(My) Uncovering Exploitable Firmware Internship

Arjun Vedantham
2023 Summer Research Intern
TSE-EVS

Agenda

- About Me
- UEFI
  - Background
    - A brief history
    - Supply chains
  - Vulnerabilities
- HARDEN
  - Detecting vulnerabilities
  - Scaling up detection

About Me

- Originally from Allentown, PA
- Currently: University of Maryland at College Park
    - Studying computer science
        - Minor in robotics and autonomous systems
    - Planning to graduate in Dec. 2023
- UMDLoop
    - Avionics Systems Lead
        - Telemetry for a tunnel boring machine
        - Software and hardware for a Mars rover

Background on UEFI

A Brief History

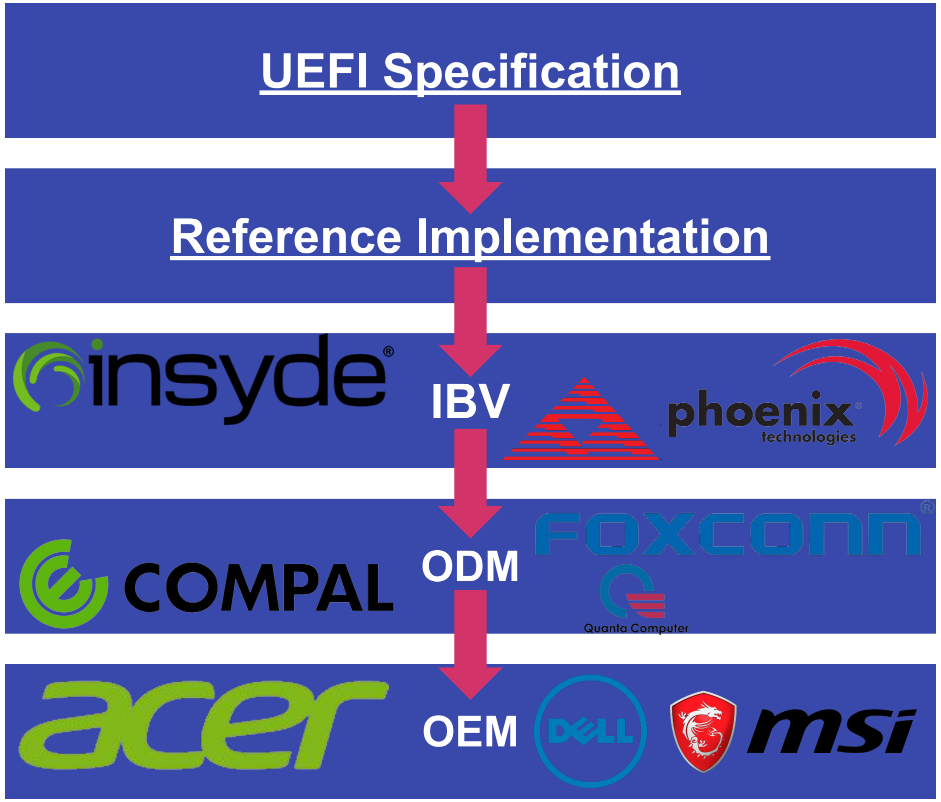
- PCs use firmware to initialize hardware/software
  - Historically: [Legacy] BIOS (c. 1981)
    - Limited functionality
      - Address-constrained
      - No support for fancy features
        - Network boot?
    - Development was difficult + highly machine specific
  - Today: UEFI (c. 2005)
    - **U**niversal **E**xtensible **F**irmware **I**nterface

UEFI

- First developed by Intel (2005)
- A common standard
  - Vendors implement independently
  - Reference implementation: Tianocore EDK II
- Not limited to just boot-time services
  - Manages runtime kernel - hardware interaction
  - Most UEFI drivers run in driver execution environment (DXE)
    - Equivalent to ring 0/kernel mode

