SecureBoot

Translation(s): English - Français

Contents

1. What is UEFI?
2. What is UEFI Secure Boot?
3. What is UEFI Secure Boot NOT?
4. Shim
5. Finding out information on your system
   1. Has the system booted via Secure Boot?
   2. What keys are on my system?
   3. I have shim and the Debian bootloader installed, and with secure boot enabled the machine fails to boot!
6. Finding Debian's Secure Boot keys
7. MOK - Machine Owner Key
   1. DKMS and Secure Boot
      1. Virtualbox Packages
      2. Enrolling your key
      3. Adding your key to DKMS
      4. DKMS Sign Helper
   2. Manually signing kernels and kernel modules
      1. Using your key to sign your kernel
      2. Using your key to sign modules
   3. Verifying if a module is signed
   4. Disabling/re-enabling Secure Boot
8. Supported architectures and packages
9. Secure Boot setup with systemd-boot
10. Testing UEFI Secure Boot
11. Secure Boot limitations
12. Infrastructure - how signing works in Debian
13. Testing Secure Boot in a virtual machine
14. arm64 problems
15. Alternatives to Secure Boot

What is UEFI?

UEFI is the Unified Extensible Firmware Interface. See the main UEFI page for more details about it.

What is UEFI Secure Boot?

UEFI Secure Boot (SB) is a verification mechanism for ensuring that code launched by a computer's UEFI firmware is trusted. It is designed to protect a system against malicious code being loaded and executed early in the boot process, before the operating system has been loaded.

Secure Boot works using cryptographic checksums and signatures. Each program that is loaded by the firmware includes a signature and a checksum, and before allowing execution the firmware will verify that the program is trusted by validating the checksum and the signature. When Secure Boot is enabled on a system, any attempt to execute an untrusted program will not be allowed. This stops unexpected / unauthorised code from running in the UEFI environment.

Most x86 hardware comes from the factory pre-loaded with Microsoft keys. This means the firmware on these systems will trust binaries that are signed by Microsoft. Most modern systems will ship with Secure Boot enabled - they will *not* run any unsigned code by default. Starting with Debian version 10 ("Buster"), Debian supports UEFI Secure Boot by employing a small UEFI loader called shim which is signed by Microsoft and embeds Debian's signing keys. This allows Debian to sign its own binaries without requiring further signatures from Microsoft. Older Debian versions did not support Secure Boot, so users had to disable Secure Boot in their machine's firmware configuration prior to installing those versions.

It is possible to change the firmware configuration to either disable Secure Boot or to enroll extra signing keys. This is not required when running standard kernels provided by Debian but may be useful to some users, for example when using custom kernel builds.

Most of the programs that are expected to run in the UEFI environment are boot loaders, but others exist too. There are also programs to deal with firmware updates before operating system startup (like fwupdate and fwupd), and other utilities may live here too.

What is UEFI Secure Boot NOT?

UEFI Secure Boot is **not** an attempt by Microsoft to lock Linux out of the PC market here; Secure Boot is a security measure to protect against malware during early system boot. Microsoft act as a Certification Authority (CA) for Secure Boot, and they will sign programs on behalf of other trusted organisations so that their programs will also run. There are certain identification requirements that organisations have to meet here, and code has to be audited for safety. But these are not too difficult to achieve.

Secure Boot is also **not** meant to lock users out of controlling their own systems. Users can enroll extra keys into the system, allowing them to sign programs for their own systems. Many Secure Boot-enabled systems also allow users to remove the platform-provided keys altogether, forcing the firmware to only trust user-signed binaries.

Shim

shim is a simple software package that is designed to work as a first-stage bootloader on UEFI systems.

It was developed by a group of Linux developers from various distros, working together to make Secure Boot work using Free Software. It is a common piece of code that is safe, well-understood and audited so that it can be trusted and signed using platform keys. This means that Microsoft (or other potential firmware CA providers) only have to worry about signing shim, and not all of the other programs that distro vendors might want to support.

