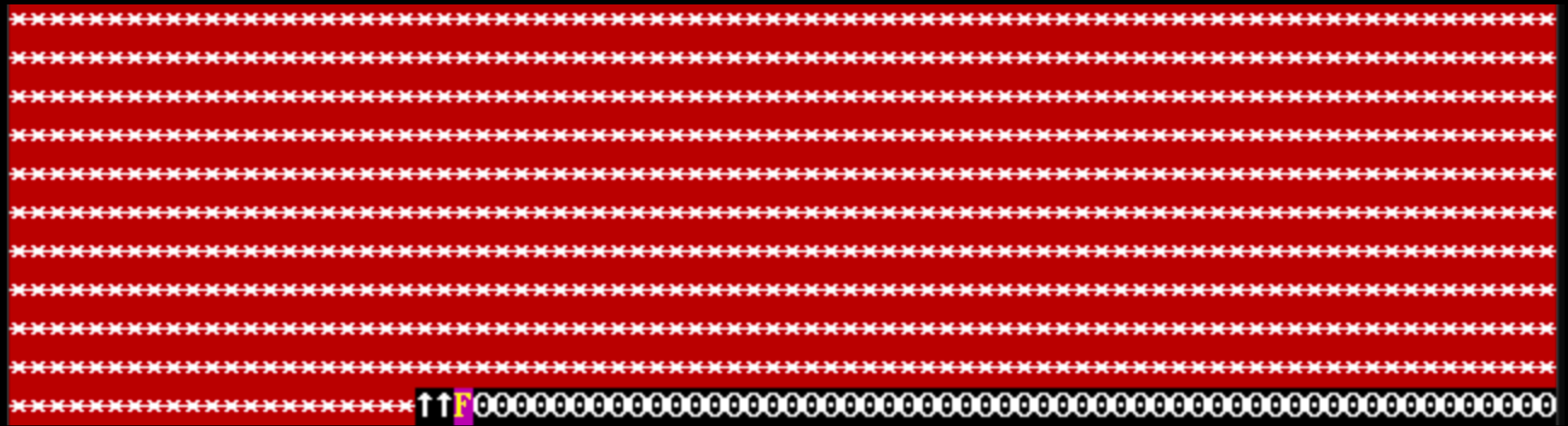
Reflections on UEFI and The Task of the Translator

The art of binary golf - elegant assembly language programming, malware art, and using cross-architecture UEFI quines as a framework for UEFI exploit development

Nika Korchok Wakulich (ic3qu33n)
Dartmouth CS59, Guest Lecture, Fall 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
Mastodon: ic3qu33n@infosec.exchange
Website: https://ic3qu33n.fyi/
GitHub: @ic3qu33n and @nikaroxanne
bsky: @ic3qu33n

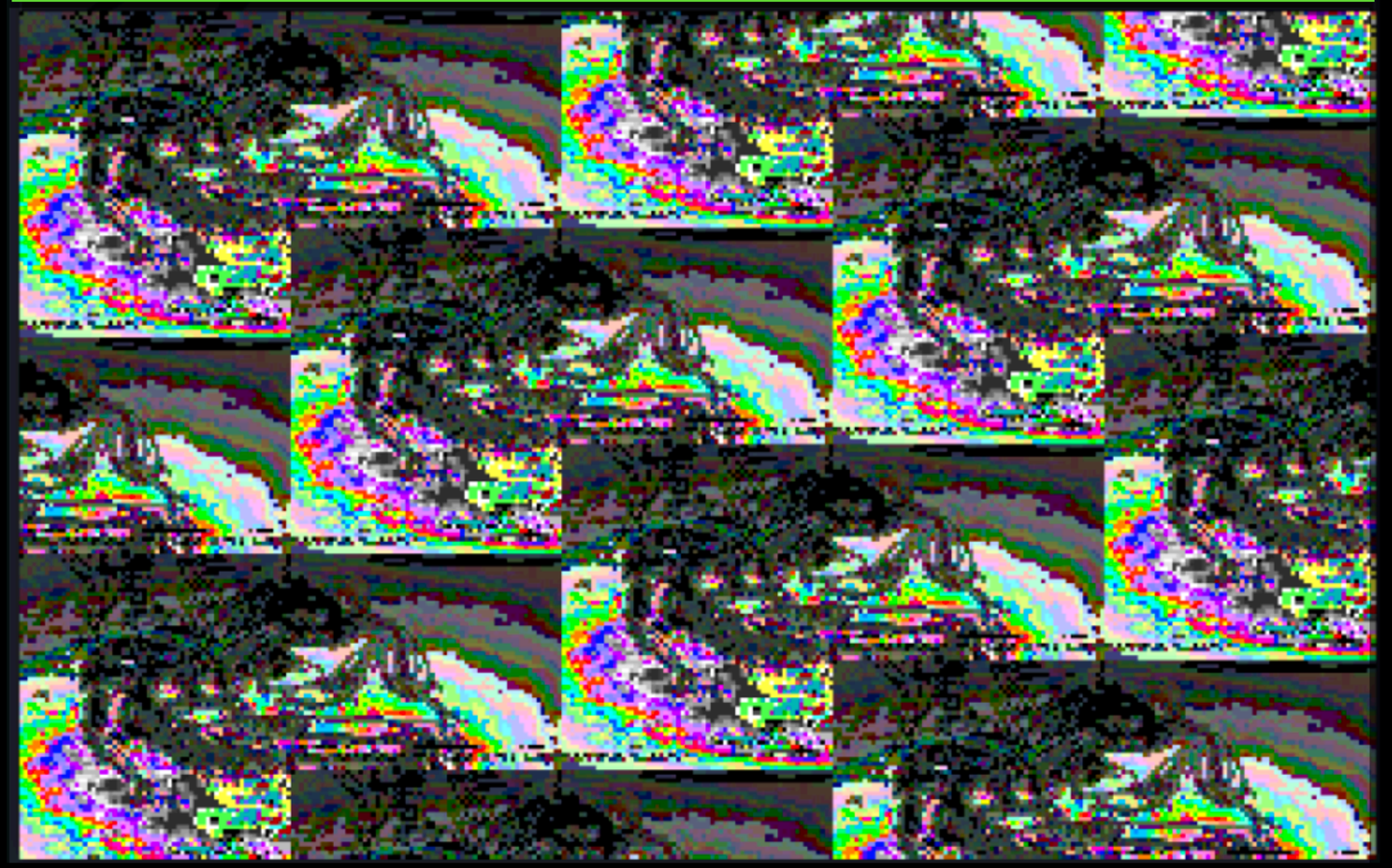
Security Consultant at Leviathan Security Group
Reverse engineer + artist + hacker
I <3 UEFI, hardware hacking, binary golf, binary exploitation, skateboarding,
learning languages, making art, writing programs in assembly languages, etc.

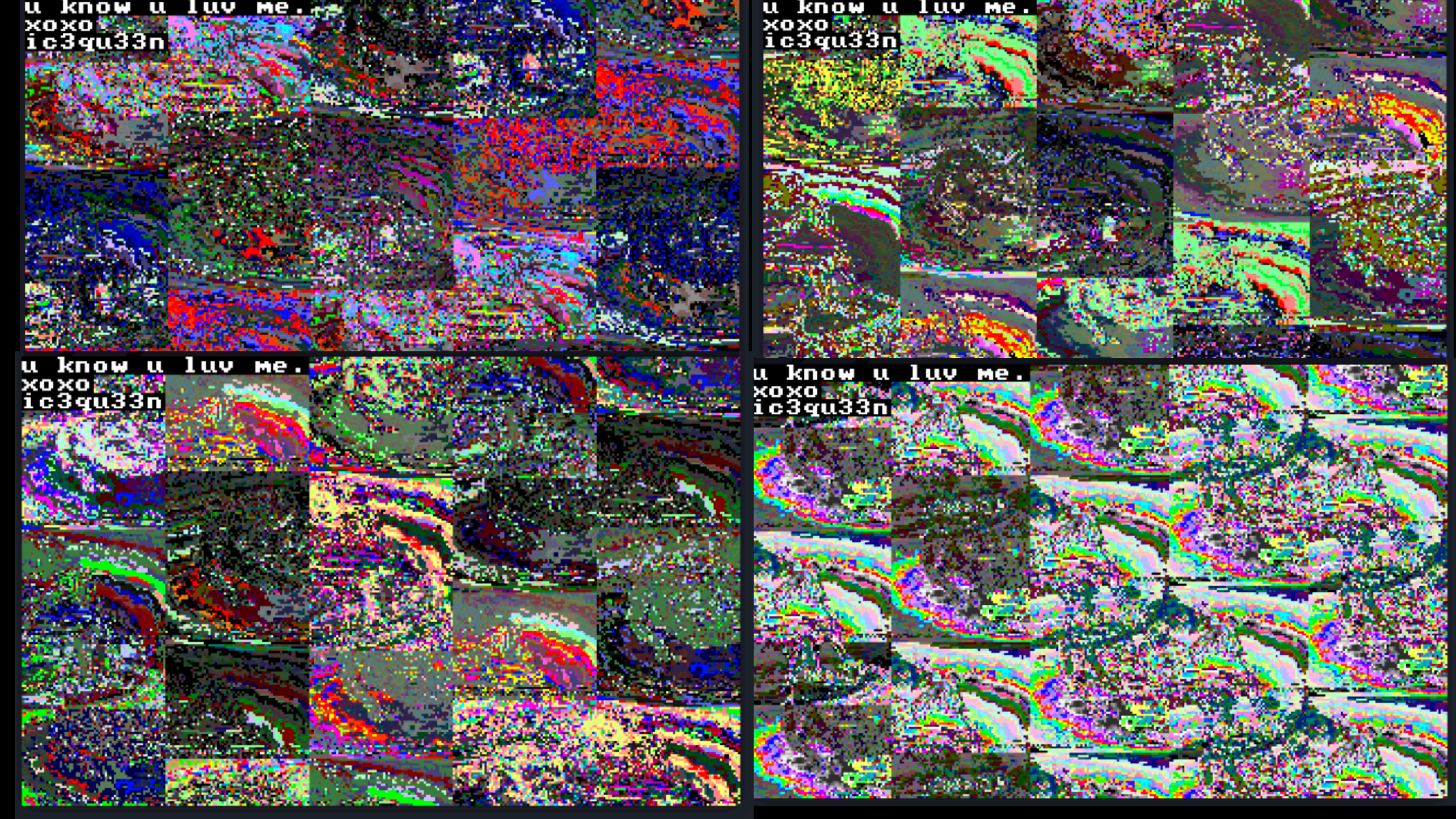
greetz 2 the following for their support w this talk:
0day (@0day_simpson), netspooky (@netspooky), dnz (@dnoiz1), zeta
Xeno Kovah (@xenokovah), Alex Matrosov (@matrosov), Emily
The team at Leviathan
Sergey Bratus and Dartmouth

leviathan security group

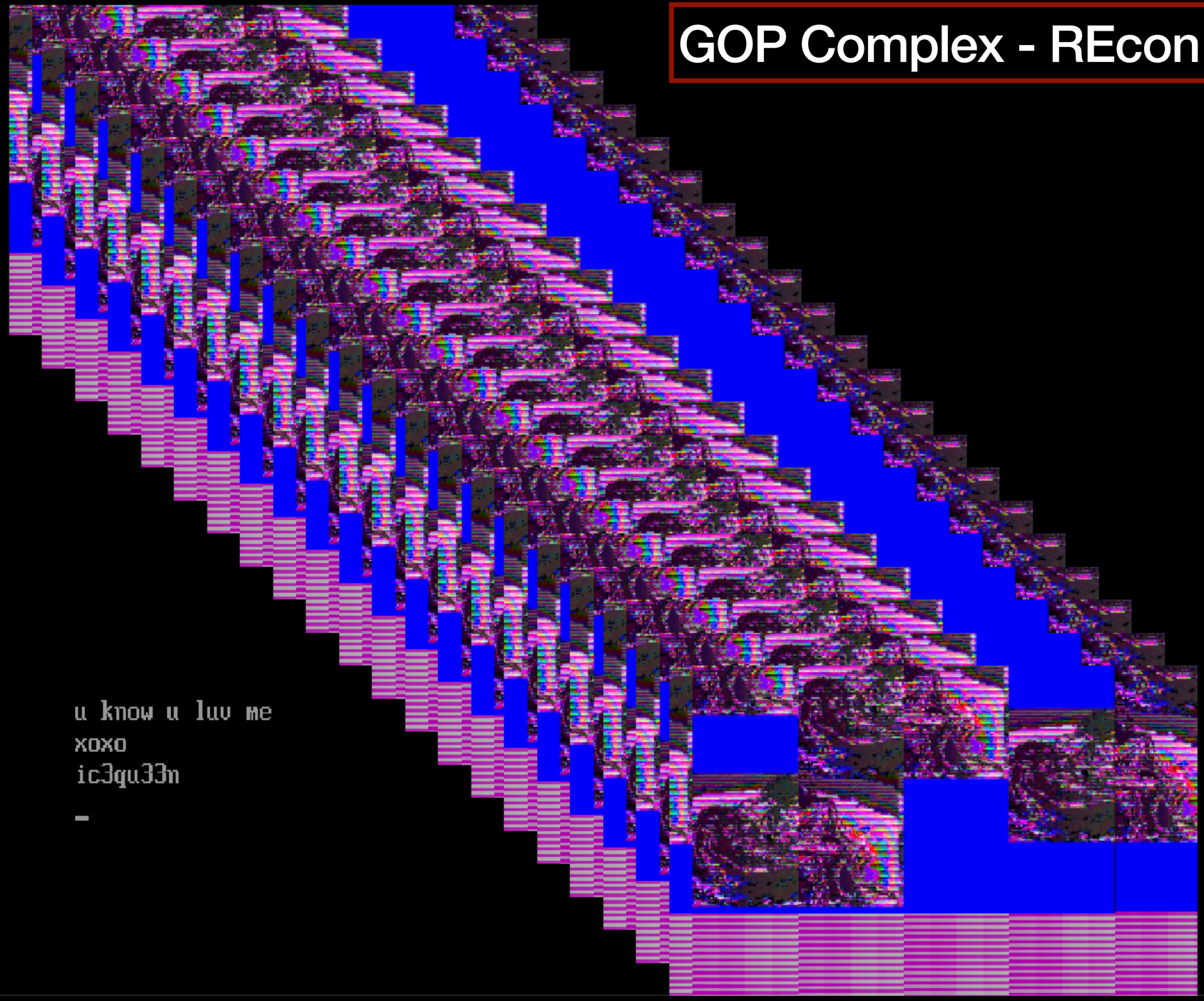
Prior work: malware art

“Michelangelo REanimator,” MBR Bootkit REcon 2023:

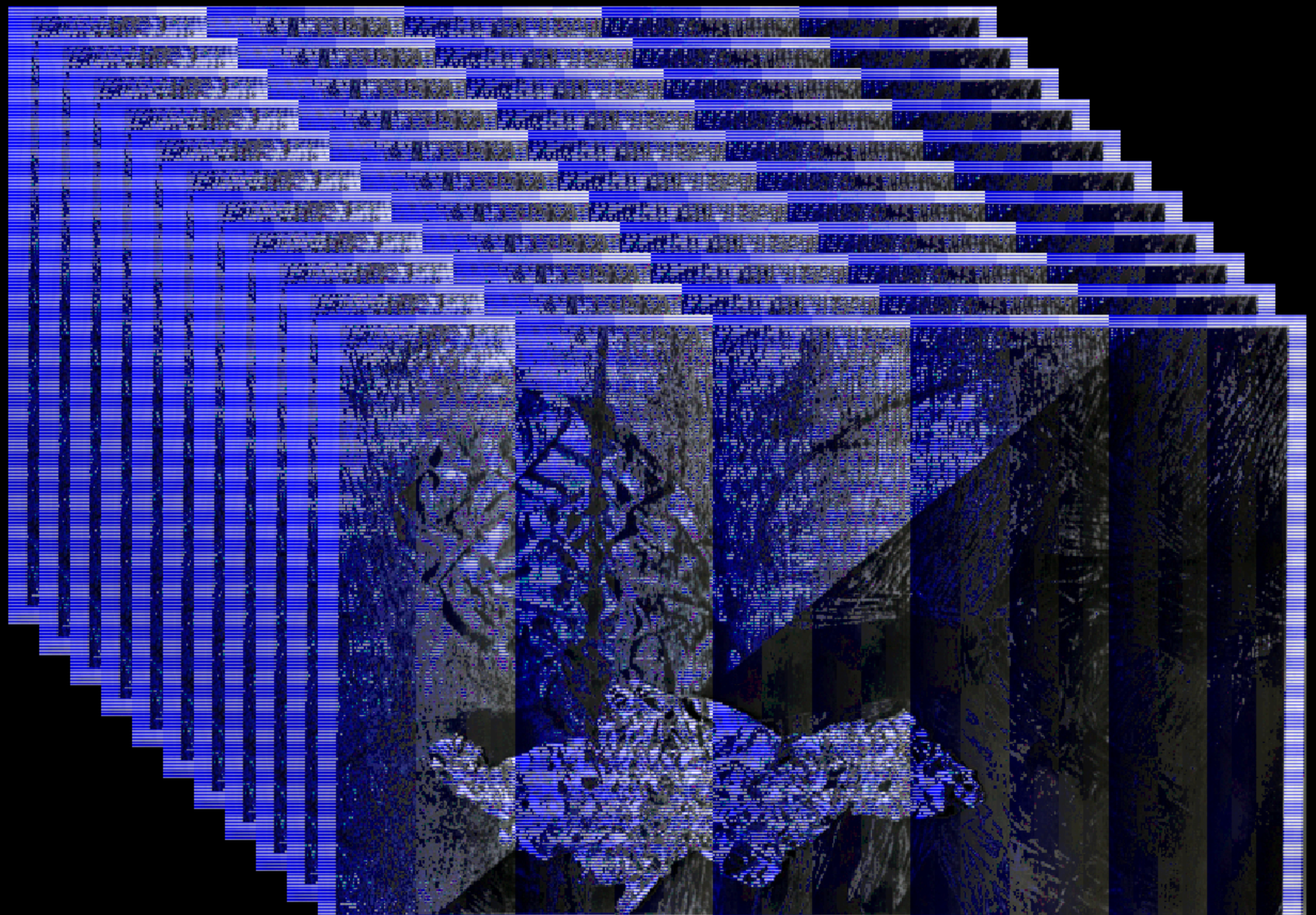



u know u luv me.
xoxo
ic3qu33n

GOP Complex - REcon 2024



GOP Complex - REcon 2024



Format of this Talk

This is a talk about translation, the art of binary golf and UEFI malware art

1. Assembly language programming across architectures

2. The art of binary golfing

3. Applications of “The Task of the Translator” to UEFI/Low-level firmware exploitation, xdev and malware art

Format of this Talk

This is a talk about translation, the art of binary golf and UEFI malware art

Part 1: Assembly language programming across architectures

- UEFI Quines (self-replicating UEFI apps) in three architectures:
- x86-64, arm64, EBC

Part 2: The art of binary golfing

Part 3: Applications of “The Task of the Translator” to UEFI/Low-level firmware exploitation

- Case study: the evolution of an SMM exploit
- From simple Chipsec PoC to standalone malicious driver [brief overview]

Definitions

Notes on terminology [This is a talk about translation after all]

- **Binary golf**: the art of writing the smallest program that performs a specific task
  - e.g. A self-replicating program (quine), famously featured in the paper “Reflections on Trusting Trust” by Ken Thompson
- **Quine**: a self-replicating program
  - Formally “a program whose output is a copy of its own source code”
- **Binary Golf Grand Prix**: an annual competition focused on the art of binary golfing, challenge specs/target/theme changes annually
- **PoC**: Proof of Concept
- Exploit: a program/PoC that successfully leverages a vulnerability in a piece of software/hardware/system/etc. to cause a desired outcome
- Exploit development (aka “xdev”): the process of writing a working PoC for a vulnerability

“Binary Golf Grand Prix,” https://binary.golf/
netspooky

Housekeeping

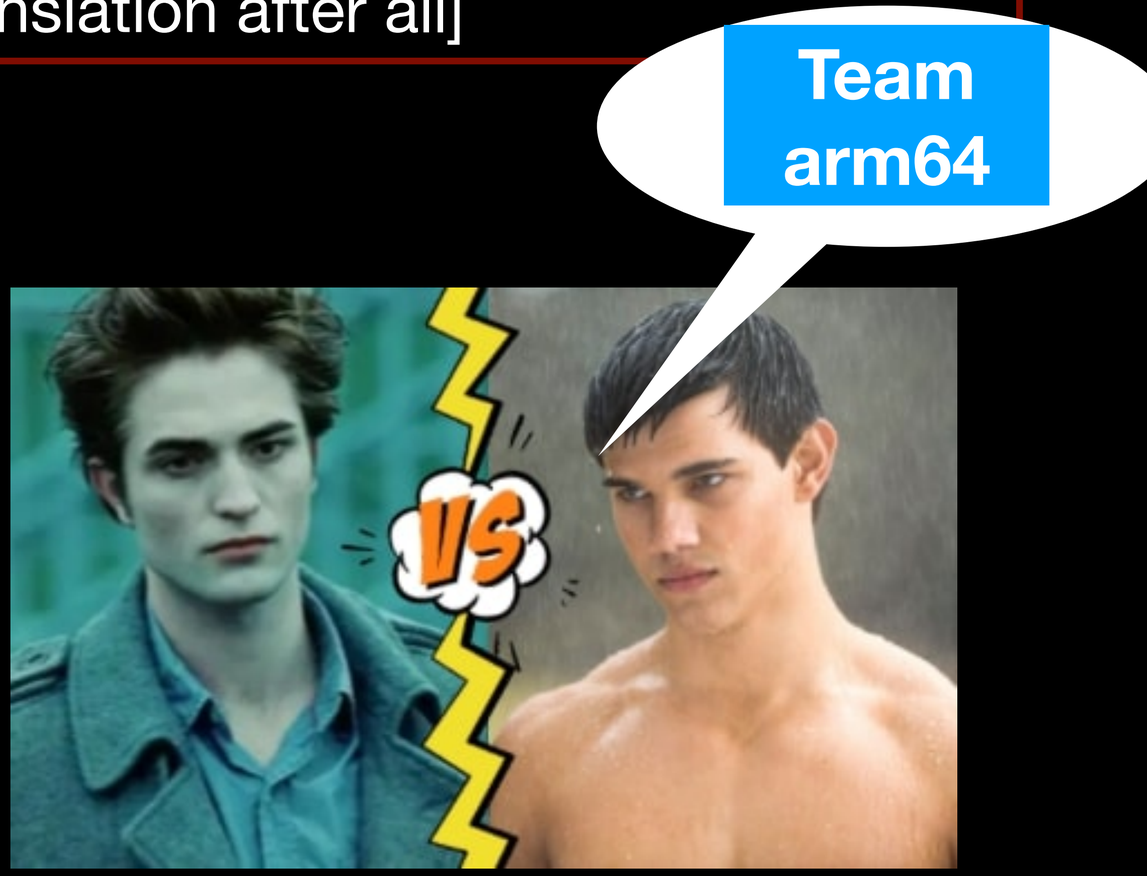
Notes on terminology [This is a talk about translation after all]

> “EBC isn’t an “architecture,” it’s a platform agnostic intermediary language that leverages natural-indexing to automatically adjust its instruction width to either 32-bit or 64-bit dependent on the architecture of the host machine. It uses a VM! That sounds like ring -1 to me!”