UEFI Supply Chains

- Lots of different companies/organizations involved!
  - Bug fixes take a *long* time to reach end-users
    - Independent BIOS Vendors
    - Original Design Manufacturers
    - Original Equipment Manufacturers
- How UEFI firmware is developed/distributed is dependent on all of these organizations
  - Annoying packaging practices (Dell 😠)



Taken from Rylan’s JTB on UEFI (thanks Rylan)

UEFI Vulnerabilities

- Types
  - Double GetVar
  - GetSet
  - SMM Callouts
  - SMM CommBuffer poisoned pointers
- In general
  - Vulnerabilities are simple, but hard to find because of the supply chain

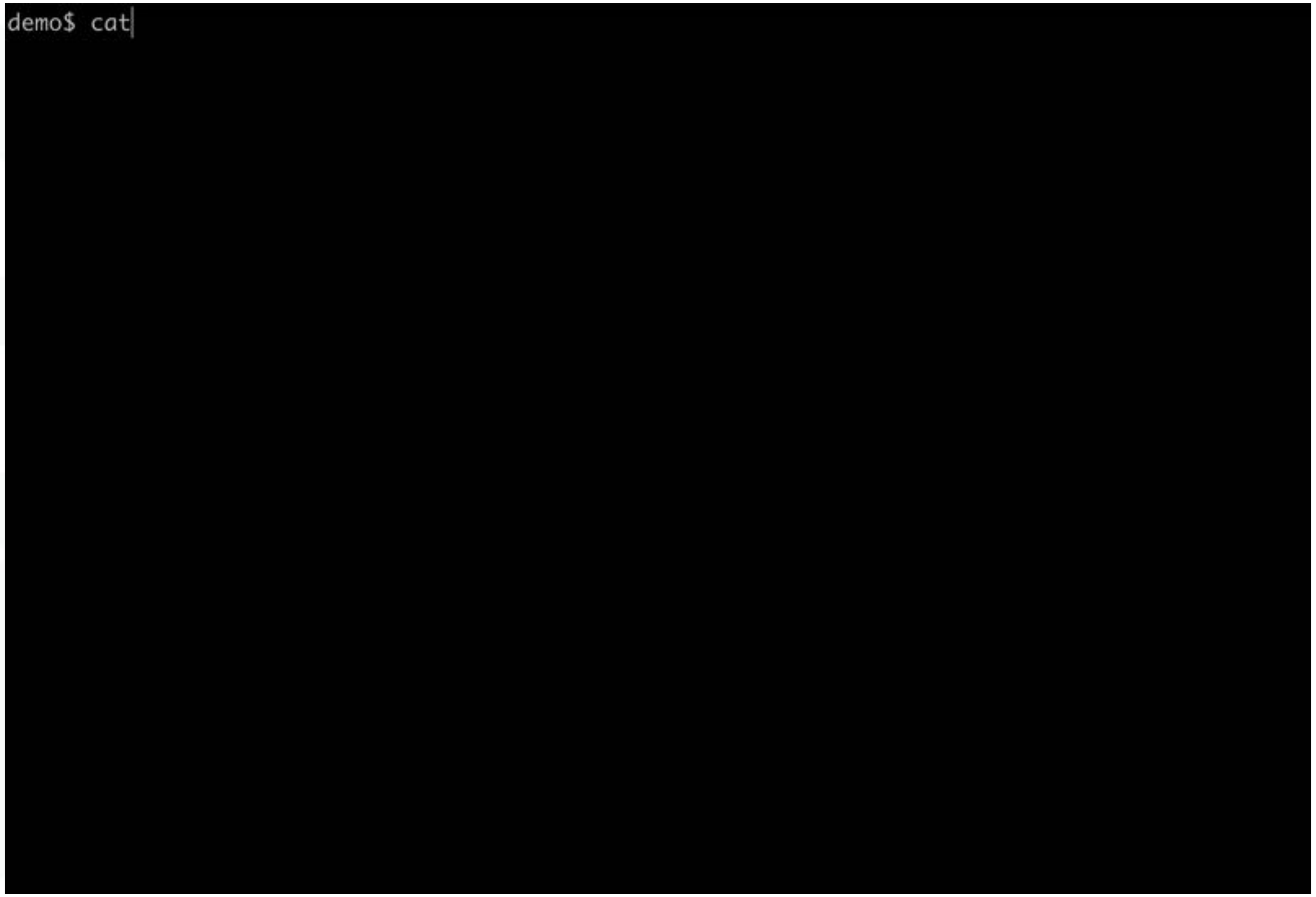
Double GetVar & GetSet

- Runtime service that gets a value from NVRAM
    - NVRAM - non-volatile RAM
        - Persistent key/value storage for UEFI variables
- Takes a *DataSize* pointer
        - Input: Size of the data buffer we are writing the variable value to
        - Output: # of bytes that the variable occupies