Shim then becomes the root of trust for all the other distro-provided UEFI programs. It embeds a further distro-specific CA key that is itself used for as a trust root for signing further programs (e.g. Linux, GRUB, fwupdate). This allows for a clean delegation of trust - the distros are then responsible for signing the rest of their packages. Shim itself should ideally not need to be updated very often, reducing the workload on the central auditing and CA teams.

For extra trust and safety, from version 15 onwards the shim binary build is 100% reproducible - you can rebuild the Debian shim binary yourself to verify that no unexpected changes have been embedded in this key piece of security software.

Finding out information on your system

Has the system booted via Secure Boot?

```
$ sudo mokutil --sb-state
SecureBoot enabled
```

Note that you can be only sure that the above answer is correct if your system has not been tampered with in the first place.

What keys are on my system?

If you want to know what keys are in use on your system, various other mokutil calls will help, e.g. `sudo mokutil --list-enrolled` to show the current MOK key list. See the man page for more.

You can also see which keys your kernel knows about by looking in kernel boot messages.

First, the kernel will list keys included at build time, e.g. on current Debian kernels:

```
[    1.378223] Loading compiled-in X.509 certificates
[    1.395916] Loaded X.509 cert 'Debian Secure Boot CA: 6ccece7e4c6c0d1f6149f3dd27dfcc5cb
[    1.395954] Loaded X.509 cert 'Debian Secure Boot Signer 2022 - linux: 14011249c2675ea8
```

Second, the kernel will list the keys it has found in the UEFI key variables (DB, and MokListRT if Secure Boot is enabled). The following is example output from a Lenovo Thinkpad with Secure Boot enabled:

```
[    1.378223] Loading compiled-in X.509 certificates
[    1.398449] integrity: Loading X.509 certificate: UEFI:db
[    1.399705] integrity: Loaded X.509 cert 'Lenovo Ltd.: ThinkPad Product CA 2012: 838b1f
[    1.400994] integrity: Loading X.509 certificate: UEFI:db
[    1.402315] integrity: Loaded X.509 cert 'Lenovo UEFI CA 2014: 4b91a68732eaefdd2c8ffffc
[    1.403607] integrity: Loading X.509 certificate: UEFI:db
[    1.404966] integrity: Loaded X.509 cert 'Microsoft Corporation UEFI CA 2011: 13adbf430
[    1.406361] integrity: Loading X.509 certificate: UEFI:db
[    1.407692] integrity: Loaded X.509 cert 'Microsoft Windows Production PCA 2011: a92902
[    1.410410] integrity: Loading X.509 certificate: UEFI:MokListRT (MOKvar table)
[    1.411856] integrity: Loaded X.509 cert 'Debian Secure Boot CA: 6ccece7e4c6c0d1f6149f3
```

I have shim and the Debian bootloader installed, and with secure boot enabled the machine fails to boot!

Some recent machines are shipping with the Microsoft UEFI 3rd Party CA certificate disabled by default. This stops shim from being trusted by the firmware, and it will cause the machine to fail to boot with a security violation. All the machines that ship in this state should have an option in the firmware menu to re-enable the certificate. In order to boot Debian with secure boot enabled, it is necessary to enable this option, if present. Beware that there is no standard location or wording, and it depends on the OEM and firmware where it is placed and what the text says.

Finding Debian's Secure Boot keys

In normal use, you're unlikely to ever need these directly. But, in case they're useful, here's where to find them.

- The Debian UEFI CA cert is the root of trust, and can be found in our shim source package - here in git.
- The public keys used directly for signing binaries are also found in git, in the ftp-team code-signing repo here.

MOK - Machine Owner Key

A key part of the shim design is to allow users to control their own systems. The Debian distro CA key is built in to the shim binary itself, but there is also an extra database of keys that can be managed by the user. The user keys are called **Machine Owner Key**, or MOK for short.

