# Too Young to be Secure: Analysis of UEFI Threats and Vulnerabilities

Vladimir Bashun, Anton Sergeev, Victor Minchenkov, Alexandr Yakovlev
St. Petersburg State University of Aerospace Instrumentation
St. Petersburg, Russia
{bashun, slaros, Victor, yakovlev.a}@vu.spb.ru

***Abstract*—Unified Extensible Firmware Interface (UEFI) is a software interface between an operating system and platform firmware designed to replace a traditional BIOS. In general, UEFI has many technical advantages over BIOS (pre-OS environment, boot and run-time services, CPU-independent drivers etc.) including also powerful security mechanisms (e.g. secure boot, update, etc.). They are aimed to provide platform integrity, be root of trust of security architecture, control all stages of boot process until it pass control to authenticated OS kernel. From the other side UEFI technology is the focus of many new potential threats and exploits and presents new vulnerabilities that must be managed. The main goal of this research is to provide analysis of the UEFI security issues, find the point and source of the security problems and classify them. The paper describes the architectural and implementation troubles of UEFI which lead to threats, vulnerabilities and attacks. It also includes extensive review of the previous research activities in this area and the results of our own experiments. As the result of the work some recommendation about how to make this young technology more safe and secure are provided.**

***Keywords–UEFI Secure Boot, Boot firmware, Bootkit, Rootkit.***

## I. Introduction

Secure Boot mechanism aimed to provide *platform integrity, be root of trust of security architecture, control all stages of boot process* until it pass control to authenticated OS kernel.

The task of hardware initialization process and transition control to the operating system and system power on is accomplished by specialized system firmware, usually referred to as Basic Input-Output system (BIOS). The executable code (usually called boot firmware) is stored in non-volatile memory of computer.

BIOS place in architecture and in procedure of system start up is very important, because it is a root of trust for platform. BIOS accomplish full control of platform from computer power on until it passes control to operating system kernel. This special role of boot firmware makes it potentially attractive target for attack. Unauthorized BIOS firmware update, injection of malicious code to BIOS can be used to compromise and take control over components loaded later in the boot process, like operating system or hypervisor. And such infection shall be persistent, since malware written into boot firmware can be used to re-infect operating system even if it was re-installed. Since boot firmware works before operating system kernel even loaded to memory or takes control, special security software like antivirus can't detect the invasion.

Two factors raise attention to boot firmware:

*1) Change of technologies in boot firmware*. An evolutionary change of boot firmware technologies occurs right now. A new Unified Extensible Firmware Interface (UEFI) takes place of old legacy BIOS that prevailed without significant changes for over 30 years.

*2) In increased number of bootkit and rootkit attacks*. Many companies that develop antivirus software expect that such attacks shall be main trend in nearest future.

New standard brings both new possibilities and new threats. With deployment of new unified interface based on well-documented specification and common API and protocols gives malware developers an opportunity to infect a wide range of new systems. Rich possibilities of new interface give them additional ways to attack the platform. At the same time, these possibilities may be used by UEFI developers to protect system from malware.

The rest of the paper organized as follows. In section II a brief review of UEFI interface, its architecture and key components, is presented. In section III, security issues of UEFI technologies are analyzed, potential threat, attacks on UEFI and recommendations for protection of UEFI BIOS.

## II. UEFI Technology

### A. *Unified Extensible Firmware Interface*

UEFI – unified extensible firmware interface – is an interface between the operating system and firmware that control hardware. Its main role in the boot process is the following [1]:

1) *Execute core root of trust* (small core block of firmware that executes first and is capable of verifying the integrity of other firmware components).

2) *Initialize and test low-level hardware*.
3) *Load and Execute Additional Firmware Modules.*
4) *Select boot device, call Boot Loader and pass control to operating system kernel.*

In addition, UEFI gives possibility to develop extensions (drivers and applications).

UEFI is developed by the UEFI Forum [2]. The forum developed several specifications, the most interesting for current research are the UEFI Specification [3] (the specification of standard itself, defines basic interfaces, prtocols, data structures, etc.) and PI Specification [4] (defines the boot process, UEFI boot stages and interfaces between them). The main stages of UEFI boot process and key components are discussed in the next chapter.

