UEFI Bootkit Hunting: In-Depth Search for Unique Code Behavior

Takahiro Haruyama

Binarly REsearch Team

Takahiro Haruyama
@cci_forensics

Fabio Pagani
@pagabuc

Yegor Vasilenko
@yeggorv

Anton Ivanov
@ant_av7

Sam Thomas
@xorpse

Alex Matrosov
@matrosov

Overview

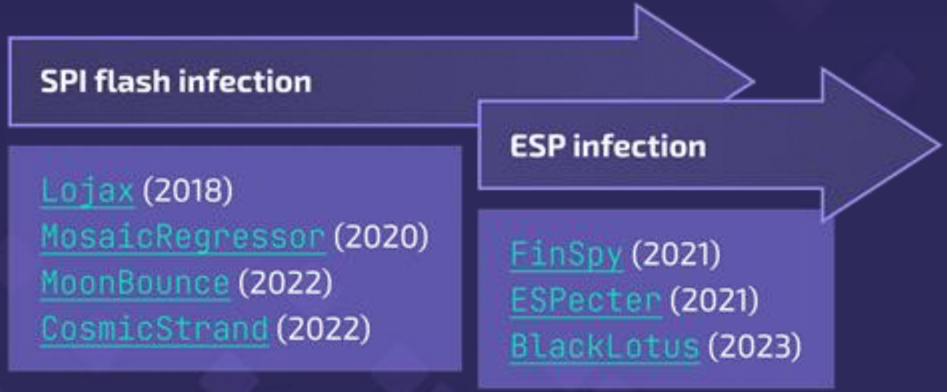
- Background
- Hunting Approach Based on Known Bootkit Analysis
- Hunting Rules and Results
- Going Beyond the Limits of YARA
- Conclusion

Background

What's a UEFI Bootkit?

- **A bootkit is a type of rootkit running before OS boot**
  - Harder to detect than OS-level malware
  - Bypass all OS security mechanisms (e.g., PatchGuard and DSE)
    - Enable to patch OS kernel or run arbitrary kernel shellcode/driver

- **Past UEFI bootkits discovered in the wild**
  - Infection target has moved from SPI flash to ESP due to hardware-based security features (e.g. Intel Boot/ BIOS Guard)



Motivation

- Risk of bootkit infection still exists even if UEFI Secure Boot is enabled
  - Physical access, vulnerability exploits, supply-chain attacks, etc.
- Not only Windows, but also Linux PoCs discovered recently
  - e.g., Bootkitty (Ubuntu) and Pacific Rim (SF-OS?)
- There might be more?
- Motivation
  - Discover unknown bootkits that the world does not recognize

Hunting Approach Based on Known Bootkit Analysis

Hunting Approach Based on Known Bootkit Analysis

- Analyzed known bootkits to extract generic code patterns found in multiple samples
  - Three perspectives: hook chain, additional components/features, OS persistence

	Lojax	MosaicRegressor	MoonBounce	CosmicStrand	ESPecter	BlackLotus
Year of Discovery	2018	2020	2022	2022	2021	2023
Infection target	SPI flash	SPI flash	SPI flash	SPI flash	ESP	ESP
Firmware components	DXE driver	Two DXE drivers and one EFI application	Modified DXE Foundation (CORE_DXE)	Modified CSMCORE DXE driver	Modified Windows Boot Manager binary (bootmgfw.efi)	EFI application disguised as bootloader (grubx64.efi)
Code reuse	NTFS DXE driver (ntfs-3g)	Hacking Team's Vector-EDK	BootLoader	—	—	umap, EfiGuard

binarly

Bootkit Hook Chain

- Lojax and MosaicRegressor set a hook using BS.CreateEventEx() with EFI_EVENT_GROUP_READY_TO_BOOT
- Other bootkits use two types of hooks
  - BS function table hook (MoonBounce and CosmicStrand)
  - Inline code hook (MoonBounce, CosmicStrand, ESPecter and BlackLotus)

MoonBounce	CosmicStrand	ESPecter	BlackLotus
1. Multiple BS function table hooks (CORE_DXE) 2. **OslArchTransferToKernel** (winload.efi) 3. ExAllocatePool (ntoskrnl.exe)	1. BS.HandleProtocol (CSMCORE) 2. Archpx64TransferTo64BitApplicationAsm (bootmgfw.efi) 3. **OslArchTransferToKernel** (winload.efi) 4. ZwCreateSection (ntoskrnl.exe)	1. Entrypoint (bootmgfw.efi) 2. Archpx64TransferTo64BitApplicationAsm (bootmgfw.efi) 3. **OslArchTransferToKernel** (winload.efi) 4. CmGetSystemDriverList (ntoskrnl.exe)	1. ImgArchStartBootApplication (bootmgfw.efi or bootmgr.efi) 2. BlImgAllocateImageBuffer and **OslArchTransferToKernel** (winload.efi)

Bootkit Hook Chain (Cont.)

- Lojax and MosaicRegressor hook pattern (CreateEventEx) is too common and simple
- BS function table hook detection is only effective for CosmicStrand
  - MoonBounce will be missed as it's a DxeCore module with hooked BS
- Generic inline hook detection (e.g., OslArchTransferToKernel) is difficult
  - Memory scanning with code-like byte signatures
    - Signatures and scan algorithms are different for each bootkit
  - Code Patching
    - The patched instructions are also different

Memory scan algorithm comparison

	MoonBounce	CosmicStrand	ESPecter	BlackLotus
Signature size	4 bytes	Combination of 4 bytes and 2 bytes	Combination of one byte and 4 bytes	Flexible length
Search direction	forward	forward/backward	forward	forward

Additional Components/Features

- Inline NTFS DXE driver (Lojax), UEFI application (MosaicRegressor)
  - The BS functions are very common
- Especter and BlackLotus disable security functionalities
  - They are all inline hooks except disabling VBS
  - We can’t define strings generically due to obfuscations

