Secure Boot

Analysis of Secure Boot and the Trusted Boot Chain

>_ DEV v1.3-RC1

Ernesto Martínez García[1]
me@ecomaikgolf.com
ecomaikgolf#3519

Graz University of Technology

Secure Application Design VO SS23

23rd of June 2023

SLIDES & REPORT

ls.ecomaikgolf.com/slides/secureboot/

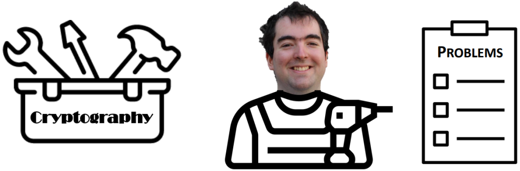
Motivation

Why Secure Boot?

We've been studying how cryptography can be applied to solve real world problems:

RKSV Green Pass eIDAS ID Austria ...

Secure Boot also uses a "cryptographic toolbox" to solve a real world problem



This was the goal of SEAD, learning how "toolboxes" are used to solve problems.

We'll see how cryptography is being used in Secure Boot and which problem solves

Table of Contents

- Introduction
  - History of Booting Mechanisms
  - What is Secure Boot?
  - Unpopularity of Secure Boot
- Threat Model
  - What Secure Boot Defends Against?
- Signature Verification Chain
  - EFI Firmware
  - Shim
  - Bootloader
  - Kernel
- Past Vulnerabilities
  - Defended By Secure Boot
  - Secure Boot Open Problems
  - Secure Boot Advanced Targeting
- Experiment
  - Attacking non Secure Boot systems
- Conclusion
  - Current State
  - Future Personal Opinion
  - Takeaways

Introduction

Classical Booting Mechanisms

¶ Basic Input Output System (BIOS) has been the main booting mechanism for years

- x86 processors jump to BIOS code in ROM (originally) to execute it
- BIOS will perform POST ✔, device initialization and will jump to our bootloader
- Acts as a layer between firmware and the rest ⇄

📅 Years passed and BIOS accumulated many limitations ☹

- 512 byte boot sector
- 16 bit real mode calls to BIOS
- ....

⚗ Intel, with its Itanium architecture, started looking for alternatives

Modern Booting Mechanisms

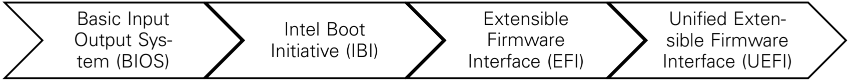
In 1998 Intel started the development of the Intel Boot Initiative (IBI)

Intel Boot Initiative (IBI) got renamed to Extensible Firmware Interface (EFI)

EFI aimed to be a standarized, easier and better BIOS replacement

Intel moved EFI `v1.10` to Unified Extensible Firmware Interface (UEFI)

UEFI was managed by a consortium of companies (AMD, Apple, Microsoft, Intel, etc.).



Nowadays we use UEFI with an extension that support BIOS booting: CSM

Modern OS support UEFI, CSM should be disabled in UEFI configuration

Secure Boot History

Origin

In November 2010, Secure Boot got introduced in the UEFI `v2.2` standard

Deployment

Consumer deployment was a disaster, by default could only be used by Microsoft

This was seen as a movement to destroy Linux . Had to be disabled ☹

Expansion

A solution was found by the Linux community, but it was still unpopular

Secure Boot support got introduced in Fedora 18, RHEL 7, Debian 10, Ubuntu 12.04 ...

Secure Boot Goal I

- UEFI `v2.2` provided a protocol to verify a image signature upon execution

Microsoft could sign the bootloader and UEFI would only boot after checking the signature

This has more implications that “the bootloader is originally from Microsoft”.

Implications

An authentic bootloader lets us execute authentic non-tampered code

- We could implement trusted checks against the Operating System
  - Filesystem contain signs of malware?
  - Are Merkle Tree hashes correct?
- We could extend the trusted chain to verify more code
  - Is Windows signature authentic too?

Secure Boot Goal II

- Achieving this goals by using cryptography introduces a new set of challenges.
  - Key Management
    - Key Storage
    - Key Decommission
    - ...
  - Trust Management
    - Is Windows Trusted?
    - Who Decides Trust?
    - ...
  - Root of Trust
    - Is UEFI Firmware Trusted?
    - ...
- .... basically the common suspects in cryptography

Threat Model

Defend Against What?

