A Practical Analysis of UEFI Threats Against Windows 11

Joshua Machauer

December 25, 2022

Version: Draft 1.0

Technische Universität Berlin

Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Security in Telecommunications (SecT)

Bachelor's Thesis

A Practical Analysis of UEFI Threats Against Windows 11

Joshua Machauer

1. Reviewer Prof. Dr. Jean-Pierre Seifert
Electrical Engineering and Computer Science
Technische Universität Berlin

2. Reviewer Prof. Dr. Stefan Schmid
Electrical Engineering and Computer Science
Technische Universität Berlin

Supervisors Hans Niklas Jacob and Christian Werling

December 25, 2022

Joshua Machauer
A Practical Analysis of UEFI Threats Against Windows 11
Bachelor's Thesis, December 25, 2022
Reviewers: Prof. Dr. Jean-Pierre Seifert and Prof. Dr. Stefan Schmid
Supervisors: Hans Niklas Jacob and Christian Werling

Technische Universität Berlin
Security in Telecommunications (SecT)
Institute of Software Engineering and Theoretical Computer Science
Electrical Engineering and Computer Science
Ernst-Reuter-Platz 7
10587 Berlin

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, den 25. Dezember 2022

Joshua Machauer

Abstract

In Computer Security firmware attacks are one of the most feared security threats, executing during the boot process, they can already have full control over the system before an operating system and accompanying antivirus programs are even loaded. With widespread adaption of standardized UEFI firmware these threats have become less machine dependent, and able to target a host of systems at once. Their appearances in the wild are rare as they are stealthy by nature. We categorize past analyses of UEFI threats (against Windows) by their attack vector and perform our own. With a deep-dive into the UEFI environment we learn hands on about encountered security mechanisms targeting pre-boot attacks, setting our focus on Secure Boot and TPM-assisted BitLocker. We were able to achieve system level privileged execution on Windows 11 by exploiting unrestricted hard drive access to deploy our payload and modify the Windows Registry. With BitLocker enabled, our *BitLogger* was able to decrypt and mount the drive using a keylogged Recovery Key, or when part of the chain of trust using a VMK sniffed from TPM communication. UEFI threats are very powerful and discredit all system integrity, making it impossible to put any further trust into the system.

Zusammenfassung

Hello, here is some text without a meaning. This text should show what a printed text will look like at this place. If you read this text, you will get no information. Really? Is there no information? Is there a difference between this text and some nonsense like “Huardest gefburn”? Kjift – not at all! A blind text like this gives you information about the selected font, how the letters are written and an impression of the look. This text should contain all letters of the alphabet and it should be written in of the original language. There is no need for special content, but the length of words should match the language.

Acknowledgement

Contents

III

1 Introduction

Regardless of the Operating System (OS) installed on a computer, the Platform Firmware (PF) is the very first piece of code that is executed, when powering on the system. It performs basic initialization of all platform components and manages the system's resources until the OS takes over. The PF allows for basic system and boot configuration and serves as a means to create a uniform environment for boot applications. After discovering and selecting the appropriate boot device it hands over execution to the bootloader and with that subsequently control over the entire system.

This process has been previously performed by the Basic Input/Output System (BIOS) and while the term supersedes the underlying implementation, modern systems are mostly UEFI-based, with even backwards compatibility being phased out by Original Equipment Manufacturers (OEMs). Legacy BIOS is dead and UEFI is here to stay.

UEFI as the successor addresses the previous limitations, modernizing the boot process in a standardized fashion. Allowing for to fully abstract the underlying hardware, while providing mechanisms for extensibility in the short and long term. UEFI provides a family of specifications with the UEFI specification itself defining the environment and interfaces for boot applications to communicate with the firmware. The UEFI PI specification defines a model for system designers to follow when implementing the PF. Its design focuses on simplifying the collaboration process between different hardware vendors, whose components come together in a system. Dynamic module discovery, disptach and intermodule communication allow for the independent development of early boot components. A well-defined storage solution brings it all together in a single firmware image.

The extensibility and abstractions can become a double-edged sword, when threat actors are introduced to the system, as the same concepts apply to software with malicious intent. They can gain control over the system before the OS or even its bootloader are executed. The early execution environment can be abused to achieve persistence across reboots and with access to the firmware iamge even across OS installations. They often deploy additional malware further up into the OS environment. Leveraging their privileges, they remain hidden to the OS and its antivirus software. Through abstractions of the underlying hardware UEFI based attacks can become platform independent and target a host of platforms at once.

UEFI threats fall under the category of rootkits. Rootkits are traditonally a collection of software designed to grant a threat actor control over a system. There are different types of rootkits depending on their entry vector and where they reside. They range from User Mode, Kernel Mode over Hypervisors to bootloaders and firmware [1–3]. UEFI threats are of two categories, bootloader rootkits (Bootkits) or firmware rootkits. While bootkits rely on intervening the process when the PF hands over execution to boot applications, firmware rootkits reside within the firmware image and are executed as part of the PI. Within this thesis we will reserve the alone standing term *rootkit* for UEFI firmware rootkits and mostly refer to bootloader rootkits with bootkit.

Discoveries of UEFI threats are still rather rare occurances, which may depend more on the fact that they are hard to detect than the amount of them that exists. Despite this, recent years have lead to more and more being found and analysed by security researchers. Bootkits such as FinSpy, ESPecter, Dreamboot or rootkits like Mosaicregressor, LoJax, Moonbounce and CosmicStrand. Given the small sample size the infection method is often unknown and general information very limited. Their attack approaches can be categorized into memory-and storage-based approaches. Memory-based attacks modify boot applications when they are loaded in memory. This allows them to propagate malicious code execution into the OS kernel. Storage-based attacks do not rely on the OS boot process and instead modify the OS's hard drive contents *at rest*.

By performing our own UEFI attacks we look at some infection methods for root-and bootkits, mainly involving physical access. We want to analyze Windows security mechanisms and standard security policies, whether they protect the system and if so, how. Focusing on storage-based attacks comparable to LoJax and Mosaicregressor, as Windows offers BDE with TPM 2.0 protection to prevent unauthorized access of the hard drive. TPM measurements provide a mechanism to verify system integrity and should be able to pick up malicious code execution such as rootkits and deny hard drive access and stop the system from booting.

Structure

We start off in Chapter 2 by introducing all necessary knowledge about the UEFI environment, defined through the family of UEFI specifications, listing the interface and its implementation. We briefly go over relevant security concepts of the TPM and its interaction with the PF in Chapter 3. This allows us to go over Windows 11's interaction with the UEFI environment as well as relevant security mechanisms in Chapter 4. With this knowledge we then look at analyses of previously discovered UEFI threats in Chapter 5, categorizing them by their attack vector and threat model. In Chapter 6 we discuss the test setups, we performed our attacks on, consisting of emulation and hardware. We then lay out our practical approach of implementing our own UEFI attacks in Chapter 7, analyzing security mechanism faced when attempting attacks from the UEFI environment. Chapter 8 describes the results reflected through our attacks. Afterwards we discuss the impact of our findings, the restrictions that apply, as well as potential mitigation techniques in Chapter 9. Chapter 10 concludes the thesis by summarizing the achievements of our attacks and lays out potential future topics.

2 UEFI/PI

"The UEFI specifications define a new model for the interface between personal-computer OS and PF. [...] Together, these provide a standard environment for booting an OS and running pre-boot applications" [4]. The specifications making up this model are:

- ACPI Specification
- UEFI Specification
- UEFI Shell Specification
- UEFI PI Specification
- UEFI PI Distribution Packaging Specification
- TCG EFI Platform Specification
- TCG EFI Protocol Specification

The Advanced Configuration and Power Interface (ACPI) and UEFI PI Distribution Packaging Specification are not required to be able to follow this thesis. As for the other specifications, we briefly summarize the relevant content.

2.1 Unified Extensible Firmware Interface (UEFI)

The UEFI specification is a pure interface specification, describing a programmatic interface for boot applications to interact with the PF. It states what interfaces, structures and abstractions a PF has to offer and implement and what an boot applications such as OS loaders may use [5].

It was designed to replace the legacy Boot Firmware BIOS, while also providing backwards compatibility with a Compatibility Support Module (CSM) allowing UEFI firmware to boot legacy BIOS applications [5]. It is aimed to be a complete solution, abstracting all platform features and capabilities in a way so that bootloaders require no knowledge about the underlying hardware [6, p. 1.3].

The UEFI interfaces are defined in the `C` programming language. During boot system resources are owned and managed by the UEFI firmware until the OS explicitly assumes control over the system. On `x64` Central Processing Unit (CPU) architecture the PF hands over execution in `64-bit` long mode which includes memory protection. Paging is also enabled and the virtual memory is identify mapped, meaning virtual addresses equal physical addresses, while most regions are read, write and execute. The CPU is in uniprocessor mode and a sufficent amount of stack is available [6, Section 2.3.4].

Since UEFI mainly exists to offer a well-defined boot environment for OS loaders, its services are not required anymore upon OS takeover. A large part of UEFI can thus be unloaded, leaving only a portion of the intially offered services. The remaining runtime services can be used by the OS to configure or update the PF.

2.1.1 Globally Unique Identifier (GUID)

The UEFI environment depends on GUIDs, also known Universally Unique Identifiers (UUIDs) to uniquely identify a variety of things, such as protocols, files, hard drive partitions. GUIDs are 128-bit long, statistically unique identifiers and can be generated on demand and without a centralized authority, statistically guaranteeing that there will be no duplicates on a system that combines hard and software from multiple vendors [7].

2.1.2 GUID Partition Table (GPT)

Partitions allow a disk to be distinctly separated into logical disks, allowing for each to be formatted with a different file systems. Prior to UEFI disks have been partitioned using the Master Boot Record (MBR) partition table, supporting up to 4 different partitions. The MBR is stored within the first sector, also optionally containing 424 bytes of bootable code through which the BIOS boots [6, Section 13.3.1]. UEFI is still backwards compatible with MBR partitioned disks and contained on each disk, but UEFI does not execute the boot code. The MBR is used in two different ways by the UEFI environment, either as a legacy MBR or a protective MBR. With the legacy MBR, UEFI uses the partitions defined in the MBR partition table, where as the protective MBR only has one partition spanning the entire disk. The protective partition is for legacy devices and in reality GPT partitioning is used to separate the disk. For this UEFI defines two OS types used in MBR partition entries. One identifies the ESP, the partition UEFI boots from, within the legacy MBR partition table and the other indicates that a protective partition is used [6, Section 5]. [6, Section 5] defines the GPT disk layout, with the GPT format Logical Block Address (LBA) are 64 bit instead of 32 bit, allowing to support drives with up to 9400000000 Terra Byte (TB) of storage, where as MBR is limited to 2 TB. This is accompanied by allowing many more than 4 partitions, with Windows supporting up to 128 [8]. GUID are used to identify partitions and partition types, but also offering a human readable partition name. GPT also has a primary and a backup partition table for redundancy pruposes, the primary table follows the MBR sector and the backup is at the end of the disk.

2.1.3 EFI System Partition (ESP)

The ESP can reside any media that is supported by the UEFI firmware and has to be File Allocation Table (FAT)32 formatted [6, Section 13.3]. It must contain an EFI root directory [6, Section 13.3.1.3] and all UEFI applications, that are to be launched directly by the UEFI firmware have to be located in subdirectories below the EFI driectory [6, Section 13.3.1.3]. Drivers and indirectly loaded applcations have no storage restrictions. Vendors are to use vendor-specifically named subdirectories within the EFI directory. Fixed disks have no restrictions on the amount of ESPs present, whereas removable media is only allowed to have one ESP, so that boot behavior is deterministic. In general the ESP is identified by a specific GUID, but implementations are allowed to support accordingly structured FAT partitions. Since there is no limitation on the amount of ESPs, boot applications can share the drive with their OS, or can be accumulated in a single system-wide ESP [6, Section 13.3.3].

2.1.4 UEFI Images

UEFI Images are files containting executable code, they use a subset of the Portable Executable 32-Bit (PE32)+ file format with a modified header signature. The format comes with relocation tables, making it possible for the images to be executed in place or to be loaded at non pre-determined memory addresses. They support multiple CPU architectures such as IA, ARM, RISC-V and x86. There are three different subtypes of executables: applications, boot and runtime drivers. They mainly differ by their memory type and how it behaves. Loading and transferring execution are two separate steps, so that security policies can be applied before executing a loaded image [6, Section 2.1.1].

Applications are always unloaded when they return execution, while drivers are only unloaded when they return an error code. This allows drivers to install their offered functionality upon intial execution and function calls can jump back into the driver image where the function body remains loaded. Boot drivers are unloaded when an OS loader application transitions to runtime by taking over the memory management through the call of the boot service function ExitBootServices, while runtime drivers remain loaded and are translated into the virtual memory mapping. OS loaders only return execution in error cases.

2.1.5 Protocols and Handles

Protocols are created and discovered dynamically and provide a mechanism to allow extension of firmware capabilities over time [9, Section 3.6]. They are C structures and may contain services, in the form of function pointers, or other data structures, they are identified by GUIDs and stored in a single global database implemented by the firmware [5]. This database is called the handle database, handles describe a logical grouping of one or more protocols [9, Section 3.6]. Handles are unique per session and should not be saved across reboots [5]. Multiple instances of a protocol identified by the same GUID can exist on different handles, offering the same service on different devices.

[9] explains the categories of handles that are formed by the type of protocols that are grouped. Figure 2.1 shows these categories.

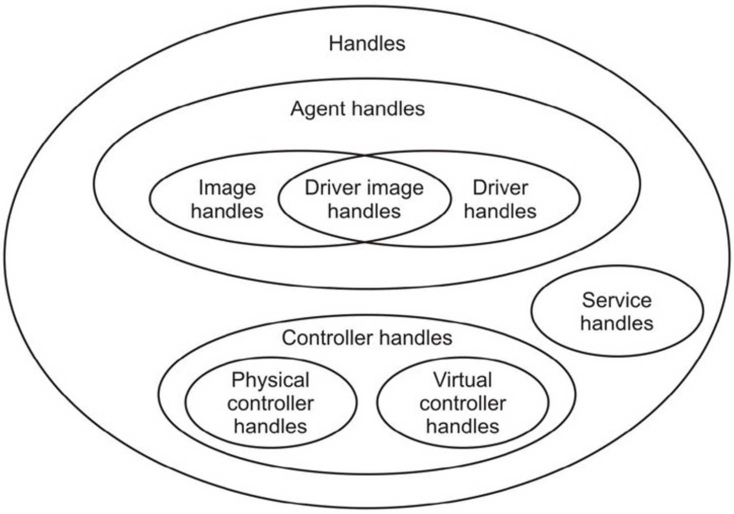


Fig. 2.1.: Handle types (taken from [9, Figure 3])

Image handles are handles of UEFI images loaded into memory, as they support the Loaded Image Protocol, giving access to information about the image in memory. This includes the image's address, size, memory type, origin and optional load options.

Driver handles are handles that group the UEFI Driver Model related protocols (Driver Binding Protocol, the two Component Name Protocols and the two Driver Diagnostics Protocols)

Driver image handles are UEFI Driver Model related protocols installed onto images loaded in memory.

Agent handles is a term used in the UEFI Driver Model, they describe tracked consumers of other protocols.

Controller/Device handles are interchangably used to refer to physical and virtual devices that offer I/O abstraction protocols. Physical device handles support the Device Path Protocol for generic path/location information [6, Section 10.2].

Service handles are used for generic hardware unrelated abstractions.

2.1.6 UEFI Driver Model

[6, Section 2.5] describes the UEFI Driver Model, it is designed to simplify the implementation process of device drivers by moving common device mangement and discovery functionality into the firmware. This leaves drivers with only the responsibility to offer interfaces for installation and removal.

An UEFI platform is assumed to consist of one or more CPUs to be connected to one or more core chipsets. The core chopsets produce host bus controllers which offer initial I/O abstractions like the Peripheral Component Interconnect (PCI) Root Birdge I/O Protocol. Bus drivers can be connected to these host bus controllers to discover and create child controllers, these can either be further buses or devices. This forms a tree of buses with devices as leaves. Devices can be keyboards, mice, monitors, etc. (for user in-and ouput) or hard drives, network adapters, etc. (boot devices) [6, Section 2.5].

Device drivers are very similar to bus drivers, but they do not create new device handles, they instead offer abstractions built upon existing bus driver I/O abstractions. A driver following the UEFI Driver Model is not allowed to interact with any hardware in its entry point and instead installs protocols on its own image handle. It is required to install the Driver Binding Protocol and may additionally install configuration or diagnostic related protocols. The Loaded Image Protocol also offers a field where a driver can provide a function through which it can be unloaded. Runtime drivers usually register a notification function that is triggered when an OS loader calls `ExitBootServices()`, allowing them to translate any allocated memory to their virtual addresses [6, Section 2.5.2].

The firmware will try to connect device drivers to a controller by using the drivers instance of the Driver Binding Protocol and call the `Supported` function on a controller handle. The device driver then checks whether it supports the controller. This could be, for example, looking for specific I/O protocols, that it will want to later use and further abstract. If the driver supports the device, the firmware will call the `Start` function of the Driver Binding Protocol to have the driver install its offered protocols on the controller handle. The firmware connection process can be done recursively as the newly installed device driver might now fullfil the requirements for another driver. If a driver needs to be uninstalled from a specific controller, the firmware can call the `Stop` of the Driver Binding Protocol function on the controller handle. An example for this would be another device driver wanting to exclusively manage a controller. To support this functionality all consumers of a protocol are tracked in the form of agent handles [6, Section 2.5.4].

The part of the firmware that will connect the device drivers is typically the UEFI boot manager. This allows for fast startup, where it may choose to connect only drivers related to a certain boot device [6, Section 2.5.6].

2.1.7 Systemtable

```
typedef
EFI_STATUS
(EFIAPI *EFI_IMAGE_ENTRY_POINT)(IN EFI_HANDLE ImageHandle,
                                IN EFI_SYSTEM_TABLE *SystemTable);
```

Listing 2.1: UEFI Image Entry Point

Listing 2.1 shows the entry point of UEFI images, when they are loaded it is the only part that is *linked*, the rest of the communication has to be discovered programmatically through the UEFI System Table. It serves as the entrance door into the UEFI environment, providing access to the generic boot and runtime services, as well as system configuration information [9, Section 3.3]. The Loaded Image Protocol instance provides an interface to hand over optional load options to an image [5].

Figure 2.2 shows the system table. The functionality of the system table is only available in its entirety during boot as the boot services and structures are eventually unloaded, when the OS takes over control.

