

A Practical Analysis of UEFI Threats Against Windows 11

Joshua Machauer

June 21, 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

- | | |
|--------------------|--|
| <i>1. Reviewer</i> | Prof. Dr. Jean-Pierre Seifert
Electrical Engineering and Computer Science
Technische Universität Berlin |
| <i>2. Reviewer</i> | Prof. Dr. Stefan Schmid
Electrical Engineering and Computer Science
Technische Universität Berlin |
| <i>Supervisors</i> | Hans Niklas Jacob and Christian Werling |

June 21, 2022

Joshua Machauer

A Practical Analysis of UEFI Threats Against Windows 11

Bachelor's Thesis, June 21, 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 and Berlin

Eigenständigkeitserklärung

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 10. Juli 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.

Abstract (deutsch)

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

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. [crowdstrike; techtarget] [TODO consult abstract for similar definition, how easy uefi makes it to write hardware independent payload] Firmware rootkits targets 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. [crowdstrike]

look at UEFI + threats against windows danger of uefi infection in recent years root and bootkits have popped up in the wild and been analysed differences of root-/bootkits reason about infection scenarios we will discuss their commonalities attack vectors: - storage based - memory based implement a storage based ourselves analyse security mechanism to prevent these attacks by attempting an attack itself discuss security mechanisms we encounter increasing security mechanisms add onto past threats by attacking bitlocker reflect their weaknesses how to potentially evade them - analyse countermeasures against UEFI threats - Trusted Boot: KMCI from windows - Secure Boot - TPM - Bitlocker - firmware lock + signed capsule update -

1.1 Overview

We start off in Chapter 2 by introducing all necessary knowledge about the UEFI! (UEFI!) environment, defined by the UEFI! and PI! (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.

UEFI!/PI!

“The **UEFI!** specifications define a new model for the interface between personal-computer **OS!** (**OS!**) and **PF!** (**PF!**). [...] Together, these provide a standard environment for booting an **OS!** and running pre-boot applications” [**uefi-spec-overview**]. The specifications making up this model are:

- **ACPI!** (**ACPI!**) Specification
- **UEFI!** Specification
- **UEFI!** Shell Specification
- **UEFI! PI!** Specification
- **UEFI! PI!** Distribution Packaging Specification
- **TCG!** (**TCG!**) **EFI!** (**EFI!**) Platform Specification
- **TCG! EFI!** Protocol Specification

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 [**beyond-bios**].

The **UEFI! PI!** [**TODO mention of EFI and frame work into UEFI and pi?**]

2.1 **UEFI! (UEFI!)**

[**TODO MEMORY LAYOUT no memory protection, RWE everywhere**]

It was designed to replace the legacy **BF! BIOS!** (**BIOS!**) [**TODO which wasnt very standardized**], while also providing backwards compatibility by defining the **CSM!** (**CSM!**) allowing **UEFI!** firmware to boot legacy **BIOS!** applications.

boot- and runtime service functions for the bootloader and os to call datatables containing platform-related information - 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

2.1.1 **GUID! (GUID!)**

GUID!

2.1.2 **GPT! (GPT!)**

MBR! (MBR!) boot code, four partitions **GUID!** for uniquely identifying each partition **GUID!** for partition type content **GPT! LBA! (LBA!)** legacy **MBR!** or protective **MBR!**

[uefi-spec]

2.1.3 **ESP! (ESP!)**

FAT! (FAT!)³² [uefi-spec] can reside on any media that is supported by EFI Boot Services [uefi-spec]

2.1.4 **UEFI! Images**

Images that can be executed in the **UEFI!** environment are of the **PE32! (PE32!)** + file format, which is a relocatable, meaning they can either be executed in place or loaded into arbitrary memory addresses. They support IA, ARM, RISC-V and x86 CPU architectures. 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. Loading and execution of images are two separate steps, at first memory large enough to hold the image is allocated, then relocation fix-ups are [uefi-spec]

UEFI Images are files containing executable code, they use a subset of the PE32+ (Microsoft Portable Executable and Common Object File Format Specification) format with a modified header signature. The format comes with relocation tables, this makes it possible that the images can be loaded at non predetermined addresses.

The images come in three different types: - UEFI Applications - UEFI Boot Services Drivers - UEFI Runtime Drivers

Main differences between these types is how and where they reside in memory. Applications are always unloaded when they return execution while drivers are only unloaded when they return an error code. Boot Services are unloaded after the bootloader calls 'ExitBootServices()' while Runtime Drivers remain.

2.1.4.1 UEFI! Applications

Applications example efi shell loaded by boot manager or other applications return or calling exit specifically always unloaded from memory

2.1.4.2 UEFI OS Loaders

example windows boot manager normally take over control from the firmware upon load behaves like a normal UEFI application - only use memory allocated from the firmware - only use services/protocols to access devices that the firmware exposes - conform to driver specifications to access hardware on error can return allocated resources with Exit boot service with error specific information given in ExitData on success take full control with ExitBootServices boot service all boot services in the system are terminated, including memory management UEFI OS loader now responsible

2.1.4.3 UEFI Drivers

loaded by boot manager, UEFI firmware (DXE foundation), or other applications example payload unloaded only when returning error code persistent on success boot and runtime drivers only difference is that runtime are available after Exit-BootServices was called boottime drivers are terminated and memory is released runtime drivers are fixed up with virtual mappings upon SetVirtualAddressMap call has to convert its allocated memory

2.1.5 **UEFI! Driver Model**

2.1.6 **Protocols and Handles**

[uefi-spec]

consists of GUID and protocol interface structure containing functions and instance data used to access a device

provide software abstractions for devices such as consoles, mass storage devices and networks They can also be used to extend the number of generic services that are available in the platform [uefi-spec] boot services provide function to install, locate, open, close and monitor protocols [uefi-spec] identified with guids

2.1.7 **Variables**

key/value pairs store arbitrary data passed between UEFI environment and applications/os loaders type of data is defined through usage storage implementation is not specify but must support non volatility if demanded to be able to be retained after reboots variables are defined by their Vendor GUID, Name and attributes such as: their scope (boot time, run time, non-volatile), whether writes require authentication or result in appending data instead of overriding [uefi-spec] [TODO deep dive in authenticated variables] architectually defined variables are called Globally Defined Variables where vendor GUID is defined with the macro EFI_GLOBAL_VARIABLE [uefi-spec] relevant for secure boot and boot manager

2.1.8 **Systemtable**

The UEFI System Table is an important data structure, it provides access to system configuration information, boot services, runtime services and protocols.

system tables offers boot and runtime services supplied by drivers implementing architectural protocols

