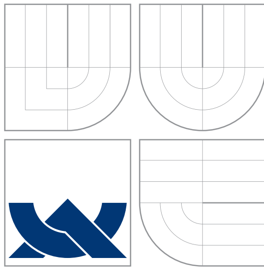
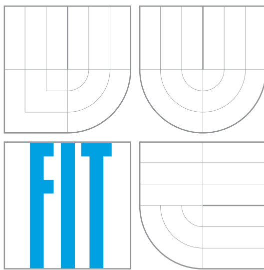


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

ROZŠÍRENÍ PROJEKTU SYSTEMD-BOOT O PODPORU PROTOKOLU SECURE BOOT

SUPPORT OF SECURE BOOT IN SYSTEMD-BOOT PROJECT

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. MICHAL SEKLETÁR
AUTHOR

VEDOUCÍ PRÁCE — Ing. ALEŠ SMRČKA, Ph.D.
SUPERVISOR

BRNO 2016

Zadání diplomové práce/18342/2015/xsekle00

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Sekletár Michal, Bc.**
Obor: Počítačové sítě a komunikace
Téma: **Rozšíření projektu Systemd-boot o podporu protokolu Secure Boot**
Support of Secure Boot in Systemd-Boot Project
Kategorie: Operační systémy

Pokyny:

1. Nastudujte protokol Secure Boot v rámci standardu UEFI pro autentizaci nízkoúrovňových systémů. Nastudujte standard X.509. Prozkoumejte aktuální stav projektu Grub a jeho alternativy Systemd-boot určené pro start systémů Linux.
2. Navrhněte úpravu Systemd-boot pro podporu Secure Boot zprostředkující autentizační mechanismy za účelem ověření důvěryhodnosti operačního systému. Navrhněte ucelené řešení důvěryhodného startu systému bez nutnosti využít originální páry šifrovacích klíčů ve firmware hardwarových zařízeních.
3. Implementujte podporu protokolu Secure Boot jako rozšíření pro projekt Systemd-boot.
4. Vytvořte testovací sadu prokazující kvalitu vámi vytvořených zdrojových kódů na úrovni přijatelné pro upstream projektu Systemd-boot.
5. Demonstrujte dosažené výsledky na reálném stroji podporující standard UEFI.

Literatura:

1. Secure Boot, Standard UEFI v2.5, sekce 30.1, duben 2015.
2. Domovská stránka projektu Systemd-boot. Dostupné na URL: http://www.freedesktop.org/wiki/Software/systemd/systemd-boot/

Při obhajobě semestrální části projektu je požadováno:

- Bez požadavků.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrčka Aleš, Ing., Ph.D.**, UITS FIT VUT
Datum zadání: 1. listopadu 2015
Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt

Cieľom tejto diplomovej práce je poskytnúť ucelený prehľad problematiky autentizácie v inicializačnom procese počítačov, pomocou technológie Secure Boot. Ďalej sa práca venuje prehľadu aktuálnych implementácií Secure Boot v operačných systémoch založených na jadre Linux. V závere práce je predstavená realizácia podpory Secure Boot vrámci projektu systemd-boot.

Abstract

The aim of this master thesis is to convey an ellaborate overview of Secure Boot, the technology used for an authentization during a platfrom boot up. Overview is followed by a description of contemporary implementations of Secure Boot found in the operating systems based on the Linux kernel. Finally, we propose a new implemenation of Secure Boot support in the systemd-boot project.

Klíčová slova

systemd, BIOS, UEFI, X.509, Secure Boot

Keywords

systemd, BIOS, UEFI, X.509, Secure Boot

Citace

Michal Sekletár: Support of Secure Boot in Systemd-Boot Project, diplomová práce, Brno, FIT VUT v Brně, 2016

Support of Secure Boot in Systemd-Boot Project

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Aleše Smrčky, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Michal Sekletár
May 24, 2016

Poděkování

Ďakujem vedúcemu práce pánovi Ing. Aleši Smrčkovi, Ph.D. za jeho čas a vedenie práce a kolegovi Ing. Lukáši Nykrýnovi za pripomienky počas tvorby práce.

© Michal Sekletár, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Contents

I Unified Extensible Firmware Interface (UEFI) 5

1 A Note on Terminology 5

1.1 Related Definitions 5

2 History of UEFI 6

2.1 Why Itanium Failed? 7

2.2 From EFI to UEFI 7

3 UEFI Architecture 8

3.1 UEFI Platform Initialization 8

3.2 Security (SEC) Phase 9

3.3 Pre-EFI Initialization Phase 10

3.4 Driver Execution Environment 14

4 Summary 21

II Public Key Insfrastructure and X.509 22

1 Cryptographic hash functions 22

2 Public Key Encryption 23

2.1 RSA Public Key Encryption 24

2.2 Public Key Infrastructure 25

3 X.509 26

3.1 History of X.509 26

3.2 Format of X.509 certificate 26

3.3 Certification Authorities . . . 27

3.4 Certificate validation and certification path validation . . . 28

3.5 Certification Requests . . . 29

3.6 Certificate Revocation Lists . . . 30

III UEFI Secure Boot **32**

1 Key Databases . . . 32

2 Secure Boot Modes . . . 34

3 Secure Boot Key Database Updates . . . 36

4 UEFI and Secure Boot in Fedora Linux distribution . . . 36

4.1 UEFI support . . . 36

4.2 Secure Boot support . . . 38

IV Implementation of Secure Boot support in systemd-boot **39**

1 CLI tool systemd-secure-boot . . . 40

1.1 Determining current Secure Boot state . . . 40

1.2 Creating certificates . . . 41

1.3 Updating authenticated variables . . . 41

2 Querying current state of signature databases . . . 43

3 Signature database backup . . . 44

4 Testing . . . 44

V Conclusion **45**

Literature **47**

Introduction

World is interconnected. Every day, new devices are being deployed and getting connected to the Net. Internet is the enabler of many developments in recent years. It is mostly due to the proliferation of the Internet that changed the way how we consume music (Spotify), movies (Netflix) and books (Kindle). What is more important, however, it also influenced how we do software development and deliver software to its users. With advent of technologies like git and Github, software development is more transparent and distributed than ever before. On the other hand micro-services based architectures enable us to build highly distributed systems oblivious to single node failures. Obviously, these novel architectures are harder to develop and debug in production. We argue that these changes were not self-inflicted but necessary, because users today expect services they consume to behave like the Internet. That means, always operational and performant.

Free access to the computer software source code has been demanded by activist for many years (since 1980's). As mentioned before, their requirement is now satisfied in many cases. Even companies which had been big proponents of Intellectual Property rights in the past are granting access to the source code of their products, e.g. open-sourcing of Microsoft's .NET development platform. More recently, similar appeals were made to lift the „embargo“ imposed on results of scientific community. Basically, technical reports, papers, outcomes of work of the scientist must be freely available to public, should the scientific teams receive grant money provided by state founded institutions.

In the light of the described events, we see more and more companies getting together to form foundations (e.g. Linux Foundation, OpenStack Foundation) and alliances (e.g. Khronos Group) to assure interoperability of technologies and products. This benefits all involved but most importantly the end users of technologies as it reduces chance for a vendor lock-in. This however is not novel, such cross-industry institutions existed before open-source revolution. Take for example Internet Engineering Task Force (IETF). Historically, communication and information interchange systems were the ones were vendors saw greatest value to collaborate on. This is not surprising. What is on the other hand, is the fact that recently, foundations and alliances formed in areas of computer industry which were highly proprietary in the past. Prime example is area of development of platform firmware (BIOS and other kinds). Even the name itself, firmware, is testimony to the proprietary nature of software and its development. But in 2005 Unified Extensible Firmware Interface (UEFI) Forum was formed. The organization which consists of more than 200 members, both companies and individuals. One of the main goals of this organization is to produce UEFI Specification. This specification is the document which describes a new model for the interface between computer operating system and platform firmware. Part of

this specification defines platform independent way for verification of components involved in boot sequence of UEFI based system. Specification is named Secure Boot. In its current form UEFI specification is very bulky piece of writing with more than 2500 pages. Secure Boot, being just very small fragment of entire spec, received great deal of attention in the recent years.

Main goal of this master thesis is to provide detailed overview of Secure Boot protocol. As mentioned before, Secure Boot itself is platform independent, but to provide additional capabilities to system administrators we need some awareness of Secure Boot in operating system level tools. Hence thesis will examine integration of Secure Boot technology in tools found in contemporary Linux based systems. Next, we provide overview of systemd, service and system manager readily available in all major Linux distributions to date. We argue that it is very important to have tools which will allow to administer Secure Boot capabilities available to system administrators. We also provide evidence that it is not always the case. That is why we picked systemd project as foundation for the tool which aims to ease configuration and supervision of Secure Boot UEFI subsystem. Practical product of this thesis is the implementation of such tool.

Thesis is structured as follows, first chapter provides some details of boot procedure performed by UEFI compliant firmware implementation. Chapter II focuses on crypto algorithms and data formats used to implement Secure Boot. Third chapter then discusses Secure Boot protocol itself. We conclude the thesis by description of command line tool aimed to manage Secure Boot signature databases.

Chapter I

Unified Extensible Firmware Interface (UEFI)

In order to talk about Secure Boot one must study UEFI first. This prerequisite is natural, because Secure Boot is part of the UEFI specification. To make the discussion complete we also include historical notes as they provide valuable perspective on why UEFI Forum was formed and UEFI specification written. Nonetheless, they are not essential for understanding of material discussed in this thesis, hence reader should feel free to skip them if he so desires. Also note that parts of this chapter are derived from [13][12].

1 A Note on Terminology

Terminology in space of UEFI and affiliated specifications (e.g. Trusted Computing Group specifications) tend to vary. Depends on who you talk to, but in general you may hear same term used in more than one context and things may get confusing very quickly. This can be attributed, to some extent, to the fact that these specifications sometimes post date actual implementations. For readers convenience we provide definitions of selected terms which may help the reader while reading this text or some other literature on the subject.

1.1 Related Definitions

- **UEFI** Specification which defines a new model for the interface between personal-computer operating systems and platform firmware.
- **UEFI Forum** The UEFI Forum is the group responsible for developing, managing and promoting UEFI specifications.
- **EFI** Project developed by Intel Corp. in late 1990's. It's goal was to provide necessary firmware infrastructure for booting Itanium based platforms. It is considered direct ancestor of UEFI.

- **Secure Boot** Part of the UEFI specification which defines means for authentication of UEFI drivers and applications.
- **Root-of-Trust** Is a component of a system (software, hardware) which is believed to be trustworthy and hence inherently trusted to execute or enforce critical security related function. Note that terms root-of-trust and *trust-anchor* maybe used interchangeably in the rest of the text.
- **Signature Database** UEFI Variable used to store crypto material, e.g. X.509 certificates, RSA keys or hashes. Terms, *key database* and signature database are used interchangeably in rest of the thesis.

2 History of UEFI

UEFI specification was released for the first time in January 2006. When it first came out it was already a version 2.0. It was predated by multiple releases of EFI specification. It is important to note that UEFI specification is developed and published by UEFI Forum while EFI specifications were released by Intel Corporation. Also first version of UEFI specification was mostly the same as EFI 1.0.

Efforts behind EFI which later provided foundations for UEFI started at Intel in late 1990's. Reasons for this developments were very practical. At the time Intel engineers across the company were hard at work to deliver brand new chip and platform architecture, Itanium. To boot the machine based on newly developed Itanium processors, they needed some basic pre operating system software. At very minimum there is a need for software component of the system which will check the hardware and initializes basic subsystems, e.g. memory subsystem and hands of control to OS loader. Traditionally, it was the duty of software which is often referred to as BIOS (Basic Input/Output Software). Indeed, this was the original plan, to stick the conventional BIOS onto the machines which were brand new otherwise, i.e. new processor architecture, new chipset and base board. Envisioned solution however turned out to be unfeasible from multitude of reasons. Legacy BIOS in general has various shortcomings which make it not very well suited for modern enterprise grade server hardware. Some of those are,

- 16-bit code
- limited execution environment
- difficult to program and extend
- very constrained hard disk support

These challenges started an initiative within Intel which was called Intel Boot Intiative (IBI). Its goal was to evaluate other options and pick most viable one considering both technical and business aspects of the problem. Couple of options were considered, but in the end decision was made to go with green field implementation. Designers of the solution were very familiar with problems of traditional firmware implementations. They knew that

BIOS is hard to extend and that is one of the main reasons for slow rate of innovation in firmware space. Thus, they settled with the design which was radically different from its predecessor. They set out to build high level C API for the firmware which would reduce the necessity for the OS or OS loader to care about platform details and specifics. Since the new architecture defined only the firmware interface, they realized that basic scope of the project could be expanded to also support platforms built on top of classic x86 processors. High level C API and a lot focus given to the possibility to extend the firmware interface and provide ground for innovations led to the giving new specification name: Extensible Firmware Interface (EFI).

