A Practical Analysis of UEFI Threats Against Windows 11

Joshua Machauer

December 25, 2022
Version: Draft 1.0

Technische Universität Berlin

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

Bachelor's Thesis

A Practical Analysis of UEFI Threats Against Windows 11

Joshua Machauer

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

December 25, 2022

Joshua Machauer

A Practical Analysis of UEFI Threats Against Windows 11

Bachelor's Thesis, December 25, 2022
Reviewers: Prof. Dr. Jean-Pierre Seifert and Prof. Dr. Stefan Schmid
Supervisors: Hans Niklas Jacob and Christian Werling

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

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 25. Dezember 2022

Joshua Machauer

Abstract

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

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

vi

Contents

1 Introduction

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

what does it different than bios this helps write platform independent code uefi threats: A rootkit is a collection of software designed to grant a threat actor control over a system, typically with malicious intend. Rootkits set up a backdoor exploit and may deliver additional malware while leveraging their privileges to remain hidden. There are different types of rootkits such as User Mode, Kernel Mode, Bootkits (bootloader rootkits), Hypervisor and Firmware rootkits. [**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 environment, defined by the UEFI and PI specifications, listing the interface and

its implementation. This allows us to go over Windows 11's UEFI installation and boot process as well as relevant security mechanisms in Chapter 3. With this knowledge we then look at analyses of previously discovered UEFI threats in Chapter 4, categorizing them by their attack vector and threat model. In Chapter 5 we discuss the test setups, we performed our attacks on, consisting of emulation and hardware. We then lay out our practical approach of implementing our own UEFI attacks in Chapter 6, analyzing security mechanism faced when attempting attacks from the UEFI environment. Afterwards we discuss the impact of our findings, the restrictions that apply, as well as potential mitigation techniques in Chapter 7. Chapter 8 concludes the thesis by summarizing the achievements of our attacks and lays out potential future topics.

2 UEFI/PI

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

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

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 framework into UEFI and pi?]**

2.1 Unified Extensible Firmware Interface (UEFI)

[TODO MEMORY LAYOUT no memory protection, RWE everywhere]

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

boot- and runttime 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 Globally Unique Identifier (GUID)

GUID

2.1.2 GUID Partition Table (GPT)

Master Boot Record (MBR) boot code, four partitions GUID for uniquely identifying each partition GUID for partition type content GPT Logical Block Address (LBA) legacy MBR or protective MBR

[**uefi-spec**]

2.1.3 EFI System Partition (ESP)

File Allocation Table (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 Portable Executable 32-Bit (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 containting 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 pre-determined 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 presistent 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 runttime 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 arichtectural protocols

2.1.8.1. Boottime Services

2.1.8.2. Runtime Services

2.1.9 Boot Manager

Figure 2.1

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**]

