Fuzzing UEFI Interfaces

Holistic Software Security

Connor Glosner

Overview

- What is UEFI?
- Problem Definition
- Motivation
- UEFI Background
- Fuzzing
  - What is needed for fuzzing?
  - How does this apply to UEFI?
- How do we address UEFI fuzzing challenges?
- Our Solution: FuzzUEr?
- Evaluation

What is UEFI?

UEFI (Unified Extensible Firmware Interface) firmware is the modern replacement for the older BIOS (Basic Input/Output System) firmware found in computers. It serves as the interface between the computer's hardware and its operating system, handling the boot process and providing a range of services to the OS before it takes control. It provides the following improvements when compared to Legacy BIOS:

- Graphical User Interface
- Support for large drives (>2.2TB)
- Secure Boot
- Modular device driver support
- Processor Independence
- Networking and remote access (during boot)

Problem: Why is UEFI important?

The Unified Extensible Firmware Interface (UEFI) runs on nearly all modern computers, holding the highest level of privilege within a system. Despite its critical role, UEFI's complexity creates numerous potential vulnerabilities that can be exploited by attackers. Its responsibility for initializing key hardware security features, such as Secure Boot, Measured Boot, and System Management Mode (SMM), makes it a prime target for threats. Given these factors, thoroughly examining UEFI's security is essential to prevent malicious actors from compromising the foundational layers of a system’s security, which could lead to widespread damage.

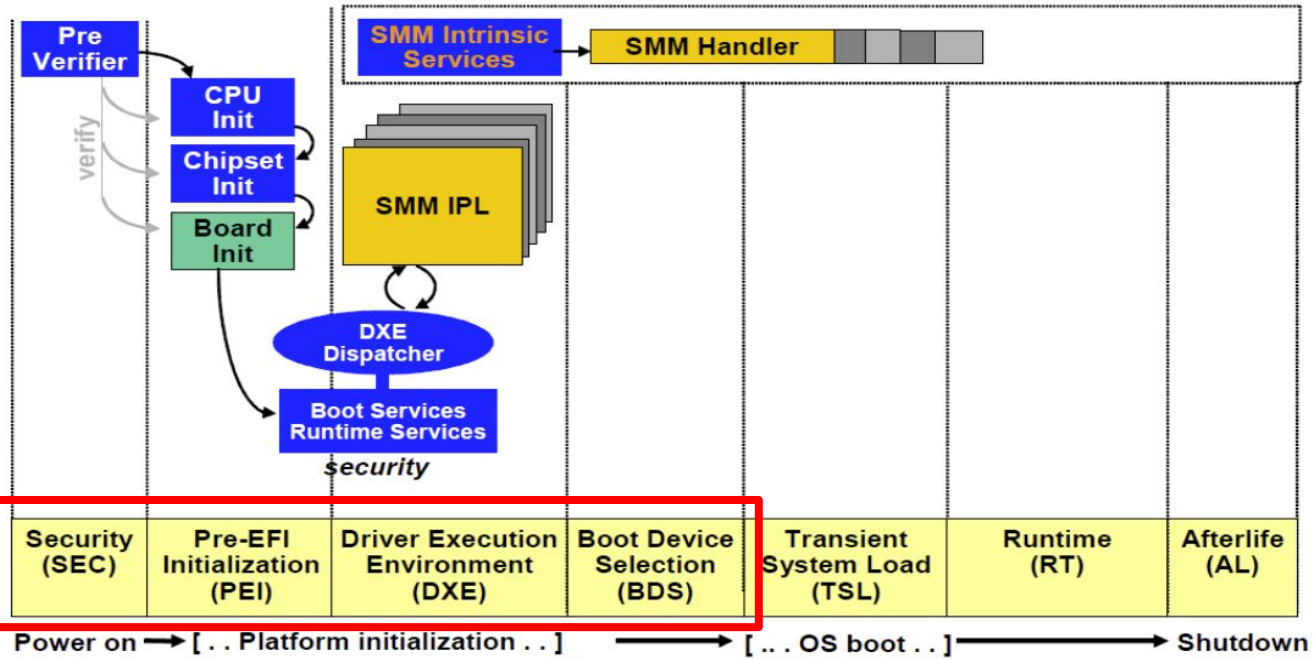
Motivation: PixieFail

- PXE(Pixie) is similar to DXE, except it is designed to allow for users to boot over the the network.
- 9 vulnerabilities discovered by Quark Labs

