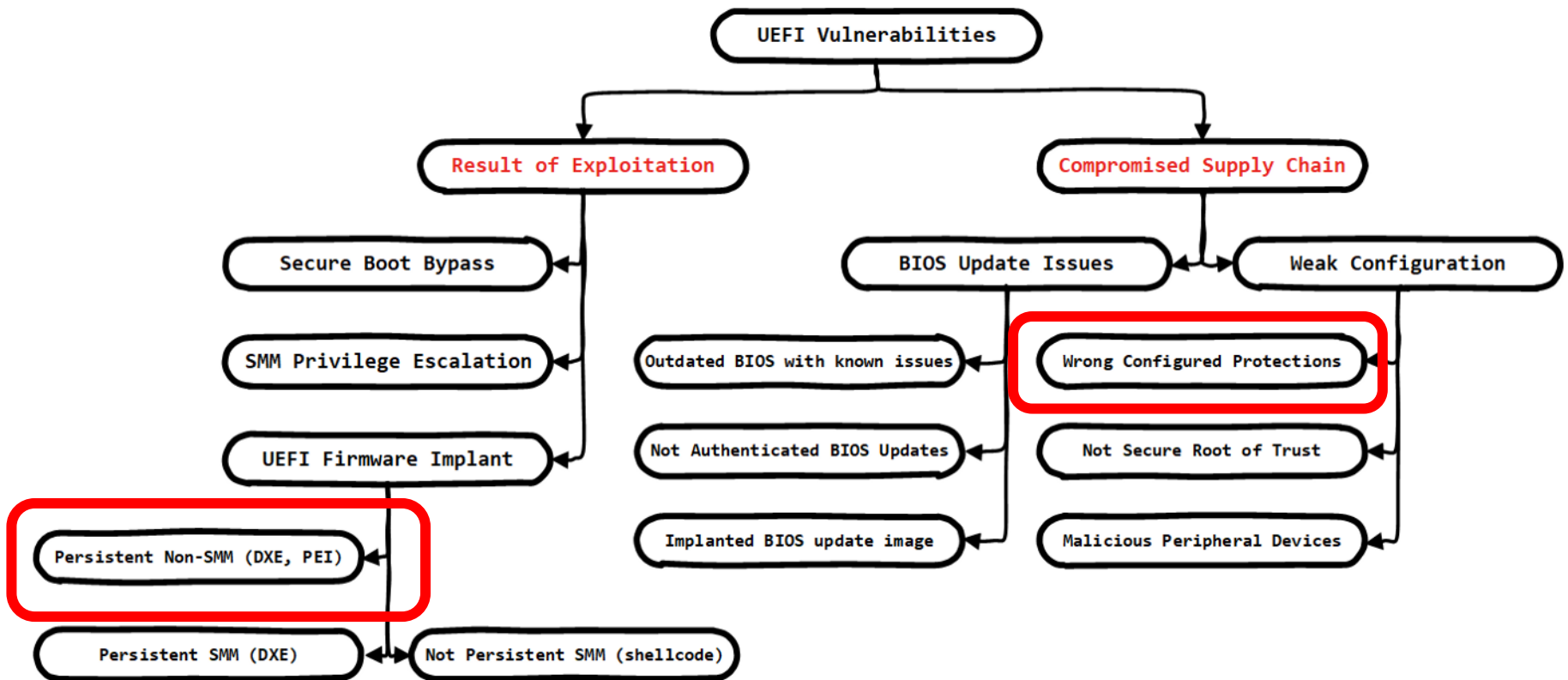
参考：ほかの最近のSMM脆弱性

- SMM callout
 - <https://github.com/binarly-io/Vulnerability-REsearch/blob/main/Lenovo/BRLY-2021-001.md>
- SMM callout via EFI_BOOT_SERVICES
 - <https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there.html>
- Write-what-where through SMM
 - <https://dannyodler.medium.com/attacking-the-golden-ring-on-amd-mini-pc-b7bfb217b437>