Keys can be added and removed in the MOK list by the user, entirely separate from the distro CA key. The **mokutil** utility can be used to help manage the keys here from Linux userland, but changes to the MOK keys may **only** be confirmed directly from the console at boot time. This removes the risk of userland malware potentially enrolling new keys and therefore bypassing the entire point of Secure Boot.

There are more docs online for how to work with MOK, for example:

- https://www.rodsbooks.com/efi-bootloaders/secureboot.html#initial_shim

DKMS and Secure Boot

Debian uses the Dynamic Kernel Module System (DKMS) to allow individual kernel modules to be upgraded without changing the whole kernel. Since DKMS modules are compiled individually on users own machines, it is not possible to sign DKMS modules using the Debian project's signing keys. Instead, modules built using DKMS will be signed using a machine owner key (MOK), which by default is located at `/var/lib/dkms/mok.key` with the corresponding public key at `/var/lib/dkms/mok.pub`. These keys are automatically generated automatically on the first build attempt of a DKMS module, but can be manually generated before that by running the following command (see `man 8 dkms`):

```
$ sudo dkms generate_mok
```

In case you get an error message that says `Error! Unknown action specified: ""`, it's due to a version mismatch, with the `dkms` package being an older version that didn't implement the `generate_mok` action. Don't worry: you can actually ignore this command, enable Secure Boot and let the driver setup take care of generating the MOK (you can check that it was generated by looking for the files above). After that, continue as below.

Regardless of whether the DKMS MOK keys are automatically or manually generated, the public key needs to be manually enrolled by running the following commands:

```
$ sudo mokutil --import /var/lib/dkms/mok.pub # prompts for one-time password
$ sudo mokutil --list-new # recheck your key will be prompted on next boot

<rebooting machine then enters MOK manager EFI utility: enroll MOK, continue, confirm, ent

$ sudo dmesg | grep cert # verify your key is loaded
```

If you encounter problems after enrolling the MOK, such as not being able to connect an external monitor, please run

```
$ sudo dpkg-reconfigure nvidia-kernel-dkms
```

This will rebuild the module and should fix every issue after a reboot.

Virtualbox Packages

Unlike Debian, Ubuntu places their auto-generated MOK at "`/var/lib/shim-signed/mok/`" which some software--such as Oracle's virtualbox package (see 989463)--expect to be present.

Note that using this same location may result in future conflicts, but results in fewer surprises due to poor coding (989463). Despite typically being poor practice, it is generally recommended to use the same location. More pedantic admins will likely prefer "`/root/.mok/`" ... with an understanding of expected headaches.

First make sure the key doesn't exist yet:

```
$ ls /var/lib/shim-signed/mok/
```

If you see the key there (consisting of the files `MOK.der`, `MOK.pem` and `MOK.priv`) then you can use these, rather than creating your own.

To create a custom MOK:

```
# mkdir -p /var/lib/shim-signed/mok/
# cd /var/lib/shim-signed/mok/
# openssl req -nodes -new -x509 -newkey rsa:2048 -keyout MOK.priv -outform DER -out MOK.de
# openssl x509 -inform der -in MOK.der -out MOK.pem
```

NOTE It seems (2022-06-21) that shim won't support adding a 4096 RSA key to the ?MokList (it might freezes when loading and verifying the grubx64.efi binary), so make sure you use a 2048 key for now.

Enrolling your key

Key enrollment requires issuing the request using `mokutil` and then confirming enrollment during a reboot. This is intended to ensure that only the device custodian can approve enrollment of a new MOK.

To enroll a key:

```
$ sudo mokutil --import /var/lib/shim-signed/mok/MOK.der # prompts for one-time password
```

At next reboot, the device firmware should launch it's MOK manager and prompt the user to review the new key and confirm it's enrollment, using the one-time password. Any kernel modules (or kernels) that have been signed with this MOK should now be loadable.