Lojax / MosaicRegressor	ESPecter	BlackLotus
Load Inline ntfs-3g DXE driver (Lojax) or UEFI application (MosaicRegressor): BS.LoadImage() and BS.StartImage()	**Disable verification of the boot manager’s own digital signature:** - Patch BmFwVerifySelfIntegrity (bootmgfw.efi) **Disable Windows Driver Signature Enforcement:** - Patch SepInitializeCodeIntegrity (ntoskrnl.exe)	**Disable VBS:** – SetVariable() – VbsPolicyDisabled (obfuscated) **Disable Windows Defender:** – Patch the driver's entry point – WdFilter.sys/WdBoot.sys (obfuscated) – Access driver list structure (LOADER_PARAMETER_BLOCK, KLDR_DATA_TABLE_ENTRY)

binarly

OS Persistence

- NTFS write functionality (Lojax and MosaicRegressor)
  - Call sequence from BS.HandleProtocol()/OpenProtocol() to EFI_FILE_PROTOCOL.Write()
    - They are common in file system drivers
  - Opened file paths (e.g., Windows folder from root)
    - Not effective if the paths are obfuscated

Lojax	**MosaicRegressor**
Write file/registry in NTFS: BS.OpenProtocol() with EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID EFI_FILE_PROTOCOL.Open() with '\Windows' EFI_FILE_PROTOCOL.Write()	Write file in NTFS: BS.HandleProtocol() with EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID EFI_FILE_PROTOCOL.Open() with '\Windows' EFI_FILE_PROTOCOL.Write()

OS Persistence (Cont.)

- Clearing Write-Protect bit in CR0 register and Shellcode-like PE parsing in multiple bootkits

MoonBounce	CosmicStrand	ESPecter	BlackLotus
Load a kernel driver: • Clear the WP bit in the CR0 register • PE parsing – Resolve kernel API address by hash (IMAGE_EXPORT_DIRECTORY) – Change PE section header flag (IMAGE_SECTION_HEADER) – Resolve relocations (IMAGE_BASE_RELOCATION) – Resolve IAT	**Run kernel shellcode:** • Clear the WP bit in the CR0 register • Copy shellcode to the slack space after .text section of kernel – Resolve kernel API address by hash (IMAGE_EXPORT_DIRECTORY)	**Run kernel shellcode to drop driver/config:** • Clear the WP bit in the CR0 register • Write file using kernel APIs - Resolve kernel API address by hash (IMAGE_EXPORT_DIRECTORY)	**Load Windows kernel driver:** • Rootkit Driver (AES-encrypted) • PE parsing - Copy sections (IMAGE_SECTION_HEADER) - Resolve relocations (IMAGE_BASE_RELOCATION) - Backup disk.sys EP (IMAGE_EXPORT_DIRECTORY) - Get BuildNumber from resource section (IMAGE_RESOURCE_DIRECTORY)

binarly

Our Approach

- We detect the OS-persistence techniques
  - Clearing bits in control registers (WP bit in CR0 register)
    - MoonBounce, CosmicStrand and ESPecter
    - Bootkits remove write protection on read-only memory pages to set inline hook code
      - It's relatively rare, especially in UEFI applications
  - Shellcode-like PE parsing
    - They are common in Windows shellcode, but must be rare in UEFI modules and applications
    - Accessing specific offsets of a structure in succession
      - Resolve kernel API address by string hash (IMAGE_EXPORT_DIRECTORY)
        - MoonBounce, CosmicStrand and ESPecter
      - Resolve code relocations (IMAGE_BASE_RELOCATION)
        - MoonBounce and BlackLotus

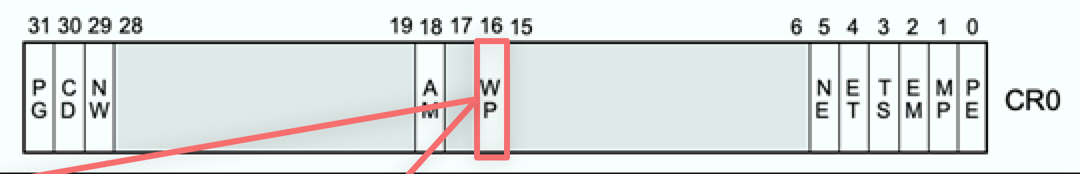
Hunting Rules and Results

Our Threat Hunting Sources and Methods

- **Sources**
  - VirusTotal → YARA (code sequence bytes)
  - Binary Risk Hunt → FwHunt (code sequence bytes + semantic information)
- **Methods**
  1. Detect suspicious samples using YARA/FwHunt rules
  2. Analyze the samples using IDA
     - If there are false positives, refine the rule and re-scan
     - If not, analyze the details to identify the purpose

- We show the YARA rules and hunting results
  - Other advanced detection methods will be introduced in the next section
    - IDAPython batch scan, FwHunt, ML clustering, etc.

Clearing Bits in Control Registers: WP bit in CR0




```
rule bootkit_disable_WP_CR0_2 {
  meta:
    author = "Binarly"
    description = "Designed to catch bootkit cl
    exemplar = "MoonBounce (2d4991c3b6da35745e0                                                  er

  strings:
    // "__writecr0(v1 & 0xFFFFFFFFFFFEFFFF);" or "__writecr0(v1 & 0xFFFEFFFF);"
    // 0f20c0                                   mov     rax, cr0; control/debug register
    // 4825fffffeff                             and     rax, 0FFFFFFFFFFFEFFFFh
    // 0f22c0                                   mov     cr0, rax; control/debug register
    $clear_wp_in_cr0 = { 0F 20 C? [1-5] ff ff fe ff 0f 22 c? }
    // FPs in edk2 (modules in OvmfPkg, UefiCpuPkg, EmulatorPkg)
    $fp_AsmCpuid = { 5?89??5?5?0fa24d85??74??4189??5?e3??89??4c89??e3??89??488b??2?38e3??89??5?5?c3 } // https://github.c
    $fp_AsmCpuidEx = { 5?89??89??5?0fa24c8b??2?384d85??74??4189??4c89??e3??89??4c89??e3??89??488b??2?40e3??89??5?5?c3 } /
    $fp_SevIoWriteFifo8 = { 4887??4987??e8????????85??75??fcf36eeb??e3??8a??ee48ffc?e2??4c89??c3 } // https://github.com/
    // bootkit-like bootloaders
    $fp_konboot = "Kon-Boot Driver loaded"
    $fp_hypersim = "Hypersim booting ..."

  condition:
    filesize < 8MB and pe.is_pe and pe.is_64bit() and
    (pe.subsystem == pe.SUBSYSTEM_EFI_APPLICATION or pe.subsystem == pe.SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER or
     pe.subsystem == pe.SUBSYSTEM_EFI_RUNTIME_DRIVER or pe.subsystem == pe.SUBSYSTEM_EFI_ROM_IMAGE) and
    $clear_wp_in_cr0 and none of ($fp_*)
```

Clearing Bits in Control Registers: WP bit in CR0 (Cont.)

