Secure Boot

Secure Boot is a security feature found in the UEFI standard, designed to add a layer of protection to the pre-boot process: by maintaining a cryptographically signed list of binaries authorized or forbidden to run at boot, it helps in improving the confidence that the machine core boot components (boot manager, kernel, initramfs) have not been tampered with.

> **ATTENTION:** When using Secure Boot it's imperative to use it with disk encryption. If the storage device that stores the keys is not encrypted, anybody can read the keys and use them to sign bootable images, thereby defeating the purpose of using Secure Boot at all. Therefore, this guide will assume disk encryption is being used.

Preparations

To determine the current state of Secure Boot execute:

```
bootctl status
```

The output looks something like this:

```
System:
      Firmware: UEFI 2.70 (American Megatrends 5.17)
 Firmware Arch: x64
   Secure Boot: enabled (user)
  TPM2 Support: yes
  Measured UKI: yes
  Boot into FW: supported

...
```

In order to proceed you need to set your firmware's Secure Boot mode into "setup" mode. This can usually be achieved by wiping the key store of the firmware. Refer to your mainboard's user manual on how to do this.

Installation

For the most straight-forward Secure Boot toolchain install `sbctl`:

```
pacman -S sbctl
```

It tremendously simplifies generating Secure Boot keys, loading keys into firmware and signing kernel images.

Generating keys

> **SEE ALSO:** The Meaning of all the UEFI Keys

Secure Boot implementations use these keys:

Key Type	Description
Platform Key (PK)	Top-level key
Key Exchange Key (KEK)	Keys used to sign Signatures Database and Forbidden Signatures Database updates
Signature Database (db)	Contains keys and/or hashes of allowed EFI binaries
Forbidden Signatures Database (dbx)	Contains keys and/or hashes of denylisted EFI binaries

To generate new keys and store them under `/var/lib/sbctl/keys`:

```
sbctl create-keys
```

Unified Kernel Image

A unified kernel image (UKI) combines an EFI stub image, CPU microcode, kernel command line and an initramfs into a single file that can be read and executed by the machine's UEFI firmware. It also makes it easier to sign for secure boot as there will be only a single file to sign.

Starting with v31, `mkinitcpio` is able to create UKIs out-of-the-box. The maintainers of `sbctl` also recommend using the system's initramfs generation tool instead of `sbctl bundle`. Additionally, `sbctl` comes with `mkinitcpio` hooks that sign kernel images automatically when they are generated during a rebuild.

Starting with v39, `mkinitcpio` will use `systemd-ukify` if it is installed. This is the preferred way of generating UKIs. As `systemd-ukify` is not part of the `systemd` package, you'll have to install it manually:

```
pacman -S systemd-ukify
```

To make `mkinitcpio` generate UKIs, edit the appropriate `*.preset` file for your kernel in `/etc/mkinitcpio.d/`:

- comment out the `default_image` and `fallback_image` lines (as they won't be needed)
- uncomment the `default_uki` and `fallback_uki` lines (prompts `mkinitcpio` to switch to UKI generation)
- point the file path to somewhere on your EFI System Partition (e.g. `/efi`)

> **NOTE:** `mkinitcpio` will automatically source command line parameters from files in `/etc/cmdline.d/*.conf` or a complete single command line specified in `/etc/kernel/cmdline`. If you need different images to use different kernel command line parameters, the `*_options` line in the `*.preset` allows you to pass additional arguments to `mkinitcpio`, i.e. the `--cmdline` argument to point it to a different file containing a different set of kernel command line parameters.

> **NOTE:** Placing the UKI under `/efi/EFI/Linux/` allows `systemd-boot` to automatically detect images and list them without having to specifically create boot entries for them.

