

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.-Ing. Friedel Gerfers
Electrical Engineering and Computer Science
Technische Universität Berlin |
| <i>Supervisors</i> | Jane Doe and John Smith |

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.-Ing. Friedel Gerfers

Supervisors: Jane Doe and John Smith

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

[TODO past tense and rewrite] In Computer Security one of the most feared security threats is a rootkit, executing at the beginning of a computers boot chain, before the operating system and accompanying antivirus programs. With the widespread adaption of standardized UEFI firmware these threats have become less machine dependent and can now target a host of systems at once. Past analyses about bootkits have been case studies of their appearances in the wild, this thesis instead aims to be a more practical approach by developing a bootkit and analyzing the challenges doing so. We restrict our analysis by assuming an attacker has already gained read and write access to the BIOS image and is thus only facing security mechanisms involved during and with execution of the bootkit. Our bootkit was able to achieve elevated execution on Windows 11 by exploiting unrestricted hard drive access to edit Windows Registries, this was also possible on BitLocker encrypted hard drives by keylogging the Recovery Key. UEFI makes it very easy for an attacker who has gained access to the System Firmware to leverage its powers and gain full control over the system.

Abstract (different language)

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

Acknowledgement

Contents

1	Introduction	1
2	Background	3
2.1	UEFI/PI	3
2.1.1	Boot Sequence	3
2.1.2	UEFI/PI Firmware Images	6
2.1.3	UEFI Images	8
2.1.4	Firmware Core	9
2.1.5	Boot Manager	10
2.1.6	EDK II	11
2.1.7	Security	11
2.2	Windows	12
2.2.1	Installation	12
2.2.2	Boot Process	13
2.2.3	Registry	13
2.2.4	Trusted Boot	14
2.2.5	BitLocker Drive Encryption (BDE)	14
3	Related Work	17
3.1	Infection	17
3.1.1	Bootkit	17
3.1.2	Rootkit	17
3.2	Approach	17
3.2.1	Storage based	17
3.2.2	Memory based	18
4	Test Setup	19
5	Attacks	21
5.1	Neither Secure Boot nor Bitlocker	21
5.1.1	Bootkit	21
5.1.2	Rootkit	28

5.2	Secure Boot	30
5.2.1	Bootkit	30
5.2.2	Rootkit	30
5.3	Bitlocker	31
5.3.1	Infection	31
5.3.2	Approach	33
6	Discussion	39
6.1	Rootkit classification	40
6.2	Mitigations	40
6.2.1	User awareness	40
7	Conclusion	43
7.1	Achieved Goals	43
7.2	Future Work	43
A	Appendix	51
B	Acronyms	53

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 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 -

We start off introducing all background information necessary to understand this thesis in Chapter 2. With this knowledge we then look at analyses of previously discovered UEFI threats in Chapter 3. In Chapter 4 we start our practical approach by implementing a UEFI attack of our own to analyse security mechanism faced when attempting attacks from the UEFI environment. Afterwards we dicuss the impact of our findings as well as potential mitigation techniques in Chapter 5. Chapter 6 concludes ...

Background

The following introduces the background information necessary to understand the employment of a Unified Extensible Firmware Interface (UEFI) rootkit. This includes the general workings of the Platform Initialization (PI) and UEFI, the UEFI programming model and interface itself; as well as its security mechanisms. It is also necessary to understand our target's defenses, for this, we briefly describe the Windows security mechanisms faced when performing our attacks.

2.1 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].

UEFI is pure interface spec [**beyond-bios**] It was designed to replace the legacy Boot Firmware Basic Input/Output System (BIOS), while also often offering a backwards compatible mode with the Compatibility Support Module (CSM). The specification is a pure interface specification thus merely states what interfaces and structures a PF has to offer and what an OS may use. how it is implemented by PF what is used by OS 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 Boot Sequence

focus will be on dxr and transient system load



1. Security (SEC)

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)

- 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 modules 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)

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

dxs 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

dxs drivers init processor, chipset and platform produce architectural protocols and 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)

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 doesn't 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)

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

ExitBootServices()

