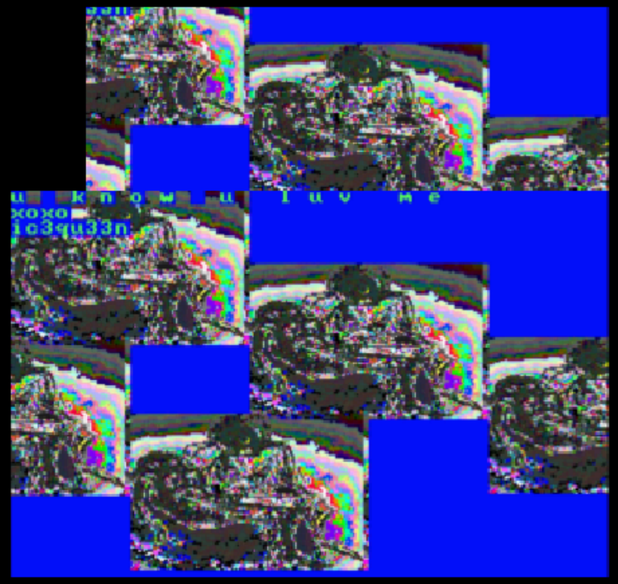
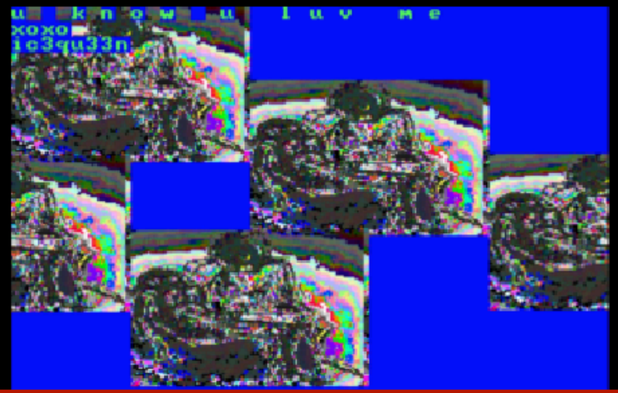



```
BMP image height: 1932525568

BMP image width: 0

BMP image coordinate x: 266694816

BMP image coordinate y: 266694824

Translate BMP File to GOP blt buffer successful!

BMP image height: 200

BMP image width: 320

_
```

GOP Complex

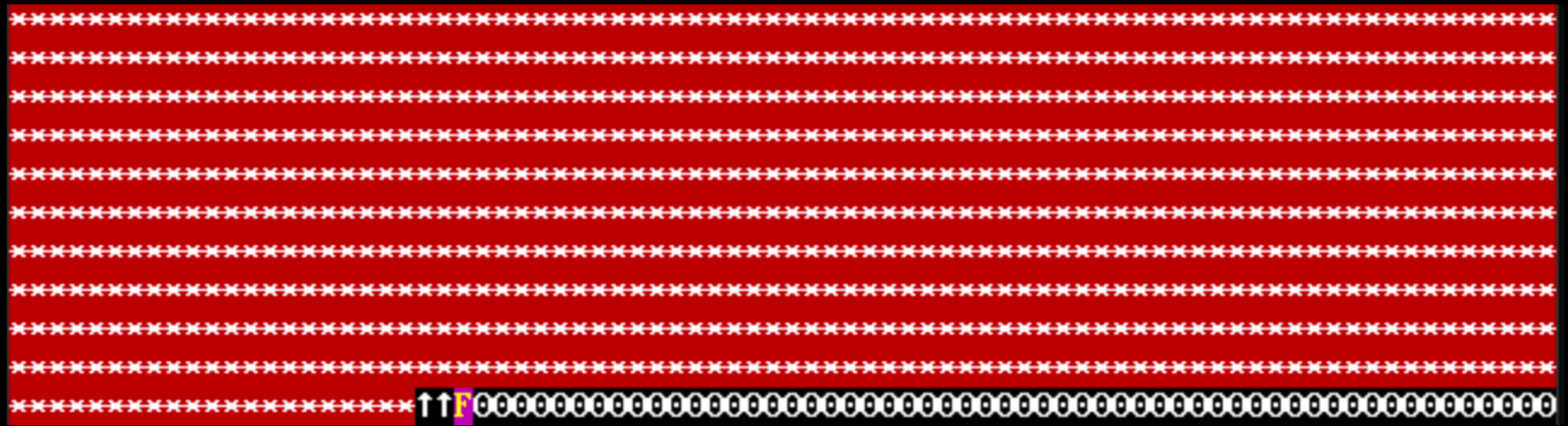
Image parsing bugs, EBC polymorphic engines and the Deus ex machina of UEFI exploit dev

Nika Korchok Wakulich (ic3qu33n)
Recon 2024



DISCLAIMER:
The views expressed in this presentation are my own and do not reflect the opinions of my past, present or future employers

Viewer Discretion is advised.



whoami

Twitter: @nikaroxanne
Discord: @ic3qu33n
Mastodon: ic3qu33n@infosec.exchange
Website: https://ic3qu33n.fyi/
GitHub: @ic3qu33n and @nikaroxanne
bsky: @ic3qu33n

Security Consultant at Leviathan Security Group
Reverse engineer + artist + hacker
I <3 UEFI, hardware hacking, binary exploitation, skateboarding,
learning languages, creating art, writing programs in assembly languages, etc.

greetz 2 the following for their feedback/support w this talk:
0day (@0day_simpson), Erik Cabetas, netspooky (@netspooky), hermit (@ackmage)
dnz (@dnoiz1), zeta, gren, xcelerator, @novafacing, Ben Mason (@suidroot),
Alex Matrosov (@matrosov)
Xeno Kovah (@xenokovah)
The team at Leviathan
REcon

leviathan security group

Format of this talk

Format of this Talk

GOP Complex = art-generating UEFI bootkit

Prior work: Michelangelo REanimator MBR bootkit to GOP Complex UEFI bootkit

New techniques combining UEFI RE + exploit dev with artistic practice

GOP Complex components:

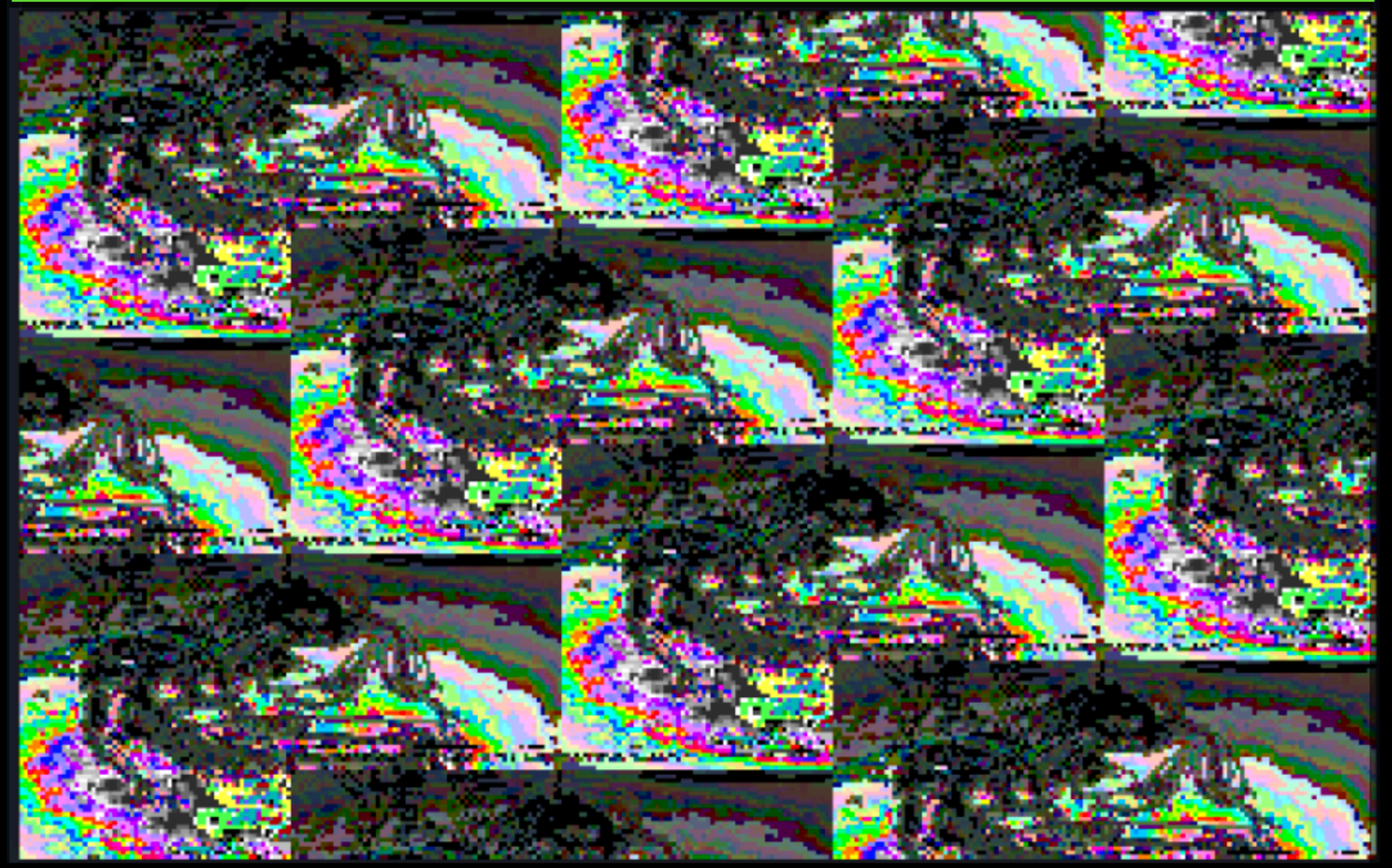
1. LogoFAIL exploit

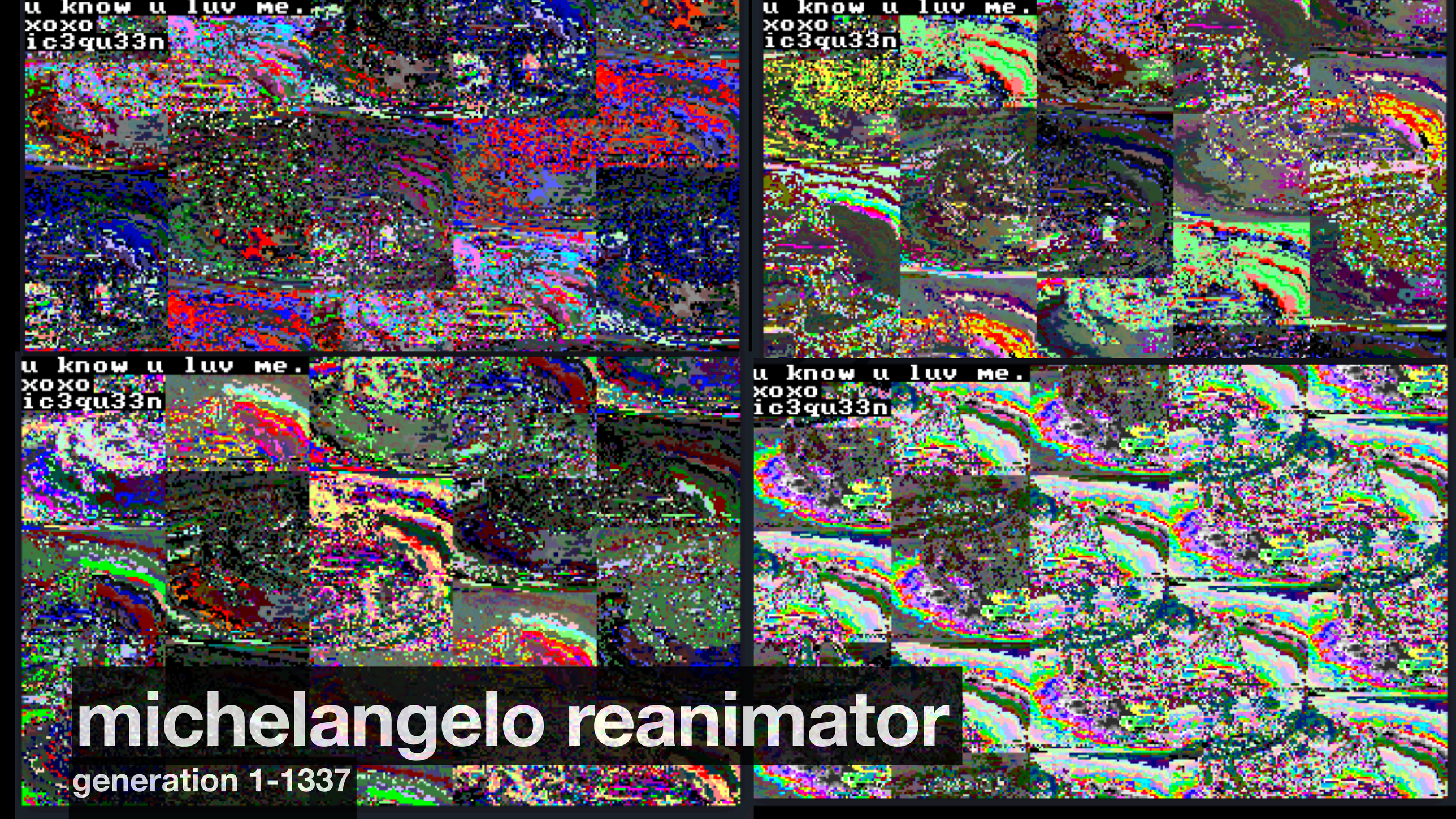
2. PCI Option ROM exploit

3. EBC + EBCVM -> Polymorphic engine

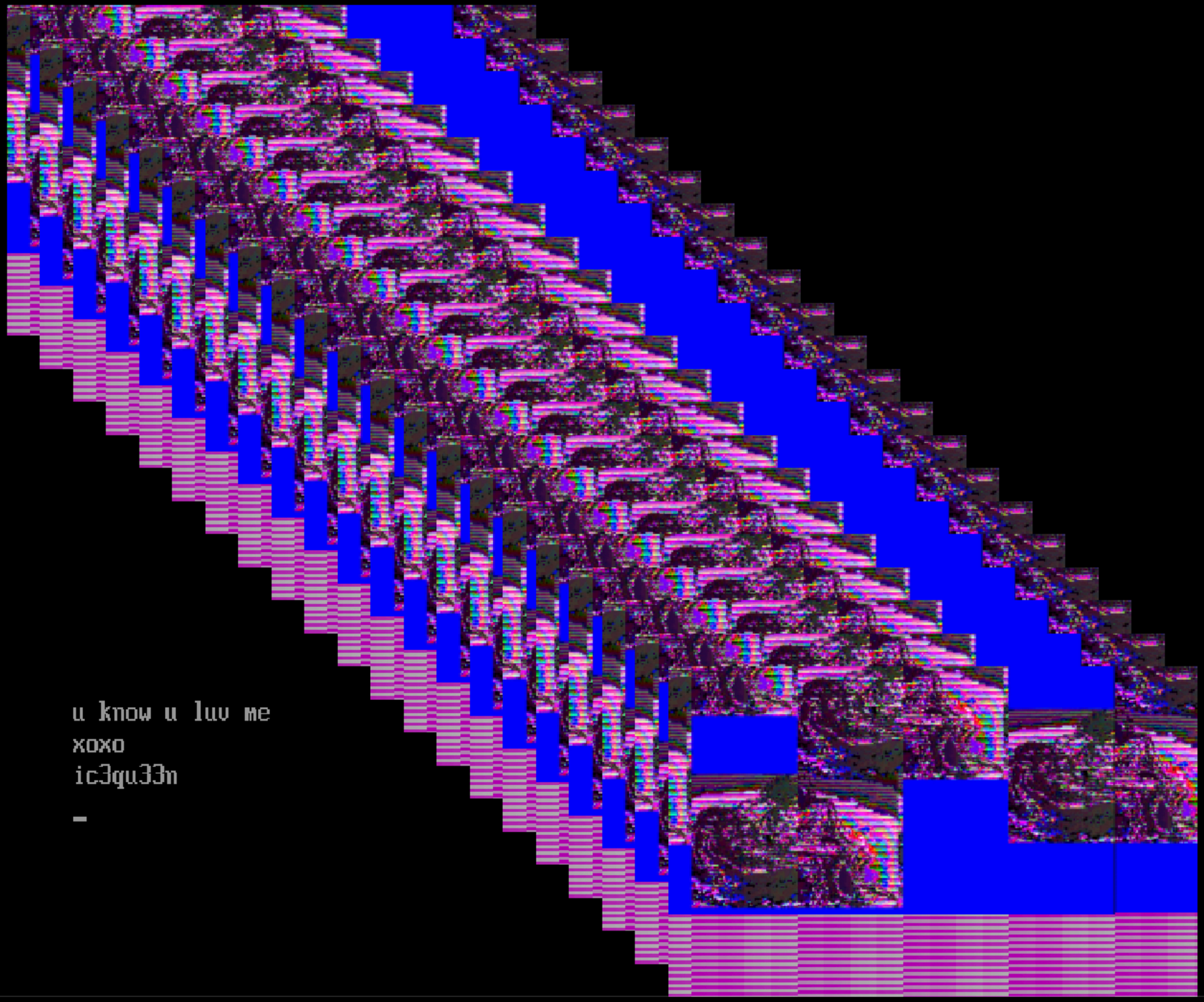
Prior work: malware art

“Michelangelo REanimator,” MBR Bootkit REcon 2023:




u know u luv me.
xoxo
ic3qu33n
u know u luv me.
xoxo
ic3qu33n
u know u luv me.
xoxo
ic3qu33n
u know u luv me.
xoxo
ic3qu33n
michelangelo reanimator
generation 1-1337

GOP Complex


u know u luv me
xoxo
ic3qu33n

Transforming my polymorphic art engine bootkit: from MBR to UEFI

I wanted to explore the UEFI ecosystem and find as many different techniques as possible that I could leverage to turn the UEFI firmware into a VX factory, into a constantly evolving warehouse art show.

GOP Complex

Goals

- Build upon my prior work — Michelangelo REanimator polymorphic art engine MBR bootkit — and write polymorphic art engine UEFI bootkit
- Reverse engineer and exploit a vulnerable image parsing DXE driver impacted by LogoFAIL
- Explore new and interesting attack vectors in the UEFI ecosystem to build elegant and thematically cohesive exploit chain
- Learn how to use GOP for graphics programming on UEFI
- Use that knowledge to RE vulnerable LogoFAIL driver and develop a LogoFAIL exploit
- PCI Option ROM hacking
- EBC + polymorphic engines
- UEFI malware that turns the UEFI ecosystem — the rich pre-OS environment — into a vx factory

“I am thinking about something much more important than bombs. I am thinking about computers.”

—John von Neumann, 1946, [Preface, “Turing’s Cathedral,” George Dyson]

The duality of exploit development: creation + destruction

War machine + creative machine = weird machine

Weird machine: an exploit, an elegant hack

How do you elevate a single exploit and build the complex exploit chain of a sophisticated PoC/malware?

