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 threats is a bootkit, 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

[Micc; Micb; Ver; kas] persistence goals 1/3 - 1/2 pages

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 Security Theory

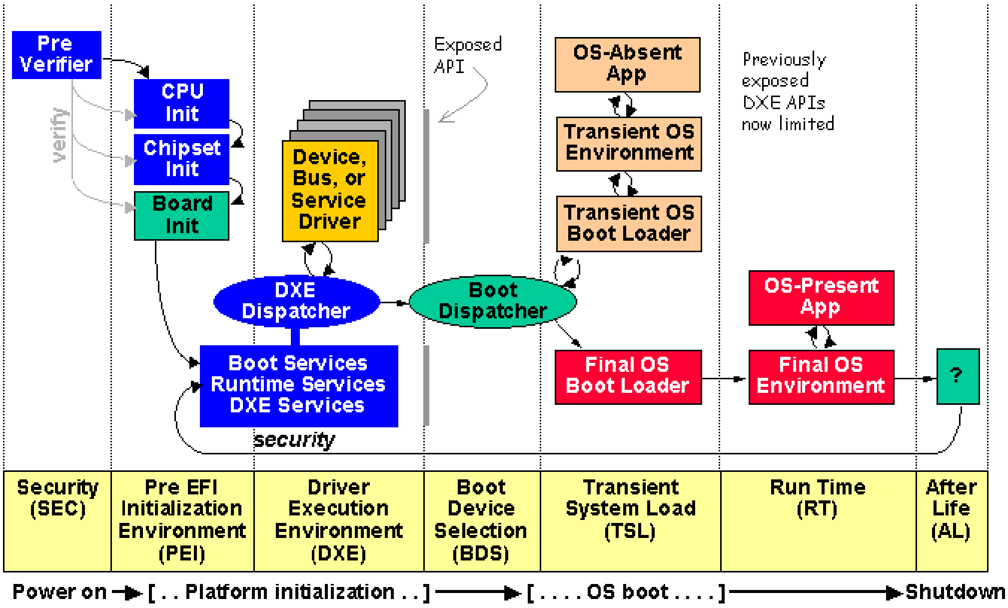
2.2 UEFI/PI

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

2.2.1 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 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 ExitBootServices()

