Masaryk University
Faculty of Informatics

Operating system boot from fully encrypted device

Bachelor's Thesis

Daniel Chromik

Brno, Fall 2016

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Daniel Chromik

Advisor: ing. Milan Brož

Acknowledgement

I would like to thank my advisor, Ing. Milan Brož, for his guidance and his patience of a saint. Another round of thanks I would like to send towards my family and friends for their support.

Abstract

The goal of this work is description of existing solutions for booting Linux and Windows from fully encrypted devices with Secure Boot. Before that, though, early boot process and bootloaders are described. A simple Linux distribution is then set up to boot from a fully encrypted device. And lastly, existing Windows encryption solutions are described.

Keywords

boot process, Linux, Windows, disk encryption, GRUB 2, LUKS

Contents

List of Tables

List of Figures

1 Introduction

The importance of disk encryption has been on the rise in the last 20 years. It protects data in case of theft, where laptop or hard drive gets stolen. Nowadays most computers are also connected to the internet. Which opens another door into the computer where encryption could be helpful .

Disk encryption is the process of making the data look like it is just a random sequence of numbers. Decryption is the reverse process, making the data readable again. On Linux one of the options for encryption would be dm-crypt, on Windows BitLocker or VeraCrypt. Only these two Operating Systems are covered in this work. Encryption programs use various encryption keys, algorithms, hashes etc., information about those is stored in headers. These headers can be stored directly on the disk, at the beginning of the encrypted partition. Or it could be stored on an external drive, like USB (Universal Serial Bus) flash disk or on a server.

Full disk encryption is where the entire drive is encrypted, including the OS. The only exception are components needed to boot the OS. This form of encryption provides the benefit of plausible deniability, it is impossible tell how many Operating Systems (OS) are there on the disk, if any. The Operating System also cannot be altered when not in use, and every new file will be automatically encrypted if it resides on the same drive.

1.1 Thesis goals

The goal of this work is to study and describe early boot process and generic structure of a boot loader used in OS on the PC (Personal Computer) platform. Focusing on the UEFI (Unified Extensible Firmware Interface) firmware specification and Secure Boot. And, using that knowledge, firstly install a Linux distribution on a fully encrypted device. Describe possible ways to set up a Linux with Secure Boot enabled to boot from a fully encrypted device. And lastly describe existing solutions for booting Windows from a fully encrypted device. The only unencrypted parts of the boot process can be parts critical for the boot process to work.

1.2 Thesis structure

The thesis begins with a description of the general early boot process up to the Boot Loader phase. Followed by a description of BIOS (Basic Input/Output System) and the newer UEFI. These are critical components of the Boot Process that tell the system how to boot an Operating System. Description of Partitioning Tables is next, these tell the Operating System, Firmware Interface or Boot Loader where each partition on the disk is located. One section is dedicated to Secure Boot, a feature of UEFI specification which makes the boot process more secure.

The second chapter is dedicated to Boot Loaders themselves, GRUB 2 (Grand Unified Bootloader) and Windows Boot Manager. GRUB is a crucial component for booting from an encrypted device, mainly for the Linux OS. It might be possible to use GRUB to boot Windows from encrypted device as well, which is described in the last chapter. The implementation part is divided into 2 chapters. First chapter describes the steps needed to install Linux on a fully encrypted device and making it bootable. The possibility of enabling Secure Boot is also looked into. Second chapter is about existing options of booting Windows from a fully encrypted device with and without Secure Boot.

2 Boot Process Description

When the system is powered on, the CPU (Central Processing Unit) needs to know how to check if the hardware, like memory or hard drives, is not faulty, so it can proceed with the bootstrap process. It also needs to know how and where to look for an operating system. This is where Firmware Interfaces and Partitioning Tables come into place, the former allowing to check hardware or how to look for an OS, while the latter helping with finding the OS. Some problems can be avoided with the knowledge of how the early boot process works, especially when installing an older Operating System on new hardware or running a multi-boot[1] configuration.

2.1 Early Boot Process

When the system is powered on it starts by loading system firmware, like BIOS[6] or UEFI[14], into memory. These can be loaded from Flash or ROM (Read-Only Memory). System firmware on flash memory is usually prioritized as it can be updated and modified. The ROM is mostly used as a backup if the Flash storage becomes unusable. The system then performs POST (Power-On Self-Test) to check the hardware and initialize it. CPU (Central Processing Unit) then searches for boot record to load into the main memory and gives it control.

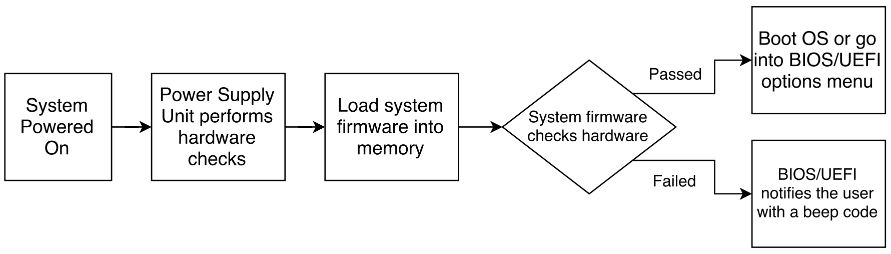
System Powered On
Power Supply Unit performs hardware checks
Load system firmware into memory
System firmware checks hardware
Passed
Failed
Boot OS or go into BIOS/UEFI options menu
BIOS/UEFI notifies the user with a beep code

Figure 2.1: Early Boot Process

1. two or more operating systems installed on a PC (e.g. Linux and Windows) and choosing which to boot on startup, more information at https://en.wikipedia.org/wiki/Multi-booting

2.2 Firmware interfaces

Firmware interfaces provide ways to interact with platform firmware. They provide ways to change how the system works before an OS is loaded, like overclocking the CPU or changing boot priorities. There are 2 firmware interfaces used on the x86 platform, BIOS and UEFI.

2.2.1 BIOS – Basic Input/Output System

BIOS[6] first appeared in the year 1975, and is still used nowadays. But because of its age, there are many shortcomings and features that are no longer useful. These are being addressed by its successor, UEFI[14], which is described in the next subsection.

BIOS can only run on the x86 architecture because it requires 16-bit mode that it provides. 16-bit mode also leads to BIOS's unused functionality, it provides an interface for the OS to communicate with hardware. It was used mainly in MS-DOS times, but today's Operating Systems are 32 or 64 bit, which renders BIOS's 16-bit interface inefficient. BIOS boot process is fairly straight forward as shown in Figure 2.2, but it is not that modifiable.