- Discovered two bootlicker (DmaBackdoorBoot) variants whose VT detection rates were 1/71 and 2/68
  - They were just detected as Win/malicious_confidence_70% and MALICIOUS
  - Hook chain:
    - ExitBootServices → OslArchTransferToKernel → ACPI.sys .rsrc shellcode → PsSetCreateThreadNotifyRoutine → shellcode in .text slack space → KeInsertQueueApc → APC callback → KeInsertQueueApc → user-mode shellcode
  - One sample has no user-mode payload, and another downloads shellcode from 192.168.1.44
    - The developers probably submitted their PoCs to check the detection rate?

Clearing Bits in Control Registers: CET bit in CR4

- An open-source bootkit EfiGuard also clears the WP bit in CR0
  - But the previous rule was not effective because the code calls AsmWriteCr0
- Created another rule to detect clearing the CET bit in CR4
  - The code (AsmDisableCet) is hardcoded in assembly so easy to define
  - Specific to EfiGuard, but it's worth creating it as EfiGuard is abused ITW

```
    AsmDisableCet();
  AsmWriteCr0(Cr0 & 0xFFFFFFFFFFFEFFFFuLL);
```

```
UINTN __cdecl AsmWriteCr0(UINTN Cr0)
{
  __writecr0(Cr0);
  return Cr0;
}
```

```
rule bootkit_disable_CET_CR4 {
  meta:
    author = "Binarly"
    description = "Designed to catch bootkit clearing the CET bit in CR4"
    exemplar = "AsmDisableCet in EfiGuard (https://github.com/Mattiwatti/EfiGuard/blob/master/EfiGuardDxe/X64/Cet.asm#L9)

  strings:
    $AsmDisableCet = { b9 a2 06 00 00 0f 32 a8 01 74 0a b8 01 00 00 00 f3 48 0f ae e8 0f 20 e0 0f ba f0 17 0f 22 e0 c3 }

  condition:
```

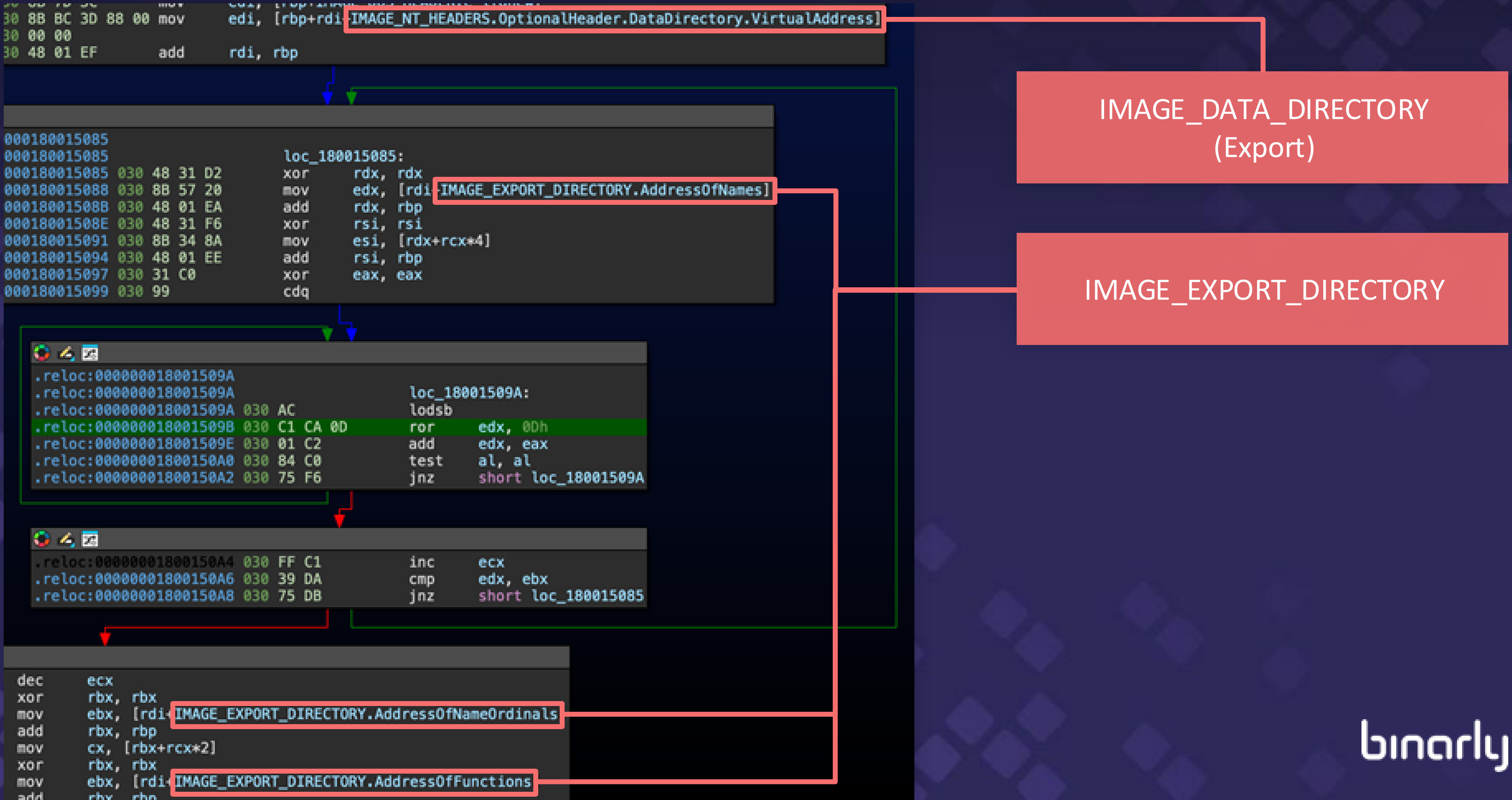

binarly

Clearing Bits in Control Registers: CET bit in CR4 (Cont.)

