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 malicious firmware is 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

1 Introduction

As the first piece of software that is run on your computer, UEFI holds an immense amount of responsibility during system initialization, attacks targeting your operating system from this environment are executed long before

what does it different than bios this helps write platform independent code uefi threats:

A rootkit is a collection of software designed to grant a threat actor control over a system, typically with malicious intend. Rootkits set up a backdoor exploit and may deliver additional malware while leveraging their privileges to remain hidden. There are different types of rootkits such as User Mode, Kernel Mode, Bootkits (bootloader rootkits), Hypervisor and Firmware rootkits. [@cro21; @Tec21; @Mic22a] **[TODO consult abstract for similar definition, how easy uefi makes it to write hardware independent payload]** Firmware rootkits target the software running during the boot process, which is responsible for the system initialization. This is done before the operating system is executed making them particularly hard to find, they are also persistent across operating system installation or hard drive replacements [@cro21].

Problem Statement

look at past UEFI threats that have popped up in the wild and been analysed look at boot and rootkits how they infected the system/initial execution vector categorize them by their attack vector whether they attack the windows installation at rest in storage or during boot in memory analyze UEFI environment from the view of a storage-base UEFI threats in form of boot- and rootkit how do we gain initial execution what can we use what security mechanisms do we encounter UEFI side Windows side Windows standard security policies we perform attacks in an emulated environment as well as on hardware we enable additional security mechanisms increasing security mechanisms add onto past threats by looking at TPM and bitlocker

Structure

We start off in Chapter 2 by introducing all necessary knowledge about the UEFI environment, defined by the UEFI and PI specifications, listing the interface and its implementation. This allows us to go over Windows 11's UEFI installation and boot process as well as relevant security mechanisms in Chapter 3. With this knowledge we then look at analyses of previously discovered UEFI threats in Chapter 4, categorizing them by their attack vector and threat model. In Chapter 5 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 6, analyzing security mechanism faced when attempting attacks from the UEFI environment. Afterwards we discuss the impact of our findings, the restrictions that apply, as well as potential mitigation techniques in Chapter 7. Chapter 8 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 Operating System (OS) and Platform Firmware (PF). [...] Together, these provide a standard environment for booting an OS and running pre-boot applications" [@For22]. The specifications making up this model are:

- Advanced Configuration and Power Interface (ACPI) Specification
- UEFI Specification
- UEFI Shell Specification
- UEFI PI Specification
- UEFI PI Distribution Packaging Specification
- Trusted Computing Group (TCG) Extensible Firmware Interface (EFI) Platform Specification
- TCG EFI Protocol Specification

We make an effort to keep a clear distinction as to what is defined in which specification.

The UEFI PI **[TODO mention of EFI and framework into UEFI and pi?]**

2.1 Unified Extensible Firmware Interface (UEFI)

The UEFI specification itself is a pure interface specification, describing the programmatic interface for interaction with the PF, merely stating what interfaces and structures a PF has to offer and what an OS may use [ZRM17].

during boot system resources are owned by the firmware

protected mode, takes care of memory management and mapping with a one to one mapping of physical to virtual

- complete solution describing all features and capabilities - abstract interfaces to support a range of processors without the need for knowledge about underlying hardware for the bootloader - sharable persistent storage for platform support code security

[TODO here]

[For21, p. 2.3.4]

It was designed to replace the legacy Boot Firmware Basic Input/Output System (BIOS) **[TODO which wasnt very standardized]**, while also providing backwards compatibility by defining the Compatibility Support Module (CSM) allowing UEFI firmware to boot legacy BIOS applications.

system table with boot- and runttime service functions for the bootloader and os to call datatables containing platform-related information

[ZRM17]

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 paritions. 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 [Gro05].

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 paritioned using the Master Boot Record (MBR) parition 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 [For21, 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 parition 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 parition entries. One identifies the ESP, the parition UEFI boots from, within the legacy MBR partition table and the other indicates that a protective parition is used [For21, Section 5]. [For21, 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 [@Mic22b]. GUID are used to identify paritions and parition types, but also offering a human readable parition name. GPT also has a primary and a backup parition 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 [For21, Section 13.3]. It must contain an EFI root directory [For21, 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 [For21, 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 [For21, 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 [For21, 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 executions and later calls to these functions jump back into the driver's image which is still 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

When UEFI binaries are loaded only the entry point is *linked*, the rest of the communication has to be programmatically discovered through protocol interfaces. Protocols are created dynamically and provide a mechanism to allow extension of firmware capabilities over time [@Tia18, 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 [ZRM17]. This database is called the handle database, handles describe a logical grouping of one or more protocols [@Tia18, Section 3.6]. Handles are unique per session and should not be saved across reboots [ZRM17]. Multiple instances of a protocol identified by the same GUID can exist on different handles, offering the same service on different devices.

[@Tia18] explains the categories of handles that are formed by the type of protocols that are grouped. Figure 2.1 shows these categories.

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)

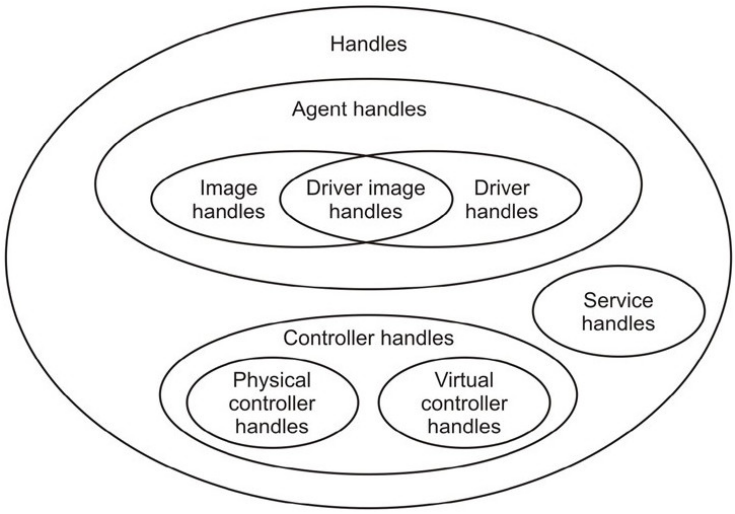


Fig. 2.1.: Handle types (taken from [@Tia18, Figure 3])

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 Input/Output (I/O) abstraction protocols. Physical device handles support the Device Path Protocol for generic path/location information [For21, Section 10.2].

Service handles are used for generic hardware unrelated abstractions.

2.1.6 UEFI Driver Model

[For21, Section 2.5.2] describes the UEFI Driver Model, it simplifies the design of device drivers by moving implementation of the device mangement and discovery into the firmware, leaving drivers with only the responsibility to offer interfaces for installation and removal.

We will focus on device drivers, these do not add any new device handles but instead offer protocol abstractions build upon already existing I/O abstractions offered by bus drivers. A driver following the UEFI Driver Model is not allowed to interact with any hardware in its entry point and is instead required to install an instace of the Driver Binding Protocol on its own image handle. The Loaded Image Protocol also offers a field where a driver can provide a function to unload itself. It may

also additionally install the confirguration or diagnostic related protocols. Runtime drivers usually register a notification function that is triggered when an OS loader calls `ExitBootServices`, this allows them to translate any allocated memory to their virtual addresses.

The firmware will try to connect device drivers to a controller by iterating over all instances of the Driver Binding Protocol in the handle database and calling the `Supported` function of the Driver Binding Protocol on a controller. The device driver then checks whether it supports the controller by for example looking for specific I/O abstraction protocols, that it will want to laters 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. This is done recursively as the newly installed device driver might now fullfill the requirements for another driver. The firmware can also call the `Stop` of the Driver Binding Protocol function if it wants a driver to uninstall its protocol instance from a controller, an example for this would be another device driver wanting to exclusively manage a controller. This is done by tracking agents of protocols, in other words the drivers who consume a protocol.

2.1.7 Systemtable

The UEFI System Table is an important data structure, it provides access to system configuration information, generic boot and runtime services [@Tia18, Section 3.3]. It also serves as the entrance in to the UEFI environment, as a loaded images receives a only pointer to the system table as well as its image handle through its entry point. Although the Loaded Image Protocol provides and interface to hand optional load options to the image [ZRM17].