2.1.8.1 Boottime Services

2.1.8.2 Runtime Services

2.1.9 Boot Manager

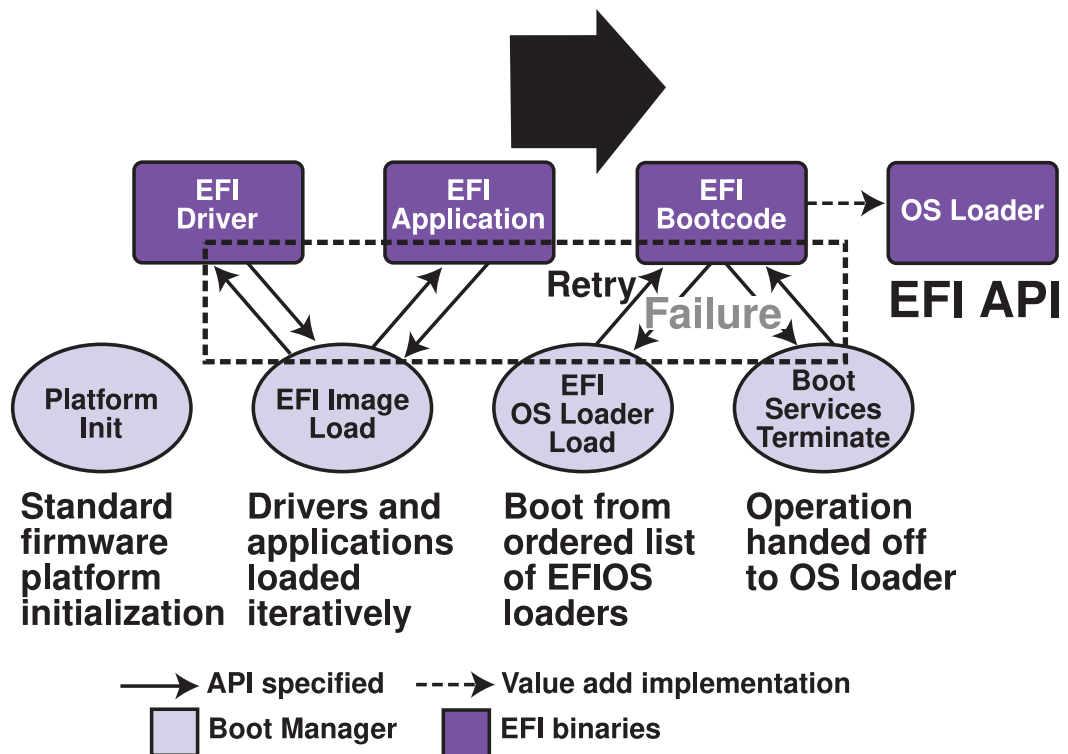
??

what is the boot manager which drivers and applications and when firmware policy engine configured by non volatile variables [uefi-spec] boot manager = bds boot behavior

2.1.9.1 Boot Variables

boot options variables boot options (network, simple file system protocol, load file)
default boot behavior for simple file system protocol

EFI boot variable must contain a short description of the boot entry, the complete device and file path of the Boot Manager, and some optional data [windows-internals-7-part2]



OM13144

Fig. 2.1: Booting Sequence [uefi-spec]

2.2 PI! (PI!)

2.2.1 Boot Sequence

focus will be on dxe and transient system load ??

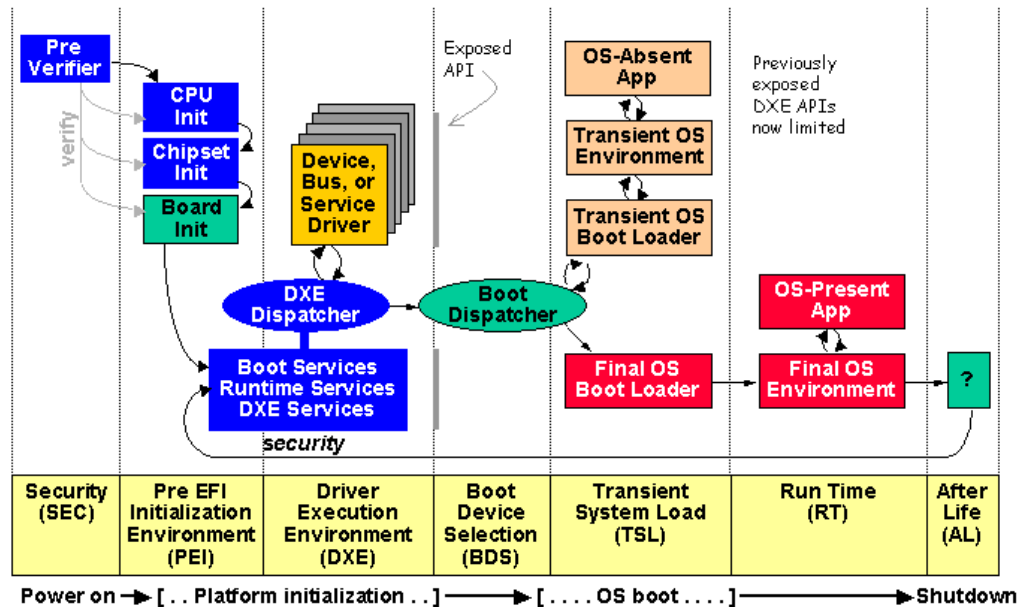


Fig. 2.2: PI! Architecture Firmware Phases [pi-spec]

1. **SEC! (SEC!)** The Security phase is the first code executed by the CPU, it is uncompressed and executed directly from flash. It consists of platform specific assembly.
 - Handles all platform restart events (power on, wakeup from sleep, etc)
 - Creates a temporary memory state by configuring the CPU Cache as RAM (CAR) "no evictions mode"
 - Serves as the root of trust in the system
 - Passes handoff information to the Pre-EFI Initialization (PEI) Foundation

Since the CPU doesn't know about UEFI or BIOS the initial step is exactly the same, it starts in 16-bit real mode and fetches it's first instruction from 'CS = 0xF000' and 'IP = 0xFFFF0' but instead of shifting 'CS' left by four bits and adding 'IP', the 'CS' base register is initialized to '0xFFFF'0000'. So the first

instruction is fetched from the physical address '0xFFFF'FFF0' ('0xFFFF'0000 + 0xFFFF0'). The CS base address remains at this initial value until the CS selector register is loaded by software (e.g. far jump or call instruction)

- Populates Reset Vector Data structure
- Saves Built-in self-test (BIST) status
- Enables protected mode (16 bit -> 32 bit)
- Configures temporary RAM (not only limited in processor cache) by using MTRR to configure CAR.

Passing of handoff information to the PEI phase:

```
typedef VOID EFIAPI (*EFI_PEI_CORE_ENTRY_POINT)(IN CONST EFI_SEC_PEI_HAND_OFF
```

SEC Core Data:

- Points to a data structure containing information about the operating environment:
- Location and size of the temporary RAM
- Location of the stack (in temporary RAM)
- Location of the Boot Firmware Volume (BFV)

PPI list:

- Temporary RAM support PPI

An optional service that moves temporary RAM contents to permanent RAM.

- SEC platform information PPI

An optional service that abstracts platform-specific information to locate the PEIM dispatch order and maximum stack capabilities.

ref to PSP

inductive security design integrity of next module checked by the previous module

handles all platform restart events applying power to system from unpowered state restarting from active state receiving exception conditions

creates temporary memory store possibly CPU **CAR!** (**CAR!**) cache behaves as linear store of memory no evictions mode every memory access is a hit eviction not supported as main memory is not set up yet and would lead to platform failure

final step Pass handoff information to the **PEI!** (**PEI!**) Foundation

- state of platform
- location and size of the **BFV!** (**BFV!**)
- location and size of the temporary RAM
- location and size of the stack
- optionally one or more **HOB!**s (**HOB!**s) via the **SEC! HOB! Data PPI!** (**PPI!**)

Part of this process is a so called **HOB!** with a function pointer to a procedure to verify PE modules.

SEC Platform Information PPI information about the health of the processor

SEC HOB Data PPI

2. **PEI!** (**PEI!**) Configures a system meeting the minimum prerequisites for the Driver Execution (DXE) phase, which is generally a linear array of RAM large enough for successful execution.

PEI provides a framework allowing vendors to supply initialization modules for each functionally distinct piece of system hardware which must be initialized before the DXE phase.

PEI design goals of the PI architecture:

- Maintenance of the "chain of trust", includes protection and authorization of PEI modules
- Provide a core PEI module
- Independent developement of intialization modules

The PEI phase consists of the PEI Foundation core and specialized plug-ins known as Pre-EFI Initialization Modules (PEIMs).

