# 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

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

*2. Reviewer*   Prof. Dr.-Ing. Friedel Gerfers
Electrical Engineering and Computer Science
Technische Universität Berlin

*Supervisors*   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

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

# Introduction

what is UEFI what does it different than bios this helps write platform independent code uefi threats: bootkits and rootkits in recent years root and bootkits have popped up in the wild and been analysed we will discuss their commonalities implement one ourselves discuss security mechanisms we encounter reflect their weaknesses how to potentially evade them

UEFI + threats against windows what are root and bootkits

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

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. [cro; Tec] **[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. [cro] bootkit definition goals

danger of uefi infection root-/bootkits attack vectors: - storage based - memory based analyse security mechanism to prevent these attacks by attempting an attack itself increasing security mechanisms

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 ...

- revise and catagorize previously analysed UEFI threats - implement a uefi threat - reason about infection scenarios - analyse countermeasures against UEFI threats - Trusted Boot: KMCI from windows - Secure Boot - TPM - Bitlocker - firmware lock + signed capsule update -

# 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 Window's 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" [For].

UEFI is pure interface spec [ZRM17] It was designed to replace the legacy Boot Firmware Basic Input/Ouput 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 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 Boot Sequence

focus will be on dxe and transient system load

Pre Verifier

verify

CPU Init

Chipset Init

Board Init

Device, Bus, or Service Driver

Exposed API

OS-Absent App

Transient OS Environment

Transient OS Boot Loader

Previously exposed DXE APIs now limited

DXE Dispatcher

Boot Dispatcher

OS-Present App

Boot Services Runtime Services DXE Services

Final OS Boot Loader

Final OS Environment

?

security

| Security (SEC) | Pre EFI Initialization Environment (PEI) | Driver Execution Environment (DXE) | Boot Device Selection (BDS) | Transient System Load (TSL) | Run Time (RT) | After Life (AL) |
|---|---|---|---|---|---|---|

Power on ➤ [ . . Platform initialization . . ] ⟶ [ . . . . OS boot . . . . ] ⟶ Shutdown

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 RAM (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 **FV!** (**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 [For20]

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)

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 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 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 [For21]

   [For20]

   - Initializing console devices

   - Loading device drivers

   - Attempting to load and execute boot selections

5. Transient System Load (TSL)

   boottime and runtime services/driver bootloader [For21, 13.3 System Partition] [For21, p. 3.5.1.1]

   ExitBootServices()

6. Runtime (RT)

   runtime services/driver

7. Afterlife (AL)

   hibernation sleep

## 2.1.2  UEFI/PI Firmware Images

[For20, Volume 3, 2.1]

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

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

**FFS!** (**FFS!**) organization of files and free space no 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

[Tiaa]

## 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 reloaction 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

what is the boot manager boot manager = bds

**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 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

## 2.1.4 Firmware Core

**Systemtable**

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

**Handles**

[For21, 7.3 Protocol Handler Services]

**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 [For21, 2.4 Protocols] boot services provide function to install, locate, open, close and monitor protocols [For21, 7.3 Protocol Handler Services] identified with guids

**Boottime Services**

**Runtime Services**

## 2.1.5 edk2

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 []

## 2.1.6 Security

others not discussed further user identification

PEI GuidedSection Extraction

**Secure Boot**

[Tiac]

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

[Micb]

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

[TODO authenticated variables]

### 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 [Gro08] [Gro19]

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

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

[Tiab]

TCG2 Protocol [**tcg-efi-protocol**]

## 2.2 Windows

### 2.2.1 Trusted Boot

**KMCI**

**HVCI**

### 2.2.2 BitLocker

[MI09, 8.4 BitLocker Drive encryption] BitLocker Drive Encryption (BDE) integrates with operating system encryption enabled per volume encrypt os and data drives supports removable data drives maximum protection with TPM 1.2 or later alternatively USB startup key or password, not system integrity verification optionally PIN or USB startup key required to unlock also network unlock and pin as fallback