Figure 2.2

[TODO during boot boot and runtime services are available]

2.1.7.1. Boot Services

UEFI applcations must use boot services functions to access devices and allocate memory. They are available until an OS loader takes control over the system via a call to the boot service `ExitBootServices()`, from which on only runtime services are available. [For21, Section 7] splits the boot services into five categories:

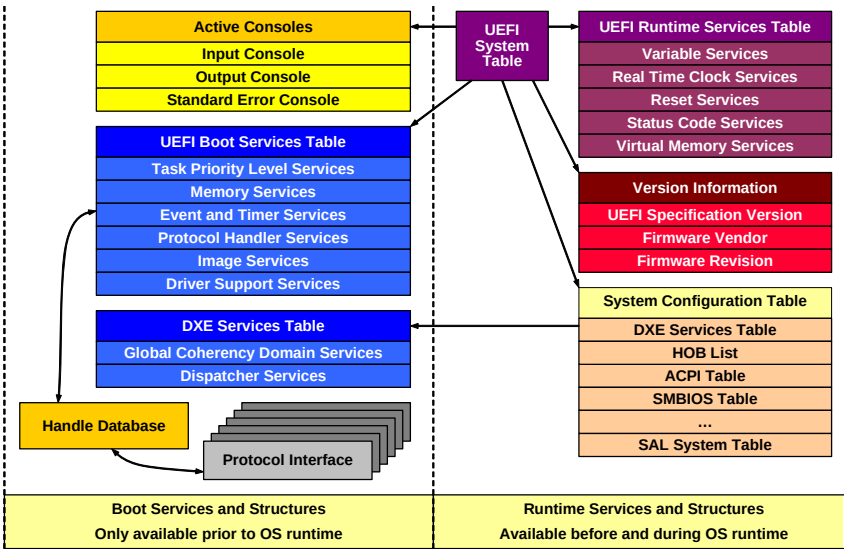


Fig. 2.2.: UEFI System Table (taken from [For20, Vol 2, Figure 2-5])

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

[TODO me]

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.

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 Runtime Services offer system reset, a monotonic counter and capsule services. Capsules allow the OS to pass data to the firmware, this includes firmware managment related data.

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 [For21, Section 8.2]. Architecturally defined UEFI variables are called Globally Defined Variables where the vendor GUID has the value `EFI_GLOBAL_VARIABLE` [For21, 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[For21, 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 [For21, 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[For21, Section 3.1.2], for x64 it uses the default path `\EFI\BOOT\BOOTX64.EFI` [For21, Section 3.5].

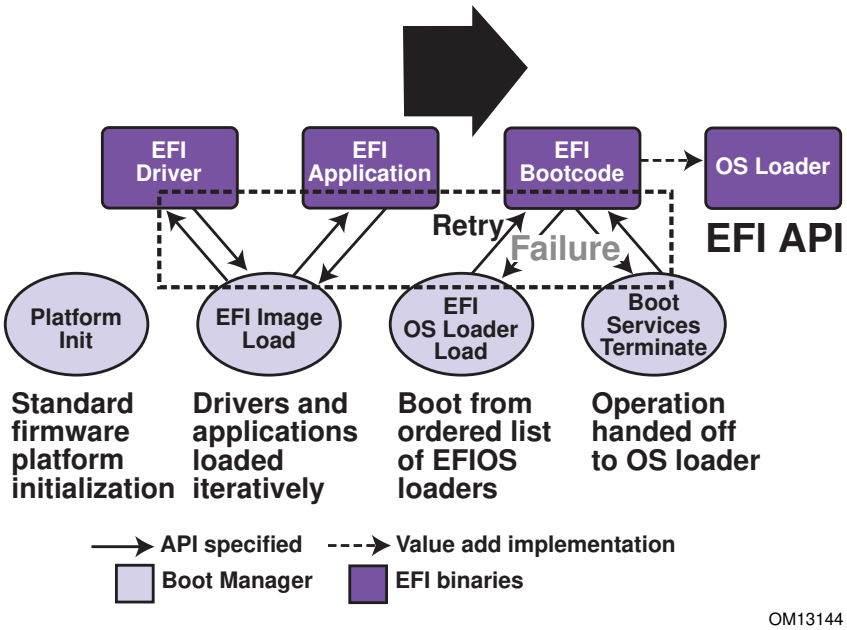


Fig. 2.3.: Booting Sequence (taken from [For21, Figure 2-1])

2.1.10 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 [@Tia19] [For21, 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 Peripheral Component Interconnect (PCI) option Read-Only Memorys (ROMs), bootloader from OS vendors and tools such as the UEFI shell [@Tia19]. Digital signatures, embedded within the UEFI images, can be used to authenticate origin and/or integrity [For21, 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 [For21, Section 32.2.2]. The signatures are created by first calculating a hash over select parts of the executable, leaving, for example, the signatures out of the hashed data 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 [@Mic19]. 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 [For21, Section 32.3]. The PK is a simple variable where the KEK and DB are implemented through signature list data structures [For21, Section 32.4.1], the variable services can be used to append entries or to read and write the list as a whole [For21, 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 Original Equipment Manufacturer (OEM) to supply platform-defined values, used during Secure Boot initialization. Their contents can be copied to their live versions, used during Secure Boot operation. The current state of Secure Boot is also reflected within a secure variable [For21, Section 3.3].

Users, who are physically present, may disable Secure Boot, enroll default or custom keys via an interactive menu. **[TODO find a good cite]**

[TODO maybe secure boot authorization process]

2.1.11 Firmware Management

[TODO me]

provides CapsuleUpdate() QueryCapsuleCapabilities() of the runtime services table

2.2 Platform Initialization (PI)

DXE does not require PEI, just the Hand-off Blocks (HOBs), PEI is one of many possible implementations **[TODO me]**

2.2.1 UEFI/PI Firmware Images

[For20, 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 [For20, Vol. 3, 2.1]. Flash often offers the ability to restrtict read and write properties differently depending on the storage region [For20, 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 [For20, 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 [@Lon22]. The cursor is on the executable section of a DXE driver.

2.2.2 PI Architecture Firmware Phases

The PI Architecture defines distinct phases. focus will be on dxe and transient system load Figure 2.5

[TODO me]

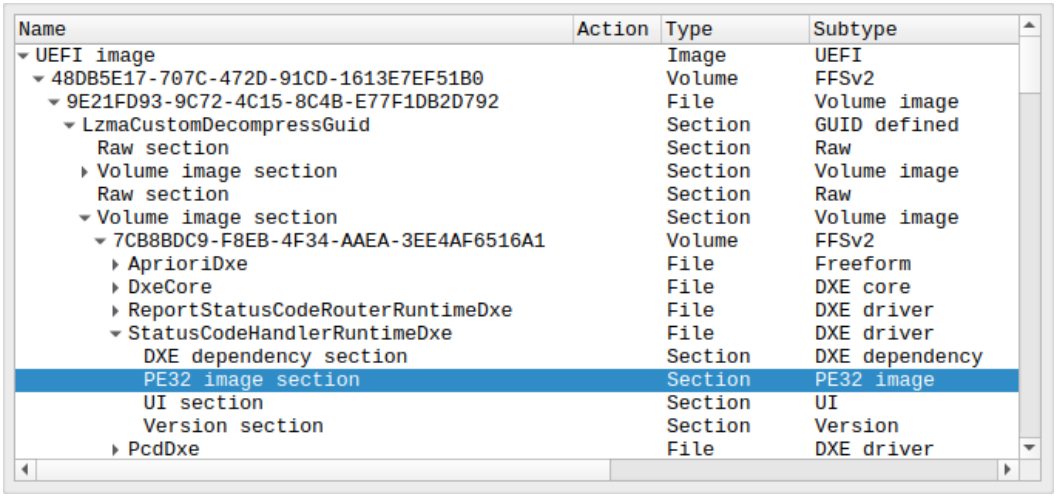


Fig. 2.4.: Open Virtual Machine Firmware (OVMF) opened in UEFITool

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 HOBs to be passed off to the DXE phase [For20, 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

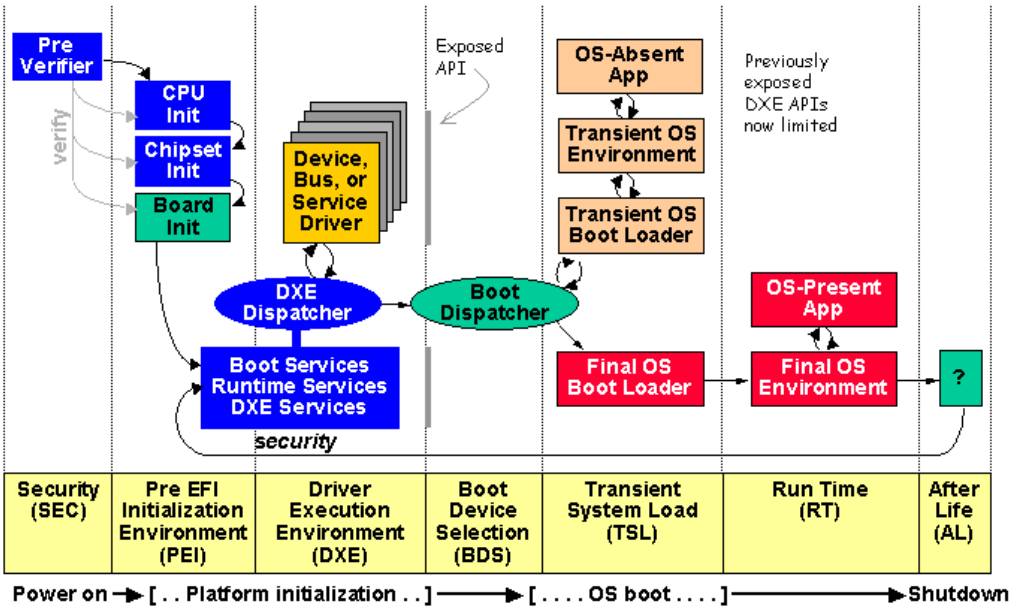


Fig. 2.5.: PI Architecture Firmware Phases [For20, Figure 2-1]

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 [For20, 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) [For20, 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 [For20, 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 [For20, 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 [For21]. 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 [For20, Vol. 2, 2.1]. The DXE foundation is only dependent on the list of HOBs it receives from the previous phase, allowing to use it with a PEI or a different implementation. This also makes it possible to unload the previous stage, freeing up memory [For20, 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[For20, 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 [For20, 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 [For20, 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 [For20, 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 [For20, 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 [For20, 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 [For20, Vol. 2, 10.13].