What Secure Boot would defend against? What would not defend against?

Yes	No
Malicious software persistence	Malicious UEFI Firmware
Malware expansion (Windows → Linux)	Physical Keyloggers
Kernel Malicious Modifications	Physical Exploits (RAM access, etc.)
Naive disk physical access	Software Exploits (Signature Bypass...)
...	...

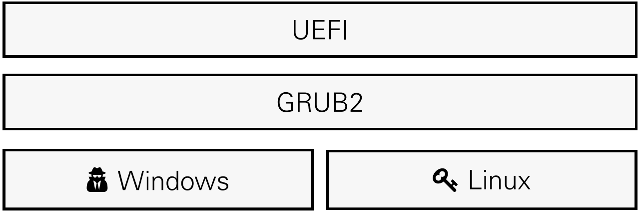
Secure Boot is far from being a perfect security measure

Vendor's UEFI has a exploit/backdoor? Secure Boot is defeated. You need to trust in it.

A regular user may not have this in its threat model

Example Case (No Secure Boot)

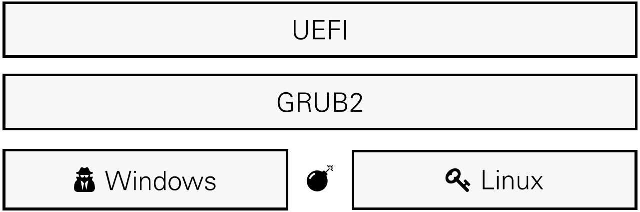
Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



Eve has breached our Windows system and wants to expand to our encrypted Linux

Example Case (No Secure Boot)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.

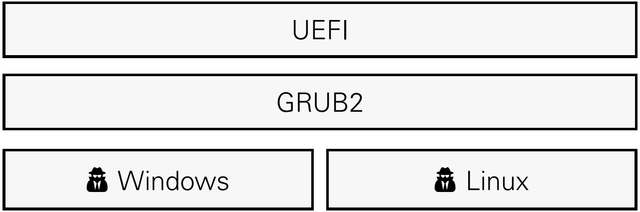


Eve has breached our Windows system and wants to expand to our encrypted Linux

Eve modifies the Linux kernel image to run malicious code (at kernel privilege level).

Example Case (No Secure Boot)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



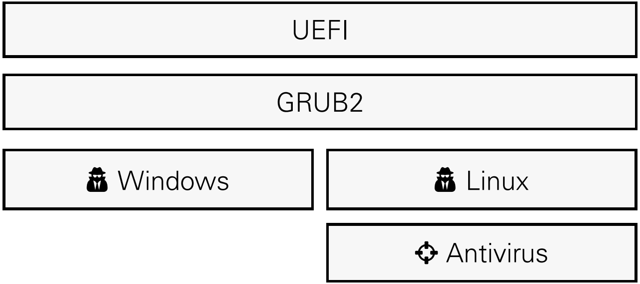
Eve has breached our Windows system and wants to expand to our encrypted Linux

Eve modifies the Linux kernel image to run malicious code (at kernel privilege level).

GRUB2 won’t detect the attack and will execute the malicious Linux kernel image

Example Case (No Secure Boot)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.

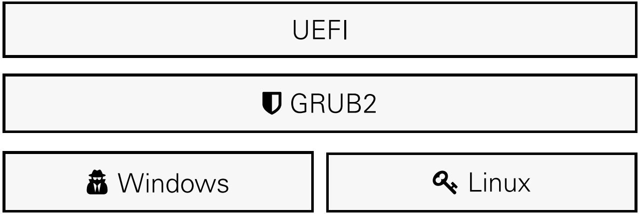


But, but I have an antivirus (or any other fancy system).

Yes, but Eve has the Kernel.

Example Case (Naive “Secure Boot”)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



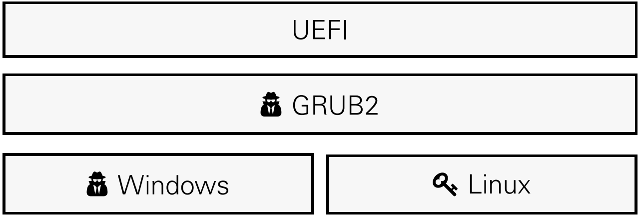
GRUB2 verifies Linux Kernel signature

...so Eve cannot modify without it getting disallowed.