I know. But referring to it as an architecture at this point in the talk is sufficient for our understanding of EBC in relation to the narrative. And it’s more succinct. We’ll get to EBC and the spec. Hang tight.

> Wait… is the architecture arm64? Or aarch64? Or is it AArch64? Aarch64? ARM64? Arm64? Which is it?
> Team Edward or Team Jacob??

arm64 is the term I will use in this presentation to refer to the assembly language of the Armv8 64-bit architecture, known as ARM64/AArch64



What is “The Task of the Translator”

An essay by Walter Benjamin

- Walter Benjamin was a philosopher, cultural critic, essayist
- Other famous works by Benjamin:
  “The Work of Art in the Age of Mechanical Reproduction”
  The Arcades Project
- His essay “The Task of the Translator” was a seminal work in translation theory
- For this presentation, I’ll be referring to Steven Rendall’s English translation of Benjamin’s essay: https://german.yale.edu/sites/default/files/benjamin_translators_task.pdf

Walter Benjamin (1892–1940) ~1930 © Charlotte Joel

What is “The Task of the Translator” in UEFI?

A framing device for understanding how to write cross-architecture exploits

Combine the work of four separate projects using the framework of Walter Benjamin’s “Task of the Translator”

1. UEFI Exploit Research and Development at Leviathan —> SMM exploits

leviathan security group

2. BGGP4 and UEFI binary golfing —> UEFI quines

3. OST2 ARM Assembly class —> UEFI exploit dev on arm64

OST2 .FYI

4. VX-Underground Black Mass article —> EBC

How do we apply “The Task of the Translator” to UEFI?

Apply “the Task of the Translator” to two tasks:

1. Translating my winning BGGP4 UEFI quine from x86-64 asm to two other architectures: arm64 and EBC

2. Developing one exploit for an SMM callout vulnerability, then creating new generations of that exploit, altering the technique used, the language the exploit is written, the architecture it targets, etc.

One goal of optimization is to eliminate redundancy. Are we creating redundancy?

No, this isn’t redundant work.

I’m not creating *copies* of the original UEFI app (the UEFI app already creates copies of itself)

“Translation is a mode. In order to grasp it as such, we have to go back to the original.”

Walter Benjamin, “The Task of the Translator,” translated by Steven Rendall, page 152.

How do we apply “The Task of the Translator” to UEFI?

Notable examples to set the precedent

Developing a “next generation” for a piece of art:

- cr4sh - SmmBackdoorNg (Smm Backdoor Next Generation): https://github.com/Cr4sh/SmmBackdoorNg
  - See cr4sh’s earlier project SmmBackdoor: https://github.com/Cr4sh/SmmBackdoor
- Star Trek [with the notable exception of Leonard Nimoy, Leonard Nimoy is eternal]

What is “The Task of the Translator” in UEFI xdev?

Research questions

- How can we use binary golfing to further develop a work of art, an exploit?
- How does one UEFI exploit differ when it is translated across multiple different architectures?
- How does my artistic practice and creative projects with UEFI graphics programming inform and enhance my work in UEFI RE and exploit development?
- What can architecture-specific requirements for an exploit teach us about how to approach finding vulnerabilities and writing new gnarly exploits?
- How many different ways can we write an exploit for a specific vulnerability?

- What ***is*** the task of the translator?

The art of binary golf

Binary golfing as an art form

Mini-golf v2.0

I love writing programs in assembly languages

Binary golfing pushes this further and asks the question: how small can you make your code?

What is the most elegant solution to a problem?

Binary golfing is a demanding art form, but when an artist/hacker is determined to create something, they will find creative ways to bypass restrictions, and work with (rather than against) extreme constraints

The result of this process often leads to innovation and great art

Binary golfing as an art form

Techniques

- File format tricks - manipulation of headers, use of “dead code” regions as location for holding
- Assembly language programming tricks — variations in opcodes to fit constraints
  - e.g. polymorphic printable ASCII shellcode
- Compilation tricks: self-compilation, linking against stripped executables etc.
- Variations in playing with syscalls for Linux binaries
- Application of sound, graphics programming techniques in new+strange+fun ways
- Many more!

Binary golfing as an art form

Resources

Netspooky's series on ELF binary mangling

Part 1: https://n0.lol/ebm/1/

Part 2: https://n0.lol/ebm/2/

Part 3: https://n0.lol/ebm/3/

Part 4: https://tmpout.sh/2/11.html

LibGolf by xcellerator: https://tmpout.sh/1/1.html

Netspooky's series on PE binary mangling:

Golf Club, netspooky: https://github.com/netspooky/golfclub

Size coding: http://www.sizecoding.org/wiki/Linux

Binary golfing as an art form

ELF Binary Mangling Pt. 1: Concepts

[: 2018-08-07 :]

Okay, so you want to see how small you can make a 64 bit binary. In the age of giant bloated applications full of impossibly convoluted machine instructions, eating up your memory and disk space, it's nice sometimes to get down to the lowest of low levels and create something so tiny, that you know what every single bit is doing and it's purpose. To do so, we need to employ some standard tricks and a little creativity to get us down there.

Building Your Binary

Let's start with a really simple program that prints a string in the terminal! I chose these smaller opcodes to save a bit more space, but we can get into assembly optimization in another post.

```
;                                                       smile.asm
;-------------------------------------------------------------------
.global _start
.text
_start:
    mov   $1, %al       # RAX holds syscall 1 (write), I chose to use
                        # %al, which is the lower 8 bits of the %rax
                        # register. From a binary standpoint, there
                        # is less space used to represent this than
                        # mov $1, %rax
    mov   %rax, %rdi    # RDI holds File Handle 1, STDOUT. This means
                        # that we are writing to the screen. Again,
                        # moving RAX to RDI is shorter than
                        # using mov $1, %rdi
    mov   $msg, %rsi    # RSI holds the address of our string buffer.
    mov   $11, %dl      # RDX holds the size our of string buffer.
                        # Moving into %dl to save space.
    syscall             # Invoke a syscall with these arguments.
    mov   $60, %al      # Now we are invoking syscall 60.
    xor   %rdi, %rdi    # Zero out RDI, which holds the return value.
    syscall             # Call the system again to exit.
msg:
    .ascii "[^0^] w!!\n"
```

Check out these amazing resources by my friends!

netspooky's series on ELF binary mangling

LibGolf by xcellerator

~ xcellerator

Ahoy, fellow ELF devotees! In this article, I want to introduce a small library I've been working on called LibGolf. It started out as simply a vehicle for better understanding the ELF and program headers, but has since spun into something reasonably practical. It makes is very easy to generate a binary consisting of an ELF header, followed by a single program header, followed by a single loadable segment. By default, all the fields in the headers are set to sane values, but there's a simple way to play with these defaults - and that's what this article is all about! I'm going to demonstrate how I used LibGolf to enumerate precisely which bytes are necessary and which are ignored by the Linux loader. Fortunately, it turns out that the loader is one of the least picky parsers among the standard Linux toolkit. Before we're through, we'll see several popular static analysis tools crumble before our corrupted ELF, while the loader continues to merrily load and jump to our chosen bytes.

```
+---------------------------+
|--[ Introducing LibGolf ]--|
+---------------------------+
```

ELF Binary Mangling Series

[: 2021-07-07 :]

This is a blog series about making super small ELF binaries.

The record set in EBM4 was beaten in 2023 by lm978, who produced an 80 byte x86_64 ELF by using ET_DYN instead of ET_EXEC.

Elf Binary Mangling Pt. 4: Limit Break // 2021-07-07
An 82 byte ET_EXEC ELF for x86_64. Later published in tmp.0ut 2

ELF Binary Mangling Pt. 3: Weaponization // 2018-12-16
Making tiny ELFs destructive

ELF Binary Mangling Pt. 2: Golfin // 2018-12-09
Creating an 84 byte ELF for x86_64 with nasm

ELF Binary Mangling Pt. 1: Concepts // 2018-08-07
What's in an ELF anyways?

UEFI Binary Golfing

tips + tricks

- The offsets of data structures in UEFI are consistent, so if we know which data structure + protocol we want to target, we can write a test program to find those offsets, then define them with macros in our final exploit
  - e.g. BootServices-> HandleProtocol is at offset 0x98 in the EFI_BOOT_SERVICES table
- We will have easy access to data structures right away:
  - e.g. on x64, EFI_SYSTEM_TABLE * is in RDX and EFI_IMAGE_HANDLE is in RCX upon program invocation
- We can also target other data structures/protocols (i.e. EFI_FILE_PROTOCOL for file operations, EFI_SIMPLE_FILESYSTEM_PROTOCOL for filesystem operations, etc.) to hook/inject our payload
- In-depth knowledge of relevant file formats for UEFI binary executables (mainly PE, occasionally TE) can be used to shrink code size and minimize reliance on external libraries
  - In-depth knowledge of PE format essential for determining how to patch UEFI binaries into valid EBC UEFI apps/drivers

A brief introduction to UEFI

Introduction to UEFI

In the beginning there was legacy BIOS

BIOS: Basic Input-Output System

- BIOS is platform firmware responsible for configuring hardware and preparing a system before loading an operating system
- tl;dr BIOS is the software responsible for properly setting up your computer when it turns on

Legacy BIOS: The non-stardardized standard for BIOS implementations prior to the mainstream adoption of 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!

```
F000:FFF0   jmp   far ptr bootblock_start
.........
F000:FFAA bootblock_start:
F000:FFAA   jmp   exec_jmp_table
.........
F000:A040 exec_jmp_table:                ;
F000:A040   jmp   _CPU_early_init
F000:A043 ; ---------------------------------------------------------------
F000:A043
F000:A043 _j2:                           ;
F000:A043   jmp   _goto_j3
```

Source: "BIOS Disassembly Ninjutsu Uncovered: Listing 5.27 AMI BIOS Boot Block Jump Table," 1st edition, Darmawan Salihun (pinczakko), page 60, https://github.com/pinczakko/BIOS-Disassembly-Ninjutsu-Uncovered

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.

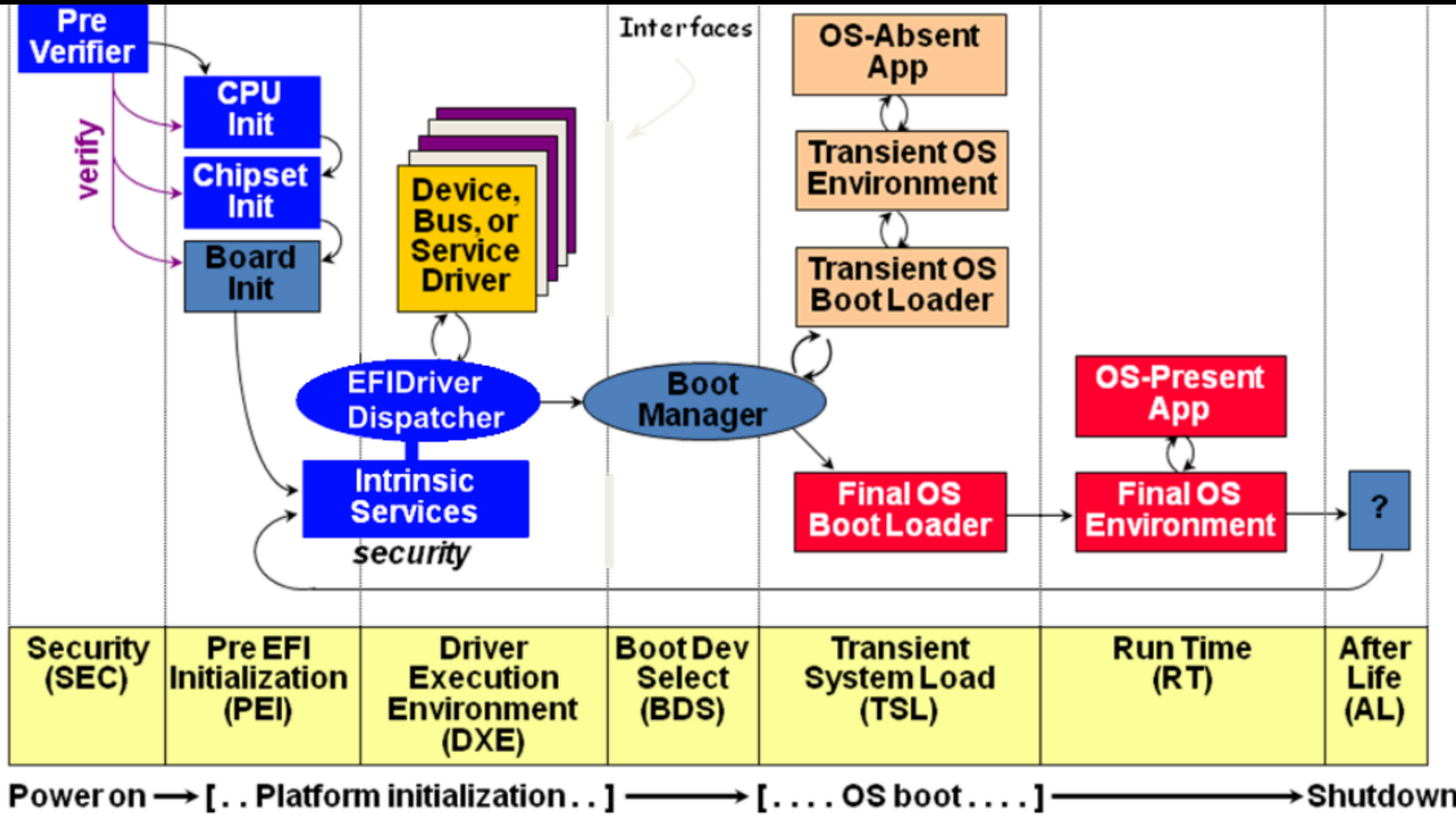


Figure 9 UEFI PI boot flow

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

Introduction to UEFI

Legacy BIOS Reverse Engineering

- BIOS code was written in 16-bit assembly and it ran in real mode
- Legacy BIOSes were non-standardized, IBV specific implementations
- Legacy BIOS was responsible for important functionality— initialization of platform hardware in preparation for loading an OS — but it was limited in scope and size
- Refer to “BIOS Disassembly Ninjutsu Uncovered” by Darmawan Salihun (pinczakko) for the holy scripture of Legacy BIOS RE + xdev

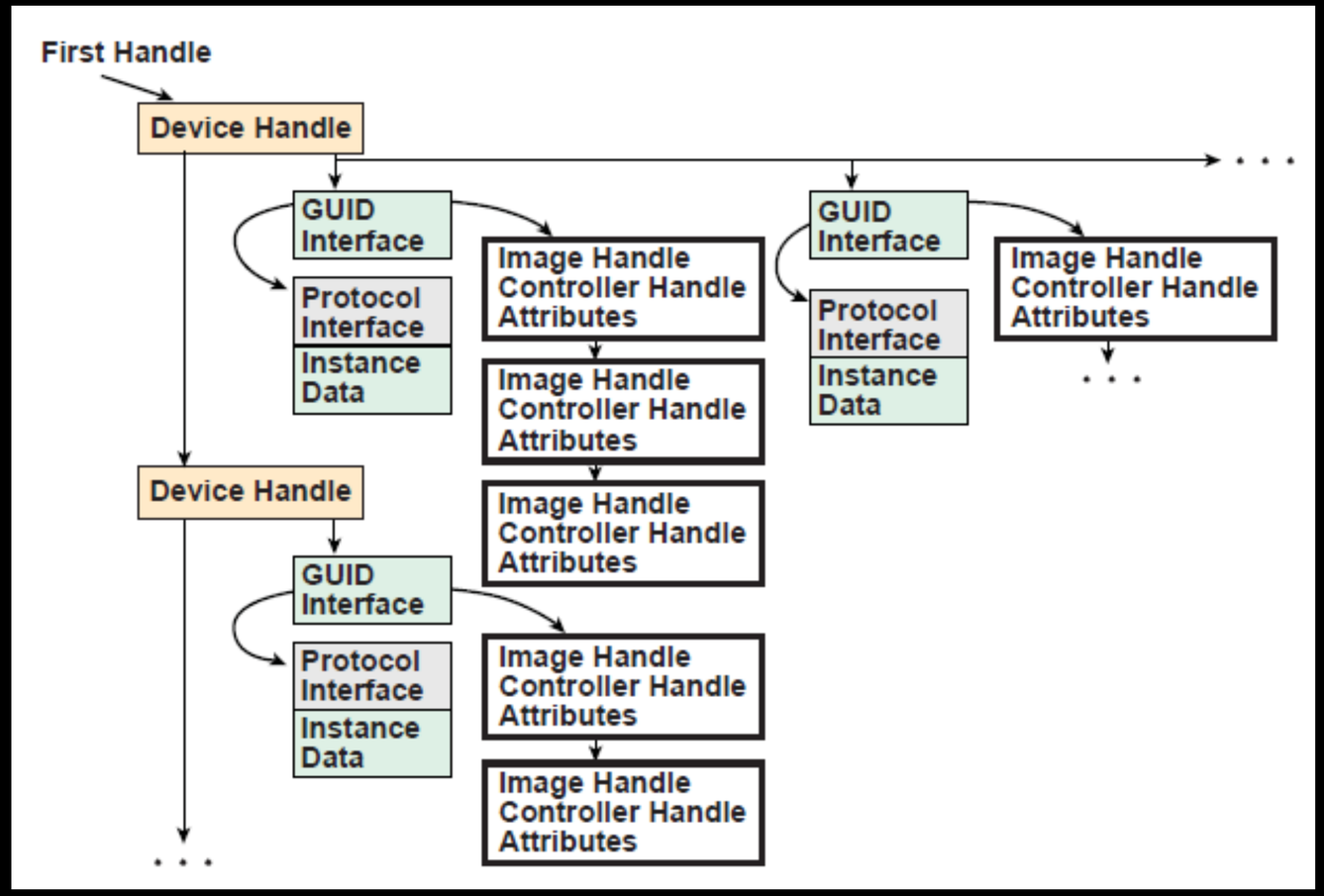
```
F000:A08E   call  near ptr copy_decomp_block
F000:A091   call  sub_F000_A440
.........
F000:A21B copy_decomp_block proc far      ; _F0000:_j27
F000:A21B   mov   al, 0D5h ; '-'          ; Boot block code is copied.
from
F000:A21B                         ; ROM to lower system memory and control
F000:A21B                         ; is given to it. BIOS now executes out of
F000:A21B                         ; RAM. Copies compressed boot block code
F000:A21B                         ; to memory in right segments. Copies BIOS
F000:A21B                         ; from ROM to RAM for faster access.
F000:A21B                         ; Performs main BIOS checksum, and updates
F000:A21B                         ; recovery status accordingly.
F000:A21D   out   80h, al         ; Send POST code D5h to diagnostic port.
F000:A21F   push  es
F000:A220   call  get_decomp_block_size   ; On return:
F000:A220                                 ; ecx = decomp_block_size
F000:A220                                 ; esi = decomp_block_phy_addr
F000:A220                                 ; At this point, ecx = 0x6000
F000:A220                                 ; and esi = 0xFFFFA000
F000:A223   mov   ebx, esi
F000:A226   push  ebx
F000:A228   shr   ecx, 2                  ; decomp_block_size / 4
F000:A22C   push  8000h
```