## *B. Platform Initialization*

The UEFI boot process pass through several stages, each stage has its own role and potentially may pose certain security risks:

- *SEC*. When an EFI system is powered on, the SEC (Security) phase of EFI is the first code that is executed within EFI. This phase serves as a root of trust for the system and handles platform reset events, among other things.
- *PEI*. The PEI (*Pre-EFI Initialization*) phase (platform-specific), which is responsible for initializing the CPU and main memory. It prepares platform for next stages, which are written on C language. It locates and executes PEIM modules.
- *DXE*. The DXE (*Driver eXecution Environment*) phase is where the majority of the system initialisation takes place. First, the DXE core produces a set of Boot and Runtime Services. Boot Services provide drivers, applications and Boot Loaders that run within the EFI environment with a number of services such as allocating memory and loading executable images. Boot services are destroyed before pass control to OS kernel. Runtime Services provide specific services to code running within the EFI environment or within the OS kernel once it has taken control of the system. Once these services have been established, the DXE dispatcher discovers and executes DXE drivers from the firmware volume, expansion ROMs on devices connected to the PCIe bus, and connected disks.
- *BDS*. The Boot Device Selection phase is responsible for discovering the possible boot devices, selecting one to boot from, loading the Boot Loader and executing it.

## *C. UEFI Security Mechanisms*

The boot firmware is potentially attractive target for attack and should face many threats. In the UEFI specification (version 2.3.1) special attention is given to the security issues. The main component of UEFI security is Secure Boot module (chapter 27 of specification).

The key components of UEFI Secure boot are:

1) *UEFI Image Signing.* The main aspect of UEFI Secure Boot – ability to use cryptography (digital signatures and hashes) for local authentication and verification of UEFI images (drivers, applications). Specification describes platform public key infrastructure based on PKI. UEFI keys are stored in Secure Boot. The following keys are define by specification (Fig. 1):

- *Platform Key (PK). This is a root key, created for this exact platform. PK Key Owner (usually hardware manufacturer) can modify all other keys.*
- *Key Exchange Key (KEK) – the owner of this key (usually OS vendor) can update db/dbx keys.*
- *db – all allowed certificates and hashes are stored here (white list).*
- *dbx – all forbidden certificates and hashes are stored here (black list).*

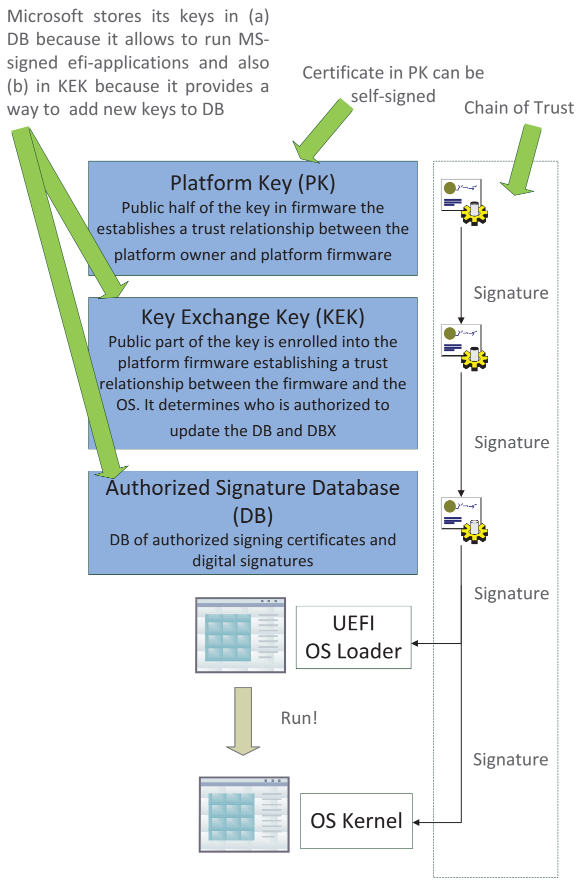

Microsoft stores its keys in (a) DB because it allows to run MS-signed efi-applications and also (b) in KEK because it provides a way to add new keys to DB
Certificate in PK can be self-signed
Chain of Trust
Platform Key (PK)
Public half of the key in firmware the establishes a trust relationship between the platform owner and platform firmware
Key Exchange Key (KEK)
Public part of the key is enrolled into the platform firmware establishing a trust relationship between the firmware and the OS. It determines who is authorized to update the DB and DBX
Authorized Signature Database (DB)
DB of authorized signing certificates and digital signatures
Signature
Signature
Signature
Signature
UEFI OS Loader
Run!
OS Kernel

Fig. 1. Hierarchy of UEFI Trusted Key Databases and "ideal" Chain of Trust (a sequence of verification of certificates and signatures) for a secure OS boot.

2) *UEFI Authenticated Variable*. Time-Based Authenticated Write Access used to update Authenticated variables like keys db. Updated variables must be signed. PK cert verifies PK/KEK update. KEK verifies db/dbx update. Certdb verifies general authenticated EFI variable updates.

3) *UEFI Secure Update.* This component is used to authenticate firmware update images using digital signature verification. In order to enable the Secure Boot mode on a platform one should do the following:

- *Enroll Platform Key (PK)* and other keys, if needed.
- *Set value of SetupMode variable to USER_MODE* (stored in NVRAM).
- *Adjust SecureBootEnable variable* (stored in NVRAM).