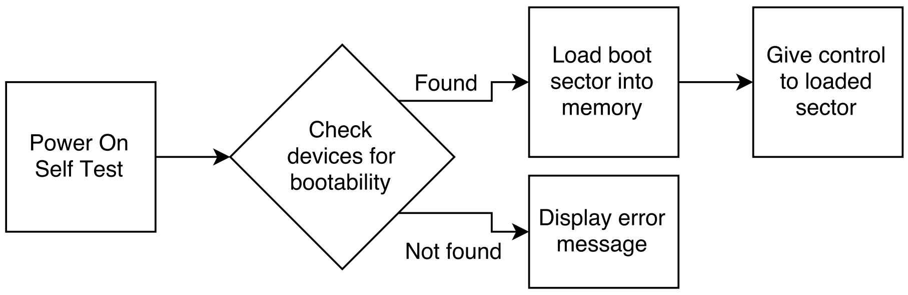
Power On Self Test
Check devices for bootability
Found
Not found
Load boot sector into memory
Give control to loaded sector
Display error message

Figure 2.2: BIOS boot process

BIOS can only boot hard drives partitioned using the MBR method, more about MBR[2] can be found in subsection 2.3.1. Booting from GPT[7], more in subsection 2.3.2, is dependant on the secondary bootloader, like GRUB[12], which is be described in section 3.1. Multi-booting on the same physical drive is also dependant on the secondary bootloader as BIOS can only change priority of the physical drives.

2.2.2 UEFI – Unified Extended Firmware Interface

UEFI[14] is a modern replacement for BIOS. It was developed by Intel, first called Extended Firmware Interface (EFI), which was later contributed to the Unified EFI forum and was renamed to UEFI. UEFI doesn't require a boot sector, it has a built-in bootloader that lets the user change the boot configuration.

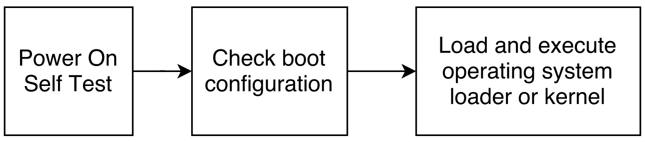

Figure 2.3: UEFI boot process

UEFI understands GPT partitioning and can boot from the EFI System Partition (ESP). ESP must be formatted using the File Allocation Table (FAT) file system specification. It contains OS bootloaders or kernels, device drivers used by the firmware itself and system utility programs. These programs must be compiled as EFI applications with .efi filename extension. UEFI also provides backwards compatibility in the form of legacy BIOS mode, which can boot from MBR partitioned disk.

2.3 Partitioning tables

Partitioning tables tell the firmware where to look for boot sectors. There are 2 mainly used schemes, Master Boot Record and GUID Partitioning Table, with a significant amount o differences between them.

2.3.1 MBR – Master Boot Record

In the MBR[2] scheme the boot record is located at the first sector of a hard disk. The general classic Master Boot Record contains first stage bootloader and partition table, as shown in figure 2.4. The first stage bootloader often redirects to a second stage bootloader, which provides more features.

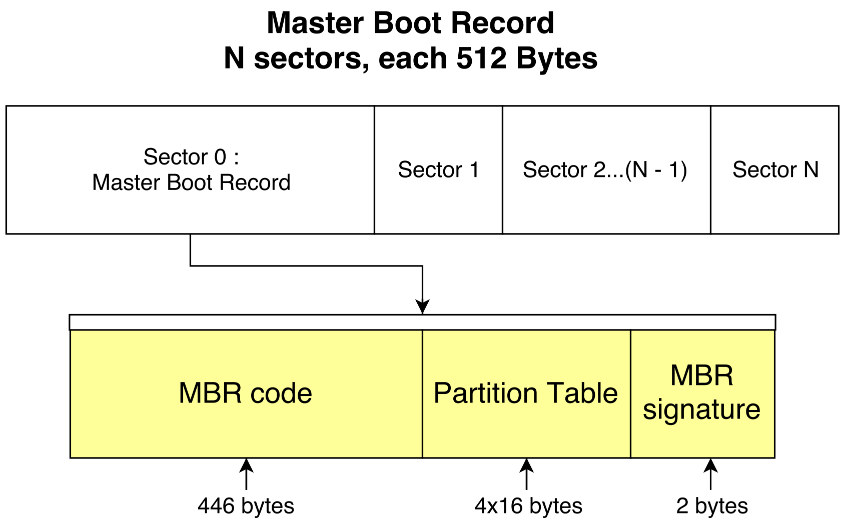

Figure 2.4: Classic Master Boot Record

There are many variations as to what the MBR contains, e.g. Disk Manager MBR can have up to 16 partition entries, but only has 252 bytes for the bootstrap area.

MBR was introduced in 1983 and has not changed much since then, its shortcomings are more and more noticeable nowadays. It can address only up to 4 TB of data, which can be insufficient. If the MBR gets damaged there is no way of restoring it, since there is no data redundancy. General MBR also has only 4 partitions, this, however, can be overcome with the use of extended logical partitions as shown in figure 2.5.

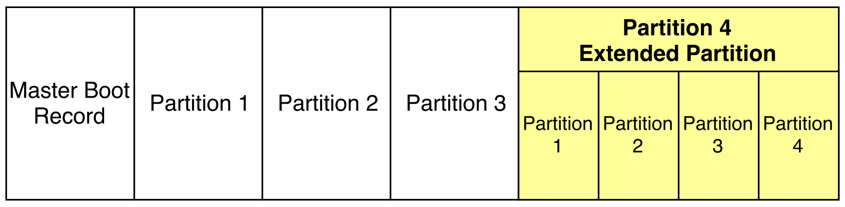

Figure 2.5: Extended Master Boot Record example

Only one entry in MBR can be an Extended Partition. An Extended Partition can contain another Extended Partition though, forming a linked list which is limited only by disk space.

2.3.2 GPT – GUID Partition Table

GPT[7] is a more modern approach to partitioning, it solves many problems posed by MBR. There can be 2^{64} sectors on the disk, a sector being 512 or 4096 bytes, providing much more space than MBR. In the GPT scheme, extended partitions are no longer needed as there can be up to 128 partitions.

To provide these features GPT requires much more space than MBR. It requires at least 16,384 bytes for the partition table entries which, on 512 byte sectors, would take 32 sectors. It is possible to boot from the disk as if it was an MBR disk thanks to protective MBR layer, as shown on figure 2.6. Booting from the disk without the protective layer, as if it was an ordinary MBR disk, would not work at all and it could even damage the GPT data.

