Logo*FAIL*

Security implications of image parsing during system boot

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

Binarly

black hat EUROPE 2023

$ 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 REsearch Team

LogoFAIL [edition]

Fabio Pagani
@pagabuc

Alex Matrosov
@matrosov

Yegor Vasilenko
@yeggorv

Alex Ermolov
@flothrone

Sam Thomas
@xorpse

Anton Ivanov
@ant_av7

Binarly

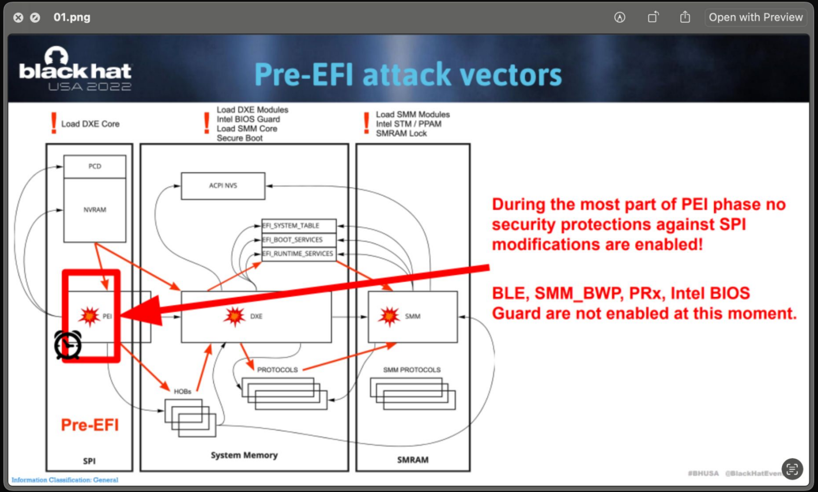
Scan

The Far-Reaching Consequences of LogoFAIL (Blog)

Inside the LogoFAIL Vulnerabilities (Video)

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**



Pre-EFI attack vectors

During the most part of PEI phase no security protections against SPI modifications are enabled!

BLE, SMM_BWP, PRx, Intel BIOS Guard are not enabled at this moment.

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

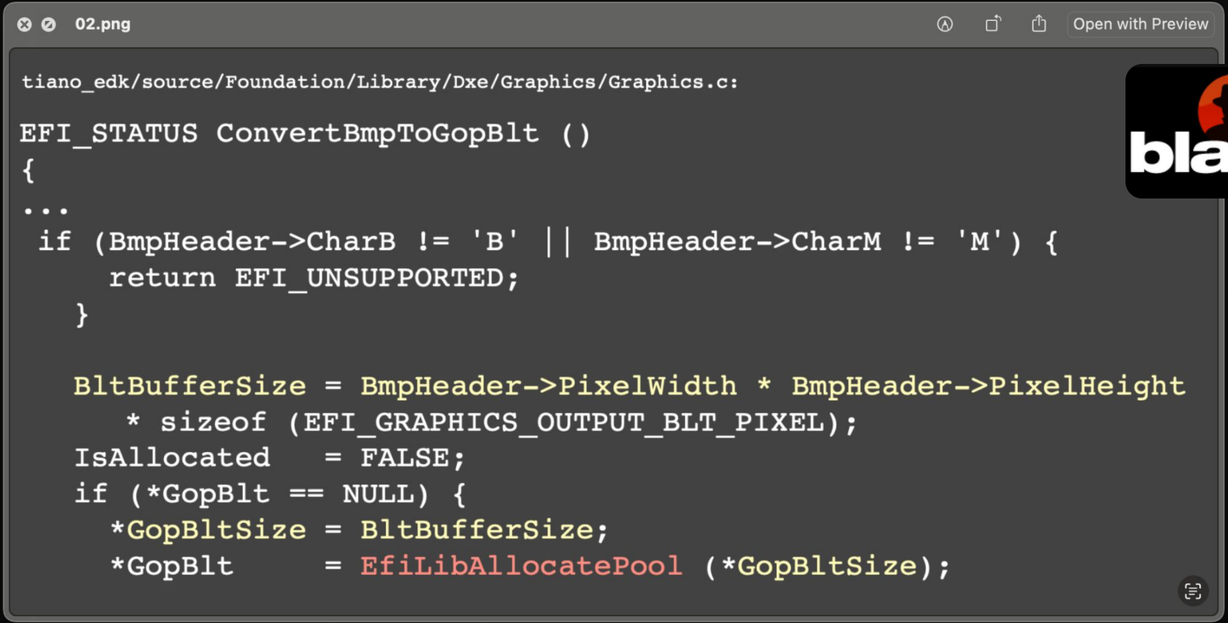
https://i.blackhat.com/USA-22/Wednesday/US-22-Matrosov-Breaking-Firmware-Trust-From-Pre-EFI.pdf

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?!

History Repeats Itself



```
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

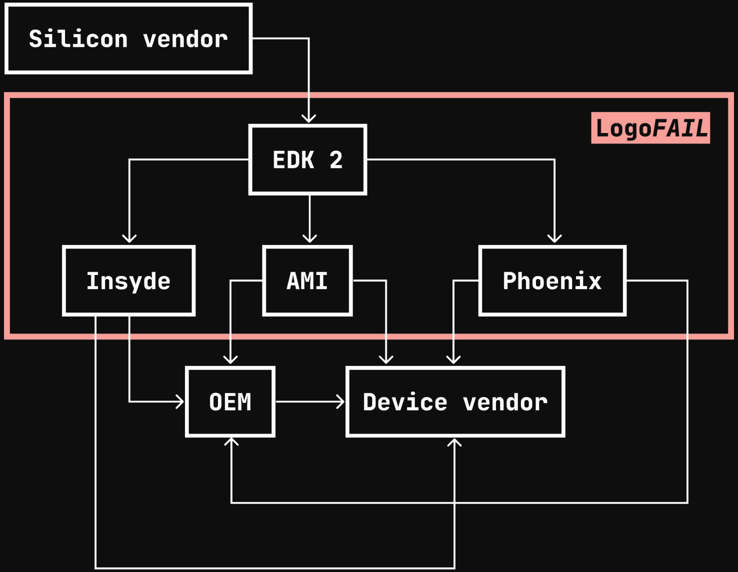
History Repeats Itself (~15 years later)

- 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?!

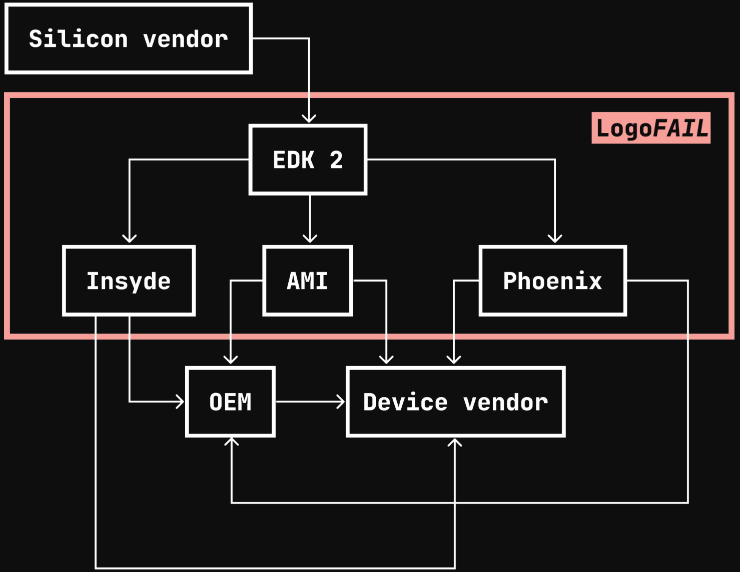
Meet Logo*FAIL*

- 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



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

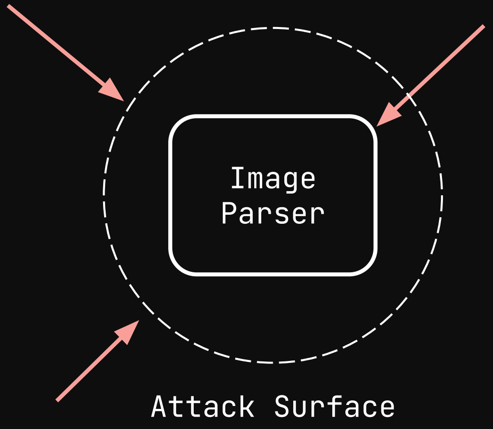


150+ days of embargo lifts TODAY

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

Attack Surface



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
```

insyde

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

phoenix technologies

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

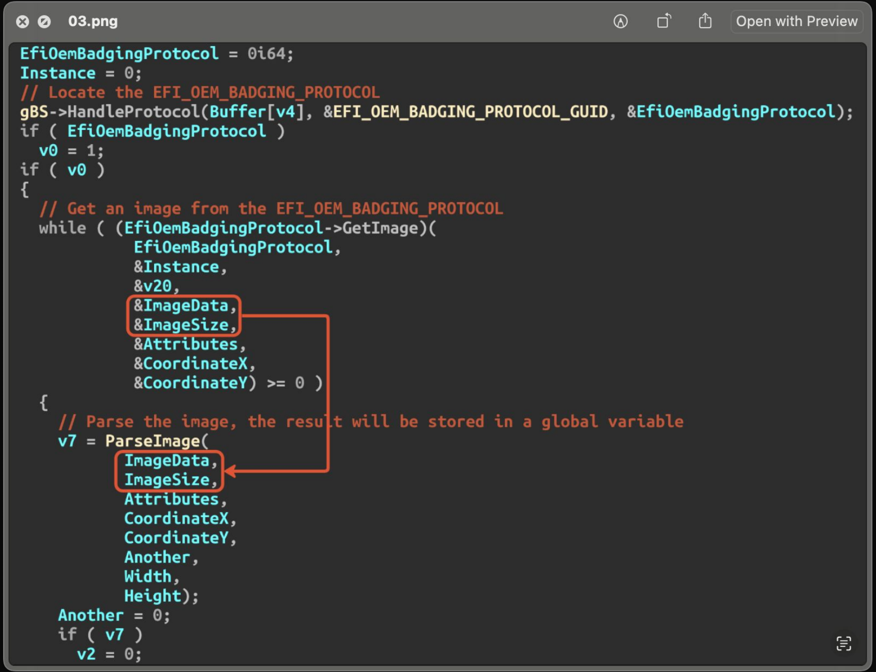
ami

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

tianocore

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”