~*The xdev to malware dev pipeline *~

Sponsored by BigVX™


Exploit
development
Artistic/creative
practice

The antidote to corporate bureaucracy crushing creativity in xdev, malware dev and vx:

A symbiotic relationship between artistic/creative practice and exploit development

Resulting in devastating exploits and incredible artwork that push the boundaries of both fields

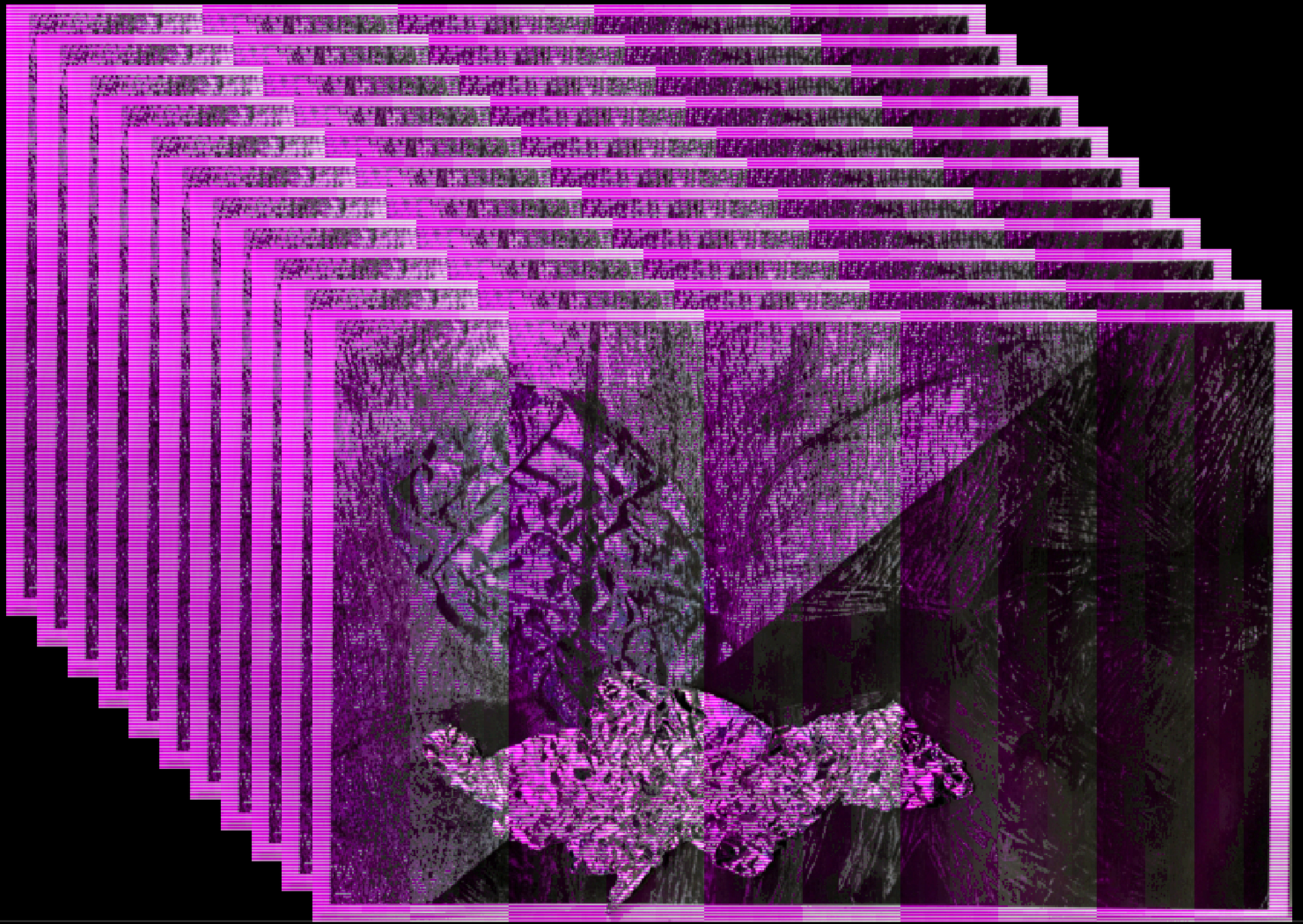
GOP Complex: Thesis

Exploit development + art
=
A match made in VX heaven

GOP Complex

Research questions

- What are the essential techniques for UEFI reverse engineering and exploit development as they relate to LogoFAIL vulnerabilities and to PCI Option ROM exploits?
- How does my artistic practice and creative projects with UEFI graphics programming inform and enhance my work in UEFI RE and exploit development?
- How can I exploit a LogoFAIL vulnerability in my Lenovo’s UEFI boot logo image parsing DXE driver to install a persistent firmware implant?
- How can I turn the UEFI firmware environment into a pre-OS art show and vx factory?



A brief introduction to UEFI

Introduction to UEFI

In the beginning there was legacy BIOS

And now we have UEFI and everything is fine! And there are no more vulnerabilities and Secure Boot wasn’t just a marketing strategy for a feature that was never intended as a security feature of UEFI in the first place!

Oh… wait, never mind.

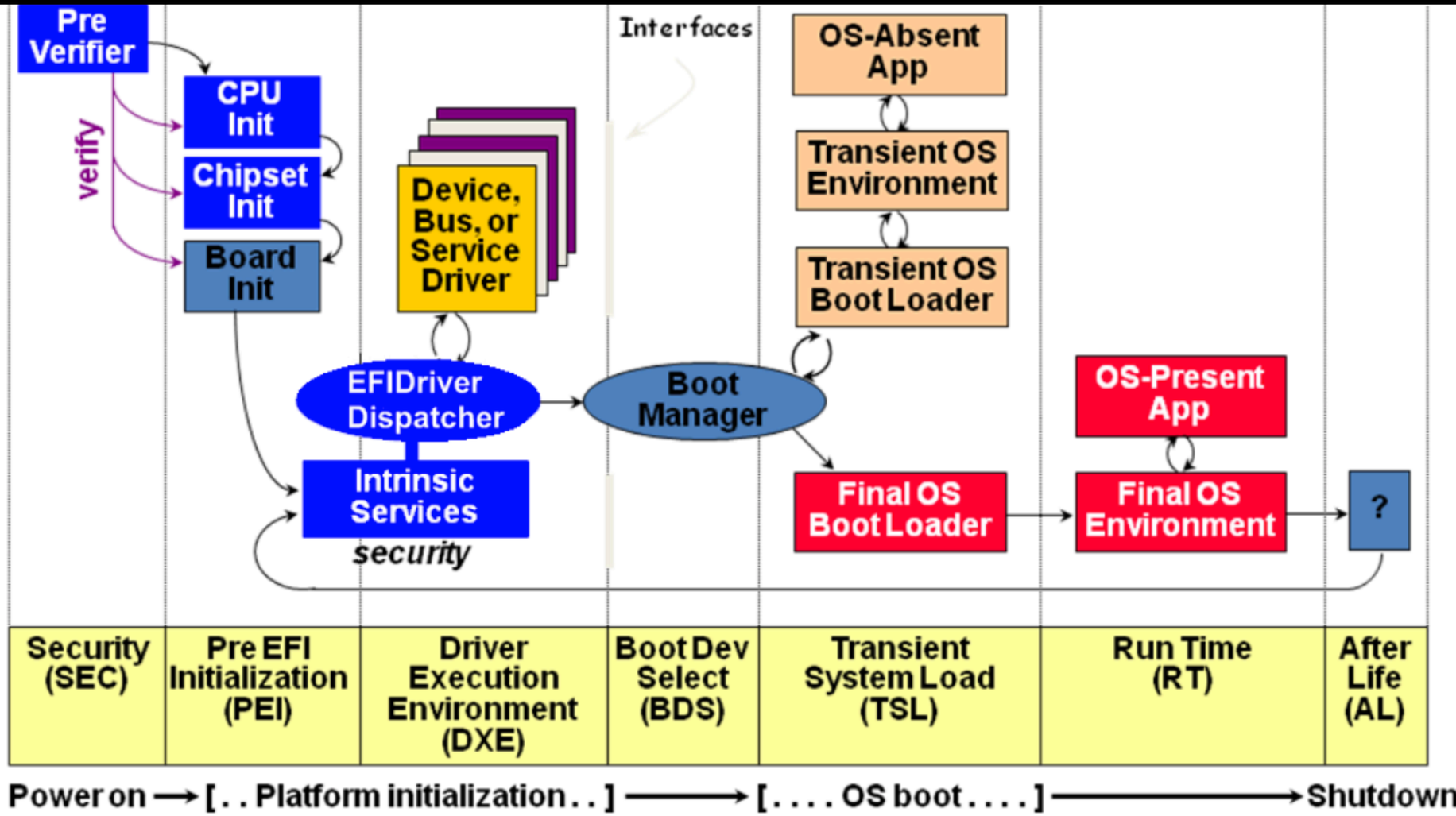


Figure 9 UEFI PI boot flow

Source:
"Trusted Platforms UEFI, PI and TCG-based firmware," Vincent J. Zimmer (Intel Corporation), Shiva R. Dasari Sean P. Brogan (IBM), White Paper by Intel Corporation and IBM Corporation, September 2009
https://www.intel.com/content/dam/doc/white-paper/uefi-pi-tcg-firmware-white-paper.pdf

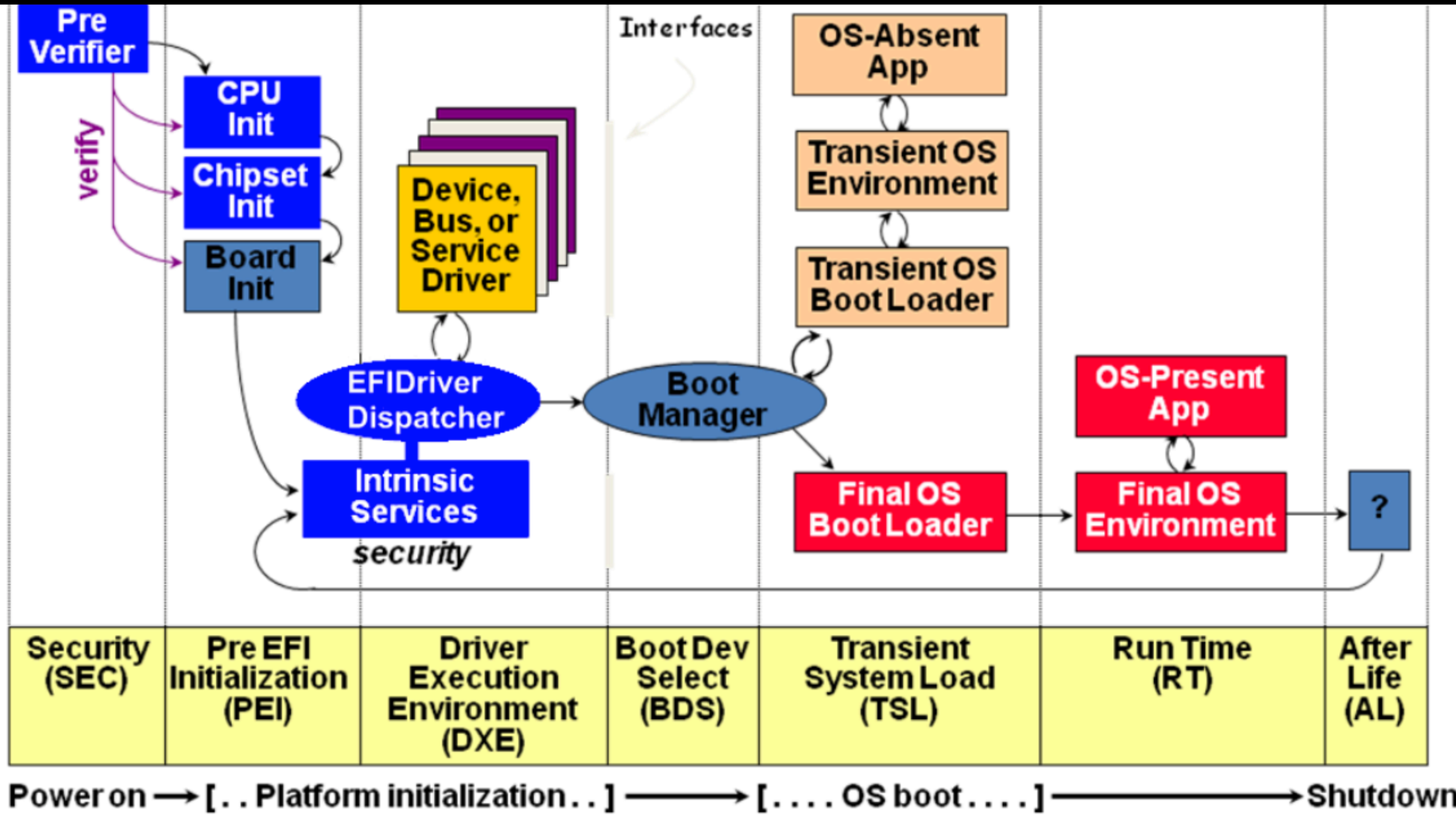


Figure 9 UEFI PI boot flow

Source:
"Trusted Platforms UEFI, PI and TCG-based firmware," Vincent J. Zimmer (Intel Corporation), Shiva R. Dasari Sean P. Brogan (IBM), White Paper by Intel Corporation and IBM Corporation, September 2009
https://www.intel.com/content/dam/doc/white-paper/uefi-pi-tcg-firmware-white-paper.pdf

Introduction to UEFI

UEFI apps/drivers + UEFI shell

- UEFI Shell: A UEFI application that provides a shell interfacing for interacting with various UEFI components (i.e. other UEFI apps and drivers, and the protocols therein)
- UEFI apps and drivers are PE/COFF executables (occasionally TE) and have a PE/COFF header
- The only difference between an UEFI app and a UEFI driver is that an app is unloaded from memory after it is run and a driver remains resident until it is unloaded

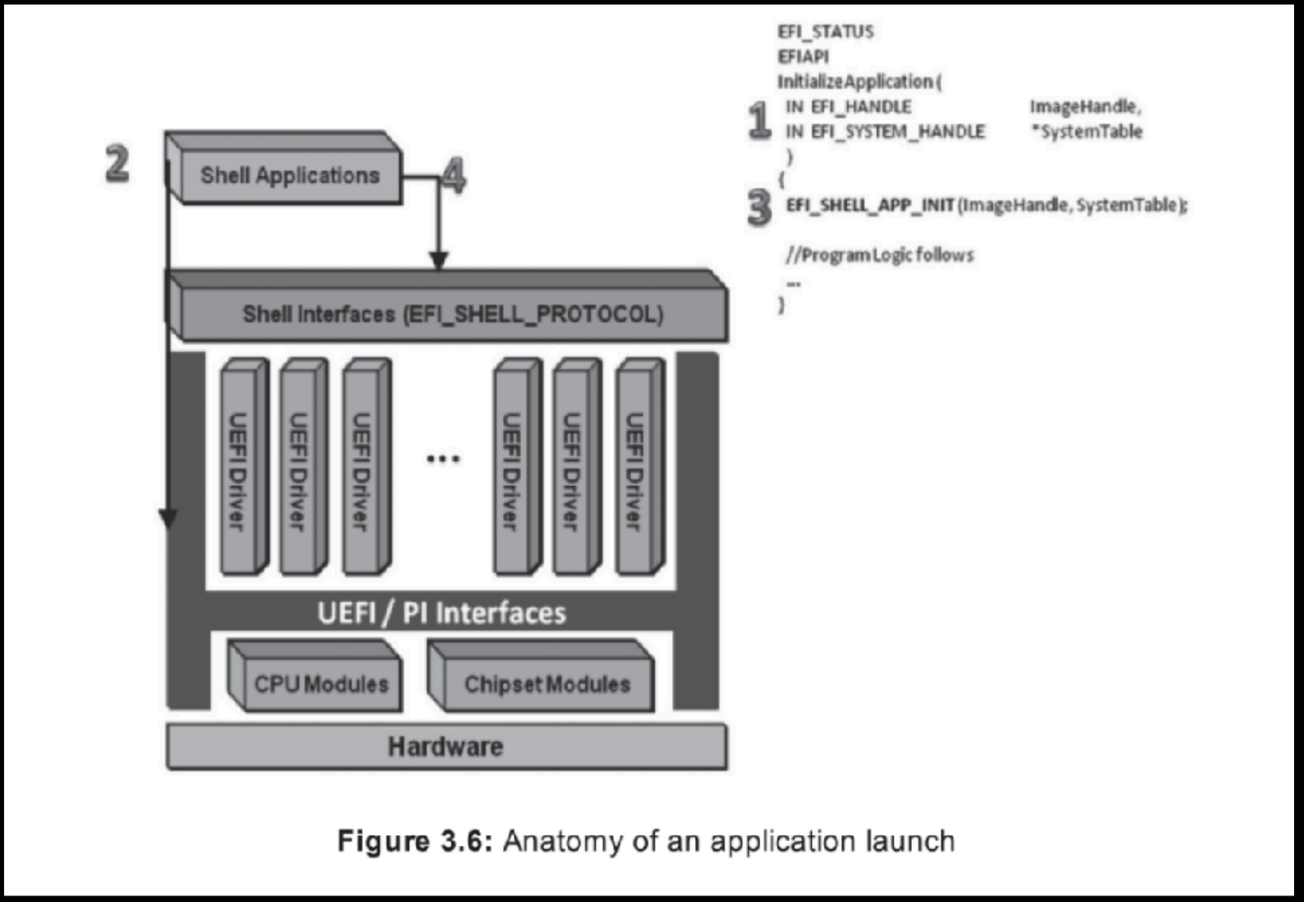


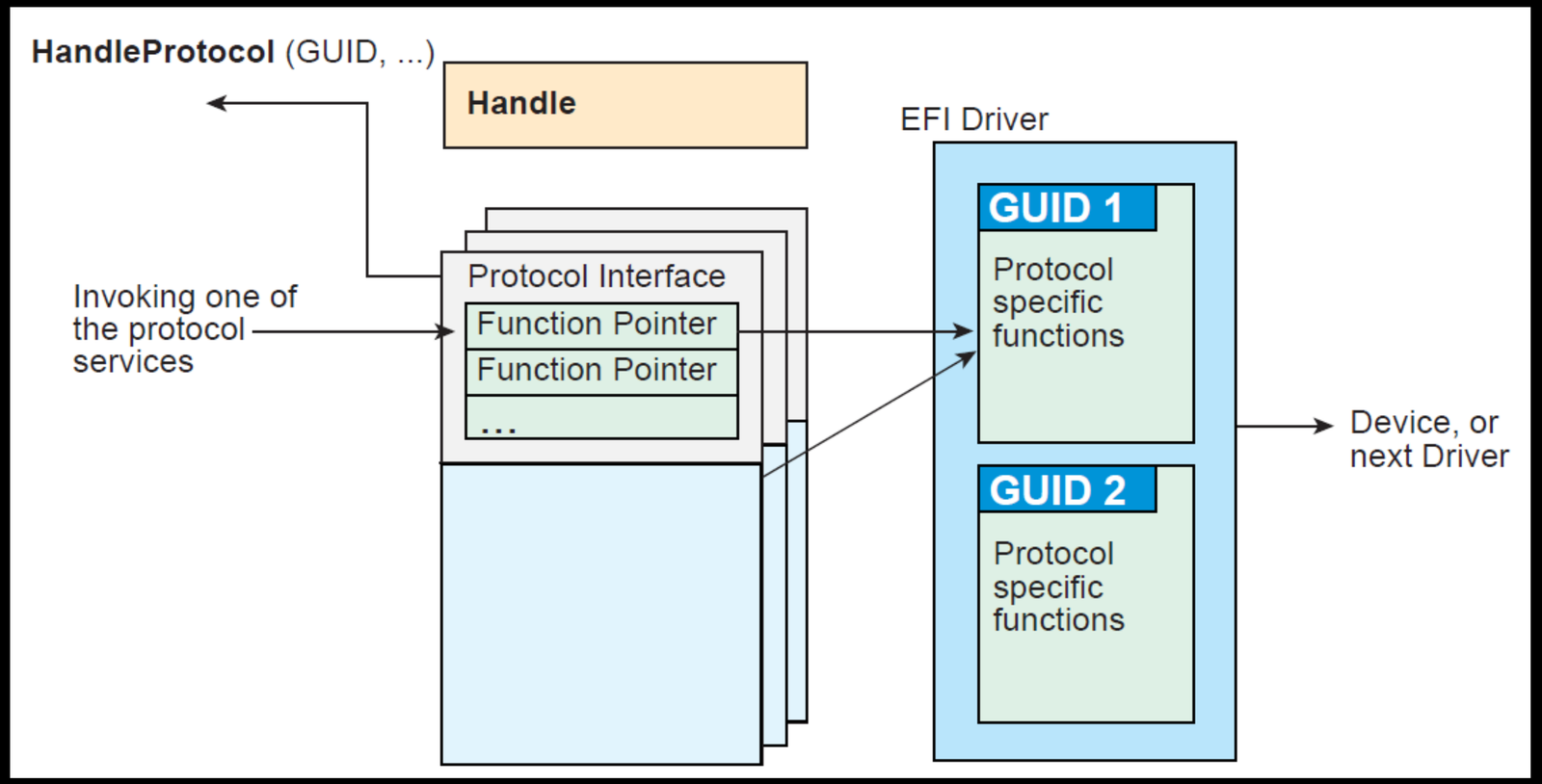
Figure 3.6: Anatomy of an application launch

Source: "Harnessing the UEFI Shell: Moving the Platform Beyond DOS, 2nd edition," Vincent Zimmer, Michael Rothman and Tim Lewis

Introduction to UEFI

Protocols

- Protocols are the keys to the empire
- UEFI is the empire
- A protocol is an interface that encapsulates data and function pointers
- Provide abstractions for hardware/firmware/OS communications
- A driver can produce one or more protocols



Source: “UEFI Specification: Fig. 2.4 Construction of a Protocol”
https://uefi.org/specs/UEFI/2.10/02_Overview.html#construction-of-a-protocol

Introduction to UEFI

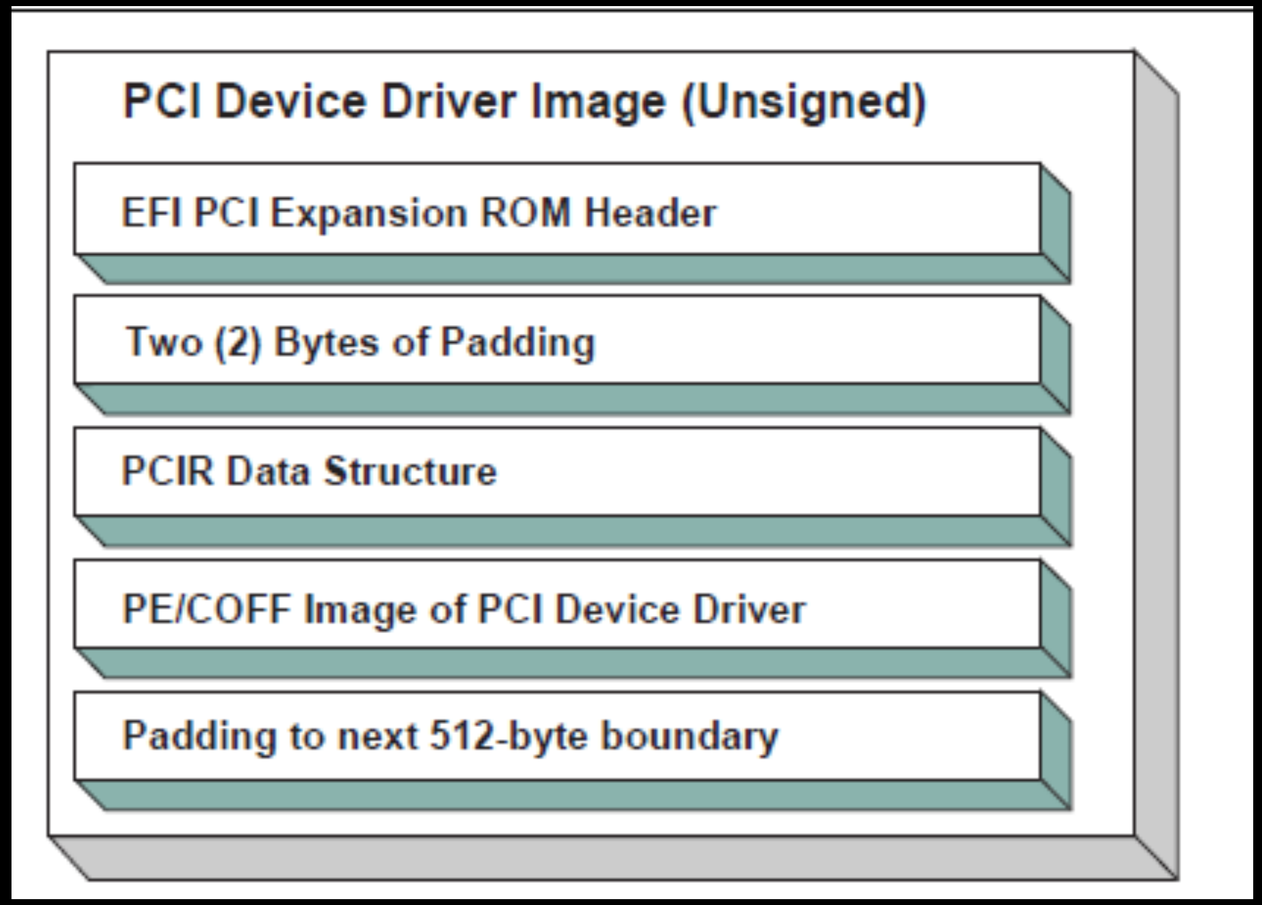
Protocols Example: LoadedImageProtocol

PCI Option ROMs

PCI Option ROMs

Overview

- PCI Option ROM = code residing on a PCI peripheral device that can be loaded/ executed at boot
- PCI Option ROMs (also called PCI expansion ROMs) extend/expand the functionality of the BIOS
- Network cards, graphics cards, etc.