            - Can be longer than the buffer
                - Returns an error if it is
    - Two GetVariable calls without sanitization? Potential overflow!

GetSet

- Very similar to Double GetVar
  - SetVariable
    - Sets a value in NVRAM
  - Consecutive calls to GetVariable and SetVariable without sanitization
    - Exposes EFI variables

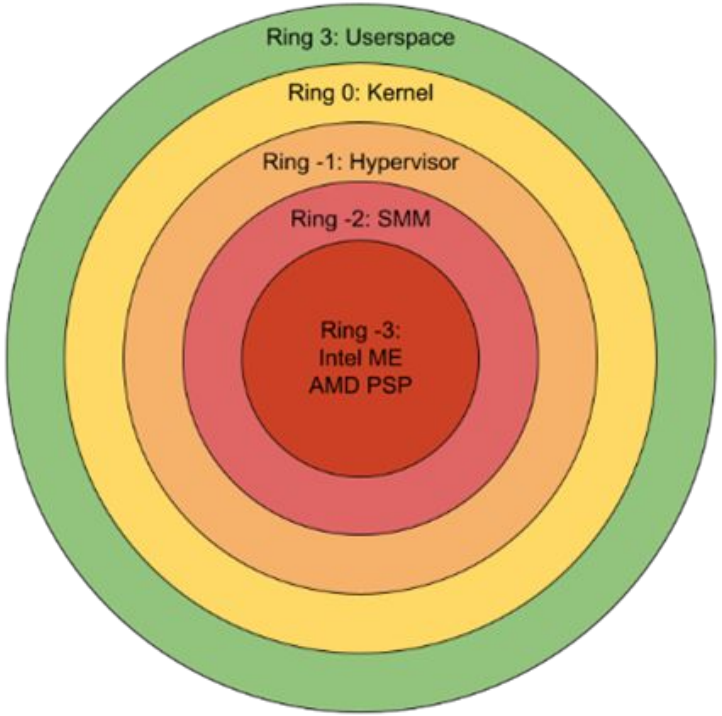
Double GetVar & GetSet (cont.)



Double GetVar in action (GIF from Binarly)

System Management Mode

- x86 processors have an execution mode called System Management Mode (SMM)
  - Runs highly privileged code
    - Can interact with all physical memory (even though it shouldn't)
      - Will run above hypervisors as well
  - Equivalent to ring -2
    - ring 0: kernel space
  - Mostly invisible to the OS
  - Invoked from the kernel, or from a hardware interrupt
- Can interact with SPI flash (and all other hardware)
  - Install rootkits that can persist *even after the OS is wiped!*



