LogoFAIL

Security implications of image parsing during system boot

Fabio Pagani
Alex Matrosov
Yegor Vasilenko
Alex Ermolov
Sam Thomas
Anton Ivanov

Binarly

H2HC
HACKERS TO HACKERS CONFERENCE

$ whoami

Fabio Pagani
@pagabuc

Research Scientist @ Binarly

- Vulnerability and Threat Research
- Program analysis
  - Fuzzing, Dynamic analysis

Academic background

- PostDoc @ UCSB SecLab
- Looked at binary code from different angles (binary similarity, fuzzing, forensics)

© BINARLY.IO

Binarly REsearch Team

Fabio Pagani
@pagabuc

Alex Matrosov
@matrosov

Yegor Vasilenko
@yeggorv

Alex Ermolov
@flothrone

Sam Thomas
@xorpse

Anton Ivanov
@ant_av7

Logo*FAIL* [edition]

Binarly

Scan

The Far-Reaching Consequences of LogoFAIL (Blog)

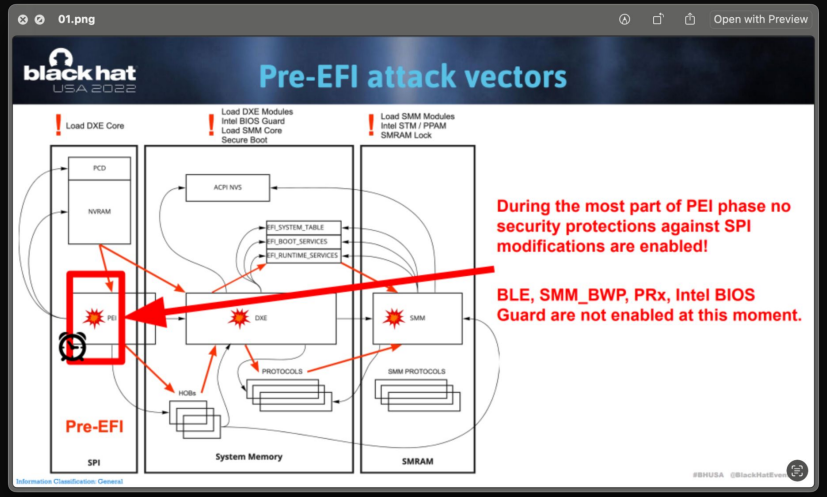
Inside the LogoFAIL Vulnerabilities (Video)

© BINARLY.IO

Binarly

Data-Only Attacks Against UEFI Firmware 🔥

- **Insecure handling of content from R/W areas (NVRAM)**
- **Allow bypassing Secure Boot and hardware-based Verified Boot:**
  - Intel Boot Guard
  - AMD Hardware-Validated Boot
  - ARM TrustZone-based verification
- **Lead to compromise of other protections in Pre-EFI like Intel PPAM**



Breaking Firmware Trust From Pre-EFI: Exploiting Early Boot Phases

https://i.blackhat.com/USA-22/Wednesday/US-22-Matrosov-Breaking-Firmware-Trust-From-Pre-EFI.pdf

© BINARLY.IO

Binarly

Exploring new Attack Surfaces 🔬

While looking at vulnerabilities discovered by our platform, we observed that image parsers in firmware are actually quite common.

But why do we even need image parsers during boot?!

© BINARLY.IO

History Repeats Itself

02.png — Open with Preview

tiano_edk/source/Foundation/Library/Dxe/Graphics/Graphics.c:

```
EFI_STATUS ConvertBmpToGopBlt ()
{
...
 if (BmpHeader->CharB != 'B' || BmpHeader->CharM != 'M') {
     return EFI_UNSUPPORTED;
   }

   BltBufferSize = BmpHeader->PixelWidth * BmpHeader->PixelHeight
      * sizeof (EFI_GRAPHICS_OUTPUT_BLT_PIXEL);
   IsAllocated   = FALSE;
   if (*GopBlt == NULL) {
     *GopBltSize = BltBufferSize;
     *GopBlt     = EfiLibAllocatePool (*GopBltSize);
```

Attacking Intel BIOS at BlackHat USA 2009 by Rafal Wojtczuk and Alexander Tereshkin

https://www.blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf

© BINARLY.IO

History Repeats Itself (~15 years later)

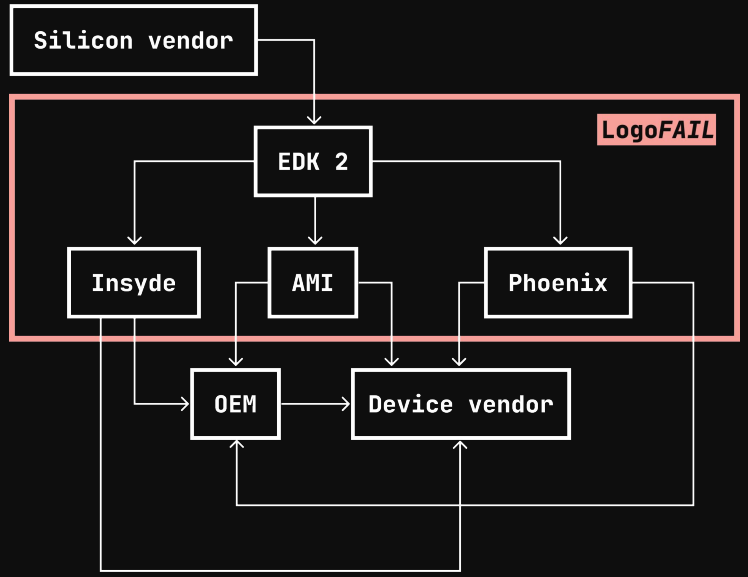
- Different image parsers available in UEFI firmware
  - BMP, GIF, PNG, JPEG, PCX, and TGA
- User can pass image data to them
  - Various logo customization features are available
- Image parsing is done during boot
  - DXE phase
  - C-written code (3rd party)
  - No mitigations for exploitation of software vulnerabilities

What could go wrong?!

© BINARLY.IO

Meet Logo*FAIL*

- New set of security vulnerabilities affecting image parsing libraries used during the device boot process
- LogoFAIL is cross-silicon and impacts x86 and ARM-based devices
- LogoFAIL is UEFI and IBV-specific
- Impacts the entire ecosystem across this reference code and device vendors

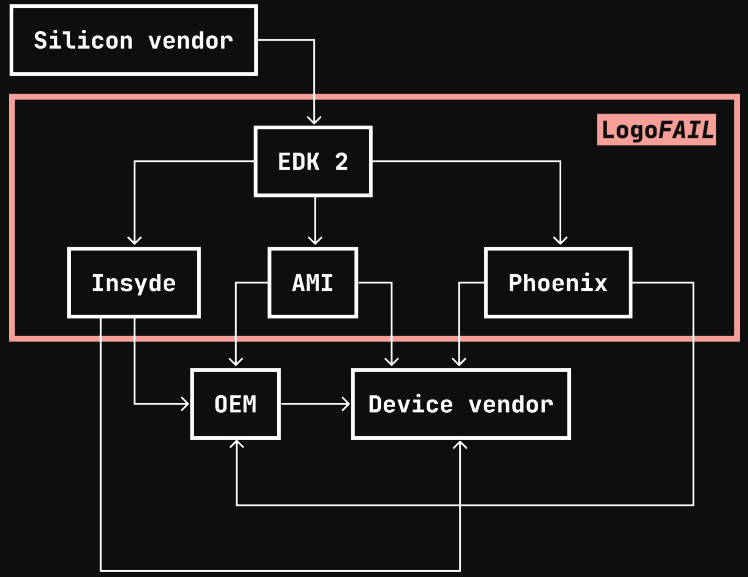


© BINARLY.IO

Binarly

Meet Logo*FAIL*

- New set of security vulnerabilities affecting image parsing libraries used during the device boot process
- LogoFAIL is cross-silicon and impacts x86 and ARM-based devices
- LogoFAIL is UEFI and IBV-specific
- Impacts the entire ecosystem across this reference code and device vendors



150+ days of embargo lifted 3 days ago!