Since the PEI phase is very early in the boot process it can't assume reasonable amounts of RAM so the features are limited:

- Locating, validating and dispatching PEIMs
- Communication between PEIMs
- Providing Hand-Off Data for DXE phase
- Initializing some permanent memory complement
- Describing the memory in Hand-Off Blocks (HOBs)
- Describing the firmware volume locations in HOBs
- Passing control into the Driver Execution Environment (DXE) phase
- Discover boot mode and possibly resume from sleep state

PEI Service Table visible to all PEIMs in the system, a pointer to this table is passed as an argument via the PEIM entry point, it is also part of each PEIM-to-PEIM Interface (PPI).

PEI Foundation code is portable across all platforms of a given instruction-set. The set of exposed services is the same across different microarchitectures and allows PEIMs to be written in C.

- Dispatches PEIMs - Maintains boot mode - Initializes permanent memory - Invokes DXE loader

The PEI Dispatcher evaluates dependencies of PEIMs in the firmware volume, these dependencies are PPIs. The Dispatcher holds internal state machines to check dependencies of PEIMs, it starts executing PEIMs whose dependencies are satisfied to build up dependencies of other PEIMs, this is done until the dispatcher cannot invoke any more PEIMs. Then the DXE Initial Program Loader (IPL) PPI is invoked to pass control to the DXE phase.

PEIMs are specialized drivers that personalize the PEI Foundation to the platform. They are analogous to DXE driver and generally correspond to the components being initialized. It is strongly recommended that PEIMs do only the minimum necessary work to initialize the system to a state that meets the prerequisites of the DXE phase. PEIMs reside in firmware volumes (FVs).

PEIMs communicate with each other using a structure called PPI. A PPI is a GUID pointer pair. The GUID is used to identify a certain service and the pointer provides access to data structures and services of the PPI.

An architectural PPI is described in the PEI Core Interface Specification (CIS) and the GUID is known to the PEI Foundation. They typically provide a

common interface to the PEI Foundation to a service with platform specific implementation.

An additional PPI is important for interoperability but isn't required by the PEI Foundation, they can be classified as mandatory or optional.

- init permanent memory
- describe memory in **HOB!**s
- describe **FV!** (**FV!**) in **HOB!**s
- pass control to **DXE!** (**DXE!**)

crisis recovery (what is this?) resuming from S3 sleep state linear array of RAM **PEIM!** (**PEIM!**) provides a framework to allow vendors to supply separate initialization modules for each functionally distinct piece of system hardware that must be initialized prior to the DXE phase [**pi-spec**]

maintenance of chain of trust, protection against unauthorized updates to the PEI phase or modules authentication of the PEI Foundation and its modules provide core PEI module (PEI foundation) processor architecture independent, supports add-in modules from vendors for processors, chipsets, RAM

Locating, validating, and dispatching PEIMs Facilitating communication between PEIMs Providing handoff data to subsequent phases

3. **DXE!** (**DXE!**)

The DXE Foundation produces a set of Boot, Runtime and DXE Services and exposes them through handle databases in the EFI System Table. It is designed to be completely portable, independent of processor, chipset and platform. The only dependent of the Hand-Off Blocks from the PEI phase, after these are processed the all prior phases can be unloaded.

The DXE Dispatcher discovers DXE drivers within the Firmware Volume (FV) and executes them in the correct order, respecting their dependencies towards each other. The Firmware Volume file format allows the DXE driver images to be packaged with expressions about their dependencies. Since the DXE Drivers are PE/COFF images the dispatcher comes with an appropriate loader to load and execute the image format.

The DXE Drivers are responsible for initializing the processor, chipset, and platform components as well as providing software abstractions for console and boot devices in the form of services.

dxefoundation platform independent is implementation of UEFI UEFI
Boot Services UEFI Runtime Services DXE Services

dxefoundation discover drivers stored in firmware volumes and execute in
proper order apriori file optionally in FV or depex of driver after dispatching
all drivers in the dispatch queue hands control over to BDS

dxefoundation init processor, chipset and platform produce architectural protocols
and I/O! (I/O!) abstractions for consoles and boot devices

initializing the processor, chipset, and platform components providing software
abstractions for system services, console devices, and boot devices.

4. **BDS! (BDS!)** The DXE Foundation will hand control to the BDS Architectural Protocol after all of the DXE drivers whose dependencies have been satisfied have been loaded and executed by the DXE Dispatcher.

During the BDS phase new Firmware Volumes (FV) might be discovered and control is once again handed to the DXE Dispatcher to load drivers found on these additional volumes.

DXE architectural protocol one function entry platform boot

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 [uefi-spec]

[pi-spec]

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

5. **TSL! (TSL!)**

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()'.

boottime and runtime services/driver bootloader [**uefi-spec**] [**uefi-spec**]

ExitBootServices()

6. **RT! (RT!)** Boot service drivers have been unloaded and only runtime services are accessible.

runtime services/driver

7. **AL! (AL!)** 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.