Source: “BIOS Disassembly Ninjutsu Uncovered: 5.2.3.2. Decompression Block Relocation,” 1st edition, Darmawan Salihun (pinczakko), page 62, https://github.com/pinczakko/BIOS-Disassembly-Ninjutsu-Uncovered/blob/master/BIOS_Disassembly_Ninjutsu_Uncovered.pdf

Introduction to UEFI

RE advantages of UEFI over Legacy BIOS

- Rich ecosystem of built-in functionality
- UEFI follows implementation standards with detailed and comprehensive spec [obvious caveats, it's not perfect but wow look at those diagrams. AMI never gave me a diagram </3]
- source code primarily written in C following a standardized specification —> easier to debug / disassemble
- A selection of great plugins and tools for UEFI RE + xdev:
  - UEFITool: https://github.com/LongSoft/UEFITool
  - efiXplorer: https://github.com/binarly-io/efiXplorer
  - Ghidra plugins:
    - efiSeek: https://github.com/DSecurity/efiSeek
    - ghidra-firmware-utils: https://github.com/al3xtjames/ghidra-firmware-utils
- UEFI has expansive breadth + depth —> greater attack surface



Source: "UEFI Specification, Fig.7.2 Handle Database"
https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#device-handle-to-protocol-handle mapping

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

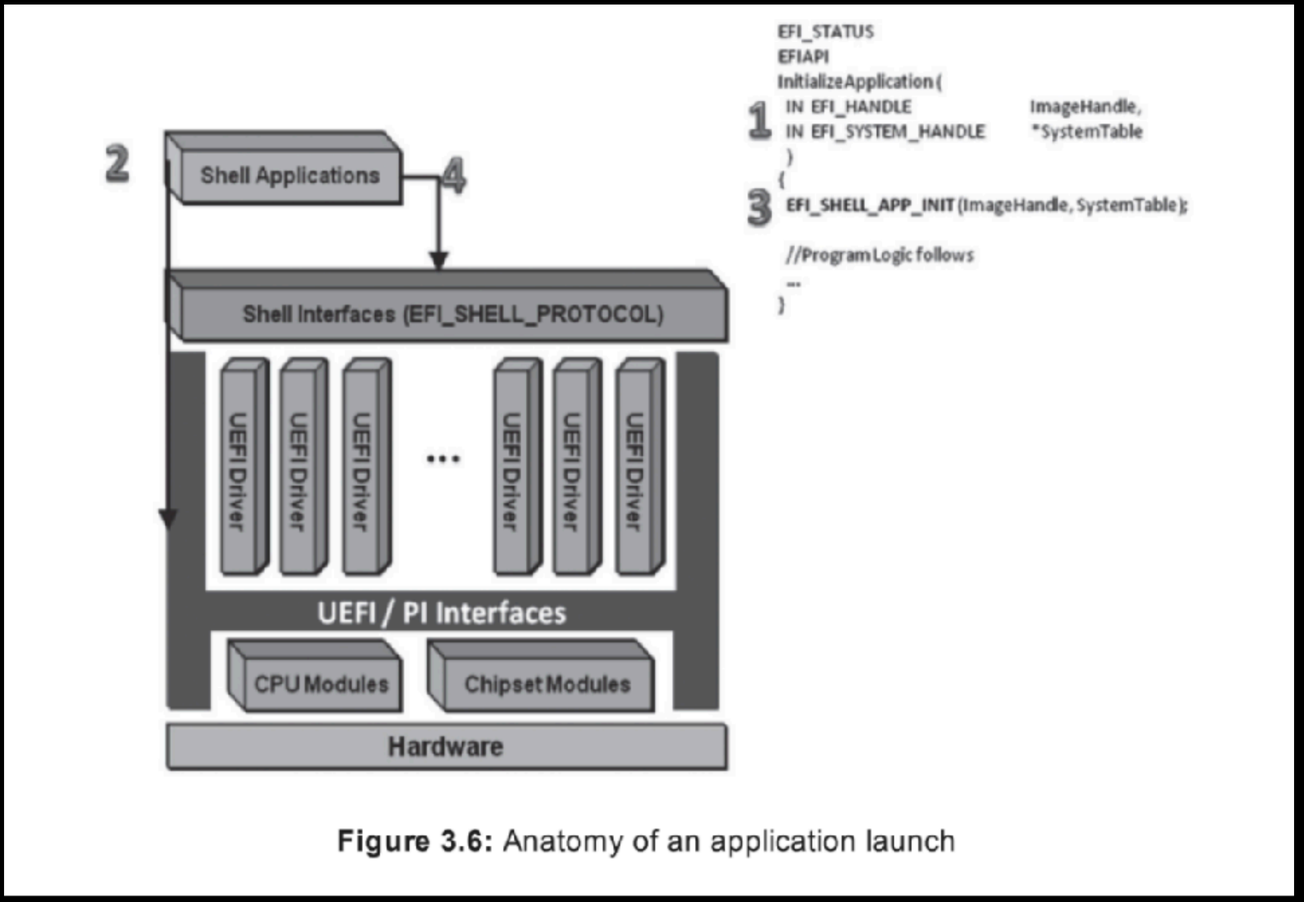


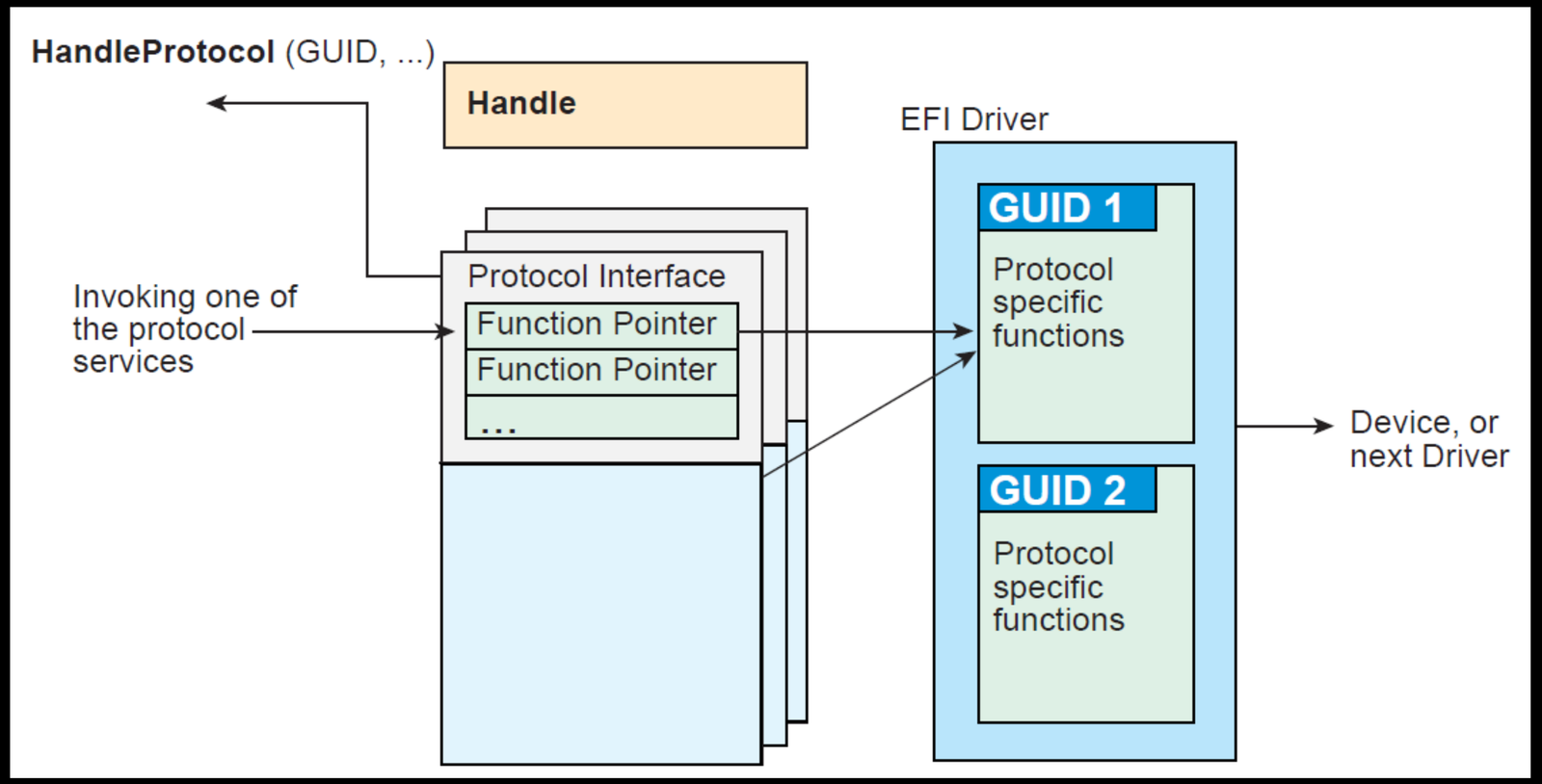
Figure 3.6: Anatomy of an application launch

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

“True translation is transparent, it does not obscure the original, does not stand in its light, but rather allows pure language, as if strengthened by its own medium, to shine even more fully on the original.”

Walter Benjamin, “The Task of the Translator,” translated by Steven Rendall, page 162.

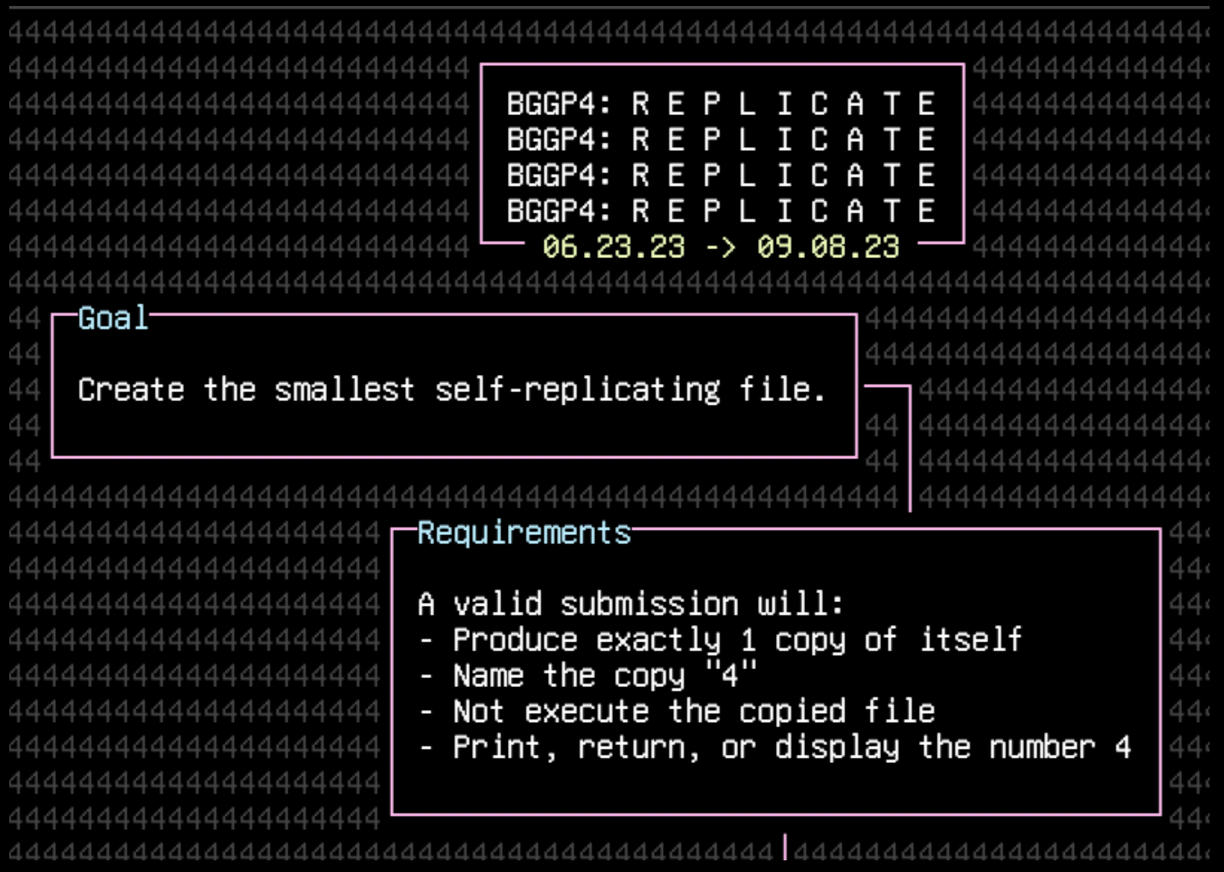
UEFI generation 1: x86-64

UEFI generation 1: x86-64

The Specs

My winning entry in the UEFI app category of Binary Golf Grand Prix 4

- BGGP: “The goal of the Binary Golf Grand Prix is to challenge programmers to make the smallest possible binary that fits within certain constraints.” [Source “Binary Golf Grand Prix”, netspooky, https://n0.lol/bggp/]



Source: “Binary Golf Grand Prix 4,” Binary Golf Association, https://binary.golf/

UEFI generation 1: x86-64

Methodology

1. Write a valid working solution (a self-replicating UEFI app) in C
2. Use the C solution as a base text and translate the quine from C to assembly —> Reverse engineer the C solution
3. Golf the assembly solution and shrink the size of the binary as much as possible
4. Reverse engineer, rewrite and refactor the assembly

Size of C quine: ~17,000 bytes

Final size of x86_64 asm UEFI quine: 1480 bytes

[Side note: shoutout to my friend @netspooky who I worked with on this project for teaching me PE binary mangling. Check out his fantastic write-up on his recent solution that set the new record to 420 bytes: https://github.com/netspooky/golfclub/tree/master/uefi/bggp4]

UEFI generation 1: x86-64

RE and development tools

- nasm
- Hex editor (xxd, hexdump)
- Ghidra, specifically using these two plugins for UEFI:
  - efiSeek: https://github.com/DSecurity/efiSeek
  - ghidra-firmware-utils: https://github.com/al3xtjames/ghidra-firmware-utils
- Radare2 for a faster option, better for disassembling and other reversing tasks near the end of the project that involved nitty gritty changes to the assembly
- QEMU and gdb for debugging/testing
- I didn’t use IDA Pro for this project, it’s a better tool for other projects

UEFI generation 1: x86-64

UEFI x64 - Handoff state upon program invocation

rcx - EFI_HANDLE

rdx - EFI_SYSTEM_TABLE*

rsp - <return address>

Source: UEFI Specification - 2.3.4.1. Handoff State

```
_start:
entrypoint:
    push rbp
    mov rbp, rsp
    sub rsp,0xc0

    mov [ImageHandle], rcx
    mov [gST], rdx

    mov rbx, [gST]
    mov rbx, [rbx + 0x60]
    mov [gBS], rbx
    mov rax, [gST]
    mov rax, [rax + 0x40]
    mov [ConOut], rax
```

Program entry point - setting up stack frame, saving gST, ImageHandle
Use gST to save gBS and ConOut

```
UEFI Interactive Shell v2.2
EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
      FS0: Alias(s):HD0a1:;BLK1:
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(1,MBR,0xBE1AFDFA,0x3F,0xFBFC1)
     BLK0: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
     BLK2: Alias(s):
          PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
Press ESC in 3 seconds to skip startup.nsh or any other key to continue.
Shell> fs0:
FS0:\>
```

Base text:
Self-replicating UEFI app
Written in C

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown

gBS->HandleProtocol()
Retrieve LoadedImageProtocol

```
mov rbx, [gBS]
mov rdi, [rbx + 0x98]       ;gBS->H
mov rcx, [ImageHandle]      ; param

;;this is how we're passing the GU
lea r8, [LoadedImageProtocol]
mov dword [rbp-0x40], 0x5b1b31a1
mov word [rbp-0x3c], 0x9562
mov word [rbp-0x3a], 0x11d2
mov rax, 0x3b7269c9a0003f8e
mov [rbp-0x38], rax
lea rdx, [rbp-0x40]
mov r9, [ImageHandle]
call rdi
cmp qword [rax], byte 0x0
jne printerror
```

gBS->HandleProtocol()
Retrieve SimpleFilesystemProtocol

```
get_sfsp:
    mov rbx, [gBS]          ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [DeviceHandle]
    ;;this is how we're passing the GUID so that it works
    mov dword [rbp-0x60],0x964e5b22,
    mov word [rbp-0x5c], 0x6459,
    mov word [rbp-0x5a], 0x11d2
    mov rax, 0x3b7269c9a000398e
    mov [rbp-0x58], rax
    lea rdx, [rbp-0x60]
    lea r8, [SimpleFilesystemProtocol]
    mov r9, [ImageHandle]
    xor r10, r10
    mov rax, [rbx + 0x98]       ;gBS->HandleProtocol()
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

SimpleFilesystemProtocol->OpenVolume()
Retrieve Root Volume

```
get_root_volume:
    mov rax, [SimpleFilesystemProtocol]
    mov rax, [rax + EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPENVOLUME_OFFSET]
    mov rcx, [SimpleFilesystemProtocol]
    lea rdx, [root_volume]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->OpenFile()
Open Host File

```
open_hostfile:
    mov rax, [root_volume]
    mov rax, [rax + EFI_FILE_PROTOCOL_OPEN_FILE_OFFSET
    mov rcx, [root_volume]
    lea rdx, [hostfile]
    lea r8, hostfilename
    mov r9, [fileopen_mode]
    mov r10, [hostattributes]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->OpenFile()
Open Target File

```
open_targetfile:
    mov rax, [root_volume]
    mov rax, [rax + EFI_FILE_PROTOCOL_OPEN_FILE_OFFSET]
    mov rcx, [root_volume]
    mov qword [rbp - 0x78], 0x0
    lea rdx, [rbp-0x78]
    lea r8, targetfilename
    mov r9, 0x8000000000000003
    mov qword [rsp+0x20], 0x0
    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    mov rax, [rbp-0x78]
    [targetfile], rax
```

gBS->AllocatePool()
Allocate buffer to hold file contents

```
allocate_tmp_buffer:
    mov rax, [gBS]
    mov rax, [rax + EFI_BOOTSERVICES_ALLOCATEPOOL_OFFSET]
    mov rcx, [EFI_ALLOCATEPOOL_ALLOCATEANYPAGES]
    mov rdx, [ImageSize]
    lea r8, [temp_buffer]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->ReadFile()
Read host file into buffer

```
read_hostfile:
    mov rax, [hostfile]
    mov rax, [rax + EFI_FILE_PROTOCOL_READ_FILE_OFFSET]
    mov rcx, [hostfile]
    lea rdx, [ImageSize]
    mov r8, [temp_buffer]

    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->WriteFile()
Write buffer to target file

```
write_targetfile:
    mov rax, [targetfile]
    mov rax, [rax + EFI_FILE_PROTOCOL_WRITE_FILE_OFFSET]
    mov rcx, [targetfile]
    lea rdx, [ImageSize]
    mov r8, [temp_buffer]

    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    jmp baibai
```

gBS->FreePool()
Free buffer

```
free_tmp_buffer:
    mov r13, [targetfile]
    call close_file
    mov rbx, [gBS]
    mov rax, [rbx + EFI_BOOTSERVICES_FREEPOOL_OFFSET]
    mov rcx, [temp_buffer]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    mov r13, [hostfile]
    call close_file
    xor r13, r13
    mov r13, [root_volume]
    call close_file
```

FileProtocol->CloseFile()
Close Target File, Host File + Root Volume

```
close_file:
    mov rax, r13
    mov rax, [rax + EFI_FILE_PROTOCOL_CLOSE_FILE_OFFSET]
    mov rcx, r13
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
    ret
```

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown

gBS->HandleProtocol()
Retrieve LoadedImageProtocol

```
mov rbx, [gBS]
mov rdi, [rbx + 0x98]        ;gBS->HandleProtocol()
mov rcx, [ImageHandle]       ; params passed in rcx, rdx, r8, r9, r10

;;this is how we're passing the GUID so that it works
lea r8, [LoadedImageProtocol]
mov dword [rbp-0x40], 0x5b1b31a1
mov word [rbp-0x3c], 0x9562
mov word [rbp-0x3a], 0x11d2
mov rax, 0x3b7269c9a0003f8e
mov [rbp-0x38], rax
lea rdx, [rbp-0x40]
mov r9, [ImageHandle]
call rdi
cmp qword [rax], byte 0x0
jne printerror
```

gBS->HandleProtocol()
Retrieve SimpleFileSystemProtocol

```
get_sfsp:
    mov rbx, [gBS]           ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [DeviceHandle]
    ;;this is how we're passing the GUID so that it works
    mov dword [rbp-0x60],0x964e5b22,
    mov word [rbp-0x5c], 0x6459,
    mov word [rbp-0x5a], 0x11d2
    mov rax, 0x3b7269c9a000398e
    mov [rbp-0x58], rax
    lea rdx, [rbp-0x60]
    lea r8, [SimpleFilesystemProtocol]
    mov r9, [ImageHandle]
    xor r10, r10
    mov rax, [rbx + 0x98]        ;gBS->HandleProtocol()
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown

SimpleFileSystemProtocol->OpenVolume()
Retrieve Root Volume

```
get_root_volume:
    mov rax, [SimpleFilesystemProtocol]
    mov rax, [rax + EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPENVOLUME_OFFSET]
    mov rcx, [SimpleFilesystemProtocol]
    lea rdx, [root_volume]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown

FileProtocol.OpenFile()
Open Host File

FileProtocol.OpenFile()
Open Target File