Source: “UEFI Specification: Fig. 14.15 Unsigned PCI Driver Image Layout”
https://uefi.org/specs/UEFI/2.10/02_Overview.html#construction-of-a-protocol

UEFI Graphics programming

GOP: Graphics Output Protocol

Drawing to the frame buffer in UEFI

- GOP is to UEFI as INT 10h functions were to Legacy BIOS
- Query/set video modes
- Draw to the framebuffer
  - Now, with bit blitting!

Example 231-Graphics Output Protocol

```
typedef struct _EFI_GRAPHICS_OUTPUT_PROTOCOL EFI_GRAPHICS_OUTPUT_PROTOCOL;

///
/// Provides a basic abstraction to set video modes and copy pixels to and from
/// the graphics controller's frame buffer. The linear address of the hardware
/// frame buffer is also exposed so software can write directly to the video hardware.
///
struct _EFI_GRAPHICS_OUTPUT_PROTOCOL {
  EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE QueryMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE SetMode;
  EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT Blt;
  ///
  /// Pointer to EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE data.
  ///
  EFI_GRAPHICS_OUTPUT_PROTOCOL_MODE *Mode;
};

extern EFI_GUID gEfiGraphicsOutputProtocolGuid;
```

Source: “23.2 Graphics Output Protocol Implementation,” EDK II Driver Writer’s Guide for UEFI
https://tianocore-docs.github.io/edk2-UefiDriverWritersGuide/draft/23_graphics_driver_design_guidelines/232_graphics_output_protocol_implementation/

BMP

Bitmap file format v3

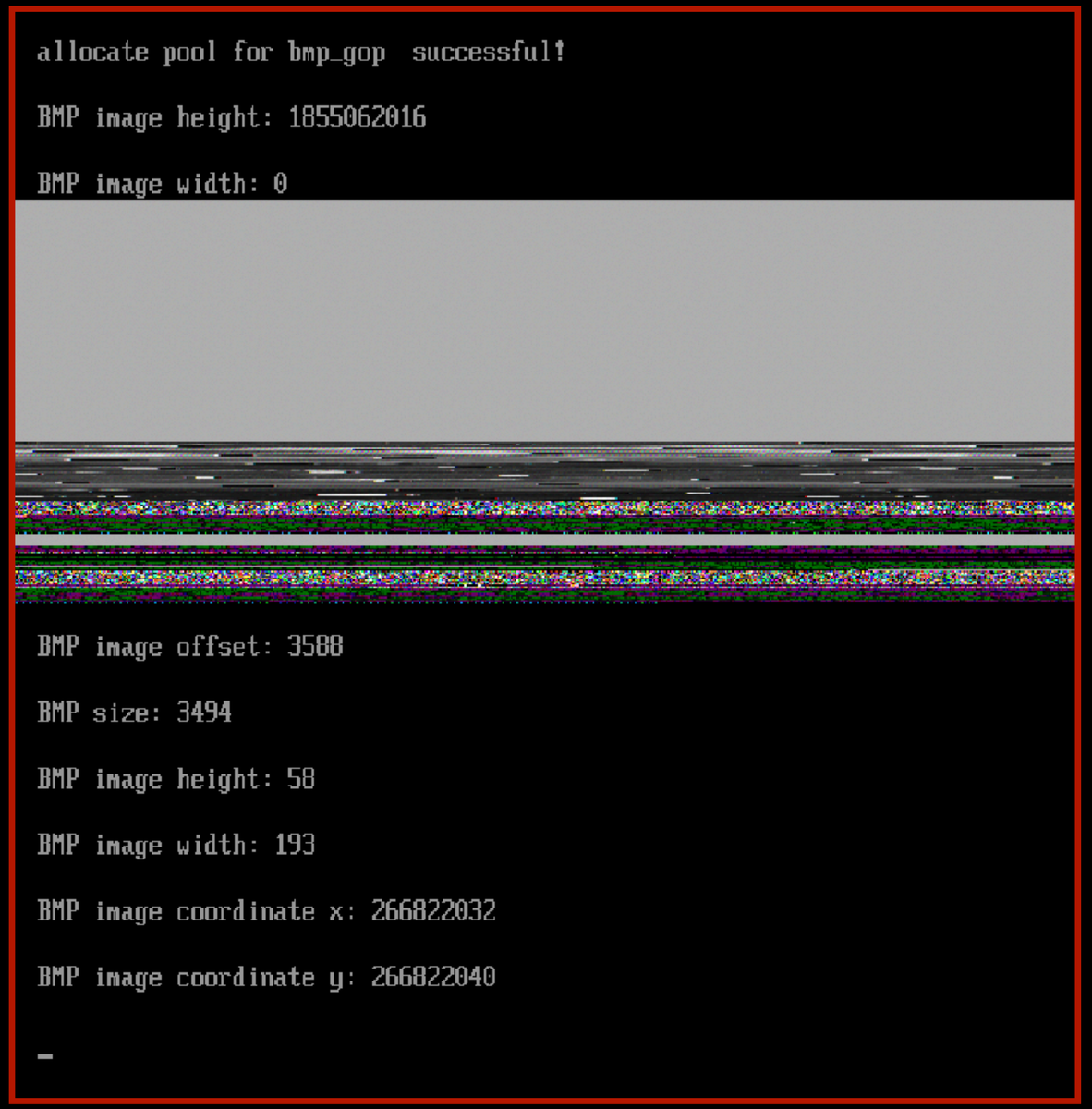
- Gain familiarity with BMP v3 file format and UEFI protocols for rendering BMP images to the framebuffer

```
typedef
EFI_STATUS
(EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_QUERY_MODE) (
 IN EFI_GRAPHICS_OUTPUT_PROTOCOL              *This,
 IN UINT32                                    ModeNumber,
 OUT UINTN                                    *SizeOfInfo
 OUT EFI_GRAPHICS_OUTPUT_MODE_INFORMATION     **Info
 );
typedef
EFI_STATUS
(EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_SET_MODE) (
 IN EFI_GRAPHICS_OUTPUT_PROTOCOL                *This,
 IN UINT32                                      ModeNumber
 );

typedef struct {
 UINT8                          Blue;
 UINT8                          Green;
 UINT8                          Red;
 UINT8                          Reserved;
} EFI_GRAPHICS_OUTPUT_BLT_PIXEL;

typedef enum {
 EfiBltVideoFill,
 EfiBltVideoToBltBuffer,
 EfiBltBufferToVideo,
 EfiBltVideoToVideo,
 EfiGraphicsOutputBltOperationMax
} EFI_GRAPHICS_OUTPUT_BLT_OPERATION;

typedef
EFI_STATUS
(EFIAPI *EFI_GRAPHICS_OUTPUT_PROTOCOL_BLT) (
 IN EFI_GRAPHICS_OUTPUT_PROTOCOL                *This,
 IN OUT EFI_GRAPHICS_OUTPUT_BLT_PIXEL           *BltBuffer, OPTIONAL
 IN EFI_GRAPHICS_OUTPUT_BLT_OPERATION           BltOperation,
 IN UINTN                                       SourceX,
 IN UINTN                                       SourceY,
 IN UINTN                                       DestinationX,
 IN UINTN                                       DestinationY,
 IN UINTN                                       Width,
 IN UINTN                                       Height,
 IN UINTN                                       Delta OPTIONAL
 );
```

UEFI Graphics Programming

BMP Parsing

- Early experiments with UEFI graphics programming
  - Load and parse image from disk
  - BMP image parsing
  - Drawing to the framebuffer with GOP



EFI GOP SetMode call

EFI BootServices Ope ocol was successful: FE76338

EFI_FILE_OPEN Open()

open root volume suc

!get info for BMP fi

Name of BMP file: me

 Physical size of BM

allocate pool for file read successful!

allocate pool for bmp_gop successful!

pixelwidth: 320pixelheight: 200_

GOP Complex


was successful: FE76338
EFI_
open roo
!get info for
Name of BMP file:
Physical size of Bm
allocate pool for file
allocate pool for bmp_gop
pixelwidth: 320pixelheight: 200_

GOP Complex

Exploit chain overview

Multistage UEFI malware:

1. Malicious BMP saved to ESP on disk, NVRAM variable updated to point to new boot logo image—> reboot, malicious BMP logo triggers LogoFAIL exploit;

2. LogoFAIL exploit: heap overflow in DXE phase leads to code exec —>
1st payload: disable UEFI Secure Boot (change NVRAM variable)

3. With UEFI Secure Boot disabled, unsigned PCI Option ROMs can be loaded

4. Reboot -> load 2nd stage payload: malicious PCI option ROM

5. PCI Option ROM hooks GOP operations -> manipulates graphics routines and rendering

6. UEFI firmware ecosystem becomes a vx art factory

Stage 1: LogoFAIL exploit

LogoFAIL

Overview

- LogoFAIL = a collection/class of vulnerabilities in the image parsers of UEFI firmwares, discovered by Binarly REsearch team [1]
- Boot logo customization feature in UEFI firmwares across IBVs allow a user to store a custom logo image displayed during boot
  - Custom logo image is stored on ESP or unprotected region of flash (“a ROM hole”), provides opportunities for bypass of firmware verification checks of UEFI Secure Boot, Intel Boot Guard, etc. [1]
  - LogoFAIL vulnerabilities can be exploited to achieve RCE and bypass Intel Boot Guard [2]
- LogoFAIL won the “Hardest to fix parser bug” award at IEEE Security and Privacy Tenth LangSec workshop
- Recent retrospective/update from Binarly [3]
  - Still many unpatched devices
  - Binarly released 30 advisories for LogoFAIL vulnerabilities

[1] “Inside the LogoFAIL PoC: From Integer Overflow to Arbitrary Code Execution,” Binarly REsearch, January 30, 2024, https://www.binarly.io/blog/inside-the-logofail-poc-from-integer-overflow-to-arbitrary-code-execution
[2] “Finding LogoFAIL: The Dangers of Image Parsing During System Boot,” Binarly REsearch, December 6, 2023, https://www.binarly.io/blog/finding-logofail-the-dangers-of-image-parsing-during-system-boot
[3] “Blind Trust and Broken Fixes: The Ongoing Battle with LogoFAIL Vulnerabilities,” Alex Matrosov, June 19, 2024, https://www.binarly.io/blog/blind-trust-and-broken-fixes-the-ongoing-battle-with-logofail-vulnerabilities

LogoFAIL

Why LogoFAIL?

LogoFAIL = a malware artist’s dream exploit primitive

High number of targets: massive industry-wide impact across nearly every IBV

High variability of different types of exploits

Extensibility —> high yield of returns from RE + xdev process

So many vulnerabilities… gotta catch ‘em all

LogoFAIL RE

Picking a target

- Desired vulnerability:
  - BMP image parser [most familiarity with BMP file format and UEFI graphics programming with that file format]
  - Heap overflow — practice UEFI heap exploitation
  - OOB Write/code exec —> LogoFAIL exploit as first stage of GOP Complex exploit chain
- LogoFAIL vulnerability:
  BRLY-LOGOFAIL-2023-027
  Phoenix PSIRT CVE: CVE-2023-5058

[BRLY-LOGOFAIL-2023-027]

Memory Corruption vulnerability in DXE driver.

Summary

BINARLY efiXplorer team has discovered an OOB Write vulnerability in the decode routine during BMP file processing in Phoenix firmware.

Vulnerability Information

- BINARLY internal vulnerability identifier: BRLY-LOGOFAIL-2023-027
- Phoenix PSIRT assigned CVE identifier: CVE-2023-5058
- CVSS v3.1: 8.2 High AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H

Affected modules with confirmed impact by Binarly team

Module name	Module GUID	Module SHA256
SystemImageDecoderDxe	5F65D21A-8867-45D3-A41A-526F9FE2C598	86E6C85A8FF7C1DB8FF7292521223C546DD4F40F5168F7DA3134916BF52DA

Chosen vulnerability: BRLY-LOGOFAIL-2023-027

GOP Complex - LogoFAIL exploit
UEFI RE + Exploit Dev Process

LogoFAIL RE + xdev

Phases

1. Reverse engineering - static and dynamic analysis*
2. Emulation-based testing and debugging
3. Hardware testing and debugging

*The majority of work on this project was spent reverse engineering the vulnerable driver

LogoFAIL RE

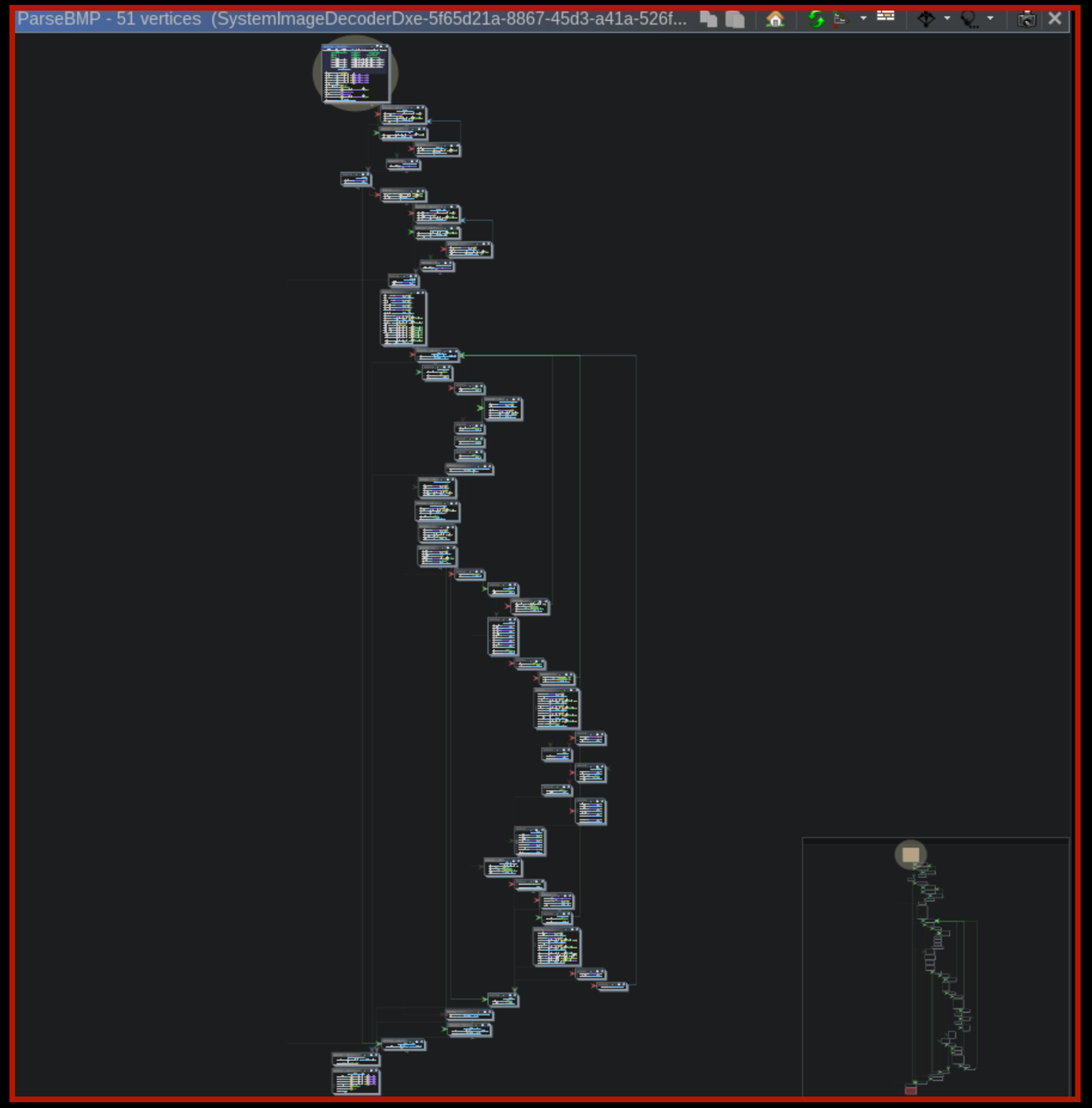
Challenges

The boot logo image parsing drivers use IBV-specific custom UEFI protocols

Type definitions aren’t automatically populated for any custom protocols

Reverse engineering the vulnerable driver had to be done from scratch, aided by knowledge of BMP parsing/boot logo UEFI protocols

**My RE and exploit dev process was guided/informed by the work of the Binarly Research team and their excellent blogs/ presentations about the LogoFAIL vulnerabilities & their development process for their PoC



Function graph of ParseBMP function in vulnerable SystemImageDecoder.Dxe

LogoFAIL RE: Step 0

Acquire hardware, build UEFI xdev + RE lab

- Target device: Lenovo Thinkpad E16 Gen1
- RE Tools:
  - Ghidra, specifically using these two plugins for UEFI:
    - efiSeek: https://github.com/DSecurity/efiSeek
    - ghidra-firmware-utils: https://github.com/al3xtjames/ghidra-firmware-utils
  - Binary Ninja
  - Chipsec: https://chipsec.github.io/
  - UEFITool: https://github.com/LongSoft/UEFITool
  - QEMU and gdb for debugging/testing