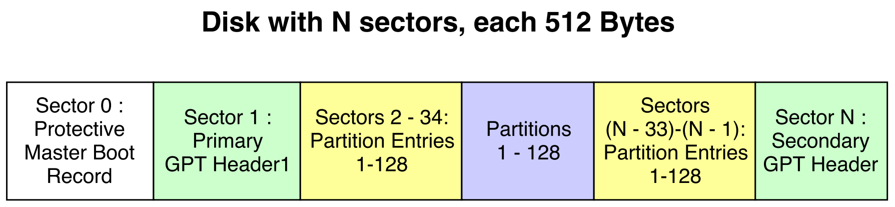
Disk with N sectors, each 512 Bytes
Sector 0 : Protective Master Boot Record
Sector 1 : Primary GPT Header1
Sectors 2 - 34: Partition Entries 1-128
Partitions 1 - 128
Sectors (N - 33)-(N - 1): Partition Entries 1-128
Sector N : Secondary GPT Header

Figure 2.6: GUID partition Table

Every partition has a long random string called Globally Unique Identifier (GUID). Redundancy is present in GPT as the partition table and GPT header are at the beginning and end of the disk to prevent data loss. The GPT header contains CRC32 of the header itself as well as the partition array to tell if the data has been corrupted.

Most modern Operating Systems (Windows, OS X, Linux...) can read from GPT partitioned disk. However, some systems (e.g. Windows, OS X) require EFI firmware to also boot from it.

2.4 Secure Boot

Secure Boot[3] is a feature of the UEFI[14] specification that should make the boot process more secure. This security is achieved by preventing drivers or OS loaders that are not signed from being loaded by the firmware.

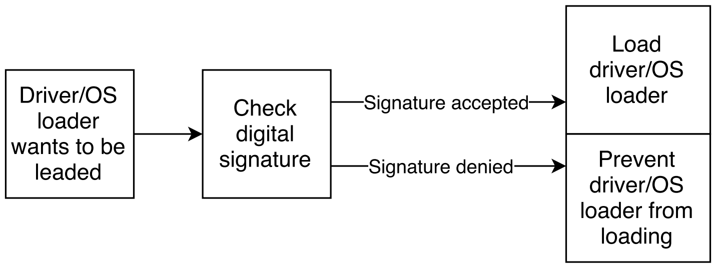

Figure 2.7: Secure boot driver/loader loading

This should prevent unwanted software from loading, such as bootkits. It also, at first, made other Operating Systems, like Linux or BSD, impossible to boot while using default PKs (Platform Keys), a PK is described after Figure2.4. Some Linux distributions, mainly Fedora and Ubuntu, can be run in Secure Boot mode with Microsoft keys. But for many others it has to be disabled or custom PK has to be used.

With the arrival of Windows 10 manufacturers can ship hardware on which Secure Boot cannot be disabled. This is used mainly on portable Windows devices, like tablets or mobile phones.

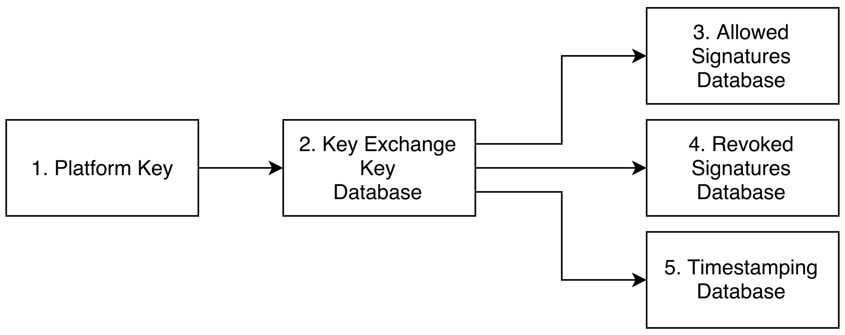
1. Platform Key
2. Key Exchange Key Database
3. Allowed Signatures Database
4. Revoked Signatures Database
5. Timestamping Database

Figure 2.8: Secure boot databases

1. Platform key - the public part of Platform Key which was generated by the Original Equipment Manufacturer (OEM). It is used to protect the KEK Database.

2. Key Exchange Key (KEK) Database - trusted certificates that allow modification of DB, DBT and DBX, described below. Usually contains certificates of OS vendor.

3. Allowed Signatures Database (DB) - CA certificates or their hashes used to sign bootloaders and other pre-boot components are stored in this database. It could also contain explicit SHA2 hashes of the bootloader images.

4. Revoked Signatures Database (DBX) - contains hashes and certificates that were revoked. If the bootloader is signed by anything from this database, or if its hash is stored here, it will not be allowed to execute.

5. Timestamp Signatures Database (DBT) - contains timestamping certificates used when signing bootloaders.

Usually the databases look like this on new Windows machines.

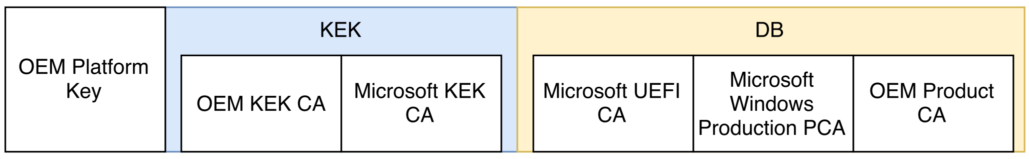

Figure 2.9: Database state on Windows machine

1. Platform key - described above. It is possible to replace it with custom key, giving the option to sign anything the user wants to, but Windows components will become unbootable.
2. Key Exchange Keys (KEK) - there are usually two of these, 1 from OEM and 1 from OS vendor.
3. Allowed Signatures Database (DB)
   (a) Microsoft UEFI CA - this certificate signs 3rd party binaries, like Fedora or Ubuntu bootloaders.
   (b) Microsoft Windows Production CA - used to sign Microsoft binaries.
   (c) OEM Product CA - certificate specific to the OEM. Gives the OEM an option to sign its own bootloaders. As an example Lenovo could make a laptop with Windows and a custom Linux distribution with Secure Boot enabled and both would work correctly since Lenovo can sign the Linux bootloaders.

3 Boot loaders

Boot loaders are an important part of the booting process. Even though UEFI[14] systems can directly boot an OS kernel, boot loaders, such as GRUB 2[12] and Windows Boot Manager, provide additional functionality. The first section takes a closer look at GRUB 2, the most widely used boot loader for Linux Systems, and its features. Windows Boot Manager and Loader is described in the second section.