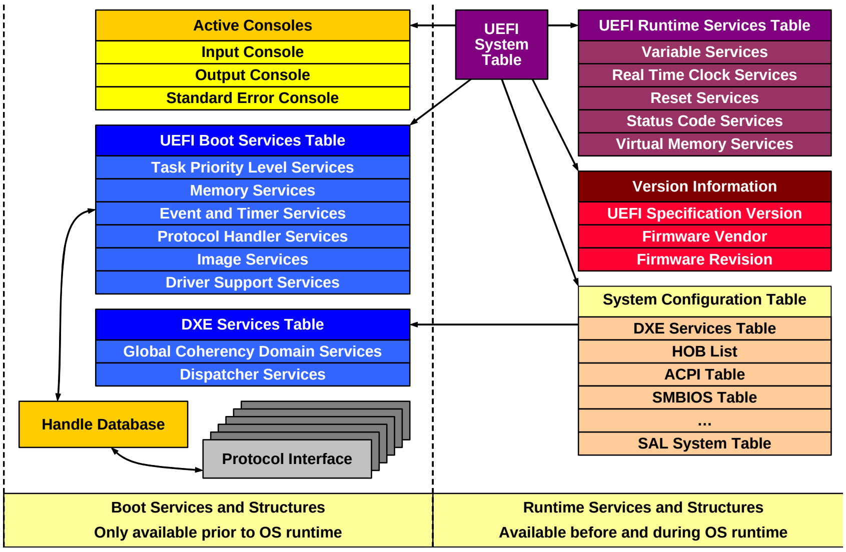


Fig. 2.2.: UEFI System Table (taken from [10, Vol 2, Figure 2-5])

2.1.7.1. Boot Services

UEFI applcations must use boot service functions to access devices and allocate memory. They are available until an OS loader takes over control with `ExitBootServices()`. [6, Section 7] splits the boot services into five categories:

Event, Timer, and Task Priority Services used to create, close, signal, wait for and check events. Setting timers and raising or restoring task priority levels.

Memory Allocation Services to allocate and free pools or whole pages of memory, as well as retrieve the UEFI managed memory map.

Protocol Handler Services used to install, uninstall and retrieve protocol instance as well as abstractions related to the UEFI Driver Model.

Image Services to load, unload and start images. Images can also use these to transfer execution back to the firmware or with `ExitBootServices()` assume control over the system

Miscellaneous Services offer basic memory manipulation, checksum calculation, watchdog timers and monotonic counters.

2.1.7.2. Runtime Services

The runtime services only offer minimal functionality for the OS to communicate with the PF during runtime.

Variable Services used to query, get and set variables.

Time Services used to get and set time as well as a system wakeup timer.

Virtual Memory Services relate to enabling virtual memory and translating memory addresses.

Miscellaneous Runtime Services offer system reset, a monotonic counter and capsule services. Capsules allow the OS to pass data to the firmware, including firmware updates.

2.1.8 Variables

UEFI variables are key/value pairs used to store arbitrary data passed between the UEFI firmware and UEFI applications. The data type has to be known beforehand and as such is specified for variables defined in UEFI. The Storage implementation is not specified by UEFI, but it must support non-volatility, to retain after reboots, or temper resistance if demanded. Variables are defined by a vendor GUID, a name and attributes. Attributes include their scope (boot, runtime, non-volatile), whether writes require authentication or result in appending data instead of overwriting

[6, Section 8.2]. Architecturally defined UEFI variables are called Globally Defined Variables where the vendor GUID has the value EFI_GLOBAL_VARIABLE [6, Section 3.3].

2.1.9 Boot Manager

The UEFI boot manager is a firmware component executed after the platform is completely initialized, it decides which UEFI drivers or applcations are loaded and when. The boot behavior is configured through architecturally defined Non-volatile RAM (NVRAM) global variables[6, Section 3.1]. Each load option entry for a driver or application resides in a variable following the naming scheme of Driver#### or Boot#### respectively. Where # stands for a hexadecimal digit forming a 4 digit number, requiring leading zeros. If a firmware implementation allows for the creation of new load options they can then be added to the ordered lists DriverOrder and BootOrder, they reference load options and dictate the order in which they are processed. Driver load options are processed before the boot load options, there also exists the BootNext variable to override the boot options once. A general depiction of the UEFI boot flow can be seen in Figure 2.3. Implementations usually allow for an interactive menu, where users can modify the order or boot entries manually [6, Section 3.1.1]. Boot options are generally first attempted to be loaded through the LoadImage boot service. If the device path of a boot option only points to a device instead to the file on a device, it attempts to load a default boot application with the Simple File System Protocol[6, Section 3.1.2], for x64 it uses the default path \EFI\BOOT\BOOTX64.EFI [6, Section 3.5].

2.1.10 Security

UEFI offers security mechanisms restriting what is allowed to modify and be modified on a system. This involves authentication of ownership over the platform.

2.1.10.1. Secure Boot

Secure Boot provides a secure hand-off from the firmware to 3rd party applications used for during the boot process, located on unsecure media [11] [6, Sections 32.2 and 32.5.1]. It assumes the firmware to be a trusted entity and all 3rd party software to be untrusted, this includes images from hardware vendors in PCI option Read-Only Memorys (ROMs), bootloader from OS vendors and tools such as the

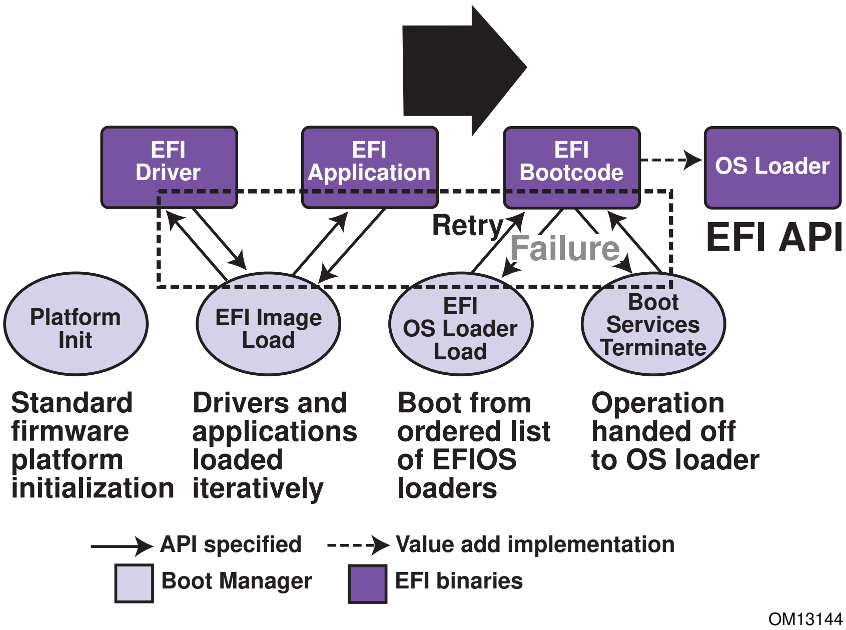


Fig. 2.3.: Booting Sequence (taken from [6, Figure 2-1])

UEFI shell [11]. Digital signatures, embedded within the UEFI images, can be used to authenticate origin and/or integrity [6, Section 32.2]. This is done through asymmetric signing, component provider must sign their executables with their private key and publish the public key. The public keys are stored in a signature Data Base (DB) and before execution the signed executable can be verified against the database. Multiple signatures can be embedded within the same image [6, Section 32.2.2]. The signatures are created by first calculating a hash over select parts of the executable and then signing it with a private key. The output of this hashing is called a digest and the algorithm for obtaining the digest is defined in [12]. Secure Boot also disallows legacy booting through the CSM.

Secure Boot is managed through three components, a Platform Key (PK), one or more Key Exchange Key (KEK) and the signature DBs.

PK The PK establishes a trust relationship between platform owner and firmware, the public half is enrolled into the firmware. The private half represents platform ownership, as it can be used to change or delete the PK as well as enroll or modify KEKs.

KEK The KEK establishes a trust relationship between OS and firmware, as its private half is used to modify the signature DBs.

Signature Data Bases (DBs) Signature DBs contain image hashes and certificates, to either allow or deny execution of associated images.

Internally these are all implemented by authenticated variables, residing in tamper resistant non-volatile storage [6, Section 32.3]. The PK is a simple variable where the KEK and DB are implemented through signature list data structures [6, Section 32.4.1], the variable services can be used to append entries or to read and write the list as a whole [6, Sections 32.3.5 and 32.5.3]. The variables are part of the Globally Defined Variables, for each variavble also exist a variant reserved for default entries. These can be used by an OEM to supply platform-defined values, used during Secure Boot initialization by a user. Their contents can be copied to their live versions, to then be used during Secure Boot operation. The current state of Secure Boot is communicated with a secure variable, which the the OS can probe [6, Section 3.3].

Users, who are physically present, may disable Secure Boot, enroll default or custom keys via an interactive menu [6, Section 3.3].

2.1.10.2. Firmware Management

The Firmware Management Protocol provides a boot abstraction for authenticated updating and management of the PF [6, Section 23]. The runtime services `QueryCapsuleCapabilities()` and `CapsuleUpdate()` may be used by the OS to pass updates to the firmware in a persistent manner, so that they can be processed on subsequent boots [6, Section 23.3]. OEMs also often provide their own ways to process firmware updates, for example via dedicated Universal Serial Bus (USB) ports, which allow to process firmware updates upon boot. As these are entirely dictated by the platform designer, it is not possible to make vendor independent assessments about their security.

2.1.10.3. User Identity Policies

UEFI enables a system to have multiple users with variying levels of privileges. This may restrict their ability to enroll other users or to boot off select drives [6, Section 36.1.2]. A trusted environment must be maintained for the integrity of the security identification, by restriting which drivers are loaded and securing the storage of drivers [6, Section 36.1.4].

2.2 UEFI Platform Initialization (PI)

[10] defines an implementation of the PI process and the UEFI environment, as well as a scalable PI firmware storage and interface solution.

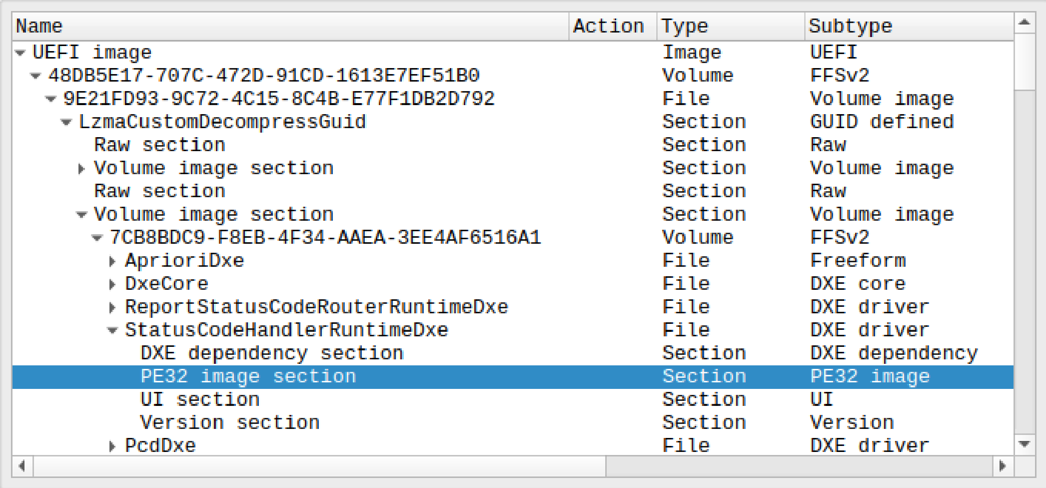
2.2.1 PI Firmware Images

[10, Vol. 3, 2] defines the firmware storage design. A Firmware Image is stored in one or more non-volatile physical storage devices called Firmware Devices (FDs), they are most commonly flash devices [10, Vol. 3, 2.1]. Flash often offers the ability to restrtict read and write properties differently depending on the storage region [10, Vol. 3, 2.1.1]. UEFI variables may reside in a region that remains read- and writable during the whole operation of a system, whereas the code storage may only be writable during the initial PI phases. Firmware images might be split over multiple physical FDs, but may also be inturn be logically be split into Firmware Volumes (FVs). FVs are comparable to hard drive volumes as they also are formatted with a file system, usually the PI Firmware Filesystem (FFS) format defined in [10, Vol. 3, 2.2]. The PI FFS is a flat file system consisting of a single list of files without any directory structure. Parsing the volume is done by iterating over all files one by one. Files contain code or data in the form of sections. Sections split a file in discrete parts with the type of a section dictating its content. File types impose restrictions on which types of sections a file may contain or not. The full list of file types defined in the PI specification can be seen in Table A.1. File sections are organized in trees, with encapsulating as well as leaf sections. Together with the file section type `EFI_SECTION_FIRMWARE_VOLUME_IMAGE` which contains an entire PI FV image, this makes up for the FFS's lack of a directory structure. The full list of section types can be seen in Table A.2.

Figure 2.4 shows a firmware image opened in UEFITool, an editor for firmware images conforming to the PI specification [13]. The cursor is on the executable section of a DXE driver.

2.2.2 PI Architecture Firmware Phases

[10] defines a multi phase architecture of the PI that can be used by system designers to implement PF. It is designed to be very extensible and to simplify the process of independent hardware vendors working together, when combining their software



Name	Action	Type	Subtype
UEFI image		Image	UEFI
48DB5E17-707C-472D-91CD-1613E7EF51B0		Volume	FFSv2
9E21FD93-9C72-4C15-8C4B-E77F1DB2D792		File	Volume image
LzmaCustomDecompressGuid		Section	GUID defined
Raw section		Section	Raw
Volume image section		Section	Volume image
Raw section		Section	Raw
Volume image section		Section	Volume image
7CB8BDC9-F8EB-4F34-AAEA-3EE4AF6516A1		Volume	FFSv2
AprioriDxe		File	Freeform
DxeCore		File	DXE core
ReportStatusCodeRouterRuntimeDxe		File	DXE driver
StatusCodeHandlerRuntimeDxe		File	DXE driver
DXE dependency section		Section	DXE dependency
PE32 image section		Section	PE32 image
UI section		Section	UI
Version section		Section	Version
PcdDxe		File	DXE driver

Fig. 2.4.: Open Virtual Machine Firmware (OVMF) opened in UEFITool

into a single PF. The proposed PI architecture phases can be seen in Figure 2.5, they are not entirely distinct and system designers may choose to implement phases differently.

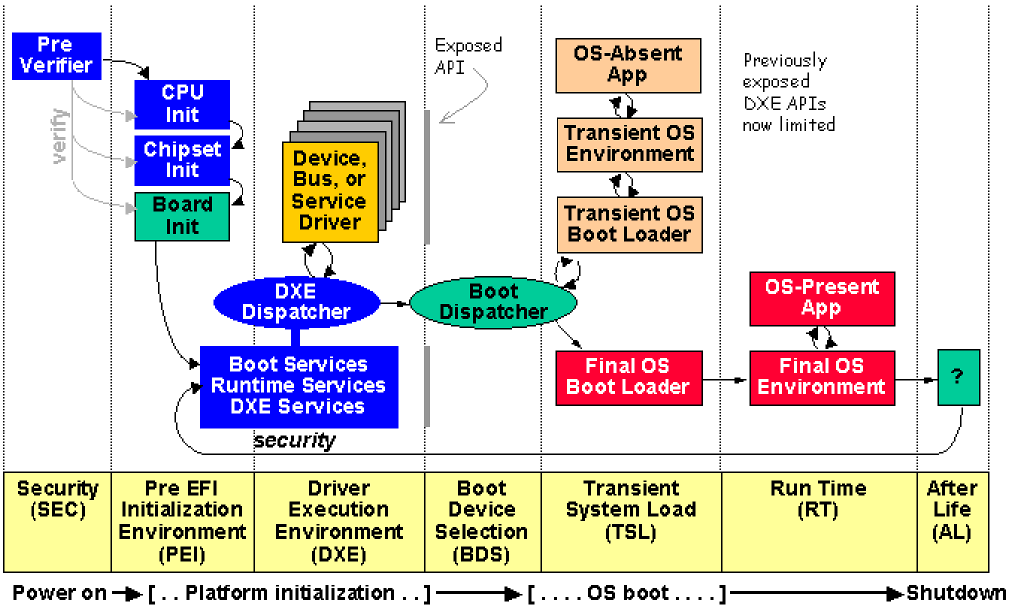


Fig. 2.5.: PI Architecture Firmware Phases [10, Figure 2-1]

2.2.2.1. Security (SEC)

The SEC phase is the first phase performed during platform initialization. Under its responsibilities fall handling all platform restart events, setting up temporary memory and establishing the system's root of trust. It serves as the foundation for all secure operations on which inductive security designs rely to build a chain of trust by having a module verify the integrity of its subsequent module. For this the SEC phase may verify the integrity of the PEI foundation before transfering execution to it. As this is very specific to how the platform is implemented, the SEC phase is only specified as the basic requirements that it needs to meet before handing over execution to the PEI phase. When it transfers execution, it passes information about the current state of the system, including location and size of the temporary stack, Random Access Memory (RAM), and Boot Firmware Volume (BFV). It can also optionally pass protocols for the PEI phase to use.

2.2.2.2. Pre-EFI Initialization (PEI)

The PEI phase configures the system to meet the minimum requirements for the DXE. Its job is the intialization of all system hardware requiring to be initialized beforehand, as well as the initialization of permanent memory, which is later described in Hand-off Blocks (HOBs) to be passed off to the DXE phase [10, Vol. 1, 2.1]. The PEI phase is architecturally a stripped down version of the DXE phase, as it offers the same extensibility through modules supplied by the different OEMs responsible for the component initialization. As the PEI's environment is still very restricted and the main memory only initialized towards the end of the phase, more complex processing is to be done in the DXE. Even though the implementation of the PEI phase is the most hardware dependent, the core functionality of the PEI is common to all processor architectures and offered through the PEI foundation [10, Vol. 1, 2.2]. It is the module initially invoked by the SEC phase and responsible of dispatching further Pre-EFI Initialization Modules (PEIMs) and offering am intermodule communication through the management of PEIM-to-PEIM Interfaces (PPIs) [10, Vol. 1, 2.5]. The PEI foundation implements a PEI dispatcher, who iterates over PEIMs found in FVs, to evaluate their Dependency Expressions (DEPEXs). DEPEXs are logical combinations of PPIs that must be present before loading a PEIM. Loading PEIMs results in the installation of new PPIs and the discovery of additional FVs. Theis leads to previously unfullfilled DEPEXs to now be fullfilled and the dispatcher loading these PEIMs on their next evaluation. This process is repeated until no more PEIMs are able to be dispatched. The foundation then invokes the *DXE IPL PPI*, which loads the DXE foundation into memory, to then transfer execution [10, Vol. 1, 2.6]. The BFV containing the PEI foundation, initially discovered by the SEC phase, and any additionally discovered FVs are also passed off to the DXE in the form of HOBs.

While the PEI phase has many architecturally required PPIs that modules have to implement, the PI also defines optional PPIs. One of which is the *Security PPI*, it used to maintain the chain of trust by offering the chance for platform builders to authenticate or log PEIMs before they are executed [10, Vol. 1, 6.3.6].