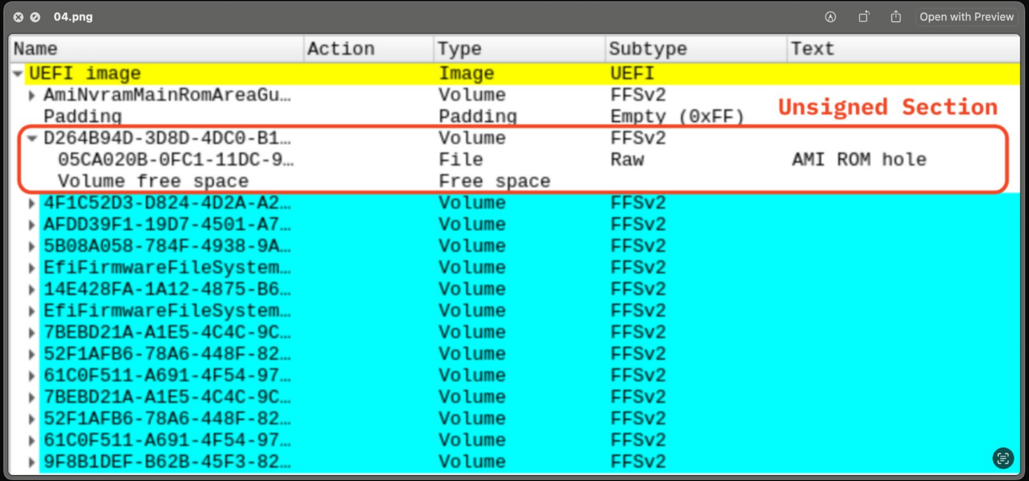
```
EfiOemBadgingProtocol = 0i64;
Instance = 0;
// Locate the EFI_OEM_BADGING_PROTOCOL
gBS->HandleProtocol(Buffer[v4], &EFI_OEM_BADGING_PROTOCOL_GUID, &EfiOemBadgingProtocol);
if ( EfiOemBadgingProtocol )
  v0 = 1;
if ( v0 )
{
  // Get an image from the EFI_OEM_BADGING_PROTOCOL
  while ( (EfiOemBadgingProtocol->GetImage)(
            EfiOemBadgingProtocol,
            &Instance,
            &v20,
            &ImageData,
            &ImageSize,
            &Attributes,
            &CoordinateX,
            &CoordinateY) >= 0 )
  {
    // Parse the image, the result will be stored in a global variable
    v7 = ParseImage(
           ImageData,
           ImageSize,
           Attributes,
           CoordinateX,
           CoordinateY,
           Another,
           Width,
           Height);
    Another = 0;
    if ( v7 )
      v2 = 0;
```

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

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



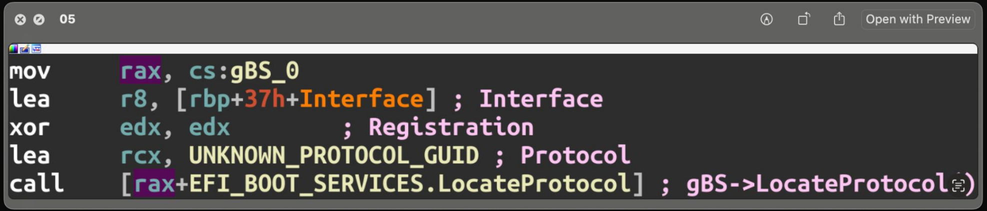
Name	Action	Type	Subtype	Text
UEFI image		Image	UEFI	
AmiNvramMainRomAreaGu…		Volume	FFSv2	
Padding		Padding	Empty (0xFF)	
D264B94D-3D8D-4DC0-B1…		Volume	FFSv2	
05CA020B-0FC1-11DC-9…		File	Raw	AMI ROM hole
Volume free space		Free space		
4F1C52D3-D824-4D2A-A2…		Volume	FFSv2	
AFDD39F1-19D7-4501-A7…		Volume	FFSv2	
5B08A058-784F-4938-9A…		Volume	FFSv2	
EfiFirmwareFileSystem…		Volume	FFSv2	
14E428FA-1A12-4875-B6…		Volume	FFSv2	
EfiFirmwareFileSystem…		Volume	FFSv2	
7BEBD21A-A1E5-4C4C-9C…		Volume	FFSv2	
52F1AFB6-78A6-448F-82…		Volume	FFSv2	
61C0F511-A691-4F54-97…		Volume	FFSv2	
7BEBD21A-A1E5-4C4C-9C…		Volume	FFSv2	
52F1AFB6-78A6-448F-82…		Volume	FFSv2	
61C0F511-A691-4F54-97…		Volume	FFSv2	
9F8B1DEF-B62B-45F3-82…		Volume	FFSv2	

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

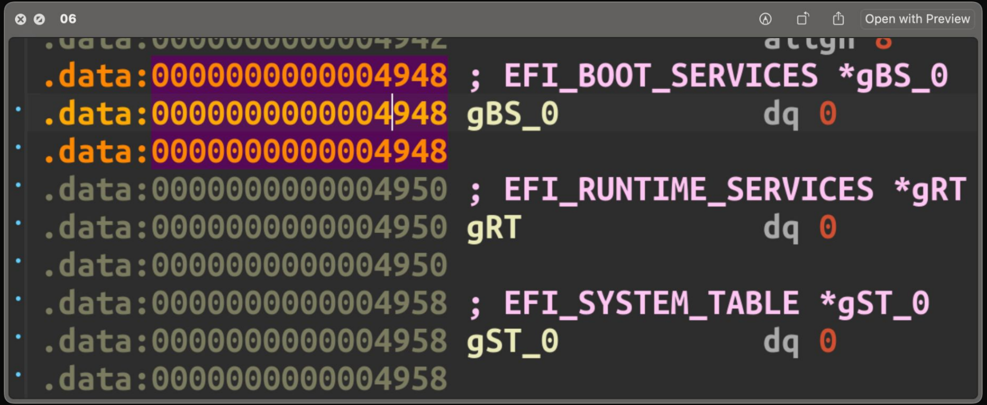
Fuzzing

Fuzzing UEFI Image Parsers

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



```
mov     rax, cs:gBS_0
lea     r8, [rbp+37h+Interface] ; Interface
xor     edx, edx        ; Registration
lea     rcx, UNKNOWN_PROTOCOL_GUID ; Protocol
call    [rax+EFI_BOOT_SERVICES.LocateProtocol] ; gBS->LocateProtocol()
```



```
.data:0000000000004948 ; EFI_BOOT_SERVICES *gBS_0
.data:0000000000004948 gBS_0           dq 0
.data:0000000000004948
.data:0000000000004950 ; EFI_RUNTIME_SERVICES *gRT
.data:0000000000004950 gRT             dq 0
.data:0000000000004950
.data:0000000000004958 ; EFI_SYSTEM_TABLE *gST_0
.data:0000000000004958 gST_0           dq 0
.data:0000000000004958
```

Fuzzing Harness

A bridge between the fuzzer and the fuzzed module:

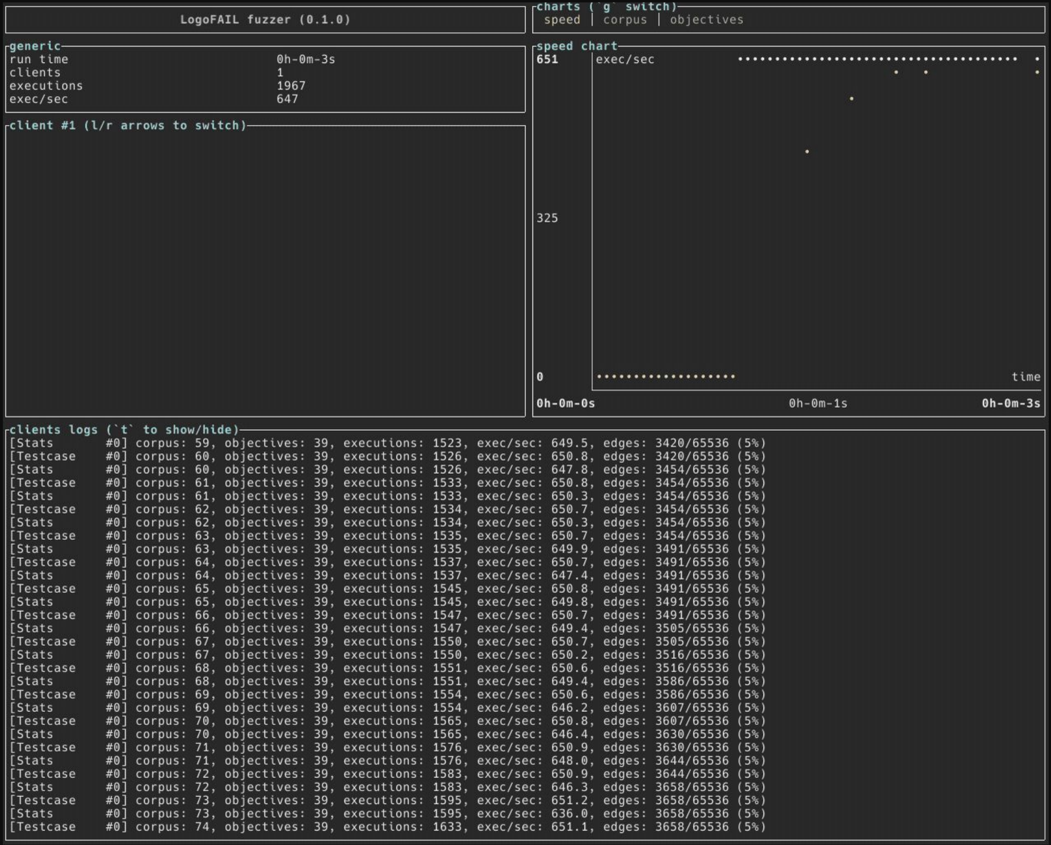
- Module initialization (protocols are installed)
- Prepare call to parsing function
- Forwards fuzzer-generated data to the target module

We are ready to fuzz!