When the DXE dispatcher is unable to load any new drivers it transfers execution to the *BDS Architectural Protocol* [For20, Vol. 2, 2.4]. This presents the advancement into the BDS phase, but not simaltaniously the end of the DXE phase [For20, 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)

This protocol implements the UEFI boot manager policy as defined in [For21, Section 3] and summarized in subsection 2.1.9. It may return or call the DXE dispatcher directly when discovering additional FVs or when not enough drivers are initialized to successfully boot from a device.

[TODO finish]

attempts to connect boot devices required to load the os discovers volumes containing new drivers calls DXE dispatcher doesnt return when successfully booting OS

UEFI itself only specifies the NVRAM variables used in selecting boot options leaves the implementation of the menu system as value added implementation space [For21]

[For20]

- Initializing console devices
- Loading device drivers
- Attempting to load and execute boot selections

2.2.2.5. Transient System Load (TSL)

[TODO me]

The Transient System Load (TSL) is primarily the OS vendor provided boot loader. Both the TSL and the Runtime Services (RT) phases may allow access to persistent content, via UEFI drivers and UEFI applications. Drivers in this category include PCI Option ROMs.

This phase ends when an OS boot loader calls 'ExitBootServices()'.

boot and runtime services/driver bootloader [For21, Section 13.3] [For21, Section 3.5.1.1]

ExitBootServices()

2.2.2.6. Runtime (RT)

[TODO me]

Boot service drivers have been unloaded and only runtime services are accessible.

runtime services/driver

2.2.2.7. Afterlife (AL)

[TODO me]

The After Life (AL) phase consists of persistent UEFI drivers used for storing the state of the system during the OS orderly shutdown, sleep, hibernate or restart processes.

hibernation sleep

2.2.3 Security

2.2.3.1. Hardware Validated Boot

Secure Boot relies for the firmware as its root of trust, hardware validated boot shifts this trusts out of the firmware image into hardware. amd intel

with pi offering security PPI and dxe protocols for this

PEI Guided Section Extraction PPI Security PPI

Guided Section Extraction Protocol Security Architecture Protocol Security2 Architecture Protocol

2.2.3.2. Firmware Protection

The DXE phase also offers drivers to register notification

DXE SMM Ready to Lock Vol4

flash device security

2.3 TPM

[TODO me]

A Trusted Platform Module (TPM) is a system component which enables trust in computing platforms helps verify if the Trusted Computing Base has been compromised securely storing passwords, certificates and encryption keys in separate state to host only communicating through a well defined interface. store platform measurements that help ensure that the platform remains trustworthy authentication attestation hardware and software implementations software special mode shielding TPM resources from normal execution [Gro08] [Gro19]

how are they used works with bitlocker to protect user data ensure computer has not been tampered with while offline

statically configured, unchangeable data not dynamic and changeable across the boot, [@Tia21]

2.3.1 Measurements

[TODO me]

[@Tia21]

TCG2 Protocol Trusted Computing Group 2 (TCG2) Protocol [Gro16, Section 6.7.3]

PCR Index	PCR Usage
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. 2.1.: [Gro21; MI12]

2.4 UEFI Shell

steal me from neither **[TODO me]**

2.5 EDK II

[TODO me]

build system at least mention that local gcc is used, relevant for porting and headers OVMF

BaseTools package process files compiled by third party tools, as well as text and Unicode files in order to create UEFI or PI compliant binary image files [@Tia22]

3 Windows 11

[TODO what is 11 compared to 10]

3.1 UEFI

3.1.1 Installation

For us to understand how UEFI threats act towards Windows we need to understand how the layout of the Windows installation integrates into the UEFI environment. This 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 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 [AS21, 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 [AS21, 12. The Windows OS Loader].

3.1.2 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 [AS21, 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 [AS21, 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 Windload.efi OS loader [AS21, Section 12]. **[TODO mention ntoskernel.exe]**

[TODO TPM interaction] [AS21, Section 12]

3.1.3 Runtime

get/set variable CapsuleUpdate, but OEM have a lot of differnt own ways to update firmware image

3.2 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 [PS17, 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 [AS21, Section 10]. At the top level it has 9 different keys [AS21, 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 [AS21, 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

[AS21, Section 10]. Some hives even reside entirely in memory as a means of offering hardware configuration through the Registry Application Programming Interface (API).

[TODO maybe fun fact that EFS cant encrypt hives] windows also has a feature called Encrypting File System (EFS) with file system level encryption but it cant be used for registry hives [MI12, Section 9]

3.3 Security

3.3.1 Secure Boot

[@Mic22a]

the two signature DBs Production and UEFI

3.3.2 Trusted Boot

Trusted Boot picks up the process that started with Secure Boot. Trusted Boot protects your PC from malware from the moment you power on your PC until your anti-malware starts can prove the system's integrity

[@Mic22c] [@Mic22a] [@Roy22]

3.3.2.1. KMCI

3.3.2.2. ELAM

3.3.2.3. VSB

Virtualization-based Security VBS formerly Device Guard

3.3.2.4. HVCI

3.3.3 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 [MI12, Section 9]. Windows uses BitLocker, integrated Full Volume Encryption (FVE), aimed to protect system files and data from unauthorized accecss while at rest [@Mic22d], while also verifying boot integrity when used with a TPM [MI12, Section 9]. The en- and decryption of the volume is done by a filter driver beneath the NTFS driver as shown in Figure 3.2. The NTFS driver translates file and directory access into block-wise operations on the volume **[TODO CITE]**, the filter driver receives these block operations, encrypting blocks on write and decrypting blocks on read, while they pass through. This leaves the en- and decryption entirely transparent, making the underlying volume appear decrypted to the NTFS driver [MI12, Section 9]. The encryption of each block is done using a modified version of the Advanced Encryption Standard (AES)128 and AES256 cypher [MI12, 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 [MI12, 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 volume [MI12, Section 9]. The VMK is encrypted by the following protectors:

Startup key stored in a .bek file with a GUID name equaling key identifier in bitlocker meta data [@lib22, Section 2.6]

TPM - tpm only no additional user interaction - tpm with startup key additional usb - tpm with PIN - tpm with startup key and PIN [@Mic22e] with tpm ensures integrity of early boot components and boot configuration tpm usage requires TCG2 compliant UEFI firmware [MI12, Section 9]

tpm is used to *seal* and *unseal* VMK **[TODO PCR table either here or at TPM section]** platform validation profile defaults are Platform Configuration Registers (PCRs) {7, 11} with PCR7 binding {0, 2, 4, 11} without PCR7 binding 11 is required

Recovery key recovery key 48 digits of 8 blocks block is converted to a 16-bit value making up a 128-bit key [@lib22, Section 2.4] when enabling manually, save on non encrypted medium [@Mic22f]

bitlocker device encryption if supported automatically enabled after clean install encrypted with clear key (bitlocker suspended state) non domain account -> recovery key uploaded to microsoft account domain account -> recovery key backed up to active directory domain services (AD DS) clear key removed [@Mic22g]

User key password with max 49 characters [@lib22, Section 2.7]

Clear key unprotected 256-bit key stored on the volume to decrypt vmk [@lib22, Section 2.5] used for suspension

[TODO decide if we add this] With Windows 11 and Windows 10, administrators can turn on BitLocker and the TPM from within the Windows Pre-installation Environment [@Mic22g]