SMM vulnerabilities in UEFI

- Privilege escalation from driver execution environment (DXE) to SMM
    - SMM callouts
        - Executing code in SMM that lives outside of protected memory (SMRAM)
    - SMM CommBuffer vulns
        - CommBuffer
            - Type that handles communication between SMM and DXE
                - Copies variables into SMRAM
        - Should check that all nested pointers in a CommBuffer are pointing into SMRAM

twosix TECHNOLOGIES

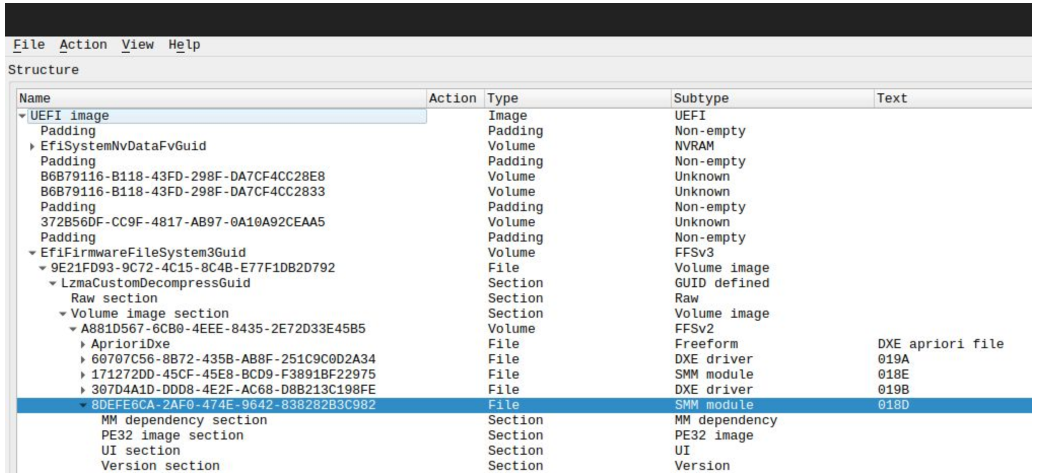
HARDEN

How do we find vulnerabilities in UEFI?

- Manual analysis (slow, requires expert knowledge)
- Fuzz testing (not scalable to an entire UEFI image)
- Alternatively: Use static analysis
    - Trace the flow of data to potentially vulnerable callsites
    - Build dataflow chains
    - Use SMT solving to see if these chains can be exploited
        - (I don’t know how to do this part)

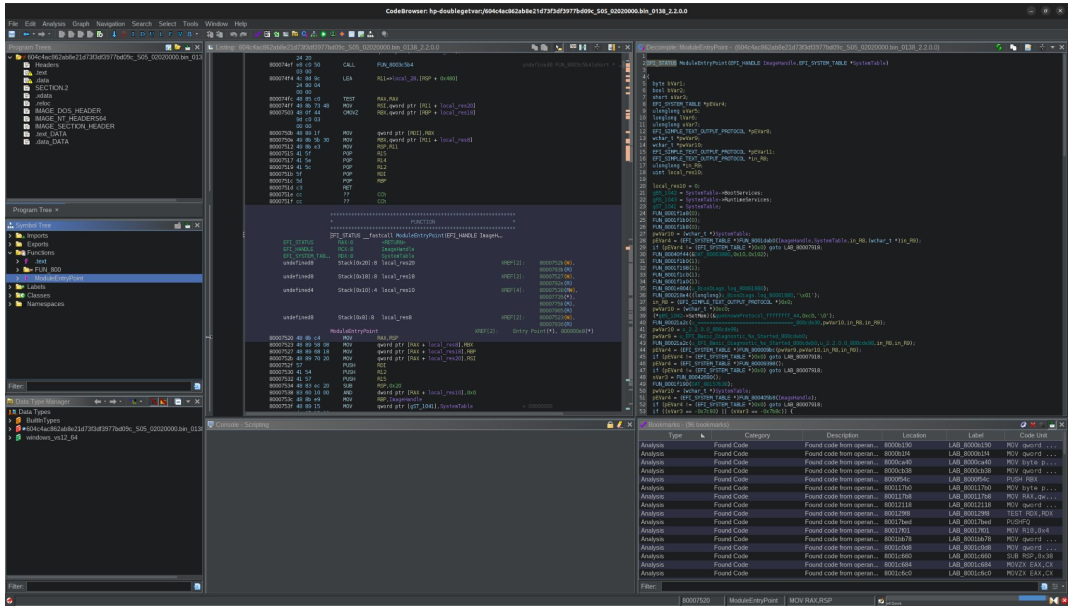
Finding SMM vulnerabilities…

- Find vulnerable UEFI drivers
  - Binarly writeups
    - Manual, expert analysis of UEFI drivers
  - CVEs
- Download the firmware from the OEM site
- Decompress it with 7-zip (unless you're Dell 😠)
- UEFITool
  - Tool that displays/extracts UEFI drivers in a firmware binary
  - Extract the vulnerable driver, as identified by its GUID



…in Ghidra?