LogoFAIL RE: Step 1

Firmware extraction

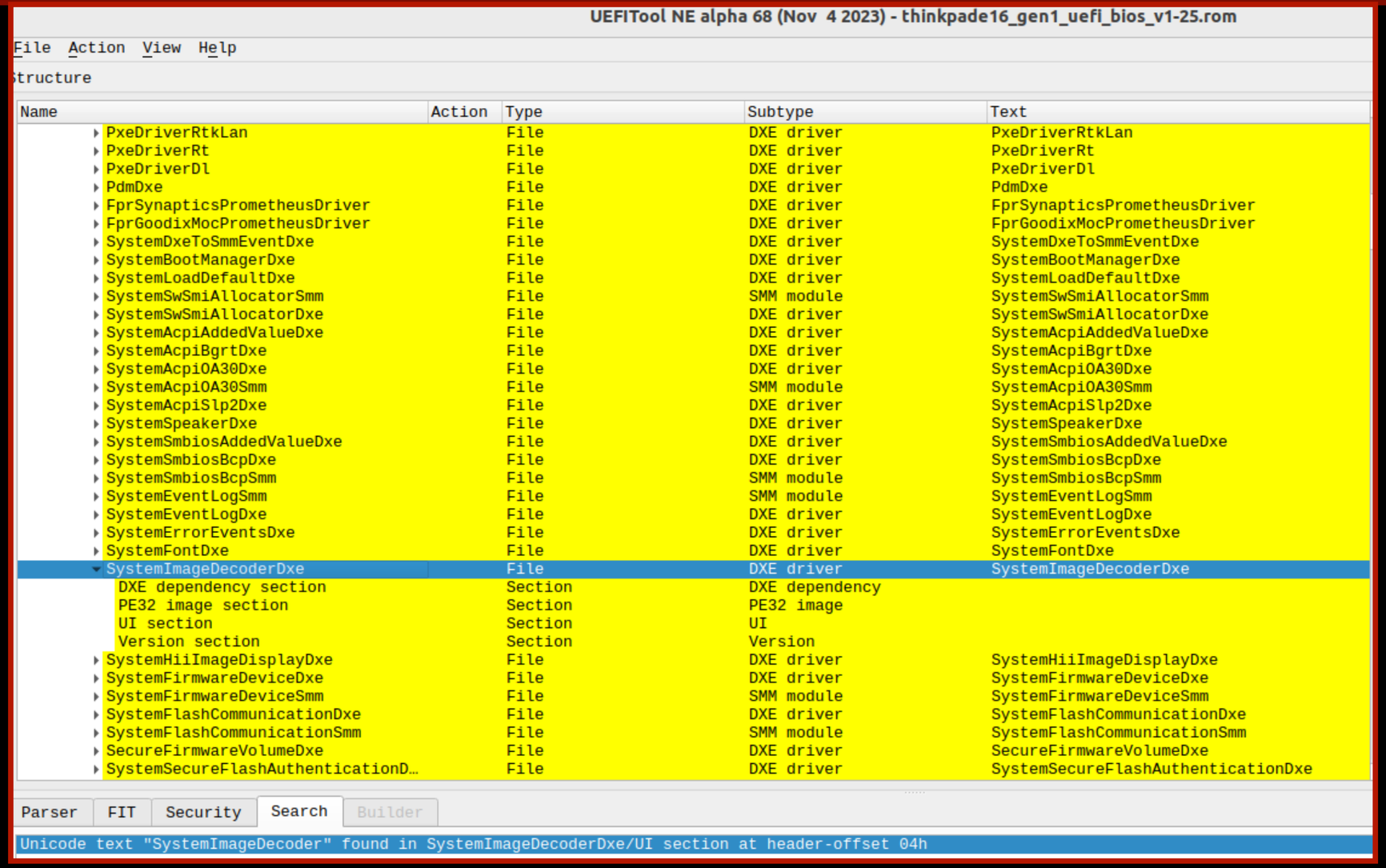
Dump UEFI firmware from running machine with chipsec

Download UEFI firmware for Thinkpad E16 Gen1 from Lenovo driver website
-> UEFI firmware image for most recent *known* vulnerable version: any version prior to version 1.25

```
ic3qu33n@ic3b0x:~/chipsec$ sudo python3 chipsec_util.py spi dump thinkpade
16-gen1-uefi-bios-v131.bin
[sudo] password for ic3qu33n:

################################################################
##                                                            ##
##  CHIPSEC: Platform Hardware Security Assessment Framework  ##
##                                                            ##
################################################################
[CHIPSEC] Version  : 1.13.0
[CHIPSEC] Arguments: spi dump thinkpade16-gen1-uefi-bios-v131.bin


[CHIPSEC] OS      : Linux 6.5.0-1024-oem #25-Ubuntu SMP PREEMPT_DYNAMIC Mo
n May 20 14:47:48 UTC 2024 x86_64
[CHIPSEC] Python  : 3.10.12 (64-bit)
[CHIPSEC] Helper  : LinuxHelper (/home/ic3qu33n/chipsec/chipsec/helper/lin
ux/chipsec.ko)
[CHIPSEC] Platform: RPL-U 2+8
[CHIPSEC]    CPUID: B06A3
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: A708
[CHIPSEC]      RID: 01
[CHIPSEC] PCH     : Intel ADL-P (600) PCH
[CHIPSEC]      VID: 8086
[CHIPSEC]      DID: 519D
[CHIPSEC]      RID: 01

[CHIPSEC] Executing command 'spi' with args ['dump', 'thinkpade16-gen1-uef
i-bios-v131.bin']

[CHIPSEC] Dumping entire SPI flash memory to 'thinkpade16-gen1-uefi-bios-v
131.bin'
[CHIPSEC] it may take a few minutes (use DEBUG or VERBOSE logger options t
o see progress)
[CHIPSEC] BIOS region: base = 0x02000000, limit = 0x03FFFFFF
[CHIPSEC] Dumping 0x04000000 bytes (to the end of BIOS region)
[spi] Reading 0x4000000 bytes from SPI at FLA = 0x0 (in 1048576 0x40-byte
chunks + 0x0-byte remainder)
[CHIPSEC] Completed SPI flash dump to 'thinkpade16-gen1-uefi-bios-v131.bin
'
[CHIPSEC] Time elapsed 353.755
ic3qu33n@ic3b0x:~/chipsec$
                                                    "ic3b0x" 18:04 23-Jun-24
```

LogoFAIL RE: Step 2

Find and extract vulnerable driver: SystemImageDecoderDxe (Phoenix UEFI BIOS)



LogoFAIL RE: Step 3

Identify custom boot image parsing protocol GUID and potential protocol functions

Project Browser | [gopcomplex-...Dxe_body.efi] | [gopcomplex-...Dxe_body.efi]

Symbols | Search symbols

Name	Address	Section
ModuleE…	0x0000003…	
sub_3d4	0x0000003…	
sub_408	0x0000004…	
sub_424	0x0000004…	
sub_454	0x0000004…	
sub_45c	0x0000004…	
sub_4ec	0x0000004…	
sub_4fc	0x0000004…	
j_sub_4…	0x0000005…	
sub_510	0x0000005…	
j_sub_5…	0x0000005…	
sub_51c	0x0000005…	
sub_520	0x0000005…	
sub_554	0x0000005…	
sub_63c	0x0000006…	
sub_770	0x0000007…	
sub_780	0x0000007…	
sub_ef4	0x000000e…	
sub_f70	0x000000f…	
sub_f94	0x000000f…	
sub_f98	0x000000f…	
sub_10a0	0x0000010…	
sub_129c	0x0000012…	
sub_1f28	0x000001f…	
sub_2244	0x0000022…	
sub_22a0	0x0000022…	
sub_23a0	0x0000023…	

PE ▾ Linear ▾ Pseudo C ▾

```
0xb040

0000b011            sub_4fc(rax);
0000b00f        }
0000afb1    }
0000afef    if ((rax == 0 || (rax != 0 && rax_3 == 0)))
0000afef    {
0000b016        rax_6 = nullptr;
0000afef    }
0000b030    return rax_6;
0000af80  }
```

```
0000b040  EFI_GUID gEfiPhoenixLenovoImageParsingProtocolGuid =
0000b040    [Guid("d2221ebf-a7c4-4e87-a89e-bce6e2485a50")]
0000b050  EFI_GUID gEfiLoadedImageProtocolGuid =
0000b050    [Guid("5b1b31a1-9562-11d2-8e3f-00a0c969723b")]
0000b060  void* data_b060 = sub_770
0000b068  void* data_b068 = sub_63c
0000b070  char const data_b070[0x19] = "XXXX PNG chunk not known", 0

0000b089                                00 00 00 00 00 00 00-00 00 00 00 ff 00 00 00-55 00 00 00 49 00 00 00  ............U...I
0000b0a0  11 00 00 00 21 00 00 00-41 00 00 00 81 00 00 00-01 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00  ....!...A.......
0000b0c0  00 00 00 00 01 00 00 00-00 00 00 00 02 00 00 00-04 00 00 00 06 00 00 00-00 00 00 00 01 00 05  ................
0000b0e0  06 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00                                                ........

0000b0f0  int32_t data_b0f0 = 0x0
0000b0f4  char data_b0f4 = 0x0
```

Cross References

Filter (2)

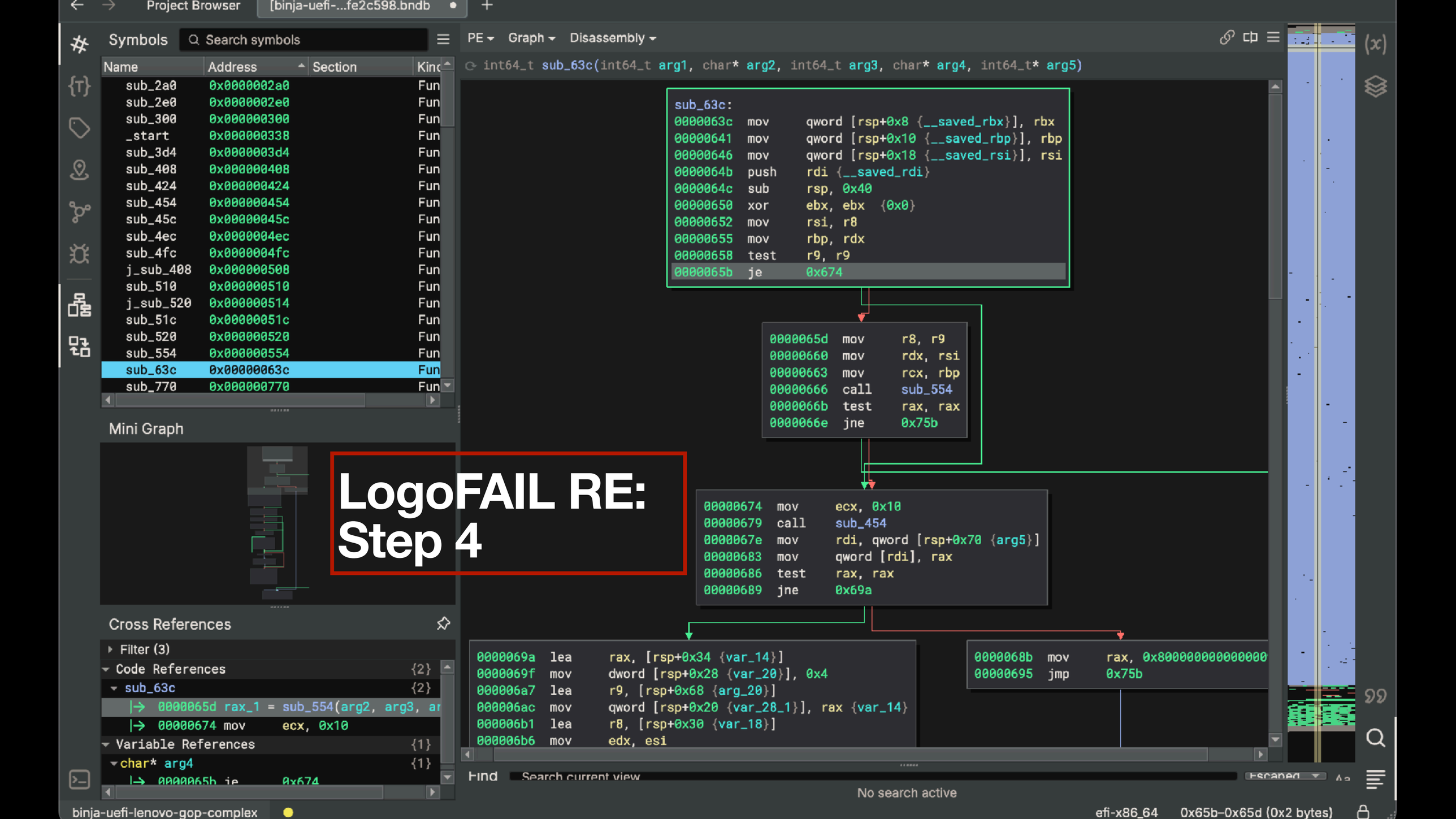
Code References {2}

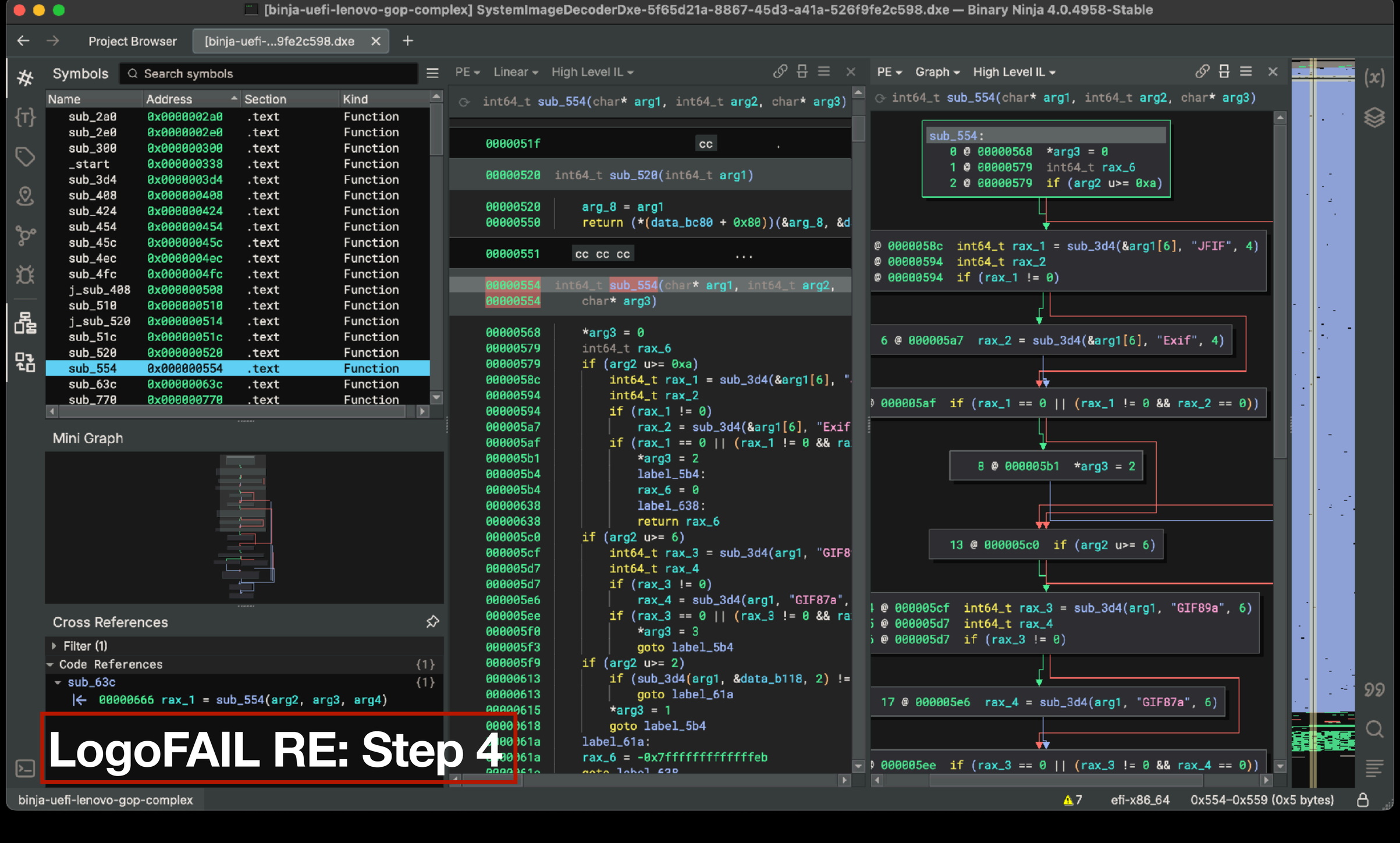
sub_520 {2}

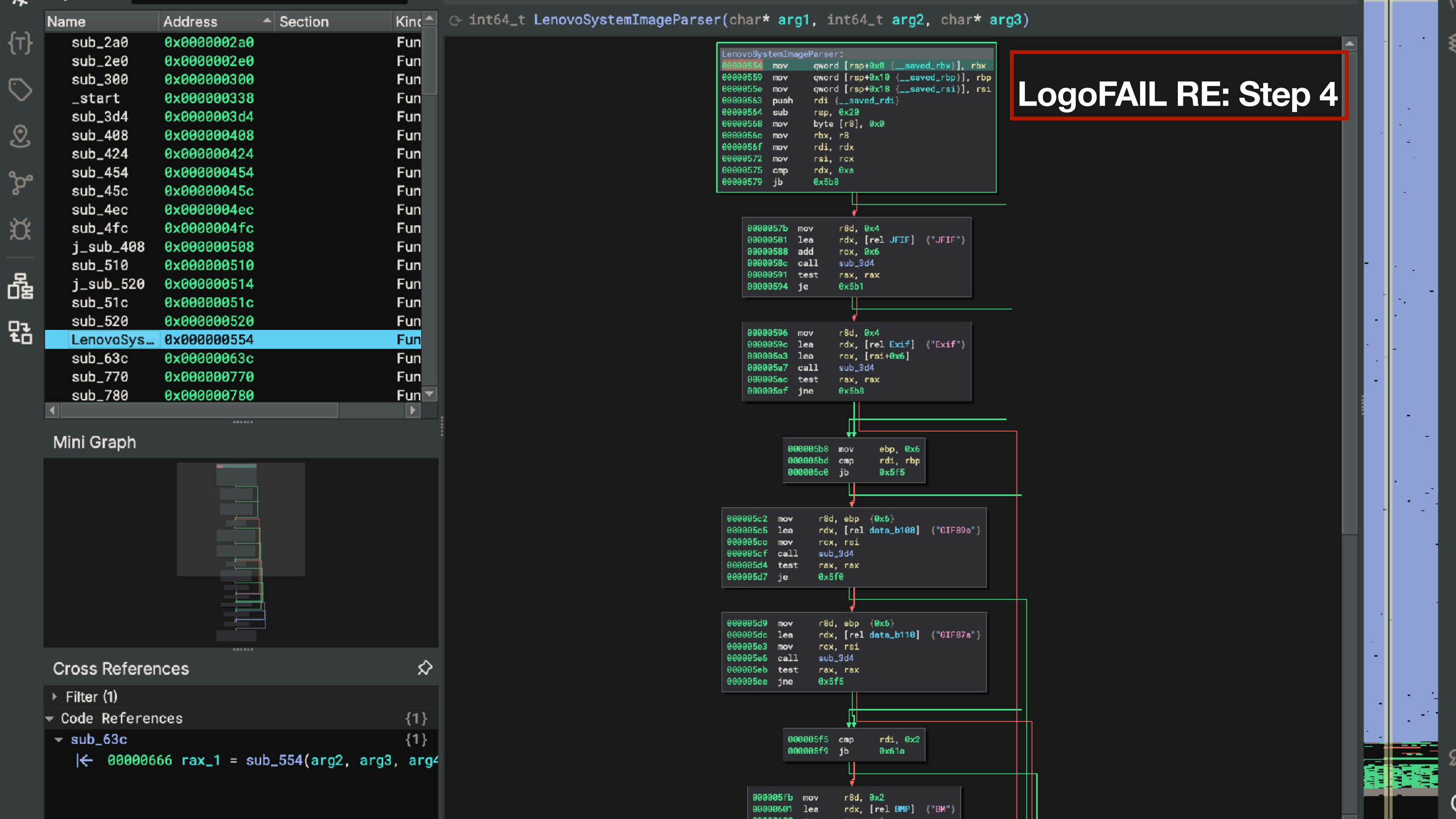
```
|← 0000053a return BootServic
|← 00000546 return BootServic
```

Log | Search log | All

```
[Default] Found EfiGraphicsOutputProtocolGuid at 0x4070 (9042a9de-23dc-4a38-96fb-7aded080516a)
[Default] Found EfiHiiDatabaseProtocolGuid at 0x4080 (ef9fc172-a1b2-4693-b327-6d32fc416042)
[Default] Found EfiSimpleFileSystemProtocolGuid at 0x4090 (964e5b22-6459-11d2-8e39-00a0c969723b)
[Default] Found EfiHiiStringProtocolGuid at 0x40a0 (0fd96974-23aa-4cdc-b9cb-98d17750322a)
[Default] Found EfiAcpiTableProtocolGuid at 0x40b0 (ffe06bdd-6107-46a6-7bb2-5a9c7ec5275c)
[Default] Found EfiAcpiSdtProtocolGuid at 0x40c0 (eb97088e-cfdf-49c6-be4b-d906a5b20e86)
[Default] Found EfiHobListGuid at 0x40d0 (7739f24c-93d7-11d4-9a3a-0090273fc14d)
[Default] Found EfiHiiFontProtocolGuid at 0x4140 (e9ca4775-8657-47fc-97e7-7ed65a084324)
[Default] Found global assignment - offset:0x18ab0 type:struct EFI_SYSTEM_TABLE*
[Default] Found global assignment - offset:0x18ac0 type:struct EFI_RUNTIME_SERVICES*
[Default] UEFI Helper completed successfully!
```