UEFIに感染する マルウェアと 脆弱性

(INTEL SPECIFIC)

分類

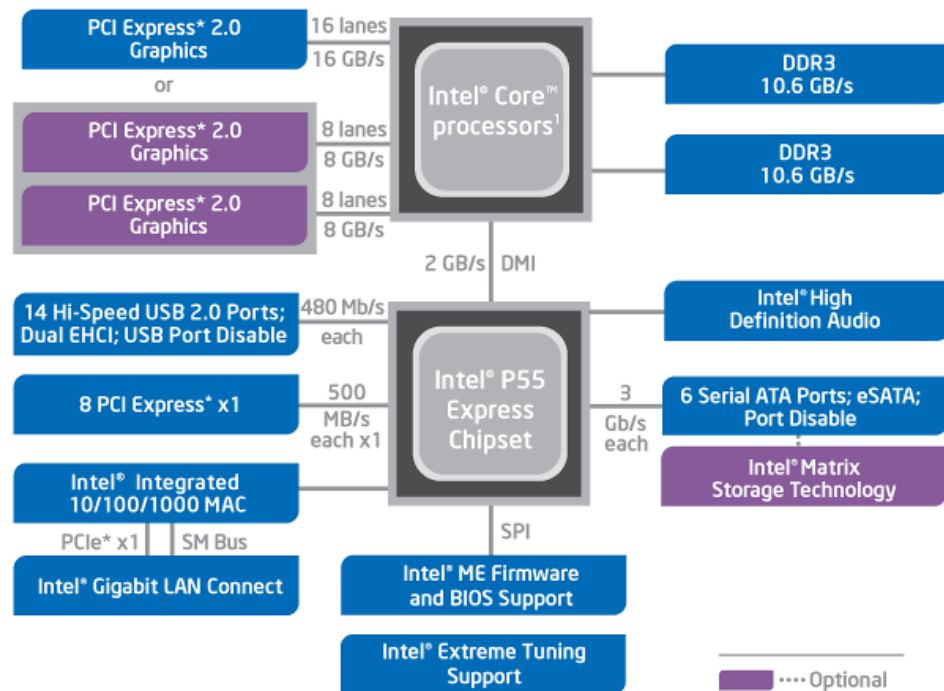


SPIフラッシュの保護

- BIOSはアップデート可能でなければならない
- BIOSはSMMからアップデートされなければならない
 - BIOSはSMMを含み、SMMはOSより権限が高い
 - ☞ OSから書き換え可能の場合、OS -> SMM特権昇格
- ハードウェアが書き込み保護に関するレジスターを提供
 - ☞ レジスターの設定はBIOSが行う
 - ☞ BIOSはOEM(とBIOSベンダー)が実装するソフトウェア
 - ☞ 設定ミスがありうる

Intelチップセットアーキテクチャ

- 上 : CPU – 最新は11th gen
- 下 : Platform Controller Hub (PCH) – 最新は500 Series



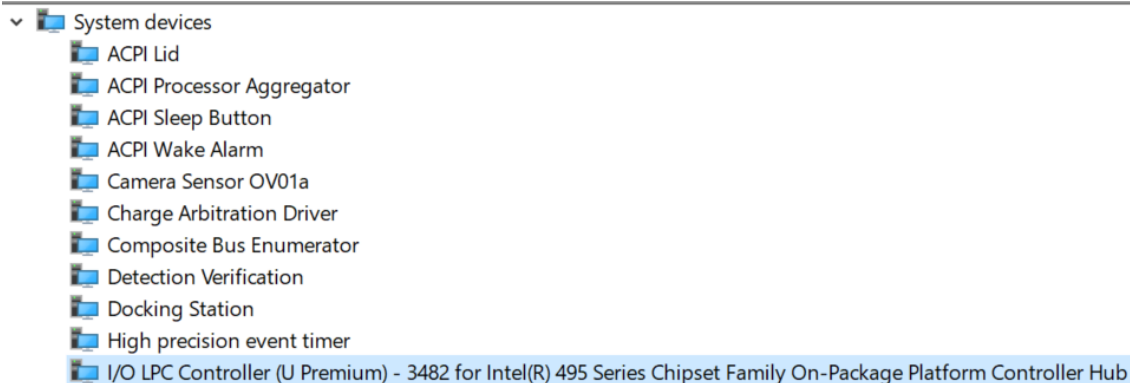
¹ Compatible with:
Intel® Core™ i7-800 processor series
and Intel® Core™ i5 processor family