2.1 Why Itanium Failed?

EFI was implemented and adopted since the first generation of Itanium based platforms and remained the firmware interface on these machines to date. Itanium was supposed to be next generation 64-bit hardware platform, which should have eventually replace x86. Today, in 2016 we can conclude that this scenario didn't happen and it is unlikely to occur in the foreseeable future. There are couple of reasons why Itanium wasn't a success story. It is important to note here, that Itanium is radically different from x86 architecture. Itanium based processors are super-scalar, however means by which they achieve instruction level parallelism is not by employing multi-stage pipelines and out-of-order execution. In contrast, Itanium platform design is based in Very Long Instruction Word (VLIW) architecture. This approach makes processor circuitry simpler. Each instruction bundles multiple sub-instructions and compiler decides which sub-instructions maybe executed in parallel, thus processor itself doesn't need to do complex dynamic instruction scheduling at run-time. Instead, instructions are scheduled statically at compile time. This trait of VLIW based architectures makes very challenging to write good compilers for.[7]. Second reason why Itanium failed is rather pragmatic. Simply put, competition. In early 2000's another US based silicon company, Advanced Micro Devices (AMD) introduced 64-bit extensions for x86 instruction set and implemented them in their new K8 micro-architecture. This required very swift reaction from Intel, but because they took radically new approach with Itanium it was unavoidable that market adoption would be gradual, because of sheer number of legacy x86 applications in production. In a sense this was the final blow to the Itanium. Intel wanted to push new architecture, but at the same time it needed to respond to AMD's actions. In the end, Intel also introduced 64-bit extensions in their x86 processors and company ended up producing two 64-bit processor designs. Itanium adoption was still very slow and lack of good compilers and developer tools for Itanium platform lead to lack of interest from the software developer community which closed the death spiral. At any rate, it was because of Itanium why EFI and later UEFI formed and we why see shift from legacy BIOS to UEFI based platforms today.

2.2 From EFI to UEFI

As EFI was later also target for machines which were built on top of legacy x86 architecture it was getting momentum and industry awareness about this project. In mid 2000's more and more people were realizing drawbacks and scalability problems of a legacy BIOS based

firmware implementations. It was recognized that this problem should be discussed and addressed more widely and openly within the industry to prevent future problems similar to ones which vendors faced with BIOS, including but not limited to, BIOS being proprietary code, very difficult to change and extend. In 2005 a group of BIOS vendors, OS vendors and hardware companies formed UEFI Forum. Shortly after the formation of the organization, in 2006, they released first version of UEFI specification version 2.0. It was based upon former EFI specification which was contributed to the forum by Intel.

UEFI specification defines only interface for the firmware. To build and provide these interfaces there is still a lot of things to get done first. In order to alleviate some problems in implementation of interfaces mandated by UEFI specification, UEFI Forum produces also related document called UEFI Platform Initialization (PI) Specification. This document is concerned with internal details of firmware implementation. PI specification is again based on prior work done Intel. In addition to the specification documents there is also the open-source firmware implementation[1] available under the permissive BSD license.

3 UEFI Architecture

UEFI is a vast subject. Only UEFI specification has more than 2500 pages and there are related documents like UEFI Platform Initialization Specification which is of the same importance if one desires to understand the UEFI architecture. Nevertheless, we will try to keep next section relatively small, but still provide all information which will support our analysis of UEFI Secure Boot in chapter III.

3.1 UEFI Platform Initialization

In this section we will learn about UEFI Platform Initialization (PI). Note that there are two distinct meanings of UEFI PI which we should recognize. First, UEFI PI is a set of specifications produced by UEFI Forum. All these document together, when implemented, bring to the existence second meaning for UEFI PI term. Which is an actual process of platform boot with all its phases, interactions between them and other pre-Boot environment components, like third party option ROMs.

Also term platform initialization is sort of misleading as this subsection will not cover only initialization part of the boot process but all the phases of the process. Before we go any further, let us first discuss UEFI PI architectural diagram.

As shown in the figure I.1, platform boot up starts in Security (SEC) phase, then transitions to Pre-EFI Initialization (PEI) phase, follows on with Driver Execution phase (DXE). Finally, we get to the point when system firmware initialized platform to the state that we are able to pick boot device in Boot Device Selection phase (BDS) and start OS loader and eventually hand off control to an operating system. For details on each phase please continue reading the following sections.

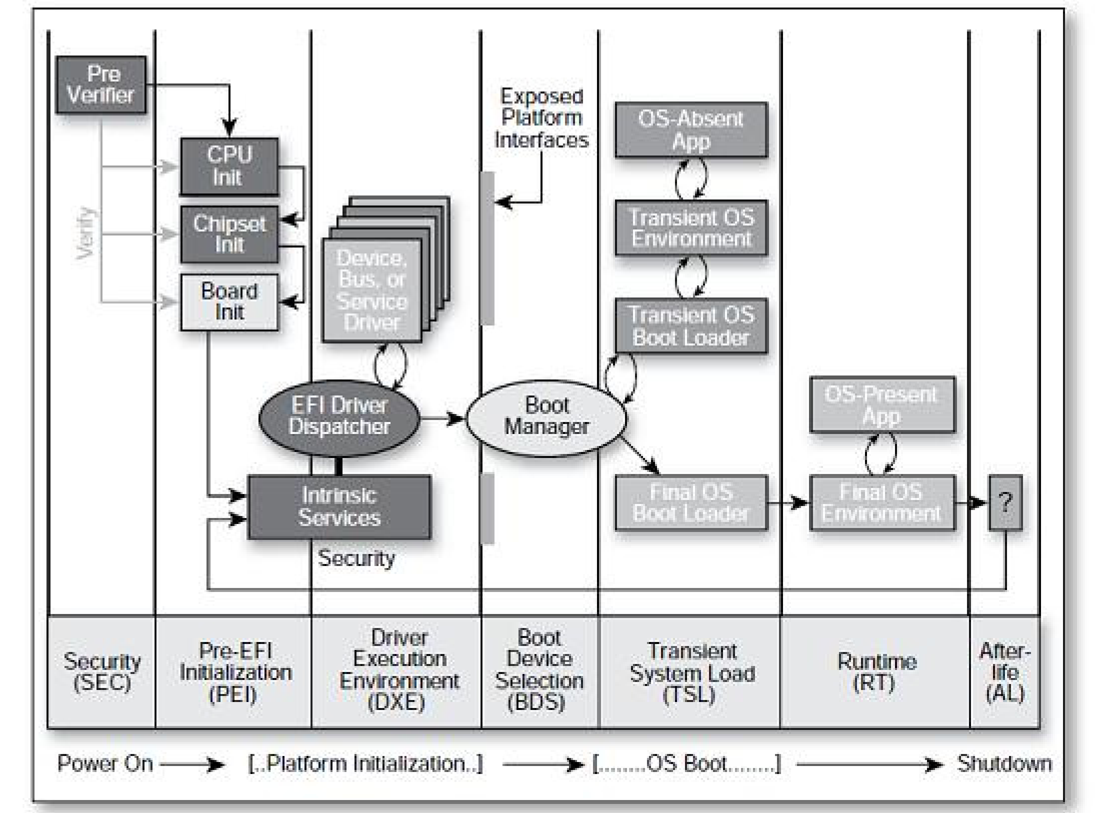


Figure I.1: UEFI Platform Initialization Phases

3.2 Security (SEC) Phase

This is the first phase of UEFI PI process and it is performed immediately after platform reset or power-on. Its main responsibility is to ensure the integrity of the platform firmware. Note that UEFI specifications are for the most part concerned with C interfaces and are platform independent, but this phase of UEFI PI is on the other hand very much platform and processor architecture dependent. This phase is generally implemented by minimum amount of code. From programmer perspective, he must deal with assembly language for given processor architecture. At this point we typically don't have access to main DRAM memory and processor is in mode where it fetches instruction directly from flash. There are couple of other things that this phase takes care of,

- Handling of all restart event
- Establishes temporary memory store
- Forms root-of-trust in the system
- Transfers control to Pre-EFI Initialization Foundation code

On picture I.2 we can see how does physical memory map looks like at the exit from the

SEC Phase[1].

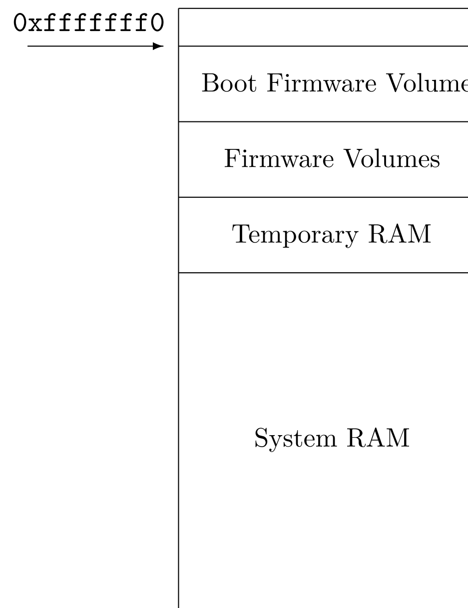


Figure I.2: Physical Memory Map

3.3 Pre-EFI Initialization Phase

After SEC phase is finished, platform transitions to the second phase of PI, namely PEI phase. The main goal of this phase is to initialize main system memory in order to support standard C run-time environment. This objective is more than obvious, because writing C code without ability to use DRAM memory is very difficult. Note however, that because previous SEC phase already initialized temporary memory store we have small amount of memory available to us. In most cases this would be processor cache which serves as RAM. Before going further, we shall summarize other goals of this phase. These include,

- Discovery of boot mode
- Launch firmware code which initializes main memory
- Determine location and start DXE core
- Transfer important platform information to the DXE core

Now let's discuss how is this phase realized in the firmware. There are two main types of firmware components responsible for implementing PEI.

- PEI Core

[1]Note that System RAM portion of the picture is only a placeholder, because DRAM memory is initialized by PEI phase

  - Core Dispatcher
  - PEI Services
- PEI Modules (PEIM)

PEI Core is the main executable implementing this phase. Its code is stored in Boot Firmware Volume (BFV). BFV is in turn stored on the flash storage device. Note that PEI Core code is mapped at some address range in physical memory, but itself doesn't reside in main DRAM memory, because memory subsystem was not initialized yet. PEI Core code is usually mostly written in C. PEI Core has two main components. First is Core Dispatcher. Its main responsibility is to locate PEIMs in the Firmware Volumes and execute them in the defined order. Next, major element are PEI Services. These are the common functions which can be used by all PEIMs. In following table 3.3 we summarize which particular PEI services there are a what are their basic functions.

Service	Basic Function
PPI	Database stored in TRAM which manages PEIM-to-PEIM (PPI) interfaces to allow intermodule calls between PEIMs
Boot Mode	Manages the boot mode (e.g. S3, S5) of the platform
HOB	Creates data structures called Hand-Off-Blocks that are used to convey information the DXE phase
FV	Helper service for Firmware File Systems (FFS) to find PEIM in flash
PEI Memory	Provides basic memory management routines
Status Code	Used for basic progress, error and debug reporting (e.g. via serial port)
Reset	Interface for initiating warm or cold reset

Table I.1: PEI Core Services

After PEI Core initialized PEI Services it will start PEI Core Dispatcher. It searches for PEIMs in Firmware Volumes and dispatches them in well specified order. Order in which PEIMs are dispatches depends on the definition of dependencies between PEIMs. Each PEIM contains dependency expressions which are logical expressions consisting of PPIs. They list PPIs which must be established before entry point of given PEIMs can be invoked by Core Dispatcher. In order to work out whether all dependencies are satisfied, Core Dispatcher will use PPI service from PEI Core Services. After PEI Core Dispatcher has no more PEIMs to execute it will exit and return control to PEI Core code. At this point main memory should be initialized, because it is responsibility of one of the PEIM to do so. PEI Core now transfers control to the DXE Initial Program Load (IPL) PPI in order to transition to the DXE phase.

We can see that most of the work by PEI phase is done by Core Dispatcher or rather by PEIM which are executed by the Dispatcher. Without going into too much detail, let's briefly look at what are some key characteristics of PEIMs.

- PEIMs are executable binaries

- They are built separately from PEI Core
- Code is not compressed
- Found in the Firmware Volumes
- Executed in place, directly from flash
- Defines interfaces which maybe used by other PEIM
- Each PEIMs defines set of PPIs on which it depends on

As we discussed earlier, each PEIM consumes some PPI and may produces PPI which maybe in turn a dependency of some other PEIM. There are two main types of PPI

1. **Architectural PPI** – GUIDs of these PPIs are known to the PEI Core code and they provide generic interface to PEI Services. They are defined for services which have platform specific implementation, e.g. advertising presence of main memory store
2. **Additional PPI** – They are important for interoperability but have no dependency with PEI Core.

PPIs are accessed through PPI database. This database contains entries about all accessible PPIs. First, author of PEIM declares the dependencies of the PEIM by creating logical expression which is generated by grammar from the appendix A.