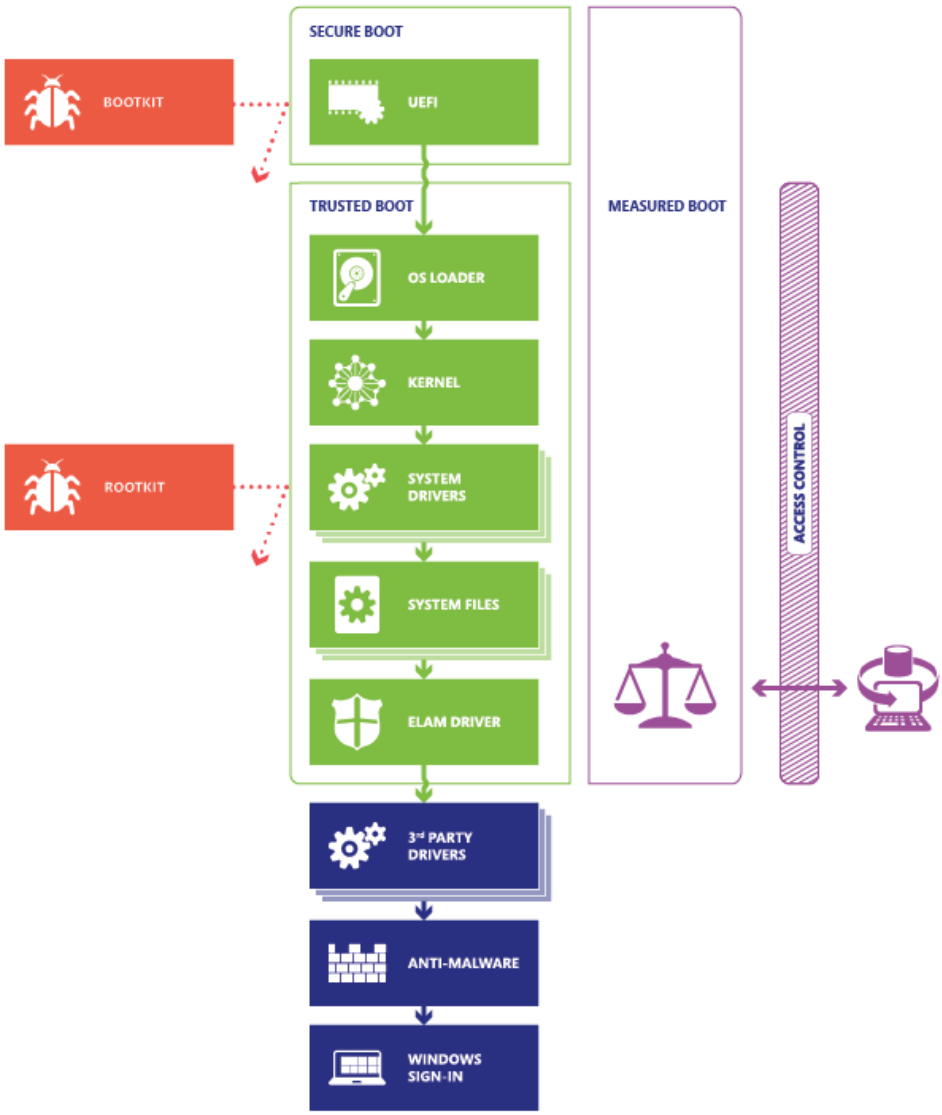


Fig. 3.1.: Windows startup process [@Mic22a]

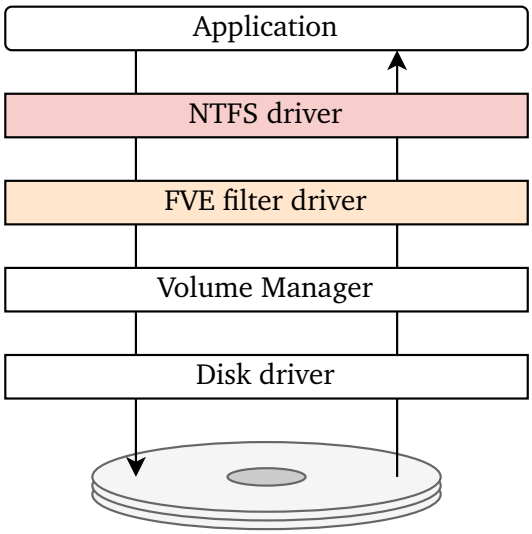


Fig. 3.2.: BitLocker Volume Access Driver Stack (inspired by [MI12, Figure 9-24])

4 Past Threats

Before we implement our own UEFI attacks, we first take a look 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.

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

4.1 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.

4.1.1 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. [@RA21] backs up and replaces the Windows Boot Manager bootmgfw.efi on the ESP. [@SC21] patches the entrypoint of bootmgfw.efi and its copy bootx64.efi in the default boot path, so that it executes malicious code upon launch. [@Qua13] and [@Mat22] are more proof of concept than real attacks and suggest to be used from removable media, but they are also able to be added to the default boot path on an ESP, or generally added

as their own boot entry [@Mat22], 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]**

4.1.2 Rootkit

Firmware rootkits have been rarer and how exactly the firmware images were infected is often not known, [@Hac15] requires booting the target machine from a USB key [@MP20] **[TODO SPI read/write]** [@Res18] dump remove previous NTFS driver add DXE drivers reflash image

The payload itself has usually simply been DXE drivers residing in a firmware volume [@MP20; @Res18], as they are automatically executed by the DXE dispatcher. [@Mat22] compiles its main UEFI payload as a DXE driver and suggesting its usage as a firmware rootkit. [@MB22] 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 [@MB22].

4.2 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 [@Hac15; @MP20; @Res18], memory-based attacks were performed by both root- and bootkit [@Qua13; @Mat22; @SC21; @RA21; @MB22; @RA22]. There is no technical limitation as we show in subsection 6.1.1 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.

4.2.1 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 [For21, Section 13.3.1.1], NTFS drivers are delivered as part of the attack. [@MP20] and [@Res18] seem to use [@Hac15]'s leaked NTFS driver. [@Res18] 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. [@MP20] simply deploys their payload in the Windows startup folder, whose contents, as its names suggests, are executed upon startup.

4.2.2 Memory-based

It seems to be unique to [@SC21] 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. [@RA21; @Qua13] when executed load bootmgfw.efi into memory and apply patches before launching it. [@Mat22]'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 [@Mat22]. [@MB22] 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 [@Mat22; @SC21]. Others deploy payload with elevated privileges [@RA21; @Qua13] or map code directly into kernel space [@MB22; @RA22].

[TODO not THAT importan but would be really cool, as it stands out as really exploiting rootkit capabilities]

5 Test Setup

We perform our attacks against Windows 11 on three different setups, as even though all three UEFI firmwares used, are [For20] compliant, there still are many things left up to the OEMs to decide, when implementing a firmware image.

5.1 QEMU

Our main development setup is an emulated environment using the emulator Quick Emulator (QEMU) [@QEM22] together with the OVMF image, from EFI Development Kit (EDK) II (edk2-stable202208). For Secure Boot we generate our own PK and use the *Microsoft Corporation KEK CA 2011* as KEK and the two signature DBs *Microsoft Windows Production PCA 2011* and *Microsoft Corporation UEFI CA 2011* from Microsoft. The former required for their UEFI executables used during the Windows boot process [@Mic22h] and the latter reserved for third party executables signed at Microsoft's discretion after manual review **[TODO better source]** [@Mic21a]. In the attacks against BitLocker we use *swtpm* for the emulation of a software TPM [@BS22]. Accessing the firmware image with this setup is just done through simple file access.

5.2 Lenovo Ideapad 5 Pro-16ACH6

Lenovo Ideapad 5 Pro-16ACH6

microsoft device guard

secure boot default keys

This can be done by using a spi flash programmer and clamping the physical chip. **[TODO FLASHROM]**

5.3 ASRock A520M-HVS

[TODO describe test setup]

secure boot und bitlocker

A520M-HVS 2.30 latest firmware at time of writing Ryzen 5 5600X Zen 3

secure boot default keys

flashrom -p internal SPI chip emulator. **[TODO EM100]**

6 Attacks

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 with the focus of the attack on BitLocker.

[TODO proper introduction of attack] transfer UEFI execution to Windows execution by installing payload elevated execution of payload

6.1 Neither Secure Boot nor BitLocker

Our first attack is performed on a system with Secure Boot and BitLocker disabled. We implement a bootkit and a rootkit, that deviate the regular boot flow to access the Windows installation and deploy a payload that is automatically executed upon Windows boot.