> We performed a cursory inspection of `NetworkPkg`, Tianocore's EDK II PXE implementation, and identified nine vulnerabilities that can be exploited by unauthenticated remote attackers on the same local network, and in some cases, by attackers on remote networks. The impact of these vulnerabilities includes denial of service, information leakage, remote code execution, DNS cache poisoning, and network session hijacking.

Background

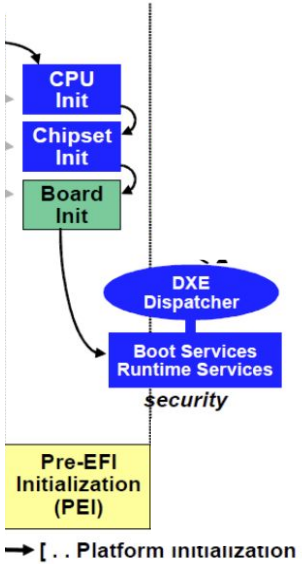
4 Phases to the UEFI boot process



Phase 1: Security (SEC)

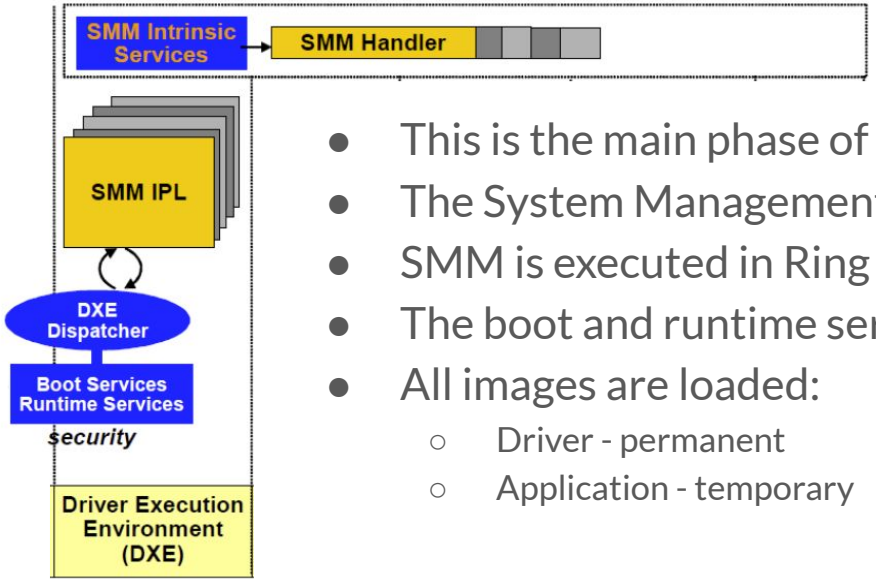
- Executes hardware specific firmware.
  - Written in assembly (16-/32-bit).
- Creates the foundation for the root-of-trust methodology.
  - Authenticates the Pre-EFI Initialization (PEI) Foundation code.
- Creates temporary memory using CPU caches.
- Locates the PEI foundation on the SPI flash.
- The SEC phase is executed on the SPI flash.
  - Address entry point is the reset vector at address space 4GB - 0x10
  - Only the bootstrap processor(BSP) is running.

Phase 2: Pre-Environment Initialization (PEI)



- The boot code is loaded from the SPI flash in this phase.
- It initializes the permanent memory, but until then everything is executed in the CPU cache.
- This is where the runtime and boot services begin execution.
- Creates hand off block (HOB) list for later phases.
- The final module is the block to load the next phase.
- The most architecture depend part of the code.