- tpm only no additional user interaction - tpm with startup key additional usb - tpm with PIN - tpm with startup key and PIN protects against unauthorized data access

with tpm ensures integrity of early boot components and boot configuration

system requirement include support for TCG-specified Static Root of Trust Measurement

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

encryption on used disk space only or whole drive former security risk if turned on after drive was already in use, deleted data accessible with disk recovery tools latter the following is recommended encrypted hard drive support

two partitions - operating system partition with os and support files, all system files on the volume, including the paging files and hibernation files, bitlocker encrypted, ntfs - system partition with windows boot manager and minimal software required for decryption of the os, fat32, unencrypted, files needed

data is encrypted blockwise with Full Volume Encryption Key (FVEK) - AES 128-bit the key is 128-bit of size - AES 256-bit the key is 256-bit of size FVEK encrypted with Volume Master Key (VMK) 256 bit VMK encrypted by multiple protectors, default configuration: - TPM, seal operation - Recovery Key

or - startup key/external key

**Recovery Key**

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

**Clear Key**

unprotected 256-bit key stored on the volume t decrypt vmk

**Startup Key**

stored in a .bek file with GUID name equaling key identifier in bitlocker meta data multiple possible for a single bitlocked volume

**Startup Key**

password with max 49 characters

# Related Work

<div style="text-align: right; font-size: 3em;">3</div>

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 catigorized 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 Storage based

### 3.1.1 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)

### 3.1.2 MosaicRegressor

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

## 3.2 Memory based

### 3.2.1 FinSpy

bootkit bootmgfw.efi replaced and orignal 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 addtional thread decrypting the next stage and loading it

### 3.2.2 ESPecter

bootkit

### 3.2.3 MoonBounce

### 3.2.4 CosmicStrand

# Attacks

<div style="text-align: right; font-size: 3em;">4</div>

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

transfer UEFI execution to Windows execution by installing payload elevated execution of payload

## 4.1 Test Setup

[TODO describe test setup] qemu + swtpm

## 4.2 Neither Secure Boot nor Bitlocker

The first attack is performed without enabling any additional security mechanisms (e.g. Secure Boot, BitLocker).

### 4.2.1 Bootkit

deviate the boot flow access to the hard drive from the UEFI Environment deploy payload so that it is executed

Windows uses NTFS formatting for its primary file system [Micc], thus we need to use an NTFS driver to access the Windows installation. The UEFI specification mandates compliant firmware to support FAT12, FAT16 and FAT32 [For21, p. 13.3.1.1], since this does not include NTFS, EDK II does not come with an NTFS driver.
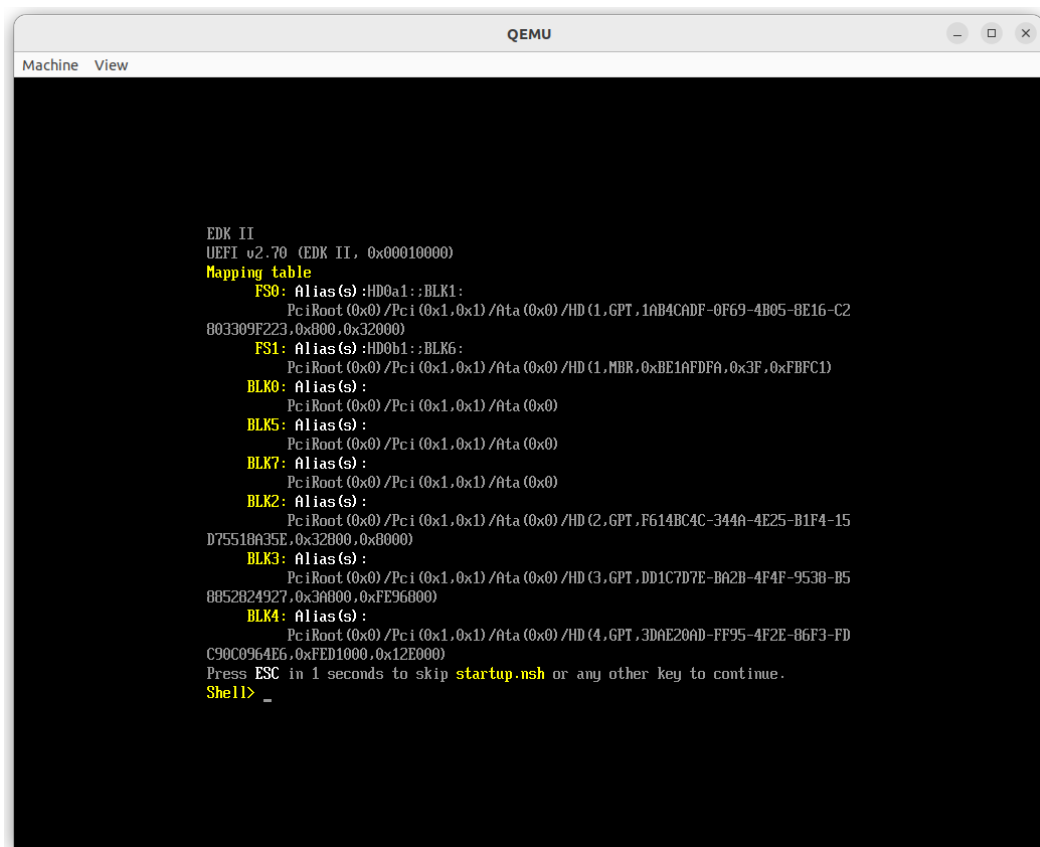
NTFS drivers are readily available such as ntfs-3g from tuxera [tux] which allow for read and write access, on top of that there already exists a port to UEFI created by pbatard [pba].