```
open_hostfile:
    mov rax, [root_volume]
    mov rax, [rax + EFI_FILE_PROTOCOL_OPEN_FILE_OFFSET]
    mov rcx, [root_volume]
    lea rdx, [hostfile]
    lea r8, hostfilename
    mov r9, [fileopen_mode]
    mov r10, [hostattributes]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

```
open_targetfile:
    mov rax, [root_volume]
    mov rax, [rax + EFI_FILE_PROTOCOL_OPEN_FILE_OFFSET]
    mov rcx, [root_volume]
    mov qword [rbp - 0x78], 0x0
    lea rdx, [rbp-0x78]
    lea r8, targetfilename
    mov r9, 0x8000000000000003
    mov qword [rsp+0x20], 0x0
    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    mov rax, [rbp-0x78]
    mov [targetfile], rax
```

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown

gBS->AllocatePool()
Allocate buffer to hold file contents

```
allocate_tmp_buffer:
    mov rax, [gBS]
    mov rax, [rax + EFI_BOOTSERVICES_ALLOCATEPOOL_OFFSET]
    mov rcx, [EFI_ALLOCATEPOOL_ALLOCATEANYPAGES]
    mov rdx, [ImageSize]
    lea r8, [temp_buffer]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol.ReadFile()
Read host file into buffer

```
read_hostfile:
    mov rax, [hostfile]
    mov rax, [rax + EFI_FILE_PROTOCOL_READ_FILE_OFFSET]
    mov rcx, [hostfile]
    lea rdx, [ImageSize]
    mov r8, [temp_buffer]

    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

UEFI generation 1: x86-64

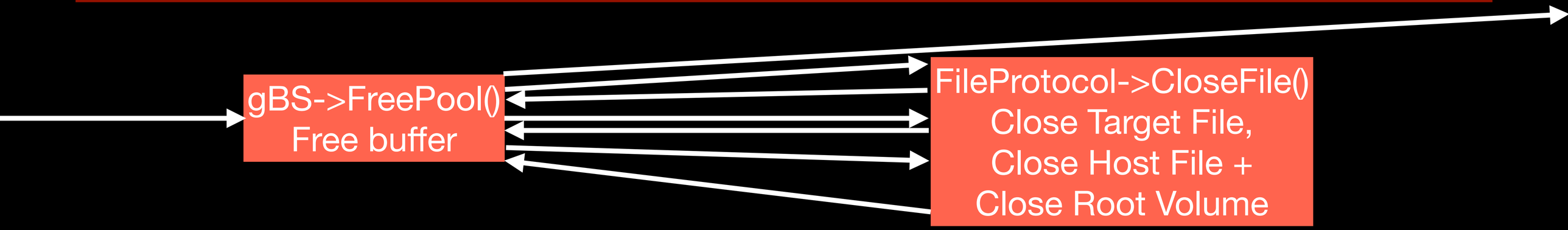
x64 self-replicating UEFI app - program logic breakdown

FileProtocol->WriteFile()
Write buffer to target file

```
get_root_volume:
    mov rax, [SimpleFilesystemProtocol]
    mov rax, [rax + EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPENVOLUME_OFFSET]
    mov rcx, [SimpleFilesystemProtocol]
    lea rdx, [root_volume]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown




```
free_tmp_buffer:
    mov r13, [targetfile]
    call close_file
    mov rbx, [gBS]
    mov rax, [rbx + EFI_BOOTSERVICES_FREEPOOL_OFFSET]
    mov rcx, [temp_buffer]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    mov r13, [hostfile]
    call close_file
    xor r13, r13
    mov r13, [root_volume]
    call close_file
```

```
close_file:
    mov rax, r13
    mov rax, [rax + EFI_FILE_PROTOCOL_CLOSE_FILE_OFFSET]
    mov rcx, r13
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
    ret
```

UEFI generation 1: x86-64

x64 self-replicating UEFI app - program logic breakdown

gBS->HandleProtocol()
Retrieve LoadedImageProtocol

```
mov rbx, [gBS]
mov rdi, [rbx + 0x98]        ;gBS->H
mov rcx, [ImageHandle]       ; param

;;this is how we're passing the GU
lea r8, [LoadedImageProtocol]
mov dword [rbp-0x40], 0x5b1b31a1
mov word [rbp-0x3c], 0x9562
mov word [rbp-0x3a], 0x11d2
mov rax, 0x3b7269c9a0003f8e
mov [rbp-0x38], rax
lea rdx, [rbp-0x40]
mov r9, [ImageHandle]
call rdi
cmp qword [rax], byte 0x0
jne printerror
```

gBS->HandleProtocol()
Retrieve SimpleFilesystemProtocol

```
get_sfsp:
    mov rbx, [gBS]          ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [DeviceHandle]
    ;;this is how we're passing the GUID so that it works
    mov dword [rbp-0x60],0x964e5b22,
    mov word [rbp-0x5c], 0x6459,
    mov word [rbp-0x5a], 0x11d2
    mov rax, 0x3b7269c9a000398e
    mov [rbp-0x58], rax
    lea rdx, [rbp-0x60]
    lea r8, [SimpleFilesystemProtocol]
    mov r9, [ImageHandle]
    xor r10, r10
    mov rax, [rbx + 0x98]       ;gBS->HandleProtocol()
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

SimpleFilesystemProtocol->OpenVolume()
Retrieve Root Volume

```
get_root_volume:
    mov rax, [SimpleFilesystemProtocol]
    mov rax, [rax + EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPENVOLUME_OFFSET]
    mov rcx, [SimpleFilesystemProtocol]
    lea rdx, [root_volume]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->OpenFile()
Open Host File

```
open_hostfile:
    mov rax, [root_volume]
    mov rax, [rax + EFI_FILE_PROTOCOL_OPEN_FILE_OFFSET
    mov rcx, [root_volume]
    lea rdx, [hostfile]
    lea r8, hostfilename
    mov r9, [fileopen_mode]
    mov r10, [hostattributes]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->OpenFile()
Open Target File

```
open_targetfile:
    mov rax, [root_volume]
    mov rax, [rax + EFI_FILE_PROTOCOL_OPEN_FILE_OFFSET]
    mov rcx, [root_volume]
    mov qword [rbp - 0x78], 0x0
    lea rdx, [rbp-0x78]
    lea r8, targetfilename
    mov r9, 0x8000000000000003
    mov qword [rsp+0x20], 0x0
    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    mov rax, [rbp-0x78]
    [targetfile], rax
```

gBS->AllocatePool()
Allocate buffer to hold file contents

```
allocate_tmp_buffer:
    mov rax, [gBS]
    mov rax, [rax + EFI_BOOTSERVICES_ALLOCATEPOOL_OFFSET]
    mov rcx, [EFI_ALLOCATEPOOL_ALLOCATEANYPAGES]
    mov rdx, [ImageSize]
    lea r8, [temp_buffer]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->ReadFile()
Read host file into buffer

```
read_hostfile:
    mov rax, [hostfile]
    mov rax, [rax + EFI_FILE_PROTOCOL_READ_FILE_OFFSET]
    mov rcx, [hostfile]
    lea rdx, [ImageSize]
    mov r8, [temp_buffer]

    call rax
    cmp qword [rax], byte 0x0
    jne printerror
```

FileProtocol->WriteFile()
Write buffer to target file

```
write_targetfile:
    mov rax, [targetfile]
    mov rax, [rax + EFI_FILE_PROTOCOL_WRITE_FILE_OFFSET]
    mov rcx, [targetfile]
    lea rdx, [ImageSize]
    mov r8, [temp_buffer]

    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    jmp baibai
```

gBS->FreePool()
Free buffer

```
free_tmp_buffer:
    mov r13, [targetfile]
    call close_file
    mov rbx, [gBS]
    mov rax, [rbx + EFI_BOOTSERVICES_FREEPOOL_OFFSET]
    mov rcx, [temp_buffer]
    call rax
    cmp qword [rax], byte 0x0
    jne printerror

    mov r13, [hostfile]
    call close_file
    xor r13, r13
    mov r13, [root_volume]
    call close_file
```

FileProtocol->CloseFile()
Close Target File, Host File + Root Volume

```
close_file:
    mov rax, r13
    mov rax, [rax + EFI_FILE_PROTOCOL_CLOSE_FILE_OFFSET]
    mov rcx, r13
    call rax
    cmp qword [rax], byte 0x0
    jne printerror
    ret
```

UEFI generation 1: x86-64

Golfing the solution

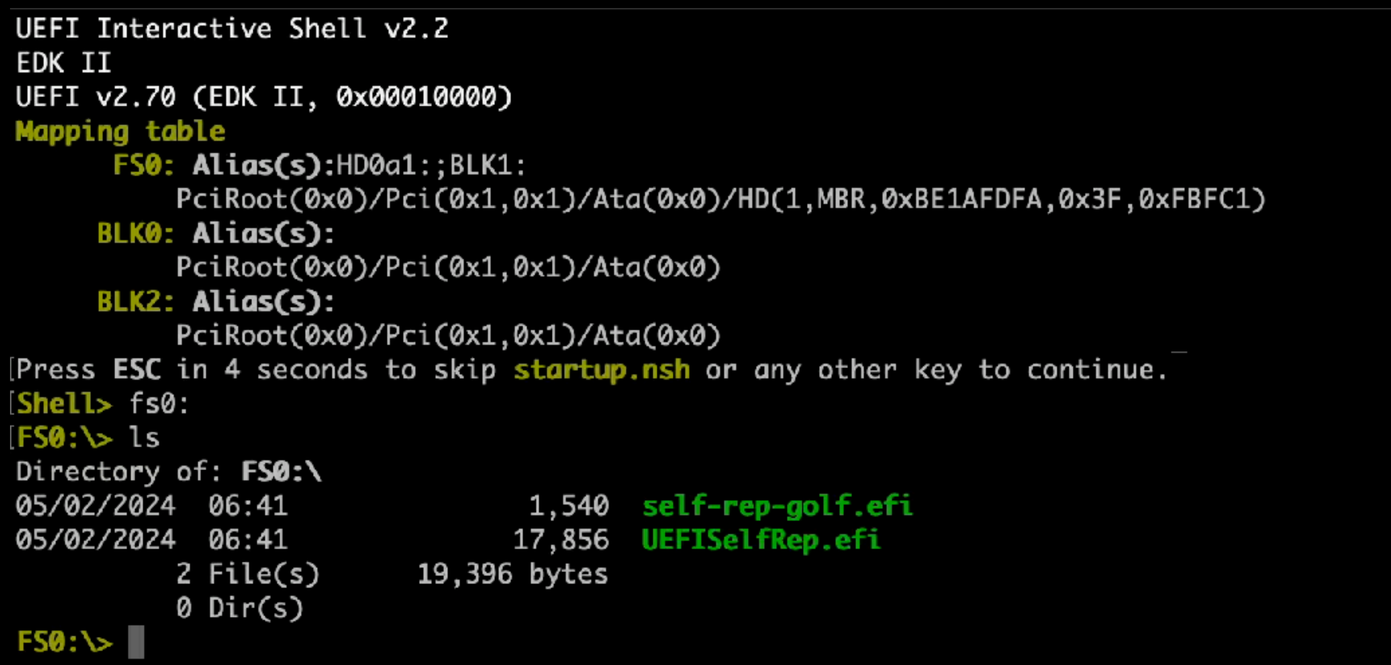
1. Remove unnecessary libraries and dependencies: Use the UEFI ecosystem

2. PE Binary Mangling [netspooky's guide to PE Binary Mangling: https://n0.lol/a/pemangle.html]

3. Use the protocols you want, not the wrappers with extra fluff:
   e.g. OpenProtocol() is a wrapper for HandleProtocol()

```
_start:
entrypoint:
    push rbp
    mov rbp, rsp
    sub rsp,0xc0

    mov [ImageHandle], rcx
    mov [gST], rdx

    mov rbx, [gST]
    mov rbx, [rbx + 0x60]
    mov [gBS], rbx
    mov rax, [gST]
    mov rax, [rax + 0x40]
    mov [ConOut], rax

    mov rbx, [gBS]
    mov rdi, [rbx + 0x98]       ;gBS->HandleProtocol()
                                ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [ImageHandle]

    ;;this is how we're passing the GUID so that it works
    lea r8, [LoadedImageProtocol]
    mov dword [rbp-0x40], 0x5b1b31a1
    mov word [rbp-0x3c], 0x9562
    mov word [rbp-0x3a], 0x11d2
    mov rax, 0x3b7269c9a0003f8e
    mov [rbp-0x38], rax
    lea rdx, [rbp-0x40]
    mov r9, [ImageHandle]
    call rdi
    cmp qword [rax], byte 0x0
    jne printerror
```

First call to gBS function HandleProtocol in my winning BGGP4 entry



Final winning entry for BGGP4: Self-replicating UEFI app

Written in x64 assembly

https://youtu.be/MglEnqr-1yY

UEFI generation 1: x86-64

What did you learn at school today?

- Leverage the UEFI ecosystem by walking from Protocol interface to Protocol interface —> better understanding of UEFI internals and base knowledge for building better exploits
  - Building ROP chains for SMM exploits to bypass Smm_CodeCheck_En
- New knowledge of PE Binary Mangling
- Knowledge of how to write UEFI shellcode
  - even if you write an exploit in C, knowing how to write UEFI shellcode for a payload is essential

```
PE:
header_start:
mzheader:
    dw "MZ"             ;  DOS e_magic
    dw 0x100

pe_header:
    dd "PE"             ;   uint32_t mMagic; // PE\0\0 or 0x00004550
    dw 0x8664           ;   uint16_t mMachine;
    dw 3                ;   uint16_t mNumberOfSections;
    dd 0x0              ;   uint32_t mTimeDateStamp;
    dd 0x0              ;   uint32_t mPointerToSymbolTable;
    dd 0x0              ;   uint32_t mNumberOfSymbols;
    dw sectionHeader - opt_header;  uint16_t mSizeOfOptionalHeader;
    dw 0x0206                   ;   uint16_t mCharacteristics;
opt_header:
    dw 0x20B                    ;   uint16_t mMagic
                                ;   [0x010b=PE32, 0x020b=PE32+ (64 bit)]
    db 0                        ;   uint8_t  mMajorLinkerVersion;
    db 0                        ;   uint8_t  mMinorLinkerVersion;
    dd _codeend - codestart     ;   uint32_t mSizeOfCode;
    dd _dataend - _datastart    ;   uint32_t mSizeOfInitializedData;
    dd 0                        ;   uint32_t mSizeOfUninitializedData;
    dd entrypoint - START       ;   uint32_t mAddressOfEntryPoint;
    dd entrypoint - START       ;   uint32_t mBaseOfCode;
    dq 0x0                      ;   uint32_t mImageBase;
    dd 0x4                      ;   uint32_t mSectionAlignment;
    dd 0x4                      ;   uint32_t mFileAlignment;
    dw 0                        ;   uint16_t mMajorOperatingSystemVersion
    dw 0                        ;   uint16_t mMinorOperatingSystemVersion
    dw 0                        ;   uint16_t mMajorImageVersion;
    dw 0                        ;   uint16_t mMinorImageVersion;
    dw 0                        ;   uint16_t mMajorSubsystemVersion;
    dw 0                        ;   uint16_t mMinorSubsystemVersion
                                ;   can be blank, still times 4 db 0
    dd 0                        ;   uint32_t mWin32VersionValue;
    dd end - START              ;   uint32_t mSizeOfImage;
    dd header_end - header_start;   uint32_t mSizeOfHeaders;
    dd 0                        ;   uint32_t mCheckSum;
    dw 0xa                      ;   uint16_t mSubsystem;
    dw 0x0                      ;   uint16_t mDllCharacteristics;
    dq 0x0                      ;   uint32_t mSizeOfStackReserve;
    dq 0x0                      ;   uint32_t mSizeOfStackCommit;
    dq 0x0                      ;   uint32_t mSizeOfHeapReserve;
    dq 0x0                      ;   uint32_t mSizeOfHeapCommit;
    dd 0x0                      ;   uint32_t mLoaderFlags;
    dd 0x6                      ;   uint32_t mNumberOfRvaAndSizes;
datadirs:
    dq 0
    dq 0
    dq 0
    dq 0
    dq 0
    dq 0
```

“The translator's task consists in this: to find the intention toward the language into which the work is to be translated, on the basis of which an echo of the original can be awakened in it.”

Walter Benjamin, “The Task of the Translator,” translated by Steven Rendall, page 159.

UEFI generation 2: arm64

UEFI generation 2: arm64

The specs

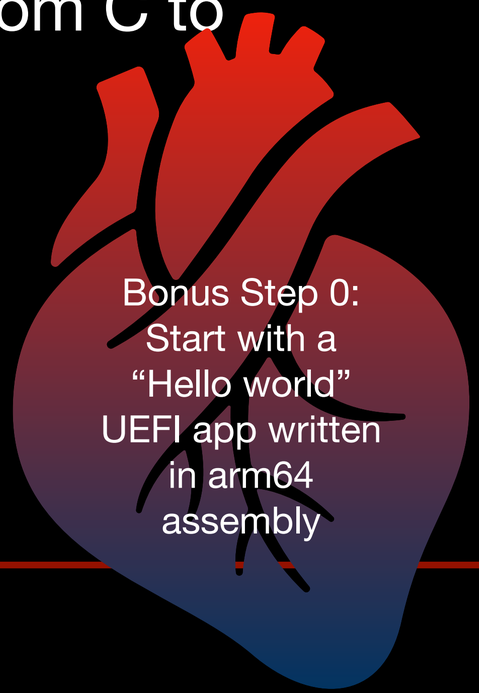
- This is not an entry for BGGP4… what are the goals of this UEFI quine?
  - Confirm that a UEFI quine is *possible* on Aarch64/ARM64 architecture
  - Translate original x64 solution to valid working solution in arm64 assembly
  - Golfing -> Optimize for small size to maximize benefit of shellcode
- What are the goals for this UEFI arm64 project?
  - Advance mastery of arm64 assembly for teaching OST2 ARM Assembly class
  - Practice writing UEFI shellcode in arm64 assembly
  - Better understand the nuances of UEFI RE and exploit dev on arm64

UEFI generation 2: arm64

Methodology

1. Recompile my valid working solution (a self-replicating UEFI app) in C with an arm64 (edk2 calls it aarch64) toolchain under the edk2 build system -> working solution to use as a base template
2. Use the C solution as a base text and translate the quine from C to assembly —> Reverse engineer the C solution
3. Reverse engineer, rewrite and refactor the assembly

The task of the translator is to be a cross-compiler?



UEFI generation 2: arm64

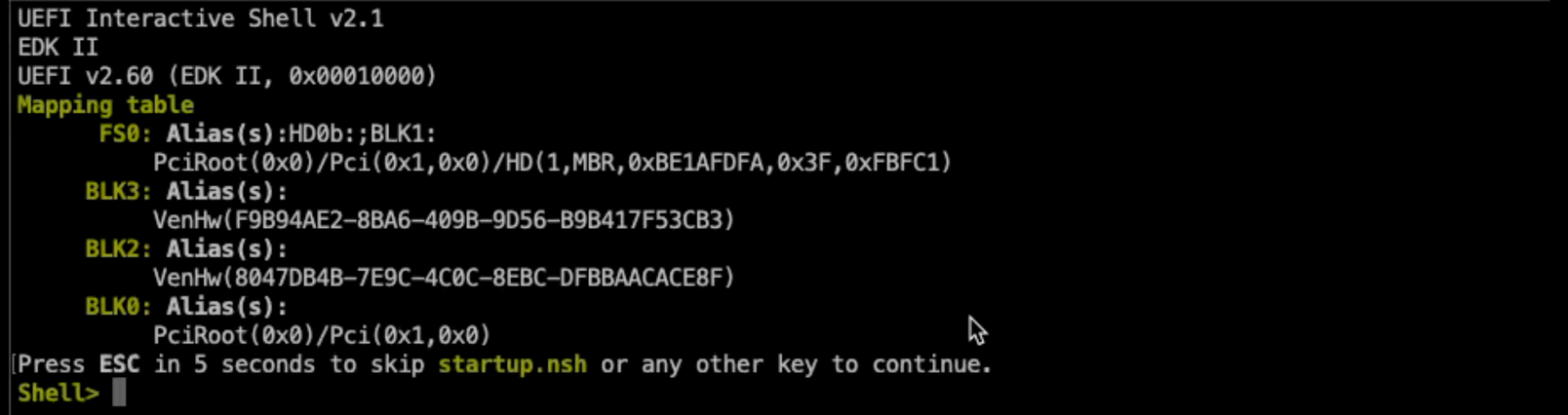
arm64 assembly building blocks: handoff state

X0 - EFI_HANDLE

X1 - EFI_SYSTEM_TABLE

X30 - Return Address

Source: UEFI Specification - 2.3.6.2. Handoff State
https://uefi.org/specs/UEFI/2.10/02_Overview.html#handoff-state-4



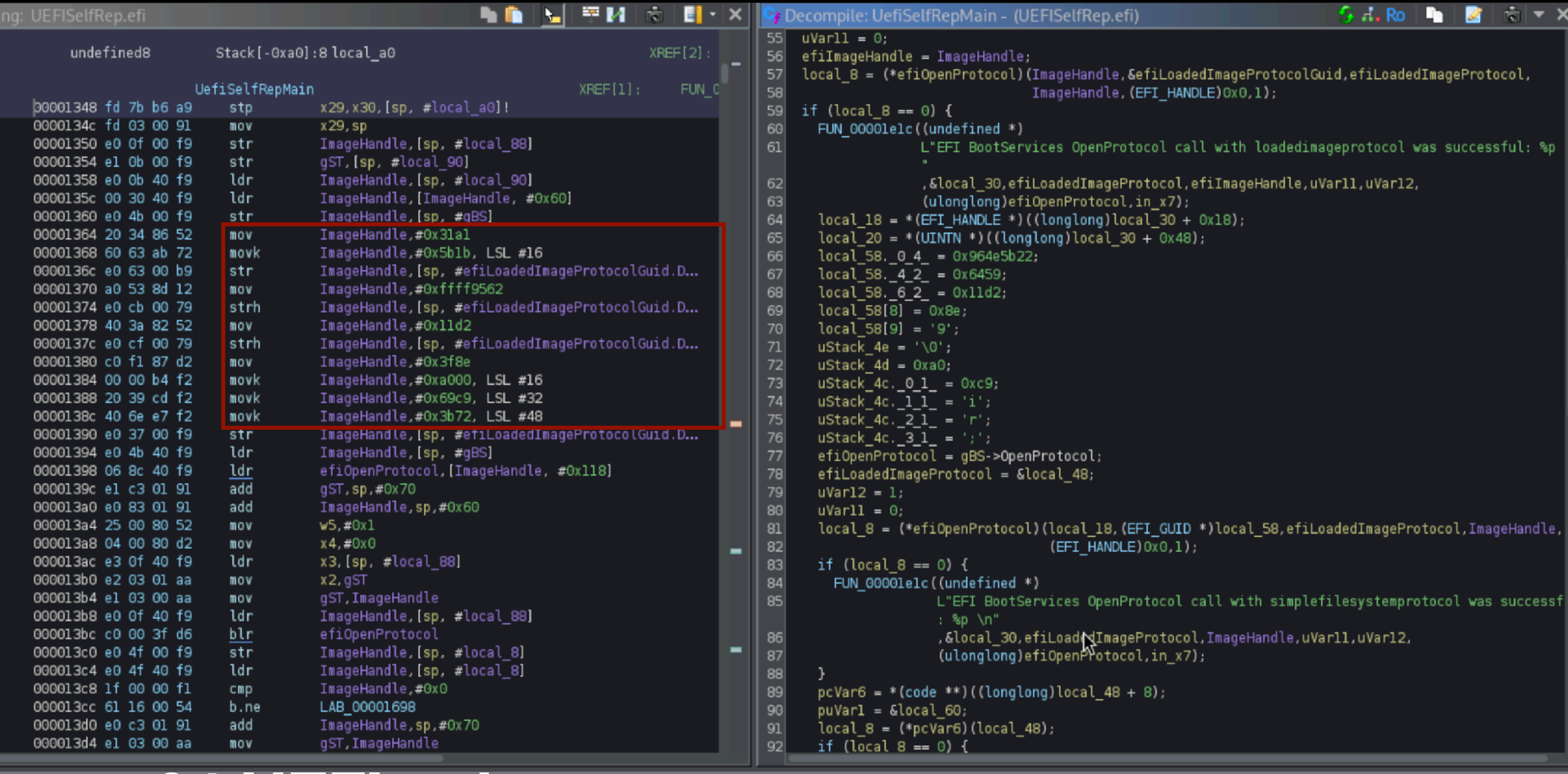
Base text: Self-replicating UEFI app

Written in C, cross-compiled for arm64

https://youtu.be/af8lanzkYyQ

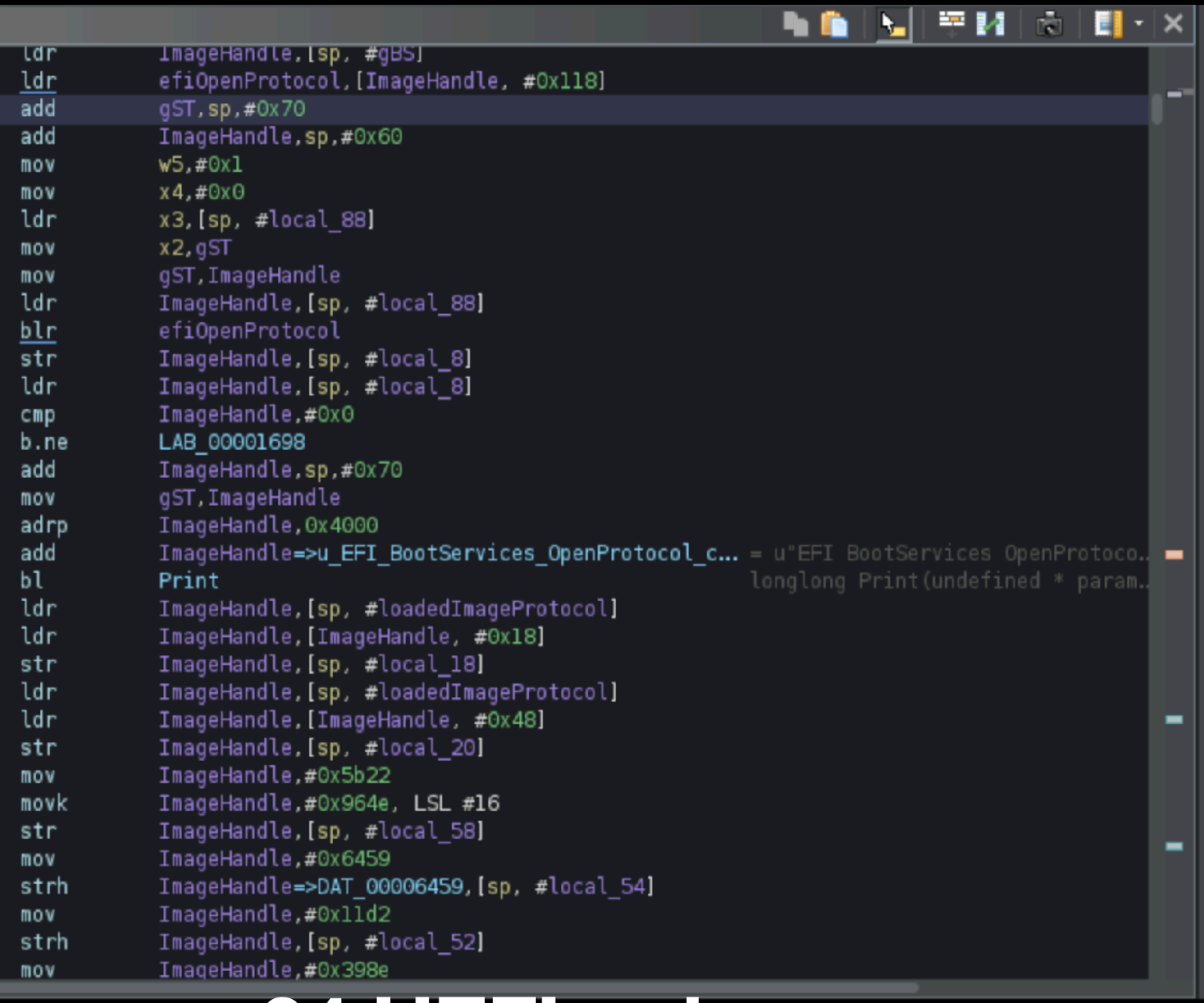
“In reality, with regard to syntax, word-for-word translation completely rejects the reproduction of meaning and threatens to lead directly to incomprehensibility.”

Walter Benjamin, “The Task of the Translator,” translated by Steven Rendall, page 161.



arm64 UEFI quine

RE and xdev



arm64 UEFI quine

RE and xdev

UEFI generation 2: arm64

Golfing the solution

1. Use architecture-specific techniques that leverage arm64 features
   - e.g. the ARM barrel shifter can be leveraged for the painstaking process of correctly loading a target 128-byte GUID into the correct register
2. Take the time to figure out the stack frame layout with all essential data for UEFI quine
3. Remember that the process for golfing a binary in one assembly language (x64) is not a 1:1 mapping of the golfed binary in a different assembly language (arm64)