UEFI variable can be updated using the UEFI runtime service SetVariable. UEFI provides write-protected service for Authenticated variables, based on asymmetric key technology. In order to update variable value, it should be signed with appropriate key. *Disabling of Secure Boot, as well as deploying new PK should be done only in users Physical Presence.*

Verifying UEFI image signature (or hash) is a task of a library called DxeImageVerificationLib. The same library defines image verification policies applied to different types of images (like IMAGE_FROM_FV (ALWAYS_EXECUTE), IMAGE_FROM_FIXED_MEDIA, IMAGE_FROM_REMOVABLE_MEDIA, IMAGE_FROM_OPTION_ROM).

If UEFI image is not signed by a trusted key, and it's hash is not found in db, the image shall not be loaded. If Boot Loader signature verification fails, the operating system shall not load.

What elements are verified depends on verification policies in specific UEFI implementation on a platform. By default, signature of drivers and components stored in firmware is not verified (policy ALWAYS_EXECUTE for IMAGE_FROM_FV). There is no signature verification on first boot stages (SEC + PEI). Thus, such modules must be stored in write-protected non-volatile memory.

In general, Secure Boot mechanism aimed to provide *platform integrity, be root of trust of security architecture, control all stages of boot process* until it pass control to authenticated OS kernel.

## III. UEFI SECURITY

### *A. Common Notes*

Analyzing the UEFI Platform Initialization stages (PI), we observe that the following blocks may be a subject of attack and may theoretically pose a threat:

- Malware PEIM modules.
- Malware DXE and UEFI drivers and UEFI applications (on DXE stage).
- Malware Boot Loader (on BDS stage).

### *B. Attack Goals*

The primary attack goals are:

*1) Modification of UEFI firmware/injection of malware DXE drive.* Malicious adding or replacing DXE driver is needed for:

- First step leading to further injection of exploit to Boot Loader or kernel.
- Malware DXE driver may be used itself, for example, for gathering information (like password interception, etc.), sending it over network.

*2) Substitution of Boot loader/OS loader, patching OS kernel on pre-boot.* It is a classical boot kit attack. It can be used for:

- *Start infected operation system with exploits* (e.g., disabling OS security modules, privileges escalation, bypass authentication, etc.). It is possible since Boot Loader hold control before it gives control to OS kernel (see [5]);
- *Injection of hidden malware hypervisor* (Virtual Machine based rootkits, VMBRs, see [6])

The attack technique may be the following:

- *Exploiting vulnerabilities in Operating System* for attack on UEFI. One can use UEFI runtime services or firmware update mechanisms (e.g., for unauthorized firmware update, injection of security keys, modifying boot order, etc.). Another way is substitution of files in EFI GPT System Partition (ESP) (e.g., substitution of Boot Loader file).
- *An "evil maid" attack.* An attack during a short-time physical access to hardware during platform power-on

### *C. Where an attack can come from*

In order to persist in a system, malware (like DXE driver or Boot Loader) should be stored somewhere. UEFI-based malware can be stored in a number of locations:

*1) Hard disks (or SSD).* Malware can be stored anywhere on system disk. In particular, it can be EFI System Partition (ESP), where UEFI Boot Loaders are usually stored.

*2) SPI-flash with EFI firmware.*

*3) Option ROM.* Storing malware DXE drivers in option ROMs of expansion cards (like video or network adapters).

### *D. Secure Boot Modes*

As mentioned earlier, UEFI specification has special part called Secure Boot designed to protect system integrity. That's why all attacks on UEFI can be separated on two classes:

- Attacks on UEFI with Secure Boot OFF
- Attacking UEFI Secure Boot

Each class of attacks shall be examined separately. At first, let's investigate basic attack vectors on UEFI with disabled Secure Boot.

## IV. UEFI Security Analysis. Secure Boot OFF

Utilization of UEFI platforms with disabled Secure Boot mode provides attacker with a variety of possibilities. This is a fee for extendibility, well documented modular structure, detailed specification and availability of a full-featured development kit. Lack of authentication of loaded DXE drivers and UEFI applications before execution gives possibility to take control over system by simply substitution of standard DXE driver or Boot Loader file. The fact that standard OS loaders are stored in known directories on disk in pre-defined paths makes this task even easier.

Several successful attacks on UEFI with disabled Secure Boot were reported. For example, in [5] described an attack on Windows 8 that injected boot kit by replacing standard Windows Boot Loader by a malware Boot Loader by copying it to EFI\Microsoft\Boot directory.

The primary attack vectors on UEFI with disabled secure boot:

*1) Modification of UEFI firmware by unauthorized firmware update.* This can be done easily, since UEFI in this case does not ensure the authenticity of the firmware update image. One can use UEFI runtime service to schedule firmware update from Operating System.