Root Causes

- We found hundreds of crashes

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



```
generic
run time                              0h-0m-3s
clients                               1
executions                            1967
exec/sec                              647
```

```
clients logs (`t` to show/hide)
[Stats       #0] corpus: 59, objectives: 39, executions: 1523, exec/sec: 649.5, edges: 3420/65536 (5%)
[Testcase    #0] corpus: 60, objectives: 39, executions: 1526, exec/sec: 650.8, edges: 3420/65536 (5%)
[Stats       #0] corpus: 60, objectives: 39, executions: 1526, exec/sec: 647.8, edges: 3454/65536 (5%)
[Testcase    #0] corpus: 61, objectives: 39, executions: 1533, exec/sec: 650.8, edges: 3454/65536 (5%)
[Stats       #0] corpus: 61, objectives: 39, executions: 1533, exec/sec: 650.3, edges: 3454/65536 (5%)
[Testcase    #0] corpus: 62, objectives: 39, executions: 1534, exec/sec: 650.7, edges: 3454/65536 (5%)
[Stats       #0] corpus: 62, objectives: 39, executions: 1534, exec/sec: 650.3, edges: 3454/65536 (5%)
[Testcase    #0] corpus: 63, objectives: 39, executions: 1535, exec/sec: 650.7, edges: 3454/65536 (5%)
[Stats       #0] corpus: 63, objectives: 39, executions: 1535, exec/sec: 649.9, edges: 3491/65536 (5%)
[Testcase    #0] corpus: 64, objectives: 39, executions: 1537, exec/sec: 650.7, edges: 3491/65536 (5%)
[Stats       #0] corpus: 64, objectives: 39, executions: 1537, exec/sec: 647.4, edges: 3491/65536 (5%)
[Testcase    #0] corpus: 65, objectives: 39, executions: 1545, exec/sec: 650.8, edges: 3491/65536 (5%)
[Stats       #0] corpus: 65, objectives: 39, executions: 1545, exec/sec: 649.8, edges: 3491/65536 (5%)
[Testcase    #0] corpus: 66, objectives: 39, executions: 1547, exec/sec: 650.7, edges: 3491/65536 (5%)
[Stats       #0] corpus: 66, objectives: 39, executions: 1547, exec/sec: 649.4, edges: 3505/65536 (5%)
[Testcase    #0] corpus: 67, objectives: 39, executions: 1550, exec/sec: 650.7, edges: 3505/65536 (5%)
[Stats       #0] corpus: 67, objectives: 39, executions: 1550, exec/sec: 650.2, edges: 3516/65536 (5%)
[Testcase    #0] corpus: 68, objectives: 39, executions: 1551, exec/sec: 650.6, edges: 3516/65536 (5%)
[Stats       #0] corpus: 68, objectives: 39, executions: 1551, exec/sec: 649.4, edges: 3586/65536 (5%)
[Testcase    #0] corpus: 69, objectives: 39, executions: 1554, exec/sec: 650.6, edges: 3586/65536 (5%)
[Stats       #0] corpus: 69, objectives: 39, executions: 1554, exec/sec: 646.2, edges: 3607/65536 (5%)
[Testcase    #0] corpus: 70, objectives: 39, executions: 1565, exec/sec: 650.8, edges: 3607/65536 (5%)
[Stats       #0] corpus: 70, objectives: 39, executions: 1565, exec/sec: 646.4, edges: 3630/65536 (5%)
[Testcase    #0] corpus: 71, objectives: 39, executions: 1576, exec/sec: 650.9, edges: 3630/65536 (5%)
[Stats       #0] corpus: 71, objectives: 39, executions: 1576, exec/sec: 648.0, edges: 3644/65536 (5%)
[Testcase    #0] corpus: 72, objectives: 39, executions: 1583, exec/sec: 650.9, edges: 3644/65536 (5%)
[Stats       #0] corpus: 72, objectives: 39, executions: 1583, exec/sec: 646.3, edges: 3658/65536 (5%)
[Testcase    #0] corpus: 73, objectives: 39, executions: 1595, exec/sec: 651.2, edges: 3658/65536 (5%)
[Stats       #0] corpus: 73, objectives: 39, executions: 1595, exec/sec: 636.0, edges: 3658/65536 (5%)
[Testcase    #0] corpus: 74, objectives: 39, executions: 1633, exec/sec: 651.1, edges: 3658/65536 (5%)
```

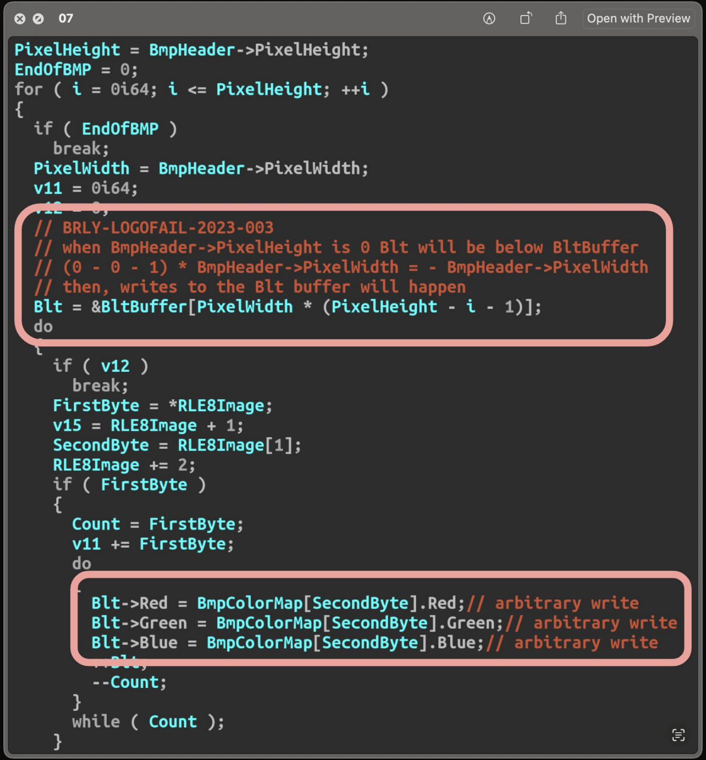
Root Causes (*Excerpt*)

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

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`

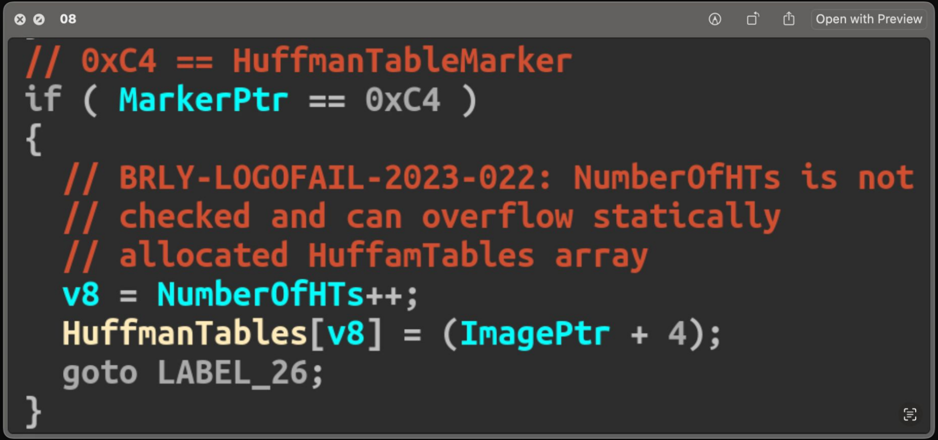


```
PixelHeight = BmpHeader->PixelHeight;
EndOfBMP = 0;
for ( i = 0i64; i <= PixelHeight; ++i )
{
  if ( EndOfBMP )
    break;
  PixelWidth = BmpHeader->PixelWidth;
  v11 = 0i64;
  v12 = 0;
  // BRLY-LOGOFAIL-2023-003
  // when BmpHeader->PixelHeight is 0 Blt will be below BltBuffer
  // (0 - 0 - 1) * 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

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