```
//Stack frame:
//  [sp]            fp (x29)
//  [sp, 0x8]       lr (x30)
//  [sp, 0x10]      x1 (gST)
//  [sp, 0x18]      x0 (ImageHandle)
//  [sp, 0x20]      x19
//  [sp, 0x28]      x20
//  [sp, 0x30]
//  [sp, 0x38]      EFI_SIMPLE_FILESYSTEM_PROTOCOL* sfsp
//  [sp, 0x40]      LIP_GUID[0]
//  [sp, 0x44]      LIP_GUID[1]
//  [sp, 0x46]      LIP_GUID[2]
//  [sp, 0x48]      LIP_GUID[3]
//  [sp, 0x50]      SFSP_GUID[0]
//  [sp, 0x54]      SFSP_GUID[1]
//  [sp, 0x56]      SFSP_GUID[2]
//  [sp, 0x58]      SFSP_GUID[3]
//  [sp, 0x60]      EFI_LOADED_IMAGE_PROTOCOL* lip
//  [sp, 0x68]      UINT64 img_size (== lip->ImageSize)
//  [sp, 0x70]      EFI_HANDLE devicehandle (== lip->DeviceHandle)
//  [sp, 0x78]      UINT64 host_attributes;
//  [sp, 0x80]      EFI_FILE_PROTOCOL* rootVolume
//  [sp, 0x88]      EFI_FILE_PROTOCOL* hostFile
//  [sp, 0x90]      EFI_FILE_PROTOCOL* targetFile
//  [sp, 0x98]      UINTN newfile_buffersize
//  [sp, 0xa0]      gBS
//  [sp, 0xa8]      status
//  [sp, 0xb0]      VOID * tmp_buffer
//  [sp, 0xb8]
//  [sp, 0xc0]
```

Stack frame layout of arm64 solution

arm64 UEFI quine

x64 solution

```
_start:
entrypoint:
    push rbp
    mov rbp, rsp
    sub rsp,0xc0

    mov [ImageHandle], rcx
    mov [gST], rdx

    mov rbx, [gST]
    mov rbx, [rbx + 0x60]
    mov [gBS], rbx
    mov rax, [gST]
    mov rax, [rax + 0x40]
    mov [ConOut], rax

    mov rbx, [gBS]
    mov rdi, [rbx + 0x98]       ;gBS->HandleProtocol()
                                ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [ImageHandle]
    ;;this is how we're passing the GUID so that it works
    lea r8, [LoadedImageProtocol]
    mov dword [rbp-0x40], 0x5b1b31a1
    mov word [rbp-0x3c], 0x9562
    mov word [rbp-0x3a], 0x11d2
    mov rax, 0x3b7269c9a0003f8e
    mov [rbp-0x38], rax
    lea rdx, [rbp-0x40]
    mov r9, [ImageHandle]
    xor r10, r10
    call rdi
    cmp qword [rax], byte 0x0
    jne printerror
```

arm64 solution

```
UefiMain:
    //function prologue
    stp x29, x30, [sp, #-0xc0]!
    mov x29, sp                 // Defining local vars+storing them on the stack
    stp x19, x20, [sp, #0x20]

    str x0, [sp, #0x18]         // store imageHandle var on stack
    str x1, [sp, #0x10]         // store efiSystemTable var on stack
    ldr x0, [sp, #0x10]         // load efiSystemTable into x0
    ldr x0, [x0, #0x60]         // load x0 =SystemTable + 0x60 == gBS
    str x0, [sp, #0xa0]         // store gBS var on stack
    str xzr, [sp, #0x60]        // store LoadedImageProtocol* lip=NULL on stack
    str xzr, [sp, #0x68]        // store lip->ImageSize = 0 on stack
    str xzr, [sp, #0x70]        // store lip->DeviceHandle=NULL on stack
    str xzr, [sp, #0x80]        // store rootvolume=NULL on stack
    str xzr, [sp, #0x88]        // store hostfile=NULL on stack
    str xzr, [sp, #0x90]        // store targetfile=NULL on stack
    str xzr, [sp, #0xb0]        // store tmp_buffer=NULL on stack

    //load Loaded Image Protocol guid
    mov w0, #0x31a1
    movk w0, #0x5b1b, LSL #16
    str w0, [sp,#0x40]      // store final part of LIP GUID at sp-0x40
    mov w0, #0x9562
    strh w0, [sp, #0x44]        // store 2nd part of LIP GUID at sp-0x44
    mov w0, #0x11d2
    strh w0, [sp, #0x46]        // store 3rd part of LIP GUID at sp-0x46
    mov x0, #0x3f8e
    movk x0, #0xa000, LSL #16
    movk x0, #0x69c9, LSL #32
    movk x0, #0x3b72, LSL #48
    str x0, [sp, #0x48]     // store final part of LIP GUID at sp-0x48
    add x0, sp, #0x40       // move address of sp+40 to x0
    mov x1, x0      // address of $sp+0x40 (LIP GUID) into x1

    ldr x0, [sp, #0xa0]     // load x0 = gBS
    ldr x6, [x0, #0x98]     // load gBS+0x98 (HandleProtocol) into x6
    ldr x3, [sp, #0x18]     // load ImageHandle from stack into x3
    add x0, sp, #0x60       // move address of $sp+0x60 into x0
    mov x2, x0      // addr of $sp+0x60 (loadedImageProtocol*) into x2

    ldr x0, [sp, #0x18]     // load ImageHandle from stack into x0
    blr x6
    cmp x0, 0x0             // check if EFI_STATUS==EFI_SUCCESS (0x0)
    b.ne Print
```

arm64 UEFI quine

x64 solution

```
_start:
entrypoint:
    push rbp
    mov rbp, rsp
    sub rsp,0xc0

    mov [ImageHandle], rcx
    mov [gST], rdx

    mov rbx, [gST]
    mov rbx, [rbx + 0x60]
    mov [gBS], rbx
    mov rax, [gST]
    mov rax, [rax + 0x40]
    mov [ConOut], rax

    mov rbx, [gBS]
    mov rdi, [rbx + 0x98]       ;gBS->HandleProtocol()
                                ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [ImageHandle]
    ;;this is how we're passing the GUID so that it works
    lea r8, [LoadedImageProtocol]
    mov dword [rbp-0x40], 0x5b1b31a1
    mov word [rbp-0x3c], 0x9562
    mov word [rbp-0x3a], 0x11d2
    mov rax, 0x3b7269c9a0003f8e
    mov [rbp-0x38], rax
    lea rdx, [rbp-0x40]
    mov r9, [ImageHandle]
    xor r10, r10
    call rdi
    cmp qword [rax], byte 0x0
    jne printerror
```

arm64 solution

```
UefiMain:
    //function prologue
    stp x29, x30, [sp, #-0xc0]!
    mov x29, sp                 // Defining local vars+storing them on the stack
    stp x19, x20, [sp, #0x20]

    str x0, [sp, #0x18]         // store imageHandle var on stack
    str x1, [sp, #0x10]         // store efiSystemTable var on stack
    ldr x0, [sp, #0x10]         // load efiSystemTable into x0
    ldr x0, [x0, #0x60]         // load x0 =SystemTable + 0x60 == gBS
    str x0, [sp, #0xa0]         // store gBS var on stack
    str xzr, [sp, #0x60]        // store LoadedImageProtocol* lip=NULL on stack
    str xzr, [sp, #0x68]        // store lip->ImageSize = 0 on stack
    str xzr, [sp, #0x70]        // store lip->DeviceHandle=NULL on stack
    str xzr, [sp, #0x80]        // store rootvolume=NULL on stack
    str xzr, [sp, #0x88]        // store hostfile=NULL on stack
    str xzr, [sp, #0x90]        // store targetfile=NULL on stack
    str xzr, [sp, #0xb0]        // store tmp_buffer=NULL on stack

    //load Loaded Image Protocol guid
    mov w0, #0x31a1
    movk w0, #0x5b1b, LSL #16
    str w0, [sp,#0x40]      // store final part of LIP GUID at sp-0x40
    mov w0, #0x9562
    strh w0, [sp, #0x44]        // store 2nd part of LIP GUID at sp-0x44
    mov w0, #0x11d2
    strh w0, [sp, #0x46]        // store 3rd part of LIP GUID at sp-0x46
    mov x0, #0x3f8e
    movk x0, #0xa000, LSL #16
    movk x0, #0x69c9, LSL #32
    movk x0, #0x3b72, LSL #48
    str x0, [sp, #0x48]     // store final part of LIP GUID at sp-0x48
    add x0, sp, #0x40       // move address of sp+40 to x0
    mov x1, x0      // address of $sp+0x40 (LIP GUID) into x1

    ldr x0, [sp, #0xa0]     // load x0 = gBS
    ldr x6, [x0, #0x98]     // load gBS+0x98 (HandleProtocol) into x6
    ldr x3, [sp, #0x18]     // load ImageHandle from stack into x3
    add x0, sp, #0x60       // move address of $sp+0x60 into x0
    mov x2, x0      // addr of $sp+0x60 (loadedImageProtocol*) into x2

    ldr x0, [sp, #0x18]     // load ImageHandle from stack into x0
    blr x6
    cmp x0, 0x0             // check if EFI_STATUS==EFI_SUCCESS (0x0)
    b.ne Print
```

arm64 UEFI quine

x64 solution

```
_start:
entrypoint:
    push rbp
    mov rbp, rsp
    sub rsp,0xc0

    mov [ImageHandle], rcx
    mov [gST], rdx

    mov rbx, [gST]
    mov rbx, [rbx + 0x60]
    mov [gBS], rbx
    mov rax, [gST]
    mov rax, [rax + 0x40]
    mov [ConOut], rax

    mov rbx, [gBS]
    mov rdi, [rbx + 0x98]       ;gBS->HandleProtocol()
                                ; params passed in rcx, rdx, r8, r9, r10
    mov rcx, [ImageHandle]
    ;;this is how we're passing the GUID so that it works
    lea r8, [LoadedImageProtocol]
    mov dword [rbp-0x40], 0x5b1b31a1
    mov word [rbp-0x3c], 0x9562
    mov word [rbp-0x3a], 0x11d2
    mov rax, 0x3b7269c9a0003f8e
    mov [rbp-0x38], rax
    lea rdx, [rbp-0x40]
    mov r9, [ImageHandle]
    xor r10, r10
    call rdi
    cmp qword [rax], byte 0x0
    jne printerror
```

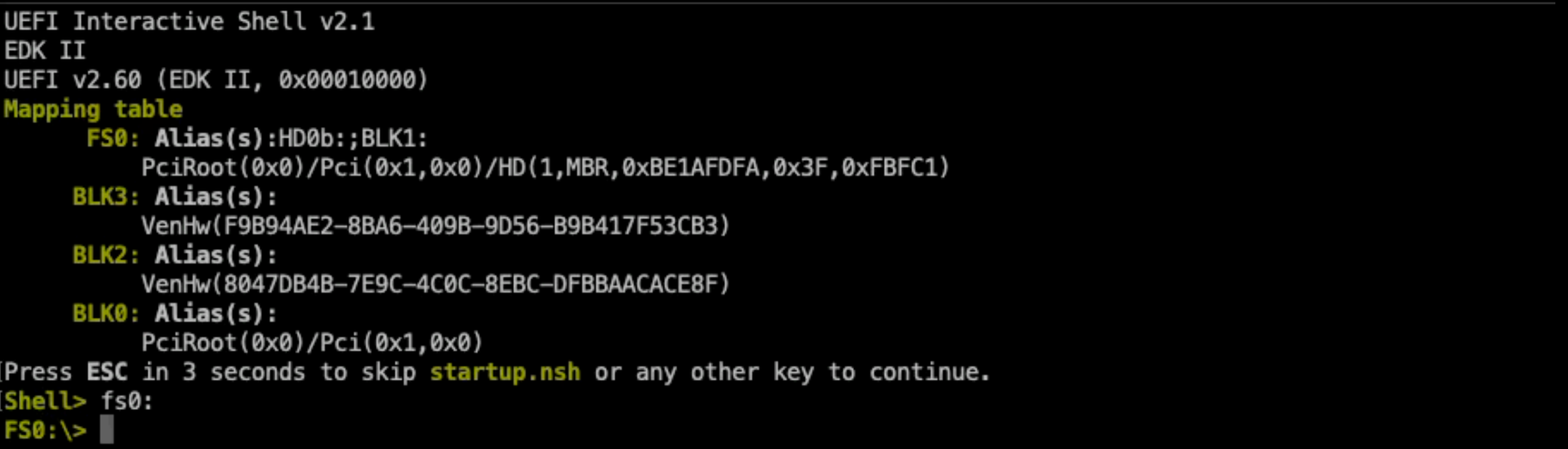
arm64 solution

```
UefiMain:
    //function prologue
    stp x29, x30, [sp, #-0xc0]!
    mov x29, sp                 // Defining local vars+storing them on the stack
    stp x19, x20, [sp, #0x20]

    str x0, [sp, #0x18]         // store imageHandle var on stack
    str x1, [sp, #0x10]         // store efiSystemTable var on stack
    ldr x0, [sp, #0x10]         // load efiSystemTable into x0
    ldr x0, [x0, #0x60]         // load x0 =SystemTable + 0x60 == gBS
    str x0, [sp, #0xa0]         // store gBS var on stack
    str xzr, [sp, #0x60]        // store LoadedImageProtocol* lip=NULL on stack
    str xzr, [sp, #0x68]        // store lip->ImageSize = 0 on stack
    str xzr, [sp, #0x70]        // store lip->DeviceHandle=NULL on stack
    str xzr, [sp, #0x80]        // store rootvolume=NULL on stack
    str xzr, [sp, #0x88]        // store hostfile=NULL on stack
    str xzr, [sp, #0x90]        // store targetfile=NULL on stack
    str xzr, [sp, #0xb0]        // store tmp_buffer=NULL on stack