6.1.1 Bootkit

6.1.1.1. Infection

We have two ways to infect a system, we can either use a bootable medium such as a CD-ROM or Universal Serial Bus (USB) stick with a UEFI application installing the bootkit or a Windows executable mounting the ESP with admin privileges. Booting into the installer application requires either the firmware implementation or the boot order ti prefer booting from the removable media over Windows. This can be forced when booting into the BIOS menu at startup, given that it is not password protected. 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 drop 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, the file path EFI\\Microsoft\\Boot\\bootmgfw.efi will cause our bootkit to be executed. A dump of the Windows boot entry can be seen in Figure 6.1.

```
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\bootmgf
w.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. 6.1.: Windows boot entry dumped via the UEFI shell

6.1.1.2. File access

For our storage-based approach we now need to access the Windows installation from within the UEFI environment to deploy our payload. We can use a fork of the open source NTFS driver ntfs−3g from Tuxera [@tux22], that was ported to the UEFI environment by *pbatard* [@pba22].

We can compile this driver with EDK II to receive a .efi executable file.

[TODO better summary of 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 [For16, Section 1.1]. It offers commands related to boot and general configuration, device and driver management, file system access, networking [For16, Section 5.1] and supports scripting [For16, Section 4]. We can use the file system related commands to test the NTFS driver. Figure 6.2 depicts an exemplary output of an EDK II UEFI shell emulated under QEMU.

The UEFI shell may already be part of the boot options but can always be supplied on a USB stick in the default boot path.

Upon invocation, the shell application performs an initialization during which it **[TODO does what? whats important for us here]** and produces output what is equivalent to the output of the execution of the commands ver and map −terse [For16, Section 3.3]. ver displays the version of the UEFI specification the firmware conforms to [For16, Section 5.3].

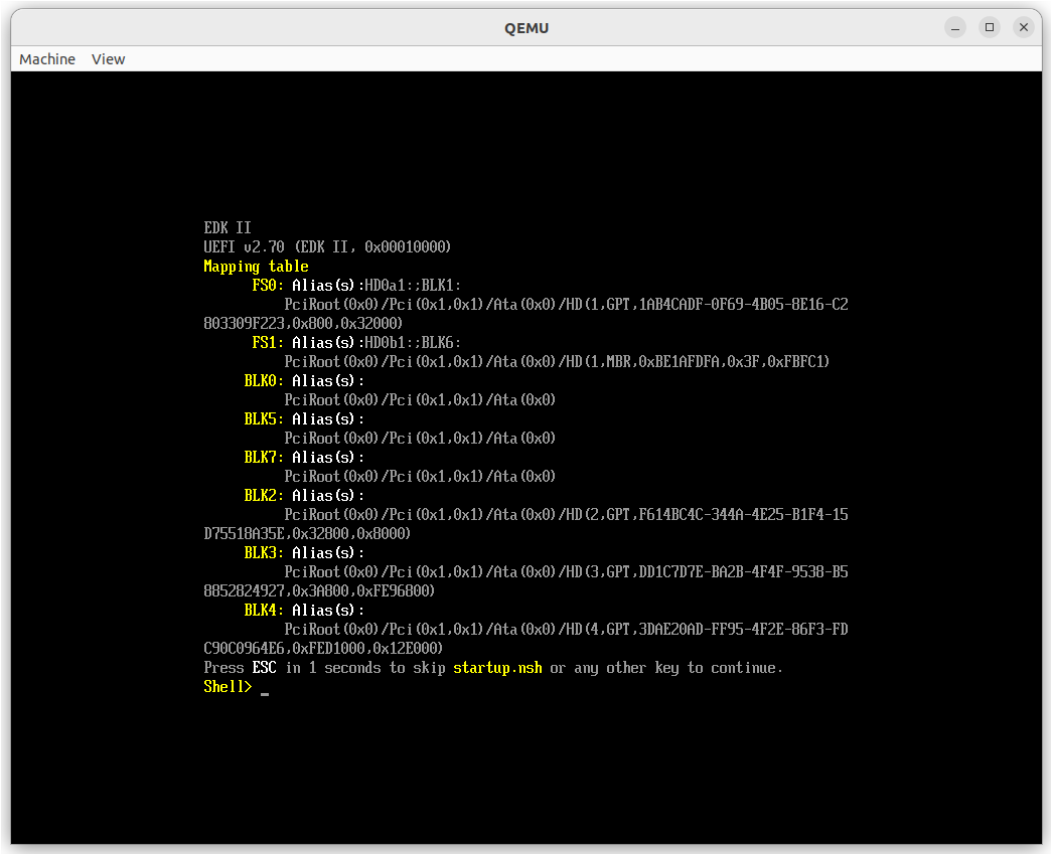


Fig. 6.2.: UEFI command prompt

The map command is very interesting for file access with the shell, it displays a mapping table between user defined alias names and device handles. The aliases can be used instead of a device path when submitting commands via the command line interface. The UEFI shell also produces default mappings, notably for file systems [For16, Section 3.7.2]. These mappings are designed to be consistent across reboots as long as the hardware configuration stays the same, they are comparable to Windows partition letters [For16, Appendix A].

[TODO find in spec what precise mapping mechanism] When we inspect the mapping table we can see FSx: and BLKx: aliases, FSx: maps to file systems and BLKx: to block devices. This identification is performed via instances of the Simple File System Protocol and **[TODO double check]** Block I/O Protocol. The Simple File System Protocol [For21, Section 13.4] provides, together with the File Protocol, file-type access to the device it is installed on [For21, Section 13.5]. The two protocols are independent of the underlying file system the media is formatted with.

Our NTFS UEFI Driver is one such abstraction and needs to be loaded, this is done by first entering the alias, for the file system containing the NtfsDxe.efi. This effectively

switches the console's working directory to be the root of the entered file system, now we can invoke load with the path to the executable. The output indicates whether loading the driver was successful. With the command drivers, we can list all currently loaded drivers and some basic information about them, such as number of devices managed. We can see that the NTFS driver already manages devices.

We can now reset all default mappings with the map -r command to receive an updated list including the file systems now provided by the NTFS driver. The mapping also shows us that the file system now sits on top of a device which previously was only listed as a block device.

As done before we now type the alias of the new file system to switch to NTFS formatted file system. With ls we can list the current directory's content and confirm by the presence of the Windows folder that we are on the volume containing the Windows installation. **[TODO maybe vol]**

[TODO Windows file access privileges] We now navigate into the Windows folder to test whether we have unrestricted read and write access, since is not the case if done by an unprivileged user when performed from within Windows. Accessing folders and viewing their contents is possible but creation of a new folder fails.

Upon debugging the NTFS driver it appears to be that the drivers falls back to read only when it encounters a file that indicates that the Windows system is in hibernation mode. Windows seems to have hibernation enabled by default and as such our rootkit should not rely on it being disabled, we can change the code of the NTFS driver to not fallback when encountering this file. **[TODO this is might not be the hibernation file but something else]** On our hardware setups we noticed that the firmware already is shipped with an NTFS driver, in the case of our rootkit we would be able to remove this driver, but we can implement a solution applying to both UEFI payloads. We can change the NTFS driver to install the Simple File System Protocol under a different GUID instead of gEfiSimpleFileSystemProtocolGuid, making it possible to install our instance alongside any other driver's instance on the same controller. TheGUID can then be used to retrieve our specific protocol instance in the root- and bookit. We also open the protocols, consumed by the driver, in a non-exclusive way. This prevents our NTFS driver from being removed off of the controller as well as being blocked from opening the protocols in the first place [For21, Section 7.3]. This would be a likely scenario as filesystem drivers are encouraged to get exclusive control over their block device [For21, Section 13.5].

We now know that provided we get to load the NTFS driver we can now access a Windows installation and subsequently the entire data of unencrypted hard drives.

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 [For21, 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 [For21, Section 13.4]. This instance can then be used to open and read our payload with the absolute path on the ESP into memory.

6.1.1.3. 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.

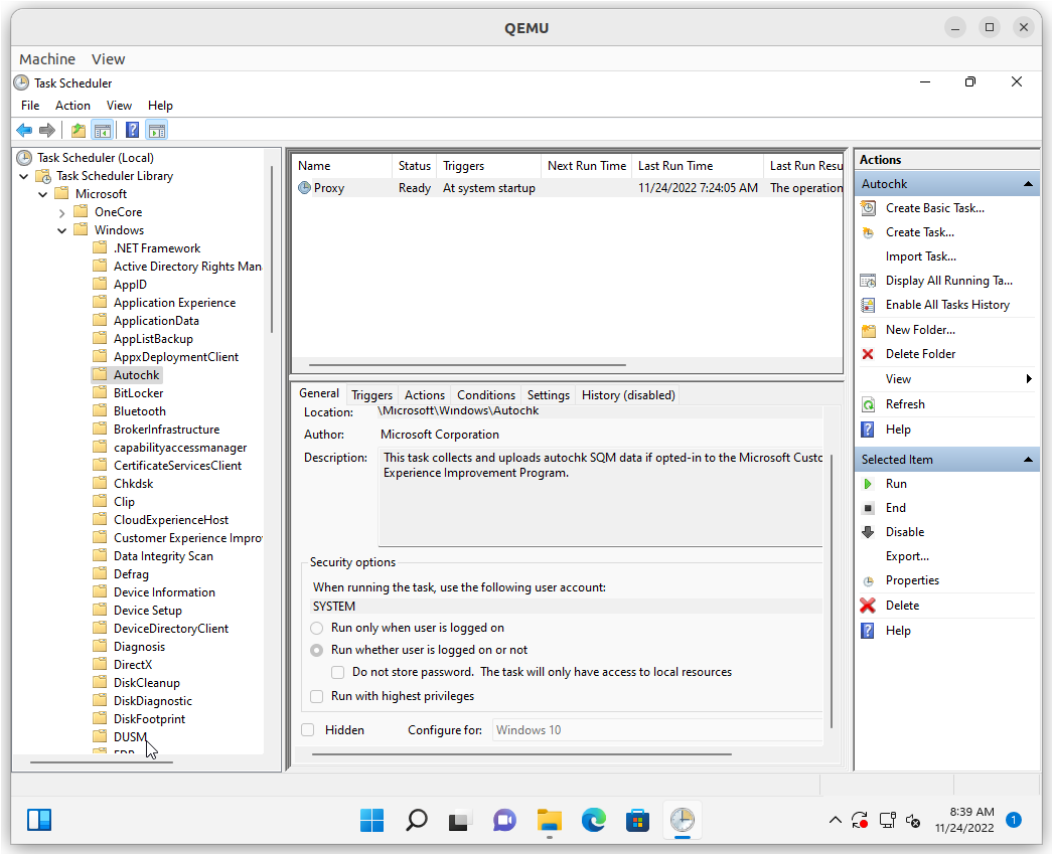
[TODO DLL proxy loading] [TODO modifying Windows Executables KMCI]

The Task Scheduler is a Windows service responsible for managing the automatic execution of background tasks [AS21, 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 [@Mic21b]. A task can perform various actions upon invocation [@Mic21c], 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 [@Mic20]. 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