© BINARLY.IO

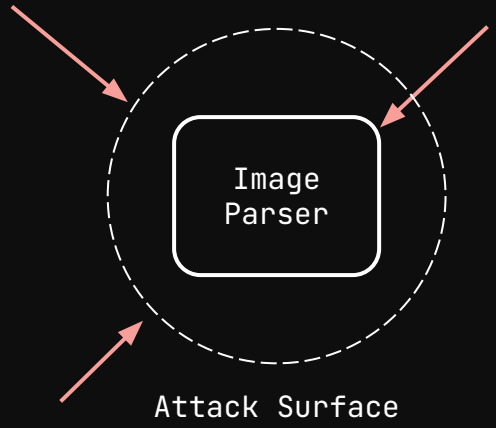
Binarly

Implications of LogoFAIL

Attack Vector	Vulnerability ID	Exploited in-the-wild	Impact	CVSS Score	CWE
	VU#811862 CVE-2023-40238 CVE-2023-5058 CVE-2023-39539 CVE-2023-39538 and more ...	Unknown	HW-based Verified Boot and Secure Boot Bypass x86 and ARM	8.2 High 6.7Medium	CWE-122: Heap-based Buffer Overflow CWE-125: Out-of-bounds Read
Baton Drop	CVE-2022-21894 CVE-2023-24932		Secure Boot Bypass x86	6.7 Medium	CWE-358: Improperly Implemented Security Check for Standard
3rd-party Bootloaders	VU#309662	Unknown	Secure Boot Bypass x86	6.7 Medium	CWE-358: Improperly Implemented Security Check for Standard
BootHole	VU#174059	Unknown	Secure Boot Bypass x86	8.2 High	CWE-120: Buffer Copy without Checking Size of Input

© BINARLY.IO

Attack Surface



© BINARLY.IO

Binarly

Different Shades of UEFI Image Parsers 

```
BmpDecoderDxe-A9F634A5-29F1-4456-A9D5-6E24B88BDB65
TgaDecoderDxe-ADCCA887-5330-414A-81A1-5B578146A397
PngDecoderDxe-C1D5258B-F61A-4C02-9293-A005BEB3EAA1
JpegDecoderDxe-2707E46D-DBD7-41C2-9C04-C9FDB8BAD86C
PcxDecoderDxe-A8F634A5-28F1-4456-A9D5-7E24B99BDB65
GifDecoderDxe-1353DE63-B74A-4BEF-80FD-2C5CFA83040B
```

```
SystemImageDecoderDxe-5F65D21A-8867-45D3-A41A-526F9FE2C598
```

```
AMITSE-B1DA0ADF-4F77-4070-A88E-BFFE1C60529A
```

```
MdeModulePkg/Library/BaseBmpSupportLib/BmpSupportLib.c
```

insyde®

phoenix technologies

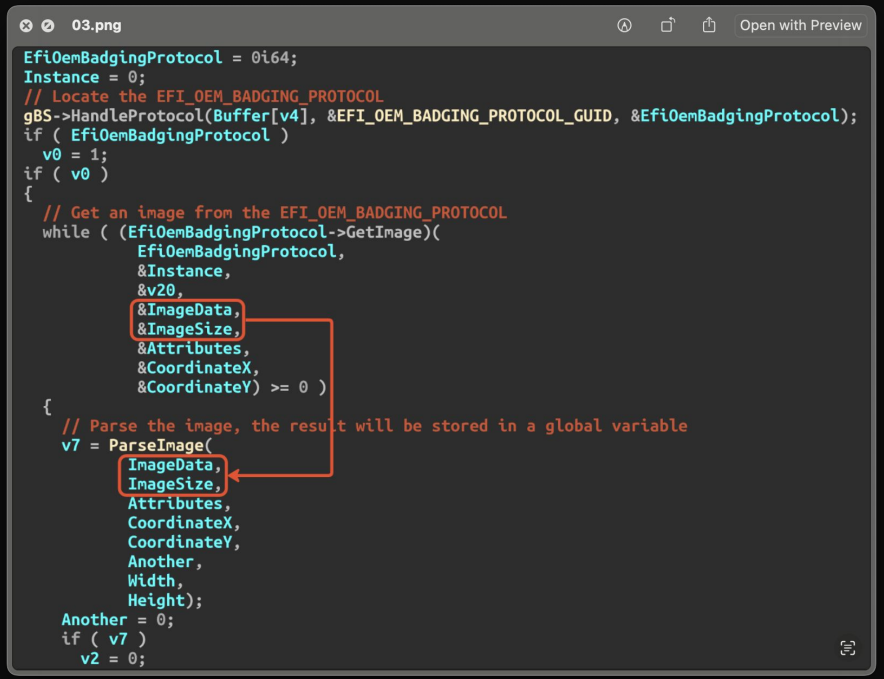
ami

tianocore

© BINARLY.IO

Identifying the Attack Surface

- All the channels used by firmware to read a logo image
- A lot of reversing with `efiXplorer`
- Start from image parsers, then looks “backwards”



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

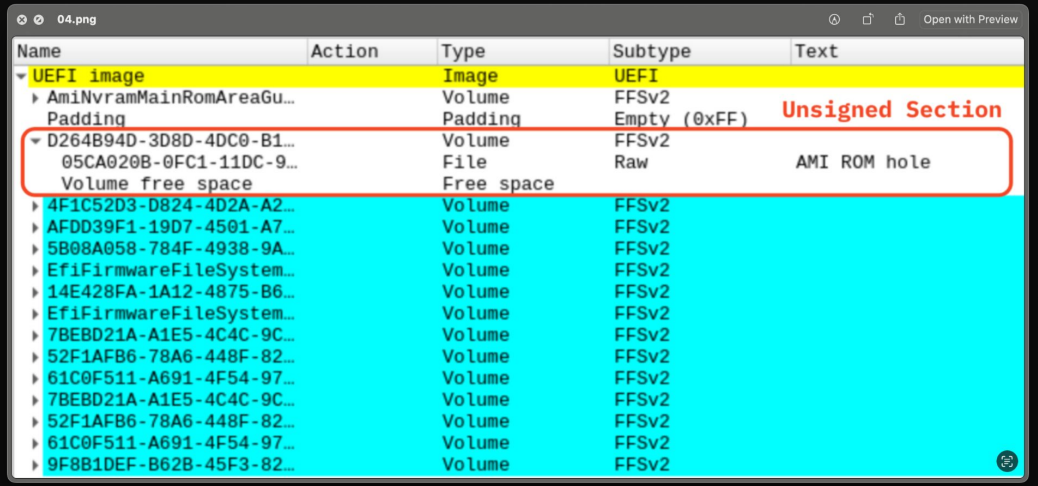
© BINARLY.IO

Binarly

Attack Surface

Several OEM-specific customizations:

1. Logo is read from a fixed location (e.g., “\EFI\OEM\Logo.jpg”)
2. Logo is stored into an unsigned volume of a firmware update
3. An NVRAM variable contains the path of the logo
4. An NVRAM variable contains the logo itself



https://binarly.io/advisories/BRLY-2023-006
https://binarly.io/advisories/BRLY-2023-018

© BINARLY.IO

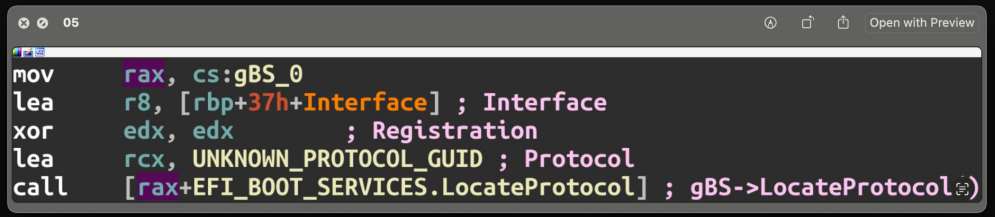
Binarly

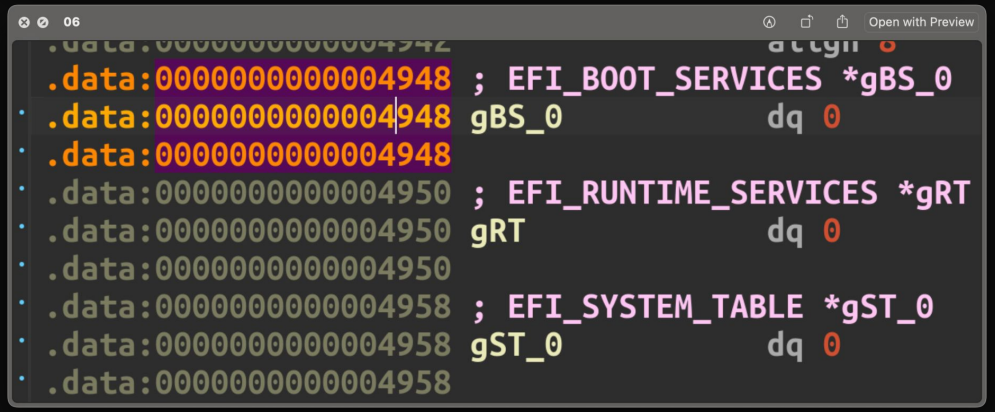
Fuzzing

© BINARLY.IO

Fuzzing UEFI Image Parsers

- UEFI DXE modules are normal PE files
- Minimal UEFI runtime environment needs to be re-hosted
- Fuzzer based on newly-developed emulation capabilities which we integrated with LibAFL