Eve has breached our Windows system and wants to expand to our encrypted Linux.

Example Case (Naive "Secure Boot")

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



GRUB2 verifies Linux Kernel signature

...so Eve cannot modify without it getting disallowed.

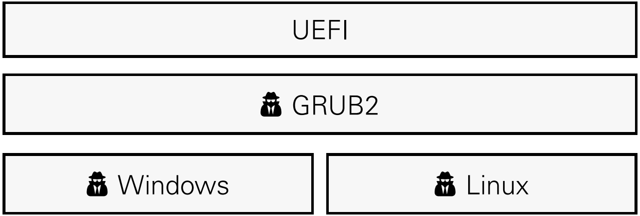
Eve has breached our Windows system and wants to expand to our encrypted Linux.

Eve can modify GRUB2 with a version that doesn't do checks

...UEFI won't complain.

Example Case (Naive “Secure Boot”)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



GRUB2 verifies Linux Kernel signature

...so Eve cannot modify without it getting disallowed.

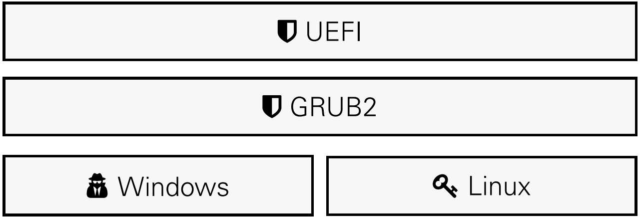
Eve has breached our Windows system and wants to expand to our encrypted Linux.

Eve can modify GRUB2 with a version that doesn’t do checks

...UEFI won’t complain. Then modify the Linux kernel image

Example Case (Secure Boot)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



UEFI verifies GRUB2. GRUB2 verifies Linux.

Eve has breached our Windows system and wants to expand to our encrypted Linux.

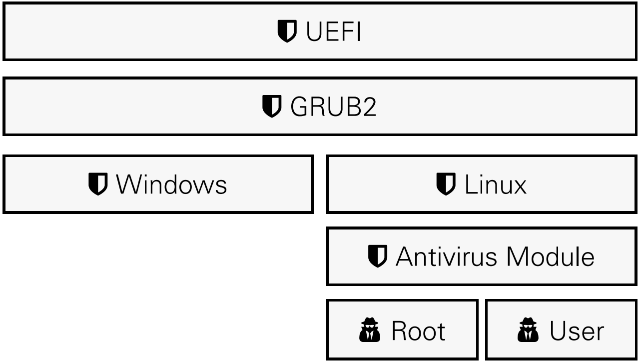
Eve cannot modify Linux kernel image because GRUB2 would notice.

cannot modify GRUB2 image because UEFI would notice.

cannot modify UEFI firmware code without a major attack.

Example Case (Secure Boot)

Dual Boot Windows/Linux Setup. Linux is Full Disk Encrypted with LUKS.



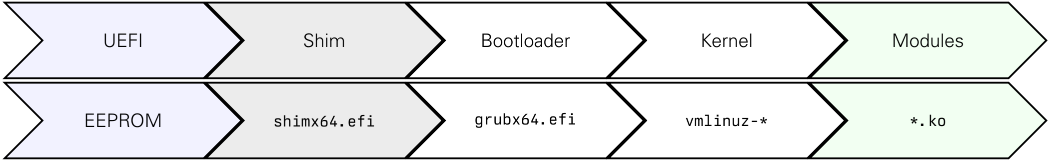
See how it can be extended to other threat models and usecases.

Signature Verification Chain

🔗

Overview

What does have to verify what, and where?