3.1 GRUB

GRUB[12], which stands for Grand Unified Bootloader, is used mainly to boot UNIX like systems. It is a reference implementation of the Free Software Foundation's Multiboot Specification. There are two versions, GRUB legacy and GRUB 2. GRUB Legacy is no longer supported and should be used only for older systems where GRUB 2 would not work. GRUB 2 is cleaned up and rewritten GRUB Legacy in order to support further development.

3.1.1 Boot process on BIOS

Booting on MBR[2] disk has three stages on GRUB, stage 1, 1.5 and 2. These three stages are illustrated in figure 3.1. Boot process on GPT[7] is described in next subsection.

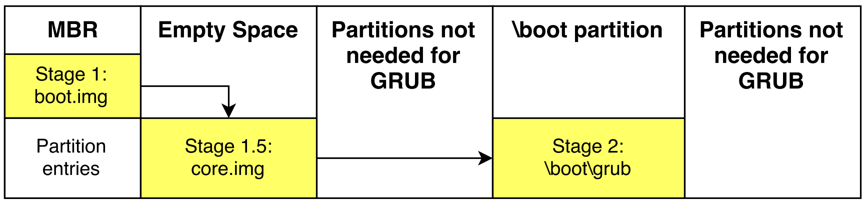

Figure 3.1: GRUB boot process on BIOS-MBR

First stage, boot.img, happens when BIOS[6] gives control to GRUB code located in the Master Boot Record. Boot.img is configured at installation time to only load the first sector of core.img. The data for 1.5 stage, core.img, are located in the unused space between MBR and first partition. Once core.img is executed, it loads configuration files and any needed modules, like file system drivers, and loads stage 2 by its file path. Once stage 2 is loaded it presents the user with a Text User Interface based operating system selection. There is a slight difference between MBR and GPT as GPT does not have any free space between the protective MBR layer and first partition. On GPT disk, core.img needs to be stored in a separate partition as shown in figure 3.2.

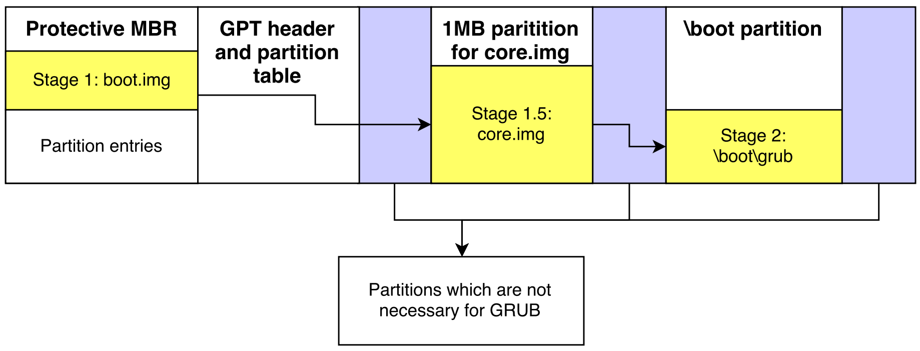

Figure 3.2: GRUB boot process on BIOS-GPT

3.1.2 Boot process on UEFI

On UEFI-based systems the boot process itself is a lot simpler on the GRUB[12] side. UEFI[14] systems provide their own bootloaders which load grub.efi into memory and give it control.

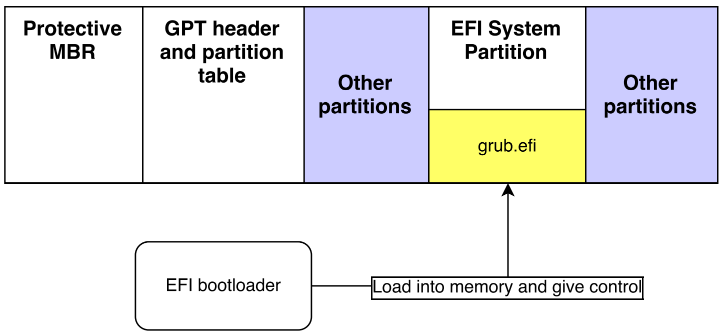

Figure 3.3: GRUB boot process on UEFI with GPT disk

This process becomes more complicated if secure boot is used, detailed description in section 3.1.3. Booting from MBR disk is the same as in section 3.1.1 since UEFI provides legacy BIOS mode.

3.1.3 GRUB and Secure Boot

When secure boot is enabled both GRUB and Linux kernel need to be signed and verified before booting. In this work only the Fedora implementation will be described, however, there are more implementations of GRUB with secure boot.

Fedora uses a first stage bootloader called Shim [13]. Firstly shim is verified and executed by UEFI. It then verifies and loads GRUB bootloader, the verification is done with a CA signed by Fedora. When GRUB is loading a kernel it calls back into shim to verify the signature of the kernel.

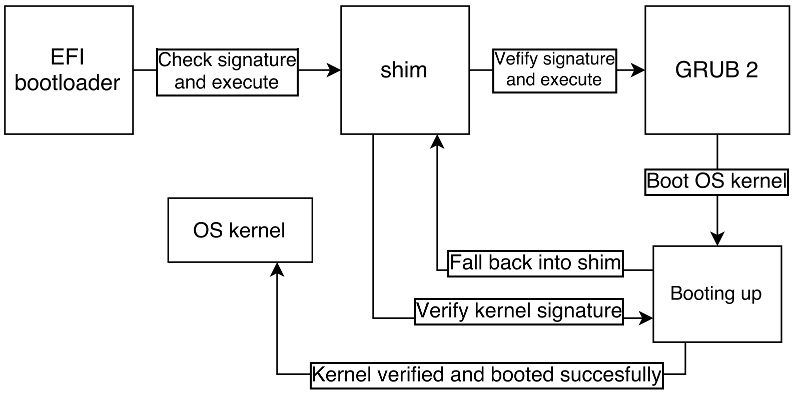

Figure 3.4: GRUB secure boot process on Fedora

Fedora also checks kernel modules at load time and refuses to load them if they are not signed or have invalid signature. This last step is where other Linux distributions are different as they do not require that modules are signed. Rest of the process should be similar as most distributions use shim in their boot process in the same way as Fedora.

3.2 Windows Boot Manager and Loader

There are two versions of windows boot loaders, NTLDR, which was used to boot windows XP and older, and Windows Vista startup process. NTLDR needed to change because it could not boot in EFI mode, it was split off into Boot Manager and Boot Loader.