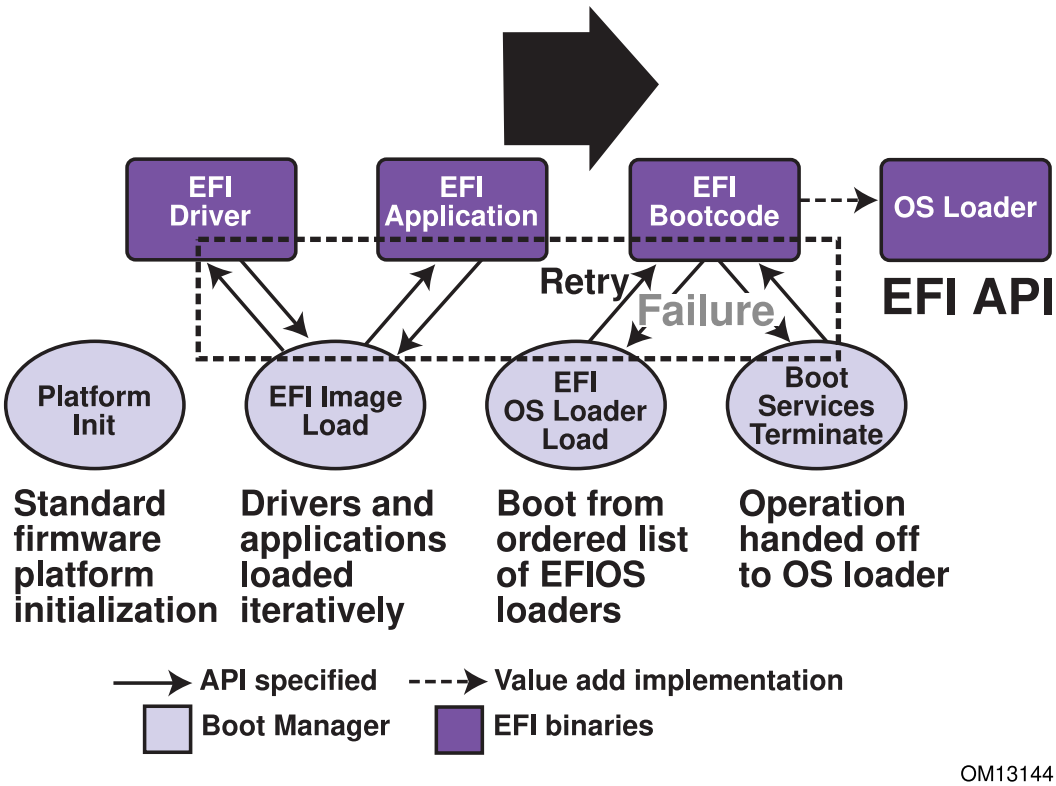


Fig. 2.1.: Booting Sequence [**uefi-spec**]

2.2 Platform Initialization (PI)

2.2.1 Boot Sequence

focus will be on dxe and transient system load Figure 2.2

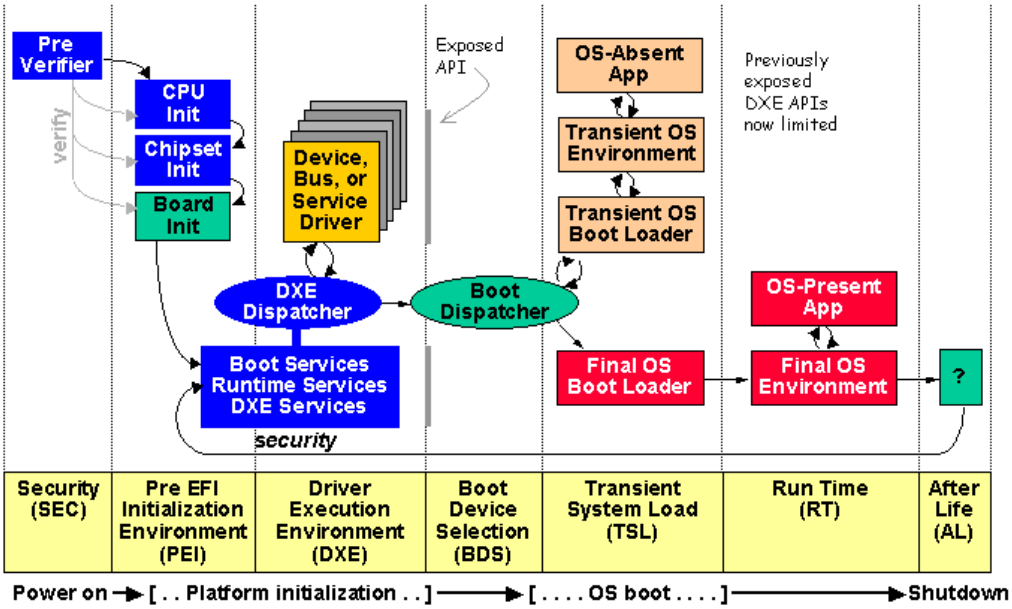


Fig. 2.2.: PI Architecture Firmware Phases [**pi-spec**]

1. Security (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, wakup 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
    - 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 Cache as Random Access Memory (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 Pre-EFI Initialization (PEI) Foundation