Pre-EFI Module Dependency Evaluation

We discussed in previous section that each PEIM declares dependencies which must be satisfied in order for PEI Core Dispatcher to execute PEIM. Whether given PEIM has all it dependencies satisfied is evaluated using Dependency Expression (DEPEX) algorithm. Before we continue our discussion of how DEPEX works, let's first consider how are the dependencies defined. We mentioned in previous section that dependencies are expressed as logical formulas and those formulas are generated by the grammar. Here is an example of such formula,

```
EFI_PEI_CPU_IO_PPI_GUID AND EFI_PEI_READ_ONLY_VARIABLE_ACCESS_PPI_GUID
END
```

In this simple example we can see that PEIM declares two dependencies connected using logical AND. Hence, if both PPIs identified by respective GUID are retrieved using PPI Service, then this PEIM can be executed by PEI Core Dispatcher. Recall that PEIM are standalone pieces of executable code found within Firmware Volumes. DEPEX expressions are embedded in the PEIM binary so they can be evaluated in PEI phase. However expressions are not encapsulated into the executable as text strings. They get translated to the form of byte code which is then interpreted at run-time in order to evaluate given

Op Code	Instruction
0x02	PUSH <PPI GUID>
0x03	AND
0x04	OR
0x05	NOT
0x06	TRUE
0x07	FALSE
0x08	END

Table I.2: DEPEX Interpreter Instructions

logical expression. Byte code used to encode expressions consists of instructions shown in table I.2. Notation. Above logical expression gets translated to following byte code, `PUSH PUSH AND END`. Such an arrangement of two `PUSH` instructions followed by binary operation `AND` is analogous to expression written in Reverse Polish Notation. It is well known that stack based machines are well suited for evaluation of expressions in Reverse Polish Notation [5]. Indeed, this is exactly how is it implemented in the firmware, well at least in Intel's open-source implementation[1].

Transition to DXE Phase

Once there is no more PEIMs to be dispatched, PEI Core core will locate and execute PPI called DXE Initial Program Load which begins transition from PEI to DXE phase. On exit from PEI phase there is an information transfer between PEI phase and DXE phase. Information describing system state must be conveyed by PEI to DXE components. This transfer of system information is done using data structure called Hand-Off Blocks (HOB). System information is encoded in more than one HOB. HOBs are organized into a linked list called HOB list. This list must hold at least HOBs listed in table I.3

Summary

In previous section we introduced PEI phase of UEFI PI process. Main goal of this part of system boot up is to initialize main memory. PEI code executes in place from Boot Firmware Volume stored in flash memory. As a first step of this phase PEI Services are established. Then PEI core locates PEIMs stored within Firmware Volumes and dispatches them in well defined order. Ordering is evaluated using DEPEX algorithm. PEIMs may produces interfaces available for other PEIMs to use (PPIs). One of the PEIMs initializes main memory. As a final step of PEI phase, PEI code executes PPI named DXE Initial Program Load which transition execution to the DXE phase. Information about system state is stored in data structures called Hand-off Blocks. List of these structures is handed over to DXE on exit from PEI phase.

Required HOB type	**Usage**
Phase Handoff Information Table (PHIT)	HOB This HOB is required.
One or more Resource Descriptor HOB(s) describing physical system memory	The DXE Foundation will use this physical system memory for DXE.
Boot-strap processor (BSP) Stack HOB	The DXE Foundation needs to know the current stack location so that it can move it if necessary, based upon its desired memory address map. This HOB will be of type **EfiConventionalMemory**
Backing Store Pointer Store	The DXE Foundation needs to know the current store location so that it can move it if necessary, based upon its desired memory address map.
One or more Resource Descriptor HOB(s) describing firmware devices	The DXE Foundation will place this into the GCD.

Table I.3: List of necessary HOB types in the HOB list

3.4 Driver Execution Environment

This section will introduce Driver Execution Environment (DXE) phase of UEFI PI. In this phase majority of boot process is occurring. This include initialization of platform chipset, enumeration of devices and their initialization. Also, system data structures which support UEFI Services are established.

Similarly to PEI phase, also in DXE phase there are multiple components at play,

- DXE Foundation
- DXE Dispatcher
- A set of DXE drivers

The DXE Foundation is responsible for establishing UEFI Boot Services, UEFI Runtime Services and DXE Services. Dispatcher discovers and executes DXE drivers in a well defined fashion. Drivers initialize and configure attached hardware and produce architecture protocols (AP) which abstract DXE core from the platform specifics. This means that many DXE components can be written in hardware agnostic way. This is very desirable, because this allows firmware vendors to easily cooperate, share and integrate components created by others. Paradigm of firmware programming shifts from highly proprietary single BIOS code base for single platform to more collaborative development which allows for more participation because code doesn't rely on platform details but is built around standardized interface. With this paradigm shift, firmware engineers are able to adopt work-flow used by OS engineers. In next paragraph we will discuss contents of UEFI System Table, the core data-structure containing pointers to other tables maintained by firmware, i.e. UEFI

Boot Services and UEFI Runtime Services. It also provides access to console devices and related I/O protocols. System Table contains also pointer to additional data-structures, but we will omit describing those as they are not essential for our overview.

UEFI Boot Services Table

In table I.4 you can review some of the services exposed by Boot Services Table. Note that these APIs can be called from within DXE environment and after DXE phase exits, but only up to the point when some component executes method `ExitBootServices()` [2].

Service	Description
Task Priority	Services for incresing or descreasing current task priority level
Memory	Routine for memory management, allocation of 4KiB pages, memory pools aligned at byte boundary and functions for obtaining physical memory map
Event and Timer	Services providing ability to create, signal and close event as well as capability to wait for event and check its status
Protocol Handler	Used to add or remove handles to handle database
Image	Makes possible to load, unload, start and exit PE/COFF binary images
Driver Support	Set of services to connect or disconnect drivers to devices in the platform

Table I.4: Overview of UEFI Boot Services

UEFI Runtime Services Table

These UEFI firmware APIs are always available to call by all parties, including firmware, OS loader and OS itself. Next table (I.5) summarizes which Runtime Services are provided by UEFI firmware.

Service	Description
Variable	Services to look up, add and remove UEFI Variables
Real-Time Clock	Provides services to set and get current time and date
Reset	Routines to shutdown or reset a machine
Virtual Memory	Functions which allow for the DXE component to be converted from running on top of physical memory map to run on virtual address map.

Table I.5: Overview of Runtime Services

[2] There are machines out there where this is not the case. Most notably some Apple machines from 2011

It is important to note that both UEFI Boot and Runtime Services depend on their respective architectural protocols. Recall that architectural protocols (APs) are software interfaces implemented and published by DXE drivers. They provide ability to the DXE Foundation to implement Boot and Runtime Services in platform independent manner.

DXE Dispatcher

We've already briefly touched on the responsibilities of the DXE dispatcher but let's look at them in the greater detail. The dispatcher is DXE component which is responsible for discovering and executing DXE drivers. These drivers are modular chunks of firmware code stored in Firmware Volumes. UEFI PI specification requires complaint implementation of DXE dispatcher to support these functions,

- Driver execution is done in an order (using DEPEX)
- Support for predefined order (a priori file)
- Ability to define ordering dependencies between DXE drivers to support *emergency patch* DXE drivers
- DXE dispatcher must use facilities provided by platform dependent Security AP every time new FV is discovered or DXE driver is loaded

Clearly, there are similarities between how PEIMs are processed and how DXE drivers are dispatched. Both use DEPEX expressions to specify dependencies and DEPEX algorithm is used to compute weak ordering in which modules or drivers are executed. However, DXE dispatcher is also required to support predefined order of execution of DXE drivers. This fixed order is specified in a file called *a priori file*. Each firmware volume may contain at most one such file. If firmware volume contains a priori file, then all DXE drivers which are specified in this file are executed and then for the rest of the drivers DEPEX expressions are evaluated and drivers are dispatched in sequence given by DEPEX ordering. After all drivers are dispatched execution is transitioned to Boot Device Selection (BDS) architectural protocol. Its responsibility is to initialize console and boot devices and try to boot an operating system. In this phase as boot devices are discovered firmware may find out about additional firmware volumes. In cases that BDS is unable to get access to console or boot device it may transfer control back to the DXE dispatcher in order for it to find and execute ancillary DXE drivers stored in newly discovered firmware volumes. After the dispatcher executes all drivers it discovered, it transfers execution back to the BDS architectural protocol.

DXE Drivers

It should be very obvious by now that UEFI specifications are written in a way that allows for very modular firmware implementation. Specifically in the DXE phase the modules we deal with are DXE drivers. In this section we briefly describe what types of DXE drivers there are and what are their main characteristics. First let's look at categorization of DXE drivers. There are two main types of DXE drivers,

- Early DXE drivers
- DXE Drivers that follow the UEFI Driver Model

First category of drivers are so called *early* drivers. These will typically include code which initializes basic platform components such as processor and chipset. Another common responsibly of early drivers is to establish Architectural Protocols supporting all UEFI Boot and Runtime Services. Hence, author of early DXE driver must cope with the fact that in this phase of the boot process not all of the UEFI Boot and Runtime Services are available. Same applies for DXE services.

In the second class there are drivers which follow UEFI Driver Model as described in UEFI Specification 2.0 and in the later versions of the specification. These drivers don't define DEPEX expressions because they instead register Device Binding Protocol interface in the handle database. These bindings specify relation between some hardware present on the platform and respective firmware driver. Dependency expressions are not needed because registration of the binding doesn't require use of any Architectural Protocols.

As noted previously, some of the DXE drivers provide direct support for implementation of UEFI Boot and Runtime Services. We can then divide the drivers with respect to the type of the service they support. Hence there are,

- Boot service drivers
- Runtime drivers

Boot service drivers provide facilities which are available up until `ExitBootservices()` is called. After that, all system memory used by these drivers is released and made available for use by an operating system. DXE Foundation itself is also in this category. This implies that Runtime drivers may not depend on any UEFI Boot Service or DXE Service.

Runtime drivers provide service available before and after `ExitBootServices()` is called. All UEFI Runtime Services are produced by these drivers.

DXE Architectural Protocols

DXE Foundation code is platform independent, this is due to existence of Architectural Protocols. They have platform dependent implementation but their interface is mandated by UEFI PI Specification. They provide basic services needed to implement full range of UEFI Runtime and Boot Services and DXE Services. After DXE phase starts it is given list of Hand-off Blocks from PEI phase. In order to execute DXE drivers implementing Architectural Protocols, HOB list must at least contain System Memory HOB, describing usable system memory and HOB describing at least one Firmware Volume. Details of each Architectural Protocols are not important for our next discussion, with couple of exceptions, that is Security APs and Variable related APs because these protocols directly support Secure Boot. For the sake of completeness we merely list other APs that are defined by UEFI PI specification.

- Boot Device Selection AP
- CPU Architectural AP
- Metronome AP
- Monotonic Counter AP
- Real Time Clock AP
- Reset AP
- Security AP
- Timer AP
- Variable AP
- Variable Write AP
- EFI Capsule AP
- Watchdog Timer AP

Security Architectural Protocols

UEFI PI specification describes two security related APs, `EFI_SECURITY_ARCH_PROTOCOL` and `EFI_SECURITY_ARCH2_PROTOCOL`. These two protocols are implemented by security DXE driver and provide abstraction to policy actions taken upon image invocation. Before diving into specifics, let's look at how these two protocols are defined in the specification.

```
1  typedef
2  EFI_STATUS
3  (EFIAPI *EFI_SECURITY_FILE_AUTHENTICATION_STATE) (
4    IN CONST EFI_SECURITY_ARCH_PROTOCOL *This,
5    IN UINT32 AuthenticationStatus,
6    IN CONST EFI_DEVICE_PATH_PROTOCOL *File
7  );
8
9  typedef struct _EFI_SECURITY_ARCH_PROTOCOL {
10   EFI_SECURITY_FILE_AUTHENTICATION_STATE
        FileAuthenticationState;
11 } EFI_SECURITY_ARCH_PROTOCOL;
12
13 typedef
14 EFI_STATUS
15 (EFIAPI *EFI_SECURITY2_FILE_AUTHENTICATION_STATE) (
16   IN CONST EFI_SECURITY2_ARCH_PROTOCOL *This,
17   IN CONST EFI_DEVICE_PATH_PROTOCOL *DevicePath,
18   IN VOID *FileBuffer,
19   IN UINTN FileSize,
```

```
20    IN BOOLEAN BootPolicy
21 );
22
23 typedef struct _EFI_SECURITY2_ARCH_PROTOCOL {
24 EFI_SECURITY2_FILE_AUTHENTICATION FileAuthentication;
25 } EFI_SECURITY2_ARCH_PROTOCOL;
```

EFI_SECURITY_ARCH_PROTOCOL is used for checking authentication status of the file. It allows for execution of platform specific policy as a response to the file authentication result. Possible policies implemented by this protocol are e.g. locking the flash upon failure or mandatory attestation logging.