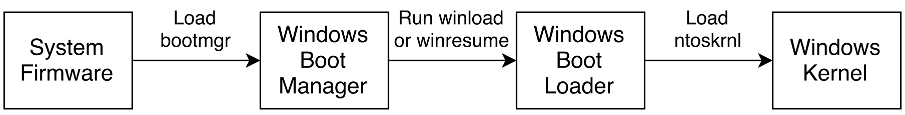

Figure 3.5: Windows Vista boot process

3.2.1 Boot Manager

This is the part that lets the user choose which operating system to boot if he has more than one. It also provides Advanced Boot Options, these contain diagnostics and repair tools, like Safe Mode or Debug Mode, which could be needed if the OS doesn't boot properly.

Boot manager reads from Boot Configuration Data (BCD), which can be edited since it uses the same format as Windows Registry hives. BCD contains entries for options like booting or resuming an OS from hibernation, invoking a NTLDR version of windows or loading and executing a Volume Boot Record, allowing to chain load GRUB[12] as an example. BCD is located at `\EFI\Microsoft\BCD` on EFI partitions or `\boot\BCD` on MBR partitioned drives.

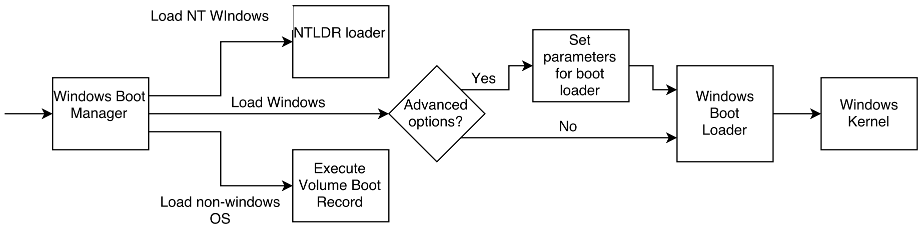

Figure 3.6: Windows boot manager

3.2.2 Boot Loader

There are two Boot Loaders in the Vista start up process, winload and winresume. Winload is executed by the Boot Loader to load the OS kernel and core device drivers, in EFI systems is it called winload.efi. Winresume, called winresume.efi on EFI systems, is also executed by the Boot Loader, but it is used to resume the system from Hibernation. Both of these files are at `\Windows\system32`.

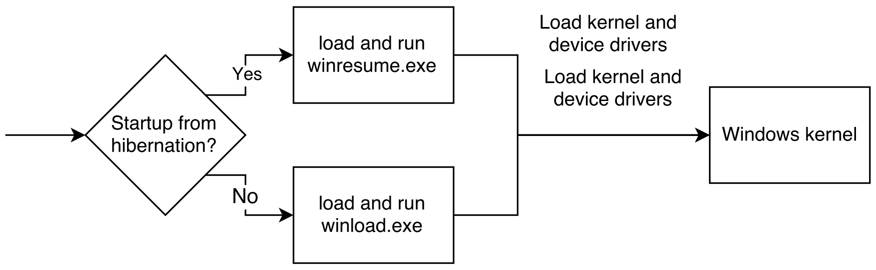
Startup from hibernation?
Yes
No
load and run winresume.exe
load and run winload.exe
Load kernel and device drivers
Load kernel and device drivers
Windows kernel

Figure 3.7: Windows boot loader

4 Setting up encrypted device

Setting up an encrypted Linux OS is described in this chapter. Only the most important steps are explained, why they need to be done and how could they be done differently. The step-by-step guide can be found in the attachment. The first part is about disc partitioning followed by the creation of filesystems and partition encryption n the final section the mkinitcpio and GRUB 2[12] will be set up

4.1 Partitioning discs

Since this is a manual installation, the partitioning will need to be done manually as well. In this case the boot partition will be omitted as well as the Logical Volume Manager partition, why these are omitted will be explained a bit further. The result of this section should be 2 visible partitions, EFI and a LUKS[10] encrypted partition.

4.1.1 Booting without boot partition

Most Linux OS Installers, like Ubuntu or Fedora, make a boot partition for GRUB. This makes the management of multiple Linux kernels or distributions simpler. The boot process would then look like this.

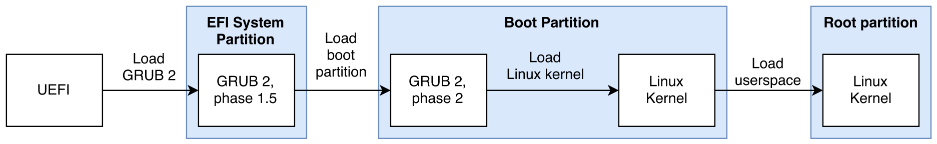

Figure 4.1: GRUB booting with boot partition

However, a dedicated boot partition is not necessary in order to boot a Linux OS. If the partition is not created the boot data is stored on the root partition. Without the boot partition full drive encryption is achieved, the only exception being ESP which cannot be encrypted, ESP was described in subsection 2.2.2. One less decryption step in the boot process could also make the process more secure.

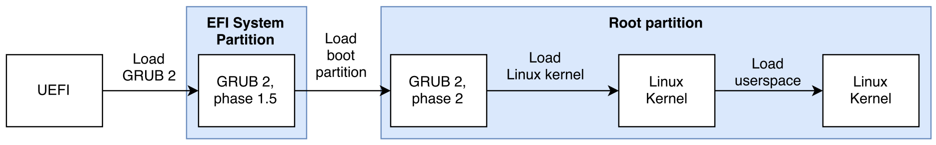

Figure 4.2: GRUB booting without boot partition

4.1.2 Logical Volume Manager

The Linux Logical Volume Manager (LVM) provides 2 main features. It allows to resize filesystems much more easily than if they were normal partitions. LVM also logically manages physical volumes, making them look as if they were just one. However, booting from LVM partitions is a bit more complicated, especially when they are encrypted and there is no boot partition.

4.2 Preparing Logical Volumes and Boot

The partitions then need to be made into filesystems so they can store data in a way that the OS understands. Encryption of these partitions will be described in this section as well.

4.2.1 Setting up filesystems

Just allocating size for partitions is not enough, the OS needs to know how to read from and write to the partitions. Filesystem types themselves tell what kinds of metadata are stored with each file and directory, how space is allocated or if they provide features such as symbolic or hard links. Filesystems can be OS specific, e.g. Linux ext4 filesystem is not supported by Windows.