We can compile this driver with EDK II following the listed steps and receive a .efi executable file.

[TODO better summary of UEFI shell] Part of the UEFI specifications is a shell specification which is a feature rich UEFI shell application to interact with the UEFI environment. [TODO better listing of capabilities] It offers commands for boot, configuration, device, driver, handle filesystem network memory scripting. For now driver loading and filesystem navigation are of relevance.

[TODO look up official guideline to booting into console] The UEFI shell can be part of the boot options or requires the user to supply the executable via a removable medium such as a USB stick.

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



The map command is very interesting for file access with the shell, it displays a mapping table between user defined alias names and device handles. The aliases can

be used instead of a device path when submitting commands via the command line interface. The UEFI shell also produces default mappings, notably for file systems [For16, 3.7.2. Mappings]. These mappings are designed to be consistent across reboots as long as the hardware configuration stays the same, they are comparable to Windows partition letters. [For16, Appendix A]

[TODO find in spec what precise mapping mechanism] When we inspect the mapping table we can see FSx and BLKx aliases, FSx maps to file systems and BLKx to block devices. This identification is performed via instances of the Simple File System Protocol and [TODO double check] Block I/O Protocol which present the interfaces to access the devices they are installed to. The Simple File System Protocol [For21, 13.4 Simple File System Protocol] provides, together with the File Protocol, file-type access to the device it is installed on. 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 now, this is done by first entering the alias, for the file system containing the NtfsDxe.efi, followed by a colon. [TODO alias for fs eingeben was macht das] This effectively switches the console's working directory to be the root of the entered file system, now we caninvoke load with the path to the executable. The output indicates whether loading the driver was successful. [TODO maybe better command here] 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.

[TODO reword maybe move the block device stuff up] We can now reset all default mappings with the map -r command to receive an updated list including the file system 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 indicating that the NTFS driver uses this block-wise access to offer the Simple File System Protocol.

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

Upon debugging the NTFS driver it appears to be that the drivers falls back to read only when it encounters a file that indicates that the Windows system is in hibernation mode. Windows seems to have hibernation enabled by default and as such our rootkit should not rely on it being disabled, we can change the code of the NTFS driver to not fallback when encountering this file.

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. We can put the NTFS driver into the DXE Volume where the DXE Dispatcher of the DXE Core will automatically try to load it.