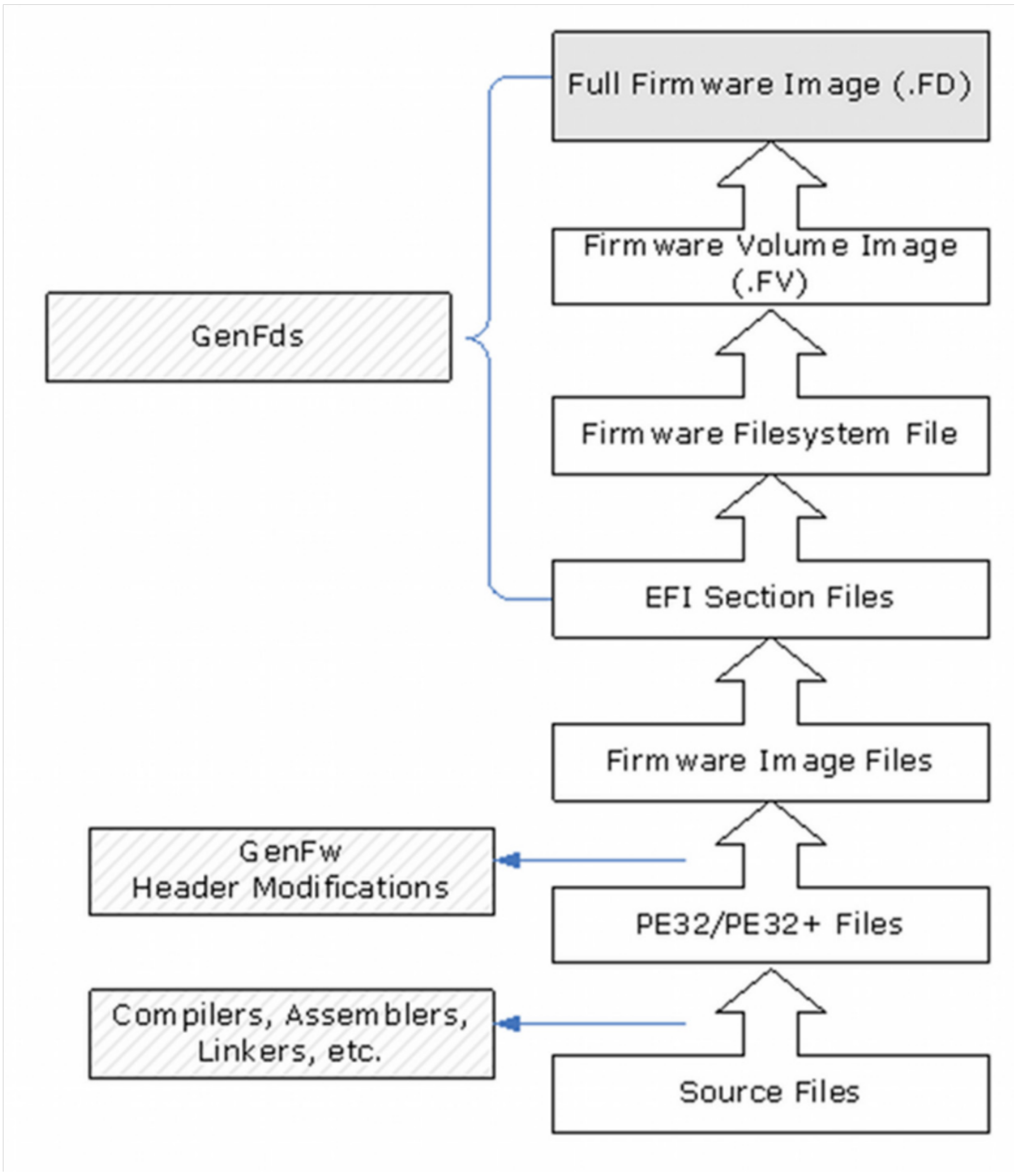
6. Runtime (RT)

   runtime services/driver

7. Afterlife (AL)

   hibernation sleep

2.2.2 UEFI/PI Firmware Images



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]

2.2.3 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

2.2.4 Firmware Core

2.2.5 Systemtable

2.2.6 Handles

2.2.7 Protocols

2.2.8 Boottime Services

2.2.9 Runtime Services

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

protocols identified with guids

2.2.10 edk2

build system

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 []

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

authenticated variables maybe

Signed Capsule Update

DXE SMM Ready to Lock Vol4

Capsule Architectural Protocol

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

flash device security

TPM measurements

[Tiab]

2.3 Windows

2.3.1 User Access Control (UAC)

2.3.2 Signing

2.3.3 Bitlocker

3 Related Work

scholar ranking

4 Attacks

Our different attacks face three escalating levels of security mechanisms. The first is with Secure Boot and Bitlocker disabled, the second is just Secure Boot enabled and the third is both Secure Boot and Bitlocker enabled with the focus of the study on Bitlocker. All attacks share the requirement of being able to 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.

4.1 Test Setup

[TODO describe test setup]

4.2 Neither Secure Boot nor Bitlocker

first aim harddrive access windows uses NTFS search for UEFI NTFS driver with write access [pba] open source fork of ntfs-3g for UEFI

compile spits out .efi file

try in EFI shell we are trying in qemu where the shell is available in boot options real hardware might require the uefi shell application .efi from the user via usb stick explain shell screen showing mapping, also available with map command consistent device mapping, comparable to partition names in windows [For16] blk and fs load NtfsDxe.efi drivers map -r "reset all the default mappings in a system this option is useful if the system configuration has changed since the last boot" new mapping and fs ontop of old blk navigate to Windows folder means read works try creating folder to test write debug why it failed find out its from windows having hibernation enabled by default change source code to not fallback to read only when encountering hibernation file

try to package in firmware image instead of loading from external medium retrieve the image for qemu we can build it ourselves for hardware use spi clamp to dump image

we can open image with UEFITool an editor for firmware images conforming to the UEFI PI spec [Lon]