2.2.2.3. Driver Execution Environment (DXE)

The DXE phase is responsible to finalize the initialization of all platform components, as well as implementing UEFI, the UEFI environment and its system abstractions as they are defined in [6]. As mentioned earlier, the DXE phase is architecturally similar to the PEI phase, as it also has a foundation with a dispatcher, extensibile modules in the form of DXE drivers and uses UEFI protocols for intermodule communication [10, Vol. 2, 2.1]. The DXE foundation is only dependent on the list of HOBs it receives from the previous phase, allowing it to be used with the PEI phase or different implementations. This also makes it possible to unload the previous stage, freeing up its memory [10, Vol. 2, 9.1]. The DXE foundation produces the UEFI boot and runtime services as well as additional DXE services and offers them through the UEFI system table to its DXE drivers as it can be seen in Figure 2.2[10, Vol. 2, 2.2.1]. DXE drivers are very similar to UEFI images as they even share the same entry point signature, they also come in boot and runtime variants [10, Vol. 2, 11.2.3]. The implementation of the UEFI system table services is done by DXE drivers, who provide architectural protocols for the DXE foundation to consume [10, Vol. 2, 12.1]. Thus the foundation has to provide the most basic services, required to load and execute DXE drivers, on its own. DXE drivers implementing architectural protocols are called early drivers, they can not assume that that all UEFI system table services are already available to them and do not follow the UEFI driver model [10, Vol. 2, 11.2.1]. To guarantee that some of the architectural drivers are loaded before others, an *a priori* file can be used. When an *a priori* file is present on a FV, it is read to provide a deterministic order of drivers, which are to be executed before the dispatcher starts its regular DXE driver discovery and DEPEX evaluation on the rest of the architectural drivers [10, Vol. 2, 10.3]. DXE drivers which follow the UEFI driver model have an empty DEPEX, as installing the Driver Binding Protocol to its own image handle does not require any architectural protocols. They are still only disptached after all architectural protocols have been installed [10, Vol. 2, 11.2.2].

The dispatcher also makes use of a Security Architectural Protocol to authenticate each DXE driver to deciding whether or not to execute it [10, Vol. 2, 10.13].

When the DXE dispatcher is unable to load any new drivers it transfers execution to the *BDS Architectural Protocol* [10, Vol. 2, 2.4]. This presents the advancement into the BDS phase, but not simaltaniously the end of the DXE phase [10, Vol. 2, 2.1]. The two phases work together until the OS takes over control of the system with the call to `ExitBootServices()`.

2.2.2.4. Boot Device Selection (BDS)

The BDS phase consists of the implementation of the *BDS Architectural Protocol*. It implements the UEFI boot manager policy as defined in [6, Section 3] and summarized in subsection 2.1.9. When discovering additional FVs the DXE dispatcher is invoked through the DXE services. Execution may also be returned to the DXE phase when not enough drivers were initialized to successfully boot from a device [10, Vol. 2, 12.2].

2.2.2.5. Transient System Load (TSL)

The TSL phase constist of the UEFI OS bootloader performing its necessary actions in preparation of assuming control over the system by calling `ExitBootServices()` and transfering execution to the OS kernel [14, Section 2.3]. The DXE phase and all boot services are unloaded.

2.2.2.6. Runtime (RT)

The RT phase offers only minimal functionality via the remaining runtime services while the OS owns the system [14, Section 2.3].

2.2.2.7. Afterlife (AL)

The AL phase facilitates drivers storing the system state during shutdown, sleep, hibernation and restart events [14, Section 2.3].

2.2.3 Security

The PI specification defines PPIs and DXE protocols which can be used to validate images when loading them. During the PEI phase the *PEI Guided Section Extraction PPI* can be used to authenticate of file sections, while the *Security PPI* implements the policy response to the authentication result. The DXE phase has counter parts in the form of *Guided Section Extraction Protocol* and *Security Architectural Protocol*. The policy response may be the locking of flash upon authentication failure or attestation logging [10, Vol. 2, Section 12.9.1]. It also has the architectural protocol *Security2 Architectural Protocol*, which implements Secure Boot validation, Trusted Computing Group (TCG) measured boot and User Identity policy for image loading. The implementation of the boot service `LoadImage` has to use these protocols in accordance to the rules defined in [10, Vol. 2, Section 12.9.2]. The Security2 protocol is invoked on every image loaded, with the Security protocol being invoked afterwards on images loaded through the Firmware Volume Protocol. When the Security2 protocol is not installed it uses the Security protocol regardless of the image's origin.

2.2.3.1. Hardware Validated Boot

Secure Boot relies on the firmware as its root of trust. Hardware Validated Boot is able to shift the root of trust out of the firmware image into a samller part in the hardware, in hopes to reduce the size of the attack vector. This part performs validation of the Initial Boot Block (IBB) before handing over execution to the firmware image [TODO cite me]

2.2.3.2. Firmware Protection

The PI speficiation defines an *End of Dxe Event*, which indicates the introduction of third party software execution to the platform. Up until this point it is assumed that the entire system software is under the control of the platform manufacturer. Drivers may react to this event by locking critical system resources, using the System Management Mode (SMM) related services [10, Vol. 2, 5.1.2.1]. The SMM is a secure execution environment, achieved by isolation from the rest of the system, through the CPU [10, Vol. 4, Section 1.3]. The PI reference implementation also makes use of this event to lock the device thats stores the firmware image [15].

2.3 UEFI Shell

Part of the family of UEFI specifications is a shell specification which defines a feature rich UEFI shell application to interact with the UEFI environment [16, Section 1.1]. It offers commands relating to boot and general configuration, device and driver management, file system access, networking [16, Section 5.1] and scripting [16, Section 4]. A shell application may already be part of the boot options but can always be supplied in the default boot path of removable media.

The UEFI shell is a great tool to visualize the UEFI environment. With the devtree command, for example, we can see the tree of all handles complying to the UEFI driver model. This also serves a great reference of how device paths are formed. Figure 2.6 shows the output of devtree cropped to show a GPT formatted hard drive and its logical partitions. When the firmware discovers a block device it is also required to search for a partition table and create a device handle for each partition. Device drivers abstracting file systems can then be connected to a partition handle and check if it is formatted. The first partition here, listed as *FAT File System*, is the ESP of this drive.

```
Shell> devtree
 Ctrl[34] PciRoot(0x0)
   Ctrl[9D] Sata Controller
     Ctrl[AD] SCSI Disk Device
     Ctrl[AB] QEMU HARDDISK
       Ctrl[A6] FAT File System
       Ctrl[AF] PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(2,GPT,F614BC4C-344A-
       Ctrl[B0] PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(3,GPT,DD1C7D7E-BA2B-
       Ctrl[B1] PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(4,GPT,3DAE20AD-FF95-
```

Fig. 2.6.: Shortened UEFI shell output of devtree

With openinfo we can see the group of protocols that a handle represents. Figure 2.7 shows the output when querying the handle of an ESP. Since this ESP is installed in a logical partition an instance of the Partition Information Protocol is present. The command also lists the agent handles of each protocol and how the protocol was accessed. `TestProt` is often used in the `Supported` function of a driver, while `GetProt` is then used to open the protocol for consumption within the `Start` function.

```
Shell> openinfo A6
Handle A6 (6B7CC98)
SimpleFileSystem
DiskIO
  Drv[7B] Ctrl[A6] Cnt(01) Driver    Image(FAT File System Driver)
PartitionInfo
BlockIO
  Drv[6C] Ctrl[A6] Cnt(01) Driver    Image(Generic Disk I/O Driver)
  Drv[6D] Ctrl[A6] Cnt(01) TestProt  Image(Partition Driver(MBR/GPT/El Torito))
  Drv[6D] Ctrl[A6] Cnt(01) GetProt   Image(Partition Driver(MBR/GPT/El Torito))
  Drv[7B] Ctrl[A6] Cnt(01) TestProt  Image(FAT File System Driver)
  Drv[7B] Ctrl[A6] Cnt(01) GetProt   Image(FAT File System Driver)
DevicePath
  Drv[7B] Ctrl[A6] Cnt(01) GetProt   Image(FAT File System Driver)
```

Fig. 2.7.: Shortened UEFI shell output of openinfo

2.4 EDK II

EFI Development Kit (EDK) II, maintained by *TianoCore*, is an open source implementation of UEFI, offering a modern, feature-rich, cross-platform firmware development environment for the UEFI and PI specifications [17]. It can be used to build modules of all types defined in the PI and UEFI specification and supports the generation of PF images, option-ROMs and bootable media. On top of implementing the PI and UEFI specifications, it defines a lot of helpful libraries and protocols that can be used to simplify the development process. With events that require policiy defined reactions, it often already offers well-defined interfaces for a platform designer to pick up on. The build process is flexible, as it can use different compilers such as GCC and MSVC.

TianoCore also offers a lot of material to learn about developing applications, drivers, firmware, and a general understanding about the UEFI ecosystem.

3 Trusted Platform Module (TPM)

With the TPM the TCG specifies a system component designated for security-related functions, providing the ability to establish trust in a system [18]. Its implementation can be accomplished through dedicated hardware or by using the CPU's isolated SMM [19, Section 9.3]. Asides the generation and secure storage of cryptographic keys, the TPM can be used for system integrity measurements. Boot code is measured into the TPM by the PF, providing evidence over the initialization process and making it possible to detect deviations [18].

3.1 Platform Configuration Registers (PCRs)

[19] defines PCRs to store the system integrity measurements. The registers can only be modified in two ways, either through a complete TPM reset or by extending their values. Extending a PCR is done by concatenating the hashed measurements together with the current PCR values to form the new contents. This creates a chain of measurements where from one diverging measurement on, all subsequent PCR extensions result in entirely differnt values.

$$PCR[i]_{(new)} = Hash(PCR[i]_{(old)} \,||\, Hash(Measurement)) \tag{3.1}$$

The TPM is a passive system components, relying on the host processor to perform measurements and extend the PCRs. The measurement chain starts with a point called the Core Root of Trust for Measurement (CRTM), consisting of the first instructions executed to establish a chain of trust. A Root of trust in a system is an element that must be trusted as its behavior is non-verifiable [19]. [20, p. 3.2.2] requires this chain to start in an immutable portion of the PI process, the Static Root of Trust for Measurement (SRTM). [20, p. 3.2.3.1] defines the PF to be composed of a Boot Block and the UEFI firmware, the Boot Block consists of the SEC and PEI phase as well as the IBB. The Boot Block forms the SRTM, while the UEFI firmware is only part of the chain of trust by being measured from the SRTM. The Root of

Trust for Measurement (RTM) can either start with the SRTM measuring itself or a Hardware-Core Root of Trust for Measurement (H-CRTM) measuring the SRTM. It falls under the responsibility of the PF to perform the integrity measurements [20], different parts of the boot process are measured into separate PCRs. Figure 3.1 shows a high level measurement flow. Table 3.1 shows PCR indexes and their type of content, that is measured, relevant for this thesis.

Interaction with the TPM is done via a well-defined interface, for external chips this is done over hardware busses such as Low Pin Count (LPC) or Serial Peripheral Interface (SPI). TCG specifies the TCG2 Protocol for the UEFI environment, providing an abstracted communication interface indepdenent of the underlying implementation.

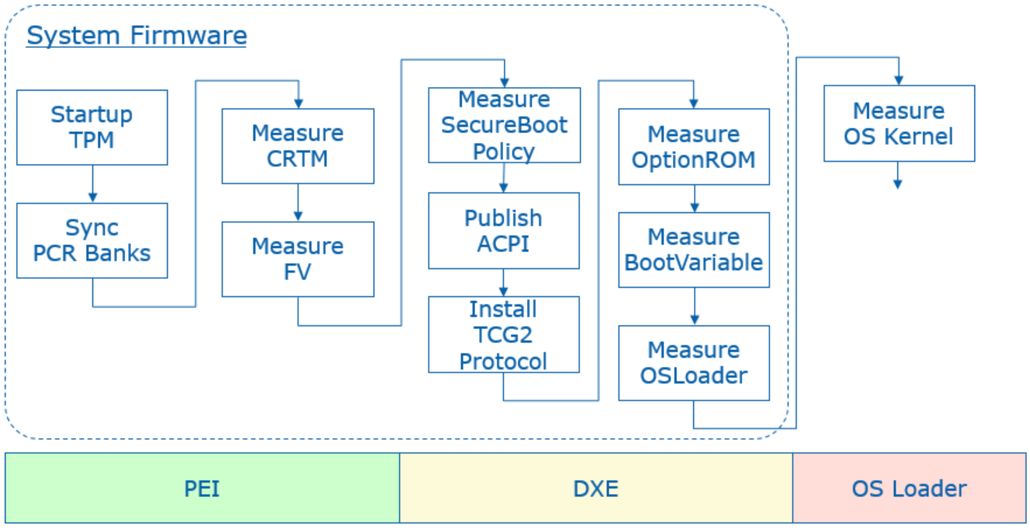


Fig. 3.1.: PF Measurement Flow (taken from [21, Figure 3])

PCR Index	Measurements
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and PI Drivers
1	Host Platform Configuration
2	UEFI driver and application Code
3	UEFI driver and application Configuration and Data
4	UEFI Boot Manager Code (usually the MBR) and Boot Attempts
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) and GPT/Partition Table
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8	First New Technology File System (NTFS) boot sector (volume boot record)
9	Remaining NTFS boot sectors (volume boot record)
10	Boot Manager
11	BitLocker Access Control

Tab. 3.1.: PCR Usage (taken from [20, Table 1] and [22, Table 9-2])

3.2 Sealing/Unsealing

The chain of measurements can be used to attest for a trusted system state. The TPM can be given data, such as a cryptographic key, in a state that is assumed to be trusted. This state is reflected by the current PCRs, consisting of the collection of lead-up measurements. The TPM then seals the data with a policy that describes which PCR indexes to use and/or proof of authentication through a Personal Identification Number (PIN) or passphrase. The data can then only be unsealed on subsequent boots when the system is in the same trusted state that it was in, when the data was sealed. Any modification of the boot code will be reflected in a deviation of PCR values leaving the TPM unable to unseal the data. [TODO CITE]

4 Windows 11

Windows 11 is the latest iteration in the line of desktop OSs from Microsoft, at the time of writing. It is build upon the foundation of Windows 10 and as such shares many security settings and policies with its predecessor [23]. Simaltaniously it raises the requirements of hardware related security features. It does not support legacy BIOS anymore, requires the PF to conform to at least UEFI version 2.3.1 and to be capable of Secure Boot. It also requires the presence of a TPM version 2.0 [24].

4.1 UEFI

To be able to analyse UEFI threats against Windows 11 it is important to understand how Windows interacts with the UEFI environment.

4.1.1 Installation

The interaction with UEFI begins with the installation process and the partitioning of the hard drive Windows is installed onto. When the Windows Installer is launched, it creates at least four GPT partitions on the target hard drive. The EFI System Partition (ESP), a recovery partition, a partition reserved for temporary storage and the boot partition containing the system files. Two copies of the Windows Boot Manager bootmgfw.efi are placed on the ESP, one under EFI\\Boot\\bootx64.efi for the default boot behavior the installed hard drive and one under EFI\\Microsoft\\Boot\\bootmgfw.efi alongside boot resources such as the Boot Configuration Data (BCD). The path of the latter boot manager is saved in a boot load option variable entry Boot####, which is then added to the BootOrder list variable. The boot load option contains optional data consiting of a GUID identifying the Windows Boot Manager entry in the BCD. The BCD, as its name suggests, contains arguments used to configure various steps of the boot process [25, Section 12]. The boot partition is the primary Windows partition and is formatted with the NTFS file system containing the Windows installation. This is also the location of the final step of the Windows UEFI boot process, Windload.efi, the application repsonsible for loading the kernel into memory [25, 12. The Windows OS Loader].

4.1.2 Boot

Now that we established the basic structure of the Windows UEFI boot environment, we can discuss the boot process. The Windows boot process begins after the UEFI Boot Manager launches the Windows Boot Manager, which starts by retrieving its own executable path and the BCD entry GUID from the boot load options. Then it loads the BCD and access its entry. If not disabled in the BCD it loads its own executable into memory for integrity verification [25, Section 12]. Depending on what hibernation status is set within the BCD it may launch the Winresume.efi application, which reads the hibernation file and resumes kernel execution [25, Section 12]. On a full boot it checks the BCD for boot entries, if the entry points to a BitLocker encrypted drive, it attempts decryption. If this fails it shows a recovery prompt, otherwise it proceeds to load the OS loader Windload.efi which maps the kernel image ntoskernl.exe into memory. After a call to `ExitBootServices()` it transfers execution to the kernel [25, Section 12].

4.1.3 Runtime

During runtime Windows uses the variable services to communicate with the PF and even exposes these to application developers. It also supports firmware and option-ROM updates via the capsule delivery services.

4.2 Registry

A crucial part to the whole Windows ecosystem is the Registry, it is a system database containing information required to boot, such as what drivers to load, general system wide configuration as well as application configuration [26, Section 1]. The Registry is a hierachical database containing keys and values, keys can contain other keys or values, forming a tree structure. Values store data through various data types. It is comparable to a file system structure with keys behaving like directories and values like files [25, Section 10]. At the top level it has 9 different keys [25, Section 10]. Normally Windows users are not required to change Registry values directly and instead interact with it through applications providing setting abstractions. Though some more advanced options may not be exposed and can be accessed through the regedit.exe application which provides a graphical user interface to travere and modify the Registry [25, Section 10]. It also supports ex- and importing registry keys along their subkeys and contained values. Internally the registry is not a single large file but instead a set of file called hives, each hive contains one tree, that is mapped into the Registry as a whole. There is no one to one mapping of registry root key to hive file, the BCD file for example is also a hive file and is mapped into the Registry under HKEY_LOCAL_MACHINE\BCD00000000 [25, Section 10]. Some hives even reside entirely in memory as a means of offering hardware configuration through the Registry Application Programming Interface (API).

4.3 Security

Figure 4.1 gives an overview over the security within the Windows startup process. With Secure Boot starting the process and Trusted Boot enventually taking over. We do not cover Measured Boot in this thesis.

4.3.1 Secure Boot

Devices shipping with Windows 11 must have Secure Boot enabled by default [24]. Windows certified devices generally must allow users to enroll custom keys and signature DBs (to allow executioon of non-Windows bootloaders), aditionally a user should be able to completely disable secure boot. Windows offers two signature DBs the *Microsoft Windows Production PCA 2011* required for the Windows boot process and *Microsoft Corporation UEFI Certificate Authority (CA) 2011*, which is reserved for third party executables signed at Microsoft's discretion after manual review [27]. Microsoft advises to only allow other third party UEFI applications if necessary and even mandates the exclusion of DBs other than *Microsoft Windows Production PCA 2011* on Secured-core Personal Computers (PCs) [3].

4.3.2 Trusted Boot

Trusted Boot picks up where Secure Boot left off and maintains the code integrity chain through the kernel into the Windows startup process. Kernel Mode Code Integrity (KMCI) verifies the digital signature of Windows boot components, including boot drivers, startup files and the Early-Launch Antimalware (ELAM) driver [28].