    //load Loaded Image Protocol guid
    mov w0, #0x31a1
    movk w0, #0x5b1b, LSL #16
    str w0, [sp,#0x40]      // store final part of LIP GUID at sp-0x40
    mov w0, #0x9562
    strh w0, [sp, #0x44]        // store 2nd part of LIP GUID at sp-0x44
    mov w0, #0x11d2
    strh w0, [sp, #0x46]        // store 3rd part of LIP GUID at sp-0x46
    mov x0, #0x3f8e
    movk x0, #0xa000, LSL #16
    movk x0, #0x69c9, LSL #32
    movk x0, #0x3b72, LSL #48
    str x0, [sp, #0x48]     // store final part of LIP GUID at sp-0x48
    add x0, sp, #0x40       // move address of sp+40 to x0
    mov x1, x0      // address of $sp+0x40 (LIP GUID) into x1

    ldr x0, [sp, #0xa0]     // load x0 = gBS
    ldr x6, [x0, #0x98]     // load gBS+0x98 (HandleProtocol) into x6
    ldr x3, [sp, #0x18]     // load ImageHandle from stack into x3
    add x0, sp, #0x60       // move address of $sp+0x60 into x0
    mov x2, x0      // addr of $sp+0x60 (loadedImageProtocol*) into x2

    ldr x0, [sp, #0x18]     // load ImageHandle from stack into x0
    blr x6
    cmp x0, 0x0             // check if EFI_STATUS==EFI_SUCCESS (0x0)
    b.ne Print
```

UEFI generation 2: arm64

RE and development tools

- Write the assembly program and build it with the edk2 build system
  - This was easiest option because I wrote this on an arm64 machine (an M1 MacBook Pro) but the bindings for arm64 with the native Xcode Tools command line tools are for *Darwin* arm64 and for generating Mach-O arm64 binaries
  - UEFI apps and drivers are predominately PE files (and occasionally TE) that don't use the Darwin bindings
  - The edk2 build system finally came through and was up to this task of generating arm64 UEFI apps
- For an assembler with solid UEFI support, there is the ARM-specific flavor of FASM: FASMARM: https://arm.flatassembler.net/
  - [Note FASMARM only supports 32-bit and 64-bit ARM architectures up until v8; valid solution for ARM32 builds but not arm64 builds)
- Hex editor (xxd, hexdump)
- Ghidra with efiSeek and ghidra-firmware-utils
- radare2 for disassembly
- QEMU and gdb for debugging/testing



Final arm64 quine:
Self-replicating UEFI app

Written in arm64 assembly

https://youtu.be/C-jaMoCwdJE

arm64 UEFI debugging

qemu & gdb

```
/Users/nika/uefi_testing/edk2/MdePkg/Library/BaseLib/DivU64x32Remainder.c
  1 /** @file
  2   Math worker functions.
  3
  4   Copyright (c) 2006 - 2008, Intel Corporation. All rights reserved.<BR>
  5   SPDX-License-Identifier: BSD-2-Clause-Patent
  6
  7 **/
  8
  9 #include "BaseLibInternals.h"
 10
 11 /**
 12   Divides a 64-bit unsigned integer by a 32-bit unsigned integer and generates
 13   a 64-bit unsigned result and an optional 32-bit unsigned remainder.
 14
```

```
   0x78760ac8 <DivU64x32Remainder>         stp     x29, x30, [sp, #-48]!
   0x78760acc <DivU64x32Remainder+4>       mov     x29, sp
   0x78760ad0 <DivU64x32Remainder+8>       str     x0, [sp, #40]
   0x78760ad4 <DivU64x32Remainder+12>      str     w1, [sp, #36]
   0x78760ad8 <DivU64x32Remainder+16>      str     x2, [sp, #24]
   0x78760adc <DivU64x32Remainder+20>      bl      0x7875e678 <DebugAssertEnabled>
   0x78760ae0 <DivU64x32Remainder+24>      and     w0, w0, #0xff
   0x78760ae4 <DivU64x32Remainder+28>      cmp     w0, #0x0
   0x78760ae8 <DivU64x32Remainder+32>      b.eq    0x78760b10 <DivU64x32Remainder+72>  // b.none
   0x78760aec <DivU64x32Remainder+36>      ldr     w0, [sp, #36]
   0x78760af0 <DivU64x32Remainder+40>      cmp     w0, #0x0
   0x78760af4 <DivU64x32Remainder+44>      b.ne    0x78760b10 <DivU64x32Remainder+72>  // b.any
   0x78760af8 <DivU64x32Remainder+48>      adrp    x0, 0x78761000 <CpuBreakpoint+912>
   0x78760afc <DivU64x32Remainder+52>      add     x2, x0, #0xe0
```

```
exec No process In:                                                    L??   PC: ??
(No debugging symbols found in UEFI_bb_disk/UefiQuineAarch64.efi)
(gdb) info files
Symbols from "/Users/nika/uefi-task-of-the-translator/Aarch64_UEFI_exploits/UEFI_bb_disk/UefiQuineAarch64.efi".
Local exec file:
        `/Users/nika/uefi-task-of-the-translator/Aarch64_UEFI_exploits/UEFI_bb_disk/UefiQuineAarch64.efi', file type pei-aarch64-little.
        Entry point: 0x1000
        0x0000000000001000 - 0x0000000000005000 is .text
        0x0000000000005000 - 0x0000000000006000 is .data
        0x0000000000006000 - 0x0000000000007000 is .reloc
(gdb) add-symbol-file ~/uefi_testing/edk2/Build/BareBonesPkg/DEBUG_GCC/Aarch64/UefiQuineAarch64.debug 0x7875e000 -s .data 0x78762000
add symbol table from file "/Users/nika/uefi_testing/edk2/Build/BareBonesPkg/DEBUG_GCC/Aarch64/UefiQuineAarch64.debug" at
        .text_addr = 0x7875e000
        .data_addr = 0x78762000
(y or n) y
Reading symbols from /Users/nika/uefi_testing/edk2/Build/BareBonesPkg/DEBUG_GCC/Aarch64/UefiQuineAarch64.debug...
(gdb)
```

UEFI generation 2: arm64

What did you learn at school today?

- Leverage the UEFI ecosystem by walking from Protocol interface to Protocol interface —> better understanding of UEFI internals and base knowledge for building better exploits
  - Building ROP chains for arm64 exploits
- Learning how to set up debugging for arm64 UEFI apps/drivers
- Knowledge of how to write UEFI shellcode for arm64
  - Expanded repertoire of knowledge and skills for UEFI exploit dev
  - Additional working payloads for arm64 UEFI exploits

UEFI generation 3: EBC

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)
- If a compiler is a translator, then EBC could be considered the holy scripture [per Benjamin's metaphor])…

“For to some degree all great writings, but above all holy scripture, contain their virtual translation between the lines. The interlinear version of the holy scriptures is the prototype or ideal of all translation.”

Walter Benjamin, “The Task of the Translator,” translated by Steven Rendall, page 165.

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:

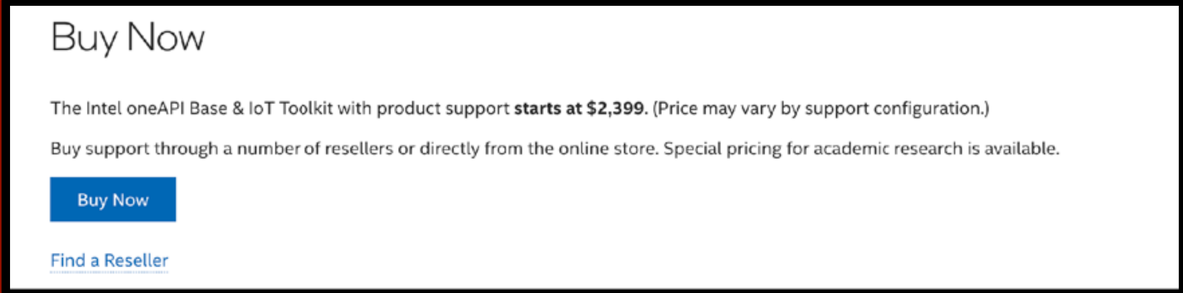
```
Offset = (c + n * (sizeof (VOID *))) * sign
```

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?

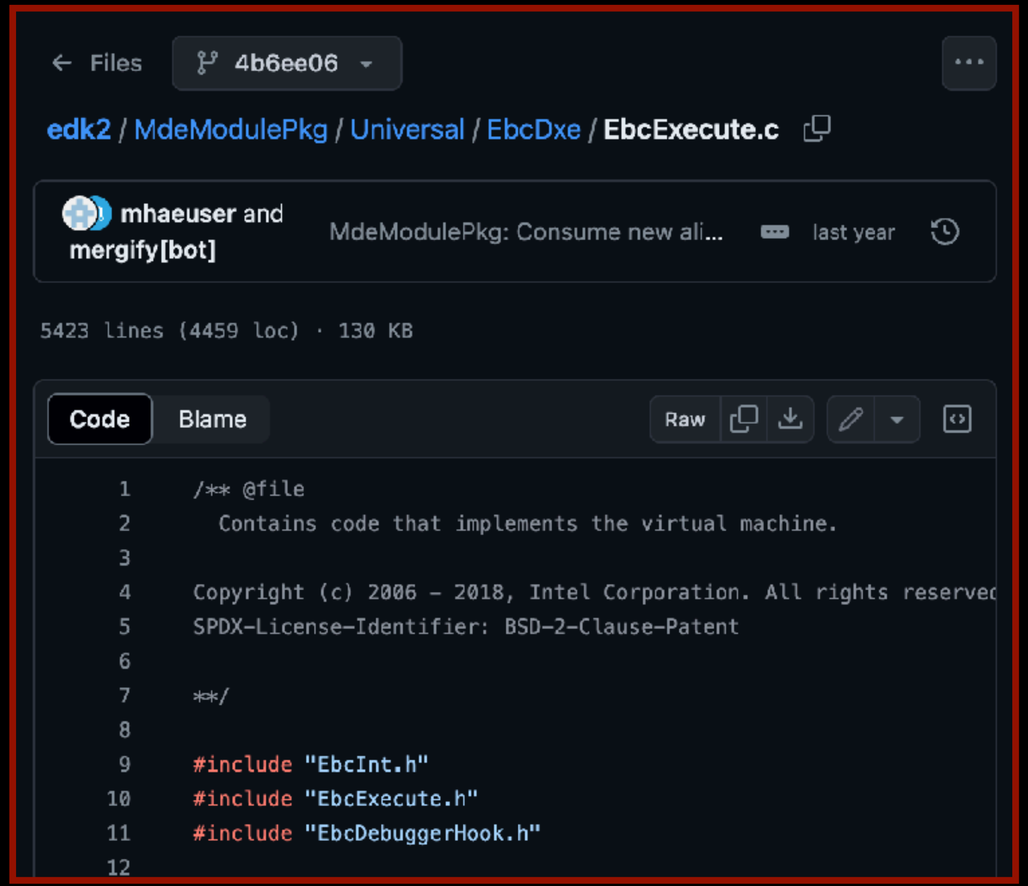
- Only one compiler specifically designed to target valid EBC binaries: the proprietary Intel C compiler for EBC
- This proprietary Intel C compiler for EBC was available for the low price of $995 [to my knowledge, it is no longer available; now the page on the Intel Products site redirects to an IoT 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]



EBC - EFI Byte Code

The nearly impossible task of writing a UEFI EBC quine

- Frankly, I became obsessed with EBC
- My goals for this project were the following:
- 1. Translate my original self-replicating UEFI app from x64 to EBC and confirm that a self-replicating EBC UEFI app could run in a standard UEFI environment.
- 2. Leveraging my new knowledge of EBC development, use my self-replicating EBC app as a template for a new EBC UEFI virus that performs graphics manipulation via the GOP (Graphics Output Protocol).
- 3. Explore uses of EBC and the EBCVM for novel UEFI malware development -- including but not limited to PCI option ROM attacks, polymorphism, metamorphism and graphics.

EBC - EFI Byte Code

Translating UEFI quine from x64 to EBC: HandleProtocol()

```
New_Handle_Protocol:
;--- Built functiion stack frame ---
        PUSHN       R4      ; Parm#4 = Image Handle
        PUSHN       R0      ;push 3rd parameter - protocol pointer
        PUSHN       R3      ;push 2nd param - Pointer to GUID
        PUSHN       R2      ;push 1st param - Image Handle
;--- Read pointer and handler call ---
        MOVNW       R3,@R1,0,_EFI_Table ; R3 = SysTable
        MOVNW       R3,@R3,9,24     ; R3 = BootServices
        CALL32EXA   @R3,16,24       ; Entry #16 = Handle Protocol
;--- Remove stack frame ---
        POPN        R2      ; push 1st param
        POPN        R3      ; pop 2nd param - pointer to GUID
        POPN        R3      ; pop 3rd param [result] - protocol pointer
        POPN        R4      ; pop 4th param
;--- Check status and result ---
        MOVSNW      R7,R7
        CMPI64WUGTE R7,1        ; Check status
        JMP8CS      Bad_Config
        CMPI64WEQ   R3,0        ; Check protocol pointer
```

My implementation of the HandleProtocol() function in EBC UEFI quine

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/

Applications of “The Task of the Translator” to UEFI xdev and malware art

> What do they say about programming in C code? It's like smoking a cigarette in a swimming pool full of gasoline. OK, enough of the jokes and back to the blog.

Source: Vincent Zimmer, “EFI Byte Code,” Saturday, August 1, 2015, https://vzimmer.blogspot.com/2015/08/efi-byte-code.html

UEFI Exploit dev

Well, how did I get here?

- My research on this began after I kept running into the same problem at work: I was *finding* UEFI vulnerabilities, but I didn't know how to write exploits for UEFI
- UEFI exploit dev is like many forms of xdev/malware development/RE: an ongoing process
- How did I learn to write UEFI exploits?
  - 1. Reverse engineering and replicating the techniques of other PoCs [Translating PoC's, if you will]
  - 2. Learning about UEFI by writing UEFI apps and drivers [How do you learn a language? How do you learn to write UEFI exploits? Exploit dev is like learning a language: it requires practice and accepting that you'll fail many times before you communicate what you want to say
    (e.g. pwn a target)]

SMM Callout Exploit dev

Reverse Engineering earlier malware/PoCs

- How does one start writing an exploit for a new system/an unfamiliar target?
- Understand the target:
  - Build foundational knowledge (RTFM - the UEFI spec, Beyond BIOS, Rootkits and Bootkits)
  - Find previous notable work in UEFI exploit development/malware, and read, re-read the base text
  - “Translate a base text” : Try to translate the same exploit technique on a different vulnerable target
    - e.g. Use cr4sh’s SMM callout PoC for a vulnerability in SystemSmmAhciAspiLegacyRt [“Exploiting SMM callout vulnerabilities in Lenovo firmware”, http://blog.cr4.sh/2016/02/exploiting-smm-callout-vulnerabilities.html], as a template for writing an SMM callout exploit for a vulnerability in an IdeBusDxe driver

UEFI Exploit dev

Reverse Engineering earlier malware/PoCs

- There is no ROP Emporium for UEFI specifically, and there are very few examples of UEFI-specific CTF challenges [Notable exception: SMM Cowsay from UIUCTF 2022, which we'll return to] that you can use for practice
- But there are good resources for learning all of the skills you'll need to write UEFI exploits
  - No exploit dev roadmap?
    Honey, that's what I call a make-your-own-adventure CTF

UEFI Exploit dev

Make-your-own-adventure CTF

- (OST2) Architecture 4021: Introductory UEFI
  https://ost2.fyi/Arch4021
- (OST2) Architecture 4001: x86-64 Intel Firmware Attack & Defense
  https://ost2.fyi/Arch4001
- (OST2) Hardware 1101: Intel SPI Analysis
  https://ost2.fyi/HW1101
- UEFI-Lessons by Kostr: https://github.com/Kostr/UEFI-Lessons/
- Tools
  - Chipsec: https://chipsec.github.io/
  - UEFITool: https://github.com/LongSoft/UEFITool

UEFI Exploit dev: SMM Callouts

Started from ring -2 now we’re calling out to an attacker-controlled region of memory

SMM Callout

?????

```
EFI_STATUS __fastcall ChildSwSmiHandler(
        EFI_HANDLE DispatchHandle,
        const void *Context,
        char *CommBuffer,
        UINTN *CommBufferSize)
{
  EFI_STATUS v5; // rbx
  EFI_STATUS v6; // rax
  UINTN v7; // rbx
  EFI_STATUS result; // rax
  INTN v9; // rsi
  EFI_STATUS v10; // r12
  EFI_HANDLE Buffer; // [rsp+30h] [rbp-40h] BYREF
  UINTN BufferSize; // [rsp+38h] [rbp-38h] BYREF
  UINTN NoHandles; // [rsp+40h] [rbp-30h] BYREF
  EFI_LOADED_IMAGE_PROTOCOL *EfiLoadedImageProtocol; // [rsp+48h] [rbp-28h] BYREF
  EFI_ACPI_SUPPORT_PROTOCOL *EfiAcpiSupportProtocol; // [rsp+50h] [rbp-20h] BYREF
  void *Table; // [rsp+58h] [rbp-18h] BYREF
  UINTN Handle[2]; // [rsp+60h] [rbp-10h] BYREF
  EFI_ACPI_TABLE_VERSION Version; // [rsp+A0h] [rbp+30h] BYREF