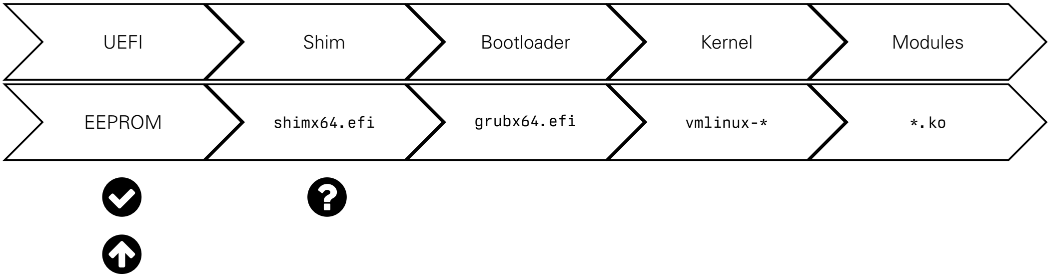


Starting in UEFI, each step in the chain verifies the next one.

We will
- [x] assume that UEFI is authentic.
- [x] verify until kernel code.

> **Warning:** Everything is based in a <u>default</u> Fedora installation (w/ Secure Boot).

Stage Roadmap



UEFI Stage: Key Management

UEFI has 4 important Secure Boot NVRAM (Non Volatile) variables:

PK	KEK	DB	DBX

PK: Platform Key

Contains one RSA 2048 public key certificate (X509)

Usually provided by the motherboard vendor

Root of Trust

>_ efi-readvar -v PK

```
PK: List 0, type X509
    Signature 0, size 858, owner 3b053091-6c9f-04cc-b1ac-e2a51e3be5f5
        CN=ASUSTeK MotherBoard PK Certificate
```

UEFI Stage: Key Management

UEFI has 4 important Secure Boot NVRAM (Non Volatile) variables:

PK	KEK	DB	DBX

- KEK: Key Exchange Key
- Contains multiple RSA 2048 public key certificates (X509)
- Adding keys require PK signature
- Usually from Operating System vendors

>_ efi-readvar -v KEK

```
KEK: List 1, type X509
    Signature 0, size 1532, owner 77fa9abd-0359-4d32-bd60-28f4e78f784b
        C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Corporation KEK CA...
```

UEFI Stage: Key Management

UEFI has 4 important Secure Boot NVRAM (Non Volatile) variables:

PK	KEK	DB	DBX

DB: Allow List

- Contains multiple RSA 2048 public key certificates (X509) or hashes
- Adding entries require KEK signature
- If signature or hash of a binary matches, allows execution

>_ efi-readvar -v db

```
db: List 5, type SHA256
    Signature 0, size 48, owner 00000000-0000-0000-0000-000000000000
        Hash:f58fbdf71be8c37cbbd6944e472c450b1043817b972914487c221033f3079e43
```

UEFI Stage: Key Management

UEFI has 4 important Secure Boot NVRAM (Non Volatile) variables:

PK	KEK	DB	DBX

DBX: Deny List

- Contains multiple RSA 2048 public key certificates (X509) or hashes
- Adding entries require KEK signature
- If signature or hash of a binary matches, disallows execution

```
>_ efi-readvar -v dbx

dbx: List 3, type SHA256
    Signature 0, size 48, owner 77fa9abd-0359-4d32-bd60-28f4e78f784b
        Hash:c55be4a2a6ac574a9d46f1e1c54cac29d29dcd7b9040389e7157bb32c4591c4c
```

UEFI Stage: Signature Verification

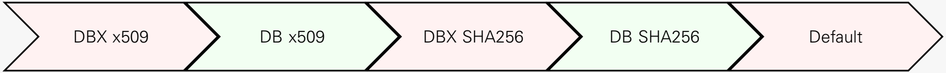
A Program Executable (PE) file is fed into UEFI with Secure Boot Enabled:

1. Check Policy

Image comes from Firmware Volume? Allow ✔

Else, consult UEFI Policy and decide ?

2. Check Variables



If image has no signatures, X509 checks skipped.