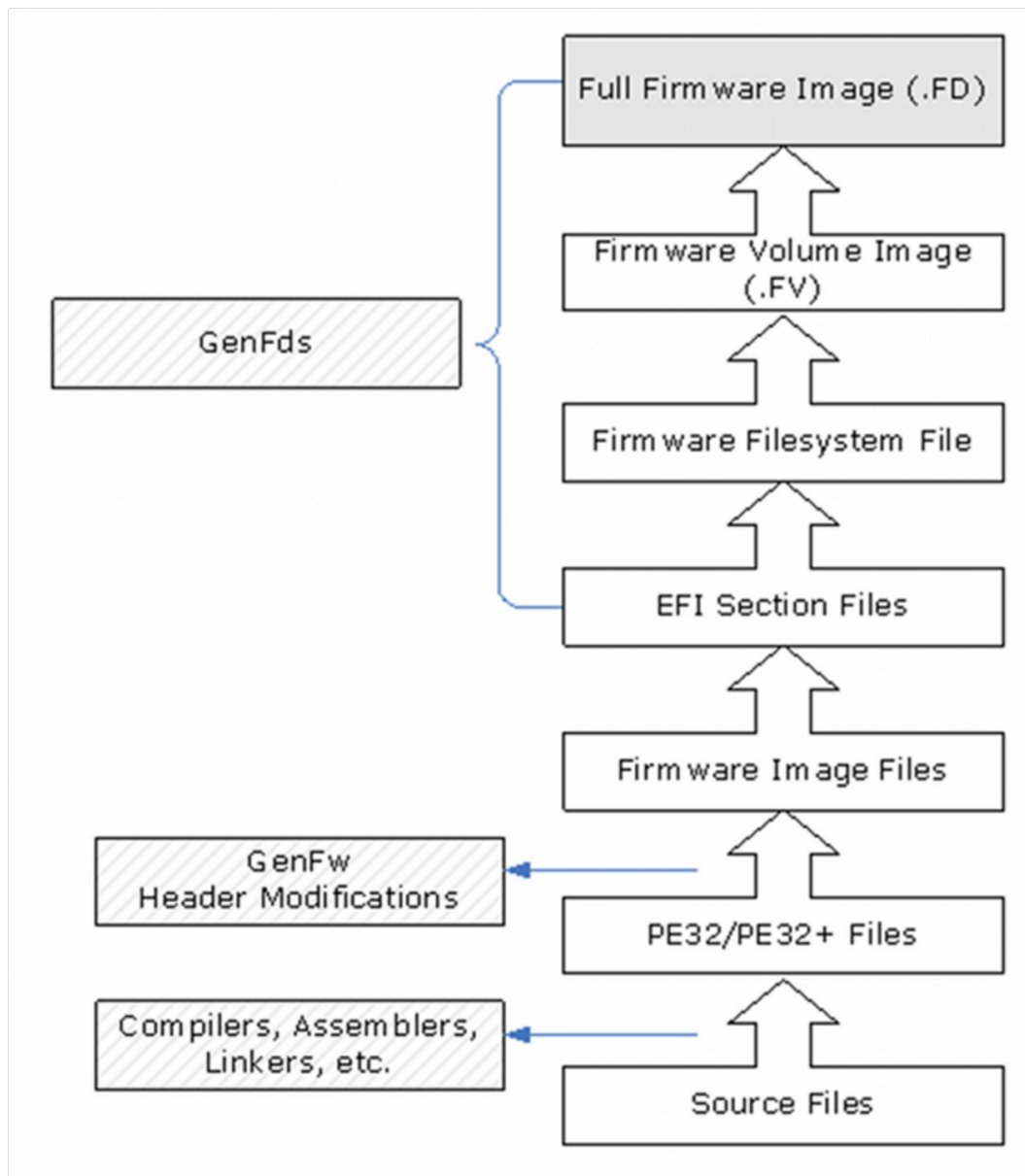
hibernation sleep

2.2.2 **UEFI!/PI!** Firmware Images

Firmware Images are stored in Flash Devices (FD), a Firmware Volume (FV) serves as file level interface. Usually multiple FVs are present in a single FD but a single FV can also be distributed via multiple FDs. A FV is formatted with a binary file system, typically with Firmware File System (FFS).

In a FFS modules are stored as files, they can be executed at the fixed address from Read Only Memory (ROM) or through relocation in loaded memory. Within a file are multiple sections which then contain the "leaf" images. These are for example PE32 images.

[**pi-spec**]



FD! (FD!) persistent physical device contains firmware code and/or data typically flash may be divided into smaller pieces to form multiple logical firmware devices multiple physical firmware devices may be aggregated into one larger logical firmware device

FV! (FV!) logical device organized into a file system attributes such as - size - formatting - read/write access

FFS! (FFS!) organization of files and free space no directory hierarchy all files flat in root dir parsing requires walking from beginning to end

firmware files types

some file types are sub-divided in file sections

file sections can be either encapsulation or leaf sections such as PE32 RAW
VERSION TE

dx drivers files contain one PE32 executable section may contain version section
may contain dx depex section

freeform files can contain any combination of sections

PEI phase Service Table FfsFindNextFile, FfsFindFileByName and FfsGetFileInfo

DXE phase

depex

[tianocore-edk2-build-spec]

2.2.3 Security

others not discussed further user identification

PEI GuidedSection Extraction

2.2.3.1 Secure Boot

[tianocore-understanding-uefi-secure-boot-chain]

driver signing executables may be located on un-secured media system provider can
authenticate either origin or integrity

digital signature data to sign public/private key pair used to verify integrity

embedded within PE file calculating the pe image hash - hashing the pe header,
omitting the file's checksum and the Certificate Table entry in Optional Header Data
Directories - sorting and hashing pe sections omitting attribute certificate table and
hash remaining data

[microsoft-pe-signature-format]

guarantees only valid 3rd party firmware code can run in OEM firmware environment
UEFI Secure Boot assumes the system firmware is a trusted entity any 3rd party
firmware code is not trusted including bootloader/osloader, PCI option ROMs, UEFI
shell tool

two parts verification of the boot image and verification of updates to the image security database [[tianocore-understanding-uefi-secure-boot-chain](#)]

Secure Boot uses the content of the SPI flash memory as its root of trust[[lojax](#)]

2.2.3.2 Firmware Protection

DXE SMM Ready to Lock Vol4

Capsule Architectural Protocol

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

flash device security

2.2.3.3 TPM measurements

A **TPM!** (**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 [[tcg-tpm-summary](#)] [[tcg-tpm-library-part1-architecture](#)]

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, [[tianocore-trusted-boot-chain](#)]

PCR! Index	PCR! Usage
0	SRTM, BIOS, Host Platform Extensions, Embedded Option ROMs and
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 Atte
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code)
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8	First NTFS! (NTFS!) boot sector (volume boot record)
9	Remaining NTFS! boot sectors (volume boot record)
10	Boot Manager
11	BitLocker Access Control

[tcg-pc-client-platform-firmware-profile-spec; winwos-internals-6-part2]

[tianocore-trusted-boot-chain]

TCG!2 Protocol Trusted Computing Group 2 (TCG2) Protocol [tcg-efi-protocol-spec]

2.3 EDK! II

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

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 [tianocore-edk2]

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 **ESP!** (**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 **BCD!** (**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 consisting 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 [**windows-internals-7-part2**]. 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, `Winload.efi`, the application responsible for loading the kernel into memory [**windows-internals-7-part2**].

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 [**windows-internals-7-part2**]. 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 [**windows-internals-7-part2**]. 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 [**windows-internals-7-part2**]. [TODO mention ntoskernel.exe]

[TODO TPM interaction] [**windows-internals-7-part2**]

3.1.3 Runtime

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

3.1.4 Secure Boot

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. [**windows-internals-7-part1**] 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 [**windows-internals-7-part2**]. At the top level it has 9 different keys [**windows-internals-7-part2**]. 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 traverse and modify the Registry [**windows-internals-7-part2**]. 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 [**windows-internals-7-part2**]. Some hives even reside entirely in memory as a means of offering hardware configuration through the Registry **API!** (**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 [**windows-internals-6-part2**]

3.3 Trusted Boot

3.3.1 KMCI

3.3.2 HVCI

3.4 BDE! (BDE!)

Windows is only able to enforce security policies when it is active, leaving the system vulnerable when accessed from outside of the **OS!** [**windows-internals-6-part2**]. Windows uses BitLocker, integrated **FVE!** (**FVE!**), aimed to protect system files and data from unauthorized access while at rest [**microsoft-bitlocker-overview**], while also verifying boot integrity when used with a **TPM!** [**windows-internals-6-part2**]. The en- and decryption of the volume is done by a filter driver beneath the **NTFS!** driver as shown in ???. 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 [**windows-internals-6-part2**]. The encryption of each block is done using a modified version of the **AES!** (**AES!**) 128 and **AES!**256 cypher [**windows-internals-6-part2**]. A **FVEK!** (**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 [**windows-internals-6-part2**]. The **FVEK!** is encrypted with a **VMK!** (**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 [**windows-internals-6-part2**]. 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 [**bde-format-spec**]

TPM - tpm only no additional user interaction - tpm with startup key additional usb - tpm with PIN - tpm with startup key and PIN [**microsoft-bitlocker-countermeasures**] with tpm ensures integrity of early boot components and boot configuration tpm usage requires TCG!2 compliant UEFI! firmware [**windows-internals-6-part2**]

tpm is used to *seal* and *unseal* VMK! [**TODO PCR table either here or at TPM section**] platform validation profile defaults are PCR!s (PCR!s) {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 [**bde-format-spec**] when enabling manually, save on non encrypted medium [**microsoft-bitlocker-basic-deployment**]

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 [**microsoft-bitlocker-device-encryption**]

User key password with max 49 characters [**bde-format-spec**]

Clear key unprotected 256-bit key stored on the volume to decrypt vmk [**bde-format-spec**] 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 [**microsoft-bitlocker-device-encryption**]

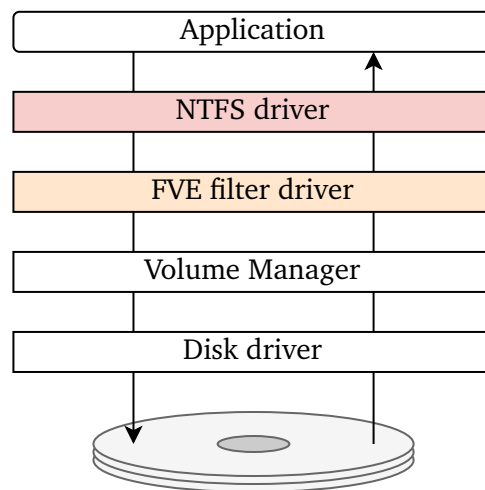


Fig. 3.1: BitLocker Volume Access Driver Stack (inspired by [windows-internals-6-part2])

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.