EFI_SECURITY_ARCH2_PROTOCOL this protocol is again used to abstract platform specific policy away from DXE Foundation. Example of the policy is to measure PE/COFF image before its invocation or authenticating image signature.

Variable Architectural Protocols

This protocol is responsible for providing services to get and set environment variables. Protocol must be produced by runtime DXE driver and can't be consumed by other DXE drivers. Only DXE Foundation is allowed to use interfaces produced by this protocol. Driver which implements this protocol must initialize following UEFI Runtime Services, GetVariable(), GetNextVariableName(), SetVariable() and QueryVariableInfo(). Note this protocol only performs minimal service initialization initialization, hence services are not fully available up until Variable Write AP is installed.

Variable Write Architectural Protocol

Protocol is again produced by runtime DXE drivers. Installation of this protocol may alter SetVariable() field of the UEFI Runtime Services Table. This protocol can be directly used only by DXE Foundation and no DXE driver can use facilities provided by this protocol. Instead, drivers that require write access to environment variables must include EFI_VARIABLE_WRITE_ARCH_PROTOCOL in their DEPEX expression. Once such driver is initialized it is assured that write protocol is already installed and environment variables can be set using standard UEFI Runtime Service.

Setting and obtaining values of UEFI variables

Our discussion in previous section introduced architectural protocols used to implement SetVariable() and GetVariable() UEFI Runtime Services. Familiarity with these two services is very important for analysis of Secure Boot protocol in the chapter III.

First we describe GetVariable runtime service and we start with its prototype.

```
1 typedef EFI_STATUS GetVariable (
```

```
2   IN CHAR16 *VariableName,
3   IN EFI_GUID *VendorGuid,
4   OUT UINT32 *Attributes OPTIONAL,
5   IN OUT UINTN *DataSize,
6   OUT VOID *Data OPTIONAL
7 );
```

Semantic of parameters should be obvious from the service declaration. Thus we simply skip to service description. Basic function fulfilled by this runtime service is obtaining value of variable specified by `VariableName` parameter. Also variable attributes can be retrieved if desired. Note that multiple vendors may define same variable name because name spaces do not clash due to use of `VendorGuid`. In case that supplied buffer is too small to save the variable content, error `EFI_BUFFER_TOO_SMALL` is returned and `DataSize` indicates actual size of variable data. Thus proper amount of memory can be allocated using e.g. `AllocatePool` UEFI Boot Service and attempt to obtain variable content can be retrieved.

Next we describe `SetVariable` runtime service. As it is customary, we start with prototype definition.

```
1 typedef
2 EFI_STATUS
3 SetVariable (
4   IN CHAR16 *VariableName,
5   IN EFI_GUID *VendorGuid,
6   IN UINT32 Attributes,
7   IN UINTN DataSize,
8   IN VOID *Data
9 );
```

As the name of the service suggests, its primary use case is for setting UEFI environment variables. Parameters of the service are self explanatory with exception of `Attributes`. We enumerate all possible flags here but we don't provide detailed description for each flag. To obtain those, please refer to [12]. However, in chapter III we cover what are the rules for updating variables that have `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE` flag set.

- `EFI_VARIABLE_NON_VOLATILE`
- `EFI_VARIABLE_BOOTSERVICE_ACCESS`
- `EFI_VARIABLE_RUNTIME_ACCESS`
- `EFI_VARIABLE_HARDWARE_ERROR_RECORD`
- `EFI_VARIABLE_AUTHENTICATED_WRITE_ACCESS`
- `EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS`
- `EFI_VARIABLE_APPEND_WRITE`

Environment variables are maintained by the firmware and can be made persistent across the reboots. Whether variable is made persistent or not depends on the value of `EFI_VARIABLE_NON_VOLATILE` flag. Note that non volatile storage used to store variables may have severely limited capacity. Variable can be deleted if value `DataSize` is zero. However storage occupied by variable may not be released even when it was deleted. Storage is reclaimed at latest on next power cycle.

4 Summary

This chapter provided basic introduction to the UEFI. We first covered bit of historical background and explained why efforts leading eventually to UEFI formed in the first place. Next we discussed selected stages on UEFI Platform Initialization process. We briefly described three Architectural Protocol that directly support functions implemented by Secure Boot Protocol. We deliberately omitted some details for the sake of brevity, but at the same time we hope that material presented is sufficient to support our discussion about Secure Boot Protocol in chapter III.

Chapter II

Public Key Insfrastructure and X.509

This chapter will provide more details on some of the technologies used by Secure Boot. We have mentioned previously that Secure Boot leverages public key encryption and hashing. In this chapter we will cover both topics. We also discuss X.509 specification.

1 Cryptographic hash functions

In this section we introduce necessary definitions used later in description of Secure Boot Protocol. If not stated otherwise all definitions are taken from [9]

Definition 1 *Hash Function H accepts a variable-size message M as input and outputs d a fixed size representation H(M) of M, sometimes called message digest. In general d will be much smaller than M.*

Definition 2 *Cryptographically strong hash function is a hash function which in addition has these properties*

1. *H can be applied to an argument of any size*
2. *H produces a fixed size output*
3. *H(M) is relatively easy to compute for given M*
4. *For any given d it is computationally infeasible to find M with H(M) = d*
5. *It is computationally infeasible to find any M, N such that H(M) = H(N)*

2 Public Key Encryption

Secure Boot may achieve authentication of UEFI images by using both, hashing and RSA public-key encryption algorithm. Optionally RSA key maybe encapsulated within X.509 certificate. We first introduce Public Key Encryption concepts and then we provide overview of RSA algorithm.

Classic cryptography achieves secrecy of a message by encrypting the message using key material which must be kept secret and is known in advance to all communicating parties. In contrast, public key crypto systems use different approach than traditional cryptography. Each user has its own key material and it is divided into two parts, one is kept secret and the other can be known to anyone else. This new idea was first introduced by Diffie and Hellman [4].

Public part of the key generates a public transformation E and private component generates a private transformation D. Note that symbolic names of these transformations do not imply encryption or decryption per se . This is because in public key crypto system we may have $D(E(M)) = M, E(D(M)) = M$, or both.

In order to support secrecy, then transformation of public key crypto system must satisfy $D(E(M)) = M$. Now, let's assume that A wants to send a secure message M to B. Then A must have access to the transformation E_b, which is public transformation of B. If A has access to this transformation it can encrypt message M and obtains ciphertext, $C = E_B(M)$. A now sends the encrypted message to B. Recipient B then uses his private transformation D_B to decrypt the ciphertext and obtain the message, $D_B(C) = D_B(E_b(M)) = M$. In case that in-flight message has been eavesdropped, then attacker can't decrypt the message however it can modify to ciphertext and resend it to B. Thus, integrity of the message is not guaranteed. Moreover since E_B is a publicly known transformation, then B has no inherent method to determine the sender of the message. Hence, authenticity nor integrity of the message are assured.

Public key cryptography is also used to provide an integrity of transmitted messages. To accommodate this function following must hold, $E(D(M)) = M$. Imagine that A wants to send a message M to B such that integrity of the message is guaranteed. Or to put it in another way, B is able to attest that message received from A was not altered in any way. To provide this functionality public key crypto systems use auxiliary hash function H. Using this function A creates message digest D of the message M, i.e. $H(M) = D$. Then using the transformation D_a is used to encrypt D and produces piece of ciphertext called digital signature S. Then signature is appended to message M and send to B. Recipient can check if message was altered by computing $H(M)$ and compare it with the value which is obtained by decryption of digital signature S. Note that message is transferred in plain text form, but it is not possible for an attacker to change the message without recipient noticing.

Nowadays, when public key cryptography is employed in communication we witness combination of possibilities described above. This way it provides secrecy and integrity guarantees to communicating parties. But in case of Secure Boot is public key cryptography used only to provide authentication, since there is no explicit communication happening.

2.1 RSA Public Key Encryption

In this section we introduce RSA algorithm. If not stated otherwise then presented information originates from [8].

RSA cryptograhic system is an abbreviation of names of its inventors, R. Rivest, A. Shamir, and L. Adleman. It provides secrecy and it is also used for creating digital signatures. Today it is among the most commonly used public key crypto systems. Its security is based on intractability of the integer factorization. Since algorithm itself consists of three distinct parts which are at the same time small enough and all can fit on the single page we decided to spell them out explicitly.

Algorithm 1 Key generation for RSA public key encryption

```
1:  procedure RSA_GENERATE_KEYS
2:      p ← random_prime()
3:      q ← random_prime()
4:      n ← pq
5:      φ ← (p − 1)(q − 1)
6:      e ← random_prime()                          ▷ 1 < e < φ, gcd(e, φ) = 1
7:      r ← (e − 1)φ^(e−2)
8:      d ← (1 + r ∗ φ)/e
9:      public_key ← (n, e)
10:     private_key ← d
```

Algorithm 1 is used by each communicating party to generate both public and private keys. Integers e and d in algorithm 1 are called the *encryption exponent* and the *decryption exponent* and n is called the *modulus*.

Algorithm 2 RSA encryption

```
1:  procedure RSA_ENCRYPT
2:      pubkey ← (n, e)
3:      Represent message m as an integer in the interval [0, n − 1]
4:      c ← m^e mod n
5:      Send the ciphertext to other party
```

Algorithm 3 RSA decryption

```
1:  procedure RSA_DECRYPT
2:      privkey ← d
3:      m ← c^d mod n
```

Algorithms 2 and 3 are used traditionally to ensure secrecy of transmitted messages. However, they can be also used to create and verify digital signatures. In case one wants to use them in this way he must compute hash of given message and encrypt it with a private key instead of a public key, i.e. $c \leftarrow m^d \bmod n$. Similarly, decryption of a digital signature is performed using a public key of a signer, $m \leftarrow c^e \bmod n$.

In previous sections we introduced basic concepts of public (asymmetric) key cryptography. We remarked what functions are provided by such crypto systems. Most notably secrecy and

integrity of a transmitted message. We briefly discussed what are the main characteristics of strong hash functions used in creation of a digital signature. As an example of such crypto system we presented RSA algorithm. It was an obvious choice, because this algorithm is also used by UEFI Secure Boot protocol for an authentication of UEFI applications.

2.2 Public Key Infrastructure

As we noted previously, Secure Boot achieves authentication of UEFI applications by employing asymmetric cryptography and hashing. In most cases today, consumer grade devices ship with crypto material already enrolled in their firmware. Almost always there are X.509 certificates present in the firmware. This is due to regulations enforced by Microsoft Corporation. If vendor wants to standardize his platform for Windows 8 or later version of Windows operating system he must comply with these regulations. We will briefly touch on this subject in the next section where we discuss crypto material databases in the firmware. At any rate, for our next analysis it is necessary to introduce concept of certificates as a corner stone of Public Key Infrastructure (PKI).

It was noted previously that in order to verify a digital signature one must posses a public key that corresponds to a private key used to create a signature. This requirement is crucial. Problem is that one would have to meet in person with other party in order to exchange public keys and verify each others identity and establish a trust relation explicitly. Unfortunately this is not realistic. In general, we can't allow for a public key to be exchanged using channel vulnerable to a man-in-the-middle attack. Otherwise, possible attacker could impersonate the other party and let us believe we are communicating with legitimate user.

In reality it is often not possible to establish a trust relation in person. However, this relation is very important in case we want to exchange sensitive information with a remote party. We need to make sure that the party we're communicating with is actually who they say they are. Exactly this problem is solved by using Public Key Infrastructure (PKI) and certificates.

Central component of PKI is a piece of data called *certificate*. Simply put, certificate ties together public key and *distinguished name*. Distinguished name is a name of a person or some other entity (e.g. company) which owns a public key and a corresponding private key. This binding is established by third party entity, in the PKI system, called Certification Authority (CA). We often say that a CA issues a certificate. Issuer of a certificate (CA), by signing the certificate, asserts that authenticity of the public key was verified and certificate itself can be trusted. However, certificates issued by a CA are trusted only as long as the CA itself is trusted. In order for a CA to issue a certificate it must be signed with the CA's private key. To validate a certificate we need to check a digital signature signed with a private key. To do that, we need to posses a CA's public key. Obtaining a public key via an insecure channel is problematic due to a possibility of a man-in-the-middle attacks. This issue is almost always workaround by pre installing a list of trusted CA's as part of an operating system or a web browser. In next section we discuss X.509 format used for storing certificates.

3 X.509

This section will provide some information about X.509 standard. Once again, X.509 and related specifications are very broad subject. Page count wise, we are again in realm of hundreds even thousands of pages. It wouldn't be practical to even try to cover such big volume of information in this thesis. However we try to pick relevant bits and pieces in a hope that they will make follow up discussion more clear.