```
// 0xC4 == HuffmanTableMarker
if ( MarkerPtr == 0xC4 )
{
  // BRLY-LOGOFAIL-2023-022: NumberOfHTs is not
  // checked and can overflow statically
  // allocated HuffamTables array
  v8 = NumberOfHTs++;
  HuffmanTables[v8] = (ImagePtr + 4);
  goto LABEL_26;
}
```

JPEG parser developed by AMI

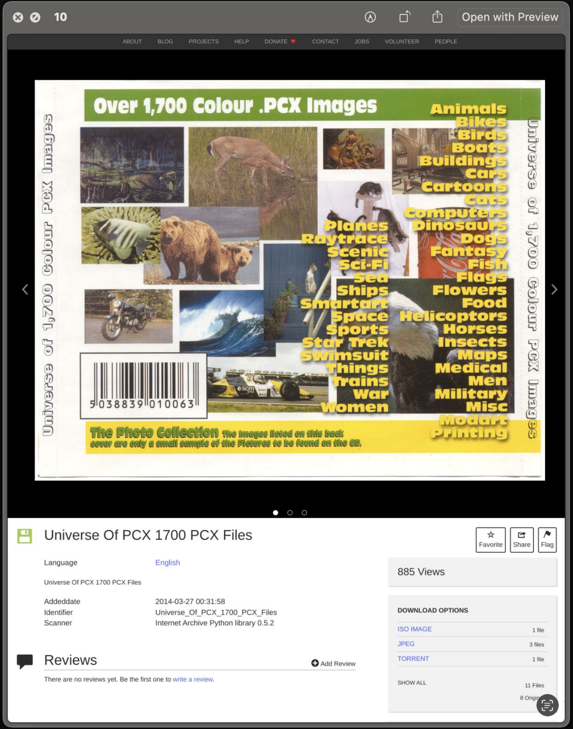
Takeaways from Fuzzing

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 :-)

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..



Universe Of PCX 1700 PCX Files

Language English

Universe Of PCX 1700 PCX Files

Addeddate 2014-03-27 00:31:58
Identifier Universe_Of_PCX_1700_PCX_Files
Scanner Internet Archive Python library 0.5.2

Reviews

Add Review

There are no reviews yet. Be the first one to write a review.

885 Views

DOWNLOAD OPTIONS

ISO IMAGE	1 file
JPEG	3 files
TORRENT	1 file
SHOW ALL	11 Files
	8 Origin

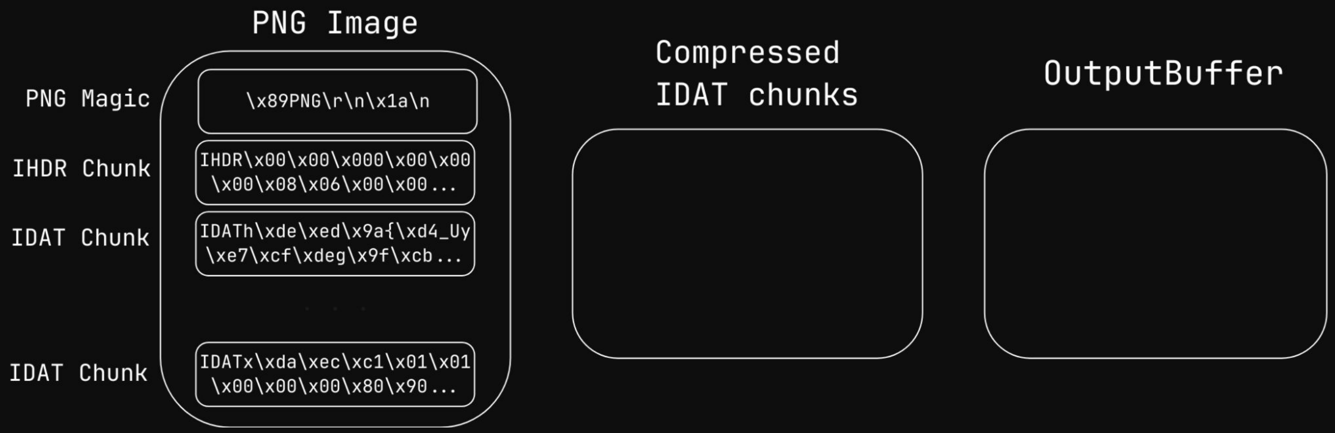
https://archive.org/details/Universe_Of_PCX_1700_PCX_Files

Proof of concept

Let’s PWN a Real Device

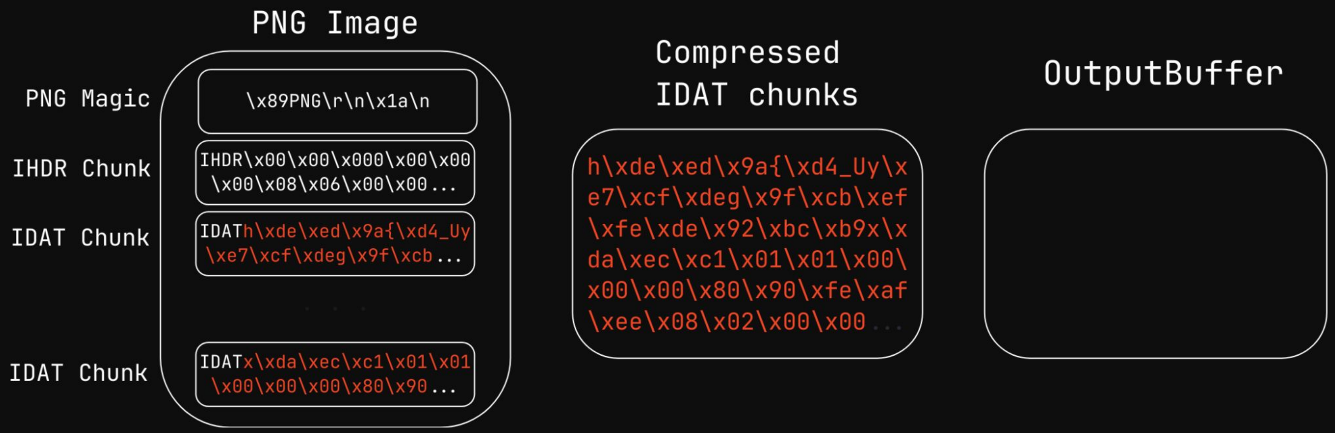
- Lenovo ThinkCentre M70s Gen 2
- 11th Gen Intel Core (Tiger Lake)
- BIOS released on June 2023

Selecting a Target



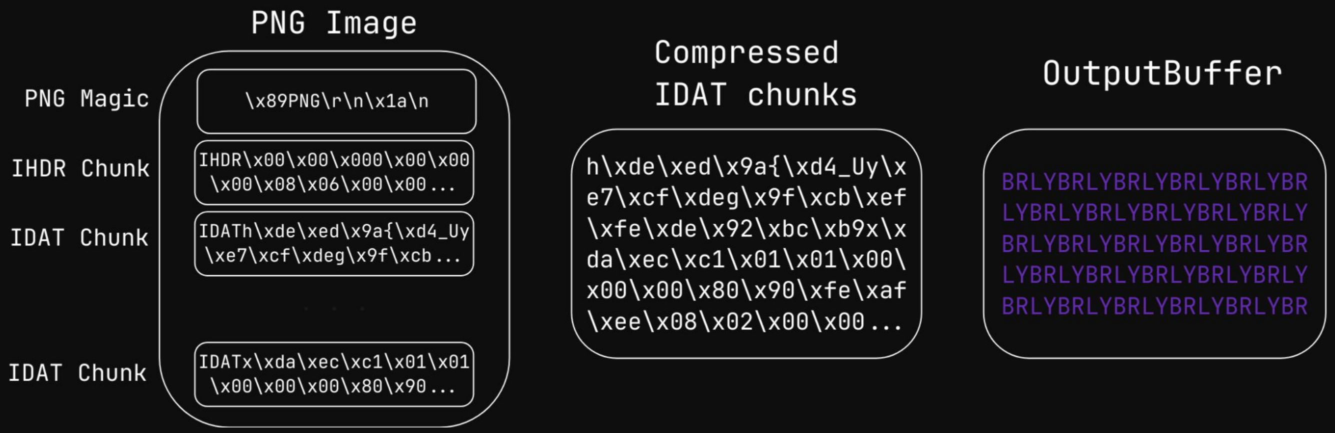
Simple format + exploitable crash: PNG parser from AMI

Selecting a Target


PNG Image
PNG Magic
\x89PNG\r\n\x1a\n
IHDR Chunk
IHDR\x00\x00\x000\x00\x00 \x00\x08\x06\x00\x00...
IDAT Chunk
IDATh\xde\xed\x9a{\xd4_Uy \xe7\xcf\xdeg\x9f\xcb...
IDAT Chunk
IDATx\xda\xec\xc1\x01\x01 \x00\x00\x00\x80\x90...
Compressed IDAT chunks
h\xde\xed\x9a{\xd4_Uy\xe7\xcf\xdeg\x9f\xcb\xef\xfe\xde\x92\xbc\xb9x\xda\xec\xc1\x01\x01\x00\x00\x00\x80\x90\xfe\xaf\xee\x08\x02\x00\x00...
OutputBuffer

Simple format + exploitable crash: PNG parser from AMI

Selecting a Target

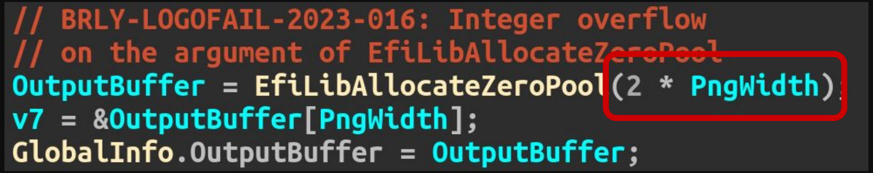

PNG Image
PNG Magic
\x89PNG\r\n\x1a\n
IHDR Chunk
IHDR\x00\x00\x000\x00\x00 \x00\x08\x06\x00\x00...
IDAT Chunk
IDATh\xde\xed\x9a{\xd4_Uy \xe7\xcf\xdeg\x9f\xcb...
IDAT Chunk
IDATx\xda\xec\xc1\x01\x01 \x00\x00\x00\x80\x90...
Compressed IDAT chunks
h\xde\xed\x9a{\xd4_Uy\xe7\xcf\xdeg\x9f\xcb\xef\xfe\xde\x92\xbc\xb9x\xda\xec\xc1\x01\x01\x00\x00\x00\x80\x90\xfe\xaf\xee\x08\x02\x00\x00...
OutputBuffer
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR

Simple format + exploitable crash: PNG parser from AMI

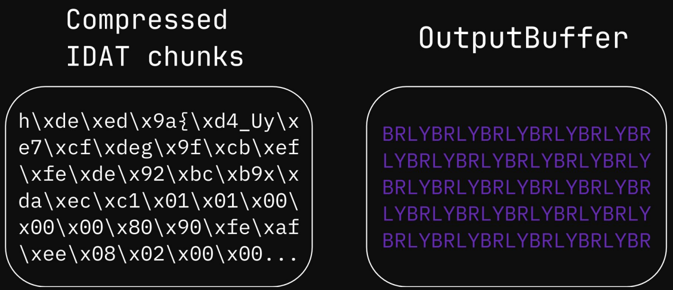
Integer Overflow to Heap Overflow

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\xe7\xcf\xdeg\x9f\xcb\xef\xfe\xde\x92\xbc\xb9x\xda\xec\xc1\x01\x01\x00\x00\x00\x80\x90\xfe\xaf\xee\x08\x02\x00\x00...
OutputBuffer
BRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBRLYBR

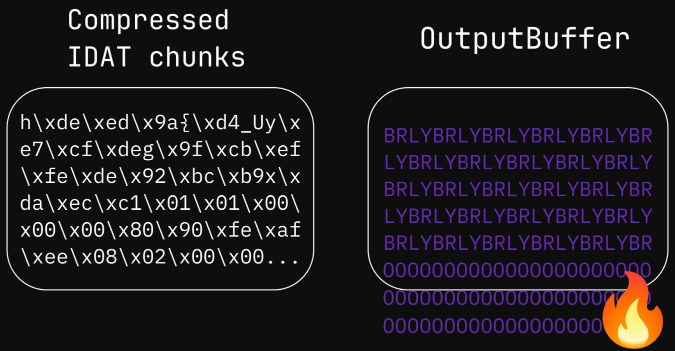
Integer Overflow to Heap Overflow

Integer overflow on 32 bit value used as allocation size:

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

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

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

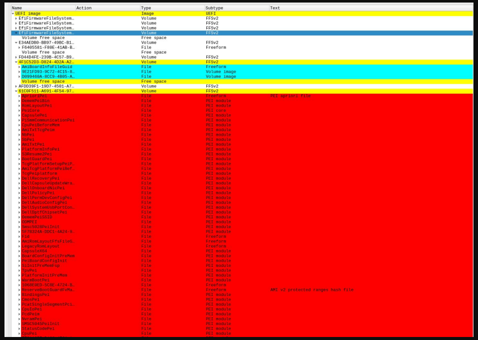