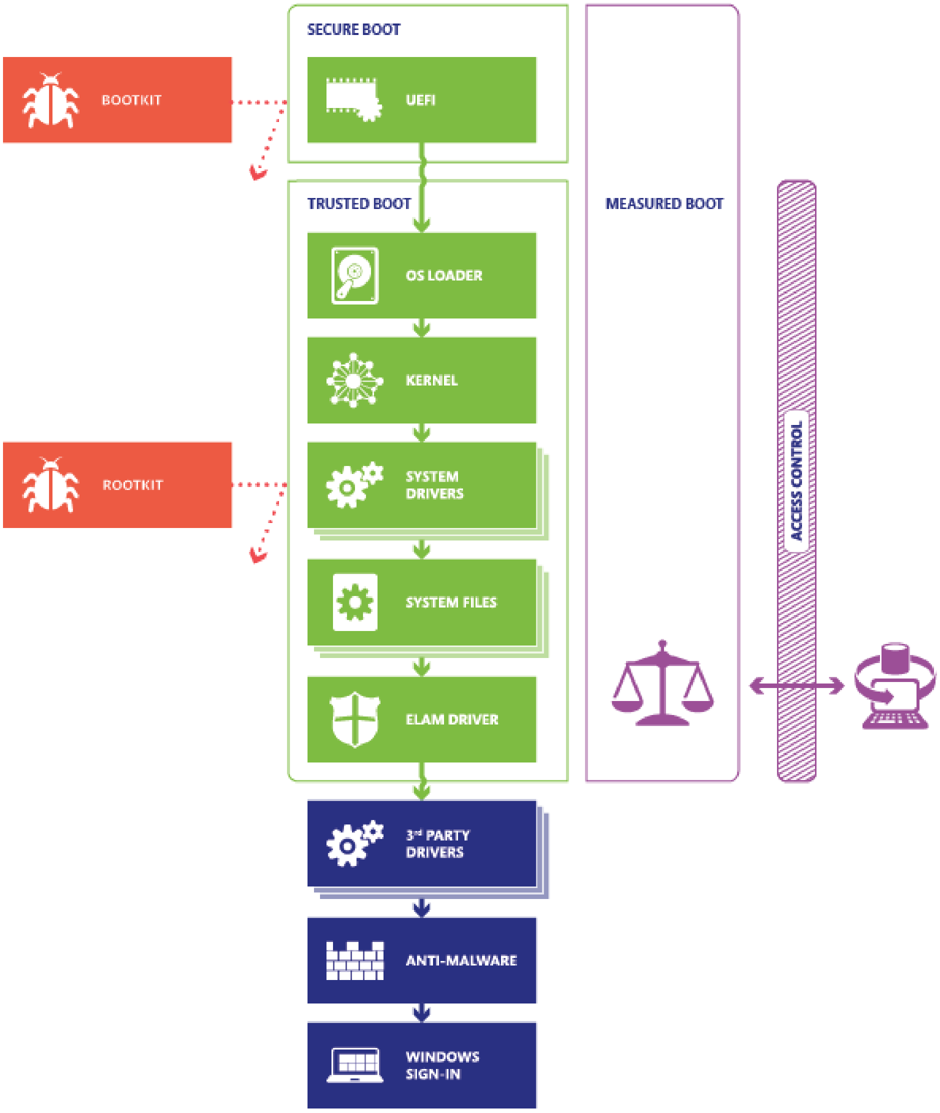


Fig. 4.1.: Windows startup process (taken from [3])

ELAM provides antimalware software developers an interface to be intialized early in the boot process, before other third-party components, to monitor the subsequent boot process [29]. [30] gives a detailed walkthrough of the trusted boot process.

Microsoft can also leverage hardware virtualization features called Virtualization Secure Mode (VSM) for Virtualization Based Security (VBS). This allows for Hypervisor-protected Code Integrity (HVCI) where the Code Integrity (CI) checks are taken out of the kernel environment and are now be performed from within the isolated hypervisor-based security environment [31].

Formerly the term *Device Guard* was used to promote the two security related features HVCI and Windows Defender Application Control (WDAC) (restricts execution of user level applcations). Microsoft has since retired the term to prevent confusion, as there is no direct dependency between the two [32].

4.3.3 BitLocker Drive Encryption (BDE)

Windows is only able to enforce security policies when it is active, leaving the system vulnerable when accessed from outside of the OS [22, Section 9]. Windows uses BitLocker, an integrated Full Volume Encryption (FVE), aimed to protect system files and data from unauthorized accecss while at rest [33]. It also serves as a mechanism to verify boot integrity when used with in combination of a TPM [22, Section 9]. The en- and decryption of the volume is done by a filter driver beneath the NTFS driver as shown in Figure 4.2. The NTFS driver translates file and directory access into block-wise operations on the volume. The filter driver then receives these block operations, encrypting blocks on write and decrypting blocks on read, while they pass through it. This results in en- and decryption, that is entirely transparent to the NTFS driver, making the underlying volume appear decrypted [22, Section 9]. The encryption of each block is done using a modified version of the Advanced Encryption Standard (AES)128 and AES256 cypher [22, Section 9]. A Full Volume Encryption Key (FVEK) is used in combination with the block index as input for the algorithm, resulting in an entirely different output for two blocks with identical data [22, Section 9]. The FVEK is encrypted with a Volume Master Key (VMK) which is in turn encrypted with multiple protectors. These encrypted versions of the VMK are stored together with the encrypted FVEK in an unencrypted meta data portion at the beginning of the BitLocker protected volume [22, Section 9]. The VMK can be encrypted by the following protectors:

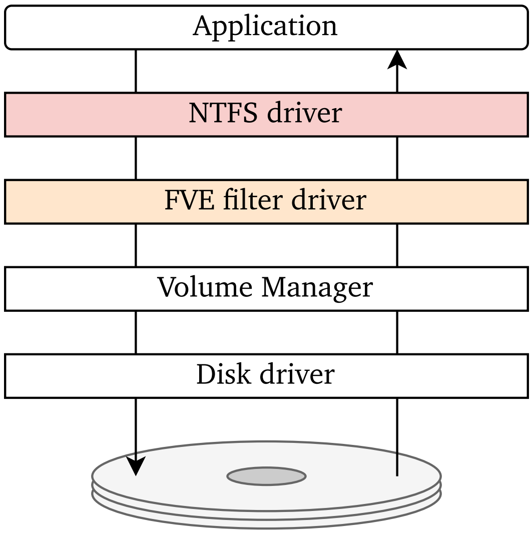


Fig. 4.2.: BitLocker Volume Access Driver Stack (inspired by [22, Figure 9-24])

Startup key The Startup key can be storead on removable media such as a USB stick and serves as physical proof of ownership. The removable media contains a .bek file, that is named with a GUID corresponding to BitLocker meta data entry, as it it possible to have multiple start up keys for the same volume [34, Section 2.6][35].

TPM When a TPM is used to seal the VMK, BitLocker can ensure integrity of early boot components and boot configuration, as the unseal operation fails upon modifcation of the boot flow. The TPM can, in combination to the PCR values, use a startup key, a pin or both to seal the VMK [36]. The collection of PCR indexes used to instruct the TPM when initially sealing the VMK is called a validation profile. The validation profile is configurable through Windows group policy settings, andPCR11 is always required as its contents measure the BitLocker Access Control. Windows configures BitLocker with a default validation profile of `{0, 2, 4, 11}` [37]. When Secure Boot is enabled and correctly configured, Microsoft defines this as only using their *Microsoft Windows Production PCA 2011* signature DB, BitLocker defaults to a validation profile of `{7, 11}` [38]. The measurements that extend PCR7 are defined in [39]. The content reflects the current state and configuration of Secure Boot, including trusted keys. BitLocker then makes use of the Secure Boot measurements for integrity validation instead of the PCR values containing the early boot components [37].

Recovery key When BitLocker is activated it always generates a recovery key serving as an additional protector to whatever primary method is selected.

This allows users to recover their data when for example the TPM fails to unseal the VMK or the startup key is lost. The recovery key can be added to the users Microsoft account or saved to unencrypted media. It consists 48 digits divided into 8 blocks, each block is converted into a 16-bit value making up a 128-bit key [34, Section 2.4].

User key When BitLocker is used without a TPM the VMK can be encrypted with the use of a user supplied password with the maximum length of 49 characters. [34, Section 2.7]

Clear key BitLocker can also be suspended when a user for example wants to update their PF. The VMK is then encrypted with an unprotected 256-bit key stored on the volume [34, Section 2.5].

5 Past Threats

Before we implement our own UEFI attacks, we first take a look at how past UEFI threats have approached this problem. The threats discussed range from actual attacks found in the wild and analyzed by security researchers, over attacks, which have similarly been implemented for research purposes, to tools to enable system owners more advanced control over their systems.

Approach	Bootkit	Rootkit
Storage-based	**ours**	Vector-edk Mosaicregressor LoJax **ours**
Memory-based	Efiguard ESPecter Dreamboot FinSpy	Moonbounce Cosmicstrand

5.1 Infection

The infection is the most important part of an attack, as it dictates when and in what environment, with what privileges the UEFI payload is executed.

5.1.1 Bootkit

Bootkits use the UEFI Boot manager to gain execution on a system, there are a variety of methods using different options of the boot mechanism. [40] backs up and replaces the Windows Boot Manager bootmgfw.efi on the ESP. [41] patches the entrypoint of bootmgfw.efi and its copy bootx64.efi in the default boot path, so that it executes malicious code upon launch. [42] and [43] are more proof of concept than real attacks and suggest to be used from removable media, but they are also able to

33

be added to the default boot path on an ESP, or generally added as their own boot entry [43], as they are both applications which launch the Windows Boot Manager upon execution. **[TODO Generally it is possible to mount the ESP from within Windows with administrative privileges]**

5.1.2 Rootkit

Firmware rootkits have been rarer and how exactly the firmware images were infected is often not known, [44] requires booting the target machine from a USB key [45] **[TODO SPI read/write]** [46] dump remove previous NTFS driver add DXE drivers reflash image

The payload itself has usually simply been DXE drivers residing in a firmware volume [45, 46], as they are automatically executed by the DXE dispatcher. [43] compiles its main UEFI payload as a DXE driver and suggesting its usage as a firmware rootkit. [47] does something different and instead patches the DXE Core over adding files to FVs. While the approach could fundamentally be done in the form of a DXE driver, it makes tge detection harder [47].

5.2 Approach

We can categorize the threats by their attack vector, rootkits and bootkits do not seem to have distinct approaches, as they both start their execution in the UEFI environment prior to the Windows boot process. We found that their approach can mainly be divided into storage-based and memory-based attacks. Storage-based attacks mostly gain execution in the operating system environment by writing their payload into the Windows installation and modifying configuration data on disk. These attacks are often performed offline, before any parts of the operating system are executed. Memory-based attacks instead hook into the operating system's boot process to execute malicious code alongside operating system in memory. For storage-based attacks we were only able to find examples of rootkits [44–46], memory-based attacks were performed by both root- and bootkit [40–43, 47, 48]. There is no technical limitation as we show in subsection 7.1.1 when we implement our own storage-based bootkit, but more likely a general perference for memory-based attacks as they are more sophisticated. Storage-based attacks face more restrictions such as BitLocker and code integrity checks.

5.2.1 Storage-based

Storage-based attacks need file based access to the Windows installation to modify its content, the primarey partition is NTFS formatted and due to the UEFI specification only mandating compliant firmware to support FAT12, FAT16 and FAT32 [6, Section 13.3.1.1], NTFS drivers are delivered as part of the attack. [45] and [46] seem to use [44]'s leaked NTFS driver. [46] deploys its payload under the file path `C:\Windows\SysWOW64\autoche.exe` and then modifies the registry entry `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute`, so that their payload is executed instead of the original executable. [45] simply deploys their payload in the Windows startup folder, whose contents, as its names suggests, are executed upon startup.

5.2.2 Memory-based

It seems to be unique to [41] to patch out the integrity self-check of the Windows Boot Manager, as it is the only bootkit to change the bootloader on disk instead of in memory. [40, 42] when executed load bootmgfw.efi into memory and apply patches before launching it. [43]'s core functionality is the same for its root- and bootkit variant. A DXE driver is loaded, either form the DXE dispatcher or through an intermediary loader application. This driver then hooks the UEFI boot service `LoadImage()`. When this is either called by the UEFI boot manager or the loader application to load bootmgfw.efi, it patches the bootloader in memory [43]. [47] applies its patches within an `ExitBootServices()` hook.

The general approach is the same for all memory-based attacks, they propagate their malicious execution further up in the boot chain, by hooking when images are loaded. From bootmgfw.efi to Winload.efi to ntoskernel.exe, the kernel image.

Some attacks patch the kernel to disable Windows Driver signing and then install a kernel driver [41, 43]. Others deploy payload with elevated privileges [40, 42] or map code directly into kernel space [47, 48].

6 Test Setup

We perform our attacks against Windows 11 on three different setups. Even though all three UEFI firmwares are PI specification compliant, there is still a lot of freedom for OEMs, when implementing the PF.

6.1 QEMU

Our main development setup is an emulated environment using the emulator Quick Emulator (QEMU) [49] together with the OVMF image, from EDK II using version `edk2-stable202208`. For Secure Boot we generate our own PK and use the *Microsoft Corporation KEK CA 2011* and the two signature DBs *Microsoft Windows Production PCA 2011* and *Microsoft Corporation UEFI CA 2011* provided by Microsoft. For BitLocker and to fullfill Windows 11's general requirement that a TPM 2.0 is present, we use swtpm. A TPM emulated in software [50]. Accessing the firmware image with this setup is done through simple file access.

6.2 Lenovo Ideapad 5 Pro-16ACH6

Our first hardware setup is a Lenovo Ideapad 5 Pro-16ACH6, an Advanced Micro Devices (AMD) based laptop with a AMD Ryzen 9 5900H [51]. It supports Microsoft Device Guard and PCR7 binding when Secure Boot is enabled. We can read and write the firmware image with physical hardware access by using an SPI flash programmer in combination with the Linux utility flashrom.

6.3 ASRock A520M-HVS

Our second hardware setup is a desktop PC with an AMD based ASRock A520M-HVS motherboard. We use the lastest firmware image as of writing which is of version 2.30. This setup also supports PCR7 binding when Secure Boot is enabled. The SPI

36

flash chip on the main board is disabled and instead an EM100 SPI flash emulator is attached. When the system thinks it is accessing the SPI flash chip, it is instead communicating with the emulator. Additionally by dualbooting into Linux we can use flashrom to communicate with the SPI chip for read and write access. The Linux distribution of our choice, Ubuntu, uses a so called Lockdown Mode when Secure Boot is enabled. This blocks direct access to the SPI chip [52], but can be disabled while Secure Boot remains enabled [53].

Attacks

7

We implement our own storage-based UEFI attacks in three different scenarios with increasing levels of security mechanisms. The first attack is with Secure Boot and BitLocker disabled, the second attack with Secure Boot enabled and the third attack with both Secure Boot and BitLocker enabled.

7.1 Neither Secure Boot nor BitLocker

We start by implementing a baseline attack, that we can use to test against Secure Boot and BitLocker. We implement it in the form of a bootkit and a rootkit, with both sharing the same approach and core functionality. The general approach of our attack is to access the hard drive containing the Windows installation from within UEFI environment. We gain then code execution with elevated privileges in the Windows environment by modifying its content.

7.1.1 Bootkit

We start with the bootkit.

7.1.1.1. Infection

We have two ways to infect a system, we can either use a bootable medium such as a CD-ROM or USB stick with a UEFI application installing the bootkit or using a Windows executable. Booting into the installer application requires either the firmware implementation or the boot order to prefer booting from the removable media over Windows. This can be forced when booting accessing the interactive firmware menu at startup, given that it is not password protected. Installation from Windows requires admin privileges to mount and modify the ESP.

The installation process is identical for both options, we access the ESP and create a copy of the Windows Boot Manager located under EFI\\Microsoft\\Boot\\bootmgfw.efi.

We then replace the original with our bootkit as well as dropping all resources required by the bootkit on the ESP. Now that our bootkit is in place of the Windows Boot Manager, when the UEFI Boot Manager selects the boot load option for the Windows Boot Manager, it will cause our bootkit to be executed. Figure 7.1 shows a dump of the Windows boot entry using the UEFI shell command bcfg. The entry contains the device path including the file path and optional data.

```
Shell> bcfg boot dump -v
Option: 00. Variable: Boot0008
  Desc     - Windows Boot Manager
  DevPath  - HD(1,GPT,1AB4CADF-0F69-4B05-8E16-C2803309F223,0x800,0x32000)/\EFI\
Microsoft\Boot\bootmgfw.efi
  Optional- Y
  00000000: 57 49 4E 44 4F 57 53 00-01 00 00 00 88 00 00 00  *WINDOWS.........*
  00000010: 78 00 00 00 42 00 43 00-44 00 4F 00 42 00 4A 00  *x...B.C.D.O.B.J.*
  00000020: 45 00 43 00 54 00 3D 00-7B 00 39 00 64 00 65 00  *E.C.T.=.{.9.d.e.*
  00000030: 61 00 38 00 36 00 32 00-63 00 2D 00 35 00 63 00  *a.8.6.2.c.-.5.c.*
  00000040: 64 00 64 00 2D 00 34 00-65 00 37 00 30 00 2D 00  *d.d.-.4.e.7.0.-.*
  00000050: 61 00 63 00 63 00 31 00-2D 00 66 00 33 00 32 00  *a.c.c.1.-.f.3.2.*
  00000060: 62 00 33 00 34 00 34 00-64 00 34 00 37 00 39 00  *b.3.4.4.d.4.7.9.*
  00000070: 35 00 7D 00 00 00 61 00-01 00 00 00 10 00 00 00  *5.}...a.........*
  00000080: 04 00 00 00 7F FF 04 00-                          *........*
```

Fig. 7.1.: Windows boot entry, part of the UEFI shell output of bcfg

7.1.1.2. File access

The most important step for a storage-based approach is gaining access to the Windows installation from within the UEFI environment. Since UEFI does not require the firmware to come with an NTFS driver, our attack has to come with its own. EDK II does not provide one, but we can use the open source NTFS driver ntfs−3g from Tuxera [54]. It was ported to the UEFI environment by *pbatard* [55]. Using EDK II to compile it, we receive a .efi executable image.

We can use the UEFI shell and its file system related commands to test the NTFS driver's capabilities. When booting into the UEFI shell we are greeted with a screen displaying the UEFI specification version the firmware supports and a list of default mappings for file systems and block devices. These mappings are created by the shell to provide a short name that can be used interchangably with longer device path when issuing commands [16, Section 3.7.2]. They are designed to be consistent across reboots as long as the hardware configuration stays the same and are comparable to Windows partition letters [16, Appendix A]. Figure 7.2 shows the mapping of a partition containing a Windows installation. As there is no NTFS driver present yet, it is only displayed as a block device.