ソフトウェアによる SPIフラッシュへのアクセス

- ハードウェアシーケンシング (Hardware sequencing)
 - 特定のレジスターに書き込むと一度に64byteまでIOできる
 - おおまかな手順
 1. Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL) レジスターの
 1. Flash Descriptor Valid (FDV)ビットが1であることを確認する
 2. Flash Cycle (FCYCLE)ビットに0(read)を書く(writeの場合は1)
 3. Flash Data Byte Count (FDBC)ビットにIOしたいサイズを書く
 2. Flash Address (BIOS_FADDR) レジスターに読み書きのオフセットを書く
 3. Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL) レジスターの
 1. Flash Cycle Go (FGO)ビットに1を書く
 2. ハードウェアが処理を始める
 3. SPI Cycle In Progress (H SCIP)ビットが0になるまで待つ
 4. Flash Data 0 .. 15 (BIOS_FDATA0 .. 15)レジスターにSPIフラッシュから読み込まれたデータが反映されるので読む
 4. 上記2-3を繰り返す
- 7.2.2 Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL)—Offset 4h**
- 7.2.3 Flash Address (BIOS_FADDR)—Offset 8h**
- 7.2.5 Flash Data 0 (BIOS_FDATA0)—Offset 10h**

レジスター位置の特定

- デバイス情報からPCHのバージョンを特定(例 495 On-Package)



- 仕様書の入手
- 仕様書にある通り、B0:D31:F5がSPIフラッシュコントローラーであることを確認



7 SPI Interface (D31:F5)

SPI Interface (D31:F5)



Intel(R) SPI (flash) Controller - 34A4

Device type: System devices

Manufacturer: INTEL

Location: PCI bus 0, device 31, function 5

<https://www.intel.ca/content/www/ca/en/products/docs/chipsets/495-series-chipset-on-package-pch-datasheet-vol-2.html>

レジスター位置の特定

- 当該レジスターはSPI_BAR0からのオフセットなので、SPI_BAR0を確認

7.2 SPI Memory Mapped Registers Summary

The SPI memory mapped registers are accessed based upon offsets from SPI_BAR0 (in PCI config SPI_BAR0 register).

Table 7-2. Summary of SPI Memory Mapped Registers

Offset Start	Offset End	Register Name (ID)—Offset
0h	3h	BIOS Flash Primary Region (BIOS_BFPREG)—Offset 0h
4h	7h	Hardware Sequencing Flash Status and Control (BIOS_HSFSTS_CTL)—Offset 4h
8h	Bh	Flash Address (BIOS_FADDR)—Offset 8h
Ch	Fh	Discrete Lock Bits (BIOS_DLOCK)—Offset Ch
10h	13h	Flash Data 0 (BIOS_FDATA0)—Offset 10h