4.2.2 Partition encryption

The partitions are encrypted using dm-crypt [1](which stands for Device-Mapper Crypt), a flexible tool available on Linux and BSD Operating Systems. It is implemented as a device mapper target, which makes it possible to stack more software layers on top of another. As an example, an LVM partition,inner layer, which would then be encrypted, outer layer, this would be a 2 layer layout. Aforementioned example is illustrated in the following figure 4.2.2.

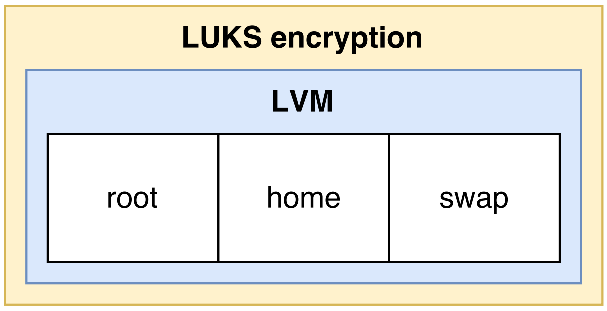

Figure 4.3: Device mapper example

4.2.3 Linux Unified Key Setup - LUKS[10]

LUKS is a disk encryption specification, it provides a standard on-disk format for use in various tools. Thanks to this standardisation it is much more compatible among Linux distributions. It also provides secure management of multiple user passwords. LUKS stores all the setup information in the partition header. In comparison, plain dm-crypt encryption does not require a header on the device, making it look like the device is filled with random data. The header, illustrated in the figure below, needs to be read by the bootloader in order to decrypt the partition.

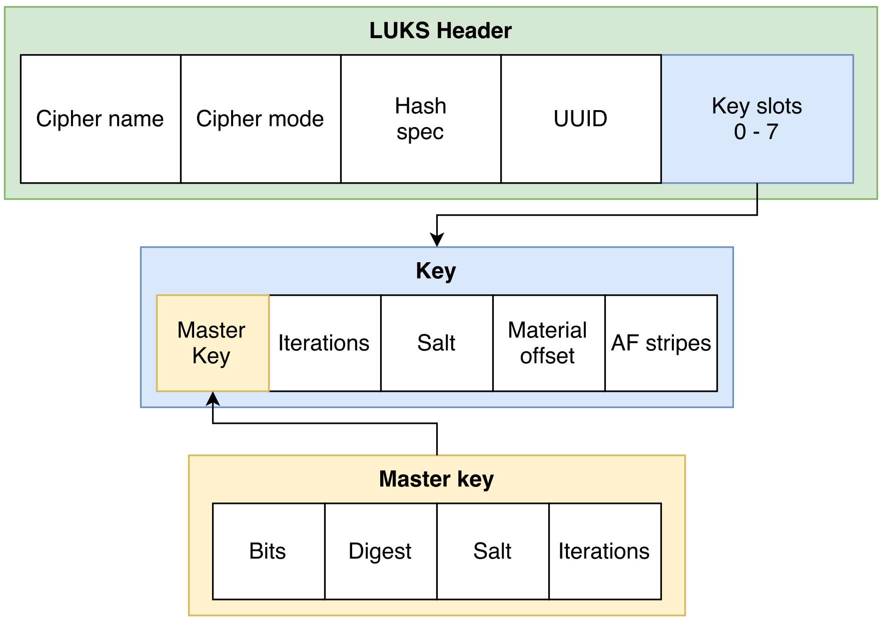

Figure 4.4: LUKS header

4.3 Configuring mkinitcpio and GRUB 2

GRUB and initialization procedures then need to be configured, manually. GRUB needs to be configured to boot without the boot partition and decrypt the initial ramdisk. Kernel initialization scripts must decrypt the root partition. Setting these components up is fairly straightforward as both have this functionality built-in.

4.3.1 Setting up GRUB

Firstly GRUB needs to be configured in order to decrypt the partition before booting it. GRUB has this functionality built-in but it can't automatically detect encrypted partitions. It needs to be told where the encrypted partition is, whether by its UUID or, if it was a LVM partition, by location of the volume. After the partition is opened it is stored as a device-mapper device so the OS can use it as well.

4.3.2 Configuring mkinitcpio

Setting the initialization scripts is done similarly to GRUB, by adding flags to files. The scripts need to know that the partition was encrypted so they can work with the device opened and mapped by GRUB.

4.3.3 Storage Stack

After the partitioning, encrypting and installing the OS is complete the storage stack should look something like this.

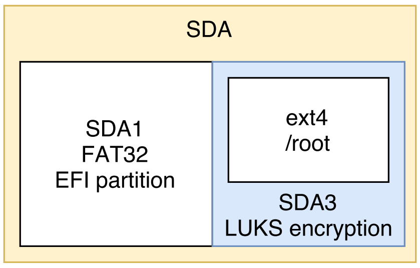

Figure 4.5: State of the storage stack

The home and swap partitions are not included as they are not necessary. Linux requires only the root partition in order to function properly. EFI partition is required by the EFI firmware, it contains the GRUB bootloader. The partitions could, of course, differ if the user has any other partitions or chooses to use the boot partition.

4.4 Enabling Secure Boot

It is not that difficult to set up a fully encrypted device, as was discovered above. One important security feature was omitted though, Secure Boot. There are two ways to set up Secure Boot, using the default PK or using a custom one. Installing an already signed Linux distribution is required if default PK is to be used, more in subsection 4.4.1. Removing default PK and installing a custom one requires more steps but any Linux distribution can be installed in this way. Custom PK way is described in subsection 4.4.2.

4.4.1 Using default PK

If secure boot is enabled when installing Fedora alongside Windows it uses first stage shim bootloader, shim was described in subsection 3.1.3. There has to be an unencrypted boot partition if root partition is to be encrypted. The reasoning behind this is that shim can only verify and load another bootloader, it can't decrypt a partition by itself. Because of that it is impossible to boot a Fedora Linux from a fully encrypted device.

4.4.2 Using custom keys

Another option to have a fully encrypted device with Secure Boot enabled is to clear existing UEFI keys and generate custom ones. This would allow the execution of any bootloader and kernel, but they will have to be signed manually. The use of Windows could become more cumbersome or it might not boot at all, when the Windows bootloader or kernel gets updated, the keys will need to be manually updated as well. Using this method only the bootloader will be verified, the kernel will be booted normally. This is, of course, depending on the bootloader, it might support some sort of kernel verification.

4.4.3 Cryptboot

Cryptboot[9] is a Linux bash script which automates custom UEFI key management. It basically automates the process described above, UEFI keys removal and generation of custom ones. Cryptboots main goal is to prevent Evil Maid attacks.