- Found 2 samples “Vixen.efi” with 0 detections (0/75, 0/73)
  - Compared with the EfiGuardDxe.efi binary built from the source
    - The differences were trivial and the purpose was the same (to disable PatchGuard and DSE)
      - Disabling debug information (print() message, pdb path, etc.)
      - Inline expansion of utility functions through compiler optimizations
      - Difference in the submodule (older version of Zydis)
    - The detection number of EfiGuardDxe is usually around 10–20
  - Also identified the loader for Vixen.efi
    - The code is almost the same as Loader.efi of EfiGuard
  - Searched the bundled files reported on VT using OSINT engines, but no other related sample found
    - Not sure, but the filename “Vixen” indicates game cheat software?

binarly

Shellcode-like PE parsing: Resolving Kernel API Address by String Hash



Resolving Kernel API Address by String Hash

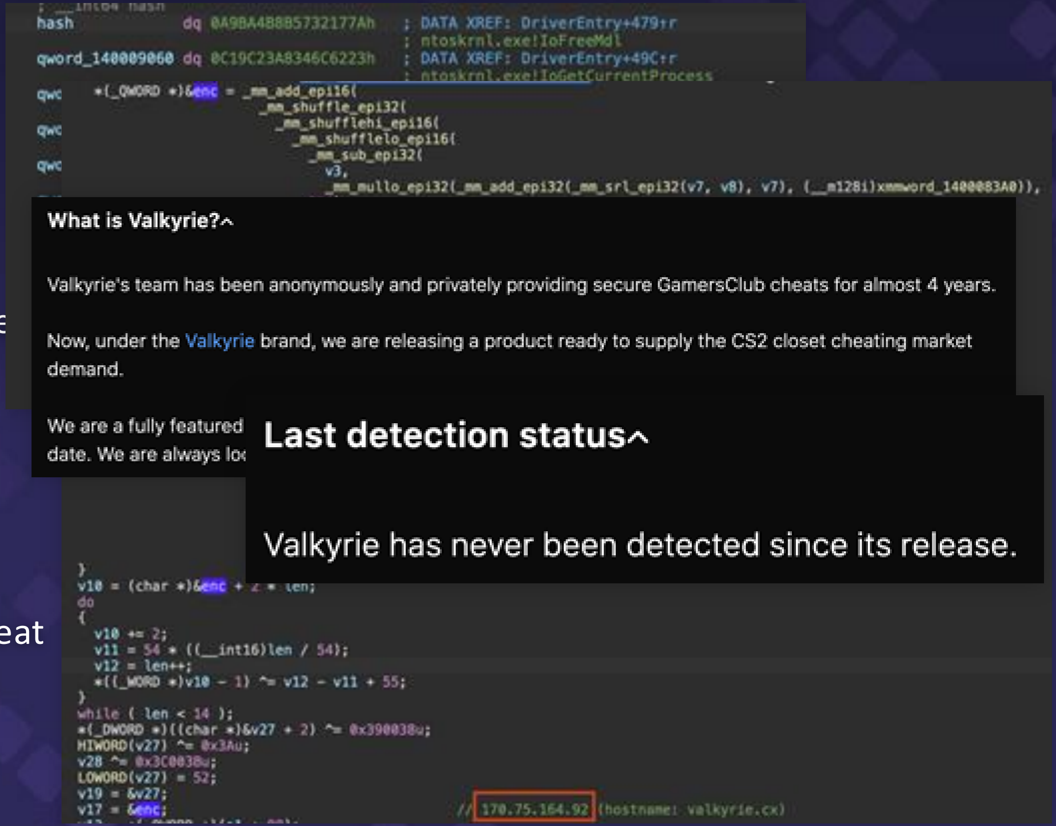
```
rule bootkit_resolve_api_addr {
  meta:
    author = "Binarly"
    description = "Designed to catch potential bootkit samples resolving kernel API address by string hash"
    exemplar = "MoonBounce (2d4991c3b6da35745e0d4f76dffbca56), CosmicStrand (ddfe44f87fac7daeeb1b681dea3300e9), ESPecter (de0743386904654b00

  strings:
    $exp_dir = {
          // 8bbc3d88000000                mov     edi, [rbp+rdi+IMAGE_NT_HEADERS.OptionalHeader.DataDirectory.VirtualAddress]
          // ...
          8b [1-2] 88 00 00 00 [0-15]
          // 8b5720                        mov     edx, [rdi+IMAGE_EXPORT_DIRECTORY.AddressOfNames]
          // 4801ea                        add     rdx, rbp
          // 4831f6                        xor     rsi, rsi
          // 8b348a                        mov     esi, [rdx+rcx*4]
          // ... (hash calculation and check loop)
          8b ?? 20 [0-75]
          // 8b5f24                        mov     ebx, [rdi+IMAGE_EXPORT_DIRECTORY.AddressOfNameOrdinals]
          // 4801eb                        add     rbx, rbp
          // 668b0c4b                      mov     cx, [rbx+rcx*2]
          // 4831db                        xor     rbx, rbx
          8b ?? 24 [0-15]
          // 8b5f1c                        mov     ebx, [rdi+IMAGE_EXPORT_DIRECTORY.AddressOfFunctions]
          // 4801eb                        add     rbx, rbp
          // 4831c0                        xor     rax, rax
          // 8b048b                        mov     eax, [rbx+rcx*4]
          8b ?? 1c [0-15] 8b 04
        }
```

Resolving Kernel API Address by String Hash (Cont.)

- Discovered two samples (1/72, 0/72)
  - An ascii art "Valkyrie" included to output in a debug mode
  - umap-based bootkit, but a more practical implementation
    - Use a JSON configuration file "go.cfg"
    - Use the custom FNV-1-64 hash algorithm for string comparison
    - More code signatures for inline hooking to support more bootloader versions
    - Store a kernel driver payload "loader" into the slack space of the UEFI application image newly allocated in the BlImgAllocateImageBuffer hook function
    - The hook chain is the same
      - ImgArchStartBootApplication → BlImgAllocateImageBuffer → OslFwpKernelSetupPhase1 → ExitBootServices → acpiex.sys entrypoint → injected "loader" entrypoint

Resolving Kernel API Address by String Hash (Cont.)