Project Browser
[binja-uefi-...fe2c598.bndb
Symbols
Search symbols
Name
Address
Section
sub_2a0 0x0000002a0
sub_2e0 0x0000002e0
sub_300 0x000000300
_start 0x000000338
sub_3d4 0x0000003d4
sub_408 0x000000408
sub_424 0x000000424
sub_454 0x000000454
sub_45c 0x00000045c
sub_4ec 0x0000004ec
sub_4fc 0x0000004fc
j_sub_408 0x000000508
sub_510 0x000000510
j_sub_520 0x000000514
sub_51c 0x00000051c
sub_520 0x000000520
sub_554 0x000000554
sub_63c 0x00000063c
sub_770 0x000000770
Mini Graph
Cross References
Filter (3)
Code References {2}
sub_63c {2}
0000065d rax_1 = sub_554(arg2, arg3, ar
00000674 mov ecx, 0x10
Variable References {1}
char* arg4 {1}
0000065b je 0x674
PE
Graph
Disassembly
int64_t sub_63c(int64_t arg1, char* arg2, int64_t arg3, char* arg4, int64_t* arg5)
sub_63c:
0000063c mov qword [rsp+0x8 {__saved_rbx}], rbx
00000641 mov qword [rsp+0x10 {__saved_rbp}], rbp
00000646 mov qword [rsp+0x18 {__saved_rsi}], rsi
0000064b push rdi {__saved_rdi}
0000064c sub rsp, 0x40
00000650 xor ebx, ebx {0x0}
00000652 mov rsi, r8
00000655 mov rbp, rdx
00000658 test r9, r9
0000065b je 0x674
0000065d mov r8, r9
00000660 mov rdx, rsi
00000663 mov rcx, rbp
00000666 call sub_554
0000066b test rax, rax
0000066e jne 0x75b
00000674 mov ecx, 0x10
00000679 call sub_454
0000067e mov rdi, qword [rsp+0x70 {arg5}]
00000683 mov qword [rdi], rax
00000686 test rax, rax
00000689 jne 0x69a
0000069a lea rax, [rsp+0x34 {var_14}]
0000069f mov dword [rsp+0x28 {var_20}], 0x4
000006a7 lea r9, [rsp+0x68 {arg_20}]
000006ac mov qword [rsp+0x20 {var_28_1}], rax {var_14}
000006b1 lea r8, [rsp+0x30 {var_18}]
000006b6 mov edx, esi
0000068b mov rax, 0x8000000000000000
00000695 jmp 0x75b
LogoFAIL RE:
Step 4
Find
Search current view
Escaped
No search active
binja-uefi-lenovo-gop-complex
efi-x86_64
0x65b-0x65d (0x2 bytes)


[binja-uefi-lenovo-gop-complex] SystemImageDecoderDxe-5f65d21a-8867-45d3-a41a-526f9fe2c598.dxe — Binary Ninja 4.0.4958-Stable
Project Browser
[binja-uefi-...9fe2c598.dxe
Symbols
Search symbols
Name Address Section Kind
sub_2a0 0x0000002a0 .text Function
sub_2e0 0x0000002e0 .text Function
sub_300 0x000000300 .text Function
_start 0x000000338 .text Function
sub_3d4 0x0000003d4 .text Function
sub_408 0x000000408 .text Function
sub_424 0x000000424 .text Function
sub_454 0x000000454 .text Function
sub_45c 0x00000045c .text Function
sub_4ec 0x0000004ec .text Function
sub_4fc 0x0000004fc .text Function
j_sub_408 0x000000508 .text Function
sub_510 0x000000510 .text Function
j_sub_520 0x000000514 .text Function
sub_51c 0x00000051c .text Function
sub_520 0x000000520 .text Function
sub_554 0x000000554 .text Function
sub_63c 0x00000063c .text Function
sub_770 0x000000770 .text Function
Mini Graph
Cross References
Filter (1)
Code References {1}
sub_63c {1}
|← 00000666 rax_1 = sub_554(arg2, arg3, arg4)
PE Linear High Level IL
int64_t sub_554(char* arg1, int64_t arg2, char* arg3)
0000051f cc .
00000520 int64_t sub_520(int64_t arg1)
00000520 arg_8 = arg1
00000550 return (*(data_bc80 + 0x80))(&arg_8, &d
00000551 cc cc cc ...
00000554 int64_t sub_554(char* arg1, int64_t arg2,
00000554 char* arg3)
00000568 *arg3 = 0
00000579 int64_t rax_6
00000579 if (arg2 u>= 0xa)
0000058c int64_t rax_1 = sub_3d4(&arg1[6], "
00000594 int64_t rax_2
00000594 if (rax_1 != 0)
000005a7 rax_2 = sub_3d4(&arg1[6], "Exif
000005af if (rax_1 == 0 || (rax_1 != 0 && ra
000005b1 *arg3 = 2
000005b4 label_5b4:
000005b4 rax_6 = 0
00000638 label_638:
00000638 return rax_6
000005c0 if (arg2 u>= 6)
000005cf int64_t rax_3 = sub_3d4(arg1, "GIF8
000005d7 int64_t rax_4
000005d7 if (rax_3 != 0)
000005e6 rax_4 = sub_3d4(arg1, "GIF87a",
000005ee if (rax_3 == 0 || (rax_3 != 0 && ra
000005f0 *arg3 = 3
000005f3 goto label_5b4
000005f9 if (arg2 u>= 2)
00000613 if (sub_3d4(arg1, &data_b118, 2) !=
00000613 goto label_61a
00000615 *arg3 = 1
0000 618 goto label_5b4
0000 61a label_61a:
0000 61a rax_6 = -0x7fffffffffffffeb
PE Graph High Level IL
int64_t sub_554(char* arg1, int64_t arg2, char* arg3)
sub_554:
0 @ 00000568 *arg3 = 0
1 @ 00000579 int64_t rax_6
2 @ 00000579 if (arg2 u>= 0xa)
@ 0000058c int64_t rax_1 = sub_3d4(&arg1[6], "JFIF", 4)
@ 00000594 int64_t rax_2
@ 00000594 if (rax_1 != 0)
6 @ 000005a7 rax_2 = sub_3d4(&arg1[6], "Exif", 4)
000005af if (rax_1 == 0 || (rax_1 != 0 && rax_2 == 0))
8 @ 000005b1 *arg3 = 2
13 @ 000005c0 if (arg2 u>= 6)
@ 000005cf int64_t rax_3 = sub_3d4(arg1, "GIF89a", 6)
@ 000005d7 int64_t rax_4
@ 000005d7 if (rax_3 != 0)
17 @ 000005e6 rax_4 = sub_3d4(arg1, "GIF87a", 6)
000005ee if (rax_3 == 0 || (rax_3 != 0 && rax_4 == 0))
LogoFAIL RE: Step 4
binja-uefi-lenovo-gop-complex
7
efi-x86_64
0x554-0x559 (0x5 bytes)


Name
Address
Section
Kind
sub_2a0 0x0000002a0 Fun
sub_2e0 0x0000002e0 Fun
sub_300 0x000000300 Fun
_start 0x000000338 Fun
sub_3d4 0x0000003d4 Fun
sub_408 0x000000408 Fun
sub_424 0x000000424 Fun
sub_454 0x000000454 Fun
sub_45c 0x00000045c Fun
sub_4ec 0x0000004ec Fun
sub_4fc 0x0000004fc Fun
j_sub_408 0x000000508 Fun
sub_510 0x000000510 Fun
j_sub_520 0x000000514 Fun
sub_51c 0x00000051c Fun
sub_520 0x000000520 Fun
LenovoSys... 0x000000554 Fun
sub_63c 0x00000063c Fun
sub_770 0x000000770 Fun
sub_780 0x000000780 Fun
Mini Graph
Cross References
Filter (1)
Code References {1}
sub_63c {1}
00000666 rax_1 = sub_554(arg2, arg3, arg4
int64_t LenovoSystemImageParser(char* arg1, int64_t arg2, char* arg3)
LenovoSystemImageParser:
00000554 mov qword [rsp+0x8 {__saved_rbx}], rbx
00000559 mov qword [rsp+0x10 {__saved_rbp}], rbp
0000055e mov qword [rsp+0x18 {__saved_rsi}], rsi
00000563 push rdi {__saved_rdi}
00000564 sub rsp, 0x20
00000568 mov byte [r8], 0x0
0000056c mov rbx, r8
0000056f mov rdi, rdx
00000572 mov rsi, rcx
00000575 cmp rdx, 0xa
00000579 jb 0x5b8
0000057b mov r8d, 0x4
00000581 lea rdx, [rel JFIF] {"JFIF"}
00000588 add rcx, 0x6
0000058c call sub_3d4
00000591 test rax, rax
00000594 je 0x5b1
00000596 mov r8d, 0x4
0000059c lea rdx, [rel Exif] {"Exif"}
000005a3 lea rcx, [rsi+0x6]
000005a7 call sub_3d4
000005ac test rax, rax
000005af jne 0x5b8
000005b8 mov ebp, 0x6
000005bd cmp rdi, rbp
000005c0 jb 0x5f5
000005c2 mov r8d, ebp {0x6}
000005c5 lea rdx, [rel data_b108] {"GIF89a"}
000005cc mov rcx, rsi
000005cf call sub_3d4
000005d4 test rax, rax
000005d7 je 0x5f0
000005d9 mov r8d, ebp {0x6}
000005dc lea rdx, [rel data_b110] {"GIF87a"}
000005e3 mov rcx, rsi
000005e6 call sub_3d4
000005eb test rax, rax
000005ee jne 0x5f5
000005f5 cmp rdi, 0x2
000005f9 jb 0x61a
000005fb mov r8d, 0x2
00000601 lea rdx, [rel BMP] {"BM"}
LogoFAIL RE: Step 4

```
8000038d 48 89 4c        MOV        qword ptr [RSP + Handle0],ImageHandle
         24 30
80000392 48 8d 15        LEA        SystemTable,[gEfiLenovoSystemImageDecoderProto..
         c7 66 01 00
80000399 48 8d 4c        LEA        ImageHandle=>Handle0,[RSP + 0x30]
         24 30
8000039e 48 89 05        MOV        qword ptr [gBS_7],RAX
         db 8d 01 00
800003a5 ff 90 80        CALL       qword ptr [RAX + 0x80]
         00 00 00
800003ab 48 83 c4 28     ADD        RSP,0x28
800003af c3              RET


                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     undefined8 __fastcall FUN_800003b0(undefined8 param_1, u...
     undefined8        RAX:8          <RETURN>
     undefined8        RCX:8          param_1
     undefined4 ?      RDX:8          param_2
     undefined4 ?      R8:8           param_3
                     FUN_800003b0                                    XREF[1]:     80
800003b0 c7 02 64        MOV        dword ptr [param_2],0x64
         00 00 00
800003b6 33 c0           XOR        EAX,EAX
800003b8 41 c7 00        MOV        dword ptr [param_3],0x7
         07 00 00 00
800003bf c3              RET


                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     longlong __fastcall LenovoSystemImageParser(undefined8 p...
     longlong          RAX:8          <RETURN>
     undefined8        RCX:8          param_1
     char *            RDX:8          param_2
     undefined8        R8:8           param_3
     char *            R9:8           param_4
     EFI_IMAGE_INPU...  Stack[0x28]:8  param_5                          XREF[1
     undefined8        Stack[0x20]:8  local_res20                       XREF[2

     undefined8        Stack[0x18]:8  local_res18                       XREF[2

     undefined8        Stack[0x10]:8  local_res10                       XREF[2

     undefined8        Stack[0x8]:8   local_res8                        XREF[2

     undefined8        Stack[-0x18]:8 local_18                          XREF[1
     undefined8        Stack[-0x20]:8 local_20                          XREF[1
     undefined8        Stack[-0x28]:8 local_28                          XREF[3


                     LenovoSystemImageParser                         XREF[1]:     80
800003c0 48 8b c4        MOV        RAX,RSP
800003c3 48 89 58 08     MOV        qword ptr [RAX + local_res8],RBX
800003c7 48 89 68 10     MOV        qword ptr [RAX + local_res10],RBP
800003cb 48 89 70 18     MOV        qword ptr [RAX + local_res18],RSI
```

```
longlong LenovoSystemImageParser
                    (undefined8 param_1,char *param_2,undefined8 param_3,char *param_4,
                    EFI_IMAGE_INPUT **param_5)

{
  char cVar1;
  char cVar2;
  char *pcVar3;
  EFI_IMAGE_INPUT *pEVar4;
  longlong lVar5;
  ulonglong uVar6;
  char *pcVar7;
  char cVar8;
  ulonglong uVar9;
  char *pcVar10;
  char *local_28;
  char *local_20;
  undefined8 local_18;

  uVar9 = 4;
  pcVar7 = param_2 + 6;
  cVar1 = *pcVar7;
  pcVar10 = s_JFIF_800190ec;
  cVar8 = '\x01';
  pcVar3 = pcVar7;
  for (uVar6 = 4; (((cVar1 != '\0' && (*pcVar10 != '\0')) && (*pcVar3 == *pcVar10)) && (1 < uVar6));
      uVar6 = uVar6 - 1) {
    pcVar3 = pcVar3 + 1;
    pcVar10 = pcVar10 + 1;
    cVar1 = *pcVar3;
  }
  if (*pcVar3 != *pcVar10) {
    pcVar3 = s_Exif_800190f4;
    for (; ((*pcVar7 != '\0' && (*pcVar3 != '\0')) && ((*pcVar7 == *pcVar3 && (1 < uVar9))));
        uVar9 = uVar9 - 1) {
      pcVar7 = pcVar7 + 1;
      pcVar3 = pcVar3 + 1;
    }
    if (*pcVar7 != *pcVar3) {
      cVar1 = *param_2;
      pcVar7 = s_GIF89a_800190fc;
      uVar9 = 6;
      pcVar3 = param_2;
      cVar2 = cVar1;
      for (uVar6 = 6;
          ((cVar2 != '\0' && (*pcVar7 != '\0')) && ((*pcVar3 == *pcVar7 && (1 < uVar6))));
          uVar6 = uVar6 - 1) {
        pcVar3 = pcVar3 + 1;
        pcVar7 = pcVar7 + 1;
        cVar2 = *pcVar3;
      }
      if (*pcVar3 != *pcVar7) {
        pcVar7 = s_GIF87a_80019104;
        pcVar3 = param_2;
        cVar2 = cVar1;
        for (; (((cVar2 != '\0' && (*pcVar7 != '\0')) && (*pcVar3 == *pcVar7)) && (1 < uVar9));
```

LogoFAIL RE

RE SystemImageParser function in custom protocol

```
8000056a 74 1d           JZ          LAB_80000589
8000056c 83 f9 01        CMP         ECX,0x1
8000056f 74 0c           JZ          LAB_8000057d
80000571 48 b8 03        MOV         RAX,-0x7ffffffffffffffd
         00 00 00
         00 00 00 80
8000057b eb 26           JMP         LAB_800005a3

                     LAB_8000057d                                    XREF[1]:     80
8000057d 48 8d 4c        LEA         RCX,[RSP + 0x20]
         24 20
80000582 e8 ad 0e        CALL        FUN_80001434
         00 00
80000587 eb 1a           JMP         LAB_800005a3

                     LAB_80000589                                    XREF[1]:     80
80000589 4c 8b c2        MOV         R8,RDX
8000058c 48 8b ce        MOV         RCX,RSI
8000058f 48 8b d5        MOV         RDX,RBP
80000592 e8 e5 14        CALL        FUN_8000la7c
         00 00
80000597 eb 0a           JMP         LAB_800005a3

                     LAB_80000599                                    XREF[1]:     80
80000599 48 8d 4c        LEA         RCX,[RSP + 0x20]
         24 20
8000059e e8 55 00        CALL        BMP_parser
         00 00

                     LAB_800005a3                                    XREF[4]:     80
                                                                                  80
800005a3 48 8b 5c        MOV         RBX,qword ptr [RSP + 0x50]
         24 50
800005a8 48 8b 6c        MOV         RBP,qword ptr [RSP + 0x58]
         24 58
800005ad 48 8b 74        MOV         RSI,qword ptr [RSP + 0x60]
         24 60
800005b2 48 8b 7c        MOV         RDI,qword ptr [RSP + 0x68]
         24 68
800005b7 48 83 c4 40     ADD         RSP,0x40
800005bb 41 5c           POP         R12
800005bd 41 5d           POP         R13
800005bf cc              INT3
```

LogoFAIL RE

From SystemImageParser function to BMP-specific parsing function

```
                     *************************************************************
                     *                          FUNCTION                         *
                     *************************************************************
                     byte __fastcall FUN_800005c0(undefined8 param_1)
     byte               AL:1           <RETURN>
     undefined8         RCX:8          param_1
```

```
52      }
53      if (*pcVar3 != *pcVar7) {
54        pcVar7 = s_GIF87a_80019104;
55        pcVar3 = param_2;
56        cVar2 = cVar1;
57        for (; (((cVar2 != '\0' && (*pcVar7 != '\0')) && (*pcVar3 == *pcVar7)) && (1 < uVar9));
58            uVar9 = uVar9 - 1) {
59          pcVar3 = pcVar3 + 1;
60          pcVar7 = pcVar7 + 1;
61          cVar2 = *pcVar3;
62        }
63        if (*pcVar3 != *pcVar7) {
64          pcVar7 = s_BM_8001910c;
65          pcVar3 = param_2;
66          for (uVar6 = 2;
67              ((cVar1 != '\0' && (*pcVar7 != '\0')) && ((*pcVar3 == *pcVar7 && (1 < uVar6))));
68              uVar6 = uVar6 - 1) {
69            pcVar3 = pcVar3 + 1;
70            pcVar7 = pcVar7 + 1;
71            cVar1 = *pcVar3;
72          }
73          if (*pcVar3 != *pcVar7) {
74            return -0x7fffffffffffffeb;
75          }
76          goto LAB_80000542;
77        }
78      }
79      cVar8 = '\x03';
80      goto LAB_80000542;
81    }
82  }
83  cVar8 = '\x02';
84LAB_80000542:
85  if (param_4 != (char *)0x0) {
86    *param_4 = cVar8;
87  }
88  pcStack_28 = param_2;
89  pcStack_20 = param_2;
90  uStack_18 = param_3;
91  pEVar4 = (EFI_IMAGE_INPUT *)allocatePool(0x10);
92  *param_5 = pEVar4;
93  if (cVar8 == '\x01') {
94    lVar5 = BMP_parser(&pcStack_28,pEVar4);
95  }
96  else if (cVar8 == '\x02') {
97    lVar5 = FUN_80001a7c((longlong)param_2,(uint)param_3,(undefined4 *)pEVar4);
98  }
99  else if (cVar8 == '\x03') {
100   lVar5 = FUN_80001434((longlong *)&pcStack_28,(undefined4 *)pEVar4);
101 }
102 else {
```