If DB X509 is ✔ but SHA256 in DBX ✖, ends up denied ✖

UEFI Stage: Variable Updating

Microsoft has released a new bootloader. Wants to include its hash into DB variable

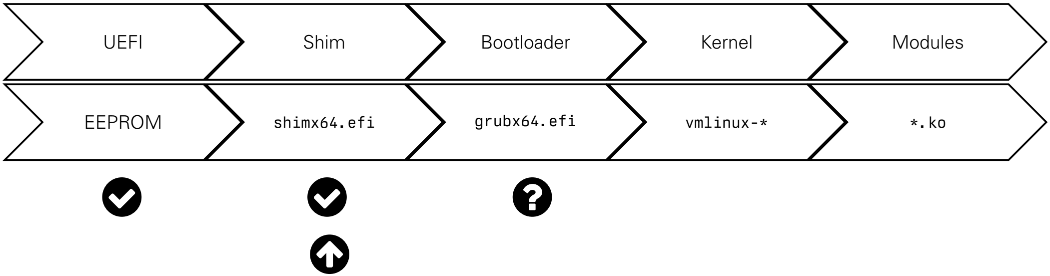
(Time Based) Write Authenticated Variables

UEFI Secure Boot DB, DBX and KEK variables are "Time Based Write Authenticated".

- Can be updated from the Operating System (others not!).
- DB/DBX updates require a signed "transaction" with a KEK
- KEK updates require a signed "transaction" with the PK

0. We have Microsoft's public key certificate in KEK.
1. Microsoft signs the new bootloader with their own private key.
2. Then rolls a new update that installs the bootloader
3. The update also makes a change to DB, signed by Microsoft
4. Our system now accepts the new bootloader ✔

Stage Roadmap



Shim Stage: Introduction

- If device vendors sell devices with Microsoft's keys only, Linux won't work by default
- We could force vendors to ship keys from Canonical, Red Hat and other big players.
- Small Distributions wouldn't work with Secure Boot.

> **Shim**
>
> EFI program that will manage verification with additional keys.
>
> - Developed by the Red Hat Bootloader Team, used everywhere
> - Small, Simple and Robust Code. Open Source.
> - Reproducible Builds

- Ask Microsoft to sign the Shim, and then we could manage the keys as we want.

Fedora's shim is signed by Microsoft ☺

Shim Stage: Introduction

```
>_ sudo osslsigncode verify /boot/efi/EFI/fedora/shimx64.efi
```

```
Signer #0:
    Subject: /C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Windows UEFI Driver
          Publisher
    Issuer : /C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Corporation UEFI CA 2011
    Serial : 3300000048C9DA2834CCE76565000100000048
    Certificate expiration date:
        notBefore : Sep  9 19:40:20 2021 GMT
        notAfter : Sep  1 19:40:20 2022 GMT
Signer #1:
    Subject: /C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Corporation UEFI CA 2011
    Issuer : /C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Corporation Third Party
          Marketplace Root
    Serial : 6108D3C4000000000004
    Certificate expiration date:
        notBefore : Jun 27 21:22:45 2011 GMT
        notAfter : Jun 27 21:32:45 2026 GMT

Authenticated attributes:
    Microsoft Individual Code Signing purpose
    Message digest: 6C96095DF9D18B0F19E694091BC43BA08FC73E802BC3B279D4E5FE777542FBD7
    URL description: https://www.microsoft.com/en-us/windows
    Text description: Red Hat, Inc.
```

Shim Stage: Key Management

Shim introduces 2 new variables:

MOK	MOKX

- MOK: Machine Owner Key
- Allowlist
- Inserted from the Operating System with "MokManager"
- Also named `MokList` or `MokListRT`

```
>_ mokutil -list-enrolled | grep "Issuer:"

Issuer: CN=Fedora Secure Boot CA
Issuer: O=akmods local, OU=akmods/emailAddress=akmods@localhost.localdomain, L=None, ST=None, C=XX, ...
```

Shim Stage: Key Management

Shim introduces 2 new variables:

MOK	MOKX

- MOK: Machine Owner (Deny) Key
- Denylist
- Inserted from the Operating System with “MokManager”
- Also named `MokListX` or `MokListXRT`

```
>_ mokutil -X -list-enrolled | grep ”Issuer:”
```

```
[empty]
[empty]
```

Shim Stage: Signature Verification

A Program Executable (PE) file is fed into the Shim:

0. Install Verification Protocol

Register a UEFI protocol for Secure Boot verification.

1. Check Secure Boot

No Secure Boot? Allow ✔

2. Check Variables

Note: Merged hash and signature checks in the diagram for brevity.

Shim Stage: Key Enrollment

A developer releases custom signed bootloaders. We want to use them with SB.