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. If we want to use emulation we can build the Open Virtual Machine Firmware (OVMF) Package from edk2 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 [Lon]. 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 controlling a device. UEFITool offers a search through the entire firmware image. We can search for "NTFS" as a case insenstive string, since most drivers either have a User Interface Section which contains a human-readable name for tools like UEFITool[For20, Vol 3, 3.2.5] or support the optional Component Name Protocol which is part of the UEFI Driver Model and returns the name of a driver [For21, p. 11.5]. If this would not suffice it would be 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.

If we now want to add our NTFS driver .efi file with UEFITool we notice that it is displayed as having an unknown subtype, this is because added a file without file sections, a DXE driver has three mandatory sections: PE32 executable, version and DEPEX section [For20, Vol 3, 2.1.4.1.4]. For these files to be generated it is easiest to simply build the NTFS driver as part of the edk2 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 PE32, version, DEPEX and an optional user interface section. We can now simply insert the .ffs file into the target firmware image with

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

If we now boot up the UEFI shell the splash screen immediately lists the NTFS formatted device in the mapping table indicating that the driver was successfully loaded during DXE dispatch.

[TODO why did we choose dxe stage]

The next step is to now use theblo NTFS driver programmatically by writing a DXE driver that leverages the new file system access to write an executable to the Windows installation.

For the rootkit to write a payload to disk it needs to know what to write, we can create an edk2 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 [For20, Vol 3, 2.1.4.1.7]. The output contains only one file section of type `EFI_SECTION_RAW` which contains the binary payload.

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 [For21, p. 7.3]. 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]

The write operation is performed by calling `LocateHandleBuffer` again, this time to query all protocol instances of the Simple File System Protocol. We iterate over all instances and open their respective volumes, opening a volume returns an instance of the File Protocol. This represents the root directory of the volume, we can now attempt to open a path that is part of the Windows installation, this will fail on volumes not containing a Windows installation, which we just skip. Eventually the volume containing Windows will be found.

[TODO Windows File Permissions] Now the question arises as to where to write our payload to, we want autmoatic and elevated execution. Earlier we discovered that the NTFS DXE driver disregards the file access permission model so we are not restricted in the same way an unprivileged user would be. MosaicRegressor writes its payload to the Windows startup folder, a folder whose contents are automatically

exectuted at system startup. The programs within the startup folder are unfortunatly not automatically run at an elevated level.

[TODO modifying Windows Executables KMCI]

There is a better mechanism used for the automatic execution of privileged programs, the Task Scheduler. It can be used to schedule task execution at a variety of different conditions, examples are disk fragmentation or antivirus scans. [TODO which priveleges are possible] Tasks are defined in XML files under `Windows/System32/Tasks`, they are initially read into memory and cached into registry keys. Subsequent startups operate on the cached registry keys. `HKEY_LOCAL_MACHINE/SOFTWARE` hive `Microsoft/Windows NT/CurrentVersion/` [TODO explain what the registry is]

[**windows-internals-7-part2**]