```
h\xde\xed\x9a{\xd4_Uy\x
e7\xcf\xdeg\x9f\xcb\xef
\xfe\xde\x92\xbc\xb9x\x
da\xec\xc1\x01\x01\x00\
x00\x00\x80\x90\xfe\xaf
\xee\x08\x02\x00\x00...
```

```
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
LYBRLYBRLYBRLYBRLYBRLY
BRLYBRLYBRLYBRLYBRLYBR
0000000000000000000000
0000000000000000000000
000000000000000000
```

Wait a Minute..

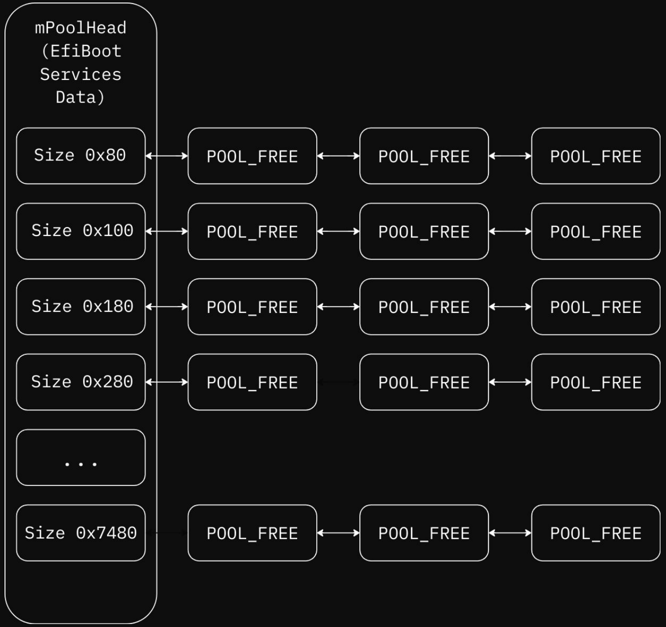
- How does heap exploitation even work for UEFI?
- No debugging capabilities:
  - Intel DCI doesn't work on new CPU models
  - Intel Boot Guard prevents replacing modules
- Not even output on crash :(



Name	Action	Type	Subtype	Text
UEFI image		Image	UEFI	
EfiFirmwareFileSystem…		Volume	FFSv2	
EfiFirmwareFileSystem…		Volume	FFSv2	
EfiFirmwareFileSystem…		Volume	FFSv2	
EfiFirmwareFileSystem…		Volume	FFSv2	
Volume free space		Free space		
E34AEDB0-BB97-49BC-B1…		Volume	FFSv2	
F6405581-F80E-41AB-B…		File	Freeform	
Volume free space		Free space		
FD44D4FE-239B-4C57-B9…		Volume	FFSv2	
4F1C52D3-D824-4D2A-A2…		Volume	FFSv2	
AmiBoardInfoFileGuid		File	Freeform	
9E21FD93-9C72-4C15-8…		File	Volume image	
D099460A-8CC9-4B05-A…		File	Volume image	
Volume free space		Free space		
AFDD39F1-19D7-4501-A7…		Volume	FFSv2	
61C0F511-A691-4F54-97…		Volume	FFSv2	
AprioriPei		File	Freeform	PEI apriori file
OememPeiBin		File	PEI module	
RomLayoutPei		File	PEI module	
PeiCore		File	PEI core	
CapsulePei		File	PEI module	
PiSmmCommunicationPei		File	PEI module	
CpuPeiBeforeMem		File	PEI module	
AmiTxtTcgPeim		File	PEI module	
NbPei		File	PEI module	
SbPei		File	PEI module	
AmiTxtPei		File	PEI module	
PlatformInfoPei		File	PEI module	
S3Resume2Pei		File	PEI module	
BootGuardPei		File	PEI module	
TcgPlatformSetupPeiP…		File	PEI module	
AmiTcgPlatformPeiBef…		File	PEI module	
TcgPeiplatform		File	PEI module	
DellRecoveryPei		File	PEI module	
DellCapsuleUpdateWra…		File	PEI module	
DellOnboardNicPei		File	PEI module	
DellPolicyPei		File	PEI module	
DellPermDevConfigPei		File	PEI module	
DellAudioConfigPei		File	PEI module	
DellSystemUsbPortCon…		File	PEI module	
DellDptfChipsetPei		File	PEI module	
OememPeiSSID		File	PEI module	
ODMPEI		File	PEI module	
Smsc5028PeiInit		File	PEI module	
5F78324A-DDC1-4A24-9…		File	PEI module	
Fid		File	Freeform	
AmiRomLayoutFfsFileG…		File	Freeform	
LegacyRomLayout		File	Freeform	
CapsuleX64		File	PEI module	
BoardConfigInitPreMem		File	PEI module	
PeiBoardConfigInit		File	PEI module	
SiInitPreMemFsp		File	PEI module	
TpvPei		File	PEI module	
PlatformInitPreMem		File	PEI module	
WarmBootPei		File	PEI module	
1068E0ED-5C8E-4724-B…		File	Freeform	
ReserveBootGuardFvMa…		File	Freeform	AMI v2 protected ranges hash file
BindingsPei		File	PEI module	
CmosPei		File	PEI module	
PcatSingleSegmentPci…		File	PEI module	
CpuIoPei		File	PEI module	
PcdPeim		File	PEI module	
NvramPei		File	PEI module	
SMSC5045PeiInit		File	PEI module	
StatusCodePei		File	PEI module	
CpuPei		File	PEI module	

UEFI Heap Internals

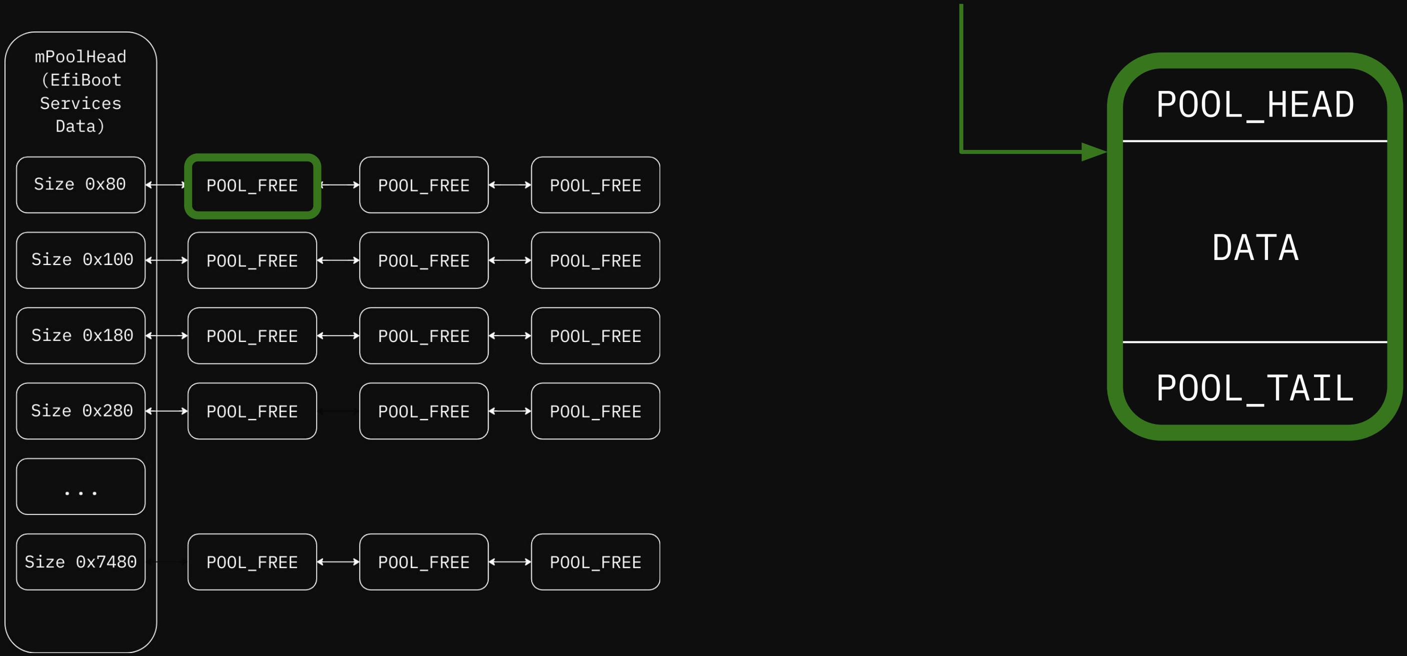
- Pool-based heap



UEFI Heap Internals

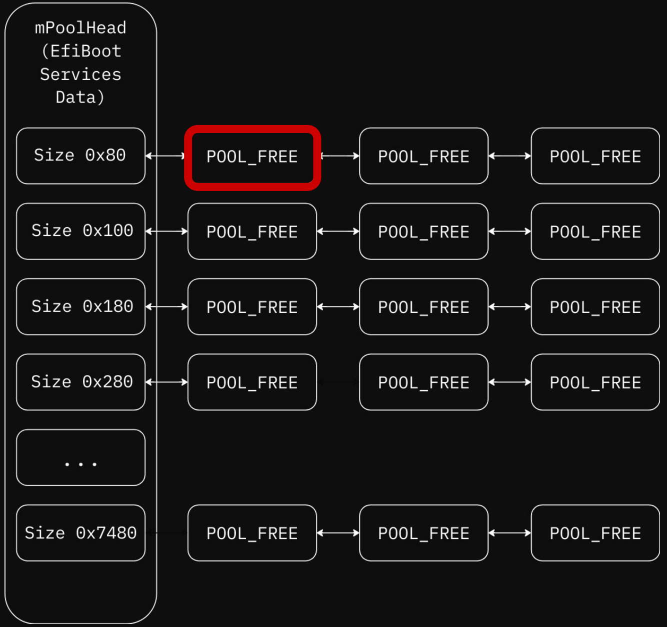
- Pool-based heap

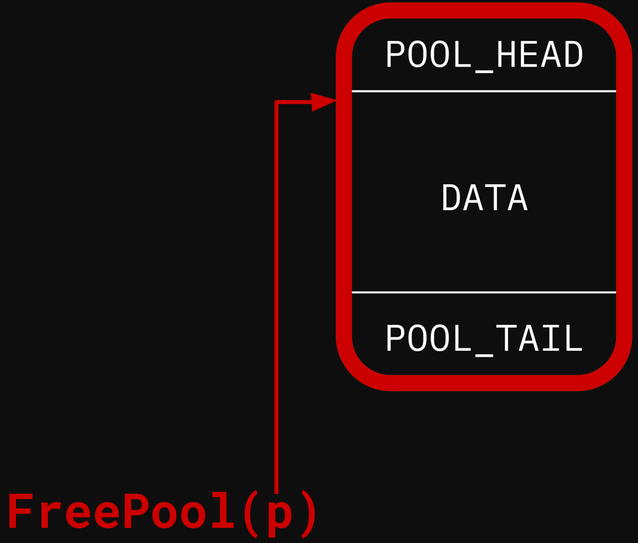
```
VOID *p = AllocatePool(0x40)
```



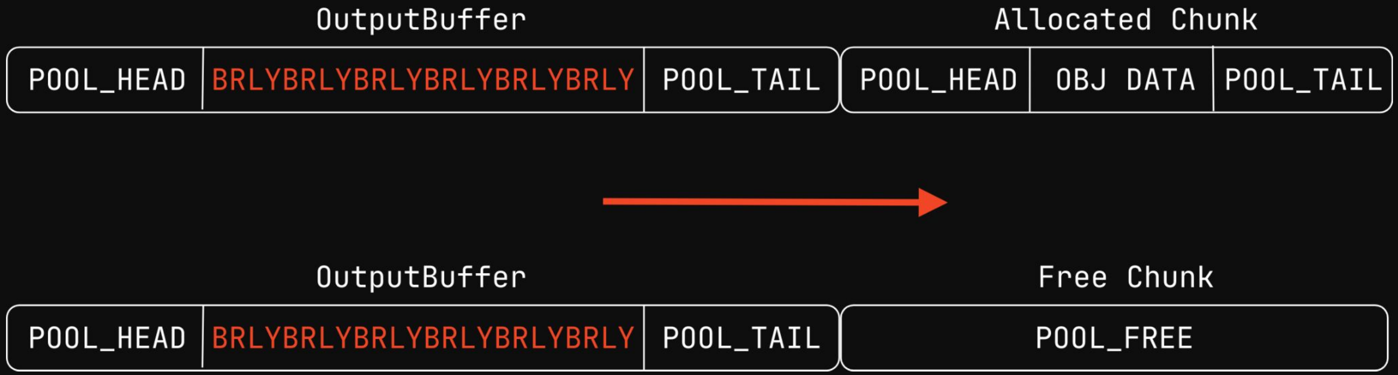
UEFI Heap Internals

- Pool-based heap





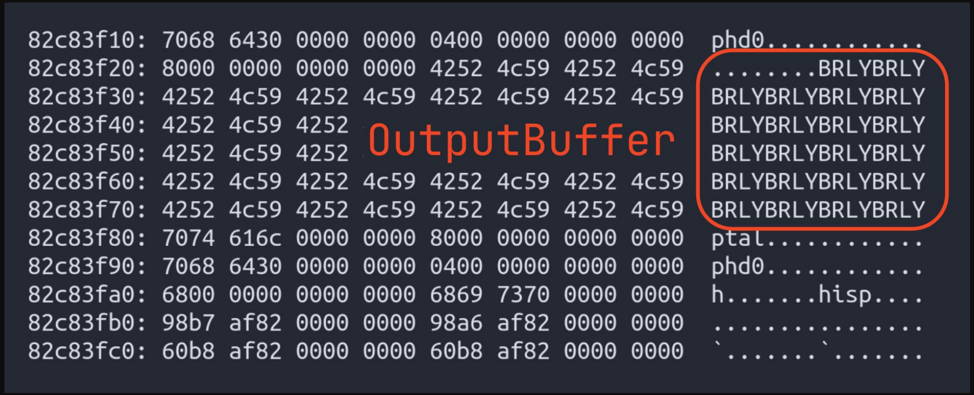
What Are We Even Corrupting?



We don’t know!!

Long Live UEFI Memory

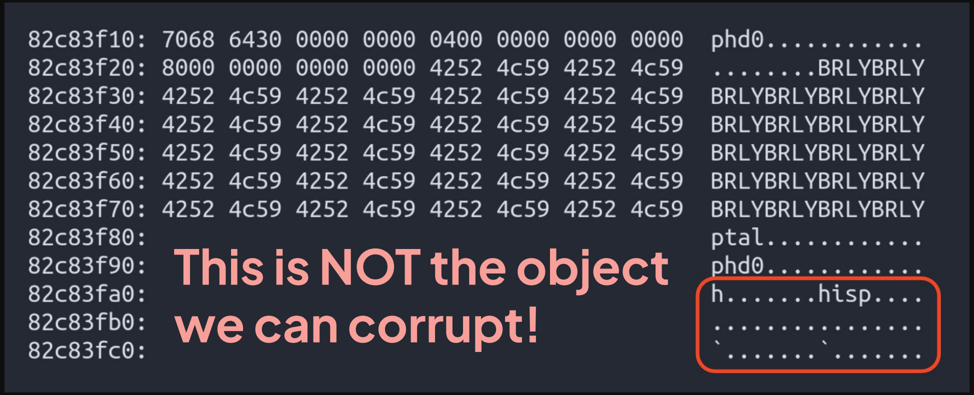
- Memory used by UEFI is not cleared
- If the OS doesn't overwrite it, we can dump it after boot
- `OutputBuffer` is not freed, so it's somewhere in memory!