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. [**finspy**] backs up and replaces the Windows Boot Manager `bootmgfw.efi` on the **ESP!**, while the path in the boot entry for Windows still contains the same file path. This forces the **UEFI!** boot manager to launch the bootkit when it decided to use the Windows boot entry. [**specter**] patches the entrypoint of `bootmgfw.efi` and its copy `bootx64.efi` in the default boot path, so that it executes malicious code upon launch. [**dreamboot**] and [**efiguard**] 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 [**efiguard**], 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, [vector-edk] requires booting the target machine from a USB key [mosaicregressor] [TODO SPI read/write] [lojax] dump remove previous NTFS driver add DXE drivers reflash image

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

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 [vector-edk; mosaicregressor; lojax], memory-based attacks were performed by both root- and bootkit [dreamboot; efiguard; especter; finspy ; moonbounce; cosmicstrand]. There is no technical limitation as we show in ?? 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 primary partition is **NTFS** formatted and due to the **UEFI** specification only mandating compliant firmware to support **FAT12**, **FAT16** and **FAT32** [**uefi-spec**], **NTFS** drivers are delivered as part of the attack. [**mosaicregressor**] and [**lojax**] seem to use [**vector-edk**]'s leaked **NTFS** driver. [**lojax**] 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. [**mosaicregressor**] 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 [**especter**] 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. [**finspy**; **dreamboot**] when executed load `bootmgfw.efi` into memory and apply patches before launching it. [**efiguard**]'s core functionality is the same for its root- and bootkit variant. A **DXE** driver is loaded, either from 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 [**efiguard**]. [**moonbounce**] 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** (**Winload.efi**!) to `ntoskernel.exe`, the kernel image.

Some attacks patch the kernel to disable Windows Driver signing and then install a kernel driver [**efiguard**; **especter**]. Others deploy payload with elevated privileges [**finspy**; **dreamboot**] or map code directly into kernel space [**moonbounce**; **cosmicstrand**].

[TODO memroy based vs storage is it really clear cut, some use combination]
[TODO not THAT importan but would be really cool, as it stands out as really exploiting rootkit capabilities]

Test Setup

We perform our attacks against Windows 11 on three different setups, as even though all three **UEFI** firmwares used, are **[pi-spec]** compliant, there still are many things left up to the **OEM**!s (**OEM**!s) to decide, when implementing a firmware image.

5.1 **QEMU**!

Our main development setup is an emulated environment using the emulator **QEMU**! (**QEMU**!)[**qemu**] together with the **OVMF**! (**OVMF**!) image, from **EDK**! (**EDK**!)II (edk2—stable202208). For Secure Boot we generate our own **PK**! (**PK**!) and use the *Microsoft Corporation KEK! CA! 2011* as **KEK**! (**KEK**!) and the two signature **DB**!s (**DB**!s) *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 **[microsoft-secure-boot-guidance]** and the latter reserved for third party executables signed at Microsoft's discretion after manual review **[TODO better source]****[microsoft-uefi-signing]**. In the attacks against BitLocker we use *swtpm* for the emulation of a software **TPM**! [**swtpm**]. 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]

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 **USB!** (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 to 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!**. [TODO dump a windows boot entry] Now that our bootkit is in place of the Windows Boot Manager, when the **UEFI!** Boot Manager

selects the boot load option `Boot####` for the Windows Boot Manager, the file path `EFI\Microsoft\Boot\bootmgfw.efi` will cause our bootkit to be executed.

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 [**ntfs-3g**], that was ported to the **UEFI** environment by *pbatard* [**ntfs-3g-uefi**].

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

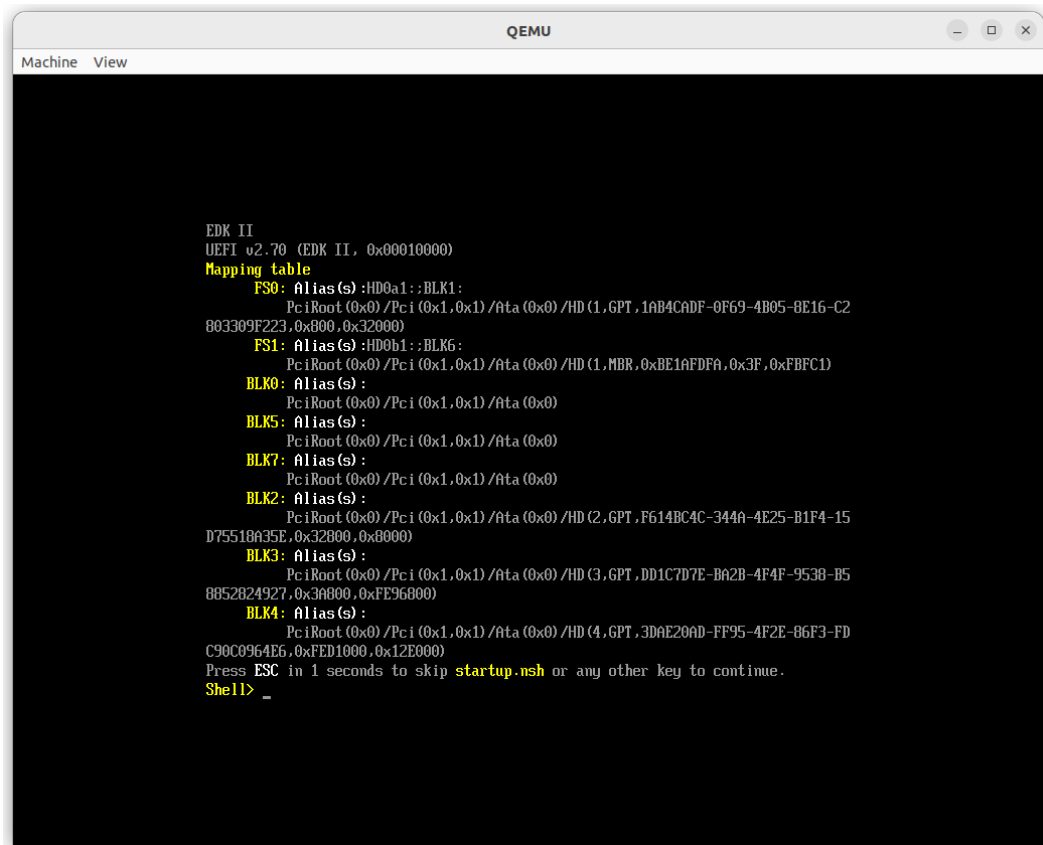
[TODO better summary of UEFI shell] Part of the family of **UEFI** specifications is a shell specification which defines a feature rich **UEFI** shell application to interact with the **UEFI** environment [**uefi-shell-spec**]. It offers commands related to boot and general configuration, device and driver management, file system access, networking [**uefi-shell-spec**] and supports scripting [**uefi-shell-spec**]. We can use the file system related commands to test the **NTFS** driver. ?? 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` [**uefi-shell-spec**]. `ver` displays the version of the **UEFI** specification the firmware conforms to [**uefi-shell-spec**].