3.1 History of X.509

X.509 is document published by Telecommunication Standardization Sector of International Telecommunication Union that specifies standard data format for exchanging public key certificates. This specification document also defines structure and semantics of some other, but related data formats, e.g. Certificate Revocation Lists (CRL). Published specification has character of a recommendation. However, the same document was also published as an ISO standard[6]. Fortunately, there is RFC 5280. This RFC conveys all essential information about X.509 v3 and about some standard extensions that are of particular importance for the Internet community. Rest of the section presents information based on this memo.

RFC 5280[3] specifies X.509 v3. That is third version of X.509 format. Original X.509 specification was published by ITU-T in 1988 as a part of X.500 directory services recommendations. In 1993, was the first version of a format revisited, more fields has been added, resulting in X.509 v2. By 1993 it was also obvious that X.509 as specified by respective versions 1 and 2 is not extensible enough. This realization was supported by prior deployment experience with Privacy Enhanced Mail (PEM) as PEM was built on top of X.509 specification. Specifically, more field were needed to carry information that PEM design and implementation experience had proven necessary. As a consequence of this requirements, the ISO/IEC, ITU-T and ANSI X9 developed the X.509 version 3 (v3) certificate format. Version 3 of X.509 introduces possibility to provision additional extension fields. Specification of new version of the format was finished by 1996.

3.2 Format of X.509 certificate

Certificates are issued by CA's to entities (subjects). Subject maybe a person or a system. After a certificate is issued it binds a public key and an identity of the certificate's subject. Subject is an entity to whom certificate was issued. This binding is asserted by a CA. A CA may base this assertion upon technical means (proof of possession through a challenge-response protocol), presentation of the private key, or on an assertion by the subject.

Format of the certificate is defined by X.509 standard. Certificate also contains the distinguished name of the certificate issuer (the signer), an issuer-specific serial number, the issuer's signature algorithm identifier, and a validity period.

Each X.509 certificate contains following information. Picture II.1 exhibits structure of a

X.509 certificate. Also, see appendix B for an example of X.509 certificate.

- version
- serial number
- signature
- issuer
- validity
- subject
- subject's public key information
- unique identifier of the issuer (optional)
- unique identifier of the subject (optional)
- extensions (optional)
- certificate signature algorithm
- certificate signature

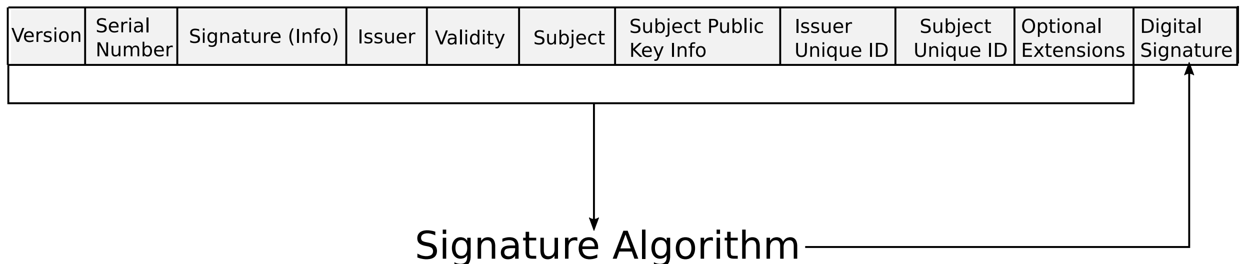


Figure II.1: Basic structure of X.509 v3 certificate

3.3 Certification Authorities

A CA is an organization or a company that issues certificates. One of the main responsibilities of the CA is to ensure legitimacy of issued certificates. CA must guarantee that each issued certificate contains a public key of a party that claims to have issued it. Same entity also must posses a corresponding private key.

We generally recognize two types of CAs,

- Public CA
- Private CA

A public CA such as VeriSign or Thawte provides services to general public and therefore must be trusted by the public. Second kind are private CAs. Usually, a private CA is setup within a given organization (e.g. company or university) and issues certificates only to members of the respective organization. Such CA is usually not trusted by people outside of an organization.

Recently we've witnessed genesis of an another type of a CA. Traditional public CAs charge money for their services. This is understandable, as operating supporting infrastructure is a costly enterprise. However, business model of public CAs was often criticized for being too profit oriented. At any rate, in 2014 there was publicly announced that new CA will be deployed on the Internet. This CA will be radically different from other public CAs in a way it operates. This project/CA is called *Let's Encrypt*. This public CA issues certificates for Internet DNS names, i.e. certificates used to setup `HTTPS` connection between web client and web server. Novelty is method of operation which is completely automated, supported by software agents on side of the CA and the subject[2]. Basic idea is that when a subject requests a certificate for `example.com`, then he must prove to a software agent running on the CA side that it controls both the private key and the respective domain. It must complete series of challenges, which include provisioning domain name under `example.com` domain and decryption of a random nonce generated by the CA, encrypted by the subject's public key. If a subject can complete challenges successfully, then a CA software agent may issue a certificate. Another novelty is a price for getting a certificate. Services provided by Let's Encrypt CA are completely free of charge.

In case of a public CAs it is necessary to ensure that they are publicly trusted. In order to achieve this, public keys of many public CAs are shipped within software (e.g. web browser, email client) that wants to use PKI. Often it is allowed for the end user to extend a database of trusted third parties and include to the trusted database a CA certificate of a private CA set up for use within some organization.

3.4 Certificate validation and certification path validation

In general, user who wants to take advantage of security related service provided by public key cryptography needs to acquire a valid certificate containing required public key. If a user doesn't already own a certificate and an associated public key of a CA which signed the former certificate, then he needs to acquire that certificate as well. We can conclude, that it maybe necessary to take possession of multiple certificates. User is effectively creating list of certificates he needs to verify before he can assume that a certificate he started with is trustworthy. We are saying that he is building certification path. Along this certification path user may encounter different types of certificates

- CA certificates
  - Cross certificates
  - Self-issued certificates
  - Self-signed certificates
- End entity certificates

A CA and an end entity certificates can be recognized from each other by looking at a specific extension defined by X.509 v3, specifically `X509v3BasicConstraints: CA: TRUE`. Naturally, CA certificates indicate that certificate assert binding between public key and CA identity. End entity certificates are issued to subjects that are not authorized to issue certificates. Cross certificates captures trust relation between two CA's. Self-issued certificates are CA certificates such that issuer and subject are the same. Self-signed certificates are usually at the end of certification path.

Going into details of a certification path validation is out of scope for this thesis, partly because in case of Secure Boot is a validation very minimal and signatures are checked only against certificates directly enrolled into a firmware, where each of the certificates is regarded as trust-anchor. However, certification validation was previously studied and [?] provides useful recommendations particularly for implementors of path validation algorithms.

Multiple criteria must be audited for each certificate along the certification path. Certificate must pass at least following checks in order for it to be considered valid,

1. Syntax check - syntax of all certificate fields as well as all extensions must be checked and pass syntax verification
2. Integrity check - check of the digital signature
3. Validity period check - guarantees that certificate is already valid and not expired yet
4. Revocation check - certificate must not be revoked
5. Criticality check - all certificate extensions which are marked critical must be supported by party doing certificate verification
6. Key usage check - all certificates on the path except an end entity certificate must have `keyCertSign` indicator set in their `keyUsageExtension`

3.5 Certification Requests

In order to obtain X.509 certificate from a CA it is necessary for the subject to present his public key which should be included in a certificate. Most commonly leveraged data format used to present necessary information for a CA to issue the certificate is Certification Request format defined by PKCS#10 specification. This format is also defined by informational RFC 2986[10]. Next paragraph captures the high level overview of the process.

Certification request includes a public key, a distinguished name of a subject and a set of attributes. Certification request is then signed by the subject's private key. Certification request is handed over to CA for signing. Upon CA's decision to issue the certificate is Certification Request transformed by the CA to a form of X.509 certificate that incorporates respective public key.

Request for certification is created using following three step process,

1. A `CertificationRequestInfo` value containing a subject distinguished name, a subject public key, and optionally a set of attributes is constructed by an entity requesting certification.

2. The `CertificationRequestInfo` value is signed with the subject entity's private key.

3. The `CertificationRequestInfo` value, a signature algorithm identifier, and the entity's signature are collected together into a CertificationRequest value

Upon receive of a request. A CA must first authenticate an entity that is requesting a certificate. Next, a CA must verify a digital signature of a request. If a CA decides to issue a certificate, then to actually create the X.509 certificate it uses information provided in a request as well as its distinguished name, serial number chosen the by CA, validity period and a signature algorithm.

3.6 Certificate Revocation Lists

A certificate is issued with an expectation that it will be used for its entire period of validity. However, there are situation that can cause invalidation of a certificate before certificate's expiration date. These may include occasions like change of name, change in relation between a subject and a CA, e.g. termination of an employment or a compromise of associated private key. A certificate doesn't retain its validity also in cases when a compromise of a private key is only suspected. When any of the events occur then a CA must revoke a certificate.

X.509 specification defines a method for certificate revocation. A scheme defined by a specification mandates that each CA must regularly (e.g. hourly, daily or weekly) issue a signed list of revoked certificates. This list is embedded into a data structure called Certificate Revocation List (CRL). A CRL is a time stamped list of serial numbers of revoked certificates issued by a CA and it is available in a public repository. Thus, if a user of a certificate wants to verify the certificate validity, then not only he has to check the certificate expiration and issuance date and a digital signature of the certificate, he has to also check some recent CRL and make sure the certificate is not on that list. A serial number of a revoked certificate can not be deleted from a CRL after it appears at least on one regularly issued CRL issued after certificate expiration.

Obviously, above a revocation scheme has some advantages but also serious drawbacks. Main advantage is a possibility to distribute CRLs via non trusted channels, same way how certificates are distributed. And here list of advantages pretty much ends. Big limitation is burden imposed on a certificate user. Before trusting a certificate's public key, he must be reasonably sure that he checked certificate's serial number against a recently issued CRL. Another problem is that downloading possibly big CRL can impose too much overhead from multiple reasons, e.g. storage or bandwidth limitations on a client side. Another problem is that even when user of certificate checks certificate serial number against CRL, still there is a possibility of a security incident in cases when a private key was stolen or otherwise compromised. This is due to a fact that CRLs are issued regularly and not immediately after a CA got a report about a disclosure of a private key. Hence, there is a time window

during which a certificate is known to be not trustworthy, but an updated CRL was not released yet. CRL scheme turned out be not practical with serious limitations.

To counter some of those impediments new protocol that can be used instead of checking CRL or as a auxiliary source of information was proposed in RFC 6960[11]. This RFC describes Online Certificate Status Protocol (OCSP). Protocol can be used for an online checking of a certificate revocation status. OSCP can be used to provide an information about certificate revocations in more timely manner. Client also queries the OSCP responder only for a revocation status of one particular certificate, hence OSCP also alleviates bandwidth issues for some clients.

Chapter III

UEFI Secure Boot

In addition to constructing a well defined and extensible pre-OS environment, UEFI was also designed to enhance security of a bootstrap process. UEFI defines a set of fundamental services that can be used to create sophisticated security solutions, e.g. user identification, platform key management. In third chapter of this thesis we focus on one of these security related technologies, on Secure Boot.

Secure Boot is a term used by UEFI specification. However, this term doesn't imply any inherent security of the boot process. As with any security technology, good understanding of implications and proper administration are necessary to actually enhance platform security by employing Secure Boot. Thus, we think that name Secure Boot was not a very good choice for this technology. Better term would something like „Image Authentication Protocol", because this is basic idea behind the Secure Boot. Secure Boot is a UEFI specification defined protocol and its main purpose is to extend chain of trust from the firmware all the way up to an operating system by authenticating every binary component of a boot chain, before it is executed. This way user can be sure that platform which has Secure Boot enabled may never load and execute untrusted[1] boot loader or operating system. Or any other UEFI application for that matter. Now when we understand what is Secure Boot good for, let's look out how this technology achieves its goals.

1 Key Databases

Main idea behind Secure Boot is really to use public key cryptography in order to run a boot loaders and operating systems which were signed with an authorized key. Naturally, it is necessary to somehow maintain list of such keys. UEFI specification mandates a way how is this task done on UEFI compliant firmware. There are five key databases maintained by a platform firmware. Each key database is maintained as non-volatile UEFI variable. Note that despite the fact that specification primarily talks about asymmetric keys, they are very rarely enrolled in a firmware directly. Instead they are encapsulated in X.509 certificates. Key databases also form a hierarchy. This hierarchy governs how can one update content

[1] How trust relation is established is discussed later in the chapter

of the respective database. Generally, each database update must be signed by private portion of the key enrolled in higher level database. Picture III.1 illustrates relationship between the key databases. Next enumeration provides the list of various key databases[2] and following paragraphs will describe each in more detail.