```
82c83f10: 7068 6430 0000 0000 0400 0000 0000 0000  phd0............
82c83f20: 8000 0000 0000 0000 4252 4c59 4252 4c59  ........BRLYBRLY
82c83f30: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f40: 4252 4c59 4252                           BRLYBRLYBRLYBRLY
                           OutputBuffer
82c83f50: 4252 4c59 4252                           BRLYBRLYBRLYBRLY
82c83f60: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f70: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f80: 7074 616c 0000 0000 8000 0000 0000 0000  ptal............
82c83f90: 7068 6430 0000 0000 0400 0000 0000 0000  phd0............
82c83fa0: 6800 0000 0000 0000 6869 7370 0000 0000  h.......hisp....
82c83fb0: 98b7 af82 0000 0000 98a6 af82 0000 0000  ................
82c83fc0: 60b8 af82 0000 0000 60b8 af82 0000 0000  `.......`.......
```

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!

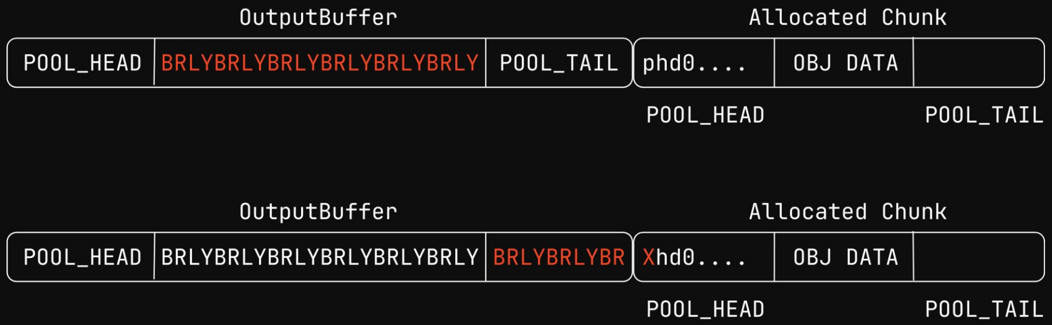


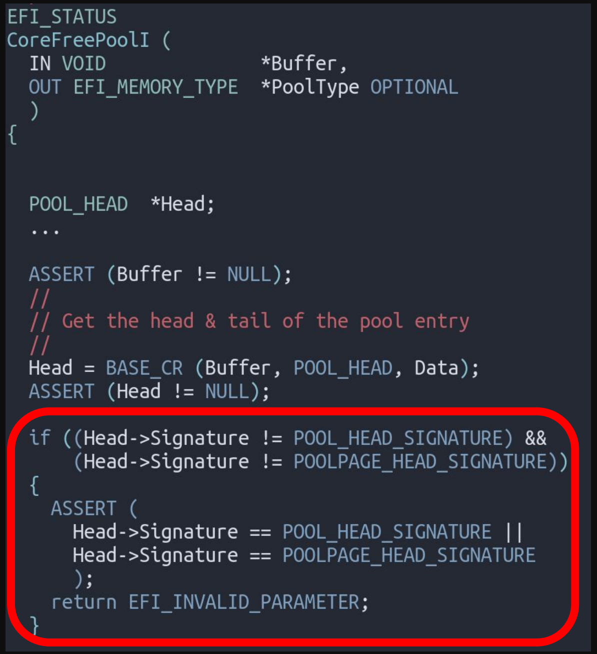
```
82c83f10: 7068 6430 0000 0000 0400 0000 0000 0000  phd0............
82c83f20: 8000 0000 0000 0000 4252 4c59 4252 4c59  ........BRLYBRLY
82c83f30: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f40: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f50: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f60: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f70: 4252 4c59 4252 4c59 4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
82c83f80:                                          ptal............
82c83f90:                                          phd0............
82c83fa0:                                          h.......hisp....
82c83fb0:                                          ................
82c83fc0:                                          `.......`.......
```

This is NOT the object we can corrupt!

Preserving Heap Chunks

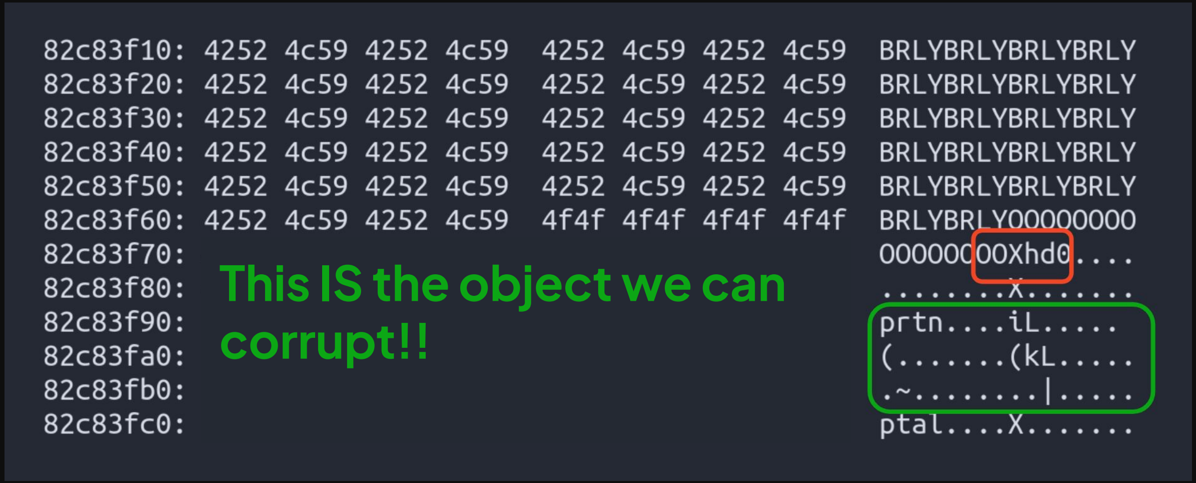
- New technique to preserve chunks
- Corrupting the signature ensures a chunk is not reused