> **WARNING:** If there are no options specified in either `/etc/kernel/cmdline` or a drop-in file in `/etc/cmdline.d/*.conf`, then `mkinitcpio` will fallback to reading the command line for the currently booted system from `/proc/cmdline`. If you're booted into the Arch installation environment, this will most likely leave you with an unbootable system. **Set at least one command line option in one of the above locations!**

A `*.preset` file edited for UKI generation could look something like this:

```
# mkinitcpio preset file for the 'linux' package

#ALL_config="/etc/mkinitcpio.conf"
ALL_kver="/boot/vmlinuz-linux"

PRESETS=('default' 'fallback')

#default_config="/etc/mkinitcpio.conf"
#default_image="/boot/initramfs-linux.img"
default_uki="/efi/EFI/Linux/arch-linux.efi"
#default_options="--splash /usr/share/systemd/bootctl/splash-arch.bmp"

#fallback_config="/etc/mkinitcpio.conf"
#fallback_image="/boot/initramfs-linux-fallback.img"
```

```
fallback_uki="/efi/EFI/Linux/arch-linux-fallback.efi"
fallback_options="-S autodetect --cmdline /etc/kernel/cmdline_fallback"
```

Kernel Command Line Parameters

`mkinitcpio` automatically looks for kernel command line parameters specified in `/etc/cmdline.d/*.conf` as drop-in files or `/etc/kernel/cmdline` as a single file.

> **WARNING:** If there are no options specified in either `/etc/kernel/cmdline` or a drop-in file in `/etc/cmdline.d/*.conf`, then `mkinitcpio` will fallback to reading the command line for the currently booted system from `/proc/cmdline`. If you're booted into the Arch installation environment, this will most likely leave you with an unbootable system. **Set at least one command line option in one of the above locations!**

First create the directory and open a new file in there:

```
mkdir /etc/cmdline.d
nano /etc/cmdline.d/root.conf
```

The parameters to include depend on the kind of initramfs used. You can use any of the persistent block device naming schemes to pass the device. You also need to specify a mapper name under which the decrypted root file system should be made available for mounting.

You can obtain the block device identifier for the LUKS container, e.g. its UUID, with `blkid` (using `/dev/nvme0n1p3` as an example):

> **NOTE:** Pressing `Ctrl + T` inside `nano` allows you to paste the result of a command at the current cursor position.

```
blkid -s UUID -o value /dev/nvme0n1p3
```

busybox

At minimum your kernel command line parameters should look like this:

```
cryptdevice=UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX:root root=/dev/mapper/root rw
```

This tells the kernel to unlock the LUKS device at the UUID specified and give it the device mapper name `root`. This makes the decrypted contents available under `/dev/mapper/root`, which is a

persistent name and can be used as the root file system by the kernel.

systemd

When using a systemd-based initramfs, there are two ways of mounting an encrypted file system: manual and GPT partition auto-mounting.

The manual way is via the command line parameter `rd.luks` to specify an encrypted device, similar to the busybox way.

```
rd.luks.name=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX=root root=/dev/mapper/root rw
```

If you would rather have a config file with all your encrypted block devices, you can create a file named `/etc/crypttab.initramfs` to specify your encrypted devices which will become `/etc/crypttab` in your initramfs and tell the kernel which devices to unlock during boot (see `crypttab(5)` for details on the syntax):

```
# <name>    <device>                                  <passphrase>    <options>
root        UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX
```

This allows you to omit any `rd.luks` parameters, which leaves you with a kernel command line that looks like this:

```
root=/dev/mapper/root rw
```

Alternatively, a systemd-based initramfs allows for device auto-discovery. Instead of specifying the root file system device directly, you can specify this in your `/etc/crypttab.initramfs`:

```
# <name>    <device>                                  <passphrase>    <options>
root        /dev/gpt-auto-root-luks
```

This is a symbolic link to the encrypted root partition as identified by its **GPT partition type.** When doing this, the decrypted device will also be auto-discovered and auto-mounted, leaving you with only the `rw` kernel command line parameter, indicating the file systems should be mounted writable.

> **ATTENTION:** Keep the specialties of your chosen root file system in mind, e.g. when using btrfs, you will still need to supply the subvolume and any other file system options as a kernel command line parameter, as auto-discovery and auto-mounting uses default file system mounting options: `rootflags=compress=zstd,subvol=@`.