*2) Add malware DXE driver (or patch existing DXE driver)*

- DXE driver can be stored on fixed drive or USB (no need to re-flash firmware).
- Modification of driver load order can results in load of malware driver on early boot stage.

*3) Injection of malware Boot Loader file* (usually stored in a known directory in FAT32 EFI System partition).

- One can substitute OS Boot Loader (e.g., bootmgrfw.efi for Microsoft Windows, see [5]). Since Boot Loader is stored in protected EFI System partition, in order to substitute bootmgrfw.efi loader for Microsoft Windows one need to start "file copy" with administrative privileges.
- Or substitute fallback Boot Loader (\EFI\BOOT\bootx64.efi) (see [7]). bootx64.efi serves as the fallback boot loader if none is specified in the firmware's flash storage.
- Add your own Boot Loader (.efi application) and modify boot order using UEFI runtime services.

*4) Patch ROMs in expansion cards (Option ROMs): network cards, video cards, storages, etc.*

- Such attacks were described in [8].
- In [9] an attack on Mac UEFI platform using re-flashing Option ROM was demonstrated.
- Also there also is such an exotic variant of attack, as re-sellingof malware-flashed hardware.

*5) Modify GUID Partition Table (GPT).*

When analyzing UEFI work and OS load with the secure boot turned off the following facts should be taken to consideration (in other words why users and administrators turn off the Secure Boot so often?):

*1) Not all operating systems and hypervisors support UEFI Secure Boot yet* (although this tends to improve).

*2) Various problems arise with UEFI Secure Boot and dual-boot systems.* The problems arise from the fact that user in most cases does not have access to PK flashed on factory, and injection of new PK/KEK is not trivial task yet for ordinary user.

*3) Secure Boot is optional and can be easily turned off by a user in UEFI BIOS settings.* A possibility to turn secure boot off is part of specification (as well as possibility to enroll new PK). Specifically, this is a requirement for all certified platforms with Windows 8. Abhorrence that some members of Linux community hold to Secure Boot mode, reckoning it a Microsoft-driven technology, and existing limitations for dual-boot systems, raise a possibility of turning Secure Boot off.

*4) Guest operating systems (in virtualized environments) do not directly interact with Secure Boot of host platform.* This means that host platform Secure Boot does not impact boot process of guest operating systems.

As a result of this section we can conclude that *UEFI with a disabled Secure Boot is exposed to various attacks*. Since UEFI provides rich possibilities for developers, UEFI platform with disabled security module is even more vulnerable than legacy BIOS.

## V. UEFI Security Analysis. Attacking Secure Boot.

With Secure Boot enabled, all UEFI images are authenticated and their digital signatures are verified before execution. Thus, enabling of Secure Boot should ensure that no malware DXE driver or application shall be loaded. The straight attack with substitution of Boot Loader file with a malware one shall fail, since its digital signature shall be invalid.

The main goal of attack on secure boot is to avoid verification somehow and execute unsigned code. If this goal is achieved, all attacks on secure boot from the previous section can be performed. There is a number of ways to achieve this goal:

*1) Disable Secure Boot (Illegally turn it off)*

- Delete or corrupt PK EFI variable in NVRAM (см. [10]).
- Change state of SetupMode and SecureBootEnable variables, stored in NVRAM.

*2) Violate the integrity of Secure Boot*

- For example, patch DxeImageVerificationLib library to change the verification policies.

*3) Execute code, signed by invalid keys*

- Add invalid certificate or hash to a db variable, stored in NVRAM.
- Now all images signed by that key shall pass verification.

*4) Execute code without signature verification*

- Inject malware code to platform firmware or Option ROM, or
- Execute malware code in compatibility with Legacy BIOS mode.

Potential vulnerabilities, possible threats and attack vectors on systems with the enabled UEFI Secure Boot shall be examined further in this section (Fig. 2). Vulnerabilities may be both in UEFI architecture, and in the UEFI BIOS implementation on specific platforms.

### *A. Threats from Compatibility Support Module*

Compatibility Support Module (CSM) is a module allowing legacy operating systems and some option ROMs that do not support UEFI to still be used. It emulates BIOS environment for such systems. And this module can be used to avoud UEFI Secure Boot verification mechanisms. When Secure Boot mode is enabled, the Compatibility Support Module must be unavailable. This is, by the way, part of the Windows Hardware Certification Requirements (section System.Fundamentals.Firmware.UEFISecureBoot in [11]): “Mandatory. When Secure Boot is Enabled, Compatibility Support Modules (CSM) must NOT be loaded.”

A proper implementation of this requirement is up to UEFI platform manufacturers.

### *B. Threats in UEFI Firmware Update Mechanism*

This is one of the most dangerous attack vectors. Since there is a necessity to update UEFI firmware, such mechanism is provided. Once malware is stored in firmware, it may infect operating system, block further firmware updated, and it is not verified by UEFI Secure Boot.