```
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;
  }
```

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

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?

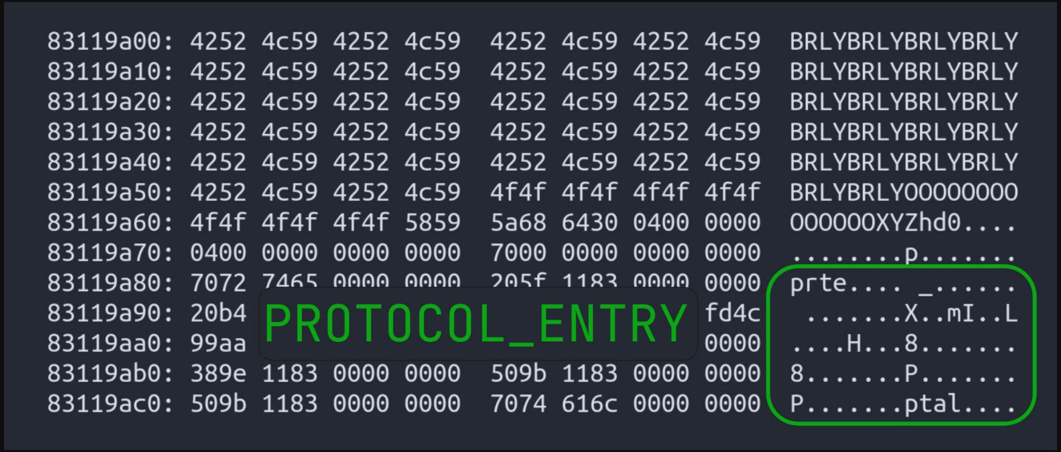
- Finding a good target for corruption
- Get code execution out of it

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

Enter the UEFI Heap Feng Shui

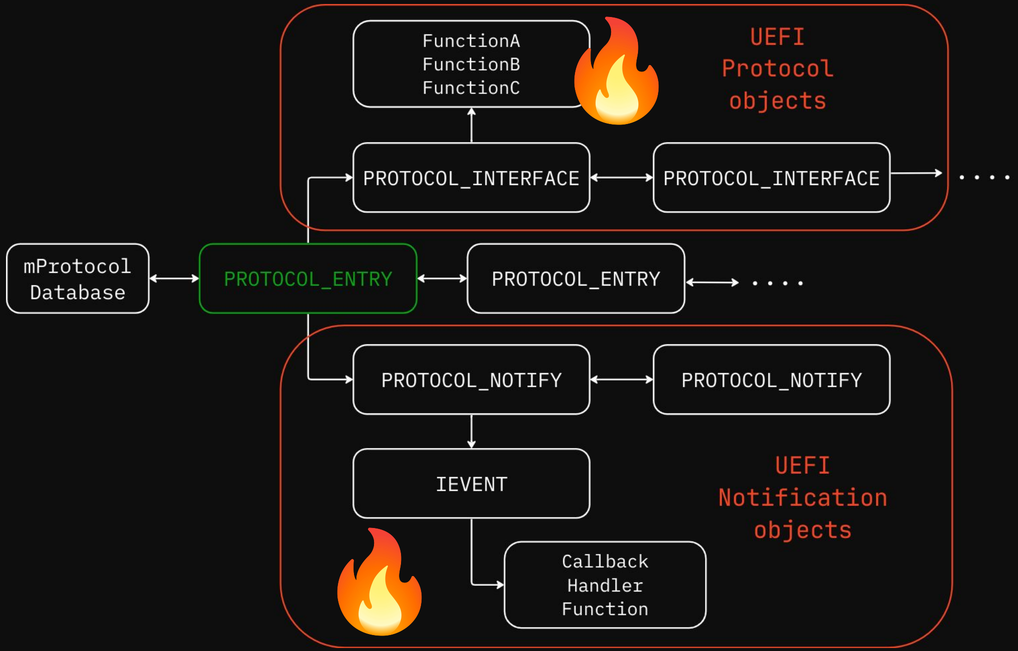
- Heap exploitation often requires strong allocation and deallocation primitives
- We can influence the heap by adding PNG chunks or changing their sizes