- SPI_BAR0はPCI Config Spaceのオフセット0x10

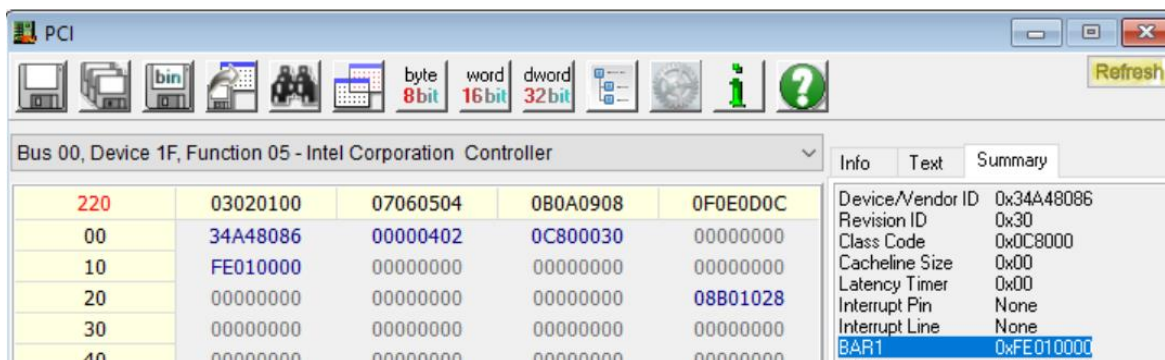
7.1 SPI Configuration Registers Summary

Table 7-1. Summary of SPI Configuration Registers

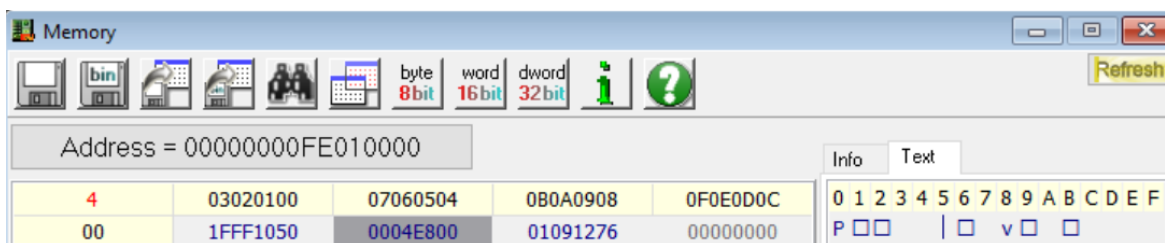
Offset Start	Offset End	Register Name (ID)—Offset
0h	3h	Device ID and Vendor ID (BIOS_SPI_DID_VID)—Offset 0h
4h	7h	Status and Command (BIOS_SPI_STS_CMD)—Offset 4h
8h	Bh	Revision ID and Class Code (BIOS_SPI_CC_RID)—Offset 8h
Ch	Fh	BIST, Header Type, Latency Timer, Cache Line Size (BIOS_SPI_BIST_HTYPE_LT_CLS)—Offset Ch
10h	13h	SPI BAR0 MMIO (BIOS_SPI_BAR0)—Offset 10h

レジスター位置の特定

- PCI Config SpaceはRWEverythingなどのツールで確認可能



- 物理メモリーアドレス0xfe01000(SPI_BAR0) + 仕様書のオフセットがレジスター



<http://rweverything.com/download/>

書き込み保護機能の一部

- SPIインターフェースのBIOS Controlレジスターが書き込み保護の一部機能进行管理

古い名	新しい名前	意味
BIOS Write Enable (BIOSWE)	Write Protect Disable (WPD)	1: BIOSへの書き込みを許可する もし0 -> 1に書き換えられかつLEビットが1ならばSMIが発生
BIOS Lock Enable (BLE)	Lock Enable (LE)	1: EISSが変更不可。WPDが1 -> 0に書き換えられたときSMIが発生
SMM BIOS Write Protect Disable (SMM_BWP)	Enable InSMM.STS (EISS)	1: BIOSへの書き込みをすべてのプロセッサがSMMでない限り禁止する

7.1.8 BIOS Control (BIOS_SPI_BC)—Offset DCh

Lojaxによるチェック

- LojaxはRWEEverything (署名済みドライバー) を用いてレジスターをチェック
 - 可能ならSPIフラッシュへの書き込みを行いBIOSに感染する