6. Runtime (RT)

runtime services/driver

7. Afterlife (AL)

hibernation sleep

2.1.2 UEFI/PI Firmware Images

[**pi-spec**]



firmware device 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 directory hierar-
 chy 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 sections such as PE32 RAW
VERSION TE

dxs drivers files contain one PE32 executable section may contain version section
may contain dxs 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]

2.1.3 UEFI Images

files containing executable code subset of PE32+ file format with modified header
signature to distinguish from normal PE32 Images + stands for addition of 64-bit
relocation fix-up extension

relocatable fixed and dynamic address loading loaded fully into memory and reloca-
tion fix ups

three different subsystems types: application, boot service driver and runtime service
driver boot and runtime memory

application vs os loader vs driver memory they reside in unloaded on return unloaded
on error

memory marked as code and data jump to entry point

UEFI Applications

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

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

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

2.1.4 Firmware Core

Systemtable

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

Handles

[uefi-spec]

Protocols

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

Boottime Services

Runtime Services

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

2.1.5 Boot Manager

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

2.1.6 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]

2.1.7 Security

others not discussed further user identification

PEI GuidedSection Extraction

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]

Firmware Protection

DXE SMM Ready to Lock Vol4

Capsule Architectural Protocol

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

flash device security

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]

[tianocore-trusted-boot-chain]

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

2.2 Windows

2.2.1 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 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 New Technology File System (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**].

2.2.2 Boot Process

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 will show a recovery prompt, otherwise it proceeds to load the `Winload.efi` OS loader [**windows-internals-7-part2**].

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

2.2.3 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 hierarchical 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 exporting and importing registry keys along with their subkeys and contained values. Internally the registry is not a single large file but instead a set of files 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 can't encrypt hives] Windows also has a feature called Encrypting File System (EFS) with file system level encryption but it can't be used for registry hives [windows-internals-6-part2]

2.2.4 Trusted Boot

KMCI

HVCI

2.2.5 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 access while at rest [microsoft-bitlocker-overview], while also verifying boot integrity when used with a TPM [windows-internals-6-part2]. The encryption and decryption of the volume is done by a filter driver beneath the NTFS

driver as shown in Figure 2.1. 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 Globally Unique Identifier (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! (TCG!) 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 default Platform Configuration Registers (PCRs) are 0, 2, 4, 5, 8, 9, 10 and 11 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**]

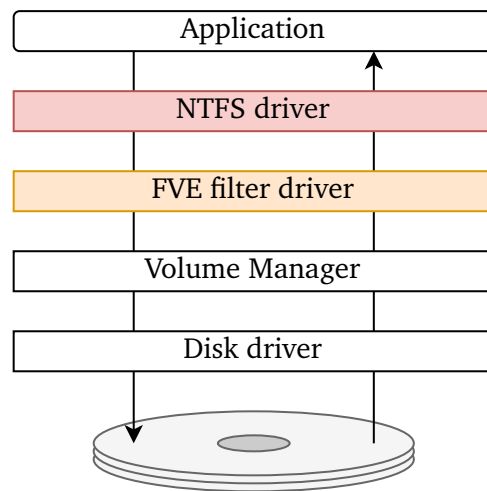


Fig. 2.1.: BitLocker Volume Access Driver Stack, inspired by [**windows-internals6-part2**]

Related Work

scholar ranking bootkits and rootkits often share the same attack vector towards windows because they are executed prior and during windows boot

UEFI threats can be categorized by their attack vector into two groups: storage based and memory based attacks. Storage based attacks mostly gain access independent of the current state of the operating system by only modifying the disk's content before the operating system access these files. These attacks are often performed before any parts of the operating system are executed. Memory based attacks instead hook into the operating system's boot process to install their payload alongside operating system in memory.