We can modify an existing task that is scheduled to execute with elevated privileges and have it execute our payload instead of the original. A fitting task is the Proxy in the Autochk folder
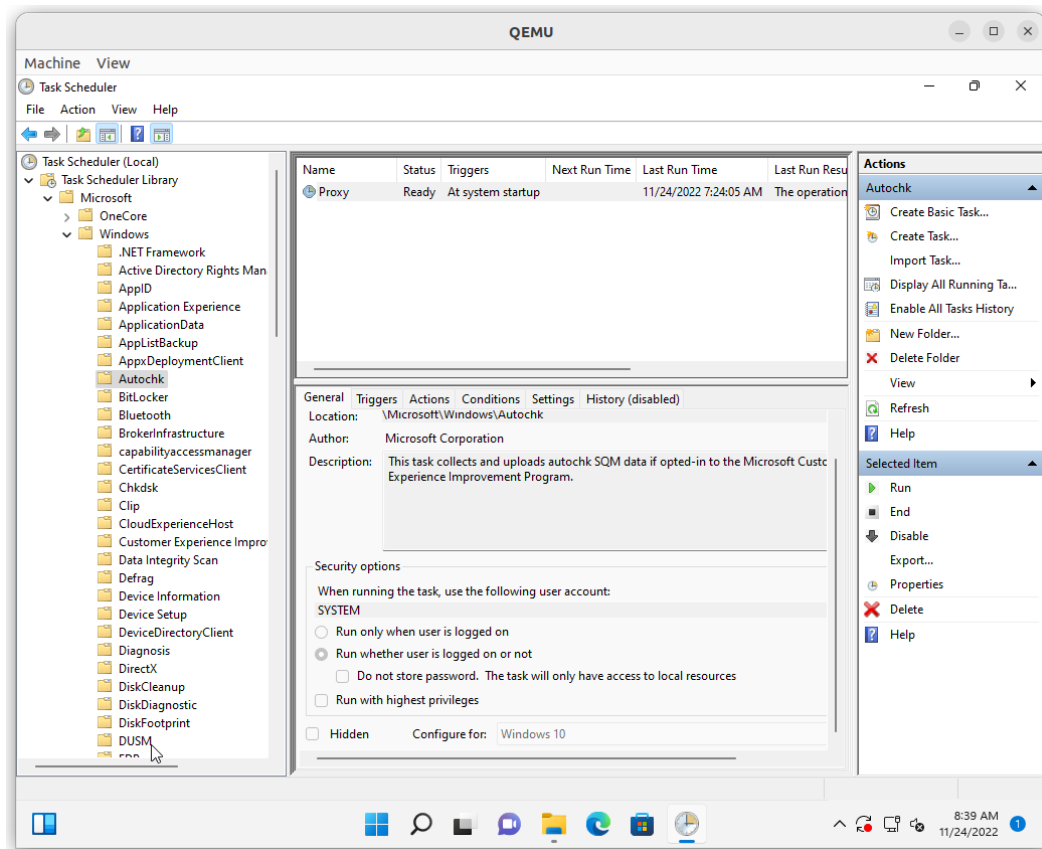


We can modfiy the action it performs to run our payload instead, to verify the privileges our payload is executed with, we can save the output of the `whoami` into a file.

The `whoami` command shows the current user and privileges [**windows-whoami**]. After manually triggering the task we see that our payload was run from the `nt authority\system` user account, which is the most privileged system account [Mica].

We can export this modified registry key and as part of our attack import it on our victim's system. To import the key we can use a linux utility called chntpw whose primary purpose it is to reset the password of local windows user accounts [Nor]. 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 dualbooting a Linux and a Windows installation. We place our payload in the Windows installation and then boot into Linux, where we can simply open the `HKEY_LOCAL_MACHINE/SOFTWARE` hive with reged and import our modified registry key. This overrides the executable path of task's action and when booting into Windows our payload is now executed.

The next step is to port the reged utiliy so that it works in the UEFI environment as a DXE driver, that our rootkit can interact with. We will call this driver RegedDxe from now on. This 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. Definitions can often be translated to UEFI equivalents, EDK2 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 UEFI protocols like Simple File System Protocol and `FileProtocol`.

[TODO no need to change reged source code]

RegedDxe unlike the original linux utility reged is not an application that is executed within a working directory, it is instead a DXE driver offering a protocol for other drivers to call.

When importing a registry key the target path for the key is read from the registry key file, cached task's registry keys do not follow a strict naming sheme and are instead named by GUIDs, thus they may differ from device to device. To combat this, we change the import mechanism, so that instead of importing a key into its exact path it iterates over the children of the target's parent key. We can then match for

our target key with a value that correctly identifies the target key, such as the value representing the path of the XML, that was initially used to load the task.

## 4.3 Secure Boot

mostly comes with default keys OEM expectation: not to boot observation: doesnt boot

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.

