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

1 Introduction

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

what does it different than bios this helps write platform independent code uefi threats: A rootkit is a collection of software designed to grant a threat actor control over a system, typically with malicious intend. Rootkits set up a backdoor exploit and may deliver additional malware while leveraging their privileges to remain hidden. There are different types of rootkits such as User Mode, Kernel Mode, Bootkits (bootloader rootkits), Hypervisor and Firmware rootkits. [@cro21; @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. [@cro21]

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 ...

2 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.

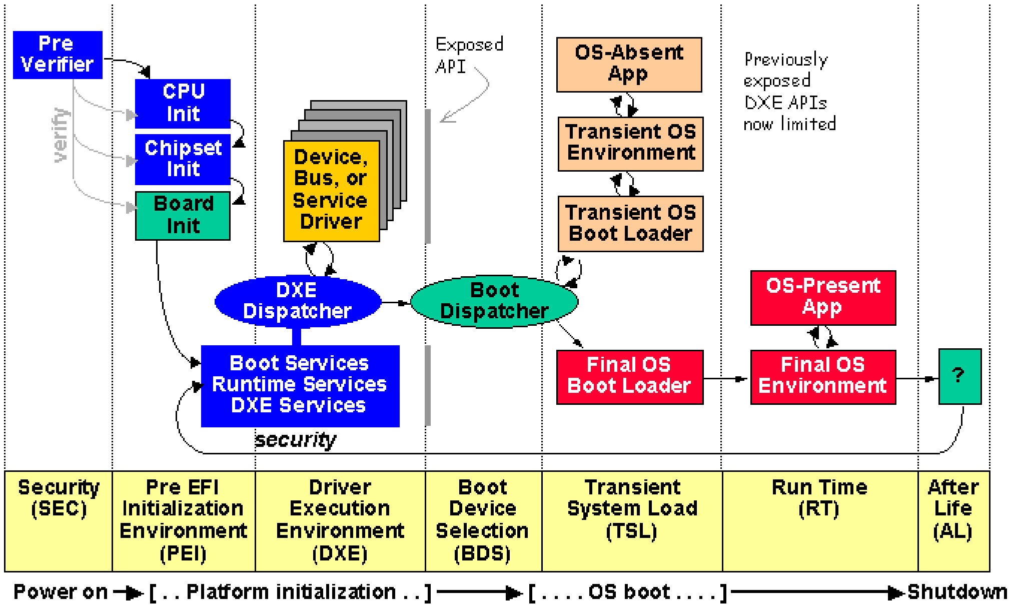
2.1 UEFI/PI

"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/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 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



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 [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)