Phase 3: Driver eXecution Environment (DXE)



- This is the main phase of the boot process.
- The System Management Mode (SMM) is initialized during this phase.
- SMM is executed in Ring -2, while everything else is in Ring 0.
- The boot and runtime services finish initialization during this phase.
- All images are loaded:
  - Driver - permanent
  - Application - temporary

What is a DXE driver?

- DXE drivers are responsible for a majority of what takes place during the boot process.
- There are 2 categories of drivers:
  - Device Drivers - Handle any external devices (like controllers or USB devices).
  - Service Drivers - These are core services that are essentially system calls.
    - Boot Services
    - Runtime Services

Device Driver

- Device drivers are responsible initializing and communicating with specific hardware peripherals.
  - USB controllers
  - SCSI controllers
- They provide an interface to communicate with these devices through UEFI protocols.
- They get started during the enumeration process.

Boot Services

- Boot services are used to create, manage, and stop events during the boot process:
  - Protocol services
  - Device Protocols - how to communicate between different peripherals
  - Device handle-based boot services
  - Global boot service interface
- These services are important for communicating between drivers.
- CopyMem, which is used when copying the drivers into permanent memory or into the SMRAM is a common example.
- Primarily needed for setting everything up for the OS loader.
- They are terminated when ExitBootService() is called in the OS Loader.

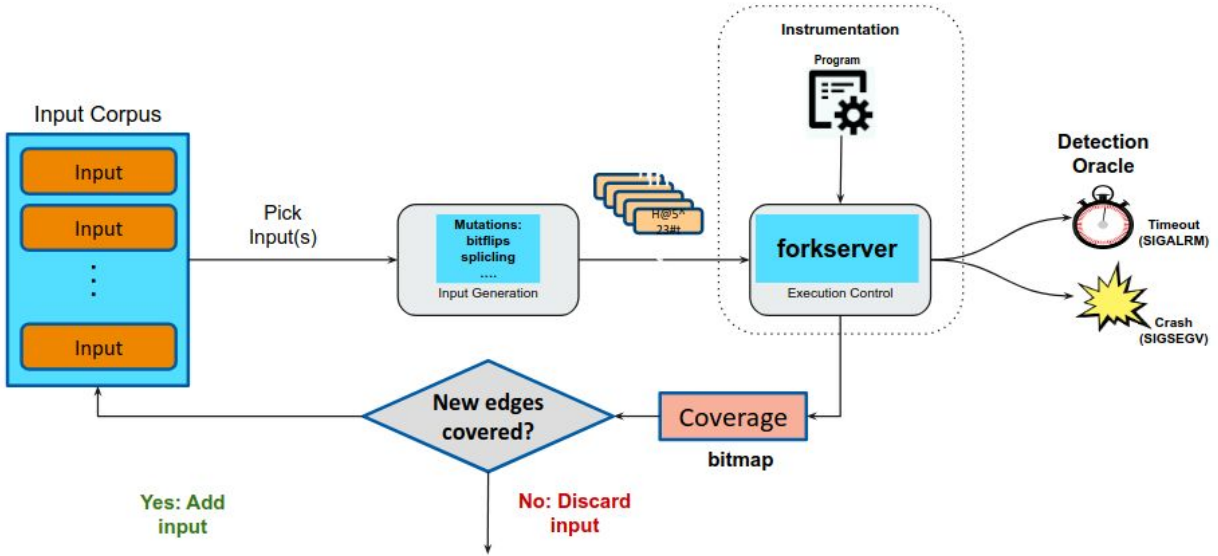
Runtime Services

- These are system call functions that create some abstraction between the kernel and the hardware.
- The memory where the runtime services are stored can’t be modified by the kernel because they interact with the hardware.
- Part of the Runtime code is stored in the SMRAM, the part pertaining to the direct hardware modification.
- The function SmmLoadImage is used to load images into SMRAM.