```
1 %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 [@Mic21d]. 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 [@Mic21e]. 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
[AS21, 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 [@Mic21f]. 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 [@Mic21g].

[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 [@Nor14]. 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 *EFI Loaded Image Protocol* [For21, Section 9.1] instance installed to our bootkit to the Windows Boot Manager.

6.1.2 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 [For20, 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.

6.1.2.1. 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 5 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 [For20, 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 [For20, 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.

6.2 Secure Boot

Our second attack is against systems with Secure Boot enabled.

6.2.1 Bootkit

For the installation via removable media we have to assume that the BIOS menu is password protected, as we otherwise could simply turn off Secure Boot. This makes the likelihood of an infection via this method smaller since we now solely rely on the boot order/firmware policy to prefer removable media. Even given this assumption we promptly see that Secure Boot already denies execution of the installer when trying to boot it. The same denial is observable for the bootkit itself, when using our Windows installer. 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.

6.2.2 Rootkit

When installing our rootkit on section 5.3, we observe that Secure Boot is not applying its verification to our DXE drivers, as they are still being executed without any restrictions. When we look at the reference implementation in EDK II, we can see why. Listing 6.1 shows, that the image origin dictates which is applied. 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 [...]” [For21, p. 32.5.3.2]. This behavior was reproducible on all our hardware setups, likely in order to prevent accidentally entirely unbootable firmware or to reduce boot time.

```
1  EFI_STATUS
2  EFIAPI
3  DxeImageVerificationHandler(..., EFI_DEVICE_PATH_PROTOCOL *File, ...)
4  {
5      // ...
6
7      switch (GetImageType(File))
8      {
9      case IMAGE_FROM_FV:
10         Policy = ALWAYS_EXECUTE;
11         break;
12
13     case IMAGE_FROM_OPTION_ROM:
```

```
14              Policy = PcdGet32(PcdOptionRomImageVerificationPolicy);
15              break;
16
17      case IMAGE_FROM_REMOVABLE_MEDIA:
18              Policy = PcdGet32(PcdRemovableMediaImageVerificationPolicy);
19              break;
20
21      case IMAGE_FROM_FIXED_MEDIA:
22              Policy = PcdGet32(PcdFixedMediaImageVerificationPolicy);
23              break;
24
25      default:
26              Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
27              break;
28      }
29
30      // ...
31  }
```

Listing 6.1: DxeImageVerificationHandler Reference Implementation

6.3 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 6.2, as well as additionally dictating the BitLocker default validation profile Windows uses as mentioned in subsection 3.3.3. 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 6.3.4. 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.

6.3.1 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 6.3, 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 [AS21, Section 12].

Fig. 6.3.: BitLocker Recovery Prompt

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 booting 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, 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.

We can look at how the user is influenced in their decision, taking a closer look at the recovery prompt in Figure 6.3, 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 (a bootable medium might change boot behavior, invalidating the PCRs). Of course this will not resolve anything in the case of an infection, but that is all the information displayed about what might have caused the issue. The rest is only about helping the user to find their recovery key to enter. This is ground enough to argue that is very reasonable to assume that the average user will react by entering their recovery key without having any malicious behavior in mind.

6.3.2 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.3.

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 6.2.

```
1  EFI_STATUS
2  EFIAPI
3  EntryPoint(
4      IN EFI_HANDLE ImageHandle,
5      IN EFI_SYSTEM_TABLE *SystemTable)
6  {
7      gBS = SystemTable->BootServices;
8
9      EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;
10
11     gBS->HandleProtocol(SystemTable->ConsoleInHandle,
12                         &gEfiSimpleTextInputExProtocolGuid,
13                         (VOID **)&SimpleTextInEx);
14
15     UINTN EventIndex;
16     gBS->WaitForEvent(1, &SimpleTextInEx->WaitForKeyEx, &EventIndex);
17
18     EFI_KEY_DATA KeyData;
19     SimpleTextInEx->ReadKeyStrokeEx(SimpleTextInEx, &KeyData);
20
21     // do something with key press
22
23     return EFI_SUCCESS;
24 }
```

Listing 6.2: 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 [For21, Section 7.3]. Listing 6.2 shows how HandleProtocol can be used to receive the Simple Text Input Ex Protocol instance installed on the active console input device [For21, 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 [@Tia18, Section 8]. In Listing 6.3 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.

```
1  typedef struct
2  {
3      UINTN Signature;
4      EFI_DISK_IO_PROTOCOL DiskIo;
5      EFI_BLOCK_IO_PROTOCOL *BlockIo;
6  } DISK_IO_PRIVATE_DATA;
7
8  EFI_STATUS
9  EFIAPI
10 DiskIoReadDisk(
11     IN EFI_DISK_IO_PROTOCOL *This,
12     IN UINT32 MediaId,
13     IN UINT64 Offset,
```

```
14      IN UINTN BufferSize,
15      OUT VOID *Buffer)
16  {
17      DISK_IO_PRIVATE_DATA *Private;
18
19      Private = DISK_IO_PRIVATE_DATA_FROM_THIS(This);
20
21      Private->BlockIo->ReadBlocks(...);
22  }
```

Listing 6.3: Example of a driver using private data in the implementation of the *Disk I/O Protocol* (snippets from [@Tia18, 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 6.4;

```
1   EFI_INPUT_READ_KEY_EX gOriginalReadKeyStrokeEx;
2
3   EFI_STATUS EFIAPI ReadKeyStrokeExHook(IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
4                                         OUT EFI_KEY_DATA *KeyData);
5   {
6       gOriginalReadKeyStrokeEx(This, KeyData);
7
8       // log keystrokes
9   }
10
11  VOID HookSimpleTextInEx()
12  {
13      gBS->LocateHandleBuffer(ByProtocol, &gEfiSimpleTextInputExProtocolGuid,
```

```
14                                  NULL, &HandleCount, Handles);
15      for (UINTN i = 0; i < HandleCount; ++i)
16      {
17          EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;
18          status = gBS->HandleProtocol(Handles[i],
19                                       &gEfiSimpleTextInputExProtocolGuid,
20                                       (VOID **)&SimpleTextInEx);
21
22          gOriginalReadKeyStrokeEx = SimpleTextInEx->ReadKeyStrokeEx;
23
24          SimpleTextInEx->ReadKeyStrokeEx = ReadKeyStrokeExHook;
25      }
26  }
```

Listing 6.4: 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 [MI12, 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 6.3 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.

6.3.3 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 [@Aor22].

In subsection 3.3.3 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 6.5

```
1 [Protocols]
2   gEfiDiskIoProtocolGuid
3   gEfiDiskIo2ProtocolGuid
4   gEfiBlockIoProtocolGuid
5   gEfiBlockIo2ProtocolGuid
6   gEfiSimpleFileSystemProtocolGuid
7   gEfiUnicodeCollationProtocolGuid
8   gEfiUnicodeCollation2ProtocolGuid
9   gEfiDevicePathToTextProtocolGuid
```

Listing 6.5: 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 [For21, 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 [For21, 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 3.2, 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.6. We can then use Dislocker on read and write operations to implement our own filter driver as shown in Figure 6.4.