- state of platform

- location and size of the Boot Firmware Volume (BFV)
- location and size of the temporary RAM
- location and size of the stack
- optionally one or more Hand-off Blocks (HOBs) via the SEC HOB Data PEIM-to-PEIM Interface (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. Pre-EFI Initialization (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 microarchitextures 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 statisfied 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 analogus 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 identifiy 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 HOBs

- describe Firmware Volume (FV) in HOBs
- pass control to Driver Execution Environment (DXE)

crisis recovery (what is this?) resuming from S3 sleep state linear array of RAM Pre-EFI Initialization Module (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 moudles from vendors for processors, chipsets, RAM

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

3. Driver Execution Environment (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 apropriate 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.

   dxe core/foundation platform independent is implementation of UEFI UEFI Boot Services UEFI Runtime Services DXE Services

   dxe dispatcher 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

dxe drivers init processor, chipset and platform produce arichtectural protocols and Input/Output (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. Boot Device Selection (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 arichtectural 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. Transient System Load (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. Runtime (RT) Boot service drivers have been unloaded and only runtime services are accessible.

   runtime services/driver

7. Afterlife (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.

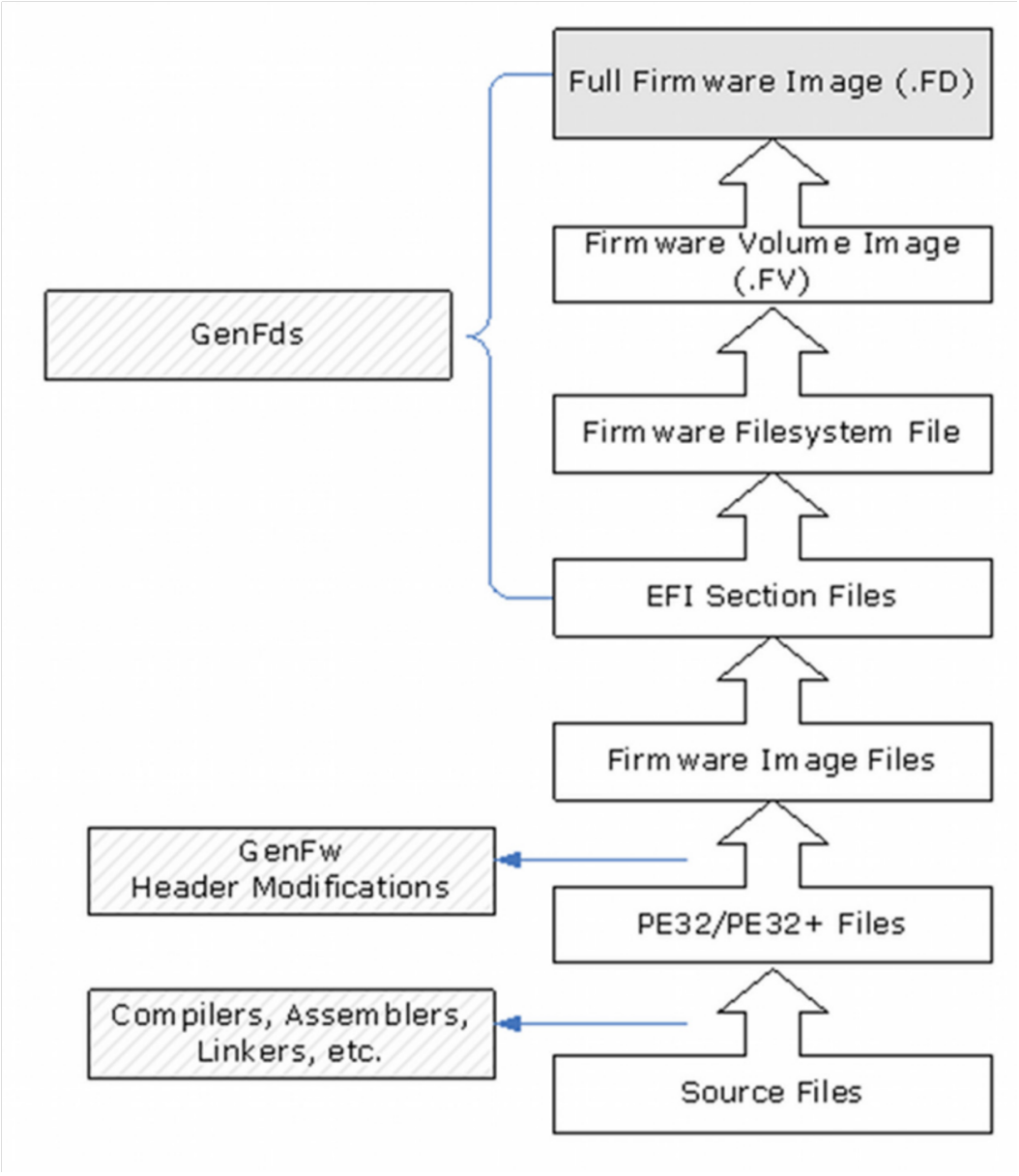
   hibernation sleep

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 signle 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**]



Flash Device (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

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

Firmware Filesystem (FFS) organization of files and free space no dierectory hierarchy all files flat in root dir parsing requires walking for beginning to end

firmware files types

some file types are sub-divided in file sections

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