5 Booting encrypted Windows

Booting a fully encrypted Windows 10 OS has some similarities and differences when compared to Linux encryption. The tools used to encrypt Windows are different. But the methods of encryption are fairly similar. Microsoft provides some encryption methods, these are described in section 5.2. There also third-party encryption options, some proprietary and some open-source, more in section 5.3.

5.1 Windows partitions on GPT

The minimum amount of partitions needed to boot and run Windows 10 is 2, ESP, called just System partition by Microsoft, and the Windows partition. In this scenario, encrypting and booting from the Windows partition would mean booting from a fully encrypted device as the ESP must not be encrypted. Windows, however, usually comes with more partitions as illustrated in the image below.

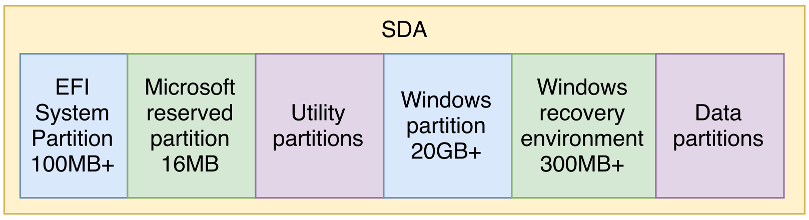
SDA
EFI System Partition 100MB+
Microsoft reserved partition 16MB
Utility partitions
Windows partition 20GB+
Windows recovery environment 300MB+
Data partitions

Figure 5.1: Windows storage stack

EFI system partition was described in subsection 2.2.2. Windows partition contains the entire Windows OS and, if not stored in a data partition, user data. Windows partition is something like Linux's home and root combined. Utility partitions can be any partitions which are not managed by Windows, like Linux root and home partitions. These partitions should be located before the Windows partition, allowing to resize Windows without affecting the utility partitions. Data partitions are optional partitions which store user data, it is a part of the Windows partition if not defined. The remaining partitions are described in the following subsections.

5.1.1 Microsoft System Reserved - MSR

This is a hidden partition that exists only on GPT partitioned disks. On MBR partitioned disks the data is stored in the empty space after bootloader. The partition itself doesn't contain any data, other components shrink this partition to create their own partitions which store data. For example, when converting basic disk to a dynamic one, Windows takes a bit of space to store information about the new dynamic disk.

5.1.2 Windows Recovery Environment

RE partition is booted when the bootloader fails to boot normal Windows. When the bootloader is given control it sets a flag which should be unset by windows kernel when it is successfully booted. This flag will be set if the kernel was never given control, thus telling the bootloader that it should go into recovery environment.

5.2 Microsoft encryption solutions

Microsoft does provide some encryption solutions. Windows 10 device encryption, which comes out of the box on Windows 8.1 and later. The second, more advanced option, is BitLocker. It is not available on Windows 10 Home, only Pro and Enterprise, though. Both of these options are proprietary, not open-source.

5.2.1 Windows 10 device encryption

Windows device encryption arrived with Windows 8.1 and is available on all editions of Windows. It is basically a limited version of BitLocker, which is discussed in subsection 5.2.2. It has 2 requirements. Firstly, the hardware must support at least TPM 1.2. (Trusted Platform Module) and Secure Boot. And secondly, the user must be logged in to a Microsoft account, encryption keys are then uploaded to Microsoft servers. This second requirement provides an option to recover files if unable to log in. But it also makes this encryption method less secure as the keys are stored in a way that the user cannot change.

5.2.2 BitLocker

BitLocker can use, but does not require, TPM. Instead of relying on TPM a volume password or an external USB startup key can be used. Microsoft account is not required as well, the keys are not sent to any servers. 2 feature of BitLocker are described in separate subsections, BitLocker to go in subsection 6.2.3 and BitLocker eDrive in subsection 6.2.4.

5.2.3 BitLocker to go

BitLocker to go allows the encryption of external devices, like flash disks or SD cards. These disks must be formatted using the NTFS, FAT16, FAT32 or exFAT filesystems. BitLocker to go usually includes an executable file at the beginning of the device, before the encrypted partitions. This fie is called BitLocker to go reader and as the name suggests it provides read-only access for devices that do not have BitLocker installed.

5.2.4 BitLocker eDrive

BitLocker eDrive[11] allows to set passwords and handle SEDs (Self Encrypted Drive). SED is an implementation of hardware encryption on SSDs (Solid Slate Drives). Software encryption on storage media is the most widely used method, this is where the CPU encrypts and decrypts the data on disk. This decryption/encryption naturally requires some processing power from the CPU. The user can, however, change how the encryption works by using different programs and configuring these programs in whichever ways they allow.

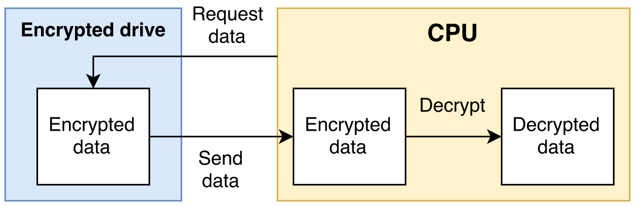

Figure 5.2: Software encryption

On the other hand, hardware encryption is done by the hardware, the CPU just receives or sends decrypted data, taking no additional processing power. But this form of encryption is not configurable, it is set in the hardware by the manufacturer.

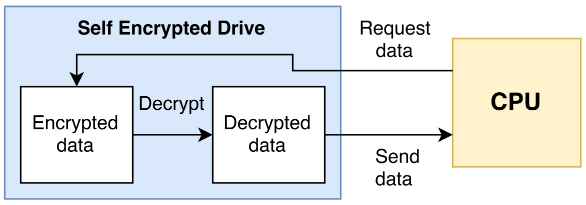

Figure 5.3: Hardware encryption

Setting up these drives usually needed to be done in the Firmware Interface, many motherboards didn't even provide this feature, it was available mostly on laptop motherboards. BitLocker eDrive simplifies this by setting the password and managing the SED while leaving the encryption to the drive.

5.3 Third party encryption

There is also a wide variety of third-party encryption solutions. In this work only some open-source implementations will be described but there are more, some open-source as well and some proprietary.