© BINARLY.IO

Binarly

Fuzzing Harness

A bridge between the fuzzer and the fuzzed module:

- Module initialization (protocols are installed)
- Prepare call to parsing function
- Forwards fuzzer-generated data to the target module

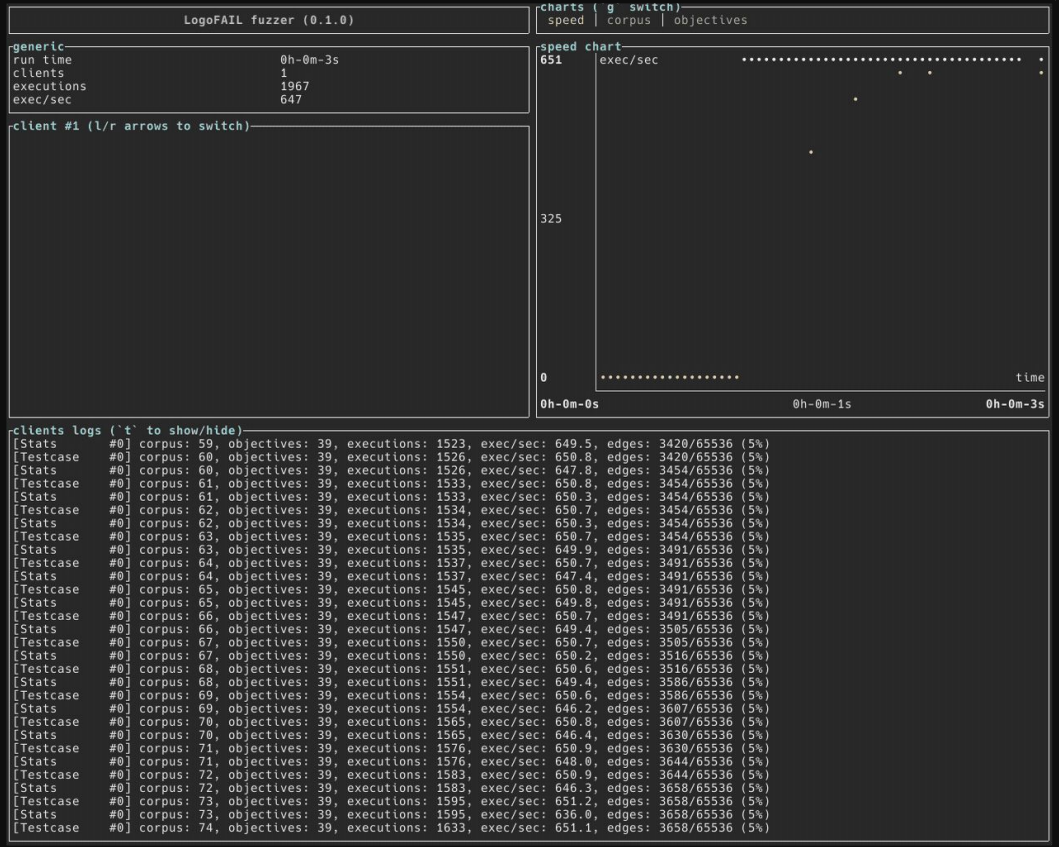
We are ready to fuzz!

© BINARLY.IO

Root Causes

- We found hundreds of crashes

- Extended Binarly's internal program analysis framework to support us in this task



© BINARLY.IO

Binarly

Root Causes (*Excerpt*)

We found 29 unique root causes, 15 of which are likely exploitable

BRLY ID	CERT/CC ID	Affected IBV	Image Library	Impact	CVSS Score	CWE
BRLY-LOGOFAIL-2023-001	VU#811862	Insyde	BMP	DXE Memory Content Disclosure	Medium	CWE-200: Exposure of Sensitive Information
BRLY-LOGOFAIL-2023-007	VU#811862	Insyde	GIF	DXE Memory Corruption	High	CWE-122: Heap-based Buffer Overflow
BRLY-LOGOFAIL-2023-016	VU#811862	AMI	PNG	DXE Memory Corruption	High	CWE-122: Heap-based Buffer Overflow CWE-190: Integer Overflow
BRLY-LOGOFAIL-2023-022	VU#811862	AMI	JPEG	DXE Memory Corruption	High	CWE-787: Out-of-bounds Write
BRLY-LOGOFAIL-2023-025	VU#811862	Phoenix	BMP	DXE Memory Corruption	High	CWE-122: Heap-based Buffer Overflow
BRLY-LOGOFAIL-2023-029	VU#811862	Phoenix	GIF	DXE Memory Corruption	High	CWE-125: Out-of-bounds Read

© BINARLY.IO

BRLY-LOGOFAIL-2023-006: Memory Corruption

- PixelHeight and PixelWidth are attacker controlled
- When `PixelHeight` and `i` are 0: `BltBuffer[PixelWidth * -1]`
- Arbitrary write anywhere below `BltBuffer`

07 Open with Preview

```
PixelHeight = BmpHeader->PixelHeight;
EndOfBMP = 0;
for ( i = 0i64; i <= PixelHeight; ++i )
{
  if ( EndOfBMP )
    break;
  PixelWidth = BmpHeader->PixelWidth;
  v11 = 0i64;
  v12 = 0;
  // BRLY-LOGOFAIL-2023-003
  // when BmpHeader->PixelHeight is 0 Blt will be below BltBuffer
  // (0 - 0 - 1) * BmpHeader->PixelWidth = - BmpHeader->PixelWidth
  // then, writes to the Blt buffer will happen
  Blt = &BltBuffer[PixelWidth * (PixelHeight - i - 1)];
  do
  {
    if ( v12 )
      break;
    FirstByte = *RLE8Image;
    v15 = RLE8Image + 1;
    SecondByte = RLE8Image[1];
    RLE8Image += 2;
    if ( FirstByte )
    {
      Count = FirstByte;
      v11 += FirstByte;
      do
      {
        Blt->Red = BmpColorMap[SecondByte].Red;// arbitrary write
        Blt->Green = BmpColorMap[SecondByte].Green;// arbitrary write
        Blt->Blue = BmpColorMap[SecondByte].Blue;// arbitrary write
        ++Blt;
        --Count;
      }
      while ( Count );
    }
```

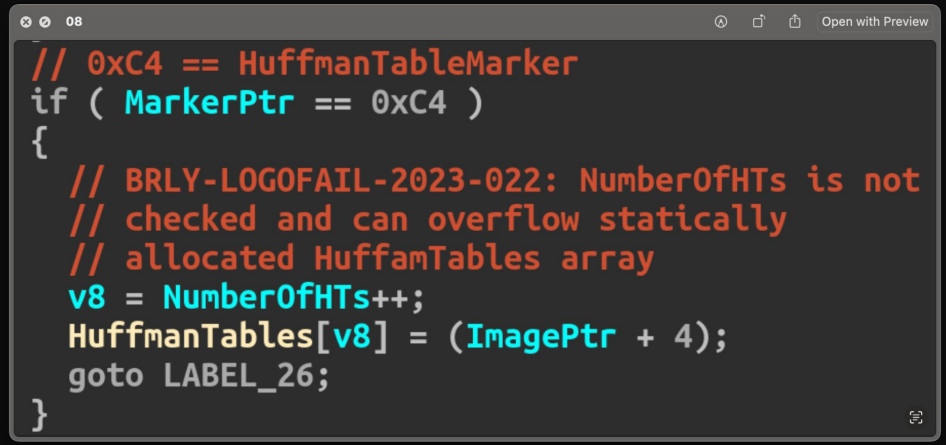
BMP parser developed by Insyde

© BINARLY.IO

Binarly

BRLY-LOGOFAIL-2023-022: Memory Corruption

- Assumption that JPEG can contain only 4 Huffman Tables
- `NumberOfHTs` variable is unchecked
- Overflow on global data with pointers to our image



JPEG parser developed by AMI

© BINARLY.IO

Binarly

Takeaways from Fuzzing

None of these libraries where ever fuzzed by IBVs/OEMs:

- We found crashes in every parser
- First crashes where found after seconds of fuzzing
- Some parsers even crash with images downloaded from the Internet :-)

© BINARLY.IO

Thanks to the Internet Archive!