LogoFAIL RE

Reverse engineer the ParseBMP function and tear it to shreds

```
800006e7 48 63 f0        MOVSXD      RSI,pVVar7
800006ea 8b 45 d8        MOV         pVVar7,dword ptr [RBP + bmpHeight]
800006ed 4c 89 65 b0     MOV         qword ptr [RBP + local_88],R12
800006f1 85 c0           TEST        pVVar7,pVVar7
800006f3 79 24           JNS         LAB_80000719
800006f5 8b 4d e0        MOV         BMP_input_buffer,dword ptr [RBP + local_58]
800006f8 85 c9           TEST        BMP_input_buffer,BMP_input_buffer
800006fa 74 14           JZ          LAB_80000710
800006fc 83 f9 03        CMP         BMP_input_buffer,0x3
800006ff 74 0f           JZ          LAB_80000710

                     LAB_80000701                                    XREF[2]:     80000860(j), 8000086
80000701 48 bf 03        MOV         RDI,-0x7ffffffffffffffd
         00 00 00
         00 00 00 80
8000070b e9 41 05        JMP         LAB_80000c51
         00 00

                     LAB_80000710                                    XREF[2]:     800006fa(j), 800006f
80000710 f7 d8           NEG         pVVar7
80000712 40 88 7d 40     MOV         byte ptr [RBP + local_res8],DIL
80000716 89 45 d8        MOV         dword ptr [RBP + bmpHeight],pVVar7

                     LAB_80000719                                    XREF[1]:     800006f3(j)
80000719 0f b7 45 d4     MOVZX       pVVar7,word ptr [RBP + bmpWidth]
8000071d 0f b7 55 d8     MOVZX       param_2,word ptr [RBP + bmpHeight]
80000721 8b c8           MOV         BMP_input_buffer,pVVar7
80000723 0f af ca        IMUL        BMP_input_buffer,param_2
80000726 66 41 89        MOV         word ptr [R14 + 0x6],param_2
         56 06
8000072b 66 41 89        MOV         word ptr [R14 + 0x4],pVVar7
         46 04
80000730 48 63 c9        MOVSXD      BMP_input_buffer,BMP_input_buffer
80000733 48 c1 e1 02     SHL         BMP_input_buffer,0x2
                     ; if you're wondering: "why is the decompilation of this next...
                     ; what's with all the casts??
                     ; *** This is where the initial integer overflow happens: ***
                     *** This routine performs the computation to determine the si...
                     *** We can leverage the fact that there are no checks to comp...
                     ; Combined with the integer overflow,
                     ; we can use this to allocate a chunk of desired size on the ...
                     ; If we allocate a chunk that is too small to contain the ent...
                     ; then we can cause a heap overflow
                     ;
                     ; BmpHeader->Width is of type UINT32
                     ; BmpHeight->Height is of type UINT32;
                     ; there's an imul instruction on UINT32 bmpHeight and UINT32 ...
                     ; before this final bmpImageAllocationSize is passed to alloc...
                     ;
                     ; There are many ways we could exploit this integer overflow ...
                     ; Here's one way we could leverage the integer overflow to cr...
                     ; A check is performed to ensure that *if* the var bmpHeight ...
                     ; we can use this check to overflow bmpHeight by storing the ...
                     ; the final 64-bit signed integer  is (UINT64)(bmpWidth * bmp...
                     ; If we set bmpHeight = 4294967295
                     ; and bmpHeight = 2
                     ; the final bmpImageSize is (1 * 2) << 2 == 8
                     ; bmpImageSize is passed to allocatePool to allocate a chunk ...
                     ; the allocated chunk is much smaller than the size of the su...
                     ; which will lead to a heap overflow

80000737 e8 b4 11        CALL        allocatePool                                    EFI_GRAPHICS_OUTP
```

```
92  pVVar7 = readBuffer(BMP_input_buffer,&stack0xffffffffffffff98,(UINT64)&local_res18,
93                      bmp_image_header + 1);
94  if (pVVar7 != (VOID *)0x0) {
95    return 0;
96  }
97  if (BMP_B != 'B') {
98    return 0;
99  }
100 if (BMP_M != 'M') {
101   return 0;
102 }
103 bmpImageSize = (UINTN)local_54;
104 pixelindex = (ulonglong)local_68;
105 bitIndex = (byte)bitsPerPixel & 0x1f;
106 if ((int)bmpHeight < 0) {
107   if ((local_58 != 0) && (local_58 != 3)) {
108     return 0x8000000000000003;
109   }
110   bmpHeight = -bmpHeight;
111   bVar3 = false;
112 }
113 param_2->Height = (UINT16)bmpHeight;
114 param_2->Width = (ushort)bmpWidth;
115 local_88 = bmpImageSize;
116                     /* ; if you're wondering: "why is the decompilation of this next function so
117                      bizarre?
118                      ; what's with all the casts??
119                      ; *** This is where the initial integer overflow happens: ***
120                      *** This routine performs the computation to determine the size of a buffer
121                      to store the BMP image data***
122                      *** We can leverage the fact that there are no checks to compare the values
123                      of bmpHeight and bmpWidth (stored in the BMP image header) against the actual
124                      size of the file
125                      ; Combined with the integer overflow,
126                      ; we can use this to allocate a chunk of desired size on the heap
127                      ; If we allocate a chunk that is too small to contain the entire BmpImage
128                      ; then we can cause a heap overflow
129                      ;
130                      ; BmpHeader->Width is of type UINT32
131                      ; BmpHeight->Height is of type UINT32;
132                      ; there's an imul instruction on UINT32 bmpHeight and UINT32 bmpWidth
133                      ; before this final bmpImageAllocationSize is passed to allocatePool()
134                      ;
135                      ; There are many ways we could exploit this integer overflow for our purposes
136                      ; Here's one way we could leverage the integer overflow to create a small
137                      buffer with allocatePool() and trigger a heap overflow
138                      ; A check is performed to ensure that *if* the var bmpHeight (from
139                      BmpHeader->Height) is a negative value, *then* bmpHeight=-bmpHeight
140                      ; we can use this check to overflow bmpHeight by storing the max value for an
141                      unsigned 32-bit integer in bmpHeight
142                      ; the final 64-bit signed integer  is (UINT64)(bmpWidth * bmpHeight) << 2)
143                      ; If we set bmpHeight = 4294967295
144                      ; and bmpHeight = 2
145                      ; the final bmpImageSize is (1 * 2) << 2 == 8
146                      ; bmpImageSize is passed to allocatePool to allocate a chunk on the heap
147                      ; the allocated chunk is much smaller than the size of the supplied BMP
148                      ; which will lead to a heap overflow
149                       */
150 bmpBitmap = allocatePool((longlong)(int)((uint)(ushort)bmpWidth * (bmpHeight & 0xffff)) << 2);
151 param_2->Bitmap = (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)bmpBitmap;
152 if (bmpBitmap == (void *)0x0) {
153 LAB_80000745:
```

LogoFAIL RE

Reverse engineer the ParseBMP function and tear it to shreds

```
800006e7 48 63 f0        MOVSXD      RSI,pVVar7
800006ea 8b 45 d8        MOV         pVVar7,dword ptr [RBP + bmpHeight]
800006ed 4c 89 65 b0     MOV         qword ptr [RBP + local_88],R12
800006f1 85 c0           TEST        pVVar7,pVVar7
800006f3 79 24           JNS         LAB_80000719
800006f5 8b 4d e0        MOV         BMP_input_buffer,dword ptr [RBP + local_58]
800006f8 85 c9           TEST        BMP_input_buffer,BMP_input_buffer
800006fa 74 14           JZ          LAB_80000710
800006fc 83 f9 03        CMP         BMP_input_buffer,0x3
800006ff 74 0f           JZ          LAB_80000710

                     LAB_80000701                                    XREF[2]:     80000860(j), 8000086
80000701 48 bf 03        MOV         RDI,-0x7ffffffffffffffd
         00 00 00
         00 00 00 80
8000070b e9 41 05        JMP         LAB_80000c51
         00 00

                     LAB_80000710                                    XREF[2]:     800006fa(j), 800006f
80000710 f7 d8           NEG         pVVar7
80000712 40 88 7d 40     MOV         byte ptr [RBP + local_res8],DIL
80000716 89 45 d8        MOV         dword ptr [RBP + bmpHeight],pVVar7

                     LAB_80000719                                    XREF[1]:     800006f3(j)
80000719 0f b7 45 d4     MOVZX       pVVar7,word ptr [RBP + bmpWidth]
8000071d 0f b7 55 d8     MOVZX       param_2,word ptr [RBP + bmpHeight]
80000721 8b c8           MOV         BMP_input_buffer,pVVar7
80000723 0f af ca        IMUL        BMP_input_buffer,param_2
80000726 66 41 89        MOV         word ptr [R14 + 0x6],param_2
         56 06
8000072b 66 41 89        MOV         word ptr [R14 + 0x4],pVVar7
         46 04
80000730 48 63 c9        MOVSXD      BMP_input_buffer,BMP_input_buffer
80000733 48 c1 e1 02     SHL         BMP_input_buffer,0x2
                     ; if you're wondering: "why is the decompilation of this next...
                     ; what's with all the casts??
                     ; *** This is where the initial integer overflow happens: ***
                     *** This routine performs th                              ...si...
                     *** We can leverage
                     ; Combined with
                     ; we can use
                     ; If we al
                     ; then w
                     ;
                     ; BmpHe
                     ; BmpHei
                     ; there's
                     ; before th
                     ;
                     ; There are
                     ; Here's
                     ; A check is performed to ensure
                     ; we can use this check to overflow bmpHeight by storing the ...
                     ; the final 64-bit signed integer  is (UINT64)(bmpWidth * bmp...
                     ; If we set bmpHeight = 4294967295
                     ; and bmpHeight = 2
                     ; the final bmpImageSize is (1 * 2) << 2 == 8
                     ; bmpImageSize is passed to allocatePool to allocate a chunk ...
                     ; the allocated chunk is much smaller than the size of the su...
                     ; which will lead to a heap overflow

80000737 e8 b4 11        CALL        allocatePool                                  EFI_GRAPHICS_OUTP
```

Tl;dr:
integer overflow leveraged to create small buffer for BMP image data with allocatePool leading to heap overflow

```
92  pVVar7 = readBuffer(BMP_input_buffer,&stack0xffffffffffffff98,(UINT64)&local_res18,
93                      bmp_image_header + 1);
94  if (pVVar7 != (VOID *)0x0) {
95    return 0;
96  }
97  if (BMP_B != 'B') {
98    return 0;
99  }
100 if (BMP_M != 'M') {
101   return 0;
102 }
103 bmpImageSize = (UINTN)local_54;
104 pixelindex = (ulonglong)local_68;
105 bitIndex = (byte)bitsPerPixel & 0x1f;
106 if ((int)bmpHeight < 0) {
107   if ((local_58 != 0) && (local_58 != 3)) {
108     return 0x8000000000000003;
109   }
110   bmpHeight = -bmpHeight;
111   bVar3 = false;
112 }
113 param_2->Height = (UINT16)bmpHeight;
114 param_2->Width = (ushort)bmpWidth;
115 local_88 = bmpImageSize;
116                     /* ; if you're wondering: "why is the decompilation of this next function so
117                        bizarre?
118                        ; what's with all the casts??
119                        ; *** This is where the initial integer overflow happens: ***
120                        *** This routine performs the computation to determine the size of a buffer
121                        to store the BMP image data***
122                        *** We can leverage the fact that there are no checks to compare the values
123                        of bmpHeight and bmpWidth (stored in the BMP image header) against the actual
124                        size of the file
125                        ; Combined with the integer overflow,
126                        ; we can use this to allocate a chunk of desired size on the heap
127                        ; If we allocate a chunk that is too small to contain the entire BmpImage
128                        ; then we can cause a heap overflow
129                        ;
130                        ; BmpHeader->Width is of type UINT32
131                        ; BmpHeight->Height is of type UINT32;
132                        ; there's an imul instruction on UINT32 bmpHeight and UINT32 bmpWidth
133                        ; before this final bmpImageAllocationSize is passed to allocatePool()
134                        ;
135                        ; There are many ways we could exploit this integer overflow for our purposes
136                        ; Here's one way we could leverage the integer overflow to create a small
137                        buffer with allocatePool() and trigger a heap overflow
138                        ; A check is performed to ensure that *if* the var bmpHeight (from
139                        BmpHeader->Height) is a negative value, *then* bmpHeight=-bmpHeight
140                        ; we can use this check to overflow bmpHeight by storing the max value for an
141                        unsigned 32-bit integer in bmpHeight
142                        ; the final 64-bit signed integer  is (UINT64)(bmpWidth * bmpHeight) << 2)
143                        ; If we set bmpHeight = 4294967295
144                        ; and bmpHeight = 2
145                        ; the final bmpImageSize is (1 * 2) << 2 == 8
146                        ; bmpImageSize is passed to allocatePool to allocate a chunk on the heap
147                        ; the allocated chunk is much smaller than the size of the supplied BMP
148                        ; which will lead to a heap overflow
149                         */
150 bmpBitmap = allocatePool((longlong)(int)((uint)(ushort)bmpWidth * (bmpHeight & 0xffff)) << 2);
151 param_2->Bitmap = (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)bmpBitmap;
152 if (bmpBitmap == (void *)0x0) {
153 LAB_80000745:
```

Listing: SystemImageDecoderDxe-5f65d21a-8867-45d3-a41a-526f9fe2c598....

```
                    46 06
80000a01 44 3b d0       CMP         uVar12,ImageBuffer
80000a04 0f 83 1f       JNC         LAB_80000c29
                    02 00 00
80000a0a 44 8b ca       MOV         R9D,pixelsPerRow
                    ;below are each of the subroutines that handle parsing the BM...
                    ; for the following cases:
                    ; case 0: bitsPerPixel == 1
                    ; case 1: bitsPerPixel == 4
                    ; case 2: bitsPerPixel == 8
                    ; case 3: bitsPerPixel == 16
                    ; case 4: bitsPerPixel == 24
                    ; case 5: bitsPerPixel == 32
80000a0d 66 41 3b       CMP         pixelsPerRow,word ptr [R14 + 0x4]
                    56 04
80000a12 0f 83 ef       JNC         LAB_80000c07
                    01 00 00
80000a18 45 33 ed       XOR         cVar16,cVar16

                    LAB_80000a1b                                    XREF[1]:     8000
80000a1b 0f b7 4d de    MOVZX       BMP_input_buffer,word ptr [RBP + bitsPerPixel]
80000a1f 2b cf          SUB         BMP_input_buffer,status
                    ; case 0: bitsPerPixel == 0
80000a21 0f 84 6c       JZ          LAB_80000b93
                    01 00 00
80000a27 83 e9 03       SUB         BMP_input_buffer,0x3
                    ; case 1: bitsPerPixel == 4
80000a2a 0f 84 e7       JZ          LAB_80000b17
                    00 00 00
80000a30 83 e9 04       SUB         BMP_input_buffer,0x4
                    ; case 2: bitsPerPixel == 8
80000a33 0f 84 a2       JZ          LAB_80000adb
                    00 00 00
80000a39 83 e9 08       SUB         BMP_input_buffer,0x8
                    ; case 3: bitsPerPixel == 16
80000a3c 74 4a          JZ          LAB_80000a88
80000a3e 83 e9 08       SUB         BMP_input_buffer,0x8
                    ; case 4: bitsPerPixel == 24
80000a41 74 2b          JZ          LAB_80000a6e
80000a43 83 f9 08       CMP         BMP_input_buffer,0x8
                    ; case 5: bitsPerPixel == 32
80000a46 0f 85 a4       JNZ         LAB_80000bf0
                    01 00 00
80000a4c 41 0f b7       MOVZX       BMP_input_buffer,word ptr [R14 + 0x4]
                    4e 04
80000a51 41 0f af ca    IMUL        BMP_input_buffer,iVar13
80000a55 41 8b c1       MOV         ImageBuffer,R9D
80000a58 41 8b d0       MOV         pixelsPerRow,iVar11
80000a5b 48 03 c8       ADD         BMP_input_buffer,ImageBuffer
80000a5e 8b 04 32       MOV         ImageBuffer,dword ptr [pixelsPerRow + BmpPixel...
```

switch statement for different values of bitsPerPixel

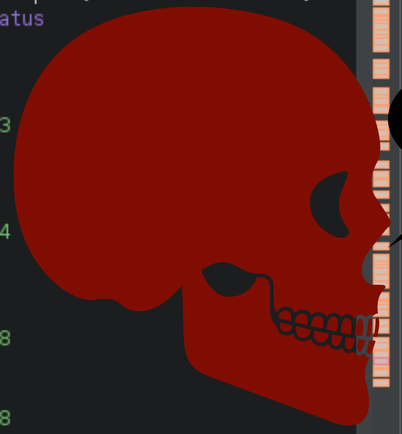
Decompile: BMP_parser - (SystemImageDecoderDxe-5f65d21a-8867-45d3-a...

```
259         pixelsPerRow = 0;
260         uVar12 = (uint)currPixel;
261         status = local_80;
262         if (param_2->Height <= uVar12) break;
263                     /* ;below are each of the subroutines that handle parsing the BMP image
264                        ; for the following cases:
265                        ; case 0: bitsPerPixel == 1
266                        ; case 1: bitsPerPixel == 4
267                        ; case 2: bitsPerPixel == 8
268                        ; case 3: bitsPerPixel == 16
269                        ; case 4: bitsPerPixel == 24
270                        ; case 5: bitsPerPixel == 32 */
271         if (param_2->Width != 0) {
272           do {
273             pixelArrayIndex = (uint)pixelsPerRow;
274             iVar13 = (int)currPixel;
275             iVar11 = (int)pixelindex;
276                     /* ; case 0: bitsPerPixel == 0 */
277             if (bitsPerPixel == 1) {
278               bitIndex = 0;
279               do {
280                 pixelArrayIndex = (uint)pixelsPerRow;
281                 if (param_2->Width <= pixelArrayIndex) break;
282                 bVar5 = 7 - bitIndex;
283                 bitIndex = bitIndex + 1;
284                 uVar1 = (ulonglong)(*(byte *)(pixelindex + (longlong)ImageBuffer) >> (bVar5 & 0x3f))
285                         & 1;
286                 _local_res18 = CONCAT71(CONCAT61(CONCAT51(stack0x0000001b,Palette[uVar1].Red),
287                                                  Palette[uVar1].Green),Palette[uVar1].Blue);
288                 pixelArrayIndex = pixelArrayIndex + 1;
289                 BMP_PixelsArrray[iVar13 * (uint)param_2->Width + pixelsPerRow] = local_res18;
290                 pixelsPerRow = (ulonglong)pixelArrayIndex;
291               } while (bitIndex < 8);
292               pixelArrayIndex = pixelArrayIndex - 1;
293     LAB_80000bed:
294               pixelindex = (ulonglong)(iVar11 + 1);
295             }
296             else {
297               uVar4 = stack0x0000001b;
298                     /* ; case 1: bitsPerPixel == 4 */
299               if (bitsPerPixel == 4) {
300                 bitIndex = *(byte *)(pixelindex + (longlong)ImageBuffer) >> 4;
301                 _local_res18 = CONCAT71(CONCAT61(CONCAT51(stack0x0000001b,Palette[bitIndex].Red),
302                                                  Palette[bitIndex].Green),Palette[bitIndex].Blue);
303                 pixelArrayIndex = pixelArrayIndex + 1;
304                 BMP_PixelsArrray[(uint)param_2->Width * iVar13 + pixelsPerRow] = local_res18;
305                 if (pixelArrayIndex < param_2->Width) {
306                   uVar6 = *(byte *)(pixelindex + (longlong)ImageBuffer) & 0xf;
307                   _local_res18 = CONCAT71(CONCAT61(CONCAT51(uVar4,Palette[uVar6].Red),
308                                                    Palette[uVar6].Green),Palette[uVar6].Blue);
309                   BMP_PixelsArrray
```

LogoFAIL RE

Reverse engineer the ParseBMP function and tear it to shreds

```
                    ; case 0: bitsPerPixel == 1
                    ; case 1: bitsPerPixel == 4
                    ; case 2: bitsPerPixel == 8
                    ; case 3: bitsPerPixel == 16
                    ; case 4: bitsPerPixel == 24
                    ; case 5: bitsPerPixel == 32
80000a0d 66 41 3b       CMP         pixelsPerRow,word ptr [R14 + 0x4]
         56 04
80000a12 0f 83 ef       JNC         LAB_80000c07
         01 00 00
80000a18 45 33 ed       XOR         cVar16,cVar16

                    LAB_80000a1b                                    XREF[1]:     8000
80000a1b 0f b7 4d de    MOVZX       BMP_input_buffer,word ptr [RBP + bitsPerPixel]
80000a1f 2b cf          SUB         BMP_input_buffer,status
                    ; case 0: bitsPerPixel == 0
80000a21 0f 84 6c       JZ          LAB_80000b93
         01 00 00
80000a27 83 e9 03       SUB         BMP_input_buffer,0x3
                    ; case 1: bitsPerPixel == 4
80000a2a 0f 84 e7       JZ          LAB_80000b17
         00 00 00
80000a30 83 e9 04       SUB         BMP_input_buffer,0x4
                    ; case 2: bitsPerPixel == 8
80000a33 0f 84 a2       JZ          LAB_80000adb
         00 00 00
80000a39 83 e9 08       SUB         BMP_input_buffer,0x8
                    ; case 3: bitsPerPixel == 16
80000a3c 74 4a          JZ          LAB_80000a88
80000a3e 83 e9 08       SUB         BMP_input_buffer,0x8
                    ; case 4: bitsPerPixel == 24
80000a41 74 2b          JZ          LAB_80000a6e
80000a43 83 f9 08       CMP         BMP_input_buffer,0x8
                    ; case 5: bitsPerPixel == 32
80000a46 0f 85 a4       JNZ         LAB_80000bf0
         01 00 00
80000a4c 41 0f b7       MOVZX       BMP_input_buffer,word ptr [R14 + 0x4]
         4e 04
80000a51 41 0f af ca    IMUL        BMP_input_buffer,iVar13
80000a55 41 8b c1       MOV         ImageBuffer,R9D
80000a58 41 8b d0       MOV         pixelsPerRow,iVar11
80000a5b 48 03 c8       ADD         BMP_input_buffer,ImageBuffer
80000a5e 8b 04 32       MOV         ImageBuffer,dword ptr [pixelsPerRow + BmpPixel...
80000a61 41 83 c0 04    ADD         iVar11,0x4
                    ; OOB Write: Allocated buffer BmpPixelsArray
                    ; BMPPixelsArray is set to the value of param2->Bitmap
                    ; param2-> Bitmap had already been set to the EFI_GRAPHICS_OU...
                    ; this location  -- BMPPixelsArray -- is the out of bounds bu...
                    ; tldr: the OOB write primitive here with BMPPixelsArray is p...
80000a65 41 89 04 8b    MOV         dword ptr [R11 + BMP_input_buffer*0x4],ImageBu...
80000a69 e9 82 01       JMP         LAB_80000bf0
         00 00

                    LAB_80000a6e                                    XREF[1]:     8000
```




```
347                                                     CONCAT41(uStack_97,
348                                                              (char)(uVar6 >>
349                                                 bitIndex))))));
350             local_res18._0_2_ = CONCAT11(bitIndex,local_res18.Blue);
351             bitIndex = FUN_800005c0(CONCAT17(cStack_89,
352                                              CONCAT16(bStack_8a,
353                                                       CONCAT15(cStack_8b,
354                                                                CONCAT41(uStack_8f,local_90))))
355                                     );
356             pixelindex = (ulonglong)(iVar11 + 2);
357             goto LAB_80000af9;
358           }
359             /* ; case 4: bitsPerPixel == 24 */
360           if (bitsPerPixel == 0x18) {
361             _local_res18 = (ulonglong)
362                            CONCAT61(CONCAT51(stack0x0000001b,
363                                              *(undefined *)
364                                               (pixelindex + 2 + (longlong)ImageBuffer)),
365                                     *(undefined *)(pixelindex + 1 + (longlong)ImageBuffer)) <<
366                            8;
367             bitIndex = *(byte *)(pixelindex + (longlong)ImageBuffer);
368             pixelindex = (ulonglong)(iVar11 + 3);
369             goto LAB_80000af9;
370           }
371             /* ; case 5: bitsPerPixel == 32 */
372           if (bitsPerPixel == 0x20) {
373             pEVar1 = (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)(pixelindex + (longlong)ImageBuffer);
374             pixelindex = (ulonglong)(iVar11 + 4);
375               /* ; OOB Write: Allocated buffer BmpPixelsArray
376                  ; BMPPixelsArray is set to the value of param2->Bitmap
377                  ; param2-> Bitmap had already been set to the EFI_GRAPHICS_OUTPUT_BLT_PIXEL *
378                  array allocated with the allocatePool call [same allocatePool call with
379                  integer overflow]
380                  ; this location  -- BMPPixelsArray -- is the out of bounds buffer located
381                  past the end of the allocated heap chunk;
382                  ; tldr: the OOB write primitive here with BMPPixelsArray is part 2 in the
383                  exploit chain combining the integer overflow to heap overflow vulnerability
384                   */
385             BMP_PixelsArrray[(uint)param_2->Width * iVar13 + pixelsPerRow] = *pEVar1;
386           }
387         }
388       }
389         uVar6 = (uint)pixelindex;
390         uVar12 = (uint)currPixel;
391         pixelsPerRow = (ulonglong)(pixelArrayIndex + 1);
392       } while (pixelArrayIndex + 1 < (uint)param_2->Width);
393     }
394     pixelindex = (ulonglong)(uVar6 + (-(uint)((uVar6 & 3) != 0) & 4 - (uVar6 & 3)));
395     currPixel = (ulonglong)(uVar12 + (int)cVar16);
396   }
397 }
398 if (Palette != (EFI_GRAPHICS_OUTPUT_BLT_PIXEL *)0x0) {
399   (*gBS_7->FreePool)(Palette);
400 }
```

