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.

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 to proceed. 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 `/usr/share/secureboot/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 machines 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`.

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

- uncomment the `default_uki` and `fallback_uki` lines
- point the file path to somewhere on your EFI System Partition (e.g. `/efi`, `/boot` or `/boot/efi`)

>

> **NOTE:** `mkinitcpio` automatically sources `/etc/kernel/cmdline` for the included kernel command line arguments. If you want the fallback image to receive a different set of kernel command line arguments, specify a different file path in `fallback_options` with the `--cmdline` argument. It also sources drop-in files under `/etc/cmdline.d/` during UKI generation. However, the latter won't allow you to pass different command line arguments for the default and fallback image.

> **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.

```
# 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"                              # NEW
#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"                    # NEW
fallback_options="-S autodetect --cmdline /etc/kernel/cmdline_fallback"    # NEW
```

Kernel Command Line Parameters

As `mkinitcpio` sources command line parameters from a specific file by default, saving them to that file further streamlines the generation process.

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

```
mkdir /etc/kernel
nano /etc/kernel/cmdline
```

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/sda1` 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/sda1
```

Continue to specify additional kernel command line parameters you need. At minimum it should look like this:

- busybox:

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

- systemd:

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

> **TIP:** You can further simplify this by using a systemd-based initramfs. Create a file named `/etc/crypttab.initramfs` and specify your encrypted devices in there (same syntax as regular `/etc/crypttab`, see crypttab(5)):
>
> ```
> # <name>   <device>                                <passphrase>   <options>
> cryptroot  UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX            discard
> ```
>
> This allows you to omit any `rd.luks` parameters, which leaves you with a kernel command line that looks like this:
>
> ```
> root=/dev/mapper/cryptroot rw
> ```

> **ATTENTION:** Keep the specialties of your chosen root file system in mind, e.g. if using btrfs you also need to supply the subvolume that should be mounted: `rootflags=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` after the device mapper name
- **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

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

Next, add the images to the list of files to be signed (one at a time):

```
sbctl sign --save /efi/EFI/Linux/arch-linux.efi
sbctl sign --save /efi/EFI/Linux/arch-linux-fallback.efi
```

The `sbctl` package comes with a pacman hook to execute `sbctl sign-all -g` on kernel upgrades or installs. The UKIs are ready to be booted directly by the UEFI firmware (EFISTUB booting) or via a bootloader like `grub`, `systemd-boot` or `rEFInd`.

> **ATTENTION:** Currently, the `sbctl` package also provides a post mkinitcpio hook which runs `sbctl` after every kernel build. This means with the default `linux` kernel installed, `sbctl` will run **at least three times**, twice for each time `mkinitcpio` runs during pacman package upgrades and once after pacman finishes. The usefulness of the hook has been disputed. A patch has been submitted.
>
> For the time being, comment out the line calling `sbctl sign-all -g` in the hook file: `/usr/lib/initcpio/post/sbctl`

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 re-sign the files listed in `sbctl`'s 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 #22
Created 12 September 2021 12:50:00 by Sebin
Updated 11 October 2024 20:51:27 by Sebin