  if ( !CommBuffer || !CommBufferSize )
    return 0i64;
  if ( *(_DWORD *)CommBuffer == 1 )
  {
    Buffer = 0i64;
    if ( gBS->LocateHandleBuffer(ByProtocol, &EFI_ATA_PASS_THRU_PROTOCOL_GUID, 0i64, &NoHandles, (EFI_HANDLE **)&Buffer)
    {
      v5 = 0x8000000000000000ui64;
```

000013A1 ChildSwSmiHandler:27 (13A1)

UEFI Exploit dev: SMM Callouts

Started from ring -2 now we're calling out to an attacker-controlled region of memory

- So… you found an SMM callout vulnerability in a combined SMM/DXE driver. Now what?
- Well… How does an exploit for an SMM callout work?
  - What process is it disrupting or manipulating or interfering with?
  - What is the starting state of the UEFI firmware's environment before and after a successful SMM callout exploit?
- What are the critical data structures to know?
  - SMRAM
  - EFI Boot Services Table & EFI Runtime Services Table
  - EFI System Table
  - SMMC

SMM (System Management Mode)

Overview

- The most privileged x86 processor mode — ring -2 [we’re going to ignore ME, but yes that’s ring -3, good job]
- The processor enters SMM only when a System Management Interrupt (SMI) is invoked
- SMIs have the highest priority of all interrupts — higher priority than NMIs (non-maskable interrupts) and MIs (maskable interrupts)
- SMM is meant to act as a privileged and *separate* (read isolated) processor mode for handling critical system functionality that needs to proceed uninterrupted (i.e. power management, etc.)
- SMM code and data reside in SMRAM

SMM (System Management Mode)

SMRAM

- SMM code and data (meaning SMI handler code and data) is stored in SMRAM
- SMRAM = a protected region of a processor's address space, dedicated to storing SMM code and data
- SMRAM is locked (or it should be) during Platform Initialization (PI), so that SMM code and data in SMRAM are not accessible by code outside of SMRAM
  - SMRAM code should not be reachable by code running in kernel space or userspace
- Entering SMM is triggered by an SMI, which includes *saving execution context of code running outside of SMRAM*
- After execution of SMI handler code, RSM instruction triggers the restoration of the initial saved state

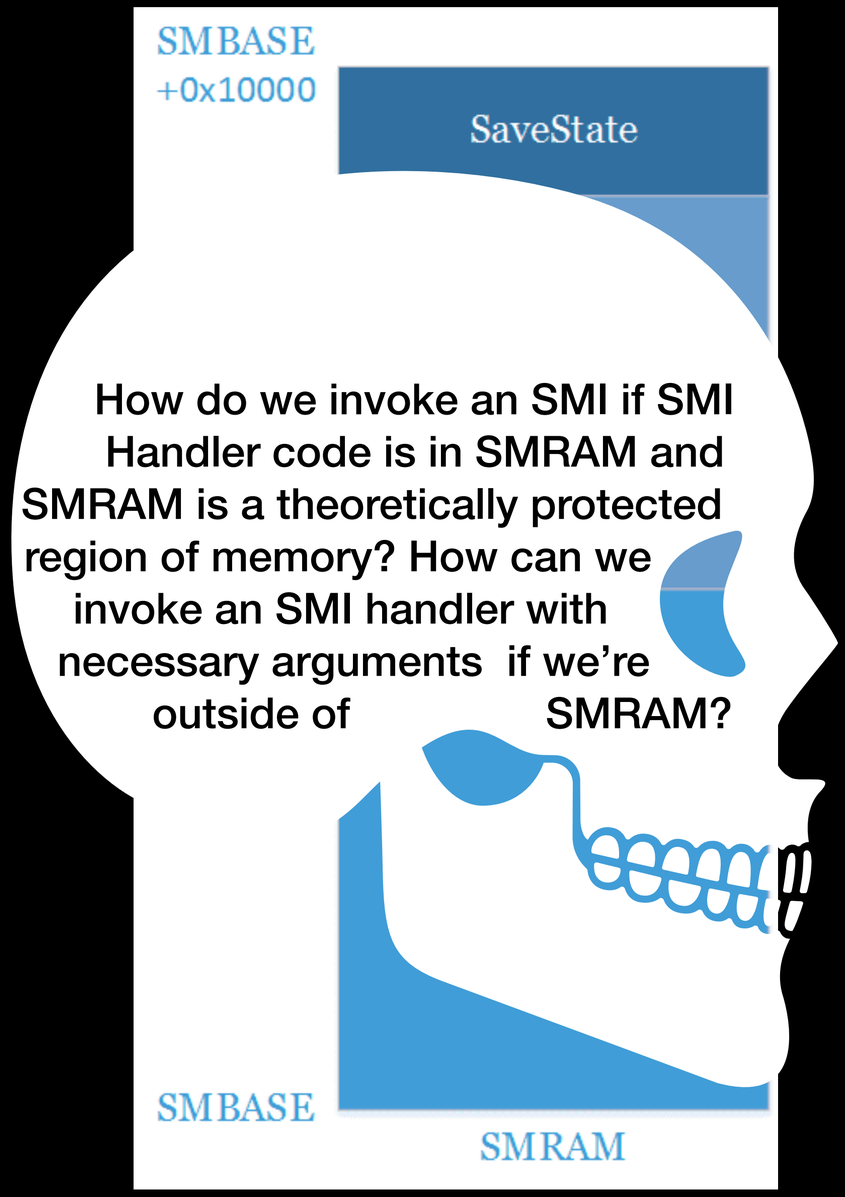


Image credit:
"Through the SMM Class and a Vulnerability Found There." Bruno Pujos, January 14, 2020, Synactiv
https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there.html

SMM (System Management Mode)

The Communication Buffer

- SMM_Core_Private_Data structure:
  - Used as a shared buffer for data during communication between SMRAM/non-SMRAM
  - Easily identifiable by “smmc” signature in memory
- EFI_SMM_Communicate Protocol requires that the Smm Communicate Buffer has the following structure:
  - GUID of SmiHandler you want to communicate with
  - The size of the data you’re sending to the SMI handler
  - The data

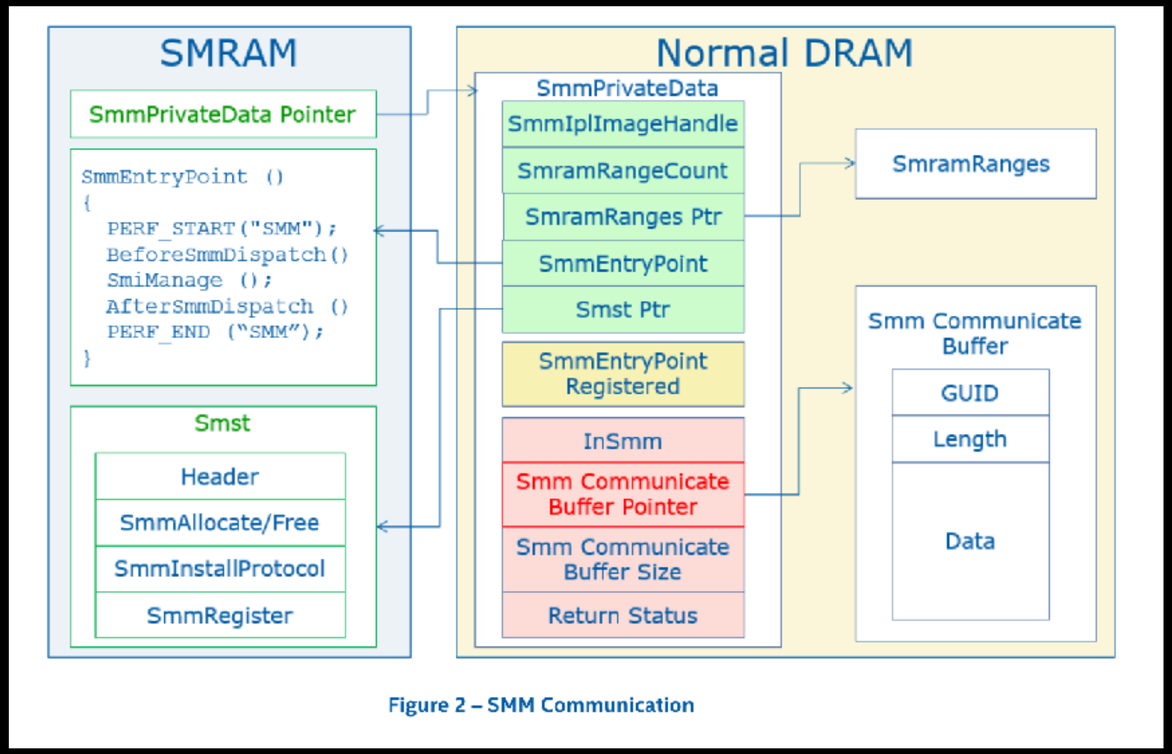


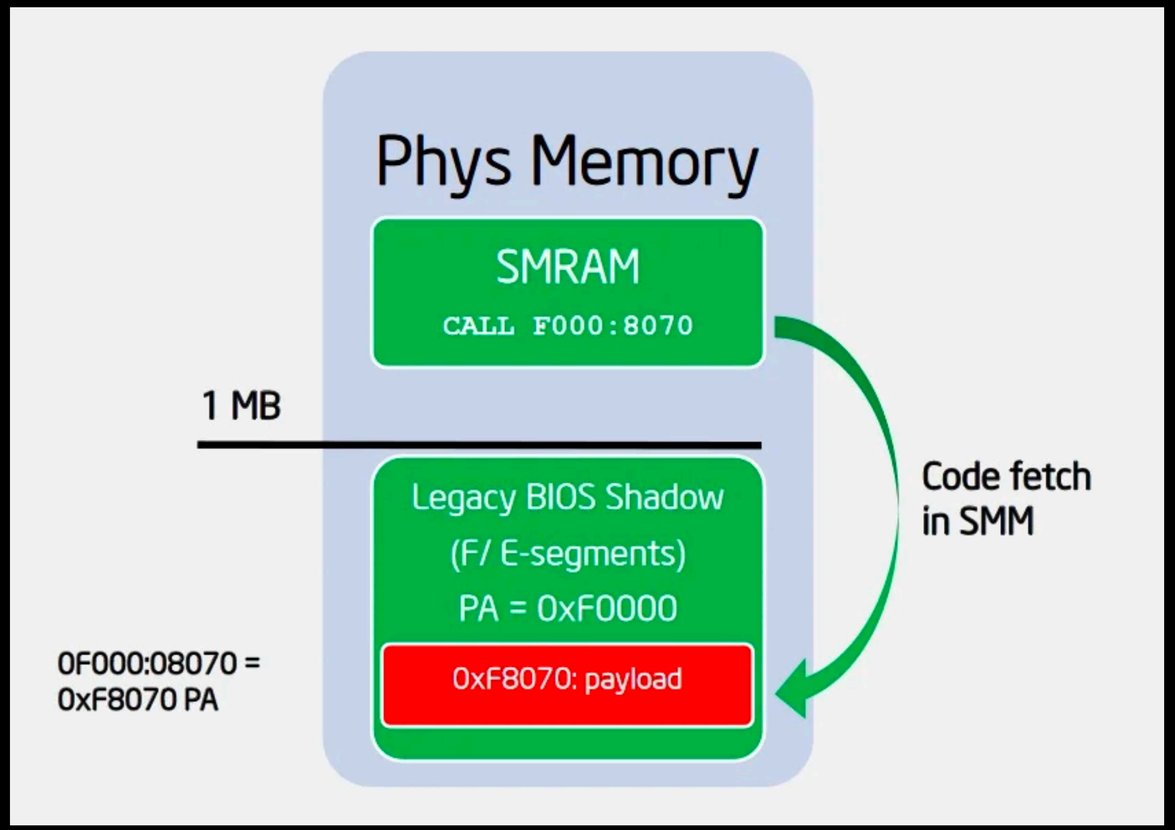
Figure 2 – SMM Communication

Source:
“A Tour Beyond BIOS Secure SMM Communication in the EFI Developer Kit II”
Jiewen Yao, Vincent J. Zimmer, Star Zeng, Intel, April 26, 2016

SMM Callouts

How

- When code running in SMM (so SMI handler code) reaches out to a data structure/code located *outside* of SMRAM, an SMM callout vuln can arise
- SMRAM == **safe** (relatively)
- EFI_BOOT_SERVICES and EFI_RUNTIME_SERVIES == data structures that are located outside of SMRAM
  - Code in either of these data structures can be attacker controlled!

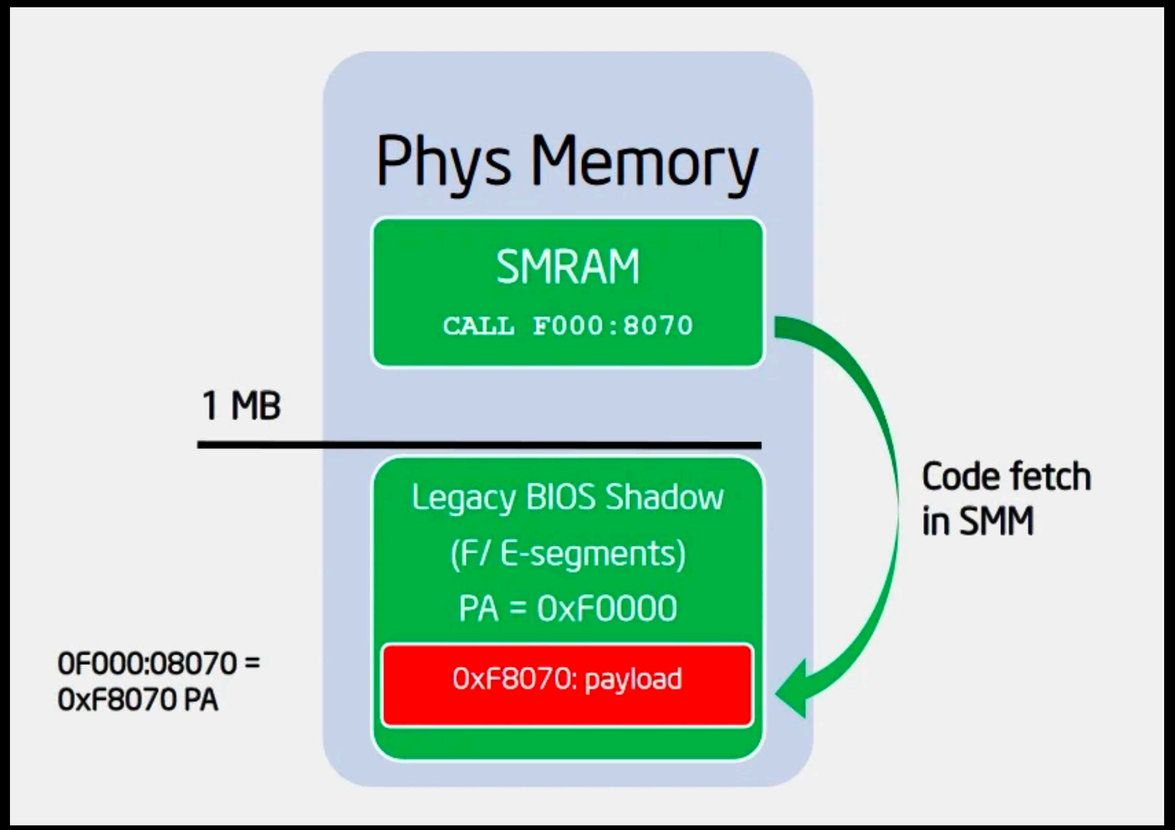


Source: "A New Class of Vulnerabilities in SMI Handlers,"
Figure 1 – Schematic overview of an SMM callout, source: CanSecWest 2015

SMM Callouts

why should I care?

- A successful SMM exploit could allow an attacker arbitrary code execution within the most privileged execution level (ring -2) of the OS
- Ring -2 code execution would effectively bypass security protections at all other execution levels and allow an attacker to install a persistent malicious firmware backdoor or implant.



Source: “A New Class of Vulnerabilities in SMI Handlers,”
Figure 1 – Schematic overview of an SMM callout, source: CanSecWest 2015

SMM Callout

IdeBusDxe
[instance of the IdeBusDxe vulnerability reported by Binarly
BRLY-2021-020
CVE-2021-45970]

```
EFI_STATUS __fastcall ChildSwSmiHandler(
        EFI_HANDLE DispatchHandle,
        const void *Context,
        char *CommBuffer,
        UINTN *CommBufferSize)
{
  EFI_STATUS v5; // rbx
  EFI_STATUS v6; // rax
  UINTN v7; // rbx
  EFI_STATUS result; // rax
  INTN v9; // rsi
  EFI_STATUS v10; // r12
  EFI_HANDLE Buffer; // [rsp+30h] [rbp-40h] BYREF
  UINTN BufferSize; // [rsp+38h] [rbp-38h] BYREF
  UINTN NoHandles; // [rsp+40h] [rbp-30h] BYREF
  EFI_LOADED_IMAGE_PROTOCOL *EfiLoadedImageProtocol; // [rsp+48h] [rbp-28h] BYREF
  EFI_ACPI_SUPPORT_PROTOCOL *EfiAcpiSupportProtocol; // [rsp+50h] [rbp-20h] BYREF
  void *Table; // [rsp+58h] [rbp-18h] BYREF
  UINTN Handle[2]; // [rsp+60h] [rbp-10h] BYREF
  EFI_ACPI_TABLE_VERSION Version; // [rsp+A0h] [rbp+30h] BYREF

  if ( !CommBuffer || !CommBufferSize )
    return 0i64;
  if ( *(_DWORD *)CommBuffer == 1 )
  {
    Buffer = 0i64;
    if ( gBS->LocateHandleBuffer(ByProtocol, &EFI_ATA_PASS_THRU_PROTOCOL_GUID, 0i64, &NoHandles, (EFI_HANDLE **)&Buffer)
    {
      v5 = 0x8000000000000000ui64;
```

SwSmi Handler executing code in SMRAM

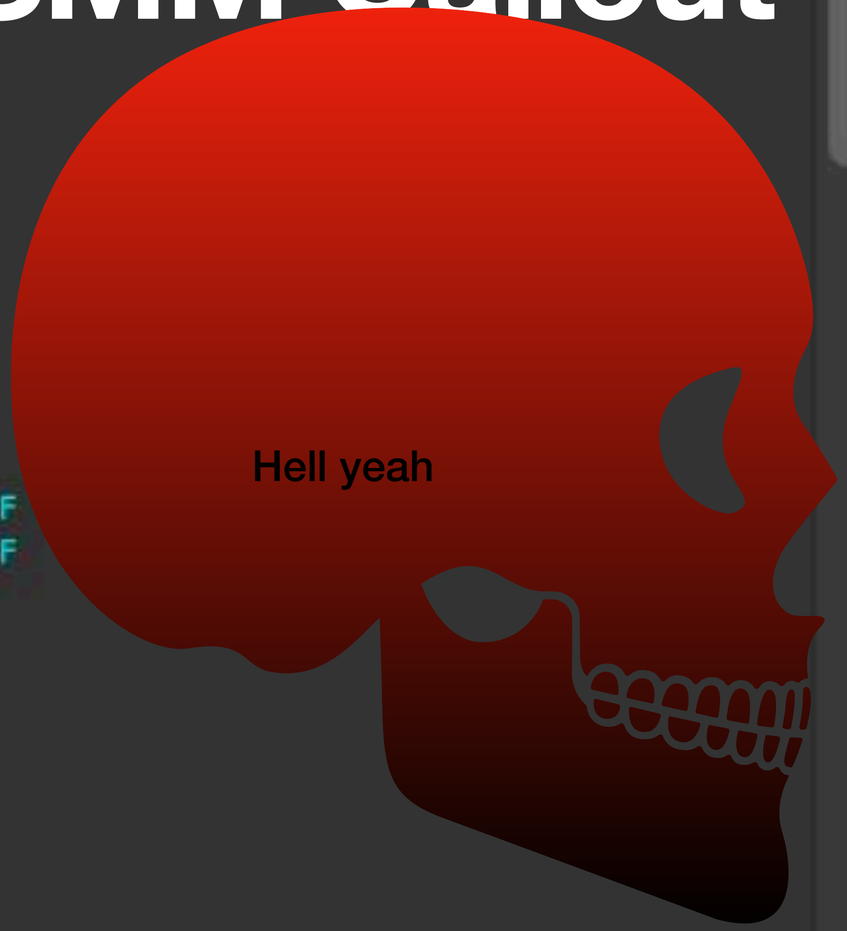
Necessary conditions for callout:
CommBuffer != NULL
CommBufferSize != NULL
first DWORD of CommBuffer == 1

SwSmi Handler calling out to function in *attacker-controlled* EFI_BOOT_SERVICES table

000013A1 ChildSwSmiHandler:27 (13A1)

SMM Callout

```
EFI_STATUS __fastcall ChildSwSmiHandler(
        EFI_HANDLE DispatchHandle,
        const void *Context,
        char *CommBuffer,
        UINTN *CommBufferSize)
{
  EFI_STATUS v5; // rbx
  EFI_STATUS v6; // rax
  UINTN v7; // rbx
  EFI_STATUS result; // rax
  INTN v9; // rsi
  EFI_STATUS v10; // r12
  EFI_HANDLE Buffer; // [rsp+30h] [rbp-40h] BYREF
  UINTN BufferSize; // [rsp+38h] [rbp-38h] BYREF
  UINTN NoHandles; // [rsp+40h] [rbp-30h] BYREF
  EFI_LOADED_IMAGE_PROTOCOL *EfiLoadedImageProtocol; // [rsp+48h] [rbp-28h] BYREF
  EFI_ACPI_SUPPORT_PROTOCOL *EfiAcpiSupportProtocol; // [rsp+50h] [rbp-20h] BYREF
  void *Table; // [rsp+58h] [rbp-18h] BYREF
  UINTN Handle[2]; // [rsp+60h] [rbp-10h] BYREF
  EFI_ACPI_TABLE_VERSION Version; // [rsp+A0h] [rbp+30h] BYREF