dxe drivers files contain one PE32 executable section may contain version section may contain dxe 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 hasing pe sections omitting attribute certifacte 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 Trusted Platform Module (TPM) is a system component which enables trust in computing platforms helps verify if the Trusted Computing Base has been compromised securely storing passwords, certificates and encryption keys in separate state to host only communicating through a well defined interface. store platform measurements that help ensure that the platform remains trustworthy authentication attestation hardware and software implementations software special mode shielding TPM resources from normal execution [**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 Attem
5	Boot Manager Code Configuration and Data (for use by the Boot Manager Code) a
6	Host Platform Manufacturer Specific
7	Secure Boot Policy
8	First New Technology File System (NTFS) boot sector (volume boo
9	Remaining NTFS boot sectors (volume boot record)
10	Boot Manager
11	BitLocker Access Control

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

[**tianocore-trusted-boot-chain**]

TCG2 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**]

3 Windows 11

[TODO what is 11 compared to 10]

3.1 UEFI

3.1.1 Installation

For us to understand how UEFI threats act towards Windows we need to understand how the layout of the Windows installation integrates into the UEFI environment. This begins with the installation process and the partitioning of the hard drive Windows is installed onto. When the Windows Installer is launched, it creates at least four partitions on the target hard drive. The EFI System Partition (ESP), a recovery partition, a partition reserved for temporary storage and the boot partition containing the system files. Two copies of the Windows Boot Manager bootmgfw.efi are placed on the ESP, one under EFI\\Boot\\bootx64.efi for the default boot behavior the installed hard drive and one under EFI\\Microsoft\\Boot\\bootmgfw.efi alongside boot resources such as the Boot Configuration Data (BCD). The path of the latter boot manager is saved in a boot load option variable entry Boot####, which is then added to the BootOrder list variable. The boot load option contains optional data consiting of a GUID identifying the Windows Boot Manager entry in the BCD. The BCD, as its name suggests, contains arguments used to configure various steps of the boot process [**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, Windload.efi, the application repsonsible 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

the two signature Data Bases (DBs) Production and UEFI

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 travere 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 Application Programming Interface (API).