Fig. 7.2.: Mapping of the Windows parition

We can enter the mapping name of the file system containing our NTFS driver to use it as our current working directory and load the driver using the load command. The command is executed successfully and can the driver is now listed when querying the currently loaded drivers with the command drivers. We can now instruct the default mappings to be reset with the map -r command, to receive an updated list including the file systems now provided through the NTFS driver. Figure 7.3 also shows us that the new file system now sit on top of a device which previously was only listed as a block device.

Fig. 7.3.: Mapping of the Windows parition after loading the NTFS driver

As done before we now type the mapping name of the new file systems, we check the root directories' contents with ls until we find the partition containing the Windows installation and then enter vol to check the access rights. This reveals that the file system is currently read-only. Upon debugging the NTFS driver it appears to be that the driver falls back to read-only when it encounters a file that indicates that the Windows system is in hibernation mode. We can remove this fallback from the NTFS driver's code and recompile.

On our hardware setups we noticed that the firmware can already ship with an NTFS driver included, which is read-only. In the case of our rootkit we would be able to remove this driver by modifying the firmware iamge, but we can implement a solution that applies to both types of UEFI payload. We can change the NTFS driver to install its Simple File System Protocol under a different GUID instead of `gEfiSimpleFileSystemProtocolGuid`. This makes it possible to install two instances of the Simple File System Protocol alongside each other on the same controller. The alternative GUID can then be used in our root- and bookit, to retrieve our specific protocol instance with write access. The driver also has to open the protocols it uses without demanding exclusive ownership over them. This also prevents the NTFS driver from being when trying to open a protocols that is already in exclusive ownership [6, Section 7.3], which would be a likely scenario as filesystem drivers are encouraged to get exclusive control over their block device [6, Section 13.5].

We now know that provided we get to load the NTFS driver we can access the data contained within a Windows installation with read and write. [TODO WEITERMACHEN] Since our rootkit will not use the UEFI shell we need to have the NTFS driver load as part of the boot process.

The next step is for our bootkit to use the NTFS driver to gain file system access and write our payload to the Windows installation. During our bootkit infection process we place the NTFS driver on the ESP, so that our bootkit can load it. In our bootkit, we can use the Loaded Image Protocol, that is installed to the handle of the bootkit's image in memory to retrieve the handle of the device our bootkit was loaded from [6, Section 9.1]. This handle can then be used to call the Boot Services LoadImage and StartImage to load and execute the NTFS driver. Since the driver conforms to the UEFI Driver Model, we need to also reconnect all controllers recursively, so it can assume controller over the NTFS formatted volumes, by installing the Simple File System Protocol on their handles. Loading the payload and other non-executable files into memory is done differently, here we use the handle from the Loaded Image Protocol to open the Simple File System Protocol installed onto the ESP, we can then call the OpenVolume resulting in an instance of the File Protocol representing the root folder of the volume [6, Section 13.4]. This instance can then be used to open and read our payload with the absolute path on the ESP into memory.

7.1.1.3. Payload deployment

To perform the write operation we now need a handle we did not yet interact with, at least directly. We can use the Boot Service LocateHandleBuffer to receive an array of all handles that support the Simple File System Protocol, this includes volumes such as the ESP but also the Windows recovery partition. We can iterate over all handles to open the volume and attempting to create a new file with a file path that's inside of the Windows installation. This operation fails on volumes not containing a Windows installation which we can just skip. Eventually the volume containing Windows is found and the file is created and opened successfully, we can then write our payload, that we read into memory earlier, onto the disk and close the file again.

Now the question arises as to where to write our payload to, we want automatic and elevated execution. Earlier we discovered that the NTFS DXE driver disregards the file access permission model [TODO Windows File Permissions] so we are not restricted in the same way an unprivileged user would be when accessing the disk. *MosaicRegressor* writes its payload to the Windows startup folder, a folder whose contents are automatically executed at system startup. The programs within the startup folder are unfortunately not automatically run at an elevated level, so this isn't a suitable target location.

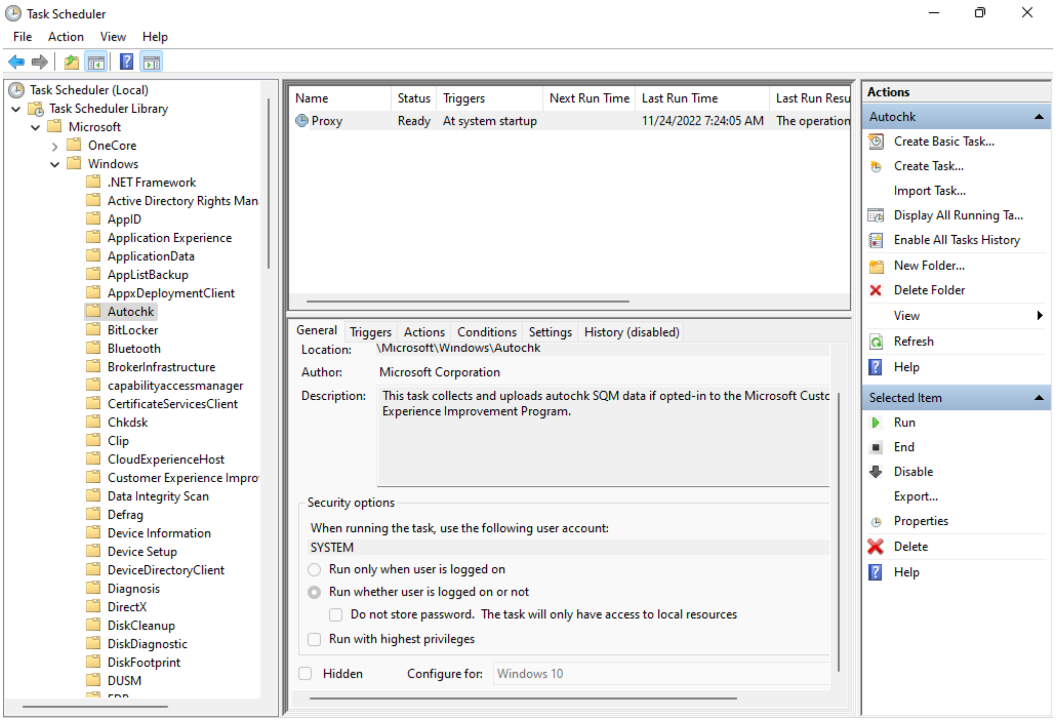
[TODO DLL proxy loading] [TODO modifying Windows Executables KMCI]

The Task Scheduler is a Windows service responsible for managing the automatic execution of background tasks [25, Section 10]. Tasks are performed on certain triggers, which may be time-based (periodically or on a specific time) or event-based, for example on user logon or system boot [56]. A task can perform various actions upon invocation [57], but we will focus on command execution. Most tasks will simply execute other programs as their action, this execution is performed under specified a security context [58]. The idea of our attack is to have a task, that performs its action with a high privilege level, execute our payload. The task of our choosing is called Autochk\Proxy, that performs the command

```
%windir%\system32\rundll32.exe /d acproxy.dll,PerformAutochkOperations
```

30 minutes after system boot, the executable rundll32.exe loads the Dynamically Linked Library (DLL) acproxy.dll and invokes the exported function PerformAutochkOperations [59]. The function name as well as the task name suggest the performed action relates to the Windows utility *autochk* which verifies the integrity of NTFS file systems [60]. The Task Scheduler keeps book of its active tasks in the registry under HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Schedule\TaskCache, grouped by four subkeys Boot, logon, plain and Maintenance. These entries consist only of a GUID that is used to look up the task descriptor saved under their respective task master (registry) keys, these task master keys are located under HKLM\SOFTWARE\Microsoft\Windows N [25, Section 10]. There also exist a secondary copy of the task descriptors, on the regular file system under %windir%\system32\Task, stored as Extensible Markup Language (XML) files.

We can use the Task Scheduler Configuration Tool to modify the target task on a system under our control, we change the executable path as well as remove the configured delay. We then use the Windows registry editor reged.exe to navigate to the task descriptor store, there we search for the task master key belonging to our task and export this key.



To verify the privileges our payload is executed with, we can save the output of whoami /all into a file. The whoami command shows the current user and privileges [61]. After manually triggering the task through the configuration tool, we see that our payload was run from the nt authority\system user account, which is the most privileged system account [62].

[TODO whoami /all snippet]

We can use this exported key and import it on our victim's system as part of our attack. This way, instead of modifying a single value of the registry key, the victim's key maintains its integrity as we also overwrite the hash value with correct data. To import the key on an offline system, we can use a Linux utility called chntpw whose primary purpose it is to reset the password of local Windows user accounts [63]. The library does this by editing the registry of a Windows installation and as such the author also offers a standalone registry editor called reged. We can test the Linux tool when dual-booting a Linux and a Windows installation. We place our payload in the Windows installation and then boot into Linux, where we can open the HKEY_LOCAL_MACHINE/SOFTWARE hive in reged and import our modified registry key. This overwrite the task descriptor and when booting into Windows our payload is executed.

The next step is to port the reged utility so that it works in the UEFI environment, so we can use it as part of our bootkit. The porting process boils down to providing semantically equivalent definitions of external function calls, such as c standard library and Linux kernel functions, to link against. Declarations and macros are still supplied by the local compiler's system headers. Function definitions can often be translated to UEFI equivalents, EDK II has libraries offering implementations of commonly used abstraction. Memory allocation maps to the MemoryAllocationLib, memory manipulation to BaseMemoryLib, basic string manipulation to BaseLib, stdout to PrintLib (only relevant for print debugging). Function calls related to standard input and output such as opening, reading and writing a file, namely the hive file, are more complex and have to be mapped to the UEFI protocols Simple File System Protocol and File Protocol. Luckily the author of reged used distinct functions to access the hive file and registry file, making it possible to keep the original source code unmodified, except for a change in the import behavior. The name of a task master key is the task's GUID, which may differ from device to device, thus we cannot import a key into its exact path, we instead iterate over the subkeys of the target's parent key. We then match for the name value of the key.

Now that we modified the Windows installation to execute our payload upon boot, we need to transfer execution from the bootkit to the original Windows Boot Manager. Loading the original application is inspired by how the UEFI Boot Manger loads boot options, this includes relaying the LoadOptions and ParentHandle of the *Extensible Firmware Interface (EFI) Loaded Image Protocol* [6, Section 9.1] instance installed to our bootkit to the Windows Boot Manager.

7.1.2 Rootkit

Performing the same attack in the form of a rootkit is very similar and mainly differs in the infection process. The UEFI payload is now compiled as a DXE driver instead of an application. When placed in the DXE volume it is automatically loaded by the DXE Dispatcher iterating over the FV, loading drivers whose dependencies are resolved. The core functionality of our UEFI payload is identical with the exception that we don't have to manually load the NTFS driver anymore and accessing the Windows payload is now done with the *Firmware Volume2 Protocol* defined in the [10, Section 3.4.1], instead of *Simple Filesystem Protocol*. There are no traditional file names on a firmware volume, and we have to search for files using the module GUIDs.

7.1.2.1. Infection

Infection with the rootkit is has a much higher barrier of entry, as it requires read and write access to the firmware image, which often requires physical access. chapter 6 potentially exploit OEM specific flash mechanism, signing with stolen private key, part of the supply chain, might also be physical **[TODO LIST ALL OPTIONS]**

We have to retrieve the image, insert our payload into a DXE volume and deploy the modified image. In UEFITool we navigate to the DXE Volume containing the DXE Core and DXE drivers. We cannot directly drop our UEFI payload in form of .efi files with UEFITool, because DXE drivers have three mandatory sections: the PE32 executable section, composed of the .efi file content, a version section and the DEPEX section [10, Vol. 3, 2.1.4.1.4]. For our UEFI payload to be generated as a sectioned FFS file we add our files to the build process of OVMF package in EDK II. When part of the Flash Description File (FDF) which is used to generate a firmware image file, the intermediary .ffs files from the build process are of much value for us. For our Windows payload we can use a special EDK II module type which takes binary files as input, resulting in a sectioned file of type EFI_FV_FILETYPE_FREEFORM, with no restrictions on the contained file sections [10, Vol. 3, 2.1.4.1.7]. The output contains only one file section of type EFI_SECTION_RAW consisting of the binary payload. This use of this special module has the benefit that its GUID is used to attribute the sectioned file when being placed in the firmware volume. Not that we have .ffs files corresponding to all our resources used in the attack we can import these into the target image with UEFITool.

[TODO this] overwrite the SPI flash with modified image by using the programmer again.

7.2 Secure Boot

Our second attack is performed against systems which have Secure Boot enabled without any complementary Hardware Validated Boot. We assume that the signature DBs used do not contain our images hashes and that the interactive UEFI setup menu is password protected. Otherwise we could simply turn off Secure Boot.

7.2.1 Bootkit

The interactive menu being password-protected makes the likelihood of an infection via booting into our installer smaller. We now solely rely on the boot order/firmware policy to prefer removable media. Even if this was to be the case, we promptly see that Secure Boot already denies execution of the installer when trying to boot it. When using our Windows installer we observe the same denial for the bootkit itself. The Windows Boot Manager boot option pointing to our bootkit is now denied execution, if we were to have overwritten the standard boot entry of the hard drive EFI\Boot\bootx64.efi, a copy of the Windows Boot Manager, Windows would now be rendered unbootable.

7.2.2 Rootkit

In subsection 2.2.3 we discussed how the PI specification defines the usage of its two security architectural protocols, with them being required to be invoked on every call to `LoadImage`. As `LoadImage` is used internally within the DXE dispatcher the security protocol invocations also apply to our rootkit's DXE drivers when being loaded. We also discussed in subsubsection 2.1.10.1, that Secure Boot relies on the firmware image as its root of trust, where Secure Boot is inherntly unable to verify the behavior of the PI process. Now these two seem to be conflicting, but when we deploy our rootkit it is unaffected by Secure Boot. When we look at the reference implementation in EDK II, we can see why. Listing 7.1 shows a snippet of the function that is used to implement the Securit2 protocol. It shows that the image origin dictates which policy is being applied. The standard policy for images from a Firmware Volume (FV) (`IMAGE_FROM_FV`) is to always allow execution. This aligns with what the UEFI specification says on Secure Boot Firmware Policy: "The firmware may approve UEFI images for other reasons than those specified here. For example: whether the image is in the system flash [. . .]" [6, p. 32.5.3.2]. This behavior was reproducible on all our test setups.

```
switch (GetImageType(File))
{
case IMAGE_FROM_FV:
    Policy = ALWAYS_EXECUTE;
    break;

case IMAGE_FROM_OPTION_ROM:
    Policy = PcdGet32(PcdOptionRomImageVerificationPolicy);
    break;

case IMAGE_FROM_REMOVABLE_MEDIA:
    Policy = PcdGet32(PcdRemovableMediaImageVerificationPolicy);
    break;

case IMAGE_FROM_FIXED_MEDIA:
    Policy = PcdGet32(PcdFixedMediaImageVerificationPolicy);
    break;

default:
    Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
    break;
}
```

Listing 7.1: Policy Selection in DxeImageVerificationHandler (EDK II reference implementation of Security2 Architecture Protocol)

7.3 BitLocker

Our final attack will target systems using BitLocker FVE with a TPM 2.0 and no additional PIN or startup key configured. This leaves the Windows boot partition encrypted, the ESP is remains unencrypted, thus not affecting the bootkit installation process. Secure Boot can be enabled in combination of BitLocker having the effects as observed in section 7.2, as well as additionally dictating the BitLocker default validation profile Windows uses as mentioned in subsection 4.3.3. We perform our attack against both default profiles, starting with `{0, 2, 4, 11}`. This means either Secure Boot is disabled or PCR7 is not bound, because of the presence of a signature DB other than *Microsoft Windows Production PCA 2011*. The default validation profile `{7, 11}` used, when Secure Boot takes care of integrity validation is covered in subsection 7.3.4. Due to the boot- and rootkit still sharing their core functionality we keep the approach abstract and make no further distinctions between the two. We refer to them with the expression UEFI payload, not to be confused with our (Windows) payload that is deployed in the Windows installation. For the most part of this attack we assume, that the infection is performed after BitLocker has been fully set up, only briefly touching the scenario of a user enabling BitLocker while being infected.

7.3.1 Infection

When booting with our previous UEFI payload, the NTFS driver is unable to recognize any file system structure on the Windows boot partition, due to the FVE. Resulting in an inability to further deploy the Windows payload on the target system. Additionally, during execution of the Windows Boot Manager, the BitLocker recovery prompt, shown in Figure 7.4, interrupts the regular boot process requiring the drive's recovery key for decryption before being able to continue booting. This happens due to TPM's PCR values differing from what was initially used to seal the VMK, leaving the Windows bootloader unable to retrieve the unencrypted VMK from and as a result unable to decrypt the Windows installation [25, Section 12].

BitLocker with TPM measurements successfully mitigates UEFI attacks and maintains system integrity by discovering deviations in the boot flow. But how does the user react to this, after all it is asking them to enter the recovery key to resume the boot process and not throw out their motherboard. There are a few options for a user to proceed: they either trust the system and enter their recovery key, mistrust the operating system or mistrust the entire system. If they were to mistrust the OS,

Fig. 7.4.: BitLocker Recovery Prompt

or they were to have neglected to properly back up their recovery key, they might perform a fresh installation. In the case of our bootkit this gets rid of the threat, but the rootkit remains in the firmware image and would be part of the chain of trust for the fresh installation. If they were to mistrust the whole system, they could recover data from the drive with another system, being careful not to accidentally boot from the drive. This would deny both our rootkit and bootkit any further access to any sensitive data. **[TODO maybe change order]**

7.3.2 BitLogger

When the user enters their recovery key the Windows Boot Manager uses the recovery key to decrypt the VMK metadata entry, that was encrypted using the recovery key when BitLocker was set up. It then proceeds to access the bitlocked NTFS drive containing the Windload.efi OS loader. This all still happens during the UEFI boot environment, before ExitBootServices is called. Unfortunately we are still unable to access the Windows installation, as BitLocker only ever decrypts read operations in memory, leaving the drive fully encrypted at all times. If we were to acquire the recovery key, we could use it to decrypt the VMK, the FVEK and in turn the drive ourselves.