- One of the parsers is for PCX images
- Finding good corpus for the fuzzer turned out to be more difficult than expected
- Until..

https://archive.org/details/Universe_Of_PCX_1700_PCX_Files

© BINARLY.IO

Binarly

Proof of Concept

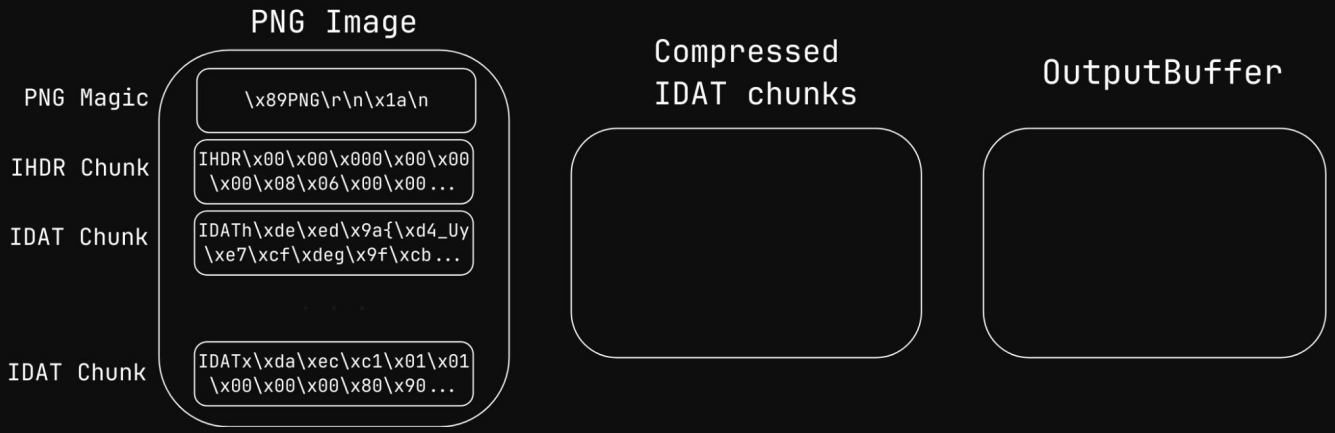
© BINARLY.IO

Let’s PWN a Real Device

- Lenovo ThinkCentre M70s Gen 2
- 11th Gen Intel Core (Tiger Lake)
- BIOS released on June 2023

© BINARLY.IO

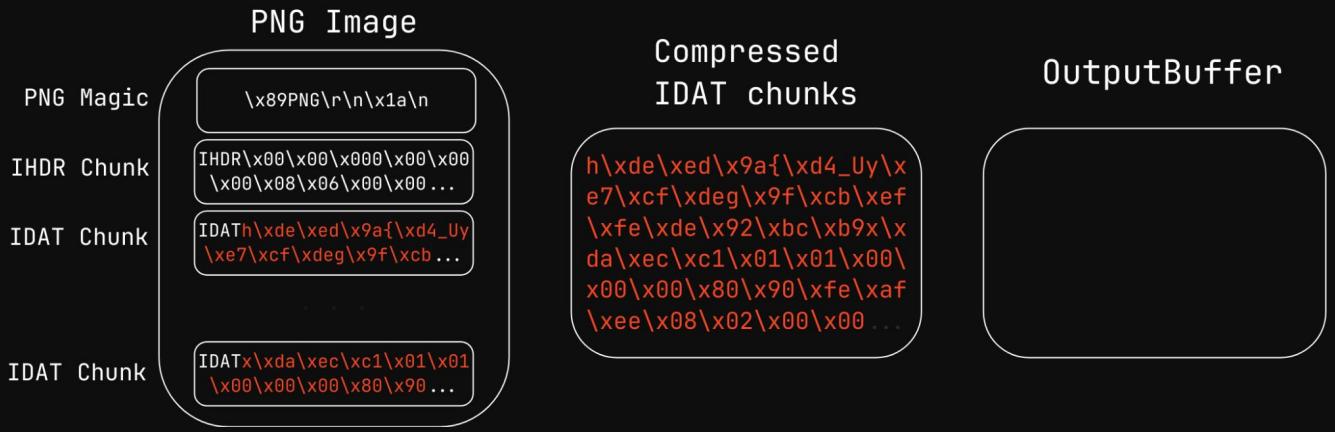
Selecting a Target



Simple format + exploitable crash: PNG parser from AMI

© BINARLY.IO

Selecting a Target

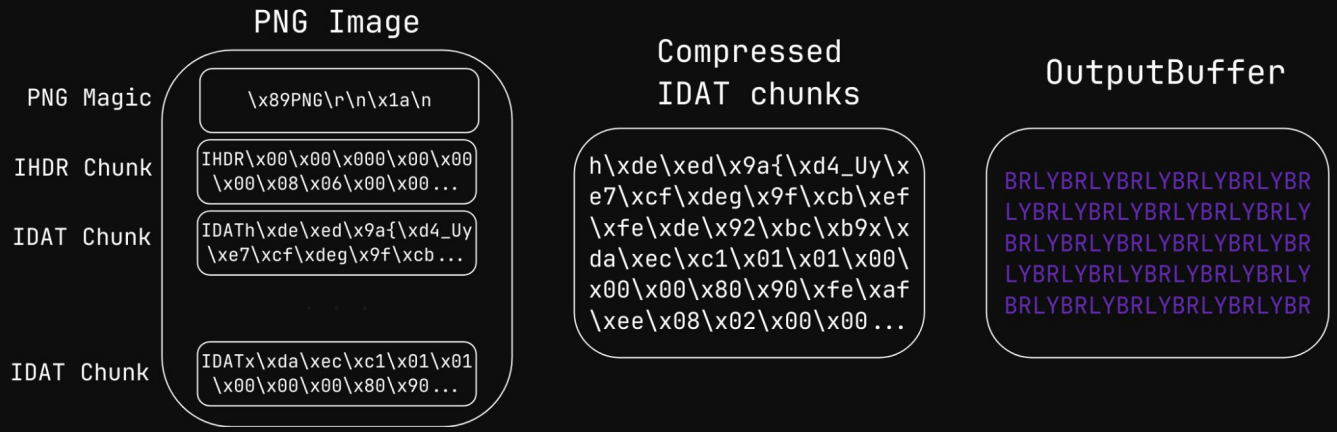


Simple format + exploitable crash: PNG parser from AMI

© BINARLY.IO

Binarly

Selecting a Target



Simple format + exploitable crash: PNG parser from AMI

© BINARLY.IO

Binarly

Integer Overflow to Heap Overflow

Integer overflow on 32 bit value used as allocation size:

- `2 * 0x20 = 0x40`
- `2 * 0x60 = 0xc0`
- `2 * 0x80000040 = 0x80`

```
// BRLY-LOGOFAIL-2023-016: Integer overflow
// on the argument of EfiLibAllocateZeroPool
OutputBuffer = EfiLibAllocateZeroPool(2 * PngWidth);
v7 = &OutputBuffer[PngWidth];
GlobalInfo.OutputBuffer = OutputBuffer;
```