Phase 4: Boot Device Selection (BDS)

- This is when the boot partition is selected.
- It is either defaulted to the active partition or will allow an option if there are multiple operating systems present.
- It will also handle executing the boot manager and OS drivers from the system partition.
- The boot manager utilizes the DXE drivers that were created to complete its tasks.
- The OS loader is stored on the EFI system partition.

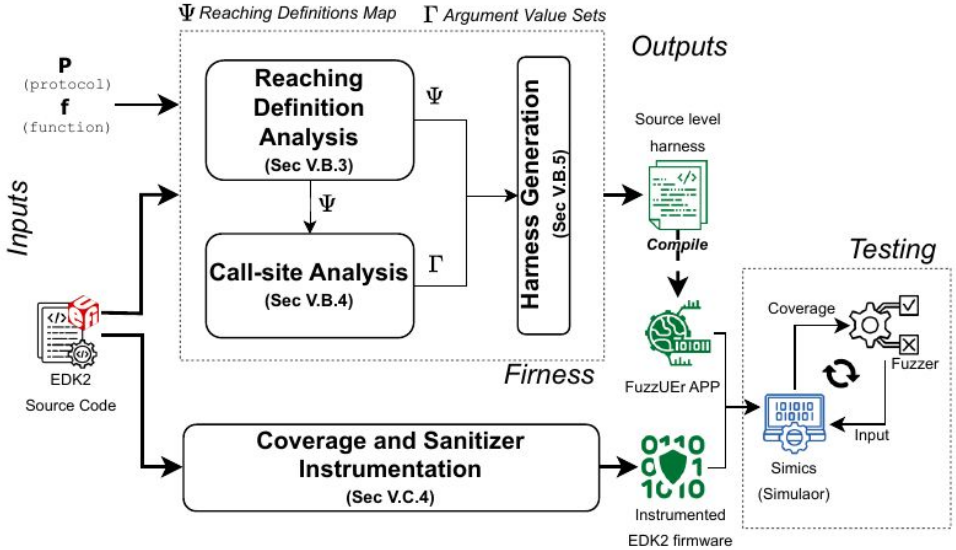
Fuzzing: High Level Idea (LibAFL/AFL++)



Input Generation

- How can we generate and feed input into UEFI drivers?
  - Random data to interrupt handlers?
  - Place random data in memory?
- Utilize UEFI internal protocol lookup (Firness):
  - Analyze the source code - Type Identification
  - Generate type aware harness automatically to mutate random input

Input Generation (Firness)



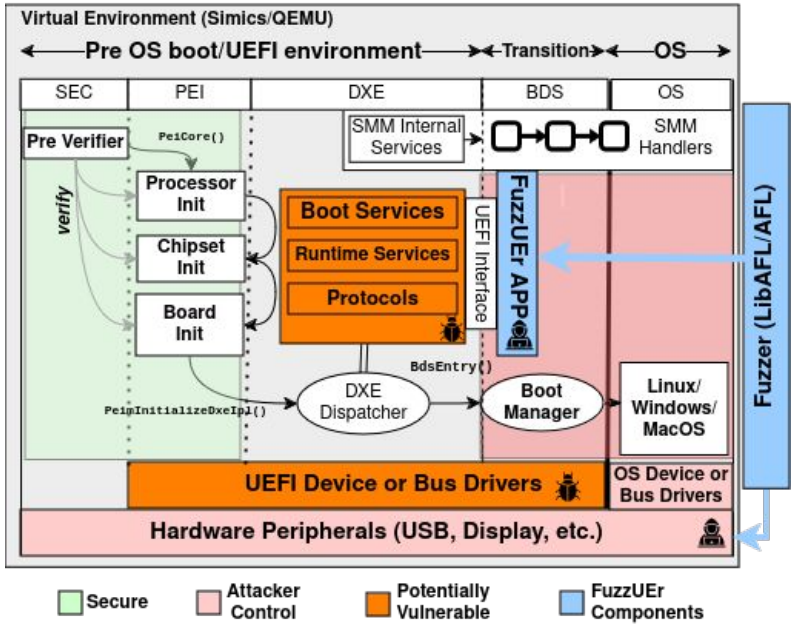
Fast Execution (Execution Environment)