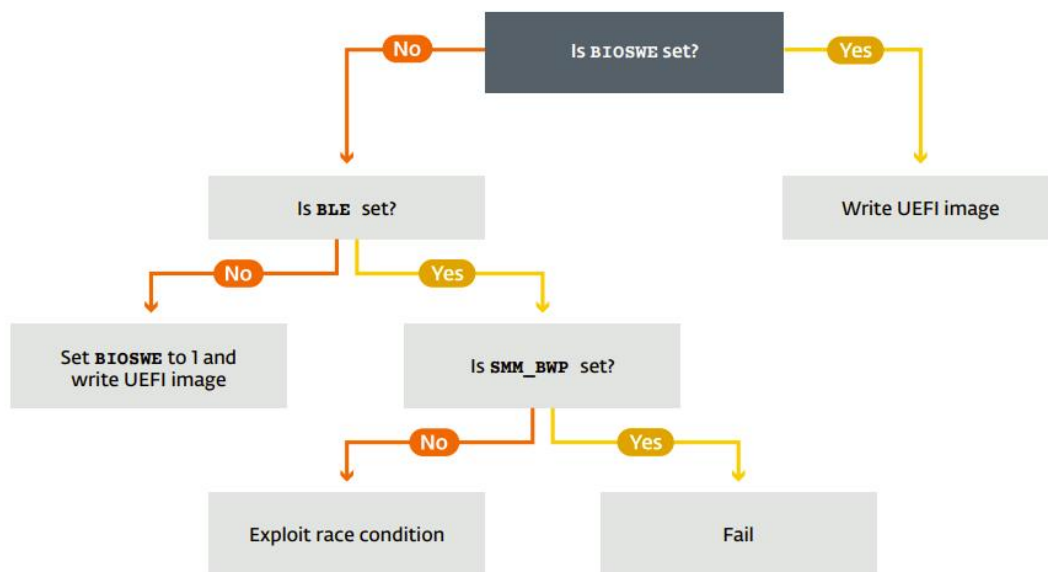


Figure 12 // Decision tree of the writing process

古い名	意味
BIOSWE	1: BIOSへの書き込みを許可する もし0 -> 1に書き換えられかつLEビットが1ならばSMIが発生
BLE	1: EISSが変更不可。WPDが1 -> 0に書き換えられたときSMIが発生
SMM_BWP	1: BIOSへの書き込みをすべてのプロセッサがSMMでない限り禁止する

<https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf>

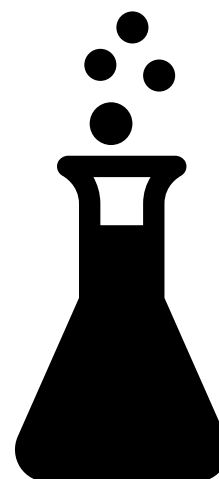
Speed Racer

- SMM_BWP/EISSが0または存在しない場合の脆弱性
- 攻撃シナリオ
 1. BIOSWE/WPDが0、BLE/LEが1とする
 2. CPU1がBIOSWE/WPDに1を書き込む
 3. CPU1でSMIが発生する
 4. CPU2がハードウェアシーケンシングを行い書き込みを行う✳
 5. CPU1がBIOSWE/WPDに0を書き戻す
- SMM_BWP/EISSが1の場合4が失敗する

https://composter.com.ua/documents/Exploiting_Flash_Protection_Race_Condition.pdf

Lab: ReWriter_binaryのレビュー

- 前述のチェックの実装を確認する
- 実装されたロジックはPCH9までしか動作しない（PCH-100以降は機能しない）。なぜか？
 - Hint: PCH9まではLPC Interface(B0:D31:F0)のRoot Complex Base Address、以降はSPI Interface(B0:D31:F5)が必要。仕様書を比較すること



<https://www.intel.com/content/www/us/en/products/docs/chipsets/9-series-chipset-pch-datasheet.html>