Compressed IDAT chunks

```
h\xde\xed\x9a{\xd4_Uy\x
e7\xcf\xdeg\x9f\xcb\xef
\xfe\xde\x92\xbc\xb9x\x
da\xec\xc1\x01\x01\x00\
x00\x00\x80\x90\xfe\xaf
\xee\x08\x02\x00\x00...
```

OutputBuffer

```
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
```

© BINARLY.IO

Integer Overflow to Heap Overflow

Integer overflow on 32 bit value used as allocation size:

- `2 * 0x40` = `0x80`
- `2 * 0x60` = `0xc0`
- `2 * 0x80000040` = `0x80`

```
// BRLY-LOGOFAIL-2023-016: Integer overflow
// on the argument of EfiLibAllocateZeroPool
OutputBuffer = EfiLibAllocateZeroPool(2 * PngWidth);
v7 = &OutputBuffer[PngWidth];
GlobalInfo.OutputBuffer = OutputBuffer;
```

```
GlobalInfo.OutputBuffer[GlobalInfo.idx] = a1;
```

Compressed IDAT chunks

```
h\xde\xed\x9a{\xd4_Uy\x
e7\xcf\xdeg\x9f\xcb\xef
\xfe\xde\x92\xbc\xb9x\x
da\xec\xc1\x01\x01\x00\
x00\x00\x80\x90\xfe\xaf
\xee\x08\x02\x00\x00...
```

OutputBuffer

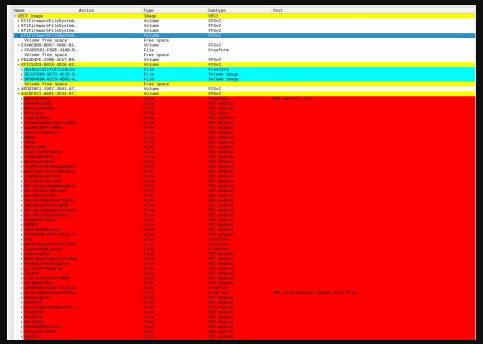
```
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
00000000000000000000
00000000000000000000
00000000000000000000
```

© BINARLY.IO

Binarly

Wait a Minute..

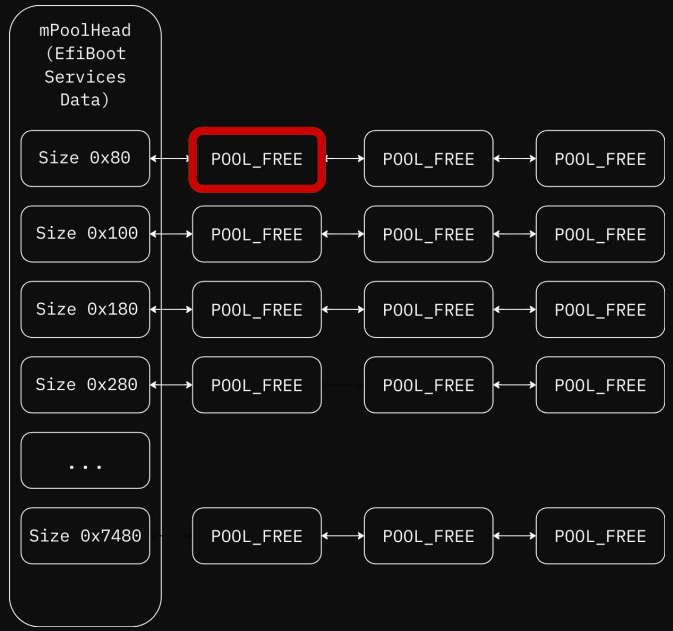
- How does heap exploitation even work for UEFI?
- No debugging capabilities:
  - Intel DCI doesn't work on new CPU models
  - Intel Boot Guard prevents replacing modules
- Not even output on crash :(



© BINARLY.IO

UEFI Heap Internals

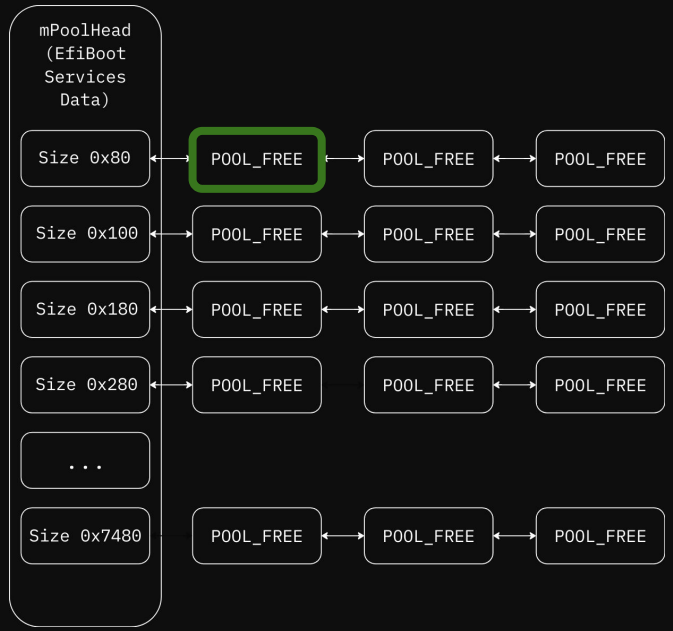
- Pool-based heap



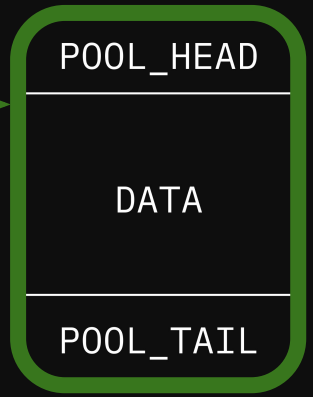
© BINARLY.IO

UEFI Heap Internals

- Pool-based heap




```
VOID *p = AllocatePool(0x40)
```

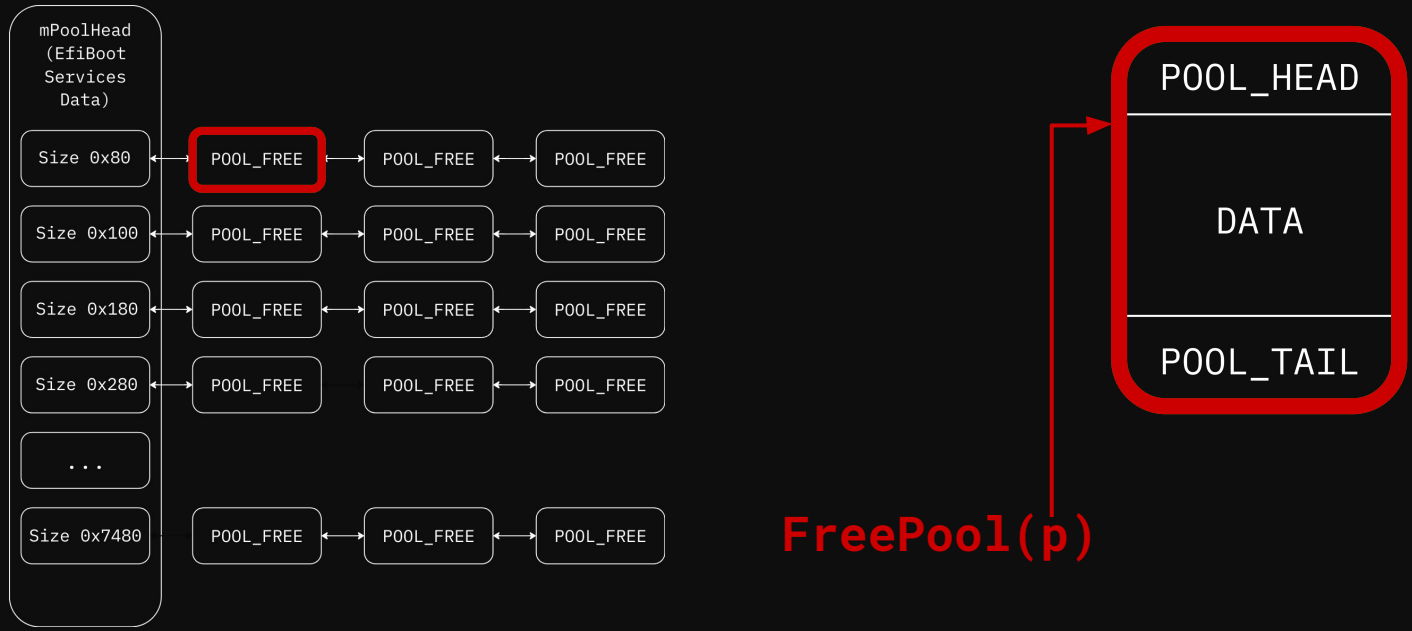


© BINARLY.IO

Binarly

UEFI Heap Internals

- Pool-based heap



© BINARLY.IO

Binarly

What Are We Even Corrupting?