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

[For21, p. 32.3]

policy defined [For21, 32.5.3.3 Authorization Process]

## 4.4 Secure Boot and Bitlocker

For our third attack we will enable BitLocker, this prevents us from trivially accessing any data before Windows has successfully booted. As we have learned from our second attack Secure Boot does not matter for our attack vector thus we can assume Secure Boot being enabled for this scenario. **[TODO secure boot and bitlocker standard and cant do much more?]**

We configure BitLocker to use automatic TPM decryption without any additional PIN or Startup Key.

When booting the system with our rootkit we are greeted with a screen prompting us to enter the BitLocker recovery key.
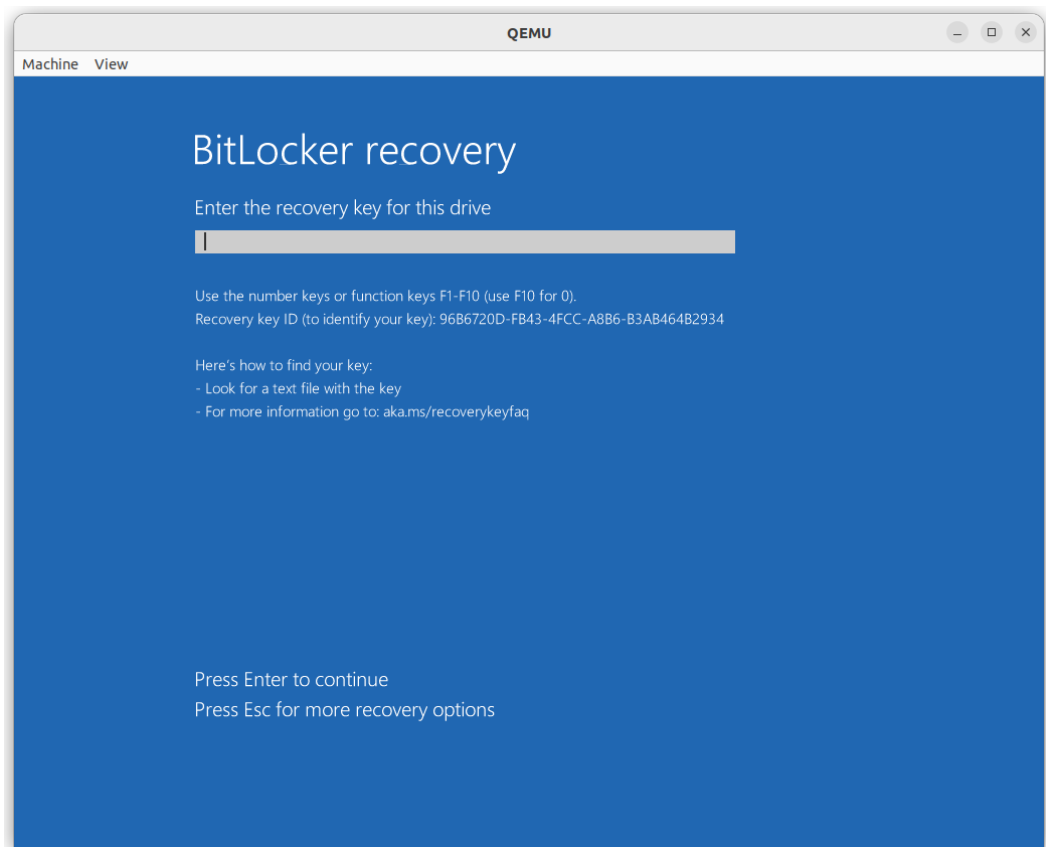
This happens due to TPM PCR values differing from what was initially used to seal the VMK, leaving the Windows bootloader unable to retrieve an unencrypted VMK from the TPM and as a result unable to decrypt the Windows installation. **[TODO the two different bitlocker volumes]**

**[TODO which measurements are used for sealing and unsealing] [TODO which ones are altered by the rootkit being present]**

In theory this is as far as we get, BitLocker in combination of TPM measurements successfully mitigates UEFI attacks by disovering 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 saviness 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 reprocussions in following the instructions. The link mentioned in

the prompt also only aids in locating the reocvery key [Micd]. **[TODO mention microsofts reasons for the prompt to be triggered] [TODO what is the actual alternative]**

Having the user enter their recovery key does not directly benefit our endevaours as BitLocker does not en/decrypt the hard drive as a whole upon boot but instead performs these respective action when reading or writing a block of data to or from the hard drive.