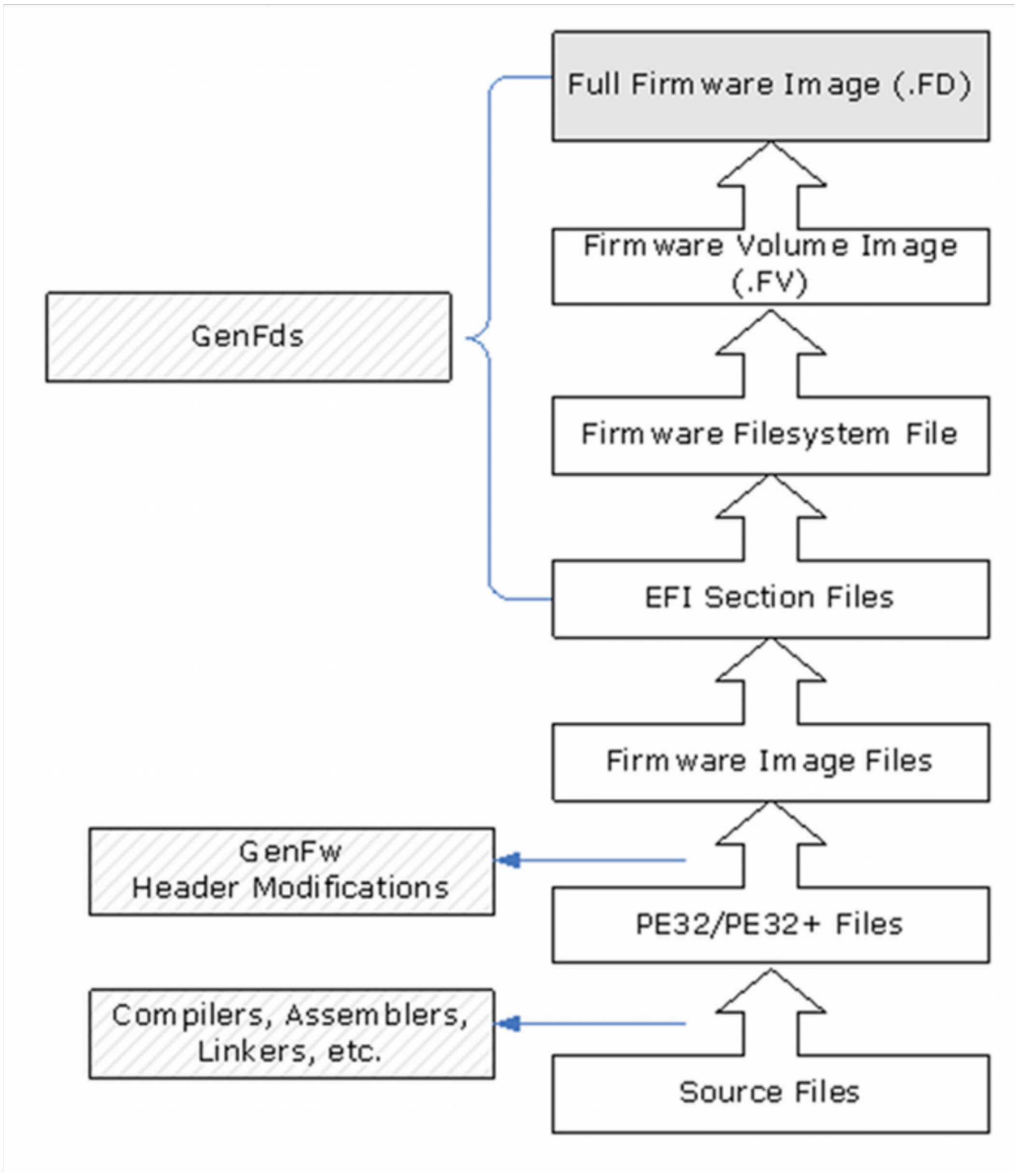
   runtime services/driver

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

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

Firmware File System (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

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

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 [For21, p. 8.2] **[TODO deep dive in authenticated variables]** architecually defined variables are called Globally Defined Variables where vendor GUID is defined with the macro `EFI_GLOBAL_VARIABLE` [For21, p. 3.3] 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 [For21, p. 3.1] 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 [AS21]

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 [@Tiab]

2.1.7 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

[@Mica]

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 [@Tiac]

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, [@Tiad]

[@Tiad]

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

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 consiting of a GUID identifying the Windows Boot Manager entry in the BCD. The BCD, as its name suggests, contains arguments used to configure various steps of the boot process [AS21, 12. The Windows Boot Manager]. 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, `Windload.efi`, the application repsonsible for loading the kernel into memory [AS21, 12. The Windows OS Loader].

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 [AS21, 12. The Windows Boot Manager]. Depending on what hibernation status is set within the BCD it may launch the `Winresume.efi` application, which reads the hibernation file and resumes kernel execution [AS21, 12. Launching a boot application]. 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 faile it will show a reocvery prompt, otherwise it proceeds to load the `Windload.efi` OS loader [AS21, 12. Launching a boot application].

[TODO TPM interaction] [AS21, 12. Launching a boot application]

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. [PS17, 1. Registry] The

Registry is a hierachical database containing keys and values, keys can contain other keys or values, forming a tree structure. Values store data through various data types. It is comparable to a file system structure with keys behaving like directories and values like files [AS21, 10. The registry - Registry data types]. At the top level it has 9 different keys [AS21, 10. The registry - Registry logical structure]. Normally Windows users are not required to change Registry values directly and instead interact with it through applications providing setting abstractions. Though some more advanced options may not be exposed and can be accessed through the `regedit.exe` application which provides a graphical user interface to travere and modify the Registry [AS21, 10. The registry - Viewing and changing the registry]. It also supports ex- and importing registry keys along their subkeys and contained values. Internally the registry is not a single large file but instead a set of file called hives, each hive contains one tree, that is mapped into the Registry as a whole. There is no one to one mapping of registry root key to hive file, the BCD file for example is also a hive file and is mapped into the Registry under `HKEY_LOCAL_MACHINE\BCD00000000` [AS21, 10. The registry - Registry logical structure]. Some hives even reside entirely in memory as a means of offering hardware configuration through the Registry Application Programming Interface (API).