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

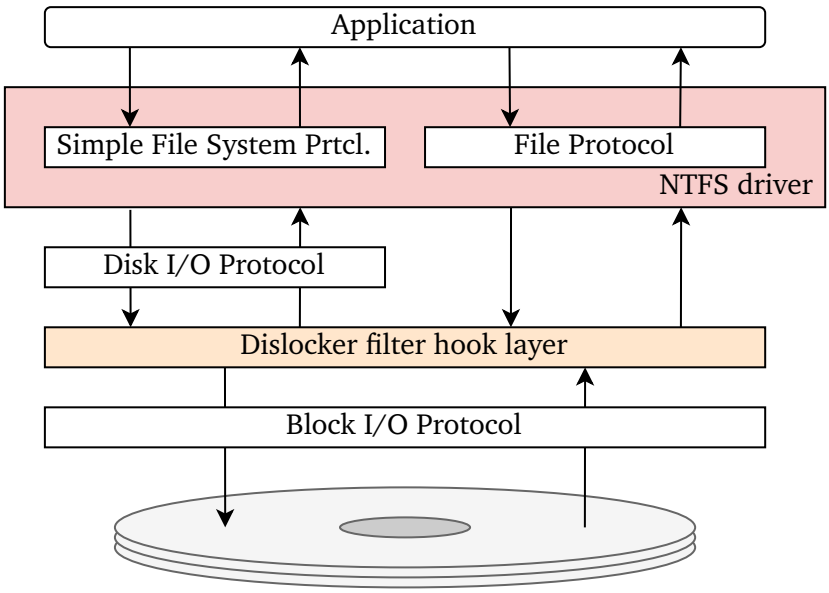


Fig. 6.4.: Dislocker Volume Access Protocol Stack

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.

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.

[TODO if we want we can explain this] [@Use22] hook ExitBootServices enable hook write payload import registry key disable hook

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 [@Mic22i] 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.

6.3.4 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.

hook TCG2 Protocol [Gro16, Section 6.7.3] TPM communication receive bitlocker VMK key and send to dislocker [@lib22] [@And19]

This increases the persistence and applicability of our attack immensely.

7 Results

We were able to implement UEFI attacks in the form of a UEFI firmware rootkit and a UEFI bootloader rootkit (bootkit), with both being able to deploy Windows level payload from within the UEFI environment using an NTFS drivers. Through our UEFI port of the reged utility we were able to modify the Windows registry, so that our Windows payload is executed with the privileges of the built-in local system account. The execution is done by the Task Scheduler at system boot. With Secure Boot enabled we showed that our bootkit was denied execution, while the execution of our rootkit is left unaffected. Although affecting infection by restricting software access to the firmware image. When BitLocker is used with a TPM and the default validation profile {0, 2, 4, 11} our root- and bootkit trigger the BitLocker recovery prompt, from which we our *BitLogger* was able to log 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 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 using a Secure Boot configuration that uses only Microsoft's signature DB required to boot Windows. This forces a default validation profile of {7, 11}, leaving out PCR0. Our rootkit attack was able to gain access to this type of system without requiring any prior knowledge or additional user input.

8 Discussion

Our attacks show, the differences between UEFI firmware rootkits and UEFI bootkits.

bootkit much easier usb stick, from windows windows installer if no password present we can disable secure boot in case of physical presence it may require to change boot order physical presence with bootable usb stick (defeated by secure boot) genrally defeated by secure boot where as the rootkit isnt even if secure boot was implemented for FV images, it could be patched if the validation change starts within the image

barrier of entry is higher exploit to overwrite spi flash or be delivered with supply chain difficult physical presence remove spi chip and emulate spi chip or modify chip content but high payoff with persistence

bootkit moves with hard drive but can be overwritten by fresh install rootkit persistence across reinstallations or hard drive replacements

didnt prevent firmware update overwriting our payload generally the bitlocker reocvery prompt can raise suspicion and may lead to investigations and the threat to be found BitLogger is more of a last resort and a social engineering aspect comparable to phishing implications of windows secure boot PCR7 binding and use of secure boot system integrity check and validation profile 7, 11 is a bad decision of microsoft, that for example allows stolen laptops to be unlocked when infecting the firmware with our rootkit

it is generally very easy to attack windows from the UEFI environment and there is little that they can do, as especially all windows code can be patched

8.1 Mitigations

bios password against secure boot removal or bootkit installation from USB

windows cant assume what the implementation of ReadKeyStrokeEx looks like (normally function patching might have a jump etc, which we dont even have here)

hardware validated boot to start the validation change from outside the image

inaccessible spi flash

tpm + pin/usb detectability

8.1.1 User awareness

you can change recovery message and URL in BCD hive

recovery guide

what causes bitlocker recovery - password wrong too often - TPM 1.2, changing the BIOS or firmware boot device order - Having the CD or DVD drive before the hard drive in the BIOS boot order and then inserting or removing a CD or DVD - Failing to boot from a network drive before booting from the hard drive. - Docking or undocking a portable computer - Changes to the NTFS partition table on the disk including creating, deleting, or resizing a primary partition. - Entering the personal identification number (PIN) incorrectly too many times - Upgrading critical early startup components, such as a BIOS or UEFI firmware upgrade - Updating option ROM firmware graphics card - Adding or removing hardware - REMOVING, INSERTING, OR COMPLETELY DEPLETING THE CHARGE ON A SMART BATTERY ON A PORTABLE COMPUTER - Pressing the F8 or F10 key during the boot process what does the recovery screen say Figure 6.3

Enables end users to recover encrypted devices independently by using the Self-Service Portal

googeln wie legitime recovery key prompt reaktion aussieht

enterprise policy recovery key einschraenkbar?

enterprise policy on recovery key loss

vermitteln was das prompt bedeuten koennte

aber kann man einfach nicht anzeigen lassen

Security Flaw of entering a Recovery Password in an inheritly unsafe System

enterprise doesnt hand out recovery keys and instead receives hard drive

!!!!!!!!!!!!!!!!!!!!!!!!! without hardware chain of trust a compromised system can patch/change any software and fixes are impossible

phishing prompts on their own

9 Conclusion

9.1 Achieved Goals

when we are already in the image we can gain full control over the system system cant be trusted anymore e.g. uefi services full file access escalate it to local system level execution bitlocker has the flaw of allowing to enter criticial information into an inherently untrustable system on the other hand one could force such a prompt themselves mere exisitence of a recovery key is a security flaw

9.2 Future Work

tpm and pin capsule update exploit in tpm measruement chain that results in not being measured can exploit the tg2 hook directly to retrieve the vmk memory based rootkit hypervisor kernel security boot vs runtime rootkit

57

Bibliography

[ZRM17] V. Zimmer, M. Rothman, and S. Marisetty, *Developing with the Unified Extensible Firmware Interface, Third Edition*. Berlin, Boston: De|G Press, 2017 (cit. on pp. 3, 4, 6, 8).

[For21] U. Forum, "Uefi specification, version 2.9," UEFI Forum, Tech. Rep., Mar. 2021 (cit. on pp. 4–8, 10–12, 16, 17, 30, 35–37, 40, 42, 47, 50).

[Gro05] R. N. W. Group, "A universally unique identifier (uuid) urn namespace," IETF, RFC 4122, Jul. 2005 (cit. on p. 4).

[For20] U. Forum, "Uefi platform initialization (pi) specification, version 1.7 errata a," UEFI Forum, Tech. Rep., Apr. 2020 (cit. on pp. 9, 13–17, 31, 40, 41, 76, 78).

[Gro08] T. C. Group, "Trusted platform module (tpm) summary," Trusted Computing Group, Tech. Rep., 2008 (cit. on p. 19).

[Gro19] T. C. Group, "Trusted platform module library, part 1: Architecture, level 00 revision 01.59," Trusted Computing Group, Tech. Rep., 2019 (cit. on p. 19).

[Gro21] T. C. Group, "Tcg pc client platform firmware profile specification," Trusted Computing Group, Tech. Rep., 2021 (cit. on p. 20).

[MI12] D. A. S. Mark E. Russinovich and A. Ionescu, *Windows Internals, Part 2*, 6th ed. Microsoft Press, Sep. 2012 (cit. on pp. 20, 23, 24, 27, 49).

[Gro16] T. C. Group, "Tcg efi protocol specification for tpm 2.0 version 1.0 revision 0.13," Trusted Computing Group, Tech. Rep., 2016 (cit. on pp. 19, 52).

[AS21] M. E. R. Andrea Allievi Alex Ionescu and D. A. Solomon, *Windows Internals, Part 2* (Developer Reference), 7th ed. Pearson Education, Inc., Sep. 2021 (cit. on pp. 21–23, 38, 44).

[PS17] M. E. R. Pavel Yosifovich Alex Ionescu and D. A. Solomon, *Windows Internals, Part 1*, 7th ed. Microsoft Press, May 2017 (cit. on p. 22).

[For16] U. Forum, "Uefi shell specification, revision 2.2," UEFI Forum, Tech. Rep., Jan. 2016 (cit. on pp. 34, 35).

Webpages

[@cro21] crowdstrike. “Rootkit malware.” (Sep. 2021), [Online]. Available: https://www.crowdstrike.com/cybersecurity-101/malware/rootkits/ (cit. on p. 1).

[@Tec21] Techtarget. “Definition rootkit.” (Oct. 2021), [Online]. Available: https://www.techtarget.com/searchsecurity/definition/rootkit (cit. on p. 1).