What we can do is try to record the keystrokes performed by the user while entering their recovery key and use it at some later point to gain control over the hard drive. A program designed to perform this type of attack is called a keylogger.

To implement such a keylogger we have to alter the code execution that happens when the recovery prompt is shown. For this our first step is to find our what execution environment or UEFI stage we are in, is Windows already using separate drivers for keyboard access or still relying on the UEFI environment and its protocols.

The UEFI specification defines two protocols which are used to abstract keyboard input these are the `{SimpleTextInputProtocol` and the `{SimpleTextInputExProtocol` [For21, pp. 12.2, 12.3]. To figure out if the recovery prompt uses these to read key strokes we can just add some print statements to the `ReadKeyStroke` and `ReadKeyStrokeEx` functions of their implementations in the edk2 OVMF package. On the next boot when typing we find that our `ReadKeyStrokeEx` print statements from the `{SimpleTextInputExProtocol` are triggered.

Now, knowing that the BitLocker recovery prompt is shown by an application running in the UEFI boottime environment, we can leverage this environment to implement our keylogger.

**[TODO how are protocols returned to the end user]** To alter the code execution when performing a key stroke we can just iterate over all instances of the `{SimpleTextInputExProtocol` and reassign the `ReadKeyStrokeEx` entry of the struct, which is a function pointer. We will save each protocol instances' original function address and instead have it point to an intermediary function. This intermediary calls the original function and performs logging of the key stroke result before relaying the result to the caller. This method is called function hooking and is intransparent to consumer of a protocol.

**[TODO how is the SimpleTextInputEx Protocol used]**

So far we are able to track each keystroke that queried via the `ReadKeyStrokeEx` function, which in our testing is only done during the recovery prompt, but may be used by BIOS interactive menu.

**[TODO key input advancment is weird and makes tracking tricky]** F keys block validity only divisable by 11 cursor can move out of incomplete or valid blocks up and down increments or decrements the cursor position

alternatively screen shot still need hook to find when enter is pressed explain how screenshotting works some basic compression wait for recovery key send recovery key on enter press

on real hardware network stack wasn't installed onto handles when boot over ip was disabled compared loaded dxe drivers between both configurations with efi shell Realtek Family driver not loaded load manually reinstall all handle to controllers to enable network stack regardless

sending key out is only good for physical access attack vector dislocker linux utility [Aor] mount encrypted drive with decryption mean read and write access dual boot in vm enter recovery key and it works port to uefi

bitlocker encrypts block-wise

**[TODO explain block devices] [TODO explain file system indepent abstraction better]** uefi protocol stack **[TODO diagram of block io, disk io, simple filesystem and file protocol interaction (with hindsight of adding dislocker beneath block io)]** [For21, 13.3.2 Partition Disocvery] Drivers providing Simple File System Protocol use the Block I/O Protocol to access the underlying media.

hook block io again hook data mapping

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. [Use] hook ExitBootServices enable hook write payload import registry key disable hook

next boot would require to recovery key again update tpm values in payload caveat pin? look into this

persistence when part of root of trust fresh install / tpm update values hook Trusted Copmuting Group 2 (TCG2) Protocol TPM communication [Gro14, p. 6.7.3] receive bitlocker vmk key and send to dislocker

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

# Discussion

<div style="text-align: right; font-size: 3em;">5</div>

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

we didnt try to be undetectable windows is very vulnerable with unrestricted disk access secure boot is very limited secure boot can easily be disabled without bios password TPM does its job in detecting PCR change bitlocker reocvery 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

boottime vs runtime rootkit

## 5.1 Rootkit classification

statisken zu bilocker und secureboot auf systemen

industrie standard zur system security in firmen

## 5.2 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

### 5.2.1 User awareness

recovery guide

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

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

googeln wie legitime recovery key prompt reaktion aussieht

enterprise policy recovery key 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

# Conclusion

<span style="color:#1b75bc; font-size:3em">6</span>

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

## 6.1 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

## 6.2 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

# Bibliography

[] (Cit. on p. 10).