3.1 Infection

3.1.1 Bootkit

3.1.2 Rootkit

3.2 Approach

3.2.1 Storage based

vector-edk

LoJax

rootkit documentation about firmware infection remove previous NTFS driver add malicious DXE driver has NTFS driver payload registry editor C:/Windows/SysWOW64/autoche.exe is not executed with elevated privileges (can't update TPM values)

EFI_EVENT_GROUP_READY_TO_BOOT (too early for BitLogger)

MosaicRegressor

rootkit has NTFS driver C:/ProgramData/Microsoft/Windows/Start Menu/Programs/Startup
no registry editing, also not privileged

3.2.2 Memory based

FinSpy

bootkit bootmgfw.efi replaced and original moved original bootloader patched in memory patches "function of the OS loader that transfers execution to the kernel" hooks the kernel's PsCreateSystemThread and which then creates an additional thread decrypting the next stage and loading it

ESpecter

bootkit

MoonBounce

CosmicStrand

Test Setup

[TODO describe test setup] qemu + swtpm

fresh Windows 11 installation

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

5.1 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.

5.1.1 Bootkit

Infection

We have two ways to infect a system, we can either use a bootable USB stick with a UEFI application installing the bootkit or a Windows executable mounting the ESP with admin privileges. 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.

Approach

[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. For our storage-based approach we now need to access the Windows installation from within the UEFI environment to deploy our payload. We want to access the NTFS formatted Windows boot partition, this requires an additional NTFS driver due to the UEFI specification only mandating compliant firmware to support FAT12, FAT16 and FAT32 **[uefi-spec]**. The EFI Development Kit (EDK) II reference implementation does not provide an NTFS driver either.

File access

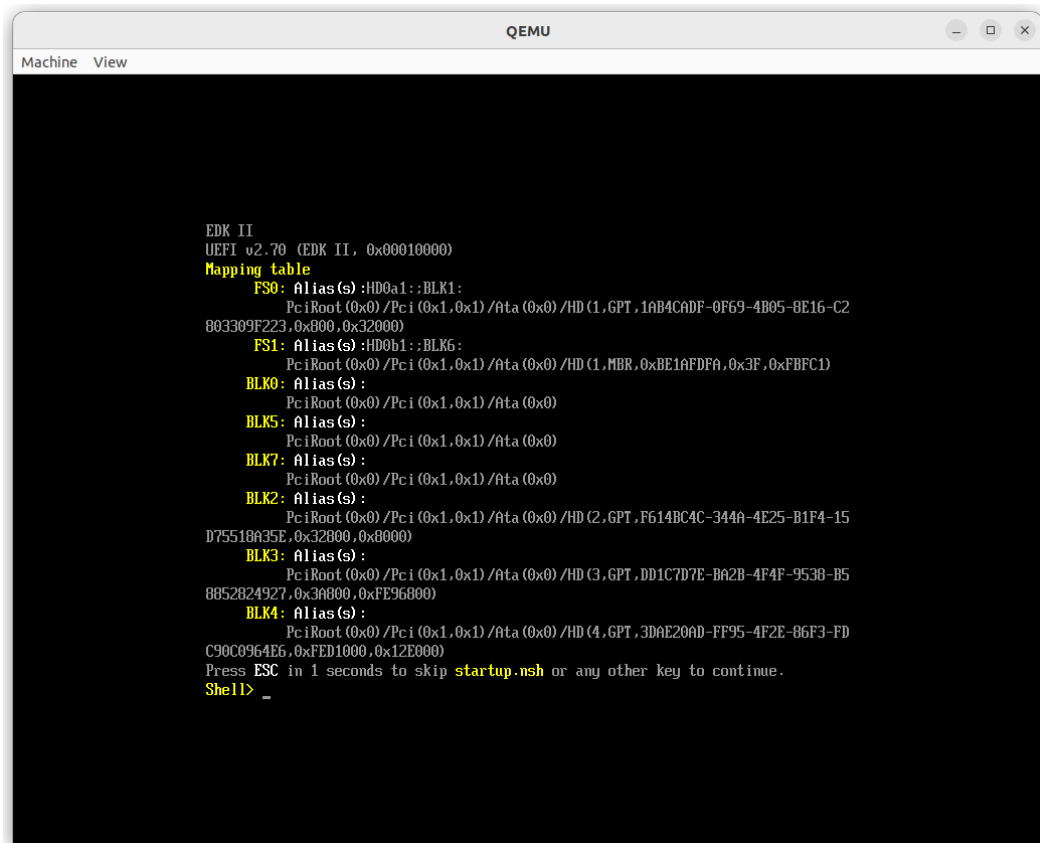
We can use a fork of the open source NTFS driver `ntfs-3g` from Tuxera **[ntfs-3g]**, that was ported to the UEFI environment by *pbatard* **[ntfs-3g-uefi]**.