[TODO maybe fun fact that EFS cant encrypt hives] windows also has a feature called Encrypting File System (EFS) with file system level encryption but it cant be used for registry hives [**windows-internals-6-part2**]

3.3 Trusted Boot

3.3.1 KMCI

3.3.2 HVCI

3.4 BitLocker Drive Encryption (BDE)

Windows is only able to enforce security policies when it is active, leaving the system vulnerable when accessed from outside of the OS [**windows-internals-6-part2**]. Windows uses BitLocker, integrated Full Volume Encryption (FVE), aimed to protect system files and data from unauthorized acceccs 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 Figure 3.1. 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 Advanced Encryption Standard (AES)128 and AES256 cypher [**windows-internals-6-part2**]. A Full Volume Encryption Key (FVEK) is used in combination with the block index as input for the algorithm, resulting in an entirely

different output for two blocks with identical data [**windows-internals-6-part2**]. The FVEK is encrypted with a Volume Master Key (VMK) which is in turn encrypted with multiple protectors, these encrypted versions of the VMK are stored together with the encrypted FVEK in an unencrypted meta data portion at the beginning of the volume [**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 TCG2 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 Platform Configuration Registers (PCRs) {7, 11} with PCR7 binding {0, 2, 4, 11} without PCR7 binding 11 is required

Recovery key recovery key 48 digits of 8 blocks block is converted to a 16-bit value making up a 128-bit key [**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**]

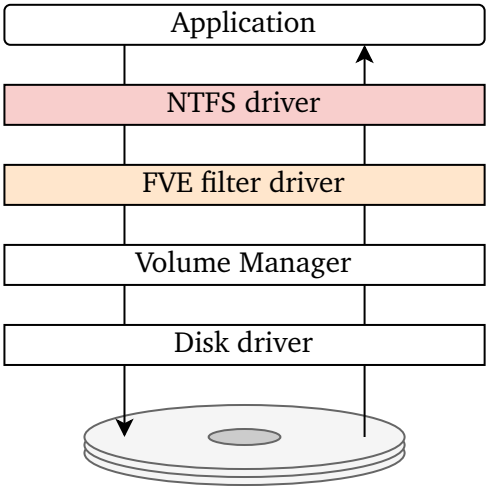


Fig. 3.1.: BitLocker Volume Access Driver Stack (inspired by [**windows-internals-6-part2**])

4 Past Threats

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

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. [**especter**] 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 FVs. 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 primarey 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 form the DXE dispatcher or through an intermediary loader application. This driver then hooks the UEFI boot service LoadImage. When this is either called by the UEFI boot manager or the loader application to load bootmgfw.efi, it patches the bootloader in memory [**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 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]

5 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 Original Equipment Manufacturers (OEMs) to decide, when implementing a firmware image.

5.1 QEMU

Our main development setup is an emulated environment using the emulator Quick Emulator (QEMU) [**qemu**] together with the Open Virtual Machine Firmware (OVMF) image, from EFI Development Kit (EDK) II (edk2-stable202208). For Secure Boot we generate our own Platform Key (PK) and use the *Microsoft Corporation KEK CA 2011* as Key Exchange Key (KEK) and the two signature DBs *Microsoft Windows Production PCA 2011* and *Microsoft Corporation UEFI CA 2011* from Microsoft. The former required for their UEFI executables used during the Windows boot process [**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]**

6 Results

We were able to implement UEFI attacks in the form of a UEFI firmware rootkit and a UEFI bootloader rootkit (bootkit), with both being able to deploy Windows level payload from within the UEFI environment using an NTFS drivers. Through our UEFI port of the reged utility we were able to modify the Windows registry, so that our Windows payload is executed with the privileges of the built-in local system account. The execution is done by the Task Scheduler at system boot. With Secure Boot enabled we showed that our bootkit was denied execution, while the execution of our rootkit is left unaffected. Although affecting infection by restricting software access to the firmware image. When BitLocker is used with a TPM and the default validation profile {0, 2, 4, 11} our root- and bootkit trigger the BitLocker recovery prompt, from which we our *BitLogger* was able to log entered keystrokes to obtain the recovery key. We were then able to use the recovery key with our UEFI port of Dislocker to mount the encrypted drive, allowing us to repeat our initial attack of deploying payload and modifying the registry. In the case of our UEFI payload being part of the TPM measurements used to encrypt the VMK or when a validation profile is used that does not include PCR0, we were able to sniff the communication between the TPM and the Windows Boot Manager to retrieve the unencrypted VMK for use with Dislocker. We showed that this is the case when using a Secure Boot configuration that uses only Microsoft's signature DB required to boot Windows. This forces a default validation profile of {7, 11}, leaving out PCR0. Our rootkit attack was able to gain access to this type of system without requiring any prior knowledge or additional user input.