- Based on the VT relation information, we identified the “[loader]” sample
  - Use the same string hash algorithm for resolving kernel APIs
  - The data and strings are highly obfuscated with SSE instructions
  - One of the decoded strings was an IP address whose hostname was resolved as valkyrie[.]cx
    - According to the website, the bootkit was a part of game cheat software



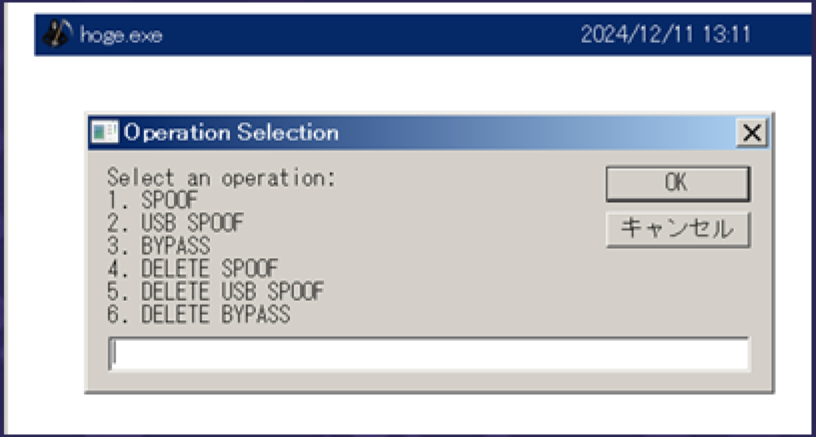
binarly

Shellcode-like PE parsing: Resolving code relocations for kernel drivers

```
rule bootkit_resolve_relocation {
  strings:
    // 8b81b0000000                   mov     eax, [rcx+0B0h]    ; baseRelocDir->VirtualAddress
    // 443991b4000000                 cmp     [rcx+0B4h], r10d    ; baseRelocDir->Size
    $dir_access1 = { 8b??b0000000 [0-30] b4000000 }
    $dir_access2 = { b4000000 [0-30] 8b??b0000000 }
    // c1e80c                         shr     eax, 0Ch    ; UINT16 type = data >> 12;
    // 83f80a                         cmp     eax, 0Ah    ; EFI_IMAGE_REL_BASED_DIR64
    $based_dir64_1 = { c1e?0c [4-12] 83f?0a }
    // be00f00000                     mov     esi, 0F000h
    // bd00a00000                     mov     ebp, 0A000h
    $based_dir64_2 = { b?00f00000 [0-8] b?00a00000 }
    // 81e1ff0f0000                   and     ecx, 0FFFh    ; UINT16 offset = data & 0xFFF;
    $rel_fix = { 81e?ff0f0000 }
    // 4883e808                       sub     rax, 8    ; UINT32 relocCount = (reloc->SizeOfBlock - sizeof(IMAGE_BASE_RELOCATION))
    // 48d1e8                         shr     rax, 1    ;                       / sizeof(UINT16);
    // 4983ea08                       sub     r10, 8
    // 49d1ea                         shr     r10, 1
    $rel_size_of_block1 = { 4?83e?08 [0-3] 4?d1e? }
    // 83c0f8                         add     eax, 0FFFFFFF8h    ; generated by old compilers?
    // ...
    // 49d1ea                         shr     r10, 1
    $rel_size_of_block2 = { 83c?f8 [0-8] 4?d1e? }

  condition:
    filesize < 8MB and pe.is_pe and pe.is_64bit() and
    (pe.subsystem == pe.SUBSYSTEM_EFI_APPLICATION or pe.subsystem == pe.SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER or
     pe.subsystem == pe.SUBSYSTEM_EFI_RUNTIME_DRIVER or pe.subsystem == pe.SUBSYSTEM_EFI_ROM_IMAGE) and
    ($dir_access1 or $dir_access2) and ($based_dir64_1 or $based_dir64_2) and $rel_fix and ($rel_size_of_block1 or $rel_size_of_block2)
```

Resolving code relocations for kernel drivers

- Found 4 umap variants
  - One of them was not detected at all (0/71), but the code was the same
  - Three of them (mp.efi/winboot.efi) have different hook chain
    - ExitBootServices → CreateEvent callback with EVT_SIGNAL_VIRTUAL_ADDRESS_CHANGE → IoInitSystem in OS kernel
    - They are likely another game cheat software



Resolving code relocations for kernel drivers (Cont.)

VirusTotal Detection Trend for BOOTKIT.efi

- Also discovered BOOTKIT.efi (4/71)
  - Detected as Boot.Malware.Bootkit or Trojan.EFI64.Agent
    - The detection number of BOOTKIT.efi **dropped sharply from 6 to 2 last month**
  - A small bootkit disabling PatchGuard and DSE
    - Reuse part of signatures from EfiGuard
  - Hook chain
    - OpenProtocol → BlImgLoadPEImageEx → Several functions in OS kernel
  - Found another variant SandboxBootkit.efi (3/71)
    - The parent compressed file contains exe/sys
    - Game cheat software too ☹

Resolving code relocations for PEI stage backdoor

- PeiBackdoor has no OS-persistence code but similar one resolving relocations of the infected backdoor image

```
rule peibackdoor_relocation {
  meta:
    author = "Binarly"
    description = "Designed to catch potential bootkit samples in PEI stage"
    exemplar = "LdrProcessRelocs in PeiBackdoor (https://github.com/Cr4sh/PeiBackdoor/bl

  strings:
    // 8b413c                              mov     eax, [rcx+3Ch]
    // ...
    // 448b80b4000000                      mov     r8d, [rax+0B4h]
    // 448b88b0000000                      mov     r9d, [rax+0B0h]
    // ...
    // 81e7ff0f0000                        and     edi, 0FFFh
    $reloc64 = { 8b??3c [0-40] 8b??b?000000 [0-20] 8b??b?000000 [0-100] 81e?ff0f0000 }
    // 8b473c                              mov     eax, [Image+3Ch]
    // ...
    // 8b88a0000000                        mov     ecx, [eax+0A0h]
    // ...
    // 8bb0a4000000                        mov     esi, [eax+0A4h]
    // ...
    // 81e1ff0f0000                        and     ecx, 0FFFh
    $reloc32 = { 8b??3c [0-40] 8b??a?000000 [0-20] 8b??a?000000 [0-100] 81e?ff0f0000 }
```

Resolving code relocations for PEI stage backdoor (Cont.)