- Platform Key (PK)
- Key Exchange Key (KEK)
- db (db)
- dbx (dbx)
- dbt (dbt)

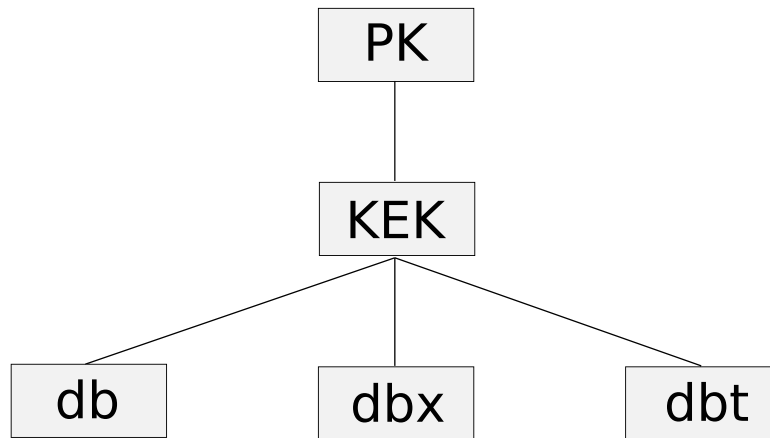


Figure III.1: Hierarchical structure of Secure Boot key databases

Platform Key (PK) is a key database which actually contains at most single key.[3]. As a name suggests, whoever controls key in PK database is de facto platform owner. He may not posses HW itself, but can govern what type of software can be run on the machine. In most cases, consumer grade machines sold today comes with UEFI firmware and PK containing certificate issued by platform manufacturer.

Key Exchange Key (KEK) is next key/certificate database and is used to update contents of the db database. Database is stored in KEK variable.

db key database contains X.509 certificates, RSA-2048 keys or SHA-256 hashes. Primary purpose of this database is to store crypto material against which all[4] binaries executed by a firmware are checked.

dbx database stores revoked certificates or SHA-256 hashes of binaries which are not allowed to run on the platform. Also it can store hashes of revoked certificates.

[2]In parentheses we provide name of respective UEFI variable holding content of given database

[3]At time of this writing there are UEFI firmware implementations (e.g. OVMF) which allow for more than one key stored in PK database due to a software bug

[4]This is actually a lie, because for binaries that are stored in Firmware Volumes firmware doesn't check any signatures

dbt database contains Timestap Authority (TSA) certificates. Certificates from this database are used to check time of certificate revocations. If we find during Secure Boot verification that given certificate is in dbx, i.e. is revoked firmware also checks if timestap associated with this digital signature was created after the time associated with TSA certificate. If yes, then verification fails, because signature was created with a certificate that was already revoked.

2 Secure Boot Modes

In previous section we covered types of key databases used by Secure Boot implementation. Content of these databases governs what Secure Boot Mode is currently active. There are four possible Secure Boot modes,

- Setup Mode
- User Mode
- Deployed Mode
- Audit Mode

In figure III.2 you can see mode transitions which are caused either by manual configuration of a physically present user or by enrolling a specific key in a key database. Now, let's describe each mode in greater detail.

Setup Mode is Secure Boot mode providing no security measures. If firmware detects this mode then it will not perform any authentication before launching UEFI driver or an application. This means that if a platform is in the Setup Mode then Secure Boot is effectively disabled. There are two ways how can user put a platform to the Setup Mode. First is a platform specific method, usually implemented as an option in a setup utility. Second way how to transition to the Setup Mode, is clearing content of the PK database, i.e. enrolling empty Platform Key. Also, while platform is in this mode, then updates of key databases do not strictly enforce rules for updates of authenticate variables.

User Mode is exactly in opposition to the benevolence of the Setup Mode. In case that a firmware is running in the User Mode then all UEFI binaries will be checked against the contents of key databases. If check fails then corresponding UEFI Boot Service `LoadImage` returns error `EFI_SECURITY_VIOLATION`. System can be transferred back to Setup Mode by deleting content of PK signature database.

Deployed Mode is recently[5] added Secure Boot mode. It is the most secure mode. If firmware is in deployed mode then it is not possible to issue updates to key databases. Only key database which can be updated in Deployed Mode is PK database. By clearing PK database platform automatically transitions back to Setup Mode. In case that administrator would like to perform updates to other key database, he needs to first transition platform

[5] in UEFI 2.5 released in 2015

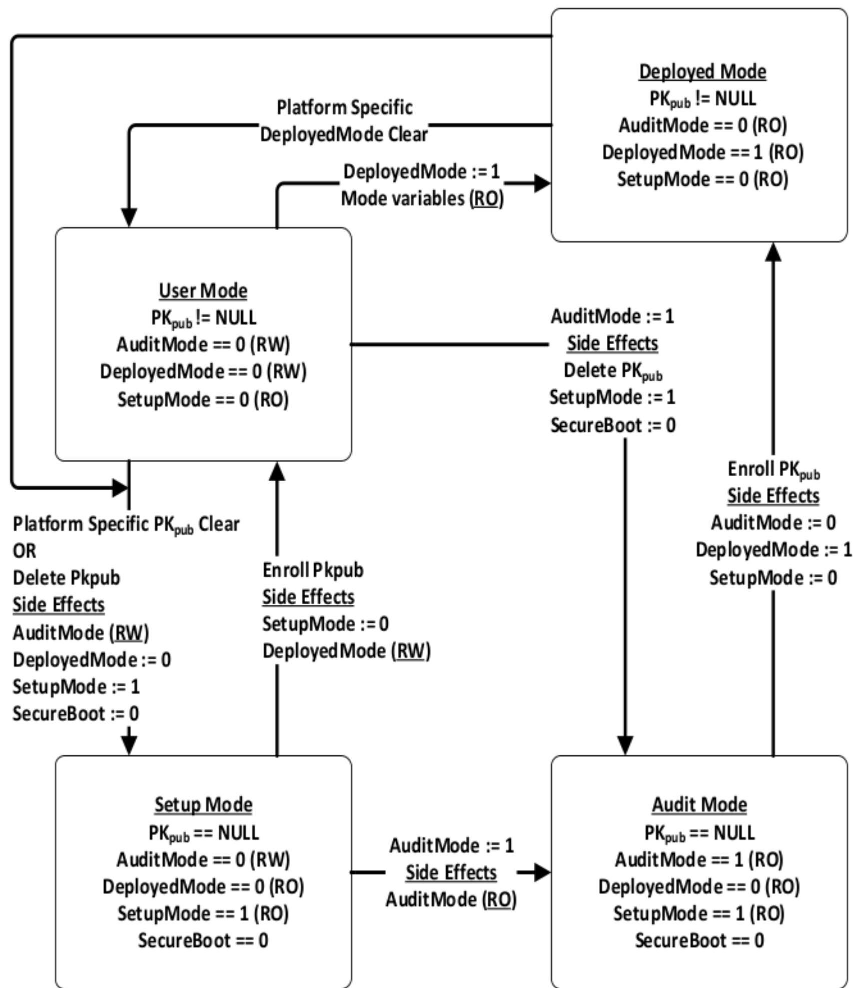


Figure III.2: Secure Boot mode transitions

to User Mode. This is done via some platform specific method available only to physically present user. Most likely is such method implemented as an option in setup utility.

Audit Mode enables programmatic discovery of signature list combinations that successfully authenticate installed EFI images without the risk of rendering a system unbootable. Chosen signature lists configurations can be tested to ensure the system will continue to boot after the system is transitioned out of Audit Mode. After transitioning to Audit Mode, signature enforcement is disabled such that all images are initialized and enhanced Image Execution Information Table (IEIT) logging is performed including recursive validation for multi-signed images.

3 Secure Boot Key Database Updates

All databases where firmware stores cryto material are maintained as UEFI Runtime Variables. We learned in chapter I what runtime service is provided for updating variables. All updates to UEFI variables must be performed by means of calling `SetVariable` UEFI Runtime Service. Recall, that behavior of the service is also governed by attributes of a particular variable we would like to set. In particular, all Secure Boot related variables are so called *authenticated variables*, i.e. all attempts to set be formatted according to rules defined by `EFI_VARIABLE_AUTHENTICATION_2` descriptor[12], page 245. We will described update process in detail in the next chapter where we discuss some implementation aspects of the command line utility that was implemented as part of this thesis.

4 UEFI and Secure Boot in Fedora Linux distribution

In this section we provide a survey of current state of support for UEFI booting and Secure Boot in Fedora distribution. In particular we discuss default solution provided by the distribution to users who wants to have Secure Boot protected boot process. Then we turn our attention to tools available for managing contents of Secure Boot databases.

4.1 UEFI support

Boot loader installation is a very last step of installation process when provisioning new Fedora machine. By default, grub2 boot loader is installed. Note that Anaconda installer doesn't provide a way how to pick alternative boot loader. During a system installation, installer detects that it is running on system which supports UEFI and it will the configure boot loader accordingly. If a firmware doesn't support UEFI then installer needs to install boot loader using legacy scheme. Also disk partitioning is different if UEFI is not available.

Legacy Boot

Bootstrap procedure of a platform with legacy (non UEFI) firmware start's with execution of CPU instructions located at reset vector, physical address `0xFFF0`. Platform manufacturer must make sure that when CPU jumps to this address and starts executing instructions it actually runs BIOS code. Legacy firmware after it finishes POST check and basic initialization will recognize attached *boot hard drive* and will load first 512 bytes long sector from this drive, also called *Master Boot Record*. This sector contains partition table and boot loader code. Firmware then relocates executable boot loader code to the system memory and jumps to boot loaders code, i.e. hands over control to the loader. Also partition table can have only four entries which is severely limiting on modern systems.

After firmware launched boot loader then it is boot loaders job to pick an operating system to start as per loader's configuration or as chosen by user of the machine. It is very challenging to write quite sophisticated piece of software as grub2 boot loader clearly is,

and at the same time make it so that its code is only 446 bytes[6] is size. Therefore, virtually all modern boot loader utilize two stage design. Boot sector then contains only first stage of the loader. Its responsibility is to locate the second stage, relocate its code to the main memory and start it. Second stage then implements rest of the process. Main function of the second stage is to locate, link and execute an operating system image. In most cases today, a second stage of a loader also loads an initial ram disk to a system memory. Main steps of a process exhibits figure III.3.

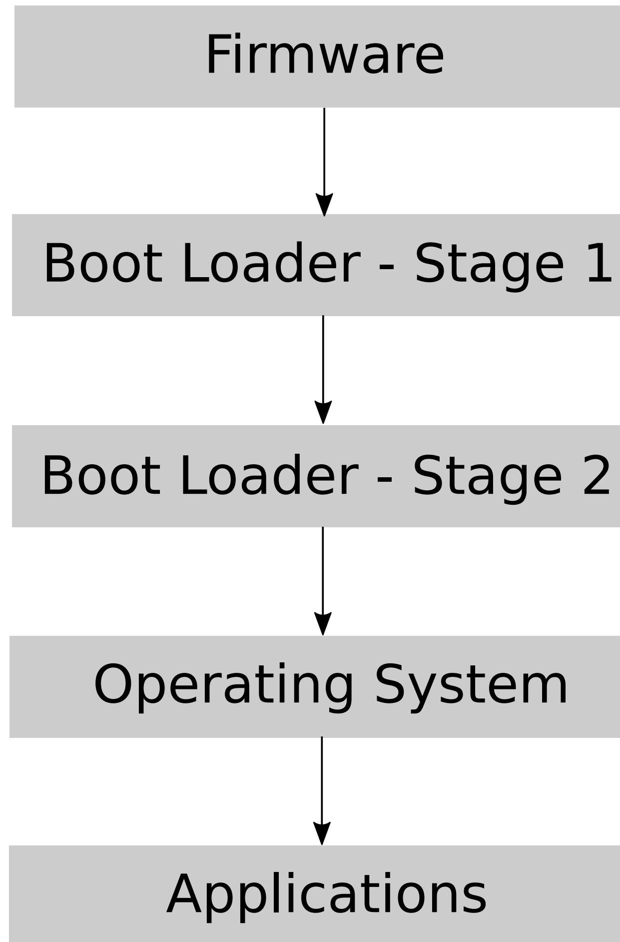


Figure III.3: Schematic diagram of legacy boot process using two stage boot loader

UEFI Boot

If platform firmware implements the UEFI Specification, then boot sequence looks a bit different from the legacy one. In particular new partitioning scheme is used and we no longer need boot loader to consist of two stages.

If we boot under UEFI firmware, then our boot partition is identified by GUID identifier in GUID Partition Table (GPT). This partition table layout is the new standard way how to store partition table information on disk. New scheme is standardized by UEFI

[6] 512 bytes minus place occupied by partition table

specification. However details of this specification are beyond the scope of this thesis. Interested reader should consult chapter 5 of UEFI specification [12]. Suffice to say that using GPT partition table has advantages over MBR partitioning scheme, probably the most important one being number of possible partitions in the table. With a GPT partition table a user is capable of defining up to 128 partitions.