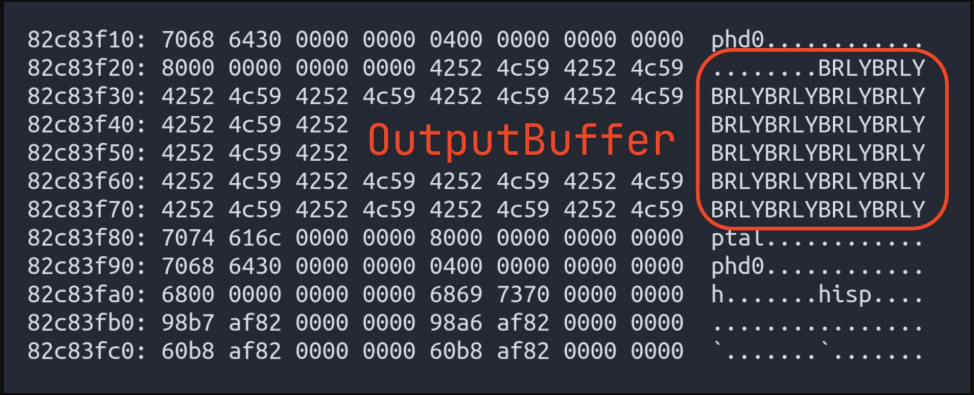
We don’t know!!

© BINARLY.IO

Binarly

Long Live UEFI Memory

- Memory used by UEFI is not cleared
- If the OS doesn't overwrite it, we can dump it after boot
- `OutputBuffer` is not freed, so it's somewhere in memory!

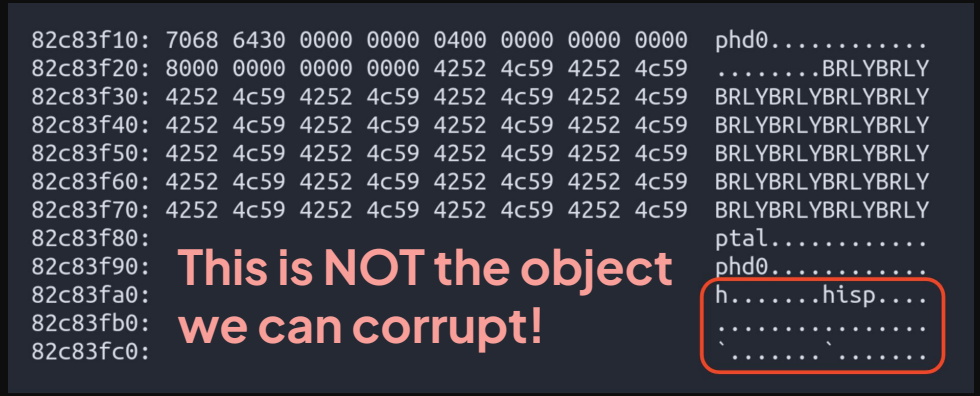


© BINARLY.IO

Binarly

Long Live UEFI Memory

- Memory used by UEFI is not cleared
- If the OS doesn't overwrite it, we can dump it after boot
- `OutputBuffer` is not freed, so it's somewhere in memory!

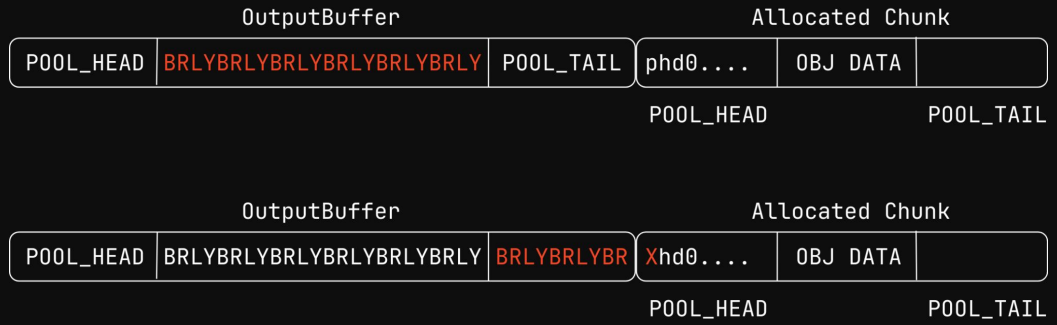


© BINARLY.IO

Binarly

Preserving Heap Chunks

- New technique to preserve chunks
- Corrupting the signature ensures a chunk is not reused




```
EFI_STATUS
CoreFreePoolI (
  IN VOID              *Buffer,
  OUT EFI_MEMORY_TYPE  *PoolType OPTIONAL
  )
{


  POOL_HEAD  *Head;
  ...

  ASSERT (Buffer != NULL);
  //
  // Get the head & tail of the pool entry
  //
  Head = BASE_CR (Buffer, POOL_HEAD, Data);
  ASSERT (Head != NULL);

  if ((Head->Signature != POOL_HEAD_SIGNATURE) &&
      (Head->Signature != POOLPAGE_HEAD_SIGNATURE))
  {
    ASSERT (
      Head->Signature == POOL_HEAD_SIGNATURE ||
      Head->Signature == POOLPAGE_HEAD_SIGNATURE
      );
    return EFI_INVALID_PARAMETER;
  }
```

© BINARLY.IO

Binarly

Preserving Heap Chunks

```
82c83f10: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f20: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f30: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f40: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f50: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f60: 4252 4c59 4252 4c59  4f4f 4f4f 4f4f 4f4f  BRLYBRLYOOOOOOOO
82c83f70:                                          OOOOOOOOXhd0....
82c83f80:                                          ........X.......
82c83f90:                                          prtn....iL.....
82c83fa0:                                          (.......(kL.....
82c83fb0:                                          .~........|.....
82c83fc0:                                          ptal....X.......
```

This IS the object we can corrupt!!

© BINARLY.IO

Binarly

Little Recap

What we achieved so far:

- We have arbitrary overflow on the heap
- We can prevent the next chunk from being freed
- We can inspect the object stored in the next chunk

What's left?

- Finding a good target for corruption
- Get code execution out of it

© BINARLY.IO

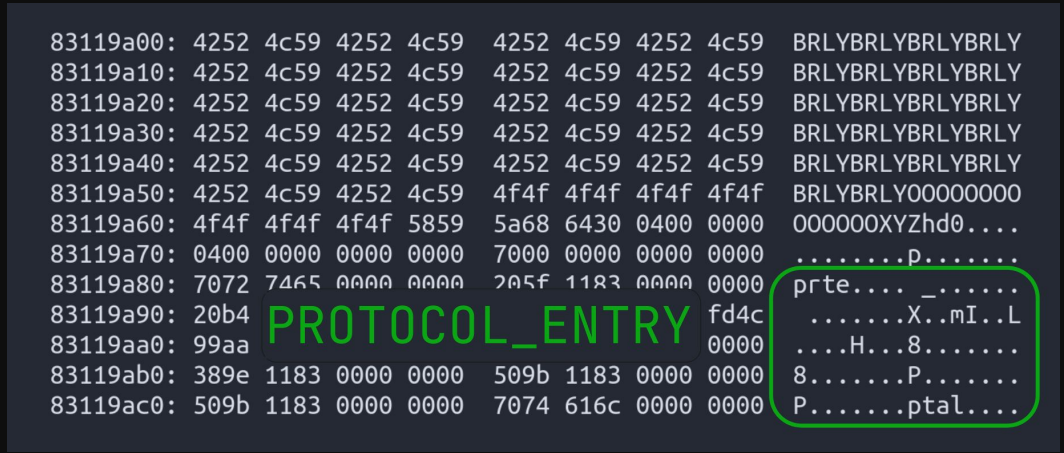
Enter the UEFI Heap Feng Shui

- Heap exploitation often requires strong allocation and deallocation primitives
- We can influence the heap by adding PNG chunks or changing their sizes