NOTE: By default, dm-crypt does not allow TRIM for SSDs for security reasons (information leak). To override this behavior:

- **busybox:** append `:allow-discards` to the device mapper name, i.e. `UUID=XXX...XXX:root:allow-discards`
- **systemd:** do one of the following:
  - add `rd.luks.options=discard` as an additional kernel command line parameter
  - specify the `discard` option in `/etc/crypttab.initramfs` in the options field

Kernel Lockdown Mode

To further strengthen security you might want to consider using the kernel's built-in Lockdown Mode. When engaging lockdown, access to certain features and facilities is blocked, even for the root user. This helps prevent Secure Boot from being bypassed through a compromised system, for example by editing EFI variables or replacing the kernel at runtime.

Lockdown Mode knows two modes of operation:

- `integrity`: kernel features that allow userland to modify the running kernel are disabled (kexec, bpf)
- `confidentiality`: kernel features that allow userland to extract confidential information from the kernel are also disabled

The recommended mode is `integrity`, as `confidentiality` can break certain applications (e.g. Docker).

To enable Lockdown Mode, set the `lockdown=MODE` kernel command line parameter with your preferred mode.

Enroll keys in firmware

WARNING: Replacing the platform keys with your own can end up bricking your machine, making it impossible to get into the UEFI/BIOS settings to rectify the situation. This is due to the fact that some device firmware (OpROMs, e.g. GPU firmware), that gets executed during boot, may be signed using Microsoft's keys. Run `sbctl enroll-keys --microsoft` if you're unsure if this applies to you (enrolling Microsoft's Secure Boot keys alongside your own custom ones) or include the TPM Event Log with `sbctl enroll-keys --tpm-eventlog` (if your machine has a TPM and you don't need or want Microsoft's keys) to prevent bricking your machine.

> **ATTENTION:** Make sure your firmware's Secure Boot mode is set to `setup` mode! You can do this by going into your firmware settings and wiping the factory default keys. Additionally, keep an eye out for any setting that auto-restores the default keys on system start.

> **TIP:** If you plan to dual-boot Windows, run `sbctl enroll-keys --microsoft` to enroll Microsoft's Secure Boot keys along with your own custom keys.

To enroll your keys, simply:

```
sbctl enroll-keys
```

Automated signing of UKIs

`sbctl` comes with a hook for `mkinitcpio` which runs after it has rebuilt an image. Manually specifying images to sign is therefore entirely optional.

Signing the Bootloader

> **NOTE:** This is the manual method. If you also want to automate the bootloader update process, skip to the section below.

If you plan on using a boot loader, you will also need to add its `*.efi` executable(s) to the `sbctl` database, e.g. `systemd-boot`:

```
sbctl sign --save /efi/EFI/BOOT/BOOTX64.EFI
sbctl sign --save /efi/EFI/systemd/systemd-bootx64.efi
```

Upon system upgrades, `pacman` will call `sbctl` to sign the files listed in the `sbctl` database.

Automate `systemd-boot` updates and signing

`systemd` comes with a `systemd-boot-update.service` unit file to automate updating the bootloader whenever `systemd` is updated. However, it only updates the bootloader **after** a reboot, by which time `sbctl` has already run the signing process. This would necessitate manual intervention.

Recent versions of `bootctl` look for a `.efi.signed` file before a regular `.efi` file when copying bootloader files during `install` and `update` operations. So to integrate better with the auto-update

functionality of `systemd-boot-update.service`, the bootloader needs to be signed ahead of time.

```
sbctl sign --save \
  -o /usr/lib/systemd/boot/efi/systemd-bootx64.efi.signed \
  /usr/lib/systemd/boot/efi/systemd-bootx64.efi
```

This will add the source and target file paths to `sbctl`'s database. The pacman hook included with `sbctl` will trigger whenever a file in `usr/lib/**/efi/*.efi*` changes, which will be the case when `systemd` is updated and a new version of the unsigned bootloader is written to disk at `/usr/lib/systemd/boot/efi/systemd-bootx64.efi`.

Finally, enable the `systemd-boot-update.service` unit:

```
systemctl enable systemd-boot-update
```

Now when `systemd` is updated the **signed** version of the `systemd-bootx64.efi` booloader will be copied to the ESP after a reboot, completely automating the bootloader update and signing process!

Revision #29
Created 12 September 2021 12:50:00 by Sebin
Updated 10 June 2025 21:40:33 by Sebin