An increasing number of attacks on BIOS (including UEFI) encouraged National institution of Standards and Technology (NIST) to issue a document NIST 800-147 BIOS Protection Guidelines [1]. This document contain general recommendations for protection of BIOS (including UEFI BIOS), focusing on secure firmware update process.

The primary sources of threats (according to [1]):

- User-initiated installation of a malicious firmware update
- Leveraging weak BIOS security controls or exploiting vulnerabilities in the system BIOS itself to reflash or modify the system BIOS.
- Launching an organization-wide attack through network-based system management tools. For example: infection of organization-maintained update server to push a malicious system BIOS to computer systems across the organization.
- Leveraging “reset to factory settings” mechanism to rollback to an authentic but vulnerable system BIOS.

NIST 800-147 document provides guidelines for preventing the unauthorized modification of BIOS firmware. These recommendations are quite general, but following them allows reducing a risk of platform infection as a result of threats discussed above. Section 3.1 concentrates on recommendations for secure BIOS update process, including:

- The authenticated BIOS update mechanism, providing authentication and integrity of BIOS update images;
- Non-bypassability features, to ensure that there are no mechanisms that allow the system processor or any other system component to bypass the authenticated update mechanism.

Specifically, a secure BIOS update process must provide:

- Employment of digital signatures to ensure the authenticity of the BIOS update image
- Protection of relevant regions of the system flash memory containing the system BIOS prior to executing firmware or software that can be modified without using an authenticated update mechanism. Protections should be enforced by hardware.

UEFI provides a number of protocols and mechanisms to follow these recommendations (signed update images, protection from “replay BIOS update” or “reset to old firmware” attacks, etc.). UEFI protects firmware from write on early stages of boot, and verifies digital signature of update image before it is applied. According to UEFI specification, in Secure Boot mode only signed updates are allowed (Signed UEFI Capsule based update). In Windows Hardware Certification Requirements (see. [11] section System.Fundamentals.Firmware.UEFISecureBoot, paragraph 8) all those mechanisms, as well as flash write-lock on early stage of boot, are stated mandatory. It also recommends platform manufacturers writing BIOS code adhere to the NIST Guidelines.

Yet it’s up to platform manufacturer to correctly implement these recommendations and specifications. And bugs in implementation may be the chief reasons of vulnerabilities.

In [12] described a proof-of-concept attack on one UEFI implementation that use buffer overflow bug to bypass secure update process. The unsigned firmware update was accomplished before the write-lock was enabled. In [10] described an attack on platform with UEFI firmware that was directly writeable in SPI flash (write protection was enabled but not locked).

Thus, in spite of well-considered security mechanisms, the following threat to firmware update process remains:

- Bugs in secure firmware update implementation
- Attacks utilizing direct write to SPI-flash before it is locked (it is enough, for example, to patch DxeImageVerificationLib in NVRAM, and defense system is broken)
- Bugs in image signature verification procedures

### *C. Threats in Trusted Certificate Base Update Process*

Trusted certificates stored on platform need to be updated. This process should be secure; otherwise violator shall be able to deploy his own certificates. The Secure Boot signature databases (PK, KEK, db, dbx) are stored in NVRAM. Two ways of updating those databases are provided:

- In UEFI Custom Mode, mechanisms for a physically present user to modify the contents of the Secure Boot signature databases and the PK.
- Using UEFI-runtime service (Set Variable), update Authenticated Variable. New keys must be signed by one of legal KEK keys.

The following potential threats to Secure Boot signature databases management should be considered:

- Bugs in implementation of Authenticated Write Access procedure, which may lead to unauthorized Authenticated variable update (attack on SetVariable runtime service), and thus inject illegal certificate.
- Attacks utilizing direct write to SPI-flash before it is locked. Malware can write necessary keys directly to NVRAM, or it may corrupt PK to disable Secure Boot. Such attacks are presented in [10].
- Bugs in implementation of Physical Presence Interface.

Such errors may result in vulnerability that shall allow violator to save his certificate or hash in Secure Boot signature databases.

### *D. Hash Injection into the Secure Boot Signature Database*

In this case all is similar to the previous one (*section C*). Besides there are several EFI-administration utilities dedicated especially to get the full control over the platform. They very useful for Linux administrators and developers who need to rebuild the Linux kernel. The hash of the new kernel differs to the hash of the standard Linux kernel. So there is a real need in tools which can add a new trusted hash to the Secure Boot Signature Database. Therefore a chain of trust (chain of verification) between the start point and kernel verification will be broken. Chain of trust here is a list of digital certificates or hashes that are related to each other: every object (efi-application in boot process) is sighed/verified by the predecessor and signs/verifies the subsequent object in the list. Obviously all these utilities could be used for attack purposes.

The main idea of the “hash injection” attack is to embed hash of the unsigned efi-applications (infected kernel, efi-malware etc.) into Secure Boot Signature Database and get an opportunity to run the unsigned applications it in a Secure Boot.