<https://www.intel.com/content/www/us/en/products/docs/chipsets/100-c230-series-chipset-pch-datasheet-vol-2.html>

脆弱なデバイスの検知

- PCH5より古いデバイス(2008年頃)
 - SMM_BWP/EISSがない
- レジスターの値を確認するソフトウェアを実行する
 - CHIPSEC
 - CrowdStrike Falcon

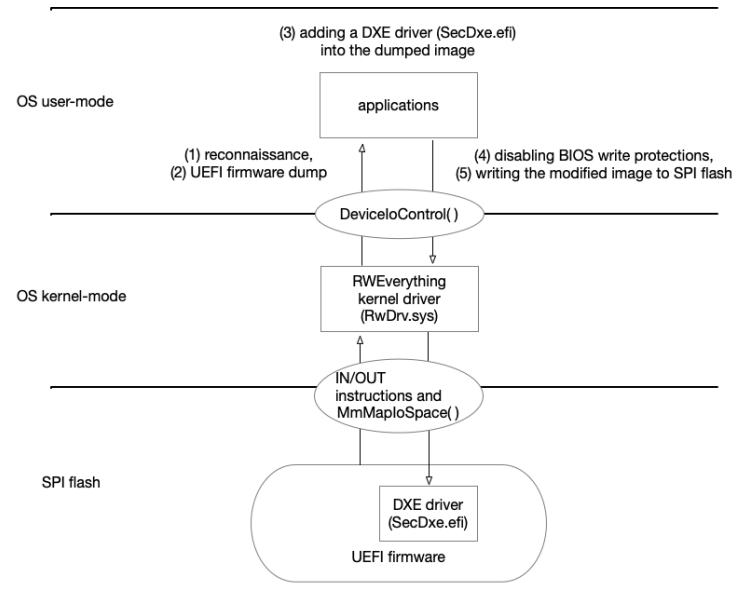
<https://www.crowdstrike.com/blog/crowdstrike-first-to-deliver-bios-visibility/>
<https://github.com/chipsec/chipsec>

同脆弱性をつく 未知のマルウェアの検知

- インストール、実行時
 - カーネルモードドライバのインストールの検知
 - カーネルモードドライバとのIOCTLの検知
 - ハイパーバイザー？
- 感染後
 - Secure Boot(実行の防止)
 - BIOSイメージ内のSecDxeの検出？
 - GUID: 832d9b4d-d8d5-425f-bd52-5c5afb2c85dc
 - SHA256: 7ea33696c91761e95697549e0b0f84db2cf4033216cd16c3264b10daa31f598c
 - BIOSイメージのハッシュ値の変化？
 - autochk.exe等Windowsコンポーネントの検知

参考：本と記事

- Detecting UEFI Bootkits in the Wild
 - <https://blogs.vmware.com/security/2021/06/detecting-uefi-bootkits-in-the-wild-part-1.html>
- Rootkits and Bootkits
 - <https://nostarch.com/rootkits>



最後に

次のステップ？

- アウトプット
 - UEFIマルウェア機能の再実装
 - SMMゼロデイの発見
 - エミュレーターやファザーの研究と実装
- インプット
 - ブログや本を読む
 - 全体像の理解に有用
 - 有名なエンジニアをTwitterでフォローする
 - カンファレンスの講演を見る

Enjoy!



リンク等

- UEFI vulnerabilities classification focused on BIOS implant delivery
- Summary of Attacks Against BIOS and Secure Boot
- Safeguarding UEFI Ecosystem: Firmware Supply Chain is Hard(coded)
- UEFI Firmware Rootkits: Myths and Reality
- MODERN SECURE BOOT ATTACKS: BYPASSING HARDWARE ROOT OF TRUST FROM SOFTWARE
- BETRAYING THE BIOS: WHERE THE GUARDIANS OF THE BIOS ARE FAILING