```
83119a00: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
83119a10: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
83119a20: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
83119a30: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
83119a40: 4252 4c59 4252 4c59  4252 4c59 4252 4c59  BRLYBRLYBRLYBRLY
83119a50: 4252 4c59 4252 4c59  4f4f 4f4f 4f4f 4f4f  BRLYBRLYOOOOOOOO
83119a60: 4f4f 4f4f 4f4f 5859  5a68 6430 0400 0000  OOOOOOXYZhd0....
83119a70: 0400 0000 0000 0000  7000 0000 0000 0000  ........p.......
83119a80: 7072 7465 0000 0000  205f 1183 0000 0000  prte.... _......
83119a90: 20b4                                fd4c   ........X..mI..L
83119aa0: 99aa                                0000  ....H...8.......
83119ab0: 389e 1183 0000 0000  509b 1183 0000 0000  8.......P.......
83119ac0: 509b 1183 0000 0000  7074 616c 0000 0000  P.......ptal....
```

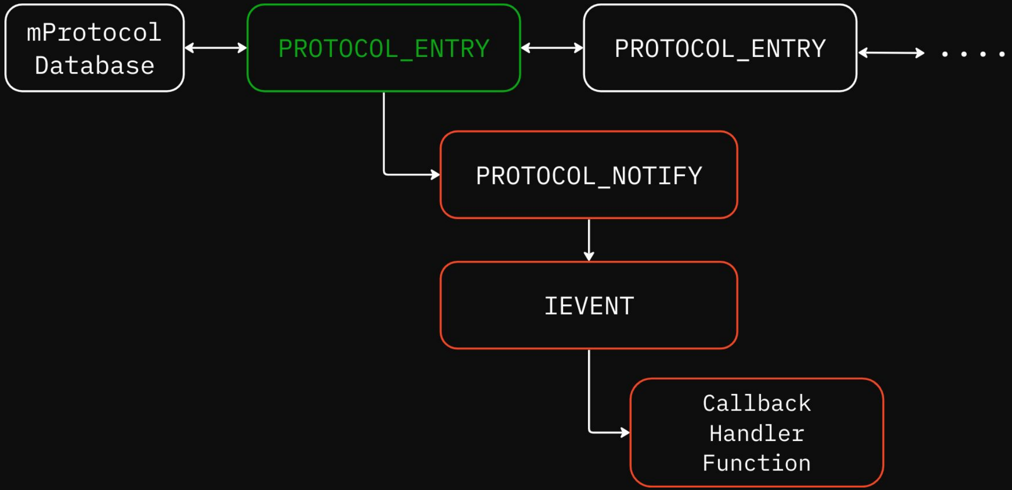
PROTOCOL_ENTRY, tell me more..

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



UEFI Event System

- Events are generated when protocols are installed

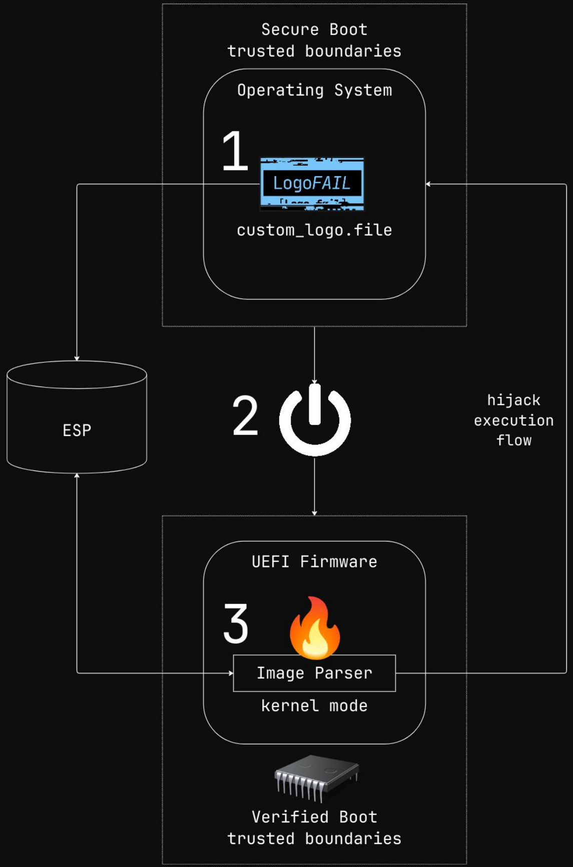


Arbitrary Code Exec in UEFI

- 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

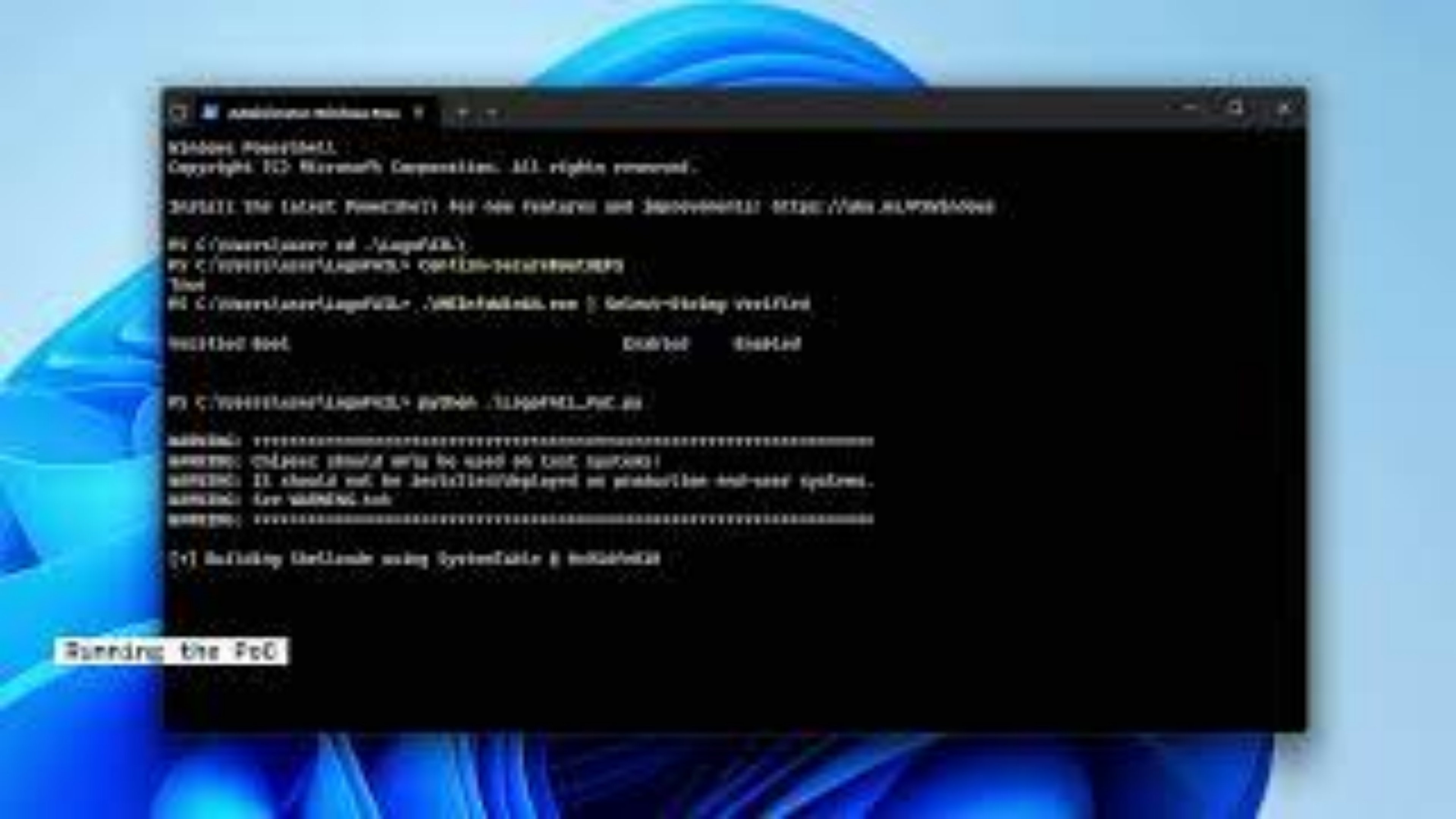
Putting it All Together

- 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



Demo

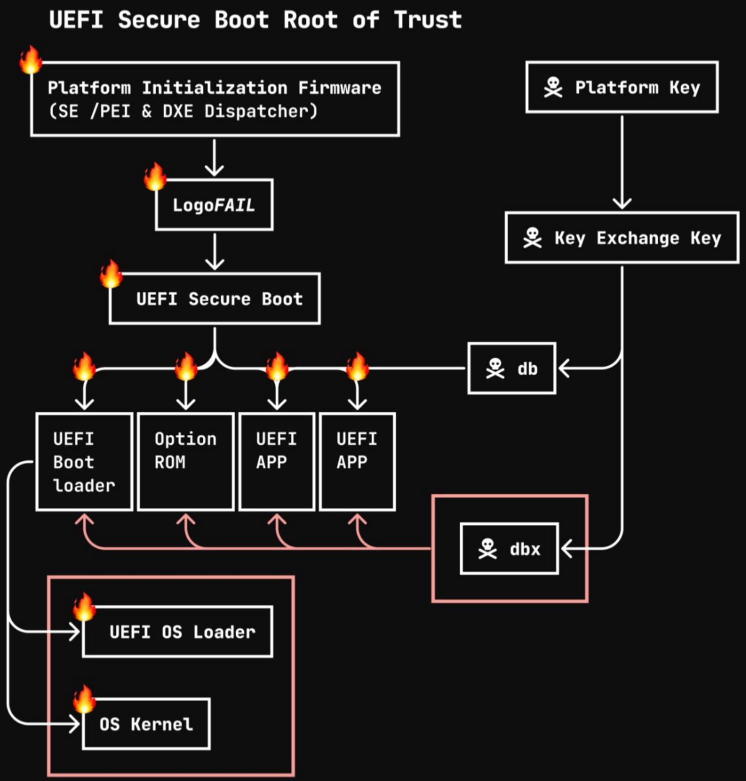
https://www.youtube.com/watch?v=Eufe0Pe6eqk



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\user> cd .\LogoFAIL\
PS C:\Users\user\LogoFAIL> Confirm-SecureBootUEFI
True
PS C:\Users\user\LogoFAIL> .\HSIInfoAdmin.exe | Select-String Verified

Verified Boot                                    Disabled     Enabled


PS C:\Users\user\LogoFAIL> python .\LogoFAIL_PoC.py

WARNING: ***********************************************************************
WARNING: This PoC should only be used on test systems!
WARNING: It should not be installed/deployed on production end-user systems.
WARNING: See WARNING.txt
WARNING: ***********************************************************************

[+] Building shellcode using SystemTable @ [illegible]
```

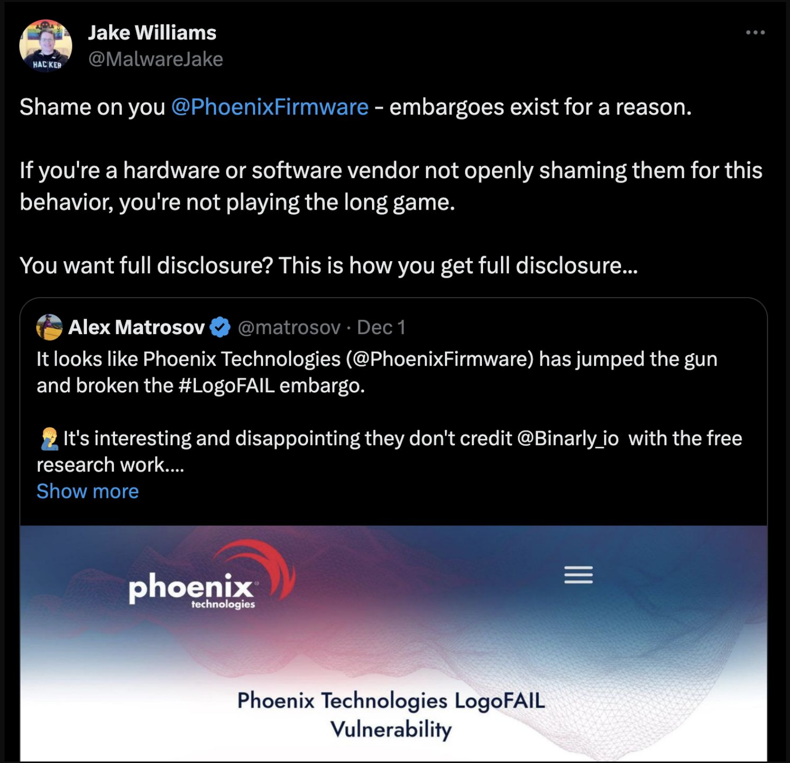
Logo*FAIL*

- 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



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

Phoenix Technology 🤦‍♂️



Jake Williams
@MalwareJake

Shame on you @PhoenixFirmware - embargoes exist for a reason.

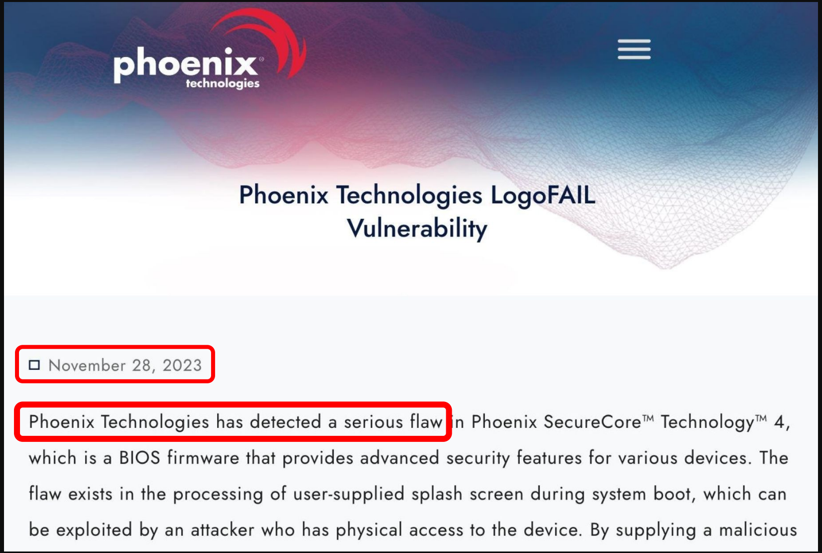
If you're a hardware or software vendor not openly shaming them for this behavior, you're not playing the long game.

You want full disclosure? This is how you get full disclosure...

Alex Matrosov @matrosov · Dec 1
It looks like Phoenix Technologies (@PhoenixFirmware) has jumped the gun and broken the #LogoFAIL embargo.

🤦‍♂️ It's interesting and disappointing they don't credit @Binarly_io with the free research work....
Show more

Phoenix Technologies LogoFAIL Vulnerability



Phoenix Technologies LogoFAIL Vulnerability

November 28, 2023

Phoenix Technologies has detected a serious flaw in Phoenix SecureCore™ Technology™ 4, which is a BIOS firmware that provides advanced security features for various devices. The flaw exists in the processing of user-supplied splash screen during system boot, which can be exploited by an attacker who has physical access to the device. By supplying a malicious

*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!