5.3.1 Truecrypt

Truecrypt[5] was released in 2004, when windows XP was most widely used, and there weren't many encryption options. It provided the options to encrypt a file, partition or even the entire storage device. Storage device encryption was possible because Truecrypt had its own bootloader which could decrypt the device before booting the Windows OS. But it was impossible to boot from a GPT[7] partitioned drive. Truecrypt is also no longer supported since 2014, allegedly because Microsoft Windows had a built-in encryption and Truecrypt was no longer needed. And since it is no longer supported it is no more a viable option for full device encryption, or any encryption at all. The source code is available to the public, thanks to this there are a few project which aim to replace Truecrypt, Veracrypt and Ciphershed are two such alternatives. DiskCryptor is, or was, a fork of Truecrypt as well, but its website can no longer be accessed, assuming that the project is dead.

5.3.2 Ciphershed

Ciphershed is one of the projects that forked from TrueCrypt, it is still in development. It aims to remain compatible with TrueCrypt volumes. At the time of writing there was no extra functionality when compared with TrueCrypt.

5.3.3 VeraCrypt

VeraCrypt[4] is also a fork of TrueCrypt but it isn't compatible with TrueCrypt volumes anymore. It not only fixes possible vulnerabilities but also provides new features, such as encrypting GPT devices. It is also possible to achieve a full device encryption with VeraCrypt. However, it doesn't support Secure Boot, at least at the time that this work was written.

5.4 LUKS encryption on Windows

There is no known LUKS[10] implementation on the Windows OS. It might be possible to modify an existing open-source solution so it will support it. Making such an implementation work with secure boot might be impossible, at least using default PK, as even VeraCrypt doesn't support it yet.

One way to approach this problem could be to modify VeraCrypt so it encrypts Windows with LUKS. VeraCrypts bootloader and driver would need to be modified as well. The most plausible solution to add Secure Boot to all this would be the usage of custom keys. Getting a custom Windows bootloader signed by Microsoft is not very probable.

6 Conclusion

In the end an Arch Linux was successfully set up to boot from a fully encrypted device. A step by step guide is provided to achieve an identical setup. While the demonstration in this work is too minimalist for practical use, with a little tweaking, an OS for a day to day use could be set up in a similar way. Secure boot[3] was not enabled in the provided demonstration. Enabling it is definitely possible, with CryptBoot[9] it should also be fairly simple, just not with default PKs.

Booting Linux from a fully encrypted device with Secure Boot using the default PK has proven to be impossible, at least at the time of writing. An already signed Linux distribution would need to be used and these don't support full disk encryption. Modifying the partitions and then GRUB accordingly would make GRUB unsigned. The only plausible option is that bootloaders shipped with these distributions start supporting full disk encryption. But a fully encrypted device would not be achievable when dual booting Windows since it uses different methods to encrypt partitions. Because of that there will be at least 3 visible partitions, ESP, encrypted Linux and Windows. To make a full device encryption with Windows and Linux possible, either Windows needs to start supporting Linux encryption methods or the other way around.

Windows and full disc encryption with secure boot is also not possible yet. While full disk encryption is possible to achieve with VeraCrypt[4], it doesn't support Secure Boot. And Microsoft's BitLocker[8] does work with secure boot, but it doesn't provide full disc encryption.

6.0.1 Future work

Possible works in the future could include modification of Fedora's Anaconda installer to include a full device encryption. If this proves to be too difficult with secure boot, at least an implementation with secure boot disabled should be possible.

When it comes to Windows, BitLocker cannot be modified as it is proprietary. And, as was described in subsection eraCrypt doesn't work with secure boot yet. Modifying VeraCrypt so it can be signed by Microsoft, or that it starts using custom keys, could possibly be implemented.

Making full disk encryption possible in dual-boot configurations could also be possible. There are 2 options to achieve that, encrypting Windows using LUKS, or encrypting Linux using one of the methods available on Windows. Encrypting Windows with LUKS could be done by modifying one of the open-source encryption solutions, VeraCrypt or CipherShed. Dm-crypt could encrypt Linux partitions in the same way VeraCrypt does. VeraCrypt could then be modified so it would be capable of booting these Linux partitions or opening them and passing control to GRUB.

Bibliography

[1] dm-crypt GitLab website. [Online, accessed 20-Nov-2016, retrieved from https://gitlab.com/cryptsetup/cryptsetup].

[2] Master boot record on Microsoft technet website [online]. [Online, accessed 20-Nov-2016, retrieved from https://technet.microsoft.com/en-us/library/cc976786.aspx].

[3] Secure Boot, Microsoft Technet blog. [Online, accessed 3-Dec-2016, retrieved from https://blogs.technet.microsoft.com/dubaisec/2016/03/14/diving-into-secure-boot/].

[4] VeraCrypt documentation. [Online, accessed 1-Dec-2016, retrieved from https://veracrypt.codeplex.com/documentation].

[5] Milan Brož and Václav Matyáš. The TrueCrypt On-Disk Format – An Independent View. *IEEE Security & Privacy*, 12, 2014.

[6] Wikipedia contributors. BIOS [online]. [Online, accessed 16-February-2016, Retrieved from `https://en.wikipedia.org/w/index.php?title=BIOS&oldid=704747098`].

[7] Wikipedia contributors. GUID Partition Table. [Online, accessed 20-Feb-2016, retrieved from https://en.wikipedia.org/w/index.php?title=GUID_Partition_Table].

[8] Niels Ferguson. AES-CBC + Elephant diffuser: A disk encryption algorithm for Windows Vista. Technical report, Microsoft Corporation, 2006. [online], accessed December 15, 2014.

[9] Michal Křenek. Cryptboot GitHub website. [Online, accessed 8-Dec-2016, retrieved from https://github.com/xmikos/cryptboot].

[10] Ondrej MOSNÁČEK. Key derivation functions and their gpu implementations [online], 2015 [cit. 2016-12-04].

[11] Tilo Müller, Tobias Latzo, and Felix Freiling. Self-Encrypting Disks pose Self-Decrypting Risks: How to break Hardware-based Full Disk Encryption. Technical report, Friedrich-Alexander-Universität Erlangen-Nürnberg, 2012. `https://www1.informatik.uni-erlangen.de/filepool/projects/sed/seds-at-risks.pdf`, [online], accessed December 15, 2014.

[12] GNU project. GNU GRUB. [Online, accessed 20-Nov-2016, retrieved from https://www.gnu.org/software/grub.

[13] Red Hat inc. shim. [Online, accessed 20-Nov-2016, retrieved from https://github.com/mjg59/shim].

[14] Richard Wilkins and Brian Richardson. UEFI Secure Boot in Modern Computer Security Solutions. [Online, accessed 20-Nov-2016, retrieved from `http://www.uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf`].