© BINARLY.IO

Enter the UEFI Heap Feng Shui

- Heap exploitation often requires strong allocation and deallocation primitives
- We can influence the heap by adding PNG chunks or changing their sizes

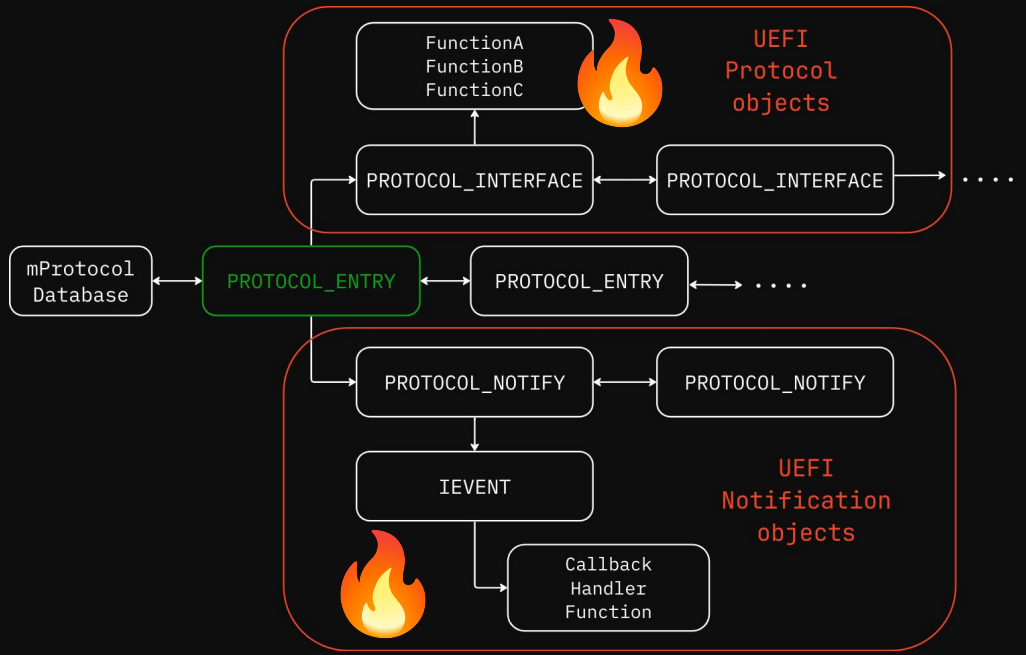


© BINARLY.IO

Binarly

PROTOCOL_ENTRY, tell me more..

- Protocols are a core concept in UEFI
- PROTOCOL_ENTRY has multiple pointers to objects with function pointers

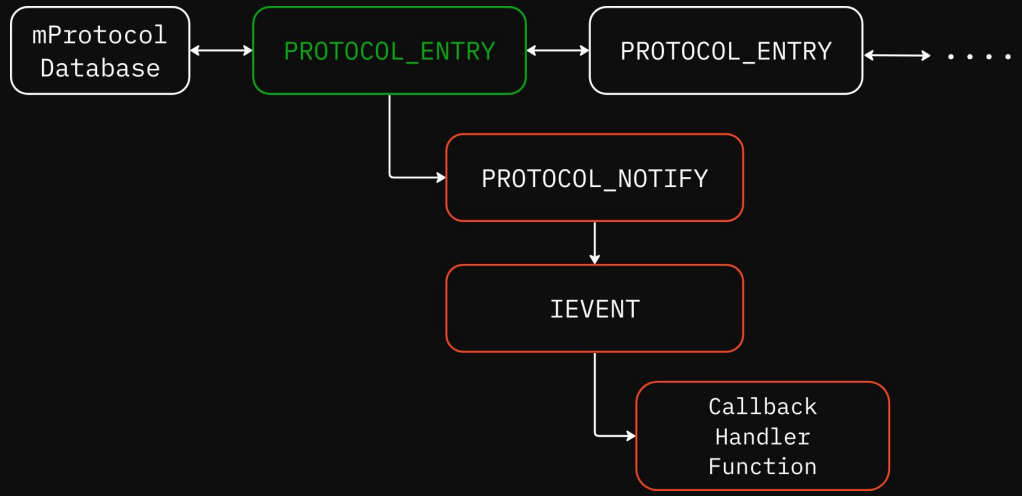


© BINARLY.IO

Binarly

UEFI Event System

- Events are generated when protocols are installed



© BINARLY.IO

Binarly

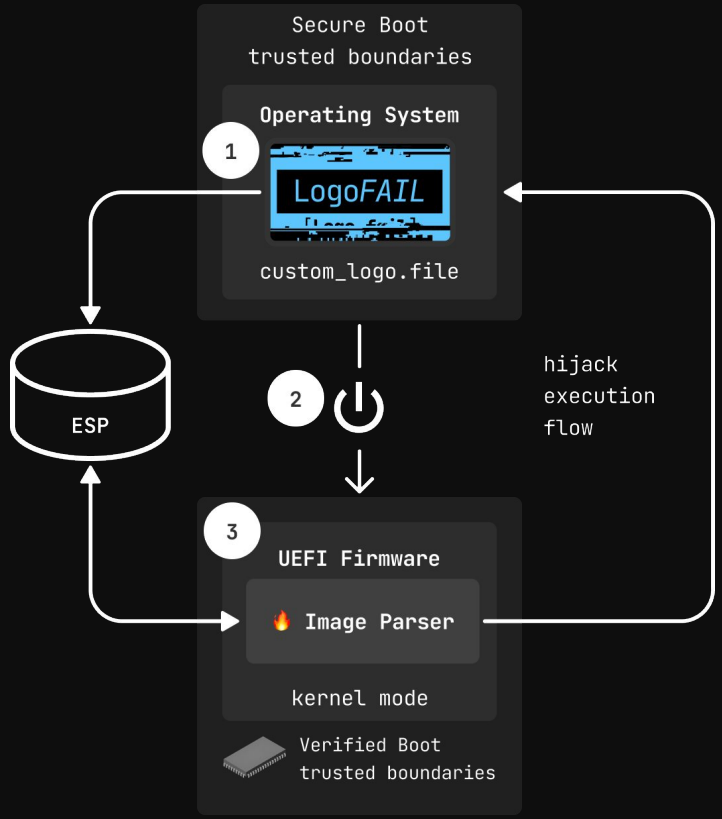
Arbitrary Code Exec in UEFI

- Memory region where NVRAM variables is often executable and always mapped at the same fixed address
- We can just store a shellcode there
- Our shellcode can:
  - Disable Secure Boot (zero a global variable)
  - Start a second-stage payload from disk:
    - Unload current NTFS driver (no write support)
    - Load new NTFS driver (with write support)
    - Creates a file on the Windows filesystem

© BINARLY.IO

Putting it All Together

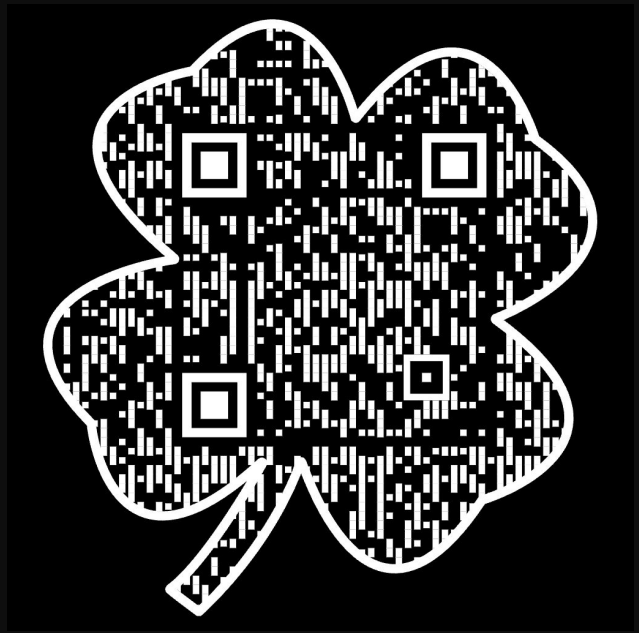
- Preparation:
  1. Malicious PNG on the ESP (or in NVRAM)
  2. `PROTOCOL_NOTIFY`, `IEVENT` and `Shellcode` in NVRAM
  3. Second-stage payload on disk: `\Users\user\LogoFAIL\SecondStageWin.efi`
- Reboot the system
- UEFI firmware will parse our PNG
- Heap overflow corrupts a `PROTOCOL_ENTRY` with pointers to `PROTOCOL_NOTIFY` and `IEVENT`
- When the protocol will be installed, we achieve arbitrary code execution
- Shellcode + Second stage payload execution