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 Universal Serial Bus (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]**.

A screenshot of a QEMU window titled "QEMU" with a "Machine View" tab. The window displays the UEFI shell's mapping table. The text is as follows:

```
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
    D75518A35E,0x32000,0x8000)
BLK3: Alias(s) :
    PciRoot (0x0) /Pci (0x1,0x1) /Ata (0x0) /HD (3,GPT,DD1C7D7E-BA2B-4F4F-9538-B5
    8852824927,0x3A000,0xFE96000)
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> _
```

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 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 `command drivers`, we can list all currently loaded drivers and some basic information about them, such as number of devices managed. We can see that the NTFS driver already manages devices.

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

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

[TODO Windows file access privileges] We now navigate into the Windows folder to test whether we have unrestricted read and write access, since 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.

We now know that provided we get to load the NTFS driver we can 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 control 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.

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.

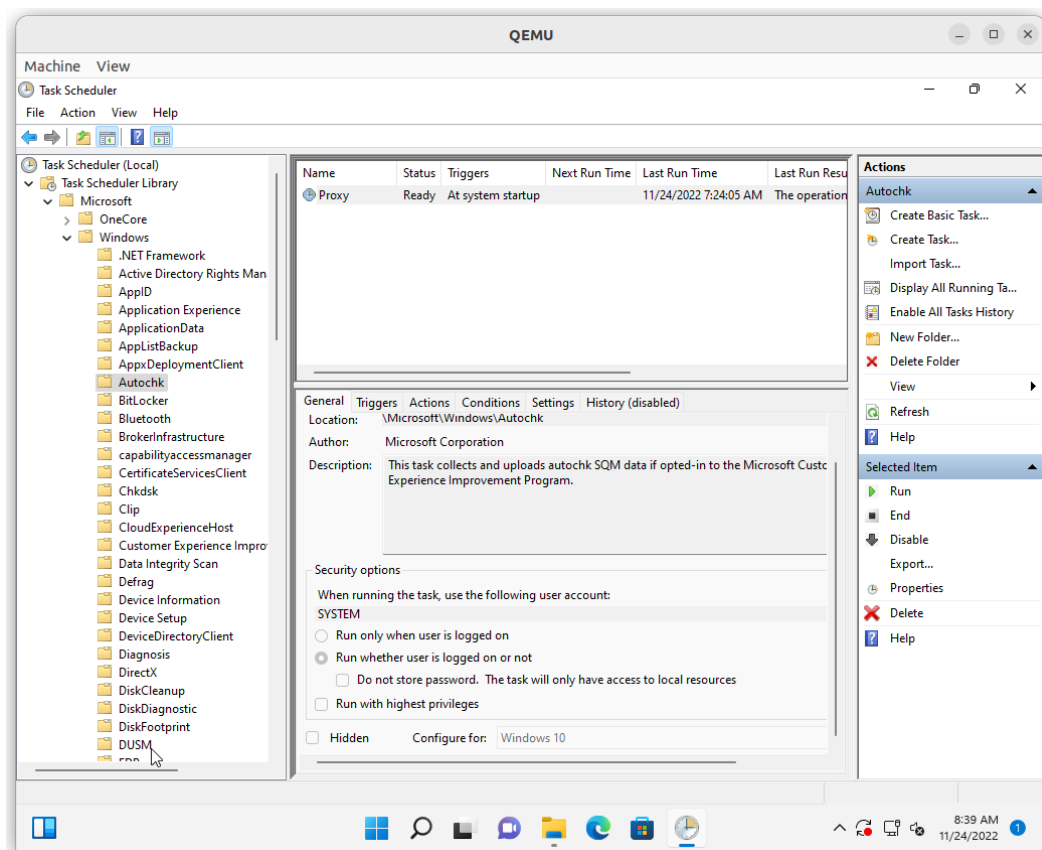
[TODO DLL proxy loading] [TODO modifying Windows Executables KMCI]

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 Dynamically Linked Library (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-internals-7-par**]. There also exist a secondary copy of the task descriptors, on the regular file system under `%windir%\system32\Task`, stored as Extensible Markup Language (XML) files.