- How can a hardware dependent software be executed properly?
- What environment can we use to snapshot and pass input into the firmware?
- We use a high fidelity simulator called Simics
    - QEMU can also be used

Crash/Error Detection

- Sanitizers (ASan, UBSan, etc.)
  - How can ASan work in an isolated environment?
    - Port ASan into UEFI environment
- CPU Exceptions
  - Page Fault
  - General Protection Fault
  - Invalid Opcode
  - Stack Overflow

Our Solution: FuzzUEr



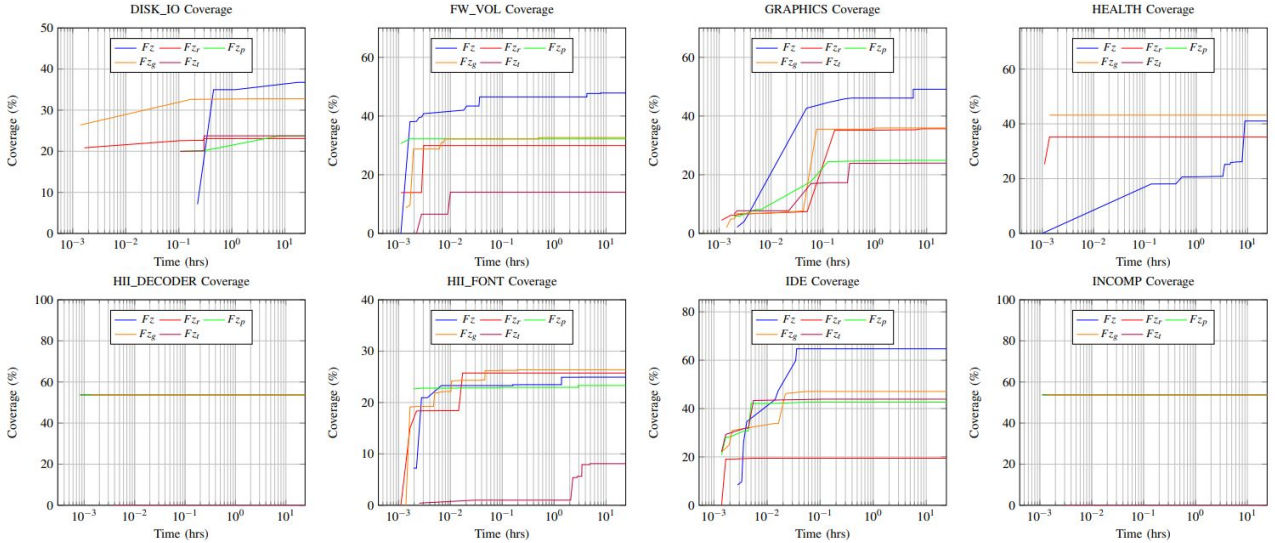
Evaluation: Bugs (Crash Detection)