To verify the MOK was loaded correctly:

```
$ sudo mokutil --test-key /var/lib/shim-signed/mok/MOK.der
/var/lib/shim-signed/mok/MOK.der is already enrolled
```

Adding your key to DKMS

In order for `dkms` to automatically sign kernel modules, it must be told which key to sign the module with. This is done by adding two configuration values to "`/etc/dkms/framework.conf`", adjusting paths as required:

```
mok_signing_key="/var/lib/shim-signed/mok/MOK.priv"
mok_certificate="/var/lib/shim-signed/mok/MOK.der"
```

DKMS Sign Helper

If these values are provided and dkms is able to build modules but does not attempt to sign them, then it is likely that `sign_tool` is missing. This is more common in older and/or custom kernels.

In "`/etc/dkms/framework.conf`", add:

```
sign_tool="/etc/dkms/sign_helper.sh"
```

Create "`/etc/dkms/sign_helper.sh`" with:

```
/lib/modules/"$1"/build/scripts/sign-file sha512 /root/.mok/client.priv /root/.mok/client.
```

Manually signing kernels and kernel modules

The commands in this section use some shared environment variables:

```
$ VERSION="$(uname -r)"
$ SHORT_VERSION="$(uname -r | cut -d . -f 1-2)"
$ MODULES_DIR=/lib/modules/$VERSION
$ KBUILD_DIR=/usr/lib/linux-kbuild-$SHORT_VERSION
```

Using your key to sign your kernel

First, install sbsigntool

```
$ sbsign --key MOK.priv --cert MOK.pem "/boot/vmlinuz-$VERSION" --output "/boot/vmlinuz-$V
$ sudo mv "/boot/vmlinuz-$VERSION.tmp" "/boot/vmlinuz-$VERSION"
```

Using your key to sign modules

Building and signing modules is **independent** of building and signing your own kernel. Debian packages that provide kernel modules will typically sign the modules automatically using DKMS, but if you want to sign a kernel module manually, you can do so as shown below.

The example below shows how to sign modules. Debian's dkms package puts those in the `updates/dkms` subdirectory of the modules directory while Oracle's VirtualBox puts them in the `misc` subdirectory, so you'll have to go to the modules directory accordingly:

```
$ cd "$MODULES_DIR/updates/dkms" # For dkms packages
$ cd "$MODULES_DIR/misc" # For Oracle packages
```

Securely record the passphrase for the private key:

```
$ echo -n "Passphrase for the private key: "
$ read -s KBUILD_SIGN_PIN
$ export KBUILD_SIGN_PIN
```

Sign just the VirtualBox module:

```
$ sudo --preserve-env=KBUILD_SIGN_PIN "$KBUILD_DIR"/scripts/sign-file sha256 /var/lib/shim
```

Or sign all modules below the current directory:

```
$ find -name \*.ko | while read i; do sudo --preserve-env=KBUILD_SIGN_PIN "$KBUILD_DIR"/sc
```

If the modules are needed to boot your machine, make sure to update the initramfs, e.g. using

```
sudo update-initramfs -k all -u
```

Verifying if a module is signed

```
# modinfo vboxdrv
filename:       /lib/modules/5.10.0-9-amd64/misc/vboxdrv.ko
version:        6.1.28 r147628 (0x00320000)
license:        GPL
description:    Oracle VM VirtualBox Support Driver
author:         Oracle Corporation
srcversion:     282AFDD3CE09DCCD935FAF2
depends:
retpoline:      Y
name:           vboxdrv
vermagic:       5.10.0-9-amd64 SMP mod_unload modversions
sig_id:         PKCS#7
signer:         My Name
sig_key:        13:FE:C2:ED:A1:40:CE:70:1A:75:91:E5:4C:1F:5F:DA:BD:17:57:A9
sig_hashalgo:   sha256
signature:      0B:72:37:DD:10:97:F2:4F:DF:DF:52:27:38:88:63:7B:CC:2F:98:59:
                66:70:D1:22:94:05:62:77:E9:04:35:B4:2D:9F:6F:92:18:D5:98:C3:
                [etc.]
```