LogoFAIL RE

Reverse engineer the ParseBMP function and tear it to shreds

The true Deus ex machina is the realization that a UEFI firmware update does not patch the underlying integer overflow vulnerability in the BMP image parsing function of the boot logo parsing DXE driver. My fully patched laptop is still vulnerable to my LogoFAIL exploit…

Me, last week

LogoFAIL RE + xdev

Hardware testing and debugging

- Reversed the UEFI firmware for my Lenovo laptop and found vulnerability leading to OOB write in Lenovo's SystemImageDecoder.dxe image parser when processing boot logo BMP image
- Created two test environments for hardware debugging on Intel dev boards — Up Xtreme and Up Squared
- Repeated the same process of reversing UEFI firmware on Up Xtreme and Up Squared boards
- Set up hardware debugging for both test boards:
  - Patched UEFI firmware to enable DCI debugging on Up Squared board
  - Modified BIOS settings for Up Xtreme board thru BIOS configuration menu to also enable DCI debugging on that board
  - Intel System Studio (no longer available publicly, copy provided by researcher colleague) and Windbg

LogoFAIL xdev

Hardware testing and debugging

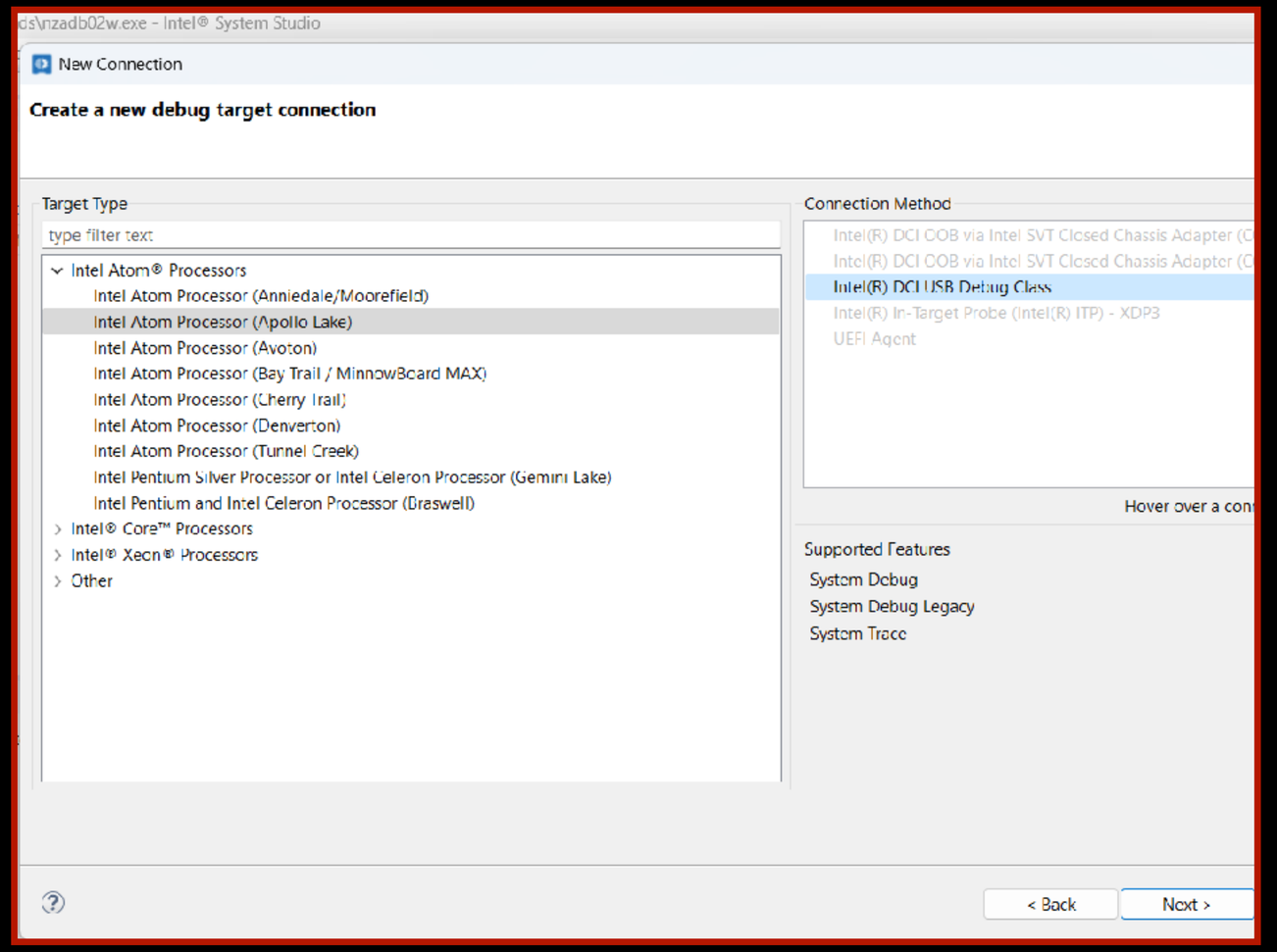
- DCI Debugging with the Aaeon Up Xtreme and Aaeon upSquared boards

Intel System Debugger Target Indicator - confirming DCI connection status

LogoFAIL xdev

Hardware testing and debugging

- Intel System Studio - create new debug target connection to Up Squared board with DCI USB cable



Intel System Studio - create new debug target connection

LogoFAIL xdev

Hardware testing and debugging

it's called
hardware debugging because
it's hard and sometimes your setup
doesn't work
RIP

Administrator: Command Prompt - windbg_iajtag_console.bat

```
    forensic.remove_init_breakpoint()#Remove Initialization Phase0 breakpoint
    forensic.dbgprint()#Show content of dbgprint ring buffer
    forensic.get_kernel_base()#get kernel address
    forensic.get_kd_version_block()#get kernel structure required for kernel debug WinDbg
    forensic.get_module_base(addr)#find MZ signature. use !dh modulebase to get image header
    forensic.dump_pt(itp.threads[0],r"c:\temp\ptdump")#dump single range pt buffer on thread using dma
    forensic.image_scans()# scans image context on all GPCs

3. To run KD shell
    kd = itpkd.KDShell()#connect to KD shell when Windows is running
    kd.execute_line("u", timeout=60)#execute disassembly command via KD

4. Event-based debugging scripts
    acpi.amli_trace()#ACPI tracing script displays the names of AML methods as they are evaluated

Connecting to IPC API....
IPC-CLI: 1.2011.1709.100, OpenIPC:Main (rev 650678) : 1.2011.4508.100
Initializing IPC API....

Info - OpenIPC configuration 'BXTP_DCI_OpenRC'; run control plug-in(s): 'OpenRC'; probe plug-in(s): 'OpenDCI'
Info - DCI: A DCI device has been detected, attempting to establish connection
Info - DCI: Target connection has been fully established
Warning - DCI: Jtag.TclkRate configured = 11000000, actual = 12500000 for device 0x00012000 (PCH)
Warning - DCI: Jtag.TclkRate configured = 11000000, actual = 12500000 for device 0x00012001 (CPU)
Info - Added new debug port 0 using probe plug-in 'OpenDCI'
Info - Detected BXTP_CLTAPC F1 on JTAG chain 0 at position 0
Warning - No TAP devices detected on JTAG chain 1
>>>
```

Hardware debugging progress with up Xtreme and up Squared boards blocked by DCI connection issues

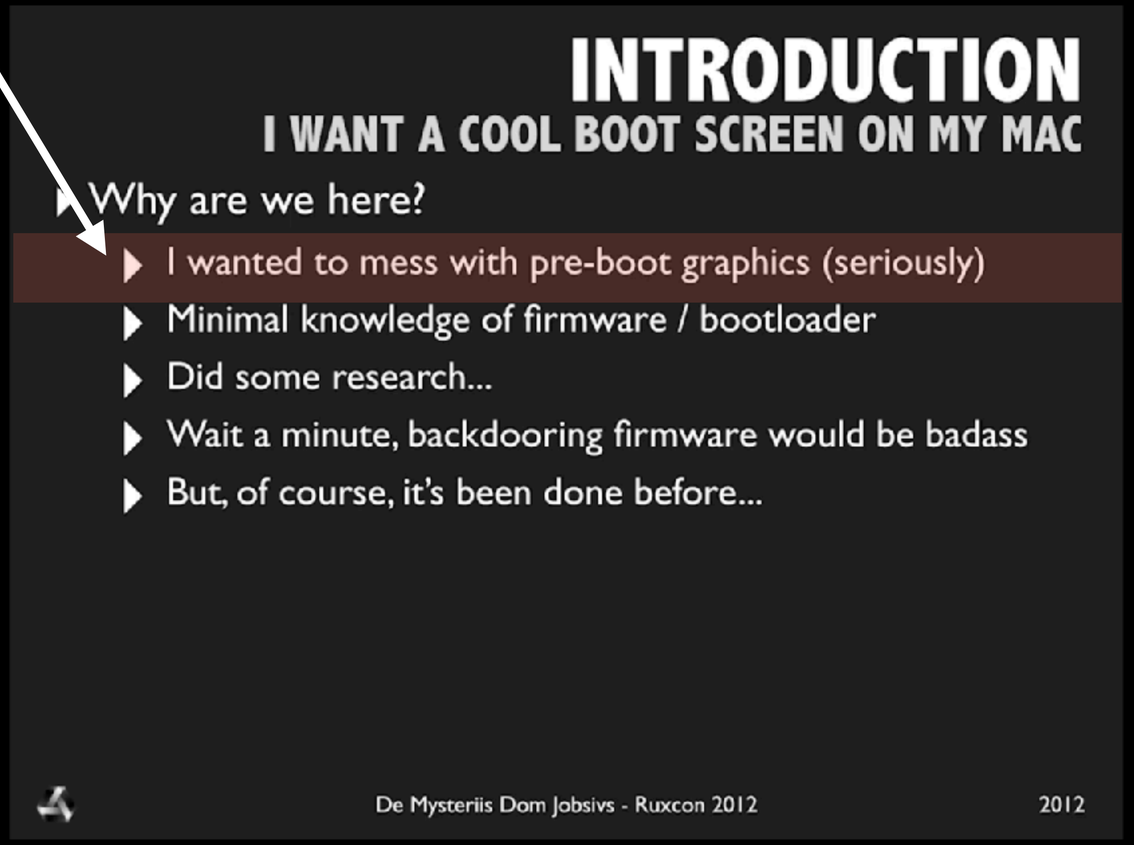
Stage 2: Malicious PCI Option ROM

PCI Option ROM Exploits

Prior work

- “De Mysteriis Dom Jobsivs: Mac EFI Rootkits” by snare, Ruxcon 2012
  https://youtu.be/XcFvgAsfdqg?si=VFJ7Ap85CUI1K6-o
- “Thunderstrike 2: Sith Strike” by Trammell Hudson, Xeno Kovah and Corey Kallenberg, Defcon 23/ BlackHat2015
  https://trmm.net/Thunderstrike2_details/
  https://youtu.be/CtEdfMP6rJo?si=ry91CYe0Qrq64_RH
- Many other siq examples, see the Dark Mentor timeline:
  https://darkmentor.com/timeline.html

relatable content!!



“De Mysteriis Dom Jobsivs: Mac EFI Rootkits,” snare, Ruxcon, 2012

PCI Option ROMs

UEFI: Malicious OpROM mitigations

- Unsigned Option ROMs are not loaded/run by default when UEFI Secure Boot is enabled
- Executing a malicious unsigned option ROM on UEFI requires that Secure Boot be bypassed…

PCI Option ROM Exploits

Primary references for GOP Complex PCI Option ROM dev

“UEFI Option ROM Bootkit,” gfoudree, x86sec https://x86sec.com/posts/2022/09/26/uefi-oprom-bootkit/	“Executing custom Option ROM on D34010WYK and persisting code in UEFI Runtime Services” , Casual Hacking, Jan 4, 2020, https://casualhacking.io/blog/2020/1/4/executing-custom-option-rom-on-nucs-and-persisting-code-in-uefi-runtime-services	“Using an Option ROM to overwrite SMM/SMI handlers in QEMU,” Casual Hacking, Jan 4, 2020 https://casualhacking.io/blog/2019/12/3/using-optionrom-to-overwrite-smmsmi-handlers-in-qemu

PCI Option ROMs

UEFI: Malicious OpROM mitigations

- Unsigned Option ROMs are not loaded/run by default when UEFI Secure Boot is enabled
- Executing a malicious unsigned option ROM on UEFI requires that Secure Boot be bypassed…
- First stage payload (LogoFAIL exploit) must disable Secure Boot:
  - Change value of NVRAM variable to disable SecureBoot
  - Modify PCD to allow for loading unsigned Option ROMs

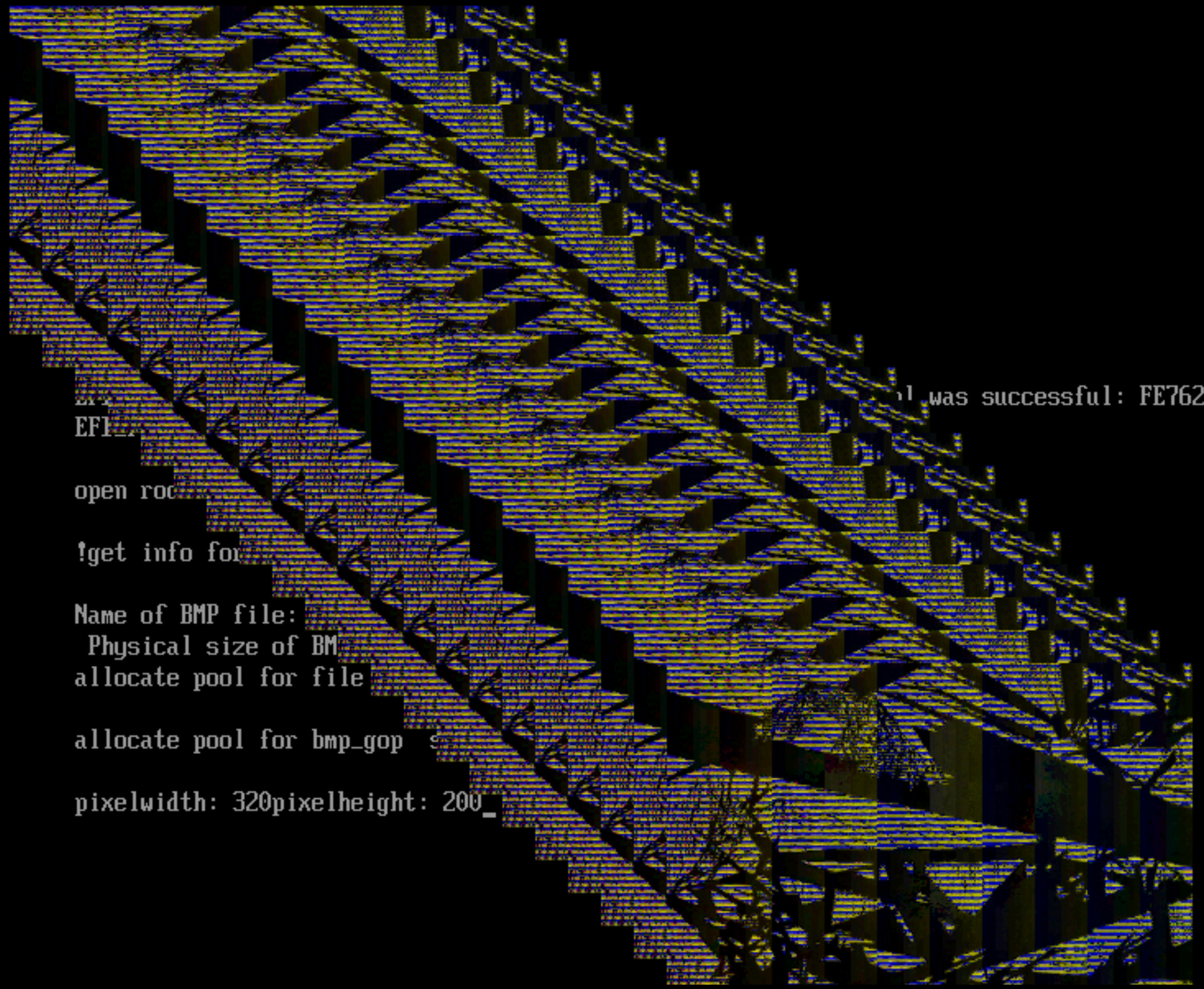
GOP Complex: Malicious PCI Option ROM

Graphics manipulation

- Malicious PCI Option ROM: graphical payload that draws BMP image of my art to the video framebuffer
- Use template for creating a test option ROM e1000e NIC [“UEFI Option ROM Bootkit,” gfoudree, x86sec] and EfiRom edk2 utility for creating Option ROM from UEFI driver
- Hook GOP functions to alter output of image drawn to the video framebuffer

```
void gop_alter(EFI_GRAPHICS_OUTPUT_BLT_PIXEL *bmp_gop, UINTN bmpgopsize){
    UINTN i = 0;
    for (i=0; i<(bmpgopsize/sizeof(EFI_GRAPHICS_OUTPUT_BLT_PIXEL)); i++){
        bmp_gop[i].Red=bmp_gop[i].Red & i;
        bmp_gop[i].Green=bmp_gop[i].Green & i;
        bmp_gop[i].Blue=bmp_gop[i].Blue & ~(i);
    }
}

EFI_STATUS
EFIAPI
GOPComplexEntryPoint (
    IN EFI_HANDLE           ImageHandle,
    IN EFI_SYSTEM_TABLE     *SystemTable
    )
```


curr pixel values: red=F blue=39 green=3ACurr pixel index: 241
curr pixel values: red=1E blue=19 green=1CCurrent pixel: AF1E1C19altered pixel index: 241
curr pixel values: red=7 blue=19 green=1CCurr pixel index: 242
curr pixel values: red=1C blue=17 green=1BCurrent pixel: AF1C1B17altered pixel index: 242
curr pixel values: red=7 blue=17 green=1BCurr pixel index: 243
green=1ACurrent pixel: AF1D1A16altered pixel index: 243
en=1ACurr pixel index: 244
=19Current pixel: AF1D1916altered pixel index: 244
Curr pixel index: 245
rrent pixel: AF1D1916altered pixel index: 245
pixel index: 246
pixel: AF1D1A17altered pixel index: 246
index: 247
l: AF1D1C18altered pixel index: 247
: 248
AF1C1C17altered pixel index: 248
249
C17altered pixel index: 249
ltered pixel index: 250
pixel index: 251
el index: 252
lex: 253
: 254
: 255


was successful: FE76288
open roo
!get info fo
Name of BMP file:
Physical size of BM
allocate pool for file
allocate pool for bmp_gop
pixelwidth: 320pixelheight: 200

Stage 3: EBC Polymorphic engine

EBC - EFI Byte Code

Why EBC?