For instance, the Linux Foundation loader (so-called pre-EFI Loader) allows adding hashes into Secure Boot Signature Database (after user’s confirmation). In this case the chain of trust is broken off at the last element (unsigned loader). It’s clear that an unsigned loader can run any application and UEFI secure mechanisms will not stop this. If the Linux Foundation Loader is signed by Microsoft CA UEFI Key (this was announced by its developers) then the operation system boot process is the following.

- UEFI Secure Boot verifies a key of the Linux Foundation loader (success)
- Linux Foundation loader can start ANY efi-application after user confirmation (success, a gap in the chain of trust). Practically it could be OS loader, e.g. gummiboot.efi. User sees a request whether he trusts to an efi-application or no. If he says YES and confirms an operation then hash of

this application (gummiboot.efi in our example) is saved as a trusted one in the UEFI Secure Boot DB. And at the next system boot user will not be asked about this. Gummiboot after that can run any other software, even malicious one.

Hence there is a considerable vulnerability is the Secure Boot: usage of Linux Foundation Loader means broken chain of trust. If a user uses Linux Foundation Loader to adds hash of an unsigned loader (e.g. gummiboot.efi) to the Secure Boot Signature Database (even only once!) then an unsigned loader can start any application without any verification. Practically this is equivalent to turning Secure Boot Off (in the UEFI BIOS settings). Result is the same: no boot security at all.

Linux Foundation loader as well as shim are one of the alternative Linux loaders, developed to support Secure Boot. Currently is not included (end of 2013) to the main Linux distributions. The main difference to the shim loader is in the loader verification method. Linux Foundation loader checks loader's hash in the trusted DB, while shim verifies the signature.

### *E. Threats to Machine Owner Key Technology*

For some systems (e.g. Fedora) it is supposed to to use Machine Owner Key (MOK) Technology. A Machine Owner Key (MOK) is a type of key that a user generates himself (using e.g. OpenSSL) and uses to sign an EFI binary. The key idea of a MOK is to give users the ability to run locally-compiled kernels, boot loaders not delivered by the distribution maintainer, and so on. Of course, the ability to use MOKs creates risk. A Linux expert says "if you're tricked into enrolling a MOK that was provided to you by a malware author, your computer will become vulnerable to attacks launched by that malware author" [14]. From the other side if the MOK key is stolen by an attacker then it could be used to sign and boot any efi-application. The problem is even worse because MOK–key can be added to the MOK DB from the level of operation system (e.g. etc/secureboot in Ubuntu) and UEFI Runtime-services.

Possible Vulnerabilities:

- Running a third-party unauthorized (incl. malicious) efi-software, signed by legal and trusted Machine Owner Key (hacker have to steel/copy this MOK-key beforehand and use it to sign a third-party SW).
- Hidden embedding of a third-party unauthorized Machine Owner Key into the system for subsequent running of any unauthorized efi-software on this infected platform.

### *F. Injection of Malware into Option ROM*

As it was already mentioned that the modifying/updating the efi-firmware of expansion cards (network cards, storages etc.) could be used for attack to UEFI. The specification allows not to verify signatures for drivers located at Option ROM. Therefore installing the hardware cards with infected efi-drivers or malicious modifying of the efi-firmware is base for attack even on the secure boot mode.

Security Threats:

- Weak control over the efi-firmware modification/update process and changing expansion cards.
- Turning off the mandatory verifying of the located at Option ROM efi-drivers' signatures (depends on default security policy of the platform developer) could also strongly affect the whole UEFI-based platform security.

### *G. Hidden Virtualization*

Since there are several described above possibilities to run Linux-based OS using a Microsoft-signed loader, the same methods could be used to run hidden hypervisor and virtualize (transform to the guest state) the active system.

### *H. SMM Threats*

System Management Mode is special operating mode of CPU which is not controlled by OS while special separate software (firmware or a hardware-assisted debugger) could be executed in high-privilege mode. This operating mode is supported by UEFI specification (Platform Initialization, vol 4 Secure Management Mode). Since some attacks to SMM mode using BIOS were demonstrated in the past [13] there is still some non-zero possibility of the similar attacks using the means of UEFI technology.