When a boot sequence is run on top of UEFI firmware then there is no need for two stage boot loader architecture. In Boot Device Selection (BDS) phase, UEFI Boot Manager will spawn boot loader just like any other UEFI application. In fact there is no need for boot loader whatsoever. Linux kernel when compiled with option `CONFIG_EFI_STUB=y` can be started directly by a firmware as a regular UEFI application.

4.2 Secure Boot support

Hopefully, it is clear to the reader of this thesis by now that Secure Boot is a useful technology which can mitigate some pre-OS threats as well prevent running untrusted or maliciously modified kernel of an operating system. However, to actually make use of Secure Boot we need to configure it first. At the minimum, one needs to do these three steps to take advantage of Secure Boot,

1. Enable Secure Boot if it is disabled
2. Initialize content of signature databases
3. Sign boot loader and operating system kernel

Note that only steps two and three need a support from tooling provided by your operating system vendor. While doing survey of tools that are currently available in Fedora distribution we found out that there are no easy to use tools readily available to system administrators for managing Secure Boot signature databases. Only set of tools available in Fedora today is set of utilities provided by `pesign` rpm package. In order to make use of those tools a user has to have a good understanding of Secure Boot. We think that in order for Secure Boot to be actually used in user's favor, it is crucial to provide good and easy to use tools as well as quality documentation with examples of most common deployment scenarios. After all, ideas behind Secure Boot are not that radically different from, for example use of asymmetric cryptography in HTTPS communication or in SSH protocol. Nevertheless, we assert that despite the similarities, Secure Boot slow adoption can be partly attributed to unsatisfactory user experience provided by the currently available tools.

Chapter IV

Implementation of Secure Boot support in systemd-boot

systemd project is together with the Linux kernel core component of every major Linux distribution. systemd is first and foremost init system and service manager, but under umbrella of systemd project is a lot more components than meets the eye. For example systemd-boot is not well known and many people have never heard of it[1].

systemd-boot is a boot manager. Boot manager in general is a piece of software which is from users perspective equivalent to a boot loader. This is because all popular boot loaders today, e.g. grub2, Lilo or Microsoft's Bootmgr incorporate also functionality of a boot manager. Boot manager is a software component that is used to manage boot configuration and when invoked it parses a configuration and usually presents this information to a user in a form of a boot menu. User then selects an operating system he would like to start and a manager hands over control to the loader. Implementation of a loader may vary, it could be completely separate component or implemented as a subroutine within a manager.

Recall from previous chapter that a main job of a boot loader is to locate and execute an operating system. Achieving this goal that can be stated in one simple sentence is not a simple task by any stretch of an imagination. Conventional boot loader must understand a filesystem used for a boot partition. Today it is not unlikely to see `/boot` formatted using an advanced filesystem as `ext4` or `xfs`. Hence a boot loader must implement a filesystem driver. Filesystem handling is not the only problem, display handling, console handling, serial console handling are only some of the issues that needs attention of boot loader's developers. On legacy systems, all this functionality must be implement by a boot loader because BIOS either doesn't support needed functionality at all, e.g. a filesystem driver or doesn't provide any standard interfaces that could be used by a boot loader, e.g. serial console handling.

In contrast, systemd-boot doesn't need to implement any of the functions mentioned above, because it uses only interfaces defined by UEFI specification and implemented by a com-

[1] Author of the thesis is regular contributor to systemd project for almost four years now, as well as one of the maintainers of systemd package in the Fedora distribution

plaint firmware. This allows for very minimal implementation[2]. Note that systemd-boot only supports UEFI based systems and has no support of a legacy boot.

We decided to implement tool for managing secure boot signature databases under umbrella of systemd project because couple of reasons,

- systemd is available on all major distributions
- Tools shipped within systemd project are available right after system installation or even as part of an installer environment
- systemd project already incorporates UEFI boot manager systemd-boot

1 CLI tool systemd-secure-boot

Practical outcome of this master thesis is an implementation of command line tool for management of Secure Boot signature databases. Before going into details of the implementation, let's review what options does tool provide. See table IV.1, for a brief overview.

Option	Purpose
status	Display current Secure Boot enablement state and mode
generate	Generate set of self-signed CA certificates
sign	Given X.509 certificate, tool creates authenticated update
enroll	Install certificate into a signature database
lockdown	Enrolls set of certificates, effectively prevents running untrusted code
list	List contents of signature databases
export	Save the current content of signature databases

Table IV.1: List of possible actions provided by `systemd-secure-boot`

1.1 Determining current Secure Boot state

`systemd-secure-boot` implements this functionality in function `get_secure_boot_mode`. This function queries current Secure Boot state by obtaining values of following UEFI Runtime variables

- PK
- AuditMode
- DeployedMode

[2]systemd-boot has only little over 3000 lines of C code

- SetupMode

Tools gathers this information by reading files from special pseudo filesystem *efivarfs*. This filesystem provides read/write access to UEFI variables. Filesystem driver in the kernel then handles calling appropriate UEFI Runtime Service. `efivarfs` filesystem is automatically mounted by systemd at boot in `/sys/firmware/efi/efivars/`.

1.2 Creating certificates

In order to take advantage of Secure Boot one must populate contents of signature databases with valid content. As we previously mentioned, it is possible to have various types of crypto material in a signature database, i.e. X.509 certificates, RSA keys or hashes. Most commonly we see that X.509 certificates are being used. All modern Microsoft Windows certified hardware is shipped by manufactures with set of X.509 certificates already enrolled. Thus we decided to also implement a support for generating X.509 certificates and corresponding RSA signing keys into our tool.

For example, in order to generate certificate and signing key suitable as Platform Key one must execute `systemd-secure-boot -s '/CN=Michal Sekletar CA/' -d PK generate`. Tool will generate RSA key pair and it will embed public key into Certification Request and sign this request with private key. Signing process will generate self-signed X.509 CA certificate. Generated certificate can then be enrolled into firmware. This signing key can then be used to sign updates to KEK signature database. In case that `-d` is omitted tool will automatically generate set of keys and certificates, one for each signature database. All generated certificates are self-signed CA certificates and thus regarded as trust-anchors. Note that currently, signing keys are stored in plain text form and thus must be generated and retained in location available only to an authorized user.

As currently implemented, generate option provided by the tool can be useful mostly to regular users. Use in data center or by large organizations is unlikely, because of storing signing key in plain text in a filesystem probably violate security guidelines in many organizations. As a future improvement we propose use of PKCS#11 standard, generating and storing signing keys offline in Hardware Security Module (HSM).

However, we think that in combination with additional security measures it is possible to achieve satisfactory level of security of signing keys. For example, author of the thesis stores his Secure Boot signing keys on encrypted USB thumb drive, stored in undisclosed location. Once keys are enrolled and Secure Boot is enforced, then author sets password for firmware setup utility, thus Secure Boot can't be deactivated by unauthorized user who has physical access to the machine[3], e.g. hotel room cleaning lady.

1.3 Updating authenticated variables

Once user prepared content of key databases, next natural step is to enroll this generated content into the firmware. Actually this is primary use case for `systemd-secure-boot`

[3] Laptop in author's case

tool. In order to implement this functionality, we need to first address multiple issues, first being data format expected by a firmware when enrolling new certificates, second is update process for authenticated UEFI variables.

Format of enrolled data

UEFI specification mandates data format for key databases. Firmware, also expects that newly written value of some authenticated variable, e.g. PK is also in this format. Specification defines following data structures,

```
typedef struct _EFI_SIGNATURE_DATA {
  EFI_GUID SignatureOwner;
  UINT8 SignatureData[0];
} EFI_SIGNATURE_DATA;
```

```
typedef struct _EFI_SIGNATURE_LIST {
  EFI_GUID SignatureType;
  UINT32 SignatureListSize;
  UINT32 SignatureHeaderSize;
  UINT32 SignatureSize;
  // UINT8 SignatureHeader[SignatureHeaderSize];
  // EFI_SIGNATURE_DATA Signatures[...][SignatureSize];
} EFI_SIGNATURE_LIST;
```

Every enrolled certificate corresponds to one `EFI_SIGNATURE_LIST` structure[4]. Signature database usually contains more than one certificate - signature list. In figure IV.1 you can see structure of a signature database. In our case signature list contains exactly one signature, that is X.509 certificate.

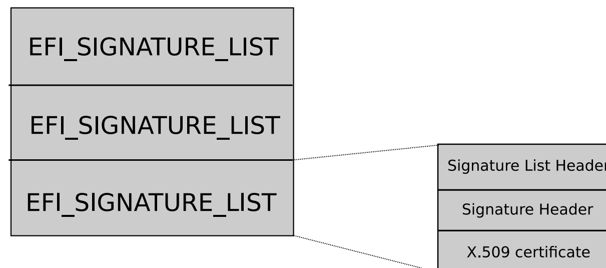


Figure IV.1: Internal structure of a signature database

PKCS#7 updates

`systemd-secure-boot` given the X.509 certificate in Privacy Enhanced Email (PEM) format creates `EFI_SIGNATURE_LIST`. Signature list is then signed using PKCS#7. In order to

[4]Note that every signature list may potentially contain more than one certificate, however OVMF doesn't support this

create PKCS#7 signature it is necessary to provide signing certificate and signing private key. Digital signature of signature list is created by the RSA algorithm. SHA-256 hash function is run over a buffer of data that contains a signature list, a name of the authenticated variable we want to update, an attributes of the variable, a GUID of a signature owner and a time stamp. `systemd-secure-boot` always appends to a signature database, hence it is not necessary to include current time because we never overwrite a content, hence firmware doesn't do a time stamp check. Also, appending to the signature database is advised in cases when reliable time source is not available.

Certificate installation and platform lockdown

`systemd-secure-boot` provides a straight forward way to enroll a certificate. User is not bothered with details of data format and update process. Administrator only provides certificate he would like to install into a signature database, signing key and signing certificate. In order to install a single certificate, user can use following command,

```
# systemd-secure-boot -c PK.crt -k PK.key -g owner-guid -d KEK enroll KEK.crt
```

Running this command would perform an authenticated update of KEK signature database, effectively adding KEK.crt to the signature database. Update is signed with PK.key, which is a private key corresponding to a public key contained within PK.crt. Note that for the enrollment to succeed it is necessary that PK.crt is already installed in the PK signature database.

Proposed tool can also perform not only enrollment of a single certificate but also install multiple certificates at once. `systemd-secure-boot` is usable for doing lockdown of an entire platform. By lockdown we mean transition from Setup Mode to User Mode and installing at least one key into each database. To perform lockdown it is necessary to put a platform into the Setup Mode first. How to do this depends on the specifics of the firmware implementation. Once a platform is in the Setup Mode and previous content of signature databases was cleared then it is possible to perform lockdown using following command,

```
# systemd-secure-boot -s „/CN=Michal Sekletar/" generate
```

```
# systemd-secure-boot -f secure-boot-keys lockdown
```

2 Querying current state of signature databases

Another useful feature implemented by `systemd-secure-boot` is listing current contents of signature databases. To get short overview of current database content, run the following command

```
# systemd-secure-boot list
```

Current content of signature databases is gathered again by reading `efivarfs`. Data is then parsed and presented to the user. See an example output.

```
# systemd-secure-boot list
OWNER                            DB  TYPE
1a80461221e746beae43076c5180b55f KEK X509
```

As we can see in the example, at the time when state of databases was gathered we had only single certificate installed in KEK database. Thus, the platform was in Setup Mode because there was no key enrolled in PK database.

3 Signature database backup

In many cases today, customers are buying consumer grade hardware (e.g. laptops, workstations) and those machines already have some certificates pre-installed and Secure Boot is enabled. However, user may want to delete pre-installed certificates and put his own certificates into the firmware. This of course means once he installed new set of certificates, then machine will refuse to boot operating signed with different key, e.g. Microsoft Windows signed with Microsoft's key. There are cases when user may decide to restore original content of a signature database, e.g. when selling the machine on eBay. In such case it comes in handy to have a backup of the original content. This use case is also supported by `systemd-secure-boot`.

```
# systemd-secure-boot export
```

Previous command will store all certificates currently installed in all signature databases to the current directory. Each certificate is stored in PEM format as well in plain text form. It is also possible to backup only given signature database by passing, `--db <DATABASE>` option to the tool.

4 Testing

During development process we were continuously evaluating and testing an implementation of a tool. Because we wanted to prevent accidental damage of hardware that could happen due to properties of a flash storage or firmware bugs, testing was performed in a virtual machine. Virtual machines (VM) used for testing were qemu VMs with KVM acceleration. Testing was performed on Fedora Linux distribution. By default, qemu doesn't support UEFI. Hence, we needed to use an another firmware implementation that is UEFI compliant. Luckily, Intel open-sourced their reference firmware implementation[1] that implements UEFI specification. qemu does provide an option to choose different firmware when spawning a VM. Thus we were able to spawn normal VM, but with a non-default firmware implementation. Due to the nature of tasks performed by `systemd-secure-boot` testing was not automated and was performed manually. Also, continuous integration framework used by systemd project, Semaphore CI, doesn't allow for running tests in VM with custom firmware. But, non-default firmware implementation is an enabler of doing any testing at all.