MokManager

Shim variables can only be update through the “MokManager”

- We request an addition from the operating system (`mokutil`)
- During the next boot, shim will ask us if we want to enroll it
- Shim code is authentic, attackers cannot do their tricks there

0. Developer gives us its public key certificate
1. `mokutil -import devkey.cer`
2. Asks us for a random password
3. While booting, shim will open its MokManager and will ask to enroll the key
4. We would have to enter the previosly introduced random password

Shim Stage: Smart UX Security I

Bob has Linux with Secure Boot Enabled. Eve wants to modify GRUB2

UEFI

Shim

GRUB2

Linux

`sudo mokutil –import eve.cer`

Eve achieved root, uses `mokutil` to insert her key. She enters a password.

Shim Stage: Smart UX Security I

Bob has Linux with Secure Boot Enabled. Eve wants to modify GRUB2

UEFI

Shim (MokManager)

Bob, a non power user, reboots the computer. After the reboot, a big blue screen pops.

Shim Stage: Smart UX Security I

Bob has Linux with Secure Boot Enabled. Eve wants to modify GRUB2

UEFI

Shim (MokManager)

Enrolling is not trivial. Bob doesn't even know what to do. And that's nice.

Shim Stage: Smart UX Security I

Bob has Linux with Secure Boot Enabled. Eve wants to modify GRUB2

UEFI
Shim (MokManager)

Magically Bob ends up enrolling that specific key, but gets asked for a password.

The password is Eve's password

Bob doesn't know what password it's talking about

Eve can't trick Bob in any way, she cannot run code here

Bob is getting worried and will force a reboot. Eve plan fails

Shim Stage: Smart UX Security II

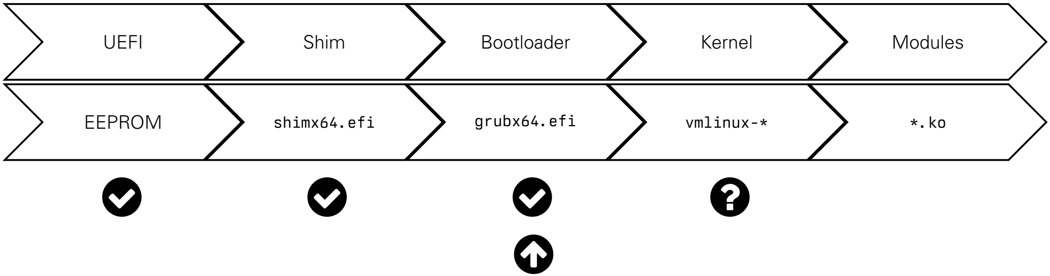
Tom has Linux with Secure Boot Enabled. Eve wants to modify GRUB2.

Eve inserts a new key with `mokutil`.

Tom is a power user, he sees the following after doing a regular reboot:

Tom puts the computer in flames (or wouldn't install the key). Eve plain fails ✖

Stage Roadmap



Bootloader Stage

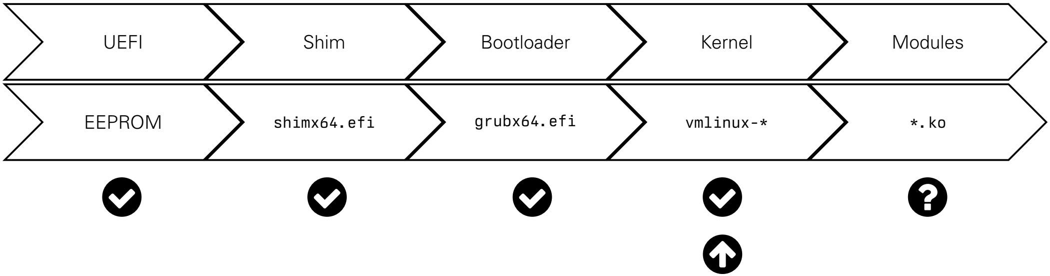
- Bootloader stage is going to be the easiest one to learn, promise
- Remember how we said that Shim installed a “UEFI protocol”?
- GRUB will look for that protocol, and use it for verification. Shim stage applies here
- You can see it as passing the function pointer and reusing code
- So GRUB does two extra things if secure boot is enabled:

1. Enables GRUB2 Lockdown Mode

> **Lockdown Mode**
>
> Disables GRUB2 “dangerous” functionalities: `outb`, `outw`, `outl`, `write_byte`....