7 Discussion

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

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

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

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

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

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

7.1 Mitigations

bios password against secure boot removal or bootkit installation from USB

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

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

inaccessible spi flash

tpm + pin/usb detectability

7.1.1 User awareness

you can change recovery message and URL in BCD hive

recovery guide

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

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

googeln wie legitime recovery key prompt reaktion aussieht

enterprise policy recovery key einschraenkbar?

enterprise policy on recovery key loss

vermitteln was das prompt bedeuten koennte

aber kann man einfach nicht anzeigen lassen

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

enterprise doesnt hand out recovery keys and instead receives hard drive

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

phishing prompts on their own

8 Conclusion

dxe runtime rootkit not rally feasible since it doesnt run without being called back by the os dxe smm rootkit makes sense

8.1 Achieved Goals

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

8.2 Future Work

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

List of Figures

List of Tables

35

List of Listings

Appendix

A

A.1 Protocols

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

extern EFI_GUID gEfiLoadedImageProtocolGuid;
extern EFI_GUID gEfiLoadedImageDevicePathProtocolGuid;
```

Listing A.1: Loaded Image Protocol

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

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

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

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

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

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

extern EFI_GUID gEfiSimpleTextInputExProtocolGuid;
```

Listing A.2: Simple Text Input Ex Protocol

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

struct _EFI_SIMPLE_FILE_SYSTEM_PROTOCOL
{
  UINT64 Revision;
  EFI_SIMPLE_FILE_SYSTEM_PROTOCOL_OPEN_VOLUME OpenVolume;
};

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

extern EFI_GUID gEfiSimpleFileSystemProtocolGuid;
```

Listing A.3: Simple File System and File Protocol

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

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

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

extern EFI_GUID gEfiDiskIoProtocolGuid;
```

Listing A.4: Disk I/O Protocol

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

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

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

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

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

extern EFI_GUID gEfiBlockIoProtocolGuid;
```

Listing A.5: Block I/O Protocol

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

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

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

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

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

extern EFI_GUID gEfiTcg2ProtocolGuid;
```

Listing A.6: TCG2 Protocol

Acronyms

B

ACPI Advanced Configuration and Power Interface

AES Advanced Encryption Standard

AL Afterlife

API Application Programming Interface

ASCII American Standard Code for Information Interchange

BCD Boot Configuration Data

BDE BitLocker Drive Encryption

BDS Boot Device Selection

BF Boot Firmware

BFV Boot Firmware Volume

BIOS Basic Input/Output System

RAM Random Access Memory

CA Certificate Authority

CAR Cache as Random Access Memory

CD Compact Disc

CSM Compatibility Support Module

DB Data Base

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 Full Volume Encryption Key

GPT GUID Partition Table

GUID Globally Unique Identifier

HOB Hand-off Block

I/O Input/Output

KEK Key Exchange Key

LBA Logical Block Address

MBR Master Boot Record

NTFS New Technology File System

OEM Original Equipment Manifacturer

OS Operating System

OVMF Open Virtual Machine Firmware

PCR Platform Configuration Register

PE32 Portable Executable 32-Bit

PEI Pre-EFI Initialization

PEIM Pre-EFI Initialization Module

PF Platform Firmware

PI Platform Initialization

PK Platform Key

PPI PEIM-to-PEIM Interface

QEMU Quick Emulator

RT Runtime

ROM Read-Only memory

SEC Security

TCG Trusted Computing Group

TPM Trusted Platform Module

TSL Transient System Load

UEFI Unified Extensible Firmware Interface

USB Universal Serial Bus

VBR Volume Boot Record

VMK Volume Master Key

XML Extensible Markup Language