in UEFITool search for DXE Volume remove previous NTFS driver if present, for full control, might be read only etc in UEFITool search for string since files are part of file sections we cant drop in the .efi compile dxe driver within ovmf generate unused volume to receive .ffs file with version, depex, user interface and pe section add in NTFS driver .ffs file qemu just use path to modified image on hardware use write access with clamp

now upon opening uefi shell we instantly see the filesystem

try to use ntfs-driver in code this is our first rootkit iteration payload will also be DXE driver access to all drivers etc package payload in firmware image steps read payload into RAM search for windows installation write payload

pack executable binary as uefi module edk2 produces freeform image with one raw section

iterate over FirmwareVolume2 protocol instances boottime services offer a function LocateHandleBuffer which returns all handles having a given protocol attached to them iterate over all handles and open the protocol call ReadSection with payload guid, to read raw section check size match was necessary on hardware when compiling payload a post build script generates a header for the rootkit dxe containing the size on disk for the payload

iterate over all FirmwareVolume2 Protocols read raw section of payload file GUID into memory

iterate over all SimpleFileSystem Protocols open volume open file to write to since we write into windows folder this also checks if the volume has a windows installation

but no automatic execution nor elevated privileges dll proxying dll hijacking registry editing

Task Scheduler defined in xml cached in registry edit with start cmd.exe and trigger manually whoami

chntpw and reged port to uefi edit Task in machine under Control maybe look if just adding a key would have also worked export target registry key modify so that registry key can differ and found via matching values import and override registry key on target machine payload whoami localsystem

4.3 Secure Boot

how does one enable it mostly comes with default keys OEM expectation: not to boot observation: no difference secure boot default policy snippet option roms and bootloader instead relies on Signed Capsule Updates assumes integrity

4.4 Secure Boot and Bitlocker

assumptions: secure boot or not bitlocker enabled with TPM auto decryption

observation: boot execution differs from executing rootkit tpm values different bitlocker auto decryption fails recovery key prompt

what is the reaction of the average user (ask admin for recovery password) type in recovery password alternative would be to remove drive and insert into safe device

prompt is done by the OS Loader ergo still during transient system load phase required to use protocol services therefor uses uefi services for IO such as SimpleTextInputEx Protocol go over the two different input protocols find out which one is used

explain more in depth how protocols are returned to the end user one instance per controller/handle

explain basic hooking explain how we retain information of the hook in question map protocol pointer to hook information keylog recovery key key input advancment is weird and makes tracking tricky 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 mount encrypted drive with decryption mean read and write access dual boot in vm enter password and it works port to uefi bitlocker encrypts block-wise uefi protocol stack hook block io again hook data mapping dislocker validate block solves recovery key advancement issue

hook ExitBootServices enable hook write payload import registry key disable hook

next boot would require to input tpm values 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 receive bitlocker vmk key and send to dislocker

5 Discussion

attack assumption reflected to real world aplicability

social engineering aspekt

driver vorhanden und was mitbringen, debloating

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

hardware validated boot

inaccessible spi flash

tpm + pin detectability

googeln wie legitime recovery key prompt reaktion aussieht

enterprise policy recovery key einschraenkbar?

enterprise policy on recovery key loss

5.2.1 User awareness

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

6 Conclusion

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

Bibliography

[] (Cit. on p. 10).

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

[For16] UEFI Forum. *UEFI Shell Specification, Revision 2.2*. 2016 (cit. on p. 15).

[For21] UEFI Forum. *UEFI Specification, Version 2.9*. 2021 (cit. on p. 6).

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

[kas] kaspersky. *What is Rootkit - Definition and Explanation* (cit. on p. 1).

[Lon] LongSoft. *UEFITool* (cit. on p. 16).

[Mica] Microsoft. *Microsoft Windows Authenticode Portable Executable Signature Format, Version 1.0* (cit. on p. 10).

[Micb] Microsoft. *Rootkits* (cit. on p. 1).

[Micc] Microsoft. *Secure the Windows boot process* (cit. on p. 1).

[pba] pbatard. *ntfs-3g* (cit. on p. 15).

[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).

[Ver] Veracode. *Rootkit: What is a Rootkit?* (Cit. on p. 1).

[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

B

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

UEFI Unified Extensible Firmware Interface