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 [**uefi-shell-spec**]. 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 [**uefi-shell-spec**].

[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



```
Machine View

EDK II
UEFI v2.70 (EDK II, 0x00010000)
Mapping table
FS0: Alias(s):HD0a1::BLK1:
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(1,GPT,1AB4CADF-0F69-4B05-BE16-C2
803309F223,0x800,0x32000)
FS1: Alias(s):HD0b1::BLK6:
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(1,MBR,0xBE1AFDFA,0x3F,0xFBFC1)
BLK0: Alias(s):
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
BLK5: Alias(s):
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
BLK7: Alias(s):
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)
BLK2: Alias(s):
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(2,GPT,F614BC4C-344A-4E25-B1F4-15
D7518A35E,0x32800,0x8000)
BLK3: Alias(s):
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(3,GPT,DD1C7D7E-BA2B-4F4F-9538-B5
8B52B24927,0x3A800,0xFE96800)
BLK4: Alias(s):
PciRoot(0x0)/Pci(0x1,0x1)/Ata(0x0)/HD(4,GPT,3DAE20AD-FF95-4F2E-86F3-FD
C90C0964E6,0xFED1000,0x12E000)
Press ESC in 1 seconds to skip startup.nsh or any other key to continue.
Shell> _
```

Fig. 6.1: UEFI! command prompt

File System Protocol [**uefi-spec**] provides, together with the File Protocol, file-type access to the device it is installed on [**uefi-spec**]. 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 `drivers` command, 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 it 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 driver 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. The **GUID!** can then be used to retrieve our specific protocol instance in the root- and bootkit. 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 [uefi-spec]. This would be a likely scenario as filesystem drivers are encouraged to get exclusive control over their block device [uefi-spec].

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 [uefi-spec]. 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 [**uefi-spec**]. 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**]

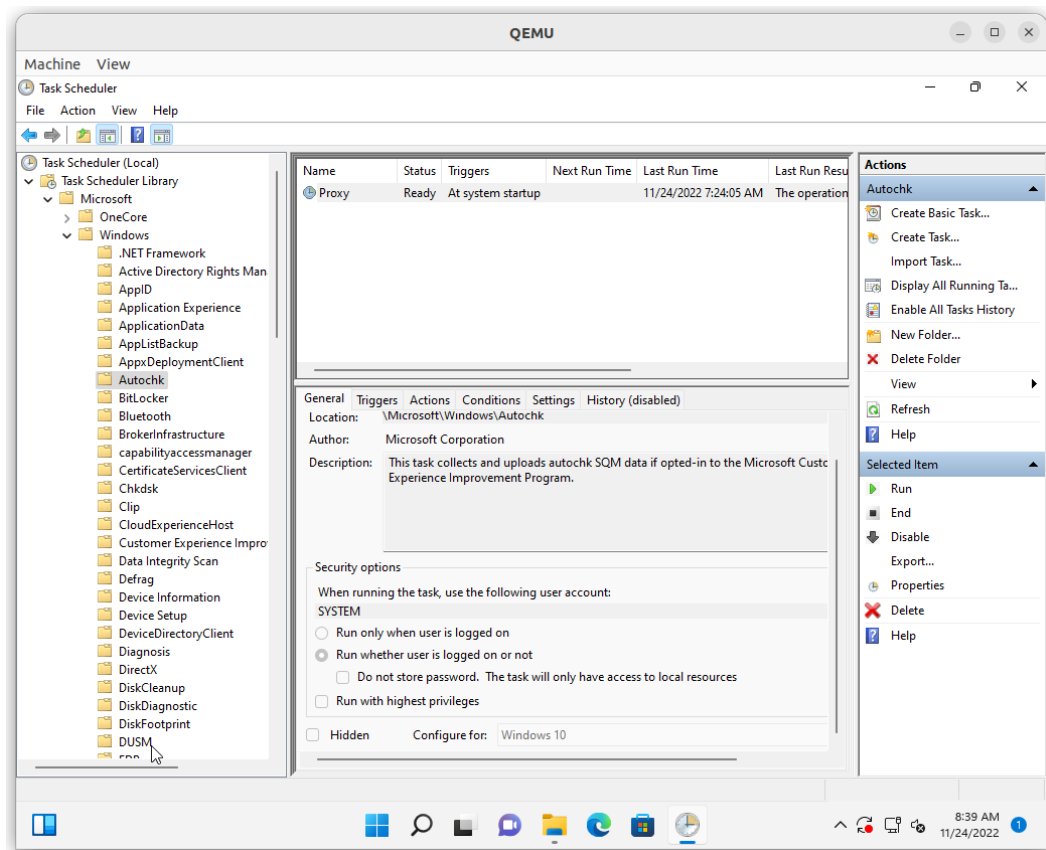
The Task Scheduler is a Windows service responsible for managing the automatic execution of background tasks [**windows-internals-7-part2**]. Tasks are performed on certain triggers, which may be time-based (periodically or on a specific time) or event-based, for example on user logon or system boot [**microsoft-task-scheduler-triggers**]. A task can perform various actions upon invocation [**microsoft-task-scheduler-actions**], 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 [**microsoft-task-scheduler-security-contexts**]. 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 **DLL! (DLL!)** acproxy.dll and invokes the exported function PerformAutochkOperations [**microsoft-rundll32**]. 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 [**microsoft-autochk**]. 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 NT\CurrentVersion\Schedule\TaskCache\Tasks [**windows-intern**]. There also exist a secondary copy of the task descriptors, on the regular file system under %windir%\system32\Task, stored as **XML! (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 [**microsoft-whoami**]. 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 [**microsoft-localsystem-account**].

[**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 [**chntpw**]. 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 overwrites 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 abstractions. 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 Manager loads boot options, this includes relaying the `LoadOptions` and `ParentHandle` of the *EFI! Loaded Image Protocol* [**uefi-spec**] 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 [**pi-spec**], 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 **GUID!**s.

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. ?? 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. When we have the image we can edit it with UEFITool, which is an editor for firmware images conforming to the **UEFI! PI!** specification **[uefityl]**. 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!** (**DEPEX!**) section **[pi-spec]**. 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 **FDF!** (**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 **[pi-spec]**. 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 ??, 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. ?? shows, that the image origin dictates which is applied. Standard policy for images from a **FV!** (**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 [...]” [**uefi-spec**]. This behavior was reproducible on all our hardware setups, likely in order to prevent accidentally entirely unbootable firmware or to reduce boot time.

```

1  switch (GetImageType(File))
2  {
3      case IMAGE_FROM_FV:
4          Policy = ALWAYS_EXECUTE;
5          break;
6
7      case IMAGE_FROM_OPTION_ROM:
8          Policy = PcdGet32(PcdOptionRomImageVerificationPolicy);
9          break;
10
11     case IMAGE_FROM_REMOVABLE_MEDIA:
12         Policy = PcdGet32(PcdRemovableMediaImageVerificationPolicy);
13         break;
14
15     case IMAGE_FROM_FIXED_MEDIA:
16         Policy = PcdGet32(PcdFixedMediaImageVerificationPolicy);
17         break;
18
19     default:

```

```

20     Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
21     break;
22 }

```

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 ??, as well as additionally dictating the BitLocker default validation profile Windows uses as mentioned in ??. We perform our attack against both default profiles, starting with {0, 2, 4, 11}. This means either Secure Boot is disabled or **PCR!7** 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 ??. 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.