[Aor] Aorimn. *Dislocker* (cit. on p. 27).

[cro] crowdstrike. *Rootkit Malware* (cit. on p. 1).

[For20] UEFI Forum. *UEFI Platform Initialization (PI) Specification, Version 1.7 Errata A*. 2020 (cit. on pp. 5, 6, 20, 21).

[For16] UEFI Forum. *UEFI Shell Specification, Revision 2.2*. 2016 (cit. on pp. 18, 19).

[For21] UEFI Forum. *UEFI Specification, Version 2.9*. 2021 (cit. on pp. 6, 9, 10, 17, 19–21, 24, 26, 27).

[For] UEFI Forum. *UEFI Specifications Overview* (cit. on p. 3).

[Gro14] Trusted Computing Group. *TCG EFI Platform Specification, Version 1.22 Revision 15*. 2014 (cit. on p. 28).

[Gro08] Trusted Computing Group. *Trusted Platform Module (TPM) Summary*. 2008 (cit. on p. 11).

[Gro19] Trusted Computing Group. *Trusted Platform Module Library, Part 1: Architecture, Level 00 Revision 01.59*. 2019 (cit. on p. 11).

[Lon] LongSoft. *UEFITool* (cit. on p. 20).

[MI09] David A. Solomon Mark E. Russinovich and Alex Ionescu. *Windows Internals*. 5th ed. Microsoft Press, June 2009 (cit. on p. 12).

[Mica] Microsoft. *LocalSystem Account* (cit. on p. 23).

[Micb] Microsoft. *Microsoft Windows Authenticode Portable Executable Signature Format, Version 1.0* (cit. on p. 11).

[Micc] Microsoft. *NTFS overview* (cit. on p. 17).

[Micd] Microsoft. *Where to look for your BitLocker recovery key* (cit. on p. 26).

[Nor] Petter Nordahl-Hagen. *Chntpw* (cit. on p. 23).

[pba] pbatard. *ntfs-3g* (cit. on p. 17).

[Tec] Techtarget. *Definition rootkit* (cit. on p. 1).

[Tiaa] TianoCore. *EDKII Build Specification* (cit. on p. 8).

[Tiab] TianoCore. *Trusted Boot Chain* (cit. on p. 11).

[Tiac] TianoCore. *Understanding UEFI Secure Boot Chain* (cit. on p. 10).

[tux]        tuxera. *ntfs-3g* (cit. on p. 17).

[Use]        User71491. *ExitBootServices Hooking* (cit. on p. 27).

[ZRM17]    Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Developing with the Unified Extensible Firmware Interface, Third Edition*. Berlin, Boston: De|G Press, 2017 (cit. on p. 3).

# List of Figures

# List of Tables

# List of Listings

# Appendix

A

# Acronyms

**AL**  Afterlife

**BDS**  Boot Device Selection

**BF**  Boot Firmware

**BFV**  Boot Firmware Volume

**BIOS**  Basic Input/Ouput System

**CAR**  Cache as RAM

**CSM**  Compatibility Support Module

**DXE**  Driver Execution Environment

**EFI**  Extensible Firmware Interface

**HOB**  Hand-off Block

**OS**  Operating System

**PEI**  Pre-EFI Initialization

**PEIM**  Pre-EFI Initialization Module

**PF**  Platform Firmware

**PI**  Platform Initialization

**PPI**  PEIM-to-PEIM Interface

**RT**  Runtime

**SEC**  Security

**TSL**  Transient System Load

**TPM**  Trusted Platform Module

**UEFI**  Unified Extensible Firmware Interface