We can achieve this by logging the keystrokes performed by a user entering the key in the recovery prompt. Since we still are in the UEFI boot environment, the Windows Boot Manager uses UEFI protocols for user input instead of the own Windows drivers. UEFI offers two protocols for this purpose the *Simple Text Input Protocol* and the Simple Text Input Ex Protocol, we can quickly determine which of these is used by the Windows Boot manager by adding a simple Print statement to the implementation in the OVMF source code, this change also is enough to trigger the recovery prompt by invalidating the PCR measurements. A keystroke now shows us that the Simple Text Input Ex Protocol is being used, the protocol structure is depicted in Listing A.6. The Windows Boot Manager uses the ReadKeyStrokeEx function to retrieve the latest pending key press. The protocol also offers the WaitForKeyEx event, signaling when keystrokes are available, execution can be blocked until this event is emitted with the WaitForEvent Boot service. Example usage of the protocol can be seen in Listing 7.2.

```
EFI_STATUS EFIAPI EntryPoint(IN EFI_HANDLE ImageHandle,
                             IN EFI_SYSTEM_TABLE *SystemTable)
{
    gBS = SystemTable->BootServices;

    EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;

    gBS->HandleProtocol(SystemTable->ConsoleInHandle,
                        &gEfiSimpleTextInputExProtocolGuid,
                        (VOID **)&SimpleTextInEx);

    UINTN EventIndex;
    gBS->WaitForEvent(1, &SimpleTextInEx->WaitForKeyEx, &EventIndex);

    EFI_KEY_DATA KeyData;
    SimpleTextInEx->ReadKeyStrokeEx(SimpleTextInEx, &KeyData);

    // do something with key press

    return EFI_SUCCESS;
}
```

Listing 7.2: Example of using *HandleProtocol* to retrieve an instance to the *Simple Text Input Ex Protocol* to use its *ReadKeyStrokeEx* function to wait for and read a pending key press

We can intercept the ReadKeyStrokeEx function call by using a technique called function hooking, there are various ways of doing this, for example patching a jump instruction at the beginning of the target function to detour the execution flow. But UEFI protocol hooking does not require such an invasive and unportable technique. When we take a closer look at how protocols are returned to their user we can see why. The UEFI Boot Services offer two functions, HandleProtocol and OpenProtocol, that can be used to retrieve a protocol instance. HandleProtocol is a simplified abstraction of OpenProtocol and is implemented by the latter internally. OpenProtocol offers many additional options such as keeping track of the protocol users and exclusivity [6, Section 7.3]. Listing 7.2 shows how HandleProtocol can be used to receive the Simple Text Input Ex Protocol instance installed on the active console input device [6, Section 4.3]. The input parameters are a device handle, the GUID identifying the protocol and the address of a pointer to the protocol structure. When calling HandleProtocol the value of the pointer is modified to point to the corresponding protocol instance. The protocol instance itself is previously allocated by a driver and installed onto the device handle in their Driver Binding Start function **[TODO Driver binding]**. The driver assigns the function fields with functions residing in the driver's image. This is why it is important for a driver's image to remain loaded even after initial execution. The important fact about this process is, that a driver installs only one protocol instance per device handle and every protocol user receives the same address for to the same protocol instance, given they use the same device handle. The function interfaces of HandleProtocol and OpenProtocol would generally allow for the return of allocated memory containing a copy of the protocol's content, but the implementors of drivers managing multiple devices are encouraged to keep track of private data, that is necessary to manage a device, but not part of the protocol interface. This private data struct contains the protocol instance, so that it is then possible to calculate the private data address using the protocol instance's address and the offset of the protocol within the struct [9, Section 8]. In Listing 7.3 we show an example of retrieving private data through the public protocol interface. This keeps the protocol interface clean and limited to the public functionality, but the UEFI boot services don't know about the size of the private data when managing protocol instances and therefor cannot make copies spanning the entire data. On top of that, private data likely contains information about the device state, changes in the state would have to occur in all instances of each protocol user instead, this would defeat the concept of private data.

```
typedef struct
{
    UINTN Signature;
    EFI_DISK_IO_PROTOCOL DiskIo;
```

```
    EFI_BLOCK_IO_PROTOCOL *BlockIo;
} DISK_IO_PRIVATE_DATA;

EFI_STATUS EFIAPI DiskIoReadDisk(IN EFI_DISK_IO_PROTOCOL *This,
                                 IN UINT32 MediaId,
                                 IN UINT64 Offset,
                                 IN UINTN BufferSize,
                                 OUT VOID *Buffer)
{
    DISK_IO_PRIVATE_DATA *Private;

    Private = DISK_IO_PRIVATE_DATA_FROM_THIS(This);

    Private->BlockIo->ReadBlocks(...);
}
```

Listing 7.3: Example of a driver using private data in the implementation of the *Disk I/O Protocol* (snippets from [9, Sections 8.2 and 8.5])

Since our UEFI payload is executed before the Windows Boot Manager we can query all instances of the Simple Text Input Ex Protocol and change the function pointer of ReadKeyStrokeEx to point to our function hook. When a user later receives a pointer to the protocol instance, accessing the ReadKeyStrokeEx field will cause our hook to be called instead of the original function. The hook has to be implemented in a driver, so that it remains loaded until the Windows Boot Manager uses ReadKeyStrokeEx. We also have to save the original function address, together with a pointer to the protocol instance, so that we can call it later. Multiple different drivers could offer the same protocol, resulting in different functions being called depending on the device, the protocol instance is retrieved from. When our hook is called we start by identifying which original function needs to be called using the protocol instance that is used as the first argument of the ReadKeyStrokeEx function signature. We then call the original to read the pending keystroke, keeping track of the keystrokes (separately for each protocol instance), before returning the key data back to the caller. We coin this BitLocker specific keylogger *BitLogger*. A simplified version of how the hooking process works can be seen in Listing 7.4;

```
EFI_STATUS EFIAPI ReadKeyStrokeExHook(IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
                                      OUT EFI_KEY_DATA *KeyData);
{
    SimpleTextInputExHook *Hook = GetHookFromProtocol(this);
    Hook->ReadKeyStrokeExOriginal(This, KeyData);

    // log keystrokes
```

```
}

VOID HookSimpleTextInEx()
{
    gBS->LocateHandleBuffer(ByProtocol, &gEfiSimpleTextInputExProtocolGuid,
                            NULL, &HandleCount, Handles);

    gHooks = AllocatePool(HandleCount * sizeof(SimpleTextInputExHook));

    for (UINTN i = 0; i < HandleCount; ++i)
    {
        EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;
        status = gBS->HandleProtocol(Handles[i],
                                     &gEfiSimpleTextInputExProtocolGuid,
                                     (VOID **)&SimpleTextInEx);

        SimpleTextInputExHook *Hook = &gHooks[gHookCount++];
        Hook->SimpleTextInEx = SimpleTextInEx;
        Hook->ReadKeyStrokeExOriginal = SimpleTextInEx->ReadKeyStrokeEx;

        SimpleTextInEx->ReadKeyStrokeEx = ReadKeyStrokeExHook;
    }
}
```

Listing 7.4: Simplified example of hooking the *Simple Text Input Ex Protocol*

We want to use the recovery key programmatically, so we can't simply log all key presses in chronological order and evaluate them by hand later. The BitLocker recovery prompt has a few rules and does not allow the user to just input any possible combination of digits, each entered block is checked for validity before allowing the cursor to advance to another block, this also applies when moving the cursor backwards to a previously entered block, while incomplete blocks are not evaluated. Each block must be divisible by 11 [22, Section 9]. For this reason and because the cursor can be used to increment and decrement the current digit by using the up and down arrow keys, we have to implement internal tracking of the cursor advancement. The recovery prompt in Figure 7.4 also tells us, that the function keys (F1-F10) are accepted as input, with F10 mapping to zero, so we have to log these key presses as well.

7.3.3 Dislocker

To make use of the recovery key we can use an open source software called *Dislocker*, which implements the Filesystem in Userspace (FUSE) interface to offer mounting of BitLocker encrypted partitions under Linux supporting read and write access [64].

In subsection 4.3.3 we discussed, how the BitLocker filter driver integrates into the Windows. To integrate Dislocker into UEFI start by analyzing how the NTFS driver works. We can start by checking the .inf file of the driver, which declares which protocol GUIDs are consumed and produced by the driver. Listing 7.5

```
[Protocols]
  gEfiDiskIoProtocolGuid
  gEfiDiskIo2ProtocolGuid
  gEfiBlockIoProtocolGuid
  gEfiBlockIo2ProtocolGuid
  gEfiSimpleFileSystemProtocolGuid
  gEfiUnicodeCollationProtocolGuid
  gEfiUnicodeCollation2ProtocolGuid
  gEfiDevicePathToTextProtocolGuid
```

Listing 7.5: Protocols section of NTFS driver's module file

We can ignore the last three protocols as they are not directly involved in media access. The Simple File System Protocol is produced by the driver, as it installs the protocol onto handles of devices it supports. So the only relevant protocols it consumes are the *Disk I/O Protocol* and the *Block I/O Protocol* as well as their respective asynchronous counterparts marked by the trailing 2. We will ignore the asynchronous protocols, as they only serve to further abstract their synchronous version [6, Sections 13.8 and 13.10]. The same could be said for the *Disk I/O Protocol*, as it abstract the *Block I/O Protocol* to offer an offset-length driven continous access to the underlying block device [6, Section 13.7], but this is the protocol primarily used by the driver and the *Block I/O Protocol* is only used directly to retrieve volume and block size, as well as read the first block to determine whether the volume is NTFS formatted. Keeping in mind the fact, that the Simple File System Protocol is only used to open a volume and any further access to the volume is done through the File Protocol. It becomes obvious that all file-wise operations are, is multiple layers of abstraction on top of block-wise access to the underlying media, performed through the *Block I/O Protocol*. Inspired by the BitLocker filter driver in Figure 4.2, which de- and encrypts each block as it passes through, we hook the *Block I/O Protocol* functions ReadBlocks and WriteBlocks, their signatures are shown in Listing A.9. We can then use Dislocker on read and write operations to implement our own filter driver as shown in Figure 7.5.

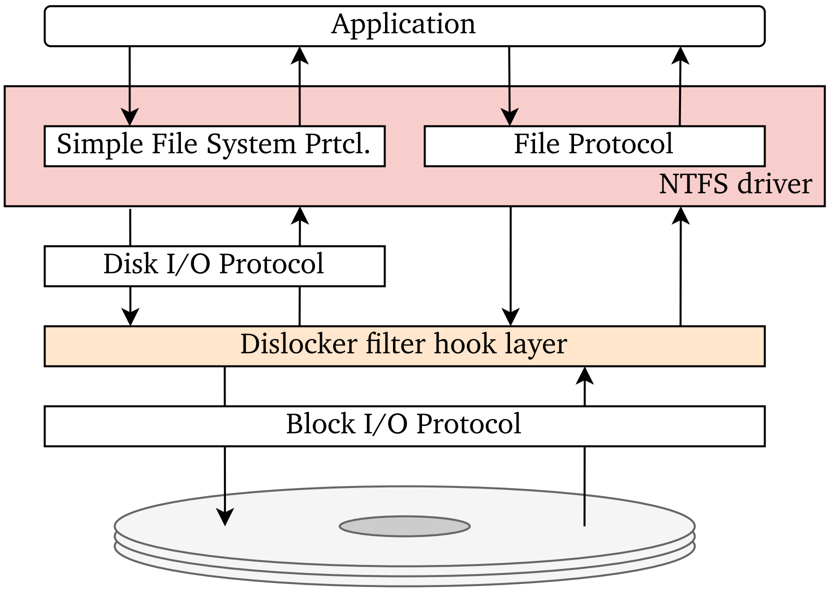


Fig. 7.5.: Dislocker Volume Access Protocol Stack

When we look at the Dislocker source code, we find that Dislocker works with two main functions dislock and enlock, they each take offset-length paremeters, comparable to the *Disk I/O Protocol* abstraction. dislock reads and decrypts, while enlock encrypts and writes. Internally Dislocker uses pread and pwrite to access the volume. These operations are always performed on whole blocks, as BitLocker encryption is done block-wise. So the starting offset is rounded down and the offset plus length is rounded up the to next block boundary. We can map pread and pwrite to call the original ReadBlocks and WriteBlocks functions. Since the two Dislocker functions expect offset-length, we simply multiply the starting block index by the block size to use as starting offset. Listing 7.6

```
EFI_STATUS EFIAPI ReadBlocksHook(IN EFI_BLOCK_IO_PROTOCOL *This,
                                 IN UINT32 MediaId,
                                 IN EFI_LBA Lba,
                                 IN UINTN BufferSize, OUT VOID *Buffer)
{
    BlockIoHook *Hook = GetHookFromProtocol(This);
    int Read = dislock(Hook->dis_ctx, Buffer, Lba * This->Media->BlockSize, BufferSize);
    return Read == BufferSize ? EFI_SUCCESS : RETURN_PROTOCOL_ERROR;
}

EFI_STATUS EFIAPI WriteBlocksHook(IN EFI_BLOCK_IO_PROTOCOL *This,
                                  IN UINT32 MediaId,
```

```
                                      IN EFI_LBA Lba,
                                      IN UINTN BufferSize, IN VOID *Buffer)
{
    BlockIoHook *Hook = GetHookFromProtocol(This);
    int Read = enlock(Hook->dis_ctx, Buffer, Lba * This->Media->BlockSize, BufferSize);
    return Read == BufferSize ? EFI_SUCCESS : RETURN_PROTOCOL_ERROR;
}

ssize_t pread(int fd, void *buf, size_t nbytes, off_t offset)
{
    BlockIoHook *Hook = GetHookDataFromIndex(fd);
    Hook->ReadBlocksOriginal(hook->protocol,
                             BlockIo->Media->MediaId,
                             offset / Hook->BlockIo->Media->BlockSize,
                             nbytes, buf);
    return nbytes;
}

ssize_t pwrite(int fd, const void *buf, size_t nbytes, off_t offset)
{
    BlockIoHook *Hook = GetHookDataFromIndex(fd);
    Hook->WriteBlocksOriginal(hook->protocol,
                              BlockIo->Media->MediaId,
                              offset / Hook->BlockIo->Media->BlockSize,
                              nbytes, (void *)buf);
    return nbytes;
}
```

Listing 7.6: Simplified example of hooking the *Block I/O Protocol*

For the previous two attacks the timing of deploying the payload did not matter, as long as it was done before Windows loads the HKLM\SOFTWARE registry hive, thus performing the deployment as soon as the UEFI payload is executed suffices, as this happens before any Windows boot related actions are performed. With BitLocker we have to deploy after our BitLogger was able to obtain the recovery key. After initializing Dislocker with the recovery key we enable the transparent *Block I/O Protocol* hook layer, so we can trigger the NTFS driver to (re-)evaluate which device handles it supports. The BitLocker encrypted drive now appearing unencrypted allows the driver to install its Simple File System Protocol instance. This allows us to deploy the payload and import our modified registry key. After doing this we need to disable the Dislocker layer again, as otherwise Windows is unable to boot and instead attempts Windows recovery, showing a second recovery prompt, but now outside the UEFI environment with their own device drivers. This recovery environment is located on the unencrypted NTFS partition created during installation and also accessible when pressing the escape key during the initial UEFI environment recovery prompt. We want to prevent the user from using this prompt instead of the UEFI prompt, as our *BitLogger* would not be able to obtain the recovery key. This can be done in our ReadKeyStrokeEx hook, where when a user presses escape we instead return another key to the Windows Boot Manager.

If we were to attack Windows 10 we would be done now, but Windows 11 will show the recovery prompt every boot. Windows 10 seems to automatically reseal the VMK, whereas Windows 11 doesn't, so our UEFI payload keeps invalidating the PCR values. We can add a few calls to the BitLocker management tool manage—bde [65] within our Windows payload, deleting the old TPM protector and adding a new one. Now our UEFI payload is part of the measurements and considered trusted. Execution does not trigger the BitLocker recovery prompt anymore.

7.3.4 BitLocker Access without Recovery Prompt

When either the BitLocker validation profile is misconfigured (for example {7, 11}) or the TPM protector already includes our UEFI payload in its PCR measurements, the TPM yields the Windows Boot Manager the unencrypted VMK with which it is able access the drive. We are unable to receive a recovery key as none has to be entered and in turn we cannot decrypt the drive. In the case of our own TPM protector update, we could simply save the recovery key in an unencrypted region of the drive, but there is a solution which does not require any prior knowledge about the recovery key.

Under these circumstances execution of our UEFI payload does not influence the outcome of the interaction between the Windows boot manager and the TPM when unsealing the encrypted VMK. This allows us to be a spectator without interrupting the process. [66, 67] have proven the viability of sniffing the TPM communication directly off the hardware bus. Inspired by this we can hook the *TCG2 Protocol* that is used to abstract TPM communication for the UEFI environment. The function `SubmitCommand()` is used to perform request the unseal operation. We can hook this function and filter for the correct command submission using the header structures of the request and response. We then extract the VMK from the command response as done in [67]. The unencrypted VMK can then be simply passed to dislocker for the initialization of hard drive decryption.

This increases the persistence and applicability of our attack immensely.

8 Results

We were able to implement UEFI attacks in the form of a UEFI firmware rootkit and a UEFI bootloader rootkit (bootkit). Both were able to deploy Windows level payload from within the UEFI environment using an NTFS drivers. By porting the Linux utility reged to UEFI we were able to modify the Windows Registry. Through the manipulation of a Task Scheduler master registry key we were able to have our Windows payload executed with the privileges of the built-in local system account. The execution happens at system boot before login.

We showed that the attack using our rootkit is unaffected by Secure Boot, as Secure Boot relies on the PF as its root of trust. This leaves Secure Boot unable to verify the behavior of the PI, including our rootkit. As Secure Boot's threat model consists of the hand-over process from the plaftorm firmware to UEFI images, the attack through our bootkit is successfully mitigated.

When the target system uses BDE with a TPM 2.0 and the default validation profile `{0, 2, 4, 11}` both our root-and bootkit trigger the BitLocker recovery prompt protecting the hard drive from unauthorized access and stopping the boot process. We also show that, Windows giving the user an ability to override the TPM's security reaction, by entering the recovery key, allows us to overcome BDE. Our *BitLogger* was able to log the entered keystrokes to obtain the recovery key. We were then able to use the recovery key with our UEFI port of Dislocker to mount the encrypted drive, allowing us to repeat our initial attack of deploying a payload and modifying the registry.

In the case of our UEFI payload being part of the TPM measurements used to encrypt the VMK or when a validation profile is used that does not include PCR0, we were able to sniff the communication between the TPM and the Windows Boot Manager to retrieve the unencrypted VMK for use with Dislocker. We showed that this is the case when enabling Secure Boot with a configuration that uses only Microsoft's signature DB required to boot Windows. Windows then forces BitLocker to use a default validation profile of `{7, 11}`, leaving out PCR0 where our rookit is measured to. Our rootkit was able to gain access to this type of system without requiring any prior knowledge or additional user input.