6.3.1 Infection

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. 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 ??, 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 [**windows-internals-7-part2**].



Fig. 6.2: 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 ??, 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 **PCR!**s). 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 ???. 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 ???;

```
1 #include <Uefi.h>
2
3 #include <Protocol/SimpleTextInEx.h>
4
5 EFI_BOOT_SERVICES *gBS;
6
7 EFI_STATUS
```



```

8  EFIAPI
9  EntryPoint(
10     IN EFI_HANDLE ImageHandle,
11     IN EFI_SYSTEM_TABLE *SystemTable)
12  {
13     gBS = SystemTable->BootServices;
14
15     EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *SimpleTextInEx;
16
17     gBS->HandleProtocol(SystemTable->ConsoleInHandle,
18                        &gEfiSimpleTextInputExProtocolGuid,
19                        (VOID **)&SimpleTextInEx);
20
21     UINTN EventIndex;
22     gBS->WaitForEvent(1, &SimpleTextInEx->WaitForKeyEx, &EventIndex);
23
24     EFI_KEY_DATA KeyData;
25     SimpleTextInEx->ReadKeyStrokeEx(SimpleTextInEx, &KeyData);
26
27     // do something with key press
28
29     return EFI_SUCCESS;
30 }

```

Listing 6.2: Simple Text Input Ex Protocol to use ReadKeyStrokeEx to read a pending key press]Example of using HandleProtocol to retrieve an instance to the Simple Text Input Ex Protocol to use ReadKeyStrokeEx to 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 [uefi-spec]. OpenProtocol offers many additional options such as keeping track of the protocol users [uefi-spec] and exclusivity. ?? shows how HandleProtocol can be used to receive the Simple Text Input Ex Protocol instance installed on the active console input device [uefi-spec]. 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 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 [tianocore-edk2-driver-writer-s-guide]. This keeps the protocol interface limited to the public functionality. In ?? we show an example of retrieving private data through the public protocol interface.

```

1  #include <Uefi.h>
2
3  #include <Protocol/DiskIo.h>
4  #include <Protocol/BlockIo.h>
5
6  typedef struct
7  {
8      UINTN Signature;
9      EFI_DISK_IO_PROTOCOL DiskIo;
10     EFI_BLOCK_IO_PROTOCOL *BlockIo;
11 } DISK_IO_PRIVATE_DATA;
12
13 EFI_STATUS
14 EFI_API
15 DiskIoReadDisk(
16     IN EFI_DISK_IO_PROTOCOL *This,
17     IN UINT32 MediaId,
18     IN UINT64 Offset,
19     IN UINTN BufferSize,
20     OUT VOID *Buffer)
21 {
22     DISK_IO_PRIVATE_DATA *Private;
23
24     Private = DISK_IO_PRIVATE_DATA_FROM_THIS(This);
25
26     Private->BlockIo->ReadBlocks(...);

```

Listing 6.3: Example of using private data in the implementation of driver

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*.

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 [**windows-internals-6-part2**]. 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 ?? 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 **FUSE!** (**FUSE!**) interface to offer mounting of BitLocker encrypted partitions under Linux supporting read and write access [**dislocker**].

In ?? 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 **GUID**!s are consumed and produced by the driver. Ignoring ones that are not further used in code have the following list:

- gEfiDevicePathToTextProtocolGuid
- gEfiDiskIoProtocolGuid
- gEfiDiskIo2ProtocolGuid
- gEfiBlockIoProtocolGuid
- gEfiBlockIo2ProtocolGuid
- gEfiSimpleFileSystemProtocolGuid

The *Device Path to Text Protocol* is just a self-explanatory utility protocol and not related to media access, so we can ignore it as well. The Simple File System Protocol is only produced, as in installed onto handles it supports and can offer the protocol on. 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. We will ignore the asynchronous protocols, as they only serve to further abstract their synchronous version [uefi-spec]. The same could be said for the *Disk I/O! Protocol*, as it abstracts the *Block I/O! Protocol* to offer an offset-length driven continuous access to the underlying block device [uefi-spec], 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, in 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 ??, 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 ?. We can then use Dislocker on read and write operations to implement our own filter driver as shown in ?.

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 parameters, comparable to the *Disk I/O! Protocol* abstraction. `dislock` reads and decrypts, while

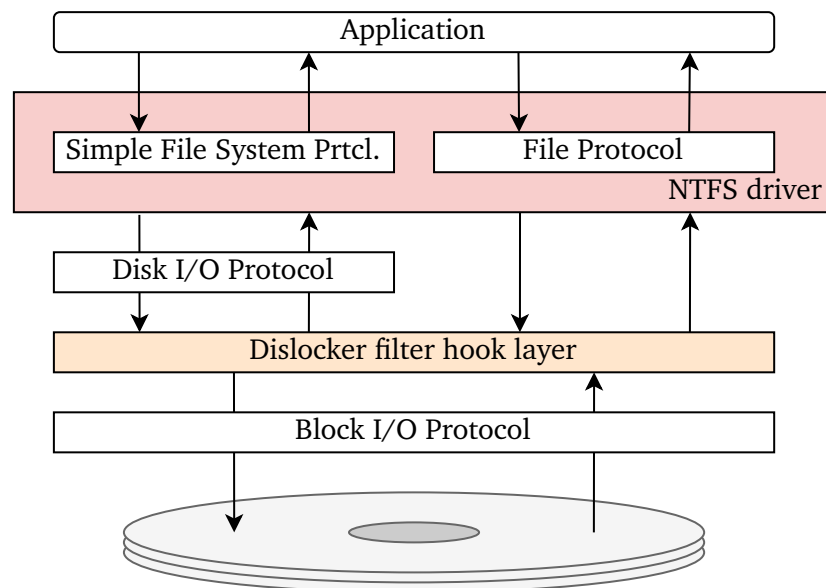


Fig. 6.3: Dislocker Volume Access Protocol Stack

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 to the next block boundary. We can map `pread` and `pwrite` to call the original `ReadBlocks` and `WriteBlocks` functions. Since the two Dislocker functions expect offset-length, we simply multiply the starting block index by the block size to use as starting offset.

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**] [**exitbootservices-hooking**] 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` [**microsoft-bitlocker-manage-bde**] 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 **TCG!2** Protocol [**tcg-efi-protocol-spec**] **TPM!** communication receive bitlocker **VMK!** key and send to dislocker [**bde-format-spec**] [**tpm-sniffing**]

This increases the persistence and applicability of our attack immensely.

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 restrictions are applied **SPI!** (**SPI!**) flash access. 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 were able to retrieve the recovery key to use it with our **UEFI!** port of Dislocker to mount the encrypted drive, allowing us to perform the same attack as on unencrypted drives. When our payload is part of the **PCR!** values used to encrypt the **VMK!** or when a misconfigured validation profile is used, 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 Secure Boot, that Microsoft labels as the preferred configuration, forcing a default validation profile of 7, 11 left the system vulnerable to **TPM!** sniffing through our rootkit.