2.2.4 Trusted Boot

KMCI

HVCI

2.2.5 BitLocker

[TODO write me] [@Micb]

operating system can only protect when it's active bitlocker protects system files and data against physical access attacks or generally outside of operating system windows supports Encrypting File System (EFS) but it cant be used for registry hives [MI12, 9. BitLocker Drive encryption]

[@Micc] [@Micd] BitLocker Drive Encryption (BDE) integrates with operating system fixed disk or BitLocker To Go 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

- 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 [@Mice]

with tpm ensures integrity of early boot components and boot configuration

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

[@Micd]

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 to 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

User Key

password with max 49 characters

3 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 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 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 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

ESPecter

bootkit

MoonBounce

CosmicStrand

4 Test Setup

[TODO describe test setup] qemu + swtpm

fresh Windows 11 installation

5 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 without enabling the optional security mechanisms Secure Boot and BitLocker. 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

[TODO write more careful about what we have and what we do not have] 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 that can mount the ESP with admin privileges. The installation process is identical for both options, we first locate and access the ESP and create a copy of the Windows Boot Manager, that is located under `EFI\\Microsoft\\Boot\\bootmgfw.efi`. We then replace the original with our bootkit as well as drop all our payload and other required files on the ESP.

Approach

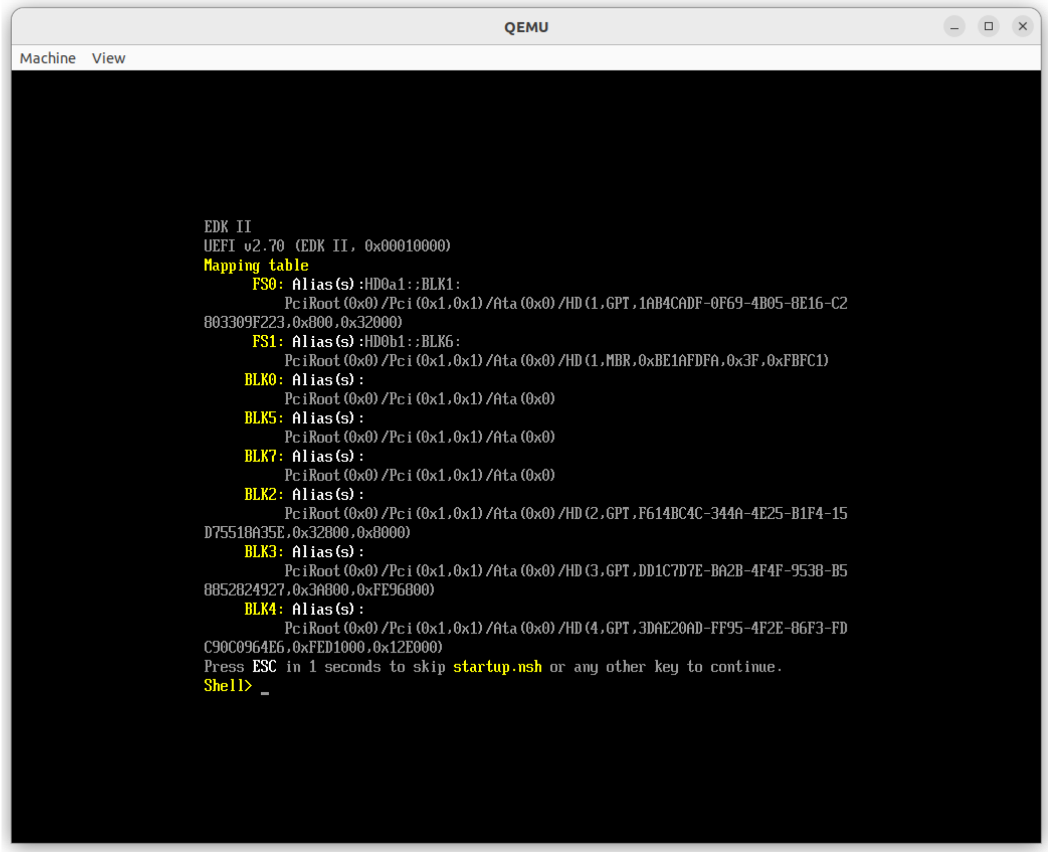
[TODO dump a windows boot entry] Now that our bootkit is in place of the Windows Boot Manager, when the UEFI Boot Manager selects the boot load option `Boot####` for the Windows Boot Manager, the file path `EFI\\Microsoft\\Boot\\bootmgfw.efi` will cause our bootkit to be executed. For our storage-based 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 [For21, p. 13.3.1.1]. The EFI Development Kit (EDK) II reference implementation does not provide an NTFS driver either. We can use a fork of the open source NTFS driver `ntfs-3g` from Tuxera [@tux], that was ported to the UEFI environment by *pbatard* [@pba].