  if ( !CommBuffer || !CommBufferSize )
    return 0i64;
  if ( *(_DWORD *)CommBuffer == 1 )
  {
    Buffer = 0i64;
    if ( gBS->LocateHandleBuffer(ByProtocol, &EFI_ATA_PASS_THRU_PROTOCOL_GUID, 0i64, &NoHandles, (EFI_HANDLE **)&Buffer)
    {
      v5 = 0x8000000000000000ui64;
```



000013A1 ChildSwSmiHandler:27 (13A1)

SMM callout exploit dev

Methodology overview, v.1
Adapted from base text: “Exploiting SMM callout vulnerabilities in Lenovo firmware” by cr4sh

Since SMI Handler is making a call *out* of SMRAM to a function in this data structure -- EFI_BOOT_SERVICES -- and EFI_BOOT_SERVICES can be attacker-controlled, an attacker would need to do the following to exploit this SMM callout and achieve arbitrary code execution in ring -2.

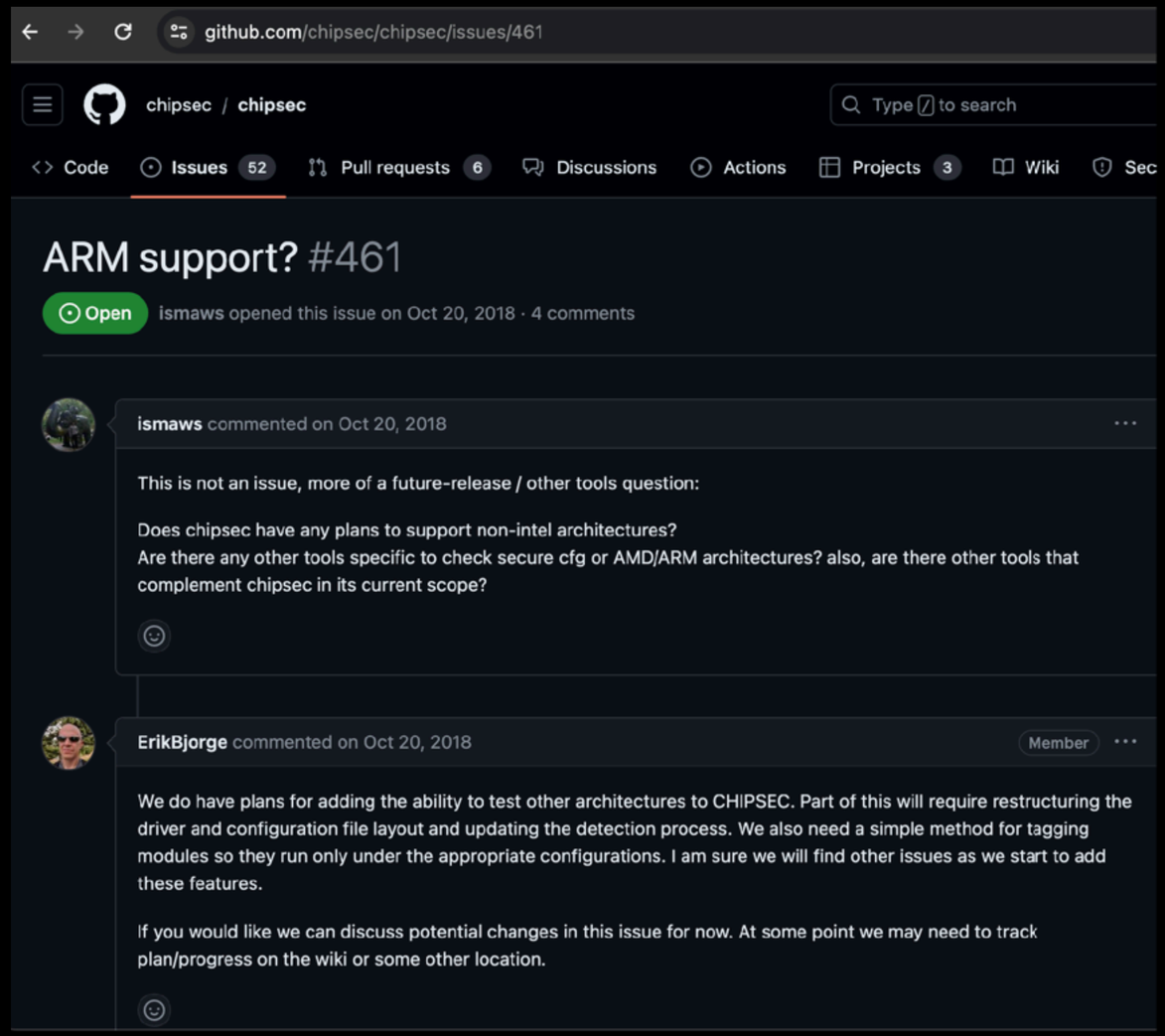
1. Identify the location of the EFI_BOOT_SERVICES data structure in memory
2. Determine the SW SMI which triggers the execution of the callout in vulnerable driver
3. Allocate space for shellcode in memory + save address of shellcode for use in step 4
4. Set the address of the LocateHandleBuffer function within the EFI_BOOT_SERVICES table to point to the address of shellcode (overwrite function pointer of LocateHandleBuffer to redirect code flow)
5. Trigger the SW SMI using the identified SW SMI number identified in step 2.
6. Attacker shellcode is executed in ring -2

SMM Callout exploit v. 1

SMM Callout v.1

Chipsec

- Back to the drawing board



UEFI Exploit dev: SMM Callouts

Started from ring -2 now we’re calling out to an attacker-controlled region of memory

~*There are no binary exploitation mitigations present in the vulnerable SMM/DXE driver but Chipsec won’t run on the host machine so now we’re reversing the swsmi function in Chipsec and replicating its functionality in C *~

SMM callout exploit dev

Methodology overview, v.2

Since SMI Handler is making a call *out* of SMRAM to a function in this data structure -- EFI_BOOT_SERVICES -- and EFI_BOOT_SERVICES can be attacker-controlled, an attacker would need to do the following to exploit this SMM callout and achieve arbitrary code execution in ring -2.

1. Identify the location of the EFI_BOOT_SERVICES data structure in memory
2. Determine the SW SMI which triggers the execution of the callout in vulnerable driver
3. Allocate space for shellcode in memory + save address of shellcode for use in step 4
4. Set the address of the LocateHandleBuffer function within the EFI_BOOT_SERVICES table to point to the address of shellcode (overwrite function pointer of LocateHandleBuffer to redirect code flow)
5. Trigger the SW SMI using the identified SW SMI number identified in step 2.
    A. Set up communication buffer
    B. SmmCommunicate()
    C. Write to I/O ports 0xb2 and 0xb3
6. Attacker shellcode is executed in ring -2

SMM Callout v. 2
Chipsec? Never heard of her.

```
FS0:\> load SmmCalloutDriver.efi
Locate Handle Buffer address: 000000007EED0110
Locate Handle Buffer offset: 000000007EED0108
EFI SYSTEM TABLE pointer address: 7E5EC018
EFI BOOT SERVICES TABLE pointer  address is: 7EEF6F60

EFI_LOADED_IMAGE_PROTOCOL pointer  address is: 7EED0118

Found Runtime Data address range in memory map: 000000007E4ED000 - 000000007E5ED000 of size 0000000000100000
Found Runtime Code address range in memory map: 000000007E5ED000 - 000000007E6ED000 of size 0000000000100000
potential smmc found at: 7E6CB140
potential smmc found at: 7E6CB140
Testing smmc_loc value, found at: 7E6CB140
 Vulnerable gBS function pointer LocateHandleBuffer is at: 7EEF7098
 Testing ... confirming gBS function pointer LocateHandleBuffer is at: 7EEF7098
 Vulnerable gBS LocateHandleBuffer function handler is at: 7EED70AF
Size of shellcode 0000000000000091
shellcode address: 7EED0143
alt shellcode address: 000000007EED00E0
SMM Base2 protocol is located at 7E6CB0E0
SMM communication protocol is located at 7E6CB400
```

UEFI Exploit dev: SMM Callouts

Mitigations: SMM_CODE_CHK_EN

- **Requires building a ROP chain and calculating SMBASE**
- Run ropper on your target UEFI driver, find some gadgets, build your exploit
- A few resources on bypassing SMM_CODE_CHK_EN with ROP:
  - Binarly: “The Dark Side of UEFI: A technical Deep-Dive into Cross-Silicon Exploitation” https://www.binarly.io/blog/the-dark-side-of-uefi-a-technical-deep-dive-into-cross-silicon-exploitation
  - Syntactiv: “Code Checkmate in SMM” by Bruno Pujos: https://www.synacktiv.com/en/publications/code-checkmate-in-smm
  - cr4sh: “Exploiting AMI Aptio firmware on example of Intel NUC” http://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html
  - Many more examples

UEFI Exploit dev: GOP Complex

UEFI Exploit dev: GOP Complex (REcon 2024)

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.

“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

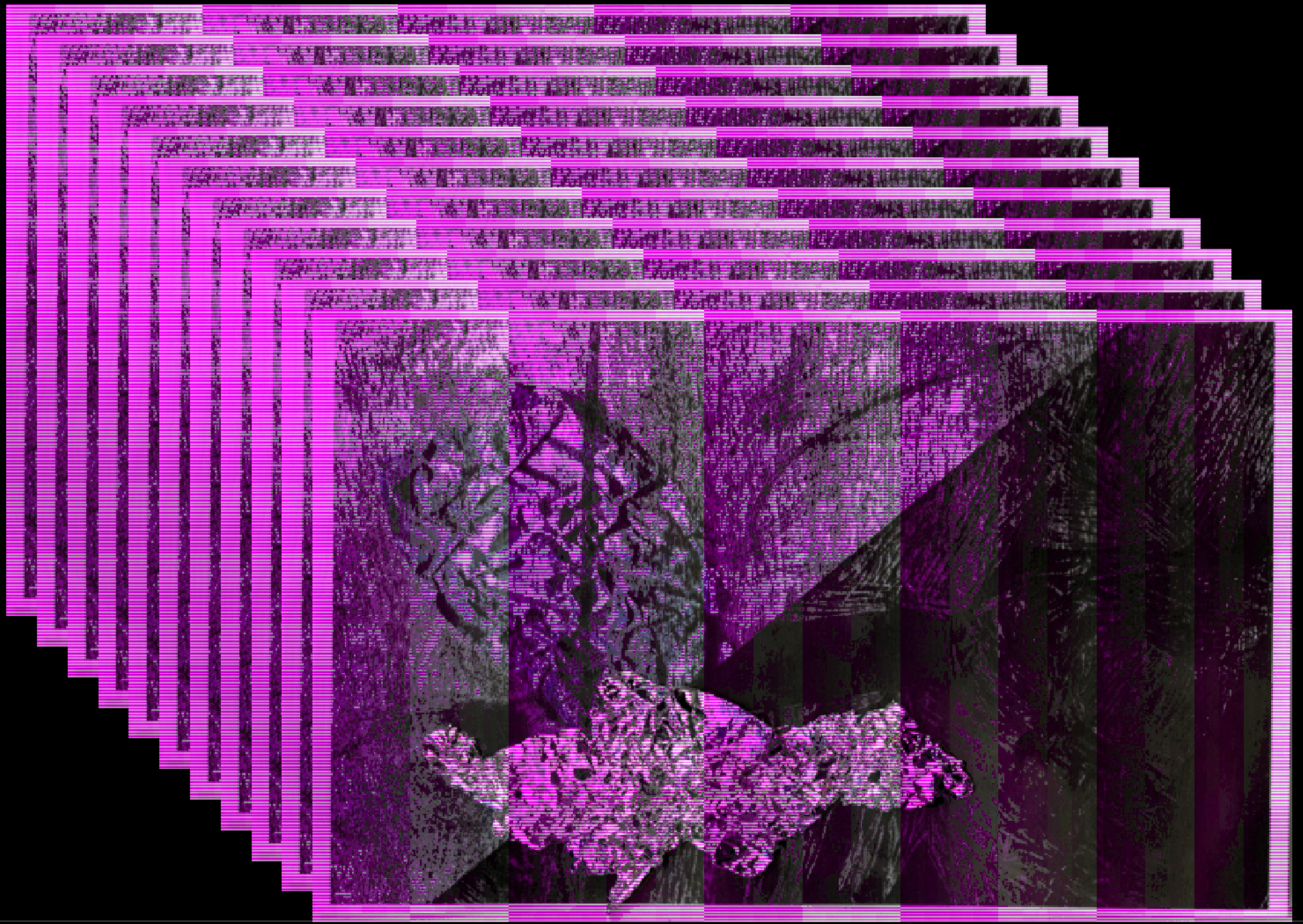
The antidote to corporate bureaucracy crushing creativity in xdev, malware dev and vx:

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



UEFI Exploit Dev is amazing and here's why

I love UEFI it's my favorite

- Exploit mitigations that are common on modern OSes (i.e. ASLR, DEP, stack canaries, etc.) aren't always implemented or implemented fully on UEFI BIOS firmwares
- If binary exploit mitigations are applied, bypass techniques aren't unfamiliar (i.e. ROP/JOP chains for bypassing SMM Code_Check_En)
- UEFI is a complex ecosystem -> error-prone and incomplete coverage of applied protections
- UEFI is so expansive and unexplored that it offers an environment for creativity in research and exploit development
- Firmware + hardware + low-level exploit dev + cross-architecture exploits == <3

Conclusion

The art of binary golfing unlocks new techniques in UEFI xdev

- 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) for GOP Complex (REcon 2024)
- Binary golfing my solutions for the same UEFI quine across 3 different architectures led to new breakthroughs in my work and the development of novel techniques for UEFI xdev

“Just as fragments of a vessel, in order to be fitted together, must correspond to each other in the tiniest details but need not resemble each other, so translation, instead of making itself resemble the meaning of the original, must lovingly, and in detail, fashion in its own language a counterpart to the original's mode of intention, in order to make both of them recognizable as fragments of a vessel, as fragments of a greater language.”

Walter Benjamin, “The Task of the Translator,” translated by Steven Rendall, page 161.

Q & A

UEFI Exploitation/Research Resources

“Low Level PC/Server Attack & Defense Timeline,” By @XenoKovah of @DarkMentorLLC https://darkmentor.com/timeline.html

“Debugging System with DCI and Windbg,” Satoshi Tanda, 29 March 2021, https://standa-note.blogspot.com/2021/03/debugging-system-with-dci-and-windbg.html

"How Many Million BIOSes would You Like to Infect?" Xeno Kovah & Corey Kallenberg, LegbaCore, http://legbacore.com/Research_files/HowManyMillionBIOSWouldYouLikeToInfect_Full2.pdf

“Leaked Intel Boot Guard keys: What happened? How does it affect the software supply chain?” Binarly Team, Binarly, 9 November 2022, https://www.binarly.io/blog/leaked-intel-boot-guard-keys-what-happened-how-does-it-affect-the-software-supply-chain

“Breaking through another Side: Bypassing Firmware Security Boundaries,” Alex Matrosov, Binarly, 14 July 2021, https://www.binarly.io/blog/breaking-through-another-sidebypassing-firmware-security-boundaries

UEFI Exploitation/Research Resources

“Now You See It... TOCTOU Attacks Against BootGuard,” Peter Bosch & Trammell Hudson, HackInTheBox Conference 2019, https://conference.hitb.org/hitbsecconf2019ams/materials/D1T1%20-%20Toctou%20Attacks%20Against%20Secure%20Boot%20-%20Trammell%20Hudson%20&%20Peter%20Bosch.pdf

“Who Watches BIOS Watchers?” Alex Matrosov, Binarly, 12 July 2021, https://www.binarly.io/blog/who-watches-bios-watchers

“Firmware is the new Black — Analyzing Past 3 years of BIOS/UEFI Security Vulnerabilities” Bruce Monroe & Rodrigo Rubira Branco & Vincent Zimmer, BlackHat USA 2017, https://github.com/rrbranco/BlackHat2017/blob/master/BlackHat2017-BlackBIOS-v0.13-Published.pdf

“The Keys to the Kingdom and the Intel Boot Process,” Eclypsium Blog, 28 June 2023, Eclypsium, https://eclypsium.com/blog/the-keys-to-the-kingdom-and-the-intel-boot-process/

“BootGuard,” Trammell Hudson, 8 November 2020, https://trmm.net/Bootguard/

UEFI Exploitation/Research Resources

“Safeguarding rootkits: Intel BIOS Guard,” Alexander Ermolov, Zero Nights, https://github.com/flothrone/bootguard/blob/master/Intel%20BootGuard%20final.pdf

“Securing the Boot Process: The hardware root of trust,” Jessie Frazelle, 2019 https://dl.acm.org/doi/fullHtml/10.1145/3380774.3382016

“CPUMicrocodes: Intel, AMD, VIA & Freescale CPU Microcode Repositories,” platomav, GitHub https://github.com/platomav/CPUMicrocodes

“Breaking Firmware Trust from Pre-EFI: Exploiting Early Boot Phases,” Binarly, BlackHat USA 2022, https://www.youtube.com/watch?v=Z81s7UIiwmI

ARM UEFI Exploitation/Research Resources

“Attacking the ARM’s TrustZone,” Joffrey Gibson, QuarksLab, 31 July 2018,
https://blog.quarkslab.com/attacking-the-arms-trustzone.html

“Introduction to Trusted Execution Environment: ARM's TrustZone,” Joffrey Gibson, QuarksLab, 19 June 2018,
https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html

“The Dark Side of UEFI: A technical Deep-Dive into Cross-Silicon Exploitation
Binarly efiXplorer Team, Binarly, 8 February 2024,
https://www.binarly.io/blog/the-dark-side-of-uefi-a-technical-deep-dive-into-cross-silicon-exploitation

“Multiple Vulnerabilities in Qualcomm and Lenovo ARM-based Devices,”
Binarly Team, Binarly, 9 January 2023,
https://www.binarly.io/blog/multiple-vulnerabilities-in-qualcomm-and-lenovo-arm-based-devices

UEFI Exploitation/Research Resources

"Moving From Common Sense Knowledge about UEFI To Actually Dumping UEFI Firmware," Assaf Carlsbad, Sentinel One, https://www.sentinelone.com/labs/moving-from-common-sense-knowledge-about-uefi-to-actually-dumping-uefi-firmware/

"Moving From Manual Reverse Engineering of UEFI Modules To Dynamic Emulation of UEFI Firmware," Assaf Carlsbad, Sentinel One, https://www.sentinelone.com/labs/moving-from-manual-reverse-engineering-of-uefi-modules-to-dynamic-emulation-of-uefi-firmware/

"Moving From Dynamic Emulation of UEFI Modules To Coverage-Guided Fuzzing of UEFI Firmware" Assaf Carlsbad, Sentinel One, https://www.sentinelone.com/labs/moving-from-dynamic-emulation-of-uefi-modules-to-coverage-guided-fuzzing-of-uefi-firmware/

"Adventures From UEFI Land: the Hunt For the S3 Boot Script," Assaf Carlsbad, Sentinel One, https://www.sentinelone.com/labs/adventures-from-uefi-land-the-hunt-for-the-s3-boot-script/

SMM Callout resources

“Exploiting SMM callout vulnerabilities in Lenovo firmware” by cr4sh

“Building reliable SMM backdoor for UEFI based platforms” by cr4sh, http://blog.cr4.sh/2015/07/building-reliable-smm-backdoor-for-uefi.html

"Code Check(mate) in SMM." Bruno Pujos, January 14, 2020, Synactiv, https://www.synacktiv.com/en/publications/code-checkmate-in-smm.html

"Through the SMM Class and a Vulnerability Found There." Bruno Pujos, January 14, 2020, Synactiv, https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there.html

"Another Brick in the Wall: Uncovering SMM Vulnerabilities in HP Firmware," Assaf Carlsbad, Sentinel One, https://www.sentinelone.com/labs/another-brick-in-the-wall-uncovering-smm-vulnerabilities-in-hp-firmware/

SMM Callout resources

“SmmExploit,” tandasat, GitHub,
https://github.com/tandasat/SmmExploit

“SmmExploit - FindSystemManagementServiceTable” tandasat, GitHub,
https://github.com/tandasat/SmmExploit/blob/main/Demo/Demo/FindSystemManagementServiceTable.cpp

“PiSmmCore: SMM Core global variable for SMM System Table (SMST) Only accessed as a physical structure in SMRAM,” tianocore, edk2, GitHub,
https://github.com/tianocore/edk2/blob/stable/202011/MdeModulePkg/Core/PiSmmCore/PiSmmCore.c#L19

“MdeModulePkg: PiSmmIpl,” tianocore, edk2, GitHub,
https://github.com/tianocore/edk2/blob/stable/202011/MdeModulePkg/Core/PiSmmCore/PiSmmIpl.c

“Platform Runtime Mechanism,” version 1.0, UEFI, November 2020,
https://uefi.org/sites/default/files/resources/Platform%20Runtime%20Mechanism%20-%20with%20legal%20notice.pdf

“Platform Runtime Mechanism,” tianocore, edk2-staging repository, GitHub,
https://github.com/tianocore/edk2-staging/tree/PlatformRuntimeMechanism

SMM Callout resources

"Advanced x86: BIOS and System Management Mode Internals, Day 7, System Management Mode (SMM)," Xeno Kovah & Corey Kallenberg, LegbaCore, https://opensecuritytraining.info/IntroBIOS_files/Day1_07_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20SMM.pdf

"Advanced x86: BIOS and System Management Mode Internals , Day 8, SMRAM (System Management RAM)," Xeno Kovah & Corey Kallenberg, LegbaCore, https://opensecuritytraining.info/IntroBIOS_files/Day1_08_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20SMRAM.pdf

"Advanced x86: BIOS and System Management Mode Internals , Day 8, SMRAM (System Management RAM)," Xeno Kovah & Corey Kallenberg, LegbaCore, https://opensecuritytraining.info/IntroBIOS_files/Day1_09_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20SMM%20and%20Caching.pdf

"Advanced x86: BIOS and System Management Mode Internals, Day 10, More Fun with SMM," Xeno Kovah & Corey Kallenberg, LegbaCore, https://opensecuritytraining.info/IntroBIOS_files/Day1_10_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20Other%20Fun%20with%20SMM.pdf

"Advanced x86: BIOS and System Management Mode Internals, Day 11, SMM Conclusion," Xeno Kovah & Corey Kallenberg, LegbaCore, https://opensecuritytraining.info/IntroBIOS_files/Day1_11_Advanced%20x86%20-%20BIOS%20and%20SMM%20Internals%20-%20SMM%20Conclusion.pdf

ARM64 UEFI Resources

“Arm SystemReady and the UEFI Firmware Ecosystem,” Dong Wei (Arm) Samer El-Haj-Mahmoud (Arm), UEFI 2021 Virtual Plugfest, January 26, 2021

“Arm SystemReady Compliance Program,” ARM,
https://www.arm.com/architecture/system-architectures/systemready-certification-program

“ARM Developer docs: UEFI,” ARM Developer,
https://developer.arm.com/Architectures/Unified%20Extensible%20Firmware%20Interface

“ARM Management Mode Interface Specification System Software on ARM,” ARM Developer,
https://developer.arm.com/documentation/den0060/a/?lang=en

“Base Boot Security Requirements 1.3,” ARM Developer,
https://developer.arm.com/documentation/den0107/latest

“Porting a PCI driver to ARM AArch64 platforms”, Olivier Martin (ARM), UEFI Spring Plugfest – May 18-22, 2015,
https://uefi.org/sites/default/files/resources/UEFI_Plugfest_May_2015_ARM.pdf

“Tailoring TrustZone as SMM Equivalent,” Tony C.S. Lo Senior Manager American Megatrends Inc., UEFI Plugfest March 2018,
https://uefi.org/sites/default/files/resources/UEFI_Plugfest_March_2016_AMI.pdf

EBC Resources

Writing and Debugging Writing and Debugging EBC Drivers EBC Drivers February 27 February 27th 2007, https://uefi.org/sites/default/files/resources/EBC_Driver_Presentation.pdf

“EFI Byte Code,” Vincent Zimmer, 1 August 2015, https://vzimmer.blogspot.com/2015/08/efi-byte-code.html

“Fasmg-ebc,” pbatard, GitHub, https://github.com/pbatard/fasmg-ebc/

“Ebcvm,” yabits, Github, https://github.com/yabits/ebcvm/

“Ghidra-EFI-Byte-Code-Processor,” meromwolff, GitHub, https://github.com/meromwolff/Ghidra-EFI-Byte-Code-Processor/

“EBC Compiler,” Ravi Narayanaswamy and Jiang Ning Liu, Intel, 2007, https://uefi.org/sites/default/files/resources/EBC_Compiler_Presentation.pdf