## VI. CONCLUSION

As can be seen, in UEFI a much greater attention is given to security issues. Special security mechanisms like UEFI Secure Boot are provided. At the same time, several moments need to be taken to account:

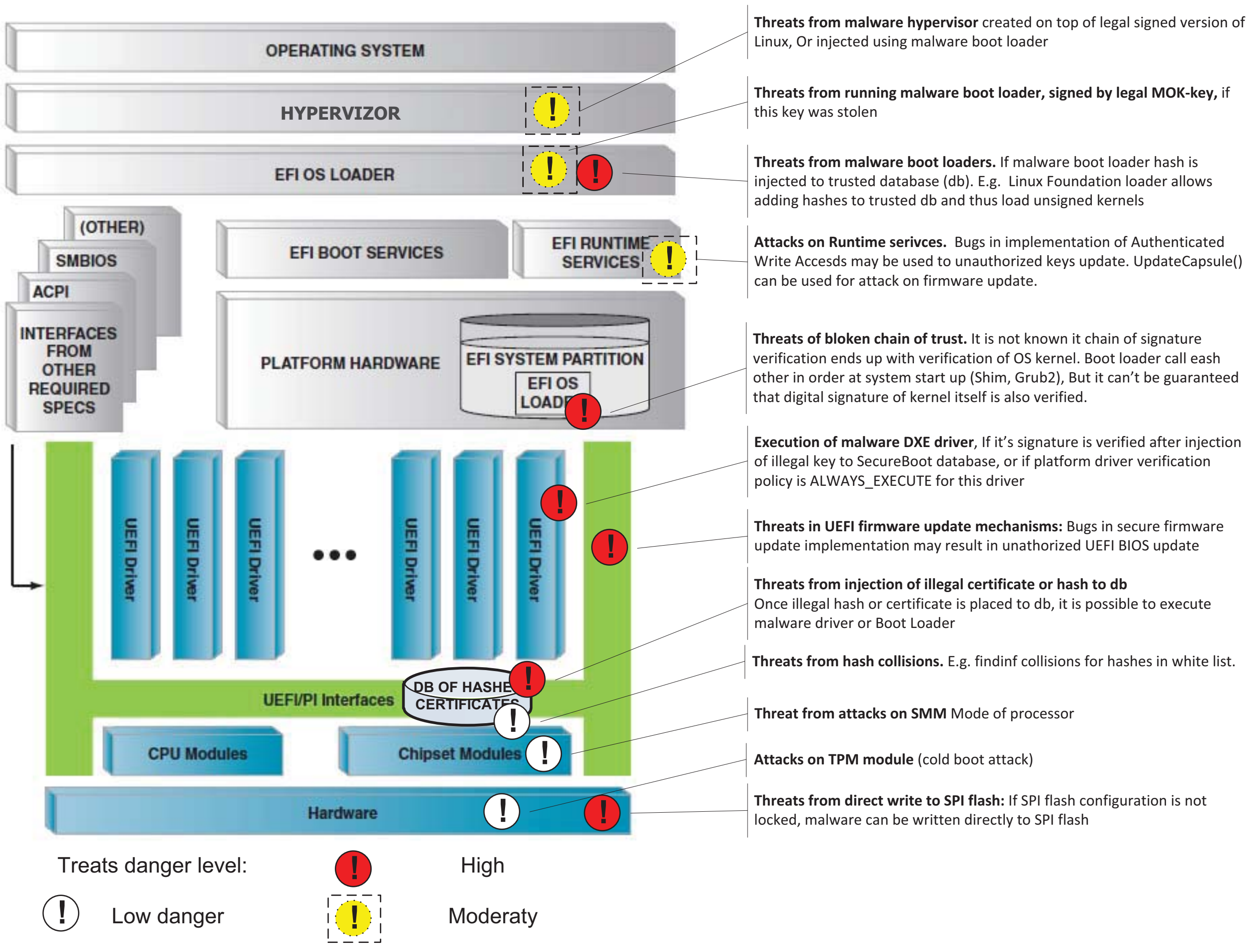
- *UEFI with disabled Secure Boot mode is totally insecure*. Reasoning of this thesis is provided above. Various successful attacks on UEFI, met in the literature during last year, are a strong evidence of this fact.
- *Even if Secure Boot is enabled, it does not guarantee the platform safety and integrity*. A variety of platform manufacturers and drivers developers, each of them implementing the specifications requirements on their own, may result in vulnerabilities in particular UEFI implementations.

Proceeding from the analysis of threats, we should note that to minimize the risk of malware injection, the one should respect following recommendations:

- *Secure Boot mode is mandatory!*
- *A secure firmware update mechanism must be provided*, in particular, UEFI BIOS update must follow all the recommendations from *NIST 800-147 BIOS Protection Guidelines*.
- *Only implementations of UEFI that strictly follow the specifications must be used.*

Specifically, system should:

- Allow only signed UEFI BIOS updates;
- Update the boot block (SEC/PEI code) securely;
- Disable Compatibility Support Modules and MBR boot loaders;
- Correctly implement UEFI modules;
- Verify signature of third-party Option ROMs;
- Correctly implement image verification policies;
- Protect PK/KEK/db/dbx storage in NVRAM from unauthorized modification;
- Protect Authenticated Variables, correctly implement Authenticated Write Access;
  - Protect UEFI firmware in SPI Flash from direct modification, lock the SPI controller configuration ;
  - Locking must not be controlled by any un-trusted programmable entities;
  - Once locked within CRTM code, it must not be un-lockable without going through a system reset;
- Correctly program and protect SPI Flash descriptor;
- Do not allow bypass of Secure Boot checks, user should not be able to bypass Secure Boot failures and boot anyway;
- Correctly implement physical presence checks;