Disabling/re-enabling Secure Boot

In case it is difficult to control Secure Boot state through the EFI setup program, mokutil can also be used to disable or re-enable Secure Boot for operating systems loaded through shim and GRUB:

1. Run: `mokutil --disable-validation` or `mokutil --enable-validation`.
2. Choose a password between 8 and 16 characters long. Enter the same password to confirm it.
3. Reboot.
4. When prompted, press a key to perform MOK management.
5. Select "Change Secure Boot state".
6. Enter each requested character of your chosen password to confirm the change. Note that you have to press Return/Enter after each character.
7. Select "Yes".
8. Select "Reboot".

After doing `mokutil --disable-validation`, shim will disable secure boot and display "Booting in insecure mode". However, if one does that, it's possible that the kernel reboots just right when it start. To remove this behaving and re-enable secure boot validation, one way is to delete the EFI variable. This may depend on your EFI shell implementation, though for me, this is what worked:

```
efi> dmpstore -all -d MockSBState
```

Note that the -all is needed to access non-defaults guids variables.

Supported architectures and packages

On each architecture, Debian includes various packages containing signed binaries:

Name	amd64	i386	arm64	Signed by	Purpose
fwupd	fwupd-amd64-signed	fwupd-i386-signed	fwupd-arm64-signed	Debian	Tools to manage UEFI firmware updates automatically
fwupdate	fwupdate-amd64-signed	fwupdate-i386-signed	fwupdate-arm64-signed	Debian	Tools to manage UEFI firmware updates manually (removed after Buster in favour of fwupd)
grub	grub-efi-amd64-signed	grub-efi-ia32-signed	grub-efi-arm64-signed	Debian	GRUB boot loader
linux	linux-image-*-amd64 (*)	linux-image-*-686* (*)	linux-image-*-arm64 (*)	Debian	Linux kernel, various flavours
shim	shim-signed	shim-signed	shim-signed	Microsoft	Main shim binary
shim-helpers	shim-helpers-amd64-signed	shim-helpers-i386-signed	shim-helpers-arm64-signed	Debian	Shim helper binaries - ?MokManager and ?FallBack
systemd-boot	systemd-boot-efi-amd64-signed	-	systemd-boot-efi-arm64-signed	Debian	systemd-boot boot loader

(*) The various linux-image packages in Debian are now signed by default. The *unsigned* packages are called linux-image-*-unsigned.

Secure Boot setup with systemd-boot

GRUB is the default bootloader of choice in Debian, so the Secure Boot integration of systemd-boot has been designed with the intention of being a no-op if GRUB is installed on the target machine.

Starting with Trixie, users of Debian (and of the Debian Installer, since Trixie RC3) can optionally choose to use systemd-boot with shim. The integration point is the `systemd-boot` package and its postinst/prerm scripts. If GRUB packages are not installed, and both the `shim-signed` and `systemd-boot-efi-[amd64|arm64]-signed` packages are installed, then the maintainer scripts will run logic to install the appropriate EFI binaries to the ESP, and to add an EFI boot entry (named `Debian`) pointing to shim and making it the default entry. If either or both packages are removed, this logic runs in reverse and completely removes the binaries from the ESP, and deletes the EFI boot entry.

A dpkg trigger is declared by the `systemd-boot` package on the directory`/usr/lib/shim`, and on a named trigger `systemd-boot-signed`, so that the maintainer script can run the install/uninstall/update logic when the content of any of the involved packages changes.

Two EFI boot entries will be created by the systemd-boot package upon install. The default one will point to shim and chainload systemd-boot, this should remain the default and be used in all normal cases. The

second one points to systemd-boot directly and should only be used for recovery in case shim does not work for any reason and secure boot is disabled.