- The hunting found another backdoor created by the same author, but no other samples were found

Result Summary

	bootmgr.exe (bootlicker variants)	Vixen.efi (EfiGuard variants)	Valkyrie (game cheat software)	bootx64.efi (old umap binary)	mp.efi / winboot.efi (game cheat software)	BOOTKIT.efi / SandboxBootkit.efi (game cheat software)
Number of samples	2	2	2	1	3	2
VT detection rate	1/71, 2/68	0/75, 0/73	1/72, 0/72	0/71	1/73, 1/71, 1/72	3/71, 3/71
VT detection names	Win/malicious_ confidence_70%, MALICIOUS	—	W64.AIDetectMalware	—	W64.AIDetectMalware	Boot.Malware. Bootkit or Trojan.EFI64.Agent
Code reuse (similarity in BinDiff)	bootlicker (0.5% and 0.4%, due to infection with bootmgfw.efi)	EfiGuard (85%)	umap (32%, 39%)	—	umap (62%, 61%)	Part of EfiGuard signatures (9%, code is not similar)
Purpose	Shellcode execution	Disabling PatchGuard and DSE	Game cheating	Mapping a kernel driver	Game cheating	Game cheating
Matched YARA rules	bootkit_disable_WP_CR0	bootkit_disable_CET_CR4	bootkit_resolve_api_addr, bootkit_resolve_relocation	bootkit_resolve_api_addr, bootkit_resolve_relocation	bootkit_resolve_relocation	bootkit_resolve_relocation

binarly

Result Summary (Cont.)

- umap was not detected even though the code was the same
  - It should be detected like DmaBackdoorBoot and EfiGuard
- Bootkits are mostly used for game cheating, not malware
- Except game cheat software, we could not determine if the samples (or improved variants) were actually used by threat actors
  - VirusTotal is just a sample repository, not a telemetry system
  - It's difficult to analyze the attribution of a sample without context

  - Using our findings, we hope AV vendors and security teams will improve their visibility against bootkits

Going Beyond the Limits of YARA

YARA Limitations

- The introduced YARA rules consist only of code-byte sequences
- We explored more detection perspectives

Detection Perspective	Improvements over YARA
Code analysis (e.g., cross-reference, function type/argument, etc.)	Detect YARA's false negatives
Semantic information (e.g., GUID, protocol usage, etc.)	Generate more readable rules with code context
Sample classification using machine learning	Classify samples without writing individual rules
Anomaly detection based on differential firmware analysis	Catch unknown threats like supply-chain attacks

binarly

Static Analysis Automation for detecting OS Kernel/Driver Hooks

- YARA code sequence is not effective to detect code clearing WP bit in CR0 when the bitmask values are passed as the function argument of AsmWriteCr0
  - e.g., EfiGuardDxe and Bootkitty
- Developed static analysis PoC using IDAPython (Hex-Rays decompiler APIs)
  - Scanned over 500 samples with AsmWriteCr0, but no FP found

```
[*] b94ee0e6bce3e5ab3271667ea5a517ed: Start
---------------------------------------------------------------------------------------------------
[*] Find and rename AsmReadCr0/AsmWriteCr0
[+] 0x180001000: AsmReadCr0 detected
[+] 0x180001010: AsmWriteCr0 detected
---------------------------------------------------------------------------------------------------
[*] Identify write-protect disable/enable instructions and code patching calls
[*] 0x18000ef02: clearing WP bit in CR0
[*] 0x18000f040: setting WP bit in CR0 (| 0x10000)
[*] 0x18000f06a: clearing WP bit in CR0
[*] 0x18000f1ca: setting WP bit in CR0 (| 0x10000)
[D] 0x18000ecc0 (get_ranges): ranges = [('0x18000ef02', '0x18000f040'), ('0x18000f06a', '0x18000f1ca')]
[D] 0x18000f021: suspicious memcpy-like call with 3 or more arguments found
[*] 0x18000f021: memcpy-like call within one of the ranges, whose source can be decoded as instructions
[D] 0x1800125d0 ( 10): mov     rax, 0
[D] 0x1800125da (  2): jmp     rax
[+] 0x18000ecc0: code patching found at 0x18000f021 (source = 0x1800125d0) and decoded instructions size (12) matched)
```

Detection example in Bootkitty

binarly

Semantic Detection using FwHunt

- Our community scanner fwhunt-scan detects threats based on semantic information
  - FwHunt: rule specification and examples
  - fwhunt-ida: IDA plugin for creating FwHunt rules
- e.g., the FwHunt rule clearing WP bit in CR0 register

Exclude FPs using GUIDs/Protocols

```
$clear_wp_in_cr0 = { 0F 20 C? [1-5] ff ff fe ff 0f 22 c? }
// FPs in edk2 (modules in OvmfPkg, UefiCpuPkg, EmulatorPkg)
$fp_AsmCpuid = { 5789??575?0fa24d85??74??4189??5?e3??89??4c89??e3??89??488b??2?38e3??89??5?5?c3 }
$fp_AsmCpuidEx = { 5789??89??5?0fa24c8b??2?384d85??74??4189??4c89??e3??89??4c89??e3??89??488b??2?
$fp_SevIoWriteFifo8 = { 488??498???e8?????????85???5??fcf36eeb??e3??8a??ee48ffc?e2??4c89??c3 } //
// bootkit-like bootloaders
$fp_konboot = "Kon-Boot Driver loaded"
$fp_hypersim = "Hypersim booting ..."
```

YARA

```
code:
  and:
    # 0f20c0                                   mov
  - pattern: 0f20c.
    # 4825fffffeff                             and
    # 0f22c0                                   mov
  - pattern: fffffeff0f22c.
guids:
  not-any:
    - name: PCD_PPI_GUID
      value: 06E81C58-4AD7-44BC-8390F10265F72480
    - name: PCD_PROTOCOL_GUID
      value: 11B34006-D85B-4D0A-A290D5A571310EF7
#protocols:
#  not-any:
#    - name: PCD_PROTOCOL_GUID
#      value: 11B34006-D85B-4D0A-A290D5A571310EF7
#      service:
#        name: LocateProtocol
strings:
  not-any:
    - Kon-Boot Driver loaded
    - Hypersim booting ...
```