ID	Protocol	Function	Bug Type	Status	Found By				
					Fz_r	Fz_g	Fz_l	Fz_p	Fz
Previously Known Bugs									
1	IP4	Ip4PreProcessPacket	Buffer Overflow	Previously Known	✗	✗	✗	✗	✗
2	HII_FONT	UefiFileHandleLib	Buffer Overflow	Previously Known	✗	✗	✓	✓	✓
3	HII_FONT	DevPathToTextUsbWWID	Buffer Overflow	Previously Known	✗	✗	✓	✓	✓
New Bugs									
4	DISK_IO	DiskIoCreateSubtaskList	Buffer Overflow	CONFIRMED	✓	✓	✓	✓	✓
5	ALL*	CR()	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
6	PRINT2S	ShellFileHandleReadLine	Buffer Overflow	CONFIRMED	✓	✓	✓	✗	✓
7	PRINT2S	ShellFindFilePathEx	Use After Free	CONFIRMED	✓	✓	✓	✗	✓
8	PRINT2S	InternalIsOnCheckList	Arbitrary Pointer Write	REPORTED	✓	✓	✓	✗	✓
9	UNICODE	EngStriColl	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
10	UNICODE	EngStrLwr	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
11	UNICODE	EngStrUpr	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
12	UNICODE	EngMetaiMatch	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
13	UNICODE	EngFatToStr	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
14	UNICODE	EngStrToFat	Null Pointer Dereference	CONFIRMED	✓	✓	✓	✓	✓
15	USB_IO	UsbIoControlTransfer	Use After Free	CONFIRMED	✗	✓	✗	✓	✓
16	S3_SMM	InternalSmBusExec	Null Pointer Dereference	REPORTED	✗	✗	✓	✗	✓
17	MANAGED_NET	EfiDhcp6InfoRequest	Arbitrary Pointer Read	REPORTED	✗	✓	✓	✓	✓
18	HII_FONT	HiiStringToImage	Arbitrary Memory Write	REPORTED	✗	✓	✓	✗	✓
19	GRAPHICS	FrameBufferBltLibVideoToBltBuffer	Buffer Overflow	REPORTED	✗	✓	✗	✓	✓
20	FW_VOL	FwVolBlockReadBlock	NULL Pointer Dereference	REPORTED	✗	✗	✓	✗	✓
21	FW_VOL	FwVolBlockReadBlock	NULL Pointer Dereference	REPORTED	✗	✗	✓	✗	✓
22	USB2_HC	EhcAsyncInterruptTransfer	NULL Pointer Dereference	REPORTED	✗	✓	✓	✗	✓
23	USB2_HC	EhcAsyncInterruptTransfer	NULL Pointer Dereference	REPORTED	✗	✓	✓	✗	✓

Evaluation: Bug Example

```
EFI_STATUS
EFIAPI
HiiStringToImage (
    ...
   IN OUT EFI_IMAGE_OUTPUT **Blt,
   IN UINTN BltX,
   IN UINTN BltY,
   ...
)
{
     ...
    Image = *Blt;
    BufferPtr = Image->Image.Bitmap + Image->Width * BltY + BltX;  /* Controlled with user input */
    ...
    GlyphToImage (..., &BufferPtr );  /* Arbitrary memory write */
    ...
}
```

Evaluation: Coverage



Evaluation: Type Identification (Input Generation)

Categories	Protocols	Number of Functions	Functions with at least 1 `void*` type	Number of Parameters			
				Min	Max	Mean	Median
USB	USB_IO	13	7	1	7	4.2	4
	USB2_HC	13	6	2	12	6.6	4
TEXT	PRINT2S	10	0	3	5	3.8	4
	HII_FONT	4	2	5	12	8	7.5
	UNICODE	6	0	2	4	3	3
	JSON	4	0	2	5	3.5	3.5
	GRAPHICS	3	0	2	10	5.3	4
CONTROLLER	NVME	4	1	2	4	3	3
	DISK_IO	2	2	5	5	5	5
	IDE	6	0	3	4	3.8	4
	SD_MMC	5	1	2	4	2.8	3
	INCOMP	1	1	7	7	7	7
	PCI_ROOT	14	8	1	7	4	4
SMM	S3_SMM	4	3	2	5	3.75	4
	SMM_BASE2	2	0	2	2	2	2
	SMM_COMM	1	1	3	3	3	3
	SMM_CONT	2	0	2	5	3.5	3.5
Driver Helper	FW_VOL	7	0	1	5	2.9	2
	HEALTH	2	2	4	6	5	5
	HII_DECODER	3	2	3	5	4	4
Network	TCP4	10	0	1	6	2.6	2
	IP4	4	4	3	4	3.5	3.5
	IP6	9	0	1	6	3.2	2.5
	SIMPLE_NET	13	4	1	7	3.6	3
	MANAGED_NET	8	0	1	4	2.4	2
Cumulative		150	44	1	12	4	3.5