2. Look for the Shim protocol and “registers” it for its usage
   Shim function will be called when loading the kernel

Stage Roadmap



Kernel Stage: Introduction

- We verified and loaded the kernel? Did we finish the chain?
- Well, depends...
- Linux Kernel is modular, we can load kernel code at runtime. It has to be verified too
- We have to verify kernel objects `.ko` files
- Verification process & keys used is a bit different

Kernel Stage: Key Management

Kernel has a complex system for key management: keyring

SECONDARY_KEYRING

PRIMARY_KEYRING

- Uses `.secondary_trusted_keys` or `.builtin_trusted_keys`
  - `.secondary_trusted_keys` contains MOK added keys
  - `.builting_trusted_keys` are bundled in the kernel at compile time
- Depending on compile time kernel options

```
>_ sudo keyctl show %:.secondary_trusted_keys
```

```
959992111  ---lswrv keyring: .machine
1053431220 ---lswrv asymmetric: Fedora Secure Boot CA: fde32599c2d61db1bf5807335d7b20e4cd963b42
463485773  ---lswrv asymmetric: akmods local signing CA: 85f92b454b0b00bcccc2dd0a76640023153bed01
```

Kernel Stage: Key Management

Kernel has a complex system for key management: keyring

SECONDARY_KEYRING

PRIMARY_KEYRING

Uses `.platform_trusted_keys` or `null`

- `.platform_trusted_keys` contains UEFI DB keys
- `null` is nothing

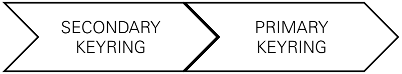
Depending on compile time kernel options

```
>_ sudo keyctl show %:.platform
```

```
496716708 ---lswrv keyring: .platform
788952781 ---lswrv asymmetric: ASUSTeK MotherBoard SW Key Certificate: da83b990422ebc8c441f8d8b039a65a2
944196719 ---lswrv asymmetric: Canonical Ltd. Master Certificate Authority: ad91990bc22ab1f517048c23...
```

Kernel Stage: Signature Verification

- Kernel has a different signature verification system
- For each keyring, checked in this order:



1. Checks if signature key is in `.blacklist_keyring`. If it’s the case, denies

   Yes, the denylist is a custom one

2. Verifies if signature key is in the checked keyring. If its the case, allows

Kernel Stage: Lockdown Mode

Kernel enables a lockdown mode when secure boot is enabled

```
>_ dmesg | grep kernel_lockdown
```

```
kern  :notice: [...] Kernel is locked down from EFI Secure Boot mode; see man kernel_lockdown.7
kern  :notice: [...] Lockdown: swapper/0: hibernation is restricted; see man kernel_lockdown.7
kern  :notice: [...] Lockdown: systemd-logind: hibernation is restricted; see man kernel_lockdown.7
```

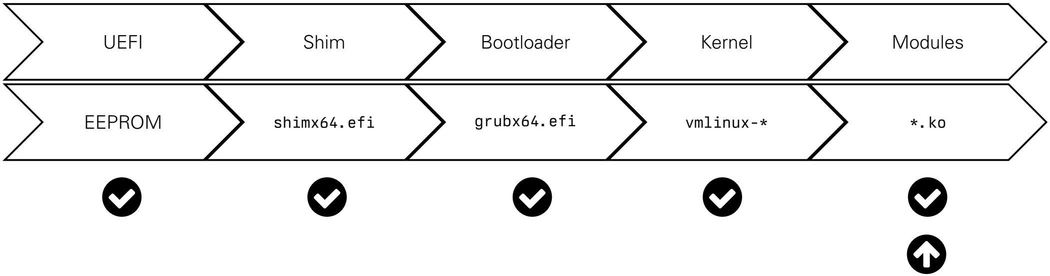
So... let's read the manual

"The Kernel Lockdown feature is designed to prevent both direct and indirect access to a running kernel image, attempting to protect against unauthorized modification of the kernel image..."