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



We can use this exported key and import it on our victim's system as part of our attack. 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 overrides the task descriptor and when booting into Windows our payload is executed.

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

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

5.1.2 Rootkit

Performing the same attack in the form of a rootkit is very similar. The UEFI payload is now compiled as a DXE driver instead of an application, so that it is loaded when the DXE Dispatcher iterates over the FV it is contained within.

Infection

retrieve current firmware image modify image by adding our DXE drivers to the DXE volume rewrite the image

For this, we have to read out the firmware image, modify the contents and write the new image back to hardware. This can be done by using a spi flash programmer and clamping the physical chip. Or using an SPI chip emulator. **[TODO properly list the options]** If we want to use emulation we can build the Open Virtual Machine Firmware (OVMF) Package from EDK II which is a firmware image for virtual machines.

Now that we have the image we can edit it with UEFITool, which is an editor for firmware images conforming to the UEFI PI spec **[uefitool]**. In UEFITool we navigate to the DXE Volume containing the DXE Core and DXE drivers.

Before adding our driver we remove any other NTFS driver packed in the image by OEMs, because they might be read-only or otherwise restricted and would inhibit our NTFS driver from installing onto a device, if it were to load earlier. UEFITool offers a search through the entire firmware image. We can search for "NTFS" as a case-insensitive string, since most drivers either have a User Interface Section which contains a human-readable name for inspection tools like UEFITool **[pi-spec]** or support the optional Component Name Protocol which is part of the UEFI Driver Model and returns the name of a driver **[uefi-spec]**. If this would not suffice it is possible to search for the NTFS magic number indicating that a volume is formatted with NTFS, this number is ASCII encoded NTFS followed by four white spaces and likely contained by an NTFS driver.

If we now want to add our NTFS driver .efi file with UEFITool we cannot do this directly, because DXE drivers have three mandatory sections: PE32 executable, version and DEPEX section **[pi-spec]** and these are not automatically generated.

For these files to be generated it is easiest to simply build the NTFS driver as part of the EDK II Open Virtual Machine Firmware (OVMF) and have it packaged in a firmware volume. This can be an unused volume or for debugging purposes the

DXE volume, for real hardware we can use the output of the build process which is a .ffs file. The .ffs file contains the Portable Executable 32-Bit (PE32), version, Dependency Expression (DEPEX) and an optional user interface section.

For the rootkit to write a payload to disk it needs to know what to write, we can create an EDK II module with a Windows targeted executable and have it packaged as a binary, this produces a .ffs file of type EFI_FV_FILETYPE_FREEFORM, which puts no restrictions on the contained file sections [pi-spec]. The output contains only one file section of type EFI_SECTION_RAW which contains the binary payload.

We can now simply insert the .ffs file into the target firmware image with UEFITool.

overwrite the SPI flash with modified image by using the programmer again.

Approach

When the rootkit is executed it starts by reading the payload into memory, this is done by calling the boot service function LocateHandleBuffer with the option ByProtocol and the GUID for gEfiFirmwareVolume2ProtocolGuid which returns all handles that have a protocol instance associated with gEfiFirmwareVolume2ProtocolGuid installed onto them [uefi-spec]. We can now iterate over all protocol instances, open the protocol and query the firmware volume for the GUID of our binary payload, we then read the content of the raw section into a buffer. [TODO maybe size match on hardware]

payload has to also reside in the firmware image, EDK II binary module

different protocol to load the payload

we dont have to load the other drivers now as this is done by the DXE Dispatcher

we dont have to load the windows boot manager as this is still done by the UEFI Boot Manager

5.2 Secure Boot

mostly comes with default keys OEM

5.2.1 Bootkit

Infection

booting into usb installer doesnt work without disabling secure boot, if you already have access to the device and have to change the boot order to include removal media, you could very well also disable secure boot have to assume that bios is password protected what happens when removable media is still before windows

windows installer executable should still work without any difference

Approach

expectation: not to boot observation: doesnt boot

5.2.2 Rootkit

Infection

add DXE Drivers to the DXE Volume. This can be achieved by having read/write access to the SPI flash or using the Signed Capsule Update. Gaining read/write access to the SPI Flash is possible either through physical access to the device by using an SPI clamp on the chip itself or through exploits like for example the . Signed Capsule Updates can be leveraged with access to private vendor information by signing the payload to make it appear legitimate or by intercepting the distribution process and employing infected firmware.