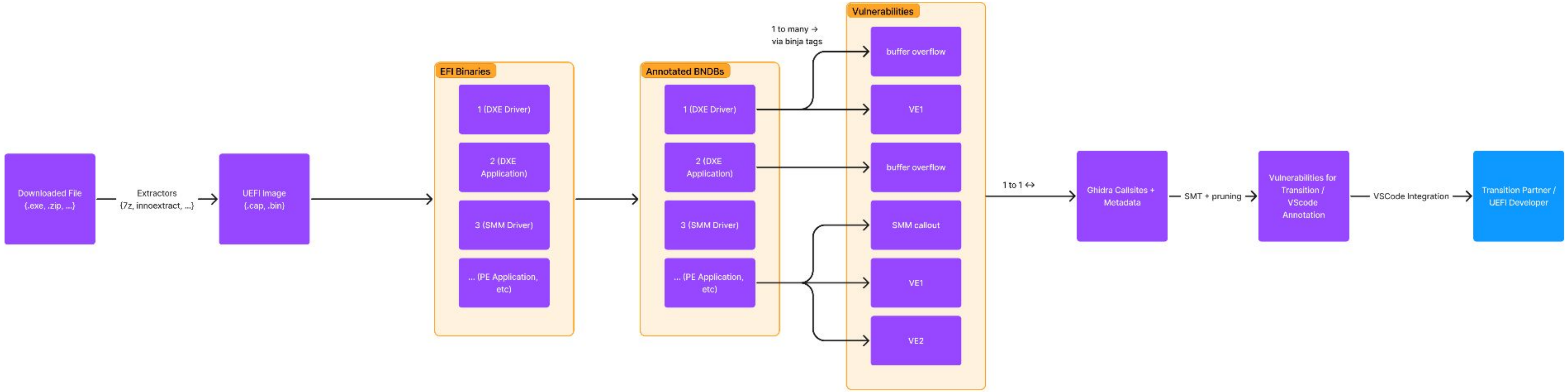
- Open the extracted binary in Ghidra
  - Run efiSeek
    - Ghidra plugin that automatically types UEFI structs
  - Look for *SMRAM descriptors*
    - Passed into the function that validates whether pointers reference SMRAM
      - Flag these callsites and trace up from them
- Problems
  - Limited intermediate languages to reason over
  - Ghidra API has poor documentation
  - Lack of existing tooling



…in Binary Ninja?

- (Arguably) better API with Python support
  - *Jython does not count, Ghidra!*
- More intermediate representations to reason over
  - Single static assignment
- More comprehensive internal tooling
  - PILOT program: already did def/use chaining for vulnerability analysis
- Create a pipeline for automated vulnerability analysis
  - Get binaries, extract them, and run analysis passes over them
  - Use SMT solving to prune the set of possible vulnerabilities