FwHunt

ML-based Sample Clustering

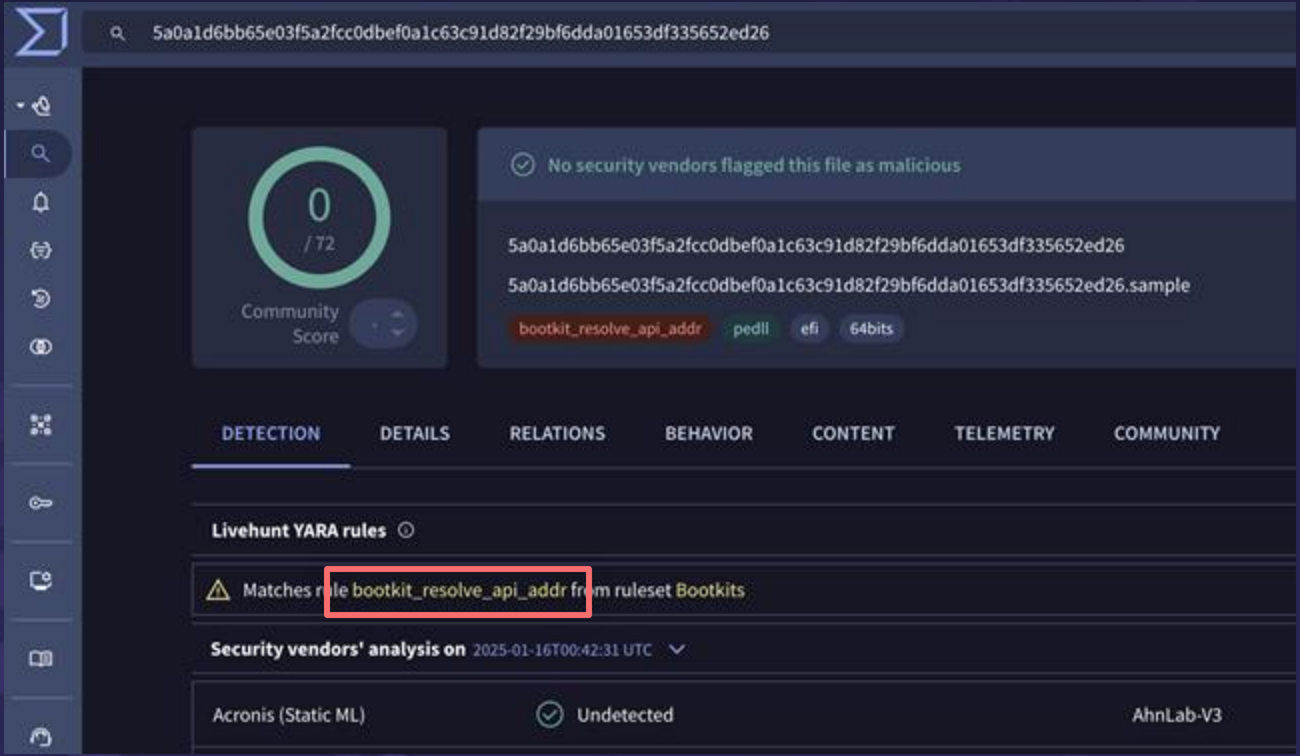
- Quickly identify UEFI binaries using semantic information
  - Similar to Windows malware categorization using IAT (Imphash and ImpFuzzy)
- Clustering process
  1. Extract semantic information (GUIDs/protocols/PPIs/NVRAM variables) using FwHunt
  2. Calculate the TLSH fuzzy hash value of the extracted information
  3. Create clusters using scikit-learn's DBSCAN algorithm based on the distance calculated by TLSH
- Much more accurate categorization than calculating only from whole binary data
  - ARI evaluation: binary data = **0.302**, semantic info = **0.78**, mixed = **0.922**

EFI_SIMPLE_BOOT_FLAG_VARIABLE_GUID-EFI_LOADED_IMAGE_PROTOCOL_GUID-...
= T17DF0651A32CD0E208AAA0C08744BCB14DD0FC894E66CC17FFECA0DC28763479D839B21

LocateHandleBuffer(EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_GUID)-HandleProtocol(EFI_LOADED_IMAGE_PROTOCOL_GUID)-...
= T11C01441D320E16E8966A5C856847B515CF0FC574EC6FCE6EB1CBE812437317AA83D710