9 Discussion

The biggest takeaway of our attacks is, that unintuetively BitLocker-protected Windows 11 installations are likely less secure against UEFI rootkits when Secure Boot is enabled compared to when it is disabled. We disprove Microsoft's statements that "Windows is secure regardless of using TPM profile `{0, 2, 4, 11}` or profile `{7, 11}`" [38] and that "Secure boot ensures that the computer's pre-boot environment loads only firmware that is digitally signed by authorized software publishers." [37]. Excluding PCR0 in the validation profile and relying soley on Secure Boot for integrity validation is a flawed approach, as this is not part of Secure Boot's threat model. Quite the contrary, Secure Boot relies on the firmware as its root of trust, meaning it is not able to validate the behavior of the PI process. Its purpose is to enforce authentiation when the PF interacts with UEFI images outside of the firmware image. Microsoft also states "When this policy [*Allow Secure Boot for integrity validation*] is enabled and the **hardware is capable** of using secure boot for BitLocker scenarios, the *Use enhanced Boot Configuration Data validation profile* group policy setting is ignored" [37], which might refer to Hardware Validated Boot. Hardware Validated Boot moves the root of trust out of the firmware image into hardware and together with Secure Boot would be a sufficient way to replace the early boot component measurements of the TPM for BitLocker. But as we have shown, the policy does not require Hardware Validated Boot. Microsoft's decision to use Secure Boot for integrity validation leaves a lot of systems more vulnerable than they should be. Any attacker with read and write access to the firmware image is able to gain control over such a system. This is especially easy for attackers with physical access, with a stolen laptop being a prime candidate for the attack. We cannot even argue for Microsoft doing this out of a trade-off between security and convenience. It is true that leaving out PCR0 does significantly reduce the chance of the TPM being unable unseal the VMK after a firmware update, making validation profiles containing PCR0 more prone to false positives in comparison. It is also the case that Secure Boot does not enforce its authentication and policies on code that is measured into PCR0. If Microsoft were to have intentionally made this trade-off, Secure Boot would not be the deciding factor on whether to leave out PCR0 or not. A validation profile that maintains a similar security across systems with and without Secure Boot could have been `{2, 4, 11}` when Secure Boot is disabled. With this profile, measurements of the TPM regarding code that Secure Boot reigns over, are used for BitLocker. Microsoft's decision instead causes Secure Boot to come with a hidden degradation of security.

As a result, we advice to override the default validation profile settings when using BitLocker. For additional security PCR0 should be included, as well as enabling Hardware Validated Boot on supported devices.

9.1 Recovery Key as Security Override

Even when BitLocker correctly picks up on an integrity violation, the security advantage still highly dependens on the user's reaction to the recovery prompt. While BitLocker protects in scenarios where the system owner is not present by preventing unauthorized access to the hard drive, we argue that the significance of the recovery prompt's appearance is dismissed by providing the system owner an immediate ability to override the security reaction with a recovery key. This leaves the burden of security enforcement to the user. It is now their responsibility to be aware of the security related implications of the recovery prompt and to act accordingly upon them. We can look at how the user is influenced in their decision, taking a closer look at the recovery prompt in Figure 7.4, we see that the message suggests a configuration change might have caused the prompt to appear. It is hinting the user, that the removal of a disk or USB stick might fix the issue. The rest is only about helping the user to find their recovery key to enter.

When taking appropriate precautions the hard drive should be removed from the system and inserted into a trustworthy system. From here the recovery key can be used to to fullfil its namesake, recovering data from the drive. Entering the recovery key into the recovery prompt is not data recovery, it is overriding a security mechanism. Presumably a user assumes to have encountered the recovery prompt through a false positive, a firmware update without prior BitLocker suspension, boot configuration related changes etc. When there is no reasonable assumption for the user to have encounter the recovery prompt they should not enter the recovery key. After all the inherent problem is that the system integrity has been violated, in what fashion is now unverifiable. It leaves the system in a state where no further trust should be put into it. A distinction about whether a false positive or malicious code has caused the recovery prompt is not possible. It is also the only time a user has a chance to react correctly, as with their decision to further put trust into an infected system, causes from hereinafter that malicious code will be part of the root of trust.

Microsoft allowing this security override created a dangerous precedence, especially since they do not display any warnings about inherent danger. Instead of a recovery prompt the user should have been made aware of the inherent loss of system integrity instead of normalising the security override. Any future adjustments in the recovery prompt, to provide more user awareness, can be manipulated by an attacker, as by definition the appearance of the prompt signals that there is no more system integrity. It is already possible to easily modify the message and URL displayed by the recovery prompt, as these can be configured through group policies [37], that are read from the BCD hive [68]. Even if the recovery prompt were to be removed in the future, phishing attacks could still replicate the prompt.

An attack that willingly triggers the BitLocker recovery prompt generally risks raising suspicion, which may lead to investigations and being discovered. It is somewhat of a last ditch effort, relying on social engineering and may be compared to phishing.

9.2 Mitigations

Enabling Hardware Validated Boot should prevent the execution of any malicious code within in the firmware image. Using the TPM with additional authentication mechanisms like a PIN or USB also adds to the security against an attacker trying to gain access to a system without the owner's interaction. When the owner is present to enter the PIN into an infected system, it unfortunately does not provide any additional security. It is always beneficial to use a password to protect the interactive firmware menu, so that Secure Boot cannot be disabled and booting into malicious removable media is prevented. System designers can also take into account to make it harder to acccess the flash, so that an attacker will have trouble modifing the firmware image.

10 Conclusion

Our practical analysis of UEFI threats against Windows 11 showed that enabling Secure Boot when using BitLocker comes with a hidden forfeiture of security. Microsoft misuses Secure Boot in an attempt to provide platform firmware integrity validation, where the TPM already offers a perfectly fine solution. With such a misconfigured BitLocker validation profile our rootkit was able to sniff the communication between the Windows Boot Manager and TPM without introducing side effects. Through interception of the *unseal* command we gained access to the unencrypted BitLocker VMK, to decrypt the hard drive and deploy further payload in the Windows installation. By then modifying the Windows registry our payload was executed with privileges of the local system account.

Microsoft also have set a dangerous precedence by offering the user a mechanism to override the security reaction to integrity violations in an inherently untrustworthy system. In the case of a correctly configured BitLocker validation profile the code of our root- or bootkit is measured into the TPM, causing the *unseal* operation to fail and the Windows Boot Manager to trigger a recovery prompt. The burden of security enforcement is now left to the user and when they decide put further trust into the system and enter the recover key, our *BitLogger* is able to record the performed keystrokes to decrypt the hard drive.

Future Work

Investigations into the RTM being established by the SRTM measuring itself could reveal flaws further up in the implementation of the measurement chain.

62

Bibliography

[1]crowdstrike. *Rootkit Malware*. Sept. 2021. URL: https://www.crowdstrike.com/cybersecurity-101/malware/rootkits/ (cit. on p. 2).

[2]Techtarget. *Definition rootkit*. Oct. 2021. URL: https://www.techtarget.com/searchsecurity/definition/rootkit (cit. on p. 2).

[3]Microsoft. *Secure the Windows boot process*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process (cit. on pp. 2, 28, 29).

[4]UEFI Forum. *Specifications and tools*. 2022. URL: https://uefi.org/specsandtesttools (cit. on p. 4).

[5]Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Developing with the Unified Extensible Firmware Interface, Third Edition*. Berlin, Boston: De|G Press, 2017 (cit. on pp. 5, 7, 10).

[6]UEFI Forum. *UEFI Specification, Version 2.9*. Tech. rep. UEFI Forum, Mar. 2021 (cit. on pp. 5–10, 12–14, 18, 19, 35, 40, 41, 44, 46, 51, 54).

[7]RFC Network Working Group. *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. IETF, July 2005 (cit. on p. 5).

[8]Microsoft. *Windows and GPT FAQ*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-and-gpt-faq?view=windows-11 (cit. on p. 6).

[9]TianoCore. *EDK II Driver Writer's Guide*. Apr. 2018. URL: https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide (cit. on pp. 7, 8, 10, 51, 52).

[10]UEFI Forum. *UEFI Platform Initialization (PI) Specification, Version 1.7 Errata A*. Tech. rep. UEFI Forum, Apr. 2020 (cit. on pp. 10, 15–20, 44, 45, 86, 88).

[11]TianoCore. *Understanding UEFI Secure Boot Chain*. June 2019. URL: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/ (cit. on pp. 12, 13).

[12]Microsoft. *Microsoft Windows Authenticode Portable Executable Signature Format, Version 1.0*. June 2019. URL: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/ (cit. on p. 13).

[13]LongSoft. *UEFITool*. Dec. 2022. URL: https://github.com/LongSoft/UEFITool (cit. on p. 15).

[14]TianoCore. *EDK II Build Specification*. July 2019. URL: https://edk2-docs.gitbook.io/edk-ii-build-specification/ (cit. on p. 19).

63

[15]TianoCore. *EDK II Project Github Repository*. 2022. URL: https://github.com/tianocore/edk2/blob/master/FmpDevicePkg/FmpDxe/FmpDxe.c (cit. on p. 20).

[16]UEFI Forum. *UEFI Shell Specification, Revision 2.2*. Tech. rep. UEFI Forum, Jan. 2016 (cit. on pp. 21, 39).

[17]TianoCore. *What is TianoCore?* URL: https://www.tianocore.org/ (cit. on p. 22).

[18]Microsoft. *Trusted Platform Module Technology Overview*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/trusted-platform-module-overview (cit. on p. 23).

[19]Trusted Computing Group. *Trusted Platform Module Library, Part 1: Architecture, Level 00 Revision 01.59*. Tech. rep. Trusted Computing Group, 2019 (cit. on p. 23).

[20]Trusted Computing Group. *TCG PC Client Platform Firmware Profile Specification*. Tech. rep. Trusted Computing Group, 2021 (cit. on pp. 23–25).

[21]TianoCore. *Trusted Boot Chain*. Mar. 2021. URL: https://tianocore-docs.github.io/edk2-TrustedBootChain/release-1.00/ (cit. on p. 24).

[22]David A. Solomon Mark E. Russinovich and Alex Ionescu. *Windows Internals, Part 2*. 6th ed. Microsoft Press, Sept. 2012 (cit. on pp. 25, 30, 31, 53).

[23]Microsoft. *Windows 11 overview*. Nov. 2022. URL: https://learn.microsoft.com/en-us/windows/whats-new/windows-11-overview (cit. on p. 26).

[24]Microsoft. *Windows minimum hardware requirements*. Sept. 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/design/minimum/minimum-hardware-requirements-overview (cit. on pp. 26, 28).

[25]Mark E. Russinovich Andrea Allievi Alex Ionescu and David A. Solomon. *Windows Internals, Part 2*. 7th ed. Developer Reference. Pearson Education, Inc., Sept. 2021 (cit. on pp. 26–28, 42, 48).

[26]Mark E. Russinovich Pavel Yosifovich Alex Ionescu and David A. Solomon. *Windows Internals, Part 1*. 7th ed. Microsoft Press, May 2017 (cit. on p. 27).

[27]Microsoft. *UPDATED: UEFI Signing Requirements*. Jan. 2021. URL: https://techcommunity.microsoft.com/t5/hardware-dev-center/updated-uefi-signing-requirements/ba-p/1062916 (cit. on p. 28).

[28]Microsoft. *Secure Boot and Trusted Boot*. Nov. 2022. URL: https://learn.microsoft.com/en-us/windows/security/trusted-boot (cit. on p. 28).

[29]Microsoft. *Overview of Early Launch AntiMalware*. Mar. 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/drivers/install/early-launch-antimalware (cit. on p. 30).

[30]Joymalya Basu Roy. *Understanding Windows Trusted Boot - Integrity Check and ELAM*. Mar. 2022. URL: https://www.anoopcnair.com/understanding-windows-trusted-boot/ (cit. on p. 30).

[31] Microsoft. *Virtualization-based Security (VBS)*. Mar. 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/design/device-experiences/oem-vbs (cit. on p. 30).

[32] Microsoft. *Windows Defender Application Control and virtualization-based protection of code integrity*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/threat-protection/device-guard/introduction-to-device-guard-virtualization-based-security-and-windows-defender-application-control (cit. on p. 30).

[33] Microsoft. *BitLocker*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview (cit. on p. 30).

[34] libde. *BitLocker Drive Encryption (BDE) format specification*. Feb. 2022. URL: https://github.com/libyal/libbde/blob/main/documentation/BitLocker%20Drive%20Encryption%20(BDE)%20format.asciidoc (cit. on pp. 31, 32).

[35] Microsoft. *Prepare an organization for BitLocker: Planning and policies*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/prepare-your-organization-for-bitlocker-planning-and-policies (cit. on p. 31).

[36] Microsoft. *BitLocker Countermeasures*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-countermeasures (cit. on p. 31).

[37] Microsoft. *BitLocker group policy settings*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-group-policy-settings (cit. on pp. 31, 59, 61).

[38] Microsoft. *autochk*. Feb. 2022. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-server/deployment/pcr7-configuration-binding-not-possible (cit. on pp. 31, 59).

[39] Microsoft. *Trusted Execution Environment EFI Protocol*. Oct. 2022. URL: https://learn.microsoft.com/en-us/windows-hardware/test/hlk/testref/trusted-execution-environment-efi-protocol (cit. on p. 31).

[40] Global Research and Kaspersky Lab Analysis Team. *FinSpy*. Sept. 2021. URL: https://securelist.com/finspy-unseen-findings/104322/ (cit. on pp. 33–35).

[41] Martin Smolár and Anton Cherepanov. *ESPecter*. Oct. 2021. URL: https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/ (cit. on pp. 33–35).

[42] Quarkslab. *Dreamboot*. Apr. 2013. URL: https://github.com/quarkslab/dreamboot (cit. on pp. 33–35).

[43] Mattiwatti. *EfiGuard*. Aug. 2022. URL: https://github.com/Mattiwatti/EfiGuard (cit. on pp. 33–35).

[44] Hackingteam. *vector-edk*. Apr. 2015. URL: https://github.com/hackedteam/vector-edk (cit. on pp. 34, 35).

[45] Igor Kuznetsov Mark Lechtik and Yury Parshin. *MosaicRegressor*. Oct. 2020. URL: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2020/10/07080558/MosaicRegressor_Technical-details.pdf (cit. on pp. 34, 35).

[46] ESET Research. *LoJax*. Sept. 2018. URL: https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf (cit. on pp. 34, 35).

[47] Denis Legezo Mark Lechtik Vasily Berdnikov and Ilya Borisov. *MoonBounce: the dark side of UEFI firmware*. Jan. 2022. URL: https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/ (cit. on pp. 34, 35).

[48] Global Research and Kaspersky Lab Analysis Team. *CosmicStrand*. July 2022. URL: https://securelist.com/cosmicstrand-uefi-firmware-rootkit/106973/ (cit. on pp. 34, 35).

[49] QEMU. *QEMU*. 2022. URL: https://www.qemu.org/ (cit. on p. 36).

[50] Stefan Berger and David Safford. *SWTPM - Software TPM Emulator*. Dec. 2022. URL: https://github.com/stefanberger/swtpm (cit. on p. 36).

[51] Lenovo. *IdeaPad 5 Pro Gen 6 (16" AMD)*. 2022. URL: https://www.lenovo.com/de/de/laptops/ideapad/500-series/IdeaPad-5-Pro-16ACH6/p/88IPS501619 (cit. on p. 36).

[52] man7. *kernel_lockdown(7) - Linux manual page*. 2021. URL: https://man7.org/linux/man-pages/man7/kernel_lockdown.7.html (cit. on p. 37).

[53] erpalma. *Fix Intel CPU Throttling on Linux*. 2022. URL: https://github.com/erpalma/throttled#writing-to-msr-and-pci-bar (cit. on p. 37).

[54] tuxera. *ntfs-3g*. Nov. 2022. URL: https://github.com/tuxera/ntfs-3g (cit. on p. 39).

[55] pbatard. *ntfs-3g*. Dec. 2022. URL: https://github.com/pbatard/ntfs-3g (cit. on p. 39).

[56] Microsoft. *Task Triggers*. Aug. 2021. URL: https://learn.microsoft.com/en-us/windows/win32/taskschd/task-triggers (cit. on p. 42).

[57] Microsoft. *Task Actions*. Aug. 2021. URL: https://learn.microsoft.com/en-us/windows/win32/taskschd/task-actions (cit. on p. 42).

[58] Microsoft. *Task Security Contexts*. Feb. 2020. URL: https://learn.microsoft.com/en-us/windows/win32/taskschd/security-contexts-for-running-tasks (cit. on p. 42).

[59] Microsoft. *rundll32*. Mar. 2021. URL: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/rundll32 (cit. on p. 42).

[60] Microsoft. *autochk*. Mar. 2021. URL: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/autochk (cit. on p. 42).

[61] Microsoft. *whoami*. Mar. 2021. URL: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/whoami (cit. on p. 43).

[62] Microsoft. *LocalSystem Account*. Jan. 2021. URL: https://learn.microsoft.com/en-us/windows/win32/services/localsystem-account (cit. on p. 43).

[63] Petter Nordahl-Hagen. *Chntpw*. Feb. 2014. URL: http://pogostick.net/~pnh/ntpasswd/ (cit. on p. 43).

[64] Aorimn. *Dislocker*. Sept. 2022. URL: https://github.com/Aorimn/dislocker (cit. on p. 54).

[65] Microsoft. *BitLocker: Use BitLocker Drive Encryption Tools to manage BitLocker*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-use-bitlocker-drive-encryption-tools-to-manage-bitlocker (cit. on p. 57).

[66] Henri Nurmi. *Sniff, there leaks my BitLocker key*. Dec. 2020. URL: https://labs.withsecure.com/publications/sniff-there-leaks-my-bitlocker-key (cit. on p. 57).

[67] Denis Andzakovic. *Extracting BitLocker keys from a TPM*. Mar. 2019. URL: https://pulsesecurity.co.nz/articles/TPM-sniffing (cit. on p. 57).

[68] Microsoft. *Boot Configuration Data settings and BitLocker*. Dec. 2022. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bcd-settings-and-bitlocker (cit. on p. 61).