Binary Ninja-based Pipeline



Def/Use Chaining

- Find vulnerable callsites
  - Look for accesses to a particular byte offset from the runtime services table
- From a callsite:
  - Trace the definition of the parameter we want (e.g. *DataSize*)
  - Pointers
    - Trace *down*
      - We don't know how the value the pointer is referencing will change
  - Functions
    - If we know what it does, trace up

```
 8 @ 00040cce  DataSize @ mem#2 -> mem#3 = 8
 9 @ 00040cd6  rax#1, mem#4 = sub_1f6f0(rcx#1, &data_cdffa) @ mem#3
10 @ 00040cde  if (rax#1 == 0) then 11 @ 0x40d17 else 17 @ 0x40ce0

11 @ 00040d17  var_38_1#1 = &arg_18
12 @ 00040d1c  r9_1#1 = &DataSize
13 @ 00040d20  rax_2#2 = [&RuntimeServices].q @ mem#4
14 @ 00040d27  rdx#1 = &VendorGuid
15 @ 00040d32  rax_3#3, r8_1#1, r9_2#2, mem#5 = rax_2#2->GetVariable @ mem#4(VariableName: u"PlatformLang", VendorGuid: rdx#1, Attributes: 0, DataSize: r9_1#1, Data: var_38_1#
16 @ 00040d38  if (rax_3#3 s>= EFI_SUCCESS) then 24 @ 0x40d86 else 26 @ 0x40d41

17 @ 00040ce0  rax_1#8 = [&BootServices].q @ mem#4
18 @ 00040ce7  rcx_1#5 = &arg_18
19 @ 00040cf0  mem#11 = rax_1#8->SetMem @ mem#4(Buffer: rcx_1#5, Size: 8, Value: 0) @ mem#4
20 @ 00040cf6  r9#6 = rbx#1
21 @ 00040d02  rcx_2#6 = &arg_18
22 @ 00040d06  mem#12 = sub_211f4(rcx_2#6, &__dos_header.e_cparhdr, &data_122464, r9#6) @ mem#11
23 @ 00040d0b  goto 32 @ 0x40d99

24 @ 00040d86  rcx_3#2 = u"PlatformLang reported as %a.\r\n"
25 @ 00040d86  goto 46 @ 0x40d8d

26 @ 00040d41  var_38_2#2 = &arg_18
27 @ 00040d46  r9_3#3 = &DataSize
28 @ 00040d4a  rax_4#4 = [&RuntimeServices].q @ mem#5
29 @ 00040d51  rdx_1#2 = &VendorGuid
30 @ 00040d5c  rax_5#5, rdx_2#3, r8_1#2, r9_2#4, mem#6 = rax_4#4->GetVariable @ mem#5(VariableName: u"Lang", VendorGuid: rdx_1#2, Attributes: 0, DataSize: r9_3#3, Data: var_38
```

```
> 0: rax_5#5, rdx_2#3, r8_1#2, r9_2#4, mem#6 = rax_4#4->GetVariable @ mem#5(u"Lang", rdx_1#2, 0, r9_3#3, var_38_2#2) @ mem#5 (TraceDirection.UP)
> 1: r9_3#3 = &DataSize (TraceDirection.UP)
> 2: r9_1#1 = &DataSize (TraceDirection.DOWN)
> 3: rax_3#3, r8_1#1, r9_2#2, mem#5 = rax_2#2->GetVariable @ mem#4(u"PlatformLang", rdx#1, 0, r9_1#1, var_38_1#1) @ mem#4 (TraceDirection.DOWN)
```

Future Plans

- Static analysis is somewhat limited when it comes to cross-driver interaction
  - Want to evaluate the composability of different vulnerabilities
    - More formal modeling?
    - Emulation?
      - QEMU (OSS)
      - Simics (Intel)
        - More closely replicates the underlying hardware
          - Interrupts between different instructions (which QEMU can't do)
- Leveraging the new tracing engine for SMM vulnerabilities
  - Sort of implemented with Ghidra, but not well
  - Generalize to other CommBuffer vulnerabilities

References & Other Resources

- Rootkits and Bootkits
  - Alex Matrosov (Binarly)
  - Sergey Bratus (DARPA PM for HARDEN)
- A Tour Beyond the BIOS
  - Jiewen Yao (Intel)
    - Also wrote Securing Firmware (on my reading list)
- SentinelOne blog
- Rylan’s excellent JTB from March on UEFI

Acknowledgements

- Jonathan Prokos (mentor for the summer)
- Jacob Denbeaux
- Michael Krasnitski

twosix TECHNOLOGIES