A user wanting to switch from GRUB to systemd-boot should do so by removing the GRUB packages and installing systemd-boot at the same time, for example on an amd64 system:

```
# apt install --allow-remove-essential systemd-boot grub-efi-amd64-signed-
```

To switch back to GRUB the inverse operation can be used:

```
# apt install systemd-boot- grub-efi-amd64-signed
```

To install systemd-boot on a system with no bootloader:

```
# apt install systemd-boot shim-signed
```

Testing UEFI Secure Boot

In preparation for the Debian 10 (Buster) release, some tests were performed to make sure that everything was ready. The information about that is is here.

Secure Boot limitations

By its very design, Secure Boot may affect or limit some things that users want to do.

If you want to build and run your own kernel (e.g. for development or debugging), then you will obviously end up making binaries that are not signed with the Debian key. If you wish to use those binaries, you will need to either sign them yourself and enroll the key used with MOK or disable Secure Boot.

Using Secure Boot activates "lockdown" mode in the Linux kernel. This disables various features that can be used to modify the kernel:

- Loading kernel modules that are not signed by a trusted key. By default, this will block out-of-tree modules including DKMS-managed drivers. However, you can create your own signing key for modules and add its certificate to the trusted list using MOK.
- Using kexec to start an unsigned kernel image.
- Hibernation and resume from hibernation.
- User-space access to physical memory and I/O ports.
- Module parameters that allow setting memory and I/O port addresses.
- Writing to MSRs through `/dev/cpu/*/msr`.
- Use of custom ACPI methods and tables.
- ACPI APEI error injection.

You can avoid this by disabling Secure Boot through the EFI setup program or through MOK.

Example docs from elsewhere:

- https://wiki.ubuntu.com/UEFI/SecureBoot/DKMS
- https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/7/html/Kernel_Administration_Guide/sect-signing-kernel-modules-for-secure-boot.html
- https://www.linuxjournal.com/content/take-control-your-pc-uefi-secure-boot

Infrastructure - how signing works in Debian

If you want to know the implementation details and the current discussions on improvements, see SecureBoot/Discussion.

Testing Secure Boot in a virtual machine

If you want to test Secure Boot in a virtual machine without having to deal with an actual machine, see SecureBoot/VirtualMachine.

arm64 problems

Debian initially had support for UEFI Secure Boot on arm64 for buster (10.0), but we found problems and had to abandon it for some time. It was re-enabled and functional again for the bookworm release (12.0). Things should work so long as a system is using **shim-signed** version **1.39** and **grub2** version **2.06-9** or greater.

If you want to know more, here's the background:

For a long time, shim and other EFI programs were difficult to build on arm64 compared to x86 platforms. Binutils for amd64 and i386 includes explicit support for creating programs in the PE/COFF binary format that EFI uses, but similar support did not exist for arm64 until during the bookworm development cycle.

In the past, shim developers added some local hacks into the shim package to generate a **mostly**-compliant PE/COFF EFI binary without this toolchain support, and that seemed to be sufficient for use. Everything seemed to work. **However**, during the development and testing phase of shim 15.3 and 15.4, we found found significant issues with this approach. New security features needed in https://wiki.debian.org/SecureBoot shim (SBAT) showed up severe problems with the lack of proper toolchain support. See https://github.com/rhboot/shim/issues/366 for more gory details. The old hacks around binutils were no longer sustainable.

Statistics tell us that very few people attempted to use arm64 Secure Boot with Debian at the time, so in the interests of releasing needed updates in a timely manner for other architectures we decided **temporarily** to disable Secure Boot support for Debian arm64.

Binutils 2.38 introduced support for PE/COFF generation on arm64 and the problem was solved.

Alternatives to Secure Boot

Debian's mission is to provide you with a free operating system, but many people don't even think about their proprietary boot firmware.

Free and open source boot firmware exists, such as coreboot and U-Boot, and several distributions of these offer their own security schemes, which are comparable and sometimes arguably better than UEFI Secure Boot; U-Boot itself can also provide its own UEFI implementation with Secure Boot, if configured properly.

We maintain a list of FOSS boot firmware projects at Firmware/Open

These firmware distributions often replace the proprietary UEFI firmware, on specific motherboards that they support.