List of Figures

68

List of Tables

69

List of Source Code Listings

70

List of Acronyms

ACPI Advanced Configuration and Power Interface
AES Advanced Encryption Standard
AL Afterlife
AMD Advanced Micro Devices
API Application Programming Interface
BCD Boot Configuration Data
BDE BitLocker Drive Encryption
BDS Boot Device Selection
BF Boot Firmware
BFV Boot Firmware Volume
BIOS Basic Input/Output System
CA Certificate Authority
CI Code Integrity
CRTM Core Root of Trust for Measurement
CSM Compatibility Support Module
CPU Central Processing Unit
DB Data Base
DEPEX Dependency Expression
DLL Dynamically Linked Library
DXE Driver Execution Environment
EDK EFI Development Kit
EFI Extensible Firmware Interface
ELAM Early-Launch Antimalware
ESP EFI System Partition
FAT File Allocation Table
FD Firmware Device
FDF Flash Description File
FFS Firmware Filesystem
FUSE Filesystem in Userspace
FV Firmware Volume
FVE Full Volume Encryption
FVEK Full Volume Encryption Key
GPT GUID Partition Table

71

GUID	Globally Unique Identifier
HOB	Hand-off Block
H-CRTM	Hardware-Core Root of Trust for Measurement
HVCI	Hypervisor-protected Code Integrity
I/O	Input/Output
IBB	Initial Boot Block
KEK	Key Exchange Key
KMCI	Kernel Mode Code Integrity
LBA	Logical Block Address
LPC	Low Pin Count
MBR	Master Boot Record
NTFS	New Technology File System
NVRAM	Non-volatile RAM
OEM	Original Equipment Manufacturer
OS	Operating System
OVMF	Open Virtual Machine Firmware
PC	Personal Computer
PCI	Peripheral Component Interconnect
PCR	Platform Configuration Register
PE32	Portable Executable 32-Bit
PEI	Pre-EFI Initialization
PEIM	Pre-EFI Initialization Module
PF	Platform Firmware
PI	Platform Initialization
PIN	Personal Identification Number
PK	Platform Key
PPI	PEIM-to-PEIM Interface
QEMU	Quick Emulator
RAM	Random Access Memory
ROM	Read-Only Memory
RT	Runtime
RTM	Root of Trust for Measurement
SEC	Security
SMM	System Management Mode
SRTM	Static Root of Trust for Measurement
SPI	Serial Peripheral Interface
TB	Terra Byte
TCG	Trusted Computing Group
TPM	Trusted Platform Module

TSL	Transient System Load
UEFI	Unified Extensible Firmware Interface
USB	Universal Serial Bus
UUID	Universally Unique Identifier
VBS	Virtualization Based Security
VMK	Volume Master Key
VSM	Virtualization Secure Mode
WDAC	Windows Defender Application Control
XML	Extensible Markup Language

Appendix A

A.1 System Table

```
typedef struct
{
    EFI_TABLE_HEADER Hdr;
    CHAR16 *FirmwareVendor;
    UINT32 FirmwareRevision;
    EFI_HANDLE ConsoleInHandle;
    EFI_SIMPLE_TEXT_INPUT_PROTOCOL *ConIn;
    EFI_HANDLE ConsoleOutHandle;
    EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *ConOut;
    EFI_HANDLE StandardErrorHandle;
    EFI_SIMPLE_TEXT_OUTPUT_PROTOCOL *StdErr;
    EFI_RUNTIME_SERVICES *RuntimeServices;
    EFI_BOOT_SERVICES *BootServices;
    UINTN NumberOfTableEntries;
    EFI_CONFIGURATION_TABLE *ConfigurationTable;
} EFI_SYSTEM_TABLE;
```

Listing A.1: System Table

74

A.1.1 Boot Services

```
typedef struct
{
    EFI_TABLE_HEADER Hdr;
    EFI_RAISE_TPL RaiseTPL;
    EFI_RESTORE_TPL RestoreTPL;
    EFI_ALLOCATE_PAGES AllocatePages;
    EFI_FREE_PAGES FreePages;
    EFI_GET_MEMORY_MAP GetMemoryMap;
    EFI_ALLOCATE_POOL AllocatePool;
    EFI_FREE_POOL FreePool;
    EFI_CREATE_EVENT CreateEvent;
    EFI_SET_TIMER SetTimer;
    EFI_WAIT_FOR_EVENT WaitForEvent;
    EFI_SIGNAL_EVENT SignalEvent;
    EFI_CLOSE_EVENT CloseEvent;
    EFI_CHECK_EVENT CheckEvent;
    EFI_INSTALL_PROTOCOL_INTERFACE InstallProtocolInterface;
    EFI_REINSTALL_PROTOCOL_INTERFACE ReinstallProtocolInterface;
    EFI_UNINSTALL_PROTOCOL_INTERFACE UninstallProtocolInterface;
    EFI_HANDLE_PROTOCOL HandleProtocol;
    VOID *Reserved;
    EFI_REGISTER_PROTOCOL_NOTIFY RegisterProtocolNotify;
    EFI_LOCATE_HANDLE LocateHandle;
    EFI_LOCATE_DEVICE_PATH LocateDevicePath;
    EFI_INSTALL_CONFIGURATION_TABLE InstallConfigurationTable;
    EFI_IMAGE_LOAD LoadImage;
    EFI_IMAGE_START StartImage;
    EFI_EXIT Exit;
    EFI_IMAGE_UNLOAD UnloadImage;
    EFI_EXIT_BOOT_SERVICES ExitBootServices;
    EFI_GET_NEXT_MONOTONIC_COUNT GetNextMonotonicCount;
    EFI_STALL Stall;
    EFI_SET_WATCHDOG_TIMER SetWatchdogTimer;
    EFI_CONNECT_CONTROLLER ConnectController;
    EFI_DISCONNECT_CONTROLLER DisconnectController;
    EFI_OPEN_PROTOCOL OpenProtocol;
    EFI_CLOSE_PROTOCOL CloseProtocol;
    EFI_OPEN_PROTOCOL_INFORMATION OpenProtocolInformation;
    EFI_PROTOCOLS_PER_HANDLE ProtocolsPerHandle;
    EFI_LOCATE_HANDLE_BUFFER LocateHandleBuffer;
    EFI_LOCATE_PROTOCOL LocateProtocol;
    EFI_INSTALL_MULTIPLE_PROTOCOL_INTERFACES InstallMultipleProtocolInterfaces;
```

```
    EFI_UNINSTALL_MULTIPLE_PROTOCOL_INTERFACES UninstallMultipleProtocolInterfaces;
    EFI_CALCULATE_CRC32 CalculateCrc32;
    EFI_COPY_MEM CopyMem;
    EFI_SET_MEM SetMem;
    EFI_CREATE_EVENT_EX CreateEventEx;
} EFI_BOOT_SERVICES;
```

Listing A.2: Boot Services

A.1.2 Runtime Services

```
typedef struct
{
    EFI_TABLE_HEADER Hdr;
    EFI_GET_TIME GetTime;
    EFI_SET_TIME SetTime;
    EFI_GET_WAKEUP_TIME GetWakeupTime;
    EFI_SET_WAKEUP_TIME SetWakeupTime;
    EFI_SET_VIRTUAL_ADDRESS_MAP SetVirtualAddressMap;
    EFI_CONVERT_POINTER ConvertPointer;
    EFI_GET_VARIABLE GetVariable;
    EFI_GET_NEXT_VARIABLE_NAME GetNextVariableName;
    EFI_SET_VARIABLE SetVariable;
    EFI_GET_NEXT_HIGH_MONO_COUNT GetNextHighMonotonicCount;
    EFI_RESET_SYSTEM ResetSystem;
    EFI_UPDATE_CAPSULE UpdateCapsule;
    EFI_QUERY_CAPSULE_CAPABILITIES QueryCapsuleCapabilities;
    EFI_QUERY_VARIABLE_INFO QueryVariableInfo;
} EFI_RUNTIME_SERVICES;
```

Listing A.3: Runtime Services

A.2 Protocols

A.2.1 Loaded Image Protocol

```
typedef struct
{
  UINT32 Revision;
  EFI_HANDLE ParentHandle;
  EFI_SYSTEM_TABLE *SystemTable;
  EFI_HANDLE DeviceHandle;
  EFI_DEVICE_PATH_PROTOCOL *FilePath;
  VOID *Reserved;
  UINT32 LoadOptionsSize;
  VOID *LoadOptions;
  VOID *ImageBase;
  UINT64 ImageSize;
  EFI_MEMORY_TYPE ImageCodeType;
  EFI_MEMORY_TYPE ImageDataType;
  EFI_IMAGE_UNLOAD Unload;
} EFI_LOADED_IMAGE_PROTOCOL;

extern EFI_GUID gEfiLoadedImageProtocolGuid;
extern EFI_GUID gEfiLoadedImageDevicePathProtocolGuid;
```

Listing A.4: Loaded Image Protocol

A.2.2 Driver Binding Protocol

```
typedef EFI_STATUS(EFIAPI *EFI_DRIVER_BINDING_SUPPORTED)(
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL);

typedef EFI_STATUS(EFIAPI *EFI_DRIVER_BINDING_START)(
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN EFI_DEVICE_PATH_PROTOCOL *RemainingDevicePath OPTIONAL);

typedef EFI_STATUS(EFIAPI *EFI_DRIVER_BINDING_STOP)(
    IN EFI_DRIVER_BINDING_PROTOCOL *This,
    IN EFI_HANDLE ControllerHandle,
    IN UINTN NumberOfChildren,
    IN EFI_HANDLE *ChildHandleBuffer OPTIONAL);

struct _EFI_DRIVER_BINDING_PROTOCOL
{
  EFI_DRIVER_BINDING_SUPPORTED Supported;
  EFI_DRIVER_BINDING_START Start;
  EFI_DRIVER_BINDING_STOP Stop;
  UINT32 Version;
  EFI_HANDLE ImageHandle;
  EFI_HANDLE DriverBindingHandle;
};

extern EFI_GUID gEfiDriverBindingProtocolGuid;
```

Listing A.5: Driver Binding Protocol

A.2.3 Simple Text Input Ex Protocol

```
typedef EFI_STATUS(EFIAPI *EFI_INPUT_READ_KEY_EX)(
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    OUT EFI_KEY_DATA *KeyData);

typedef EFI_STATUS(EFIAPI *EFI_SET_STATE)(
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN EFI_KEY_TOGGLE_STATE *KeyToggleState);

typedef EFI_STATUS(EFIAPI *EFI_KEY_NOTIFY_FUNCTION)(
    IN EFI_KEY_DATA *KeyData);

typedef EFI_STATUS(EFIAPI *EFI_REGISTER_KEYSTROKE_NOTIFY)(
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN EFI_KEY_DATA *KeyData,
    IN EFI_KEY_NOTIFY_FUNCTION KeyNotificationFunction,
    OUT VOID **NotifyHandle);

typedef EFI_STATUS(EFIAPI *EFI_UNREGISTER_KEYSTROKE_NOTIFY)(
    IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
    IN VOID *NotificationHandle);

struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL
{
  EFI_INPUT_RESET_EX Reset;
  EFI_INPUT_READ_KEY_EX ReadKeyStrokeEx;
  EFI_EVENT WaitForKeyEx;
  EFI_SET_STATE SetState;
  EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;
  EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;
};

extern EFI_GUID gEfiSimpleTextInputExProtocolGuid;
```

Listing A.6: Simple Text Input Ex Protocol

A.2.4 Simple File System and File Protocol

```
typedef EFI_STATUS(EFIAPI *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME)(
    IN EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *This,
    OUT EFI_FILE_PROTOCOL **Root);

struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL
{
  UINT64 Revision;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
};

struct _EFI_FILE_PROTOCOL
{
  UINT64 Revision;
  EFI_FILE_OPEN Open;
  EFI_FILE_CLOSE Close;
  EFI_FILE_DELETE Delete;
  EFI_FILE_READ Read;
  EFI_FILE_WRITE Write;
  EFI_FILE_GET_POSITION GetPosition;
  EFI_FILE_SET_POSITION SetPosition;
  EFI_FILE_GET_INFO GetInfo;
  EFI_FILE_SET_INFO SetInfo;
  EFI_FILE_FLUSH Flush;
  EFI_FILE_OPEN_EX OpenEx;
  EFI_FILE_READ_EX ReadEx;
  EFI_FILE_WRITE_EX WriteEx;
  EFI_FILE_FLUSH_EX FlushEx;
};

extern EFI_GUID gEfiSimpleFileSystemProtocolGuid;
```

Listing A.7: Simple File System and File Protocol

A.2.5 Disk I/O Protocol

```
typedef EFI_STATUS(EFIAPI *EFI_DISK_READ)(
      IN EFI_DISK_IO_PROTOCOL *This,
      IN UINT32 MediaId,
      IN UINT64 Offset,
      IN UINTN BufferSize,
      OUT VOID *Buffer);

typedef EFI_STATUS(EFIAPI *EFI_DISK_WRITE)(
      IN EFI_DISK_IO_PROTOCOL *This,
      IN UINT32 MediaId,
      IN UINT64 Offset,
      IN UINTN BufferSize,
      IN VOID *Buffer);

struct _EFI_DISK_IO_PROTOCOL
{
   UINT64 Revision;
   EFI_DISK_READ ReadDisk;
   EFI_DISK_WRITE WriteDisk;
};

extern EFI_GUID gEfiDiskIoProtocolGuid;
```

Listing A.8: Disk I/O Protocol

A.2.6 Block I/O Protocol

```
typedef EFI_STATUS(EFIAPI *EFI_BLOCK_RESET)(
      IN EFI_BLOCK_IO_PROTOCOL *This,
      IN BOOLEAN ExtendedVerification);

typedef EFI_STATUS(EFIAPI *EFI_BLOCK_READ)(
      IN EFI_BLOCK_IO_PROTOCOL *This,
      IN UINT32 MediaId,
      IN EFI_LBA Lba,
      IN UINTN BufferSize,
      OUT VOID *Buffer);

typedef EFI_STATUS(EFIAPI *EFI_BLOCK_WRITE)(
      IN EFI_BLOCK_IO_PROTOCOL *This,
      IN UINT32 MediaId,
      IN EFI_LBA Lba,
      IN UINTN BufferSize,
      IN VOID *Buffer);

typedef EFI_STATUS(EFIAPI *EFI_BLOCK_FLUSH)(
      IN EFI_BLOCK_IO_PROTOCOL *This);

struct _EFI_BLOCK_IO_PROTOCOL
{
   UINT64 Revision;
   EFI_BLOCK_IO_MEDIA *Media;
   EFI_BLOCK_RESET Reset;
   EFI_BLOCK_READ ReadBlocks;
   EFI_BLOCK_WRITE WriteBlocks;
   EFI_BLOCK_FLUSH FlushBlocks;
};

extern EFI_GUID gEfiBlockIoProtocolGuid;
```

Listing A.9: Block I/O Protocol

```
typedef EFI_STATUS(EFIAPI *EFI_TCG2_HASH_LOG_EXTEND_EVENT)(
    IN EFI_TCG2_PROTOCOL *This,
    IN UINT64 Flags,
    IN EFI_PHYSICAL_ADDRESS DataToHash,
    IN UINT64 DataToHashLen,
    IN EFI_TCG2_EVENT *EfiTcgEvent);

typedef EFI_STATUS(EFIAPI *EFI_TCG2_SUBMIT_COMMAND)(
    IN EFI_TCG2_PROTOCOL *This,
    IN UINT32 InputParameterBlockSize,
    IN UINT8 *InputParameterBlock,
    IN UINT32 OutputParameterBlockSize,
    IN UINT8 *OutputParameterBlock);

typedef EFI_STATUS(EFIAPI *EFI_TCG2_GET_ACTIVE_PCR_BANKS)(
    IN EFI_TCG2_PROTOCOL *This,
    OUT UINT32 *ActivePcrBanks);

typedef EFI_STATUS(EFIAPI *EFI_TCG2_SET_ACTIVE_PCR_BANKS)(
    IN EFI_TCG2_PROTOCOL *This,
    IN UINT32 ActivePcrBanks);

struct tdEFI_TCG2_PROTOCOL
{
  EFI_TCG2_GET_CAPABILITY GetCapability;
  EFI_TCG2_GET_EVENT_LOG GetEventLog;
  EFI_TCG2_HASH_LOG_EXTEND_EVENT HashLogExtendEvent;
  EFI_TCG2_SUBMIT_COMMAND SubmitCommand;
  EFI_TCG2_GET_ACTIVE_PCR_BANKS GetActivePcrBanks;
  EFI_TCG2_SET_ACTIVE_PCR_BANKS SetActivePcrBanks;
  EFI_TCG2_GET_RESULT_OF_SET_ACTIVE_PCR_BANKS GetResultOfSetActivePcrBanks;
};

extern EFI_GUID gEfiTcg2ProtocolGuid;
```

Listing A.10: TCG2 Protocol

A.2.7 Boot Device Selection Protocol

[TODO BDS protocol]

A.3 Firmware File Types

EFI_FV_FILETYPE_RAW	0x01	Binary data
EFI_FV_FILETYPE_FREEFORM	0x02	Sectioned data
EFI_FV_FILETYPE_SECURITY_CORE	0x03	Platform core code used during the SEC phase
EFI_FV_FILETYPE_PEI_CORE	0x04	PEI Foundation
EFI_FV_FILETYPE_DXE_CORE	0x05	DXE Foundation
EFI_FV_FILETYPE_PEIM	0x06	PEI module (PEIM)
EFI_FV_FILETYPE_DRIVER	0x07	DXE driver
EFI_FV_FILETYPE_COMBINED_PEIM_DRIVER	0x08	Combined PEIM/DXE driver
EFI_FV_FILETYPE_APPLICATION	0x09	Application
EFI_FV_FILETYPE_MM	0x0A	Contains a PE32+ image that will be loaded into MMRAM in MM Traditional Mode.
EFI_FV_FILETYPE_FIRMWARE_VOLUME_IMAGE	0x0B	Firmware volume image
EFI_FV_FILETYPE_COMBINED_MM_DXE	0x0C	Contains PE32+ image that will be dispatched by the DXE Dispatcher and will also beloaded into MMRAM in MM Tradition Mode.
EFI_FV_FILETYPE_MM_CORE	0x0D	MM Foundation that support MM Traditional Mode.
EFI_FV_FILETYPE_MM_STANDALONE	0x0E	Contains a PE32+ image that will be loaded into MMRAM in MM Standalone Mode.
EFI_FV_FILETYPE_MM_CORE_STANDALONE	0x0F	MM Foundation that support MM Tradition Mode and MM Standalone Mode.
EFI_FV_FILETYPE_OEM_MIN... EFI_FV_FILETYPE_OEM_MAX	0xC0–0xDF	OEM File Types
EFI_FV_FILETYPE_DEBUG_MIN... EFI_FV_FILETYPE_DEBUG_MAX	0xE0–0xEF	Debug/Test File Types
EFI_FV_FILETYPE_FFS_MIN... EFI_FV_FILETYPE_FFS_MAX	0xF0–0xFF	Firmware File System Specific File Types
EFI_FV_FILETYPE_FFS_PAD	0xF0	Pad File For FFS

Tab. A.1.: Firmware File Types [10, Vol. 3, Table 3-3]

A.4 Firmware File Section Types

EFI_SECTION_COMPRESSION	0x01	Encapsulation section where other sections are compressed.
EFI_SECTION_GUID_DEFINED	0x02	Encapsulation section where other sections have format defined by a GUID.
EFI_SECTION_DISPOSABLE	0x03	Encapsulation section used during the build process but not required for execution.
EFI_SECTION_PE32	0x10	PE32+ Executable image.
EFI_SECTION_PIC	0x11	Position-Independent Code.
EFI_SECTION_TE	0x12	Terse Executable image.
EFI_SECTION_DXE_DEPEX	0x13	DXE Dependency Expression.
EFI_SECTION_VERSION	0x14	Version, Text and Numeric.
EFI_SECTION_USER_INTERFACE	0x15	User-Friendly name of the driver.
EFI_SECTION_COMPATIBILITY16	0x16	DOS-style 16-bit EXE.
EFI_SECTION_FIRMWARE_VOLUME_IMAG	0x17	PI Firmware Volume image.
EFI_SECTION_FREEFORM_SUBTYPE_GUI	0x18	Raw data with GUID in header to define format.
EFI_SECTION_RAW	0x19	Raw data.
EFI_SECTION_PEI_DEPEX	0x1B	PEI Dependency Expression.
EFI_SECTION_MM_DEPEX	0x1C	Leaf section type for determining the dispatch order for an MM Traditional driver in MM Traditional Mode or MM Standaline driver in MM Standalone Mode.

Tab. A.2.: Firmware File Section Types [10, Vol. 3, Table 3-4]