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

[TODO better summary of UEFI shell] Part of the family of UEFI specifications is a shell specification which defines a feature rich UEFI shell application to interact with the UEFI environment [For16, p. 1.1]. It offers commands related to boot and general configuration, device and driver management, file system access, networking [For16, p. 5.1] and supports scripting [For16, p. 4]. 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` [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. 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 [For21, 13.5 File Protocol]. The two protocols are independent of the underlying file system the media is formatted with.

Our NTFS UEFI Driver is one such abstraction and needs to be loaded, this is done by first entering the alias, for the file system containing the `NtfsDxe.efi`. This effectively switches the console's working directory to be the root of the entered file system, now we can invoke `load` with the path to the executable. The output indicates whether loading the driver was successful. With the command drivers, we can list all currently loaded drivers and some basic information about them, such as number of devices managed. We can see that the NTFS driver already manages devices.

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

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

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

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

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 [For21, 9.1 EFI Loaded Image Protocol]. This handle can then be used to call the Boot Services `LoadImage` and `StartImage` to load and execute the NTFS driver. Since the driver conforms to the UEFI Driver Model, we need to also reconnect all controllers recursively, so it can assume controller over the NTFS formatted volumes, by installing the *Simple File System Protocol* on their handles. Loading the payload and other non-executable files into memory is done differently, here we use the handle from the Loaded Image Protocol to open the *Simple File System Protocol* installed onto the ESP, we can then call the `OpenVolume` resulting in an instance of the File Protocol representing the root folder of the volume [For21, p. 13.4]. This instance can then be used to open and read our payload with the absolute path on the ESP into memory. To perform the write operation we now need a handle we did not yet interact with, at least directly. We can use the Boot Service `LocateHandleBuffer` to receive an array of all handles that support the *Simple File System Protocol*, this includes volumes such as the ESP but also the Windows recovery partition. We can iterate over all handles to open the volume and attempting to create a new file with a file path that's inside of the Windows installation. This operation fails on volumes not containing a Windows installation which we can just skip. Eventually the volume containing Windows is found and the file is created and opened successfully, we can then write our payload, that we read into memory earlier, onto the disk and close the file again.

Now the question arises as to where to write our payload to, we want automatic and elevated execution. Earlier we discovered that the NTFS DXE driver disregards the file access permission model [TODO Windows File Permissions] so we are not restricted in the same way an unprivileged user would be when accessing the disk. *MosaicRegressor* writes its payload to the Windows startup folder, a folder whose contents are automatically executed at system startup. The programs within the startup folder are unfortunately not automatically run at an elevated level, so this isn't a suitable target location.