Chapter V

Conclusion

Data is becoming very valuable commodity in the world today. It is then to no one surprise that security of information systems is a great concern to their operators. UEFI Secure Boot, main topic of this thesis can help operators to counter pre-OS malware threats and also prevents many attack vectors based on prior deployment of a rootkit type of malware. This is due to an additional validation of a digital signatures of kernel drivers when Secure Boot is enabled. Thus attackers need to exploit some vulnerability in operating system kernel in order to install rootkit. Simply loading kernel module will not work, even when an attacker gains full administrator privileges on a system.

This work provided detailed analysis of UEFI Secure Boot. We think that to have an effective discussion about Secure Boot it is necessary to first cover UEFI in general as well as some aspects of the UEFI specification that are closely related to Secure Boot. UEFI specification and related documents, like UEFI PI specifications are a vast subject. It would not be reasonable to expect that we can cover them all in report fifty or so, pages long. Nevertheless, we tried to shed some light at overall boot up procedure of a platform that implements specifications published by UEFI Forum. In particular, we discussed SEC and PEI phases of platform initialization, then transition to DXE phase where majority of functionality is realized in form of a DXE and UEFI drivers. Next, Boot Device Selection phase brings a system to easily recognizable environment when boot loader is running, possibly displaying familiar boot menu. In next chapter we turned our attention to cryto related subjects. Secure Boot uses digital signatures, RSA algorithm and X.509 in order to deliver on its promise of improved system security. We briefly discussed all those subjects. Following part documented Secure Boot itself.

Main contribution of this thesis is an overview of Secure Boot protocol, delivered in form more digestible and pleasant to read than its formal specification. Secondary, contribution is an implementation and documentation of a simple command line tool mainly focused on management of Secure Boot signature databases. To the best of our knowledge, no such tool is readily available in mainstream Linux distributions. systemd project already provides simple yet very useful UEFI boot manager, systemd-boot. Also, tools provided by systemd project are available to many Linux administrators today. That is why we picked systemd project as a base and implemented tool is delivered as a patch set to systemd rather than standalone product.

Bibliography

[1] EFI Development Kit. [online; cit. 2016-05-20].
URL https://github.com/tianocore/edk2

[2] Let's Encrypt: How It Works? [online; cit. 2016-05-23].
URL https://letsencrypt.org/how-it-works/

[3] Cooper, D.; Santesson, S.; Farrell, S.; aj.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), Květen 2008, [online; cit. 2016-05-23].
URL http://www.ietf.org/rfc/rfc5280.txt

[4] Diffie, W.; Hellman, M.: New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, ročník 22, č. 6, Září 2006: s. 644–654, ISSN 0018-9448, doi:10.1109/TIT.1976.1055638.
URL http://dx.doi.org/10.1109/TIT.1976.1055638

[5] Dijkstra, E. W.: Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60„. Technická Zpráva 35, Mathematisch Centrum, Amsterdam, 1961, [online; cit. 2016-05-10].
URL http://www.cs.utexas.edu/users/EWD/MCReps/MR35.PDF

[6] International Telecommunication Union: ISO/IEC 9594-8:2014 "Information Technology - Open Systems Interconnection - The Directory: Public-Key and Attribute Certificate Frameworks„. Technická zpráva.

[7] Knuth, D. E.; Binstock, A.: Interview with Donald Knuth. *Informit*, 2008, [online; cit. 2016-05-05].
URL http://www.informit.com/articles/article.aspx?p=1193856

[8] Menezes, A. J.: *Handbook of applied cryptography*. Boca Raton: CRC Press, 1997, ISBN 0849385237.

[9] Nechvatal, J.: *Public Key Crypthography*. Gaithersburg: National Institute of Standards and Technology, 1991, [online; cit. 2016-05-18].
URL http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-2.pdf

[10] Nystrom, M.; Kaliski, B.: PKCS #10: Certification Request Syntax Specification Version 1.7. RFC 2986 (Informational), Listopad 2000, [online; cit. 2016-05-11].
URL http://www.ietf.org/rfc/rfc2986.txt

[11] Santesson, S.; Myers, M.; Ankney, R.; aj.: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 6960 (Proposed Standard), Červen 2013, [online; cit. 2016-05-23].
URL http://www.ietf.org/rfc/rfc6960.txt

[12] The Unified EFI Forum: Unified Extensible Firmware Interface Specification. 2016, [online; cit. 2016-05-20].
URL
http://www.uefi.org/sites/default/files/resources/UEFI%20Spec%202_6.pdf

[13] Zimmer, V.; Rothman, M.; Marisetty, S.: *Beyond BIOS*. Intel Press, druhé vydání, 2010, ISBN 13 978-1-934053-29-4.

Appendix A

DEPEX Grammar

```
<depex>     ::= BEFORE <guid>
              | AFTER <guid>
              | SOR <bool>
              | <bool>

<bool>      ::= <term><rightbool>

<rightbool>::= AND <term><rightbool>
              | OR <term><rightbool>
              | ''

<term>      ::= NOT <factor>
              | <factor>

<factor>    ::= '('<bool>')'<rightfactor>
              | NOT <factor> <rightbool> <rightfactor>
              | TRUE <rightfactor>
              | FALSE <rightfactor>
              | END <rightfactor>
              | <guid> <rightfactor>

<rightfactor> ::=AND <term><rightbool> <rightfactor>
              | OR <term><rightbool> <rightfactor>
              | ''

<guid>      ::= '{' <hex32> ',' <hex16> ',' <hex16> ','
                <hex8> ',' <hex8> ',' <hex8> ',' <hex8> ','
                <hex8> ',' <hex8> ',' <hex8> ',' <hex8> '}'

<hex32>     ::= <hexprefix> <hexvalue>
<hex16>     ::= <hexprefix> <hexvalue>
<hex8>      ::= <hexprefix> <hexvalue>

<hexprefix>::= '0' 'x'
              | '0' 'X'

<hexvalue> ::= <hexdigit> <hexvalue>
              | <hexdigit>

<hexdigit> ::= [0-9]
              | [a-f]
              | [A-F]
```

Appendix B

Example of Secure Boot CA certificate

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            61:07:76:56:00:00:00:00:00:08
    Signature Algorithm: sha256WithRSAEncryption
        Issuer: C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Root Certificate Authority 2010
        Validity
            Not Before: Oct 19 18:41:42 2011 GMT
            Not After : Oct 19 18:51:42 2026 GMT
        Subject: C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Windows Production PCA 2011
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:dd:0c:bb:a2:e4:2e:09:e3:e7:c5:f7:96:69:bc:
                    00:21:bd:69:33:33:ef:ad:04:cb:54:80:ee:06:83:
                    bb:c5:20:84:d9:f7:d2:8b:f3:38:b0:ab:a4:ad:2d:
                    7c:62:79:05:ff:e3:4a:3f:04:35:20:70:e3:c4:e7:
                    6b:e0:9c:c0:36:75:e9:8a:31:dd:8d:70:e5:dc:37:
                    b5:74:46:96:28:5b:87:60:23:2c:bf:dc:47:a5:67:
                    f7:51:27:9e:72:eb:07:a6:c9:b9:1e:3b:53:35:7c:
                    e5:d3:ec:27:b9:87:1c:fe:b9:c9:23:09:6f:a8:46:
                    91:c1:6e:96:3c:41:d3:cb:a3:3f:5d:02:6a:4d:ec:
                    69:1f:25:28:5c:36:ff:fd:43:15:0a:94:e0:19:b4:
                    cf:df:c2:12:e2:c2:5b:27:ee:27:78:30:8b:5b:2a:
                    09:6b:22:89:53:60:16:2c:c0:68:1d:53:ba:ec:49:
                    f3:9d:61:8c:85:68:09:73:44:5d:7d:a2:54:2b:dd:
                    79:f7:15:cf:35:5d:6c:1c:2b:5c:ce:bc:9c:23:8b:
                    6f:6e:b5:26:d9:36:13:c3:4f:d6:27:ae:b9:32:3b:
                    41:92:2c:e1:c7:cd:77:e8:aa:54:4e:f7:5c:0b:04:
                    87:65:b4:43:18:a8:b2:e0:6d:19:77:ec:5a:24:fa:
                    48:03
                Exponent: 65537 (0x10001)
        X509v3 extensions:
            1.3.6.1.4.1.311.21.1:
                ...
            X509v3 Subject Key Identifier:
                A9:29:02:39:8E:16:C4:97:78:CD:90:F9:9E:4F:9A:E1:7C:55:AF:53
            1.3.6.1.4.1.311.20.2:
                .
.S.u.b.C.A
            X509v3 Key Usage:
                Digital Signature, Certificate Sign, CRL Sign
            X509v3 Basic Constraints: critical
                CA:TRUE
            X509v3 Authority Key Identifier:
                keyid:D5:F6:56:CB:8F:E8:A2:5C:62:68:D1:3D:94:90:5B:D7:CE:9A:18:C4

            X509v3 CRL Distribution Points:

                Full Name:
                  URI:http://crl.microsoft.com/pki/crl/products/MicRooCerAut_2010-06-23.crl

            Authority Information Access:
                CA Issuers - URI:http://www.microsoft.com/pki/certs/MicRooCerAut_2010-06-23.crt

    Signature Algorithm: sha256WithRSAEncryption
         14:fc:7c:71:51:a5:79:c2:6e:b2:ef:39:3e:bc:3c:52:0f:6e:
         2b:3f:10:13:73:fe:a8:68:d0:48:a6:34:4d:8a:96:05:26:ee:
         31:46:90:61:79:d6:ff:38:2e:45:6b:f4:c0:e5:28:b8:da:1d:
```

8f:8a:db:09:d7:1a:c7:4c:0a:36:66:6a:8c:ec:1b:d7:04:90:
a8:18:17:a4:9b:b9:e2:40:32:36:76:c4:c1:5a:c6:bf:e4:04:
c0:ea:16:d3:ac:c3:68:ef:62:ac:dd:54:6c:50:30:58:a6:eb:
7c:fe:94:a7:4e:8e:f4:ec:7c:86:73:57:c2:52:21:73:34:5a:
f3:a3:8a:56:c8:04:da:07:09:ed:f8:8b:e3:ce:f4:7e:8e:ae:
f0:f6:0b:8a:08:fb:3f:c9:1d:72:7f:53:b8:eb:be:63:e0:e3:
3d:31:65:b0:81:e5:f2:ac:cd:16:a4:9f:3d:a8:b1:9b:c2:42:
d0:90:84:5f:54:1d:ff:89:ea:ba:1d:47:90:6f:b0:73:4e:41:
9f:40:9f:5f:e5:a1:2a:b2:11:91:73:8a:21:28:f0:ce:de:73:
39:5f:3e:ab:5c:60:ec:df:03:10:a8:d3:09:e9:f4:f6:96:85:
b6:7f:51:88:66:47:19:8d:a2:b0:12:3d:81:2a:68:05:77:bb:
91:4c:62:7b:b6:c1:07:c7:ba:7a:87:34:03:0e:4b:62:7a:99:
e9:ca:fc:ce:4a:37:c9:2d:a4:57:7c:1c:fe:3d:dc:b8:0f:5a:
fa:d6:c4:b3:02:85:02:3a:ea:b3:d9:6e:e4:69:21:37:de:81:
d1:f6:75:19:05:67:d3:93:57:5e:29:1b:39:c8:ee:2d:e1:cd:
e4:45:73:5b:d0:d2:ce:7a:ab:16:19:82:46:58:d0:5e:9d:81:
b3:67:af:6c:35:f2:bc:e5:3f:24:e2:35:a2:0a:75:06:f6:18:
56:99:d4:78:2c:d1:05:1b:eb:d0:88:01:9d:aa:10:f1:05:df:
ba:7e:2c:63:b7:06:9b:23:21:c4:f9:78:6c:e2:58:17:06:36:
2b:91:12:03:cc:a4:d9:f2:2d:ba:f9:94:9d:40:ed:18:45:f1:
ce:8a:5c:6b:3e:ab:03:d3:70:18:2a:0a:6a:e0:5f:47:d1:d5:
63:0a:32:f2:af:d7:36:1f:2a:70:5a:e5:42:59:08:71:4b:57:
ba:7e:83:81:f0:21:3c:f4:1c:c1:c5:b9:90:93:0e:88:45:93:
86:e9:b1:20:99:be:98:cb:c5:95:a4:5d:62:d6:a0:63:08:20:
bd:75:10:77:7d:3d:f3:45:b9:9f:97:9f:cb:57:80:6f:33:a9:
04:cf:77:a4:62:1c:59:7e

Appendix C

Content of CD

The attached CD includes following files,

- `0001-systemd-secure-boot.patch` - Patch generated against systemd source tree introducing Secure Boot support.
- `tex/` - LaTeX source code of this thesis