© BINARLY.IO

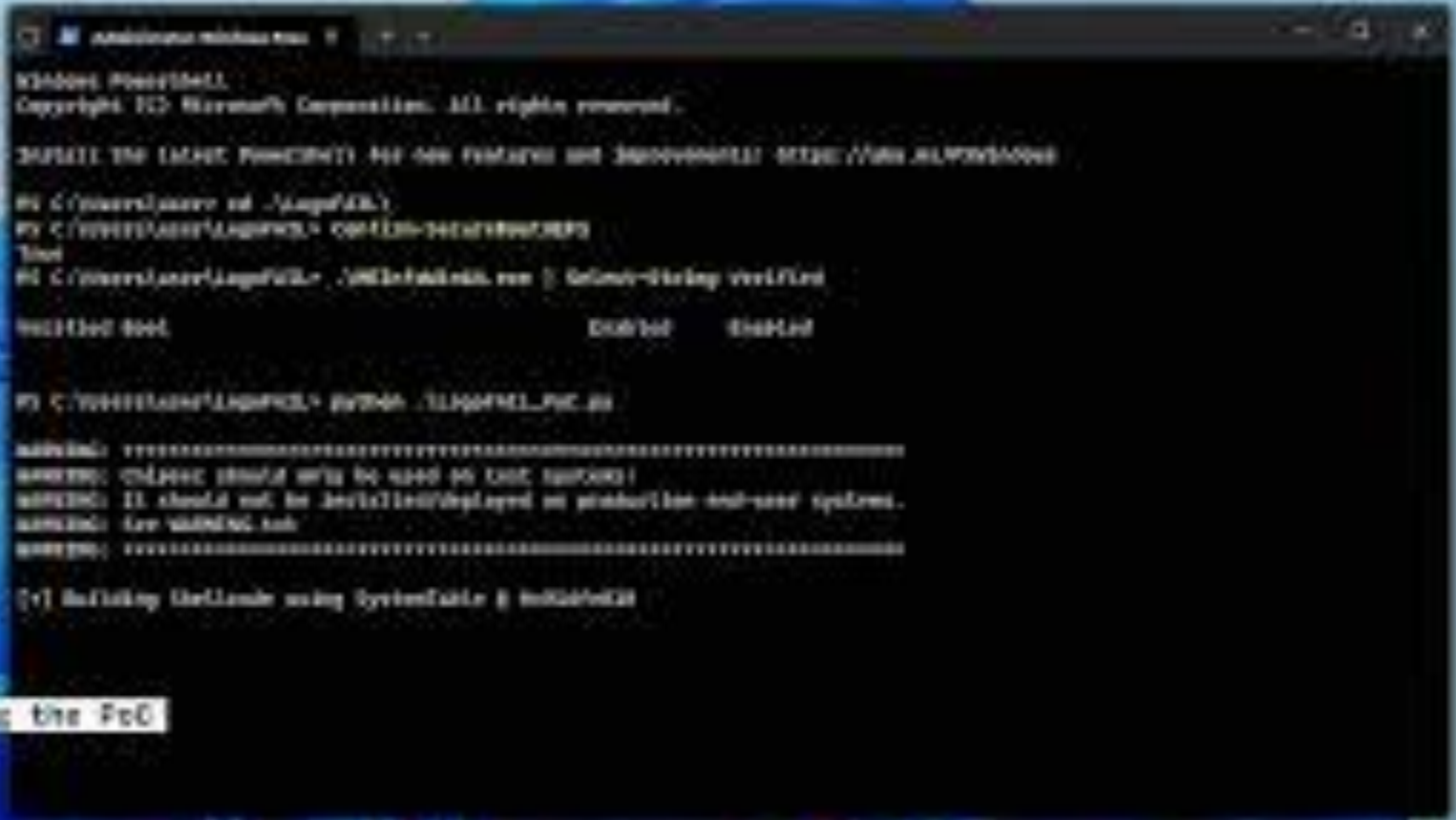
Binarly

Demo



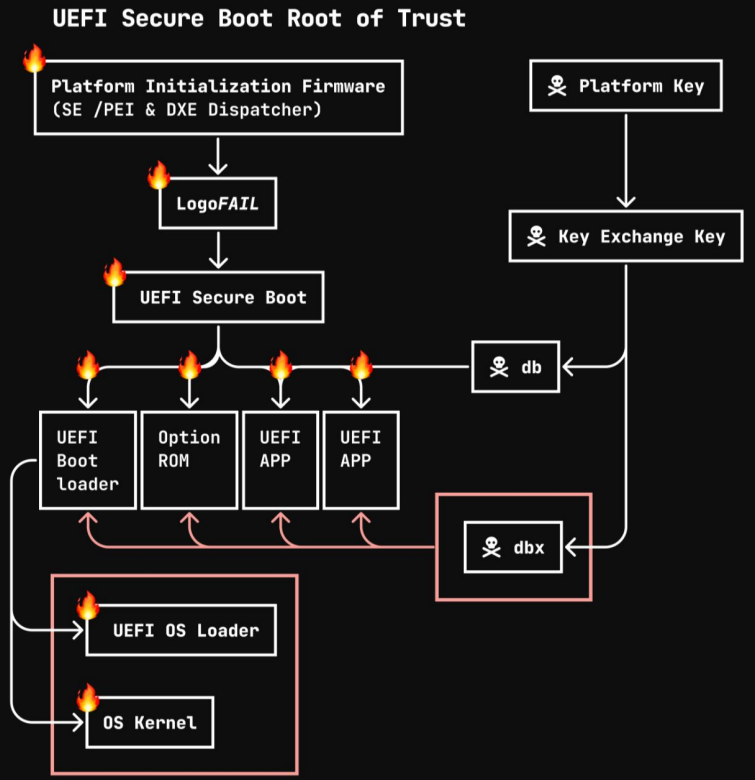
https://www.youtube.com/watch?v=EufeOPe6eqk

© BINARLY.IO


Running the PoC

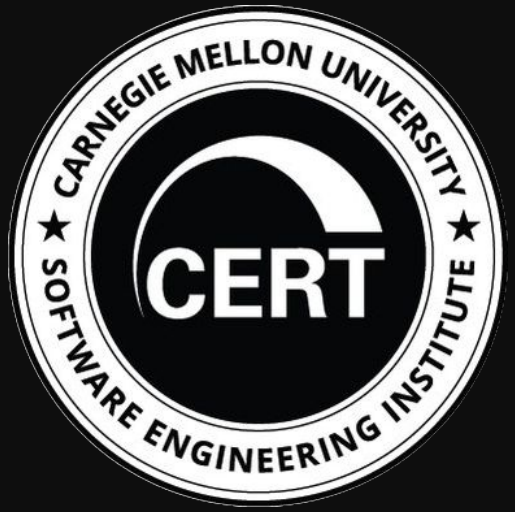
LogoFAIL

- Majority of UEFI firmware contains vulnerable images parsers
- Hundreds of devices from Lenovo, Intel and Acer allow logo customizations thus are exploitable
- Doesn't require any physical access to the device
- Targets UEFI specific code that affects both x86 and ARM devices
- Modern "below-the-OS" defenses, such as Secure Boot are completely ineffective against it



© BINARLY.IO

Binarly



Thanks to CERT/CC for coordinating this massive industry-wide disclosure! 🙏

© BINARLY.IO

Phoenix Technology 🤦

© BINARLY.IO

*https://webcache.googleusercontent.com/search?q=cache:cWlnW4oat9sJ:https://www.phoenix.com/security-notifications/cve-2023-5058/

Binarly

That’s all folks, thank you for your attention…

… and don’t forget to update your firmware!

© BINARLY.IO

Binarly