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

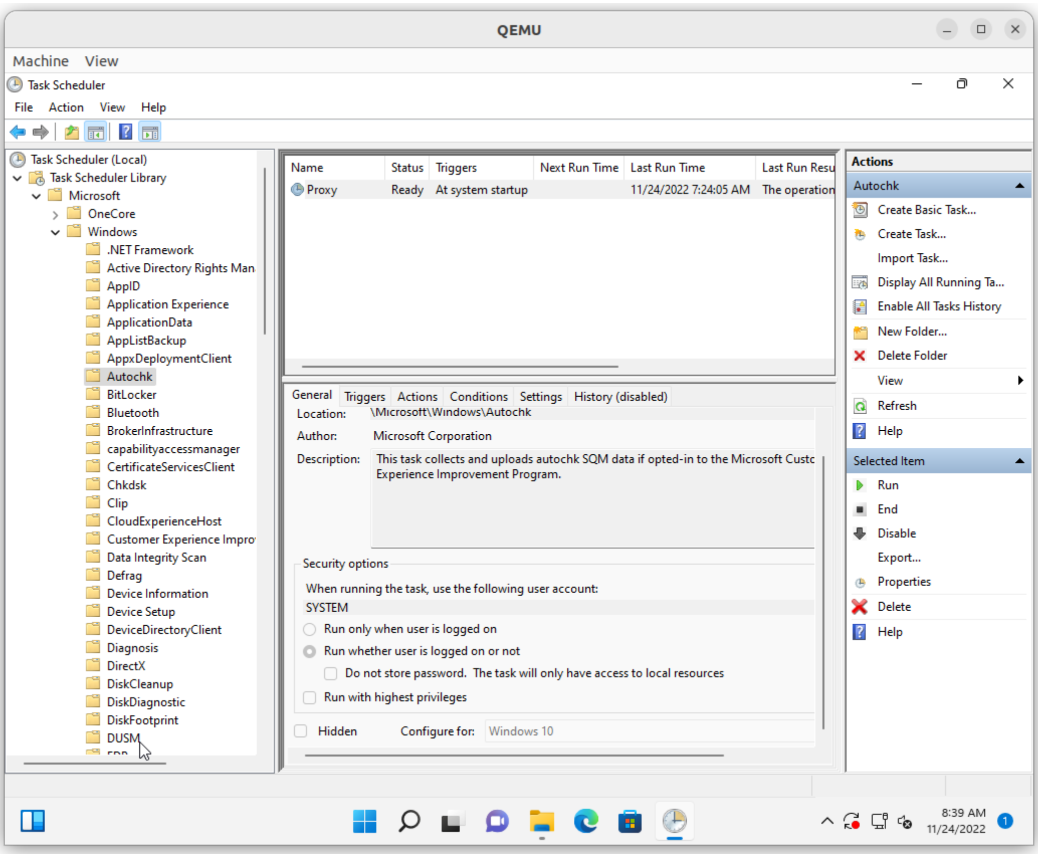
The Task Scheduler is a Windows service responsible for managing the automatic execution of background tasks [AS21, 10. The Task Scheduler]. Tasks are performed on certain triggers, which may be time-based (periodically or on a specific time) or event-based, for example on user logon or system boot[@Micf]. A task can perform various actions upon invocation [@Micg], 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 [@Mich]. The idea of our attack is to have a task, that performs its action with a high privelege level, execute our payload. The task of our choosing is called `Autochk\Proxy`, that performs the command

```
%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`

[@Mici]. 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 Globally Unique Identifier (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` [AS21, 10. The Task Scheduler - Initialization]. 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 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 [@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 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.

Extensible Firmware Interface (EFI) Loaded Image Protocol [For21, p. 9.1] and EFI Loaded Image Device Path Protocol [For21, p. 9.2] need to reflect what is normally expected **[TODO dump a windows boot entry]** pass on load options from boot entry override device path with the one from our bootkit, otherwise it will point to Windows Boot Manager backup location

5.1.2 Rootkit

general approach doesn't differ much

main executable is now dxe driver instead of an application dxe drivers are now automatically loaded from the DXE dispatcher

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 [@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 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 [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 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 [For20, Vol 3, 2.1.4.1.4] 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 [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.

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 [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]**

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

[For21, p. 32.3]

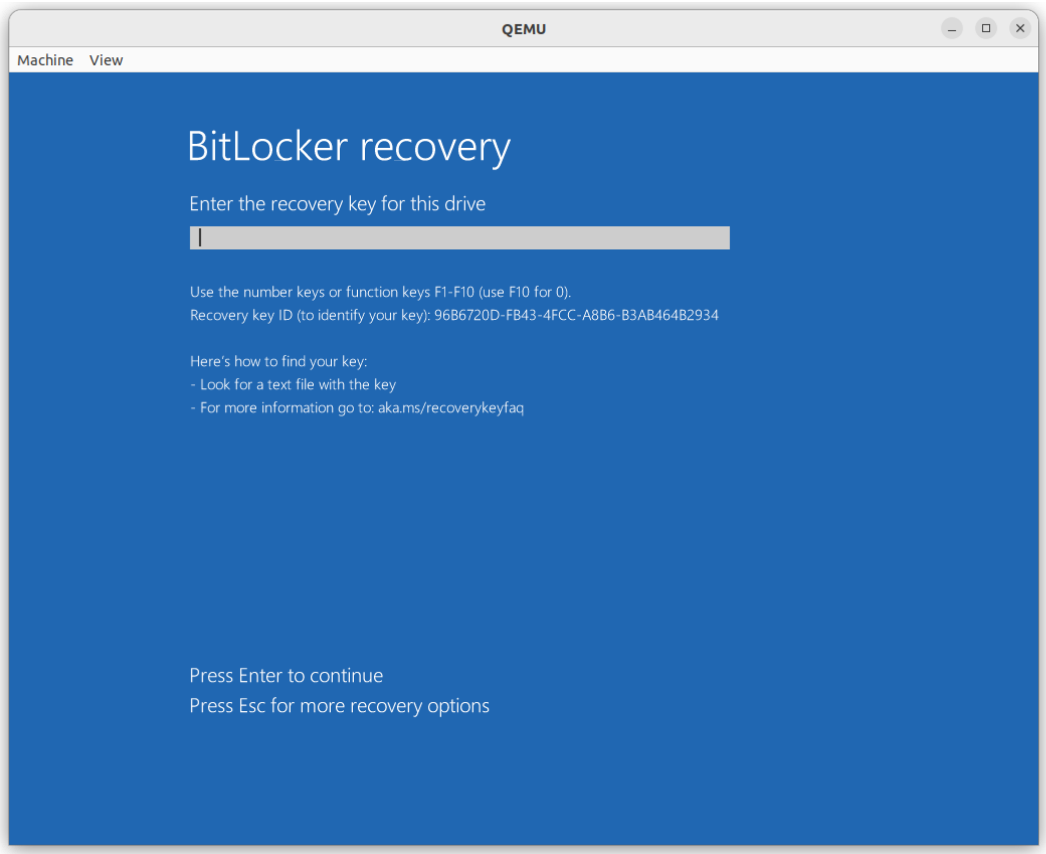
policy defined [For21, 32.5.3.3 Authorization Process]

5.3 Bitlocker

For our third attack we will enable BitLocker, this prevents us from trivially accessing any data before Windows has successfully booted. **[TODO rewrite for rootkit/-bookit split]** 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. [@Micj]

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 Platform Configuration Register (PCR) values differing from what was initially used to seal the Volume Master Key (VMK), leaving the Windows bootloader unable to retrieve an unencrypted VMK from the TPM and as a result unable to decrypt the Windows installation [AS21, p. 12.].

[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 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 prompt also only aids in locating the recovery key [@Mick]. **[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 endeavours 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 EDK II 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 boot 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 the interactive boot menu.