- EBC was a natural fit as the final architecture to choose for this project because of the inherent variability/malleability of natural indexing and the EBC spec itself
- EBC aims to become something of a tower of Babel: a platform-agnostic architecture specification for PCI option ROM implementation; it uses natural-indexing to adjust the width of its instructions (32-bit or 64-bit) depending on the architecture of the host
- EBC is an intermediate language (like LLVM byte code, Java byte code, [insert your favorite byte code here]) and it is run in the EFI Byte Code Virtual Machine (EBCVM)

UEFI generation 3: EBC

UEFI EBC architecture details

- EBCVM uses 8 general purposes registers:
  - R0-R7
- EBCVM has 2 dedicated registers:
  - IP (instruction pointer)
  - F (Flags register)
- Natural indexing: uses a natural unit to calculate offsets of data relative to a base address, where a natural unit is defined as:
  - Natural unit == sizeof (void *)

Table 22.4 Index Encoding

Bit #	Description
N	Sign bit (sign), most significant bit
N-3..N-1	Bits assigned to natural units (w)
A..N-4	Constant units (c)
0..A-1	Natural units (n)

As shown in the Table above for a given encoded index, the most significant bit (bit N) specifies the sign of the resultant offset after it has been calculated. The sign bit is followed by three bits (N-3..N-1) that are used to compute the width of the natural units field (n). The value (w) from this field is multiplied by the index size in bytes to determine the actual width (A) of the natural units field (n). Once the width of the natural units field has been determined, then the natural units (n) and constant units (c) can be extracted. The offset is then calculated at runtime according to the following equation:

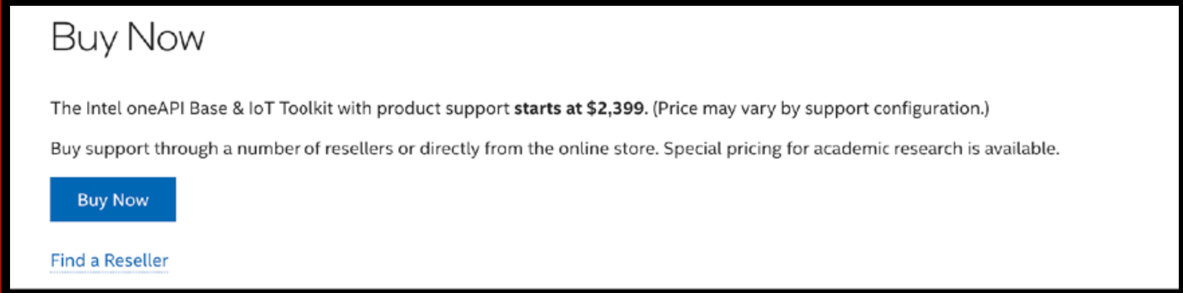
```
Offset = (c + n * (sizeof (VOID *))) * sign
```

Source: “UEFI Spec, Chapter 22: EFI Byte Code Virtual Machine,” https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#natural-indexing

EBC - EFI Byte Code

If EBC is so great then why haven't I heard of it?

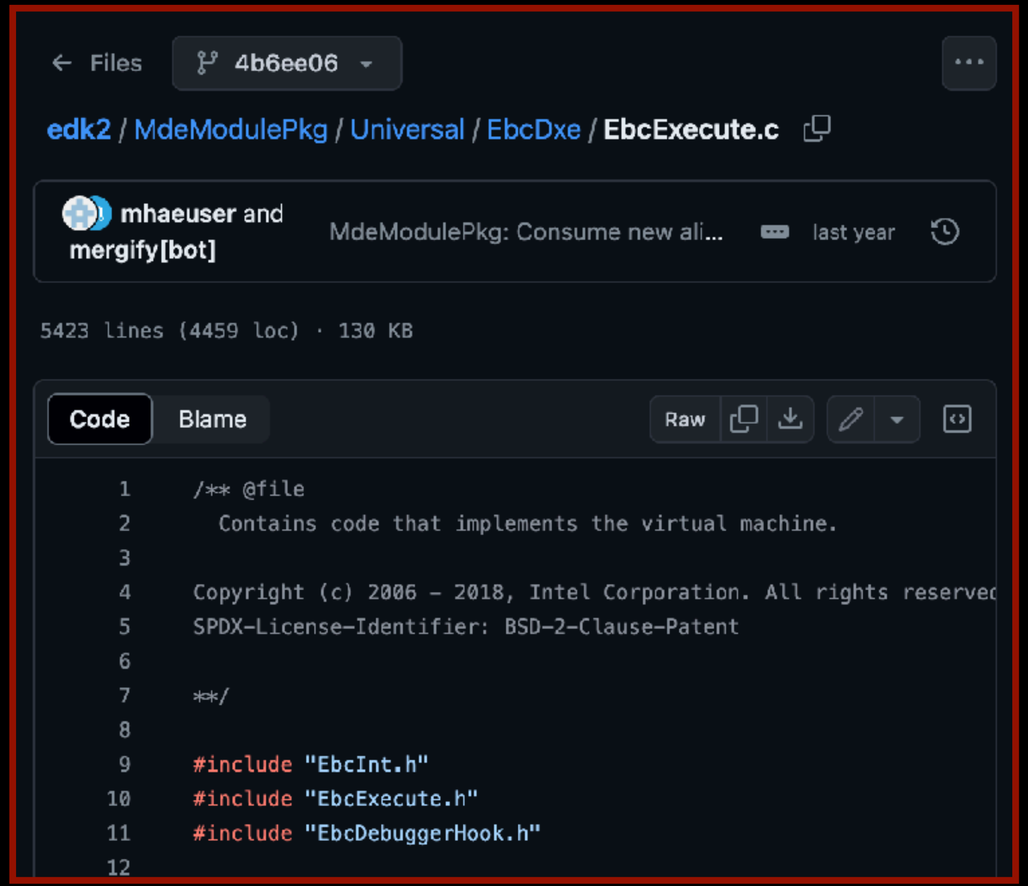
- Only one compiler specifically designed to target valid EBC binaries: the proprietary Intel C compiler for EBC
- This proprietary Intel C compiler for EBC was available for the low price of $995 [to my knowledge, it is no longer available; now the page on the Intel Products site redirects to an IoT toolkit for $2399]
- Open-source options are available … sort of
  - fasm-ebc is the closest open-source version to the Intel C compiler for EBC but it can't handle edge cases for encoding instructions with natural-indexing [see this issue in the *archived* fasm-ebc GitHub repo:]
- Very few in-the-wild reference EBC images
- EBC is technically "no longer part of the spec"
  - Chapter 22 doesn't exist. Chapter 22 never existed.



EBC - EFI Byte Code

If EBC is a dead ISA with little to no reference implementations why are you talking about it now?

- What if there were legacy/deprecated features lingering in a codebase for years…
- What if IBVs/OEMs were slow to patch platform firmware and remove legacy/deprecated features…
- EBC interpreter is still part of the main branch in Tianocore's edk2
- IBVs/OEMs fork edk2, along with the EBC interpreter…
  - … then a lot of machines might have the EBC interpreter, and can run EBC binaries
- Just because this feature is hardly (if ever) used, doesn't mean it can't be leveraged
- To be continued… [in VX-Underground Black Mass, vol. 3]



UEFI generation 3: EBC

RE and development tools

- Open-source version of the EBC compiler: fasm-ebc https://github.com/pbatard/fasmg-ebc
- Hex editor (xxd, hexdump)
- ebcvm: https://github.com/yabits/ebcvm
- Ghidra with efiSeek and ghidra-firmware-utils and an EBC plugin:
  - https://github.com/meromwolff/Ghidra-EFI-Byte-Code-Processor/

EBC - EFI Byte Code

Is that an EBC driver in the UEFI firmware of my Lenovo Thinkpad E16 Gen 1 or is it just happy to see me?

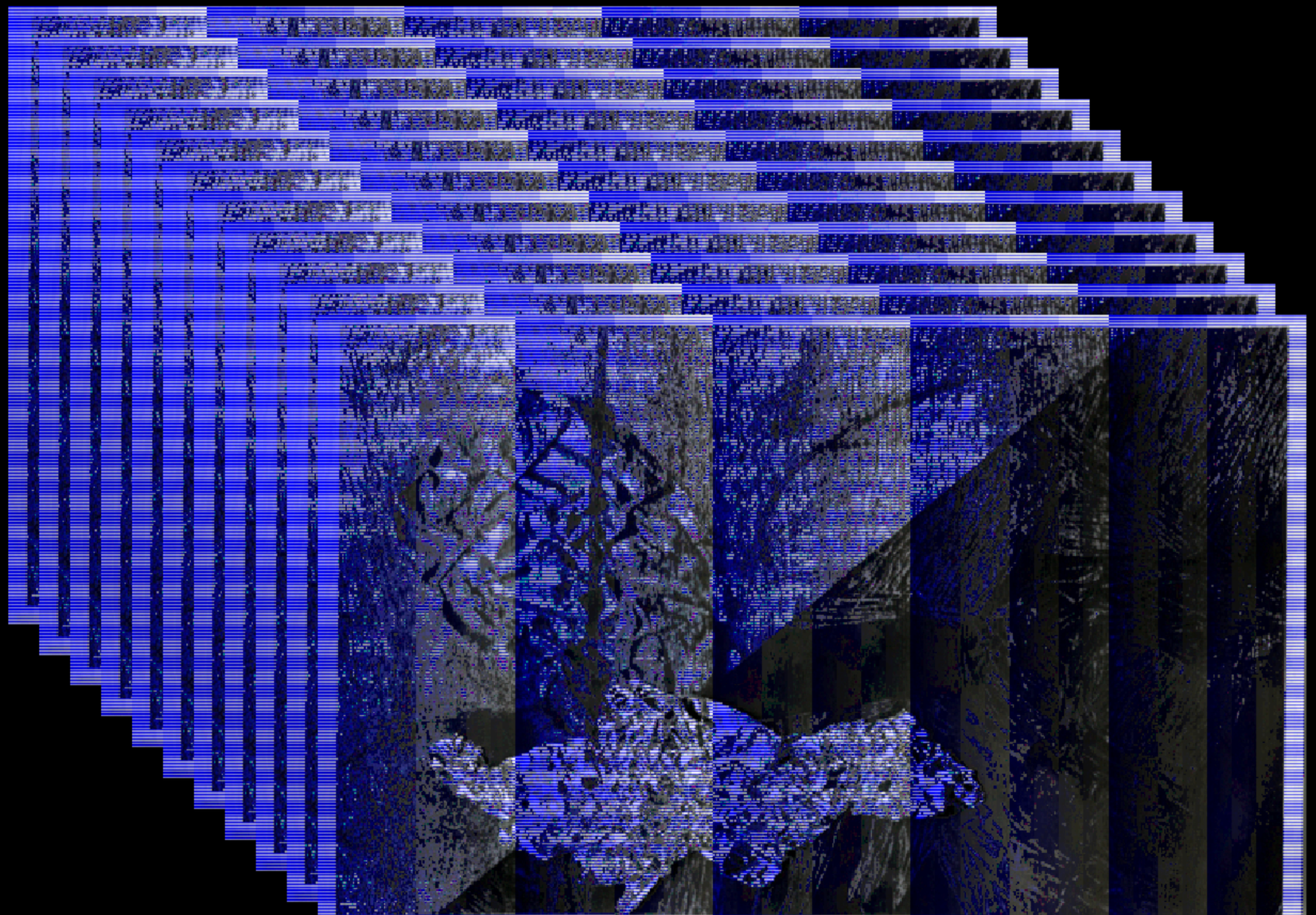
Name	Action	Type	Subtype	Text
Ps2MouseDxe		File	DXE driver	Ps2MouseDxe
WdtAppDxe		File	DXE driver	WdtAppDxe
AcpiDebugDxe		File	DXE driver	AcpiDebugDxe
AmtSaveMebxConfig		File	DXE driver	AmtSaveMebxConfig
AmtPetAlertDxe		File	DXE driver	AmtPetAlertDxe
AsfTable		File	DXE driver	AsfTable
SecureEraseDxe		File	DXE driver	SecureEraseDxe
AmtInitDxe		File	DXE driver	AmtInitDxe
Mebx		File	DXE driver	Mebx
SerialOverLan		File	DXE driver	SerialOverLan
OneClickRecovery		File	DXE driver	OneClickRecovery
RemotePlatformErase		File	DXE driver	RemotePlatformErase
PciHotPlug		File	DXE driver	PciHotPlug
UsbTypeCDxe		File	DXE driver	UsbTypeCDxe
EbcDxe		File	DXE driver	EbcDxe
DXE dependency section		Section	DXE dependency	
PE32 image section		Section	PE32 image	
UI section		Section	UI	
Version section		Section	Version	
DmarAcpiTable		File	Freeform	DmarAcpiTables
Thc		File	DXE driver	Thc
QuickSpi		File	DXE driver	QuickSpi
WdtDxe		File	DXE driver	WdtDxe
CastroCovePmicNvm		File	DXE driver	CastroCovePmicNvm
FaultTolerantWriteDxe		File	DXE driver	FaultTolerantWriteDxe
DxeDgOpregionInit		File	DXE driver	DxeDgOpregionInit
ScsiBus		File	DXE driver	ScsiBus
ScsiDisk		File	DXE driver	ScsiDisk
SataController		File	DXE driver	SataController
BiosGuardServices		File	SMM module	BiosGuardServices
BoardInfoDxe		File	DXE driver	BoardInfoDxe
BoardInfoSmm		File	SMM module	BoardInfoSmm
DxeBoardInit		File	DXE driver	DxeBoardInit
SecureBIOCamera		File	DXE driver	SecureBIOCamera_Realtek
SecureBIOCamera_Sonix		File	DXE driver	SecureBIOCamera_Sonix
SecureBIOCamera_Sunplus		File	DXE driver	SecureBIOCamera_Sunplus

Parser | FIT | Security | Search | Builder

```
Unicode text "SystemImageDecoder" found in SystemImageDecoderDxe/UI section at header-offset 04h
Unicode text "Ebc" found in UefiPxeBcDxe/UI section at header-offset 10h
Unicode text "Ebc" found in Setup/PE32 image section at header-offset 68CDEh
Unicode text "Ebc" found in Setup/PE32 image section at header-offset 68CF5h
Unicode text "Ebc" found in LenovoSecureSuiteApp/PE32 image section at header-offset 2CDA1h
Unicode text "Ebc" found in EbcDxe/UI section at header-offset 04h
Unicode text "Ebc" found in Padding at header-offset 42D6FDh
```

EbcDxe in Lenovo Thinkpad E16 Gen1 UEFI BIOS

Conclusion and Future Work



GOP Complex v0

What did you learn at vx school today?

- Improved UEFI RE + xdev skills, leveraging knowledge of BMP file format, GOP functions and boot logo image parsing routines —> symbiosis between artistic practice and UEFI RE
- Successfully turned my art into a graphical payload for a malicious PCI option ROM
- Improved skills in building more complex exploit chains in UEFI and developing exploits for new attack vectors: PCI Option ROMs
- UEFI can be leveraged as an environment for creative expression
  - VX isn’t dead, it’s undead, welcome back

Conclusion

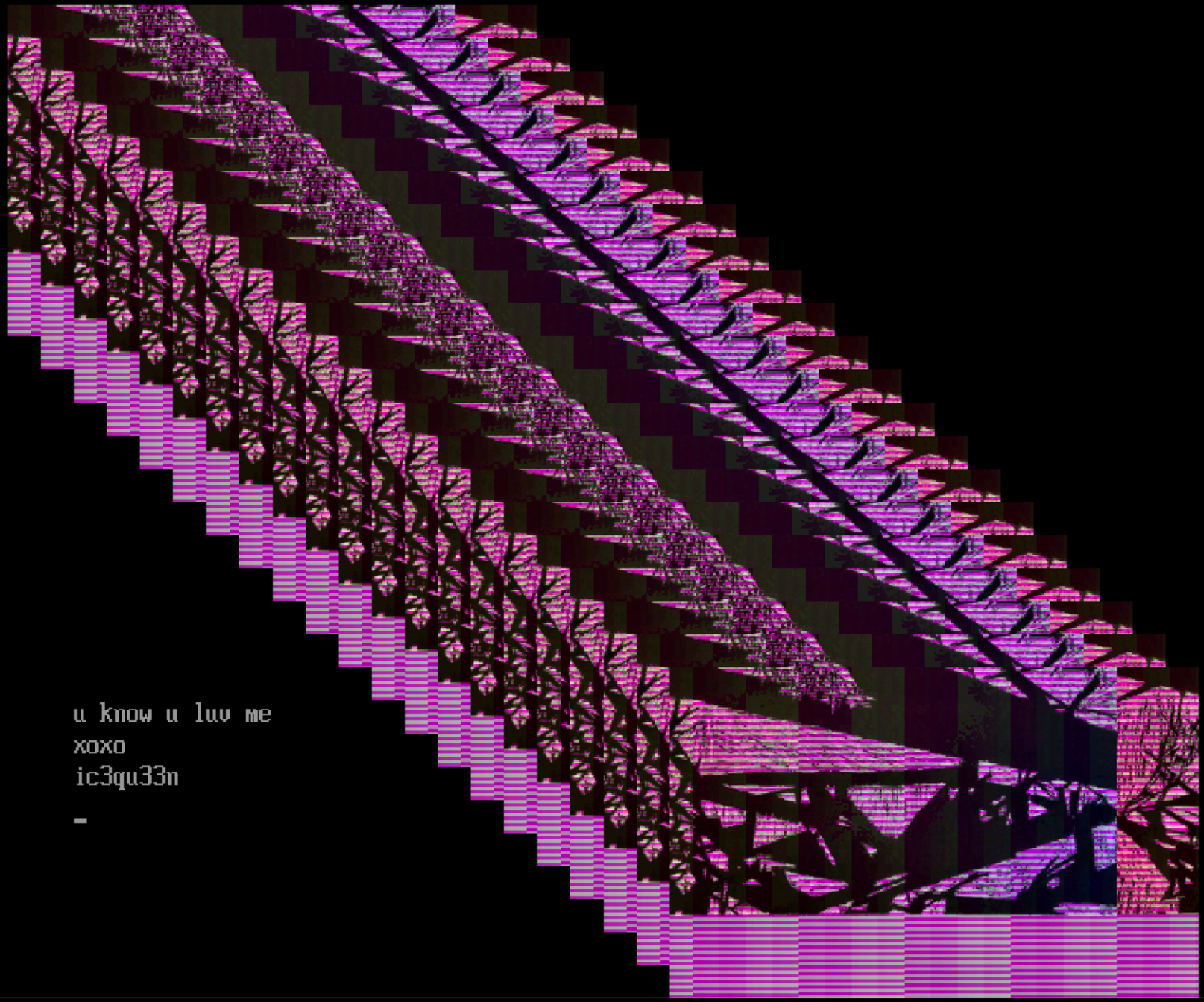
GOP Complex

- UEFI can be understood as its own ecosystem between the OS and onboard (i.e. SPI flash chip-resident) firmware. It operates like an intermediary OS in and of itself. Thus, in order to write effective UEFI-targeting exploits, we have to understand how to manipulate data structures within UEFI.
- My artistic practice and creative projects were *essential* to understanding code patterns/structs during RE process of vulnerable LogoFAIL driver SystemImageDecoder (BRLY-LOGOFAIL-2023-027/CVE-2023-5058)

Future Work

What's next for GOP Complex

- Sacrificing the Lenovo Thinkpad E16 Gen1 to the malware daemons:
  - Testing my LogoFAIL exploit on the machine —> ChipSec script
  - LogoFAIL exploit breakdown — malicious BMP image:
    - crafted values for BMP_IMAGE_HEADER->Width and BMP_IMAGE_HEADER->Height to trigger integer overflow and subsequent allocation of BmpPixelsArray with too small size
    - BitsPerPixel value set to 0x20 —>satisfies condition to reach code path for OOB write
  - UEFI shellcode payload —> testing UEFI heap feng shui techniques to achieve code execution from heap overflow
- Further develop my malicious PCI Option ROM:
  - Translate malicious PCI Option ROM to EBC
  - Test theory that I can leverage the EBCVM as a polymorphic engine
- Explore new attack vectors for replication/propagation of the virus
  - Reflashing graphics cards



u know u luv me
xoxo
ic3qu33n

Q & A