Discussion

we achieved a boot and rootkit with unrestricted disk access which results in elevated execution on the target OS persistence with rootkit/none with bootkit bootkit delivery: usb stick, from windows rootkit delivery: spi clamp, firmware delivery process, maybe windows with exploit

bootkit vs rootkit bootkit: installation is much easier: windows installer physical presence with bootable usb stick defeated by secure boot in case of physical presence it may require to change boot order bios password mitigates that if no password present we can disable secure boot not entirely persistent fresh reinstallation with partition removal and general hard drive replacements defeat it

rootkit: barrier of entry is higher physical access is more difficult than just booting from a usb stick exploit to overwrite spi flash or be delivered with supply chain difficult but high payoff persistence across reinstallations or hard drive replacements can prevent further bios updates and be unremovable secure boot does not include internal DXE drivers option ROM rootkit is defeated by secure boot spi reflash may disable secure boot by changing variable anyways SMM rootkit very powerful, complete control over the system

we didnt try to be undetectable windows is very vulnerable with unrestricted disk access we achieved highly privileged execution which the other the methods of the other two storage based rootkits didn't secure boot is very limited secure boot can easily be disabled without bios password TPM does its job in detecting PCR change bitlocker recovery prompt can raise suspicion esc for more recovery options disable esc presses very effective if part of the delivery process or in general present before os installation BitLogger somewhat last resort social engineering aspect windows secure boot PCR7 binding and use of secure boot system integrity check and validation profile 7, 11 allows stolen laptops to be unlocked by simply booting with the rootkit

you can change recovery message and URL in BCD hive

not yet done: prevent firmware update

boottime vs runtime rootkit

8.1 Rootkit classification

statistiken zu bitlocker und secureboot auf systemen

industrie standard zur system security in firmen

8.2 Mitigations

bios password against secure boot removal

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

inaccessible spi flash

tpm + pin/usb detectability

8.2.1 User awareness

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 ??

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 einschaenkbar?

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

Conclusion

dxr runtime rootkit not really feasible since it doesn't run without being called back by the OS dxr smm rootkit makes sense

9.1 Achieved Goals

when we are already in the image we can gain full control over the system system can't 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 critical information into an inherently untrustable system on the other hand one could force such a prompt themselves mere existence of a recovery key is a security flaw

9.2 Future Work

tpm and pin capsule update exploit in tpm measurement chain that results in not being measured can exploit the tga hook directly to retrieve the vmk memory based rootkit hypervisor kernel security

List of Figures

List of Tables

List of Listings

Appendix

A

A.1 Protocols

```
1 typedef
2 EFI_STATUS
3 (EFIAPI *EFI_INPUT_READ_KEY_EX)(
4     IN EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *This,
5     OUT EFI_KEY_DATA *KeyData
6 );
7 struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL {
8     EFI_INPUT_RESET_EX Reset;
9     EFI_INPUT_READ_KEY_EX ReadKeyStrokeEx;
10    EFI_EVENT WaitForKeyEx;
11    EFI_SET_STATE SetState;
12    EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;
13    EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;
14 };
```

Listing A.1: Simple Text Input Ex Protocol

```
1 typedef
2 EFI_STATUS
3 (EFIAPI *EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME)(
4     IN EFI_SIMPLE_FILE_SYSTEM_PROTOCOL *This,
5     OUT EFI_FILE_PROTOCOL **Root
6 );
7 struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL {
8     UINT64 Revision;
9     EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
10 };
```

Listing A.2: Simple File System Protocol

```
1 struct _EFI_FILE_PROTOCOL {
2     UINT64 Revision;
3     EFI_FILE_OPEN Open;
4     EFI_FILE_CLOSE Close;
5     EFI_FILE_DELETE Delete;
6     EFI_FILE_READ Read;
7     EFI_FILE_WRITE Write;
8     EFI_FILE_GET_POSITION GetPosition;
9     EFI_FILE_SET_POSITION SetPosition;
10    EFI_FILE_GET_INFO GetInfo;
11    EFI_FILE_SET_INFO SetInfo;
12    EFI_FILE_FLUSH Flush;
13    EFI_FILE_OPEN_EX OpenEx;
14    EFI_FILE_READ_EX ReadEx;
15    EFI_FILE_WRITE_EX WriteEx;
16    EFI_FILE_FLUSH_EX FlushEx;
17 };
```

Listing A.3: File Protocol

```
1  typedef
2  EFI_STATUS
3  (EFIAPI *EFI_DISK_READ)(
4      IN EFI_DISK_IO_PROTOCOL *This,
5      IN UINT32 MediaId,
6      IN UINT64 Offset,
7      IN UINTN BufferSize,
8      OUT VOID *Buffer
9  );
10 typedef
11 EFI_STATUS
12 (EFIAPI *EFI_DISK_WRITE)(
13     IN EFI_DISK_IO_PROTOCOL *This,
14     IN UINT32 MediaId,
15     IN UINT64 Offset,
16     IN UINTN BufferSize,
17     IN VOID *Buffer
18 );
19 struct _EFI_DISK_IO_PROTOCOL {
20     UINT64 Revision;
21     EFI_DISK_READ ReadDisk;
22     EFI_DISK_WRITE WriteDisk;
23 };
```

Listing A.4: Disk I/O! Protocol

```
1  typedef
2  EFI_STATUS
3  (EFI_API *EFI_BLOCK_READ)(
4      IN EFI_BLOCK_IO_PROTOCOL *This,
5      IN UINT32 MediaId,
6      IN EFI_LBA Lba,
7      IN UINTN BufferSize,
8      OUT VOID *Buffer
9  );
10 typedef
11 EFI_STATUS
12 (EFI_API *EFI_BLOCK_WRITE)(
13     IN EFI_BLOCK_IO_PROTOCOL *This,
14     IN UINT32 MediaId,
15     IN EFI_LBA Lba,
16     IN UINTN BufferSize,
17     IN VOID *Buffer
18 );
19 struct _EFI_BLOCK_IO_PROTOCOL {
20     UINT64 Revision;
21     EFI_BLOCK_IO_MEDIA *Media;
22
23     EFI_BLOCK_RESET Reset;
24     EFI_BLOCK_READ ReadBlocks;
25     EFI_BLOCK_WRITE WriteBlocks;
26     EFI_BLOCK_FLUSH FlushBlocks;
27 };
```

Listing A.5: Block I/O! Protocol

Acronyms

ASCII American Standard Code for Information Interchange

AL Afterlife

AES Advanced Encryption Standard

ACPI Advanced Configuration and Power Interface

API Application Programming Interface

BCD Boot Configuration Data

BDE BitLocker Drive Encryption

BDS Boot Device Selection

BF Boot Firmware

BFV **BF!** Volume

BIOS Basic **I/O!** System

RAM Random Access Memory

CA Certificate Authority

CAR Cache as **RAM!**

CSM Compatibility Support Module

DEPEX Dependency Expression

DXE Driver Execution Environment

DLL Dynamically Linked Library

EDK **EFI!** Development Kit

EFI Extensible Firmware Interface

ESP **EFI!** System Partition

FAT File Allocation Table

FD Flash Device

FDF Flash Description File

FFS Firmware Filesystem

FUSE Filesystem in USerspace

FV Firmware Volume

FVE Full Volume Encryption

FVEK FVE! Key

GUID Globally Unique Identifier

GPT GUID! Partition Table

LBA Logical Block Address

HOB Hand-off Block

I/O Input/Output

NTFS New Technology File System

MBR Master Boot Record

OEM Original Equipment Manufacturer

OS Operating System

OVMF Open Virtual Machine Firmware

PCR Platform Configuration Register

PE32 Portable Executable 32-Bit

PEI Pre-EFI! Initialization

PEIM PEI! Module

PF Platform Firmware

PI Platform Initialization

PPI PEIM!-to-PEIM! Interface

QEMU Quick Emulator

RT Runtime

SEC Security

TCG Trusted Computing Group

TSL Transient System Load

TPM Trusted Platform Module

UEFI Unified **EFI**

USB Universal Serial Bus

VMK Volume Master Key

XML Extensible Markup Language