Approach

no difference secure boot default policy snippet option roms and bootloader instead relies on Signed Capsule Updates assumes integrity

[uefi-spec]

policy defined [uefi-spec]

5.3 Bitlocker

Our final attack will be targeted against systems using BitLocker FVE with a TPM 2.0 and no additional PIN or startup key. This leaves the Windows boot partition encrypted, the ESP is left unencrypted, thus not affecting the bootkit installation process. Secure Boot can be enabled in combination of BitLocker having the same effects on this attack as described in the second attack section 5.2. 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.

5.3.1 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 Figure 5.1, interrupts the regular boot process requiring the drive's recovery key for decryption before being able to continue booting. This happens due to TPM PCR values differing from what was initially used to seal the VMK, leaving the Windows bootloader unable to retrieve an unencrypted VMK from the TPM and as a result unable to decrypt the Windows installation [[windows-internals-7-part2](#)].

In theory this is as far as we get, BitLocker in combination of TPM measurements successfully mitigates UEFI attacks by discovering a deviation in the boot flow. In practice we have to ask ourselves the question how a user reacts to seeing the BitLocker recovery prompt and the consequences to the action the user takes. As an immediate reaction the user has two options: entering the recovery key into the prompt or not entering the recovery key. What decision the user makes is dependent on their tech savviness and influenced by a variety of factors such as urgency of booting into Windows, knowing alternatives to what the prompt tells them. It is reasonable to assume that the average user is willingly entering their recovery key in response to the prompt as the prompt does not suggest any malicious causes or any negative repercussions in following the instructions. The link mentioned in the



Fig. 5.1.: BitLocker Recovery Prompt

prompt also only aids in locating the recovery key [[microsoft-recovery-key-faq](#)].
[TODO mention microsofts reasons for the prompt to be triggered] [TODO what is the actual alternative]

5.3.2 Approach

When the user enters their recovery key the Windows Boot Manager uses the recovery key to decrypt the VMK meta data 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 in the UEFI boottime environment, before ExitBootServices was 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.

BitLogger

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 simply log the keystrokes performed by the user when entering the key during the recovery prompt. Since we still are in the UEFI boottime environment, the Windows Boot Manager uses UEFI protocols for user input. 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 alone triggers the recovery prompt because of PCR value changes. A keystroke now shows us that the *Simple Text Input Ex Protocol* is being used, the protocol structure is depicted in Listing 5.1. The Windows Boot Manager uses the `ReadKeyStrokeEx` function to retrieve the latest pending keypress. The protocol offers the `WaitForKeyEx` event, signaling when keystrokes are available, execution can be blocked until this event is emitted with the `WaitForEvent` Boot service.

```
1 typedef struct _EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL
2 {
3     EFI_INPUT_RESET_EX Reset;
4     EFI_INPUT_READ_KEY_EX ReadKeyStrokeEx;
5     EFI_EVENT WaitForKeyEx;
6     EFI_SET_STATE SetState;
7     EFI_REGISTER_KEYSTROKE_NOTIFY RegisterKeyNotify;
8     EFI_UNREGISTER_KEYSTROKE_NOTIFY UnregisterKeyNotify;
9 } EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL;
```

Listing 5.1: Simple Text Input Ex Protocol Structure

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. **[TODO if we have time talk about downsides of function patching]** UEFI protocol hooking does not require such an invasive 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, the latter additionally keeps track of the protocol users **[uefi-spec]**, Listing 5.2 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 this protocol instance, given they use the same device handle. 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*.

```

1 EFI_SIMPLE_TEXT_INPUT_EX_PROTOCOL *simple_text_input_ex;
2 SystemTable->BootServices->HandleProtocol(SystemTable->ConsoleInHandle,
3                                           &gEfiSimpleTextInputExProtocolGuid,
4                                           (VOID **)&simple_text_input_ex);