[@Mic22a] Microsoft. “Secure the windows boot process.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/secure-the-windows-10-boot-process (cit. on pp. 1, 23, 26).

[@For22] U. Forum. “Specifications and tools.” (2022), [Online]. Available: https://uefi.org/specsandtesttools (cit. on p. 3).

[@Mic22b] Microsoft. “Windows and gpt faq.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-and-gpt-faq?view=windows-11 (cit. on p. 5).

[@Tia18] TianoCore. “Edk ii driver writer’s guide.” (Apr. 2018), [Online]. Available: https://edk2-docs.gitbook.io/edk-ii-uefi-driver-writer-s-guide (cit. on pp. 6–8, 47, 48).

[@Tia19] TianoCore. “Understanding uefi secure boot chain.” (Jun. 2019), [Online]. Available: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/ (cit. on p. 11).

[@Mic19] Microsoft. “Microsoft windows authenticode portable executable signature format, version 1.0.” (Jun. 2019), [Online]. Available: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/ (cit. on p. 12).

[@Lon22] LongSoft. “Uefitool.” (Dec. 2022), [Online]. Available: https://github.com/LongSoft/UEFITool (cit. on p. 13).

[@Tia21] TianoCore. “Trusted boot chain.” (Mar. 2021), [Online]. Available: https://tianocore-docs.github.io/edk2-TrustedBootChain/release-1.00/ (cit. on p. 19).

[@Tia22] TianoCore. “Edk ii project github repository.” (2022), [Online]. Available: https://github.com/tianocore/edk2 (cit. on p. 20).

[@Mic22c] Microsoft. “Secure boot and trusted boot.” (Nov. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/trusted-boot (cit. on p. 23).

[@Roy22] J. B. Roy. “Understanding windows trusted boot - integrity check and elam.” (Mar. 2022), [Online]. Available: https://www.anoopcnair.com/understanding-windows-trusted-boot/ (cit. on p. 23).

[@Mic22d] Microsoft. “Bitlocker.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview (cit. on p. 24).

[@lib22] libde. “Bitlocker drive encryption (bde) format specification.” (Feb. 2022), [Online]. Available: https://github.com/libyal/libbde/blob/main/documentation/BitLocker%20Drive%20Encryption%20(BDE)%20format.asciidoc (cit. on pp. 24, 25, 52).

[@Mic22e] Microsoft. “Bitlocker countermeasures.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-countermeasures (cit. on p. 24).

[@Mic22f] Microsoft. “Bitlocker basic deployment.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-basic-deployment (cit. on p. 24).

[@Mic22g] Microsoft. “Overview of bitlocker device encryption in windows.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-device-encryption-overview-windows-10 (cit. on p. 25).

[@RA21] G. Research and K. L. Analysis Team. “Finspy.” (Sep. 2021), [Online]. Available: https://securelist.com/finspy-unseen-findings/104322/ (cit. on pp. 28–30).

[@SC21] M. Smolár and A. Cherepanov. “Especter.” (Oct. 2021), [Online]. Available: https://www.welivesecurity.com/2021/10/05/uefi-threats-moving-esp-introducing-especter-bootkit/ (cit. on pp. 28–30).

[@Qua13] Quarkslab. “Dreamboot.” (Apr. 2013), [Online]. Available: https://github.com/quarkslab/dreamboot (cit. on pp. 28–30).

[@Mat22] Mattiwatti. “Efiguard.” (Aug. 2022), [Online]. Available: https://github.com/Mattiwatti/EfiGuard (cit. on pp. 28–30).

[@Hac15] Hackingteam. “Vector-edk.” (Apr. 2015), [Online]. Available: https://github.com/hackedteam/vector-edk (cit. on pp. 29, 30).

[@MP20] I. K. Mark Lechtik and Y. Parshin. “Mosaicregressor.” (Oct. 2020), [Online]. Available: https://media.kasperskycontenthub.com/wp-content/uploads/sites/43/2020/10/07080558/MosaicRegressor_Technical-details.pdf (cit. on pp. 29, 30).

[@Res18] E. Research. “Lojax.” (Sep. 2018), [Online]. Available: https://www.welivesecurity.com/wp-content/uploads/2018/09/ESET-LoJax.pdf (cit. on pp. 29, 30).

[@MB22] D. L. Mark Lechtik Vasily Berdnikov and I. Borisov. “Moonbounce: The dark side of uefi firmware.” (Jan. 2022), [Online]. Available: https://securelist.com/moonbounce-the-dark-side-of-uefi-firmware/105468/ (cit. on pp. 29, 30).

[@RA22] G. Research and K. L. Analysis Team. “Cosmicstrand.” (Jul. 2022), [Online]. Available: https://securelist.com/cosmicstrand-uefi-firmware-rootkit/106973/ (cit. on pp. 29, 30).

[@QEM22] QEMU. “Qemu.” (2022), [Online]. Available: https://www.qemu.org/ (cit. on p. 31).

[@Mic22h] Microsoft. “Windows secure boot key creation and management guidance.” (May 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/manufacture/desktop/windows-secure-boot-key-creation-and-management-guidance?view=windows-11 (cit. on p. 31).

[@Mic21a] Microsoft. “Updated: Uefi signing requirements.” (Jan. 2021), [Online]. Available: https://techcommunity.microsoft.com/t5/hardware-dev-center/updated-uefi-signing-requirements/ba-p/1062916 (cit. on p. 31).

[@BS22] S. Berger and D. Safford. “Swtpm - software tpm emulator.” (Dec. 2022), [Online]. Available: https://github.com/stefanberger/swtpm (cit. on p. 31).

[@tux22] tuxera. “Ntfs-3g.” (Nov. 2022), [Online]. Available: https://github.com/tuxera/ntfs-3g (cit. on p. 34).

[@pba22] pbatard. “Ntfs-3g.” (Dec. 2022), [Online]. Available: https://github.com/pbatard/ntfs-3g (cit. on p. 34).

[@Mic21b] Microsoft. “Task triggers.” (Aug. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/taskschd/task-triggers (cit. on p. 38).

[@Mic21c] Microsoft. “Task actions.” (Aug. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/taskschd/task-actions (cit. on p. 38).

[@Mic20] Microsoft. “Task security contexts.” (Feb. 2020), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/taskschd/security-contexts-for-running-tasks (cit. on p. 38).

[@Mic21d] Microsoft. “Rundll32.” (Mar. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/rundll32 (cit. on p. 38).

[@Mic21e] Microsoft. “Autochk.” (Mar. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/autochk (cit. on p. 38).

[@Mic21f] Microsoft. “Whoami.” (Mar. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/whoami (cit. on p. 39).

[@Mic21g] Microsoft. “Localsystem account.” (Jan. 2021), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/services/localsystem-account (cit. on p. 39).

[@Nor14] P. Nordahl-Hagen. “Chntpw.” (Feb. 2014), [Online]. Available: http://pogostick.net/~pnh/ntpasswd/ (cit. on p. 39).

[@Aor22] Aorimn. “Dislocker.” (Sep. 2022), [Online]. Available: https://github.com/Aorimn/dislocker (cit. on p. 49).

[@Use22] User71491. “Exitbootservices hooking.” (2022), [Online]. Available: https://wikileaks.org/ciav7p1/cms/page_36896783.html (cit. on p. 52).

[@Mic22i] Microsoft. “Bitlocker: Use bitlocker drive encryption tools to manage bitlocker.” (Dec. 2022), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-use-bitlocker-drive-encryption-tools-to-manage-bitlocker (cit. on p. 52).

[@And19] D. Andzakovic. “Extracting bitlocker keys from a tpm.” (Mar. 2019), [Online]. Available: https://pulsesecurity.co.nz/articles/TPM-sniffing (cit. on p. 52).

List of Figures

List of Tables

List of Source Code Listings

List of Acronyms

ACPI Advanced Configuration and Power Interface
AES Advanced Encryption Standard
AL Afterlife
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
CSM Compatibility Support Module
DB Data Base
DEPEX Dependency Expression
DLL Dynamically Linked Library
DXE Driver Execution Environment
EDK EFI Development Kit
EFI Extensible Firmware Interface
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
GUID Globally Unique Identifier
HOB Hand-off Block
I/O Input/Output
KEK Key Exchange Key
LBA Logical Block Address

66

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
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
PK Platform Key
PPI PEIM-to-PEIM Interface
QEMU Quick Emulator
RAM Random Access Memory
ROM Read-Only Memory
RT Runtime
SEC Security
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
VMK Volume Master Key
XML Extensible Markup Language

Appendix

A

A.1 Protocols

```
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.1: Loaded Image 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.2: Driver Binding 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.3: Simple Text Input Ex 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.4: Simple File System and File 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.5: Disk 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.6: 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.7: TCG2 Protocol

A.2 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 [For20, Vol. 3, Table 3-3]

A.3 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 [For20, Vol. 3, Table 3-4]