**Threats from malware hypervisor** created on top of legal signed version of Linux, Or injected using malware boot loader

**Threats from running malware boot loader, signed by legal MOK-key,** if this key was stolen

**Threats from malware boot loaders.** If malware boot loader hash is injected to trusted database (db). E.g. Linux Foundation loader allows adding hashes to trusted db and thus load unsigned kernels

**Attacks on Runtime serivces.** Bugs in implementation of Authenticated Write Accesds may be used to unauthorized keys update. UpdateCapsule() can be used for attack on firmware update.

**Threats of bloken chain of trust.** It is not known it chain of signature verification ends up with verification of OS kernel. Boot loader call eash other in order at system start up (Shim, Grub2), But it can't be guaranteed that digital signature of kernel itself is also verified.

**Execution of malware DXE driver**, If it's signature is verified after injection of illegal key to SecureBoot database, or if platform driver verification policy is ALWAYS_EXECUTE for this driver

**Threats in UEFI firmware update mechanisms:** Bugs in secure firmware update implementation may result in unathorized UEFI BIOS update

**Threats from injection of illegal certificate or hash to db**
Once illegal hash or certificate is placed to db, it is possible to execute malware driver or Boot Loader

**Threats from hash collisions.** E.g. findinf collisions for hashes in white list.

**Threat from attacks on SMM** Mode of processor

**Attacks on TPM module** (cold boot attack)

**Threats from direct write to SPI flash:** If SPI flash configuration is not locked, malware can be written directly to SPI flash

Fig. 2. UEFI threats

- Correctly implement all cryptography (signature verification, etc.).

UEFI gives a variety of mechanisms for providing information security and platform integrity. It's up to platform manufacturers to use them as well as protections offered by hardware. Bugs in implementation and incorrect usage of those mechanisms may ruin the entire defense.

We hope this article will help developers, system administrators and security specialists to understand the main security problems and key points of UEFI technology.

## ACKNOWLEDGMENT

We would like to thank a leading security solutions manufacturer Infotecs Corp. and it’s academic program for the support of this work.

## REFERENCES

[1] NIST 800-147 BIOS Protection Guidelines. Recommendations of the National institution of Standards and Technology
[2] UEFI Forum, official site of UEFI developers consortium, Web: http://www.uefi.org
[3] UEFI Specification (v2.3.1C June 2012)
[4] PI Specification (Platform Initialization) (v1.2.1 May 2012)
[5] UEFI Technology: say hello to the windows 8 bootkit, Web: http://www.saferbytes.it/2012/09/18/uefi-technology-say-hello-to-the-windows-8-bootkit/
[6] Samuel T. King, Peter M. Chen, Yi-Min Wang, Chad Verbowski, Helen J. Wang, Jacob R. Lorch. SubVirt: Implementing malware with virtual machines. 2006, Web: http://research.microsoft.com/pubs/67911/subvirt.pdf
[7] UEFI and Dreamboot by Sébastien Kaczmarek, QUARKSLAB, Web:http://www.quarkslab.com/dl/13-04-hitb-uefi-dreamboot.pdf
[8] Implementing and Detecting a PCI Rootkit – John Heasman, 2007, Web: http://www.blackhat.com/presentations/bh-dc-07/Heasman/Paper/bh-dc-07-Heasman-WP.pdf
[9] Loukas K. (snare), DE MYSTERIIS DOM JOBSIVS Mac EFI Rootkits, Black Hat USA 2012, Web: http://ho.ax/De_Mysteriis_Dom_Jobsivs_Black_Hat_Paper.pdf
[10] Yuriy Bulygin, Andrew Furtak, Oleksandr Bazhaniuk, A Tale of One Software Bypass of Windows 8 Secure Boot, Black Hat USA 2013
[11] Microsoft Corporation, Windows Hardware Certification Requirements, updated: September 18, 2012, Web: http://msdn.microsoft.com/library/windows/hardware/dn423132.aspx
[12] R. Wojtczuk and A. Tereshkin. “Attacking Intel BIOS”, Black Hat USA. Las Vegas, NV, 30 July 2009.
[13] Sherri Sparks and Shawn Embleton. “SMM Rootkits: A New Breed of OS Independent Malware”, Black Hat USA, Las Vegas, NV, USA, 2008.
[14] Rod Smith, “Managing EFI Boot Loaders for Linux:Dealing with Secure Boot”, Web: http://www.rodsbooks.com/efi-bootloaders/secureboot.html