```

Listing 5.2: Example usage of the `HandleProtocol` Boot service

The BitLocker recovery prompt does not allow the user to input any [TODO key input advancement is weird and makes tracking tricky] F keys block validity only divisible by 11 [windows-internals-6-part2] cursor can move out of incomplete or valid blocks up and down increments or decrements the cursor position

Dislocker

To make use of the recovery key we can use an open source software called *Dislocker*, which implements the Filesystem in Userspace (FUSE) interface to offer mounting

of BitLocker encrypted partitions under Linux supporting read and write access [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 GUIDs 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*, we can depict the NTFS protocol stack in Figure 5.2.

It becomes obvious that all file-wise operations are, in multiple layers of abstraction on top of block-wise access to the media, performed through the *Block I/O Protocol*. Inspired by the BitLocker filter driver in Figure 2.1, which de- and encrypts each block as it passes through, we hook the *Block I/O Protocol* functions `ReadBlocks` and `WriteBlocks`, their signatures are shown in Listing 5.3. We then perform Dislocker operations on writes and reads Figure 5.3.

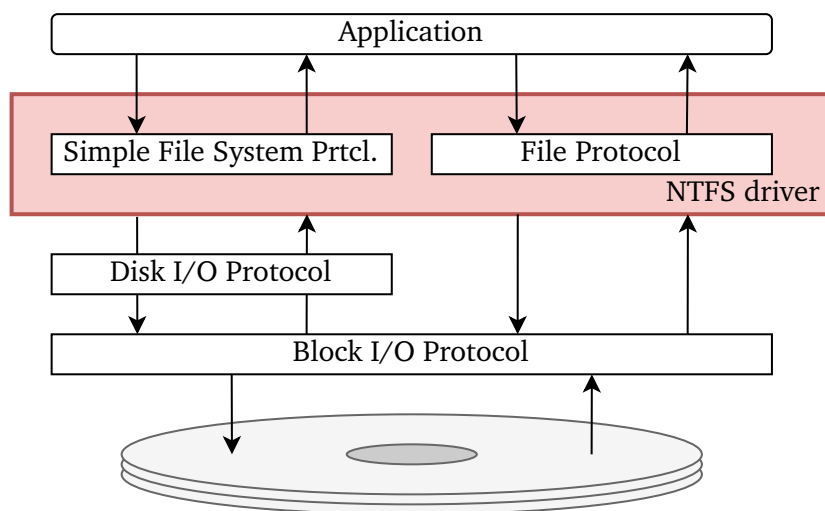


Fig. 5.2.: UEFI Volume Access Protocol Stack

```

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

```

Listing 5.3: Block I/O Protocol Structure and ReadBlocks/WriteBlocks Function Signature