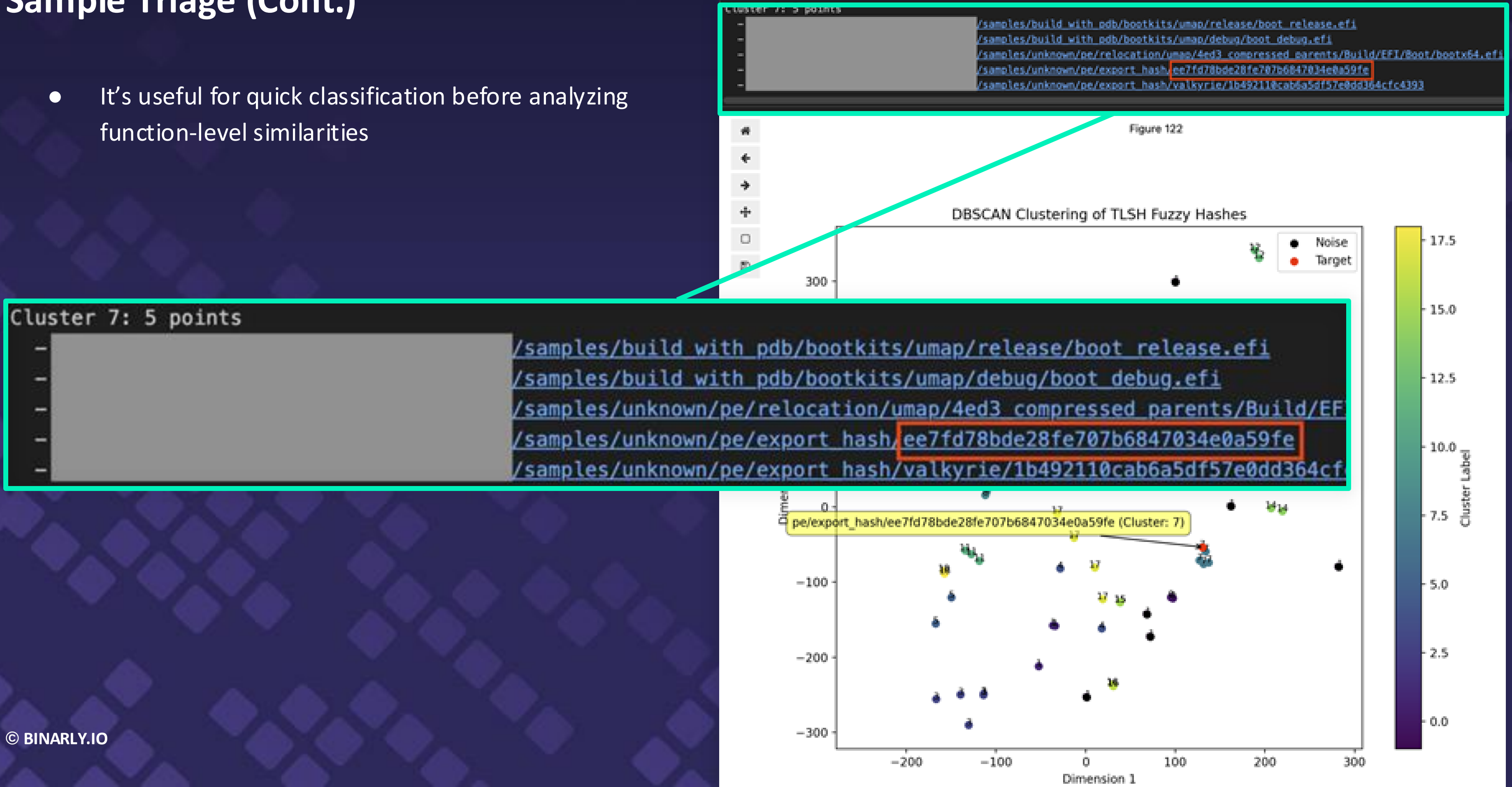
Sample Clustering Use Case: Suspicious Sample Triage

- Got one sample hit by a VT Livehunt rule



Sample Clustering Use Case: Suspicious Sample Triage (Cont.)

- It's useful for quick classification before analyzing function-level similarities



Anomaly Detection Based on Differential Firmware Analysis

- Infection-type bootkits may require detailed analysis for detection
  - e.g., MoonBounce, CosmicStrand, ESPecter and bootlicker
  - What if unknown bootkits have no unique code patterns utilized in this research?
  - The malicious code is small and has no additional semantic information

- We propose anomaly detection based on differential analysis of firmware snapshots
  - Detect unknown threats such as supply chain attacks by comparing different versions of the same firmware over time
  - Various change detection granularities supported
    - Module (added/changed/removed, dependency expression, etc.)
    - Semantic information (NVRAM variables, protocols, etc.)
    - Functions

binarly

Anomaly Detection Based on Differential Firmware Analysis (Cont.)

- Module similarity detection
  - Apply the detection only to the same modules in current/previous firmware
  - Build a function representation that allows us to find near duplicates
    - Calculate pairwise module similarity based on % of matched duplicates
  - Also perform capability diffing and its similarity measurement

- If any change is detected, additionally run generic detections
  - UEFI service table function hooks
  - Embedded executables (scanned by capa rules later)

binarly

Anomaly Detection Use Case: Firmware Implant Detection


```
"firmware1": "fbba6d87e85956a7e9f47d67a0714fbd_orig_fw",
"firmware1_sha256": "30d165d0b4b6acbb0bc4c6278596945cc6a79b810fdef15
"firmware2": "fbba6d87e85956a7e9f47d67a0714fbd_mod_fw",
"firmware2_sha256": "b4a7ff07d797412cc39aae19a7318855506817829e73f7a
"similarity": "Very similar (greater than 97.5% similarity)",
"guid_similarity": "Exact match (100% similarity)",
"variable_similarity": "Exact match (100% similarity)",
"module_similarity": "Very similar (greater than 97.5% similarity)",
"module_additions": 0,
"module_removals": 0,
"modules_new_or_changed": 1,
"modules_unchanged": 279,
```


1. Module change check


```
{
  "name": "FunctionSimilarity",
  "meta": {
    "description": "Check how similar the module's functions are to the same
    "extra_info": {
      "modules": [
        {
          "guid": "5ae3f37e-4eae-41ae-8240-35465b5e81eb",
          "hash": "6e6a2263d7bbe77d078b63363d83aad655e9d356972033582b2cc8db268e839",
          "name": "CORE_DXE",
          "similarity": "Very similar (greater than 97.5% similarity)
```


2. Function similarity check


```
"name": "artefact/embedded-executable",
"confidence": 1.0,
"kind": {
  "kind": "referenced-artefact",
  "value": {
    "id": "934d0672-0f4c-b740-69a8-70d382ac5045",
    "parent_id": "ebfa9771-8497-496c-9fbf-7ca8236f03d3",
    "parent_kind": "component",
    "name": "executable",
    "kind": "PE",
    "ranges": [
      {
        "start": 1413120,
        "end": 1451008
      }
    ],
    "hashes": {
      "md5": "934d06720f4cb74069a870d382ac5045",
      "sha1": "3d2e6f0c3b6fd0fb44966adb4f13679e4091d851",
      "sha256": "f17c1f644cef38d7083cd6ddeb52bfda2d36d0376
```


3. Additional generic check (embedded PE)

Conclusion

Conclusion

- Bootkit detection is still immature
  - Our generic detection approach focusing on OS-persistence techniques was effective in hunting previously undetected bootkits
  - We also introduced advanced detection techniques to solve YARA’s limitations

- Future work
  - More generic detection indicators
    - Considering code obfuscations?
  - ARM-based bootkits?

Thank you!

Acknowledgements

- Martin Smolár and Anton Cherepanov
- Aleksandar Milenkoski
- Brian Baskin