

Installation

Laying the foundation

- [Base System](#)
- [Time Zone & Locale](#)
- [Network](#)
- [Root Password](#)
- [sudo](#)
- [zsh](#)
- [Add User](#)
- [AUR Helper](#)
- [initramfs](#)
- [zram](#)
- [Secure Boot](#)
- [Boot Loader](#)

Base System

Setting up mirrors

The Arch installation environment comes with `reflector`, a tool that generates mirror lists for `pacman`. At boot time, `reflector` is executed once to include the most recently synced mirrors and sorts them by download rate. This file will be copied to the installation destination later on.

`reflector` allows for a few filtering options:

Filter	Description
<code>--age n</code>	Only return mirrors that have synchronized in the last n hours.
<code>--country NAME</code>	Restrict mirrors to selected countries, e.g. <code>France,Germany</code> (check available with <code>--list-countries</code>)
<code>--fastest n</code>	Return the n fastest mirrors that meet the other criteria. Do not use without filters!
<code>--latest n</code>	Limit the list to the n most recently synchronized servers.
<code>--score n</code>	Limit the list to the n servers with the highest score.
<code>--number n</code>	Return at most n mirrors.
<code>--protocol PROTO</code>	Restrict protocol used by mirrors. Either <code>https</code> , <code>http</code> , <code>ftp</code> or a combination (comma-separated)

To have `reflector` generate a list of mirrors from Germany, which synced in the past 12 hours and use HTTPS for transfer:

```
reflector --country Germany --age 12 --protocol https --save /etc/pacman.d/mirrorlist
```

Parallel downloads

By default, `pacman` downloads packages one-by-one. If you have a fast internet connection, you can configure `pacman` to download packages in parallel, which can speed up installation significantly.

Open `/etc/pacman.conf`, uncomment the line `#ParallelDownloads = 5` and set it to a value of your preference:

```
...
# Misc options
#UseSyslog
#Color
#NoProgressBar
CheckSpace
#VerbosePkgLists
ParallelDownloads = 10
#DisableSandbox
...
```

Alternatively, replace the settings directly with `sed` (e.g. setting 10 parallel downloads at a time):

```
sed -i "/etc/pacman.conf" -e "s|^#ParallelDownloads.*|&\nParallelDownloads = 10|"
```

Installing base packages

The absolute minimum set of packages required to install Arch Linux onto a machine is as follows:

```
pacstrap /mnt base linux linux-firmware
```

However, this selection lacks the tooling required for file systems, RAID, LVM, special firmware for devices not included with `linux-firmware`, networking software, a text editor or packages necessary to access documentation. It also lacks CPU microcode packages with stability and security updates.

The following table contains additional packages you most likely want to append to the above `pacstrap` command:

Package	Description
<code>base</code>	Absolute essentials (required)
<code>linux</code>	Vanilla Linux kernel and modules, with a few patches applied (required)
<code>linux-hardened</code>	A security-focused Linux kernel applying a set of hardening patches to mitigate kernel and userspace exploits
<code>linux-lts</code>	Long-term support (LTS) Linux kernel and modules
<code>linux-zen</code>	Result of a collaborative effort of kernel hackers to provide the best Linux kernel possible for everyday