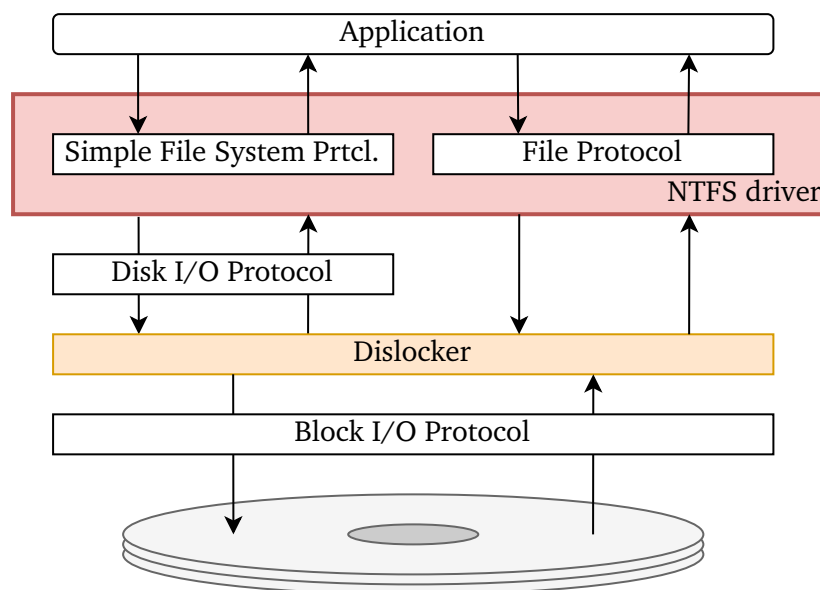


Fig. 5.3.: Dislocker Volume Access Protocol Stack

When we look at the Dislocker source code, we find that Dislocker works with two main functions `dislock` and `enlock`, they each take offset-length parameters, comparable to the *Disk I/O Protocol* abstraction. `dislock` reads and decrypts, while `enlock` encrypts and writes. Internally Dislocker uses `pread` and `pwrite` to access the volume. These operations are always performed on whole blocks, as BitLocker encryption is done block-wise. So the starting offset is rounded down and the offset plus length is rounded up 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.

It is beneficial for us to write our payload to the Windows installation as close to the end of the UEFI environment as possible, this will maximize the presence of drivers and their offered access to hardware devices. It is also a wise design decision for the attacks following to this one. The call of the function `ExitBootServices` marks the point of transitioning from boot time to runtime where the operating system takes over the control of the system, it presents a good opportunity for us to perform the write action of our rootkit. **[exitbootservices-hooking]** hook `ExitBootServices` enable hook write payload import registry key disable hook

next boot would require to recovery key again update tpm values in payload **[microsoft-bitlocker-manage-bde]** caveat pin? look into this

BitLocker post Infection

persistence when part of root of trust fresh install / tpm update values hook TCG2 Protocol [**tcg-efi-protocol-spec**] TPM communication receive bitlocker VMK key and send to dislocker [**bde-format-spec**] [**tpm-sniffing**]

meaning, importance, and relevance of your results explaining and evaluating what you found, showing how it relates to your literature review

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 override 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 priveleged 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 very effective if part of the delivery process or in general present before os installation BitLogger somewhat last resort social engineering aspect

you can change recovery message and URL in BCD hive

not yet done: prevent firmware update

boottime vs runtime rootkit

6.1 Rootkit classification

statistiken zu bitlocker und secureboot auf systemen

industrie standard zur system security in firmen

6.2 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

6.2.1 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 Figure 5.1

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 einschränkbar?

enterprise policy on recovery key loss

vermitteln was das prompt bedeuten könnte

aber kann man einfach nicht anzeigen lassen

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

enterprise doesn't 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

7.1 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

7.2 Future Work

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

List of Figures

- 2.1 BitLocker Volume Access Driver Stack, inspired by [**windows-internals6-part2**] 16
- 5.1 BitLocker Recovery Prompt 32
- 5.2 UEFI Volume Access Protocol Stack 36
- 5.3 Dislocker Volume Access Protocol Stack 37

List of Tables

List of Listings

- 5.1 Simple Text Input Ex Protocol Structure 33
- 5.2 Example usage of the HandleProtocol Boot service 34
- 5.3 Block I/O Protocol Structure and ReadBlocks/WriteBlocks Function
Signature 36

Appendix

A

Acronyms

AL	Afterlife
AES	Advanced Encryption Standard
API	Application Programming Interface
BCD	Boot Configuration Data
BDE	BitLocker Drive Encryption
BDS	Boot Device Selection
BF	Boot Firmware
BFV	Boot Firmware Volume
BIOS	Basic Input/Output System
CAR	Cache as Random Access Memory
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
FD	Firmware Device
FFS	Firmware Filesystem
FUSE	Filesystem in USErspace
FV	Firmware Volume
FVE	Full Volume Encryption

FVEK Full Volume Encryption Key

GUID Globally Unique Identifier

HOB Hand-off Block

I/O Input/Output

NTFS New Technology File System

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

PPI PEIM-to-PEIM Interface

RAM Random Access Memory

RT Runtime

SEC Security

TCG2 Trusted Computing Group 2

TSL Transient System Load

TPM Trusted Platform Module

UEFI Unified Extensible Firmware Interface

USB Universal Serial Bus

VMK Volume Master Key

XML Extensible Markup Language