Disabled Kernel functionalities:

- Unencrypted hibernation/suspend to swap
- Only validated signed binaries can be kexec'd
- ....

Stage Roadmap



Past Vulnerabilities 💣

Case I: Debug Windows Bootloader

- ☺ In 2016 Microsoft released a signed bootloader in a Windows Update
- 😐 It was a Debug Build
- ☹ Debug builds had signature checks disabled
- ❓ What we do now? How do we fix it?
- Easy! Another Windows Update rollbacking the bootloader
  - 🕵 Can Eve use it to break Secure Boot Chains?
  1. Swap the original bootloader with the debug one
  2. UEFI won't care, it's signed
  3. Change the kernel, there are no signature checks

Put the hash into DBX! That's why it exists!

Case II: BootHole

- ☺ Security researchers reviewed GRUB2 code
- 😐 They found a buffer overflow in `grub.cfg` that skipped signature checks
- ☹ More than 150 GRUB2 different signed builds
- ❓ What's the big deal? Why should we care? We already have a DBX, right?
- 🖴 Yes, but it's an NVRAM variable, it has a size limit

Welcome to the Secure Boot open problem ☺

Secure Boot Open Problem

- Remember what was the problem with JWT tokens?
- We had to mantain a denylist in order to remove access from certain tokens
- But JWT usually have a field that makes this not a problem...
- Expiration times! We don't have a infinitely growing denylist
- Could we have expiration times in Secure Boot?
  - Is the time trusted?
  - What if I don't turn on the computer for a long time?
- Could we remove entries from DBX to save space in a future?
  - No! What if Eve reverts to that version to break secure boot?

Secure Boot Advanced Targeting

Secure Boot Advanced Targeting (SBAT) is the proposed solution

SBAT Example

```
sbat,1,SBAT Version,sbat,1,https://github.com/rhboot/shim/blob/main/SBAT.md
grub,1,Free Software Foundation,grub,2.02,https://www.gnu.org/software/grub/
grub.fedora,1,Red Hat Enterprise Linux,grub2,2.02-0.34.fc24,mail:secalert@redhat.com
grub.rhel,1,Red Hat Enterprise Linux,grub2,2.02-0.34.el7_2,mail:secalert@redhat.com
```

1. Signed metadata `.sbat` section on each executable
2. Use that metadata to revoke a whole range of versions at once
3. “Grub 2.x had a vulnerability, deny all of them, no matter which vendor comes from”

Technique used in the Shim but not standarized in UEFI

Experiment

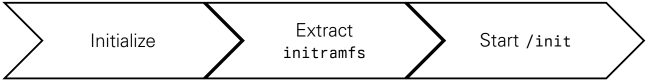
✋ Who has LUKS Encryption?

✋ Who has Secure Boot?

Experiment

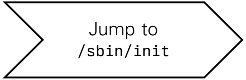
- 💣 We are going to steal LUKS keys from systems without secure boot (academically!)
- 🐧 Before, let's recap how Linux boots

0. GRUB2 loads Linux image and a `initramfs` file
1. Linux Kernel gets the control (without a filesystem):



2. Linux has a (fake) `/` filesystem from initramfs

3. Linux has the real `/` filesystem



>_ Let’s move to the terminal

Conclusion

Current State & Future of Secure Boot

- What's the current state of Secure Boot?
- People don't usually know about this mechanism
- Not widely used in Linux
  - Early days of Secure Boot, Linux wikis adviced to disable it
  - Now maybe used if the distro installs it automatically
- Similar mechanisms widely used in Android and iOS
- In the future, we might see greater default desktop support for it

Takeaways

- [x] //TODO
  - [ ] Disable CSM (Compatibility Support Module) in UEFI
  - [ ] Enable Secure Boot
  - [ ] Configure Secure Boot in Linux (or just reinstall a major distro that supports it)
  - [ ] Preferably encrypt with LUKS

Questions?

☺

Secure Boot

Analysis of Secure Boot and the Trusted Boot Chain

>_ PROD v1.3 ✔

Ernesto Martínez García [1]
me@ecomaikgolf.com
ecomaikgolf#3519

Graz University of Technology

Secure Application Design VO SS23

23rd of June 2023

SLIDES & REPORT

ls.ecomaikgolf.com/slides/secureboot/