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 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 toUEFI

bitlocker encrypts block-wise

[TODO explain block devices] [TODO explain file system independent abstraction better] UEFI protocol stack **[TODO diagram of block io, disk io, simple file system 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 [@Micl] caveat pin? look into this persistence when part of root of trust fresh install / tpm update values hook TCG2 Protocol [Gro16, p. 6.7.3] TPM communication receive bitlocker VMK key and send to dislocker [@lib] [@And]

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

6 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 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

you can change recovery message and URL in BCD hive

not yet done: prevent firmware update

boottime vs runtime rootkit

6.1 Rootkit classification

statisken zu bilocker 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

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

7 Conclusion

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

7.1 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

7.2 Future Work

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

Bibliography

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

[For20] U. Forum, *Uefi platform initialization (pi) specification, version 1.7 errata a*, 2020 (cit. on pp. 5, 6, 28, 29).

[For21] U. Forum, *Uefi specification, version 2.9*, 2021 (cit. on pp. 6, 9, 10, 22–25, 27–29, 31, 33, 34).

[AS21] M. E. R. Andrea Allievi Alex Ionescu and D. A. Solomon, *Windows Internals, Part 2* (Developer Reference), 7th ed. Pearson Education, Inc., Sep. 2021 (cit. on pp. 10, 13, 14, 25, 26, 32).

[Gro08] T. C. Group, *Trusted platform module (tpm) summary*, 2008 (cit. on p. 12).

[Gro19] T. C. Group, *Trusted platform module library, part 1: Architecture, level 00 revision 01.59*, 2019 (cit. on p. 12).

[Gro16] T. C. Group, *Tcg efi protocol specification for tpm 2.0 version 1.0 revision 0.13*, 2016 (cit. on pp. 12, 35).

[PS17] M. E. R. Pavel Yosifovich Alex Ionescu and D. A. Solomon, *Windows Internals, Part 1*, 7th ed. Microsoft Press, May 2017 (cit. on p. 13).

[MI12] D. A. S. Mark E. Russinovich and A. Ionescu, *Windows Internals, Part 2*, 6th ed. Microsoft Press, Sep. 2012 (cit. on p. 14).

[For16] U. Forum, *Uefi shell specification, revision 2.2*, 2016 (cit. on pp. 22, 23).

Webpages

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

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

[@For] U. Forum. “Uefi specifications overview.” (), [Online]. Available: `https://uefi.org/specsandtesttools` (cit. on p. 3).

[@Tiaa] TianoCore. “Edkii build specification.” (), [Online]. Available: `https://edk2-docs.gitbook.io/edk-ii-build-specification/` (cit. on p. 8).

[@Tiab] TianoCore. "Edk ii project github repository." (), [Online]. Available: https://github.com/tianocore/edk2 (cit. on p. 11).

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

[@Mica] Microsoft. "Microsoft windows authenticode portable executable signature format, version 1.0." (), [Online]. Available: https://edk2-docs.gitbook.io/understanding-the-uefi-secure-boot-chain/ (cit. on p. 11).

[@Tiad] TianoCore. "Trusted boot chain." (), [Online]. Available: https://tianocore-docs.github.io/edk2-TrustedBootChain/release-1.00/ (cit. on p. 12).

[@Micb] Microsoft. "How windows uses the trusted platform module." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/how-windows-uses-the-tpm (cit. on p. 14).

[@Micc] Microsoft. "Bitlocker." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview (cit. on p. 14).

[@Micd] Microsoft. "Overview of bitlocker device encryption in windows." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-device-encryption-overview-windows-10 (cit. on pp. 14, 15).

[@Mice] Microsoft. "Bitlocker countermeasures." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-countermeasures (cit. on p. 15).

[@tux] tuxera. "Ntfs-3g." (), [Online]. Available: https://github.com/tuxera/ntfs-3g (cit. on p. 22).

[@pba] pbatard. "Ntfs-3g." (), [Online]. Available: https://github.com/pbatard/ntfs-3g (cit. on p. 22).

[@Micf] Microsoft. "Task triggers." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/taskschd/task-triggers (cit. on p. 25).

[@Micg] Microsoft. "Task actions." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/taskschd/task-actions (cit. on p. 25).

[@Mich] Microsoft. "Task security contexts." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/taskschd/security-contexts-for-running-tasks (cit. on p. 25).

[@Mici] Microsoft. "Rundll32." (), [Online]. Available: https://learn.microsoft.com/en-us/windows-server/administration/windows-commands/rundll32 (cit. on p. 26).

[@Nor] P. Nordahl-Hagen. "Chntpw." (), [Online]. Available: http://pogostick.net/~pnh/ntpasswd/ (cit. on p. 27).

[@Lon] LongSoft. "Uefitool." (), [Online]. Available: https://github.com/LongSoft/UEFITool (cit. on p. 28).

[@Micj] Microsoft. "Bitlocker basic deployment." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-basic-deployment (cit. on p. 31).

[@Mick] Microsoft. "Where to look for your bitlocker recovery key." (), [Online]. Available: https://aka.ms/recoverykeyfaq (cit. on p. 33).

[@Aor] Aorimn. "Dislocker." (), [Online]. Available: https://github.com/Aorimn/dislocker (cit. on p. 34).

[@Use] User71491. "Exitbootservices hooking." (), [Online]. Available: https://wikileaks.org/ciav7p1/cms/page_36896783.html (cit. on p. 34).

[@Micl] Microsoft. "Bitlocker: Use bitlocker drive encryption tools to manage bitlocker." (), [Online]. Available: https://learn.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-use-bitlocker-drive-encryption-tools-to-manage-bitlocker (cit. on p. 34).

[@lib] libde. "Bitlocker drive encryption (bde) format specification." (), [Online]. Available: https://github.com/libyal/libbde/blob/main/documentation/BitLocker%20Drive%20Encryption%20(BDE)%20format.asciidoc (cit. on p. 35).

[@And] D. Andzakovic. "Extracting bitlocker keys from a tpm." (), [Online]. Available: https://pulsesecurity.co.nz/articles/TPM-sniffing (cit. on p. 35).

List of Figures

List of Tables

List of Listings

Appendix A

B Acronyms

AL Afterlife

API Application Programming Interface

BCD Boot Configuration Data

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 File System

FV Firmware Volume

FVEK Full Volume Encryption Key

GUID Globally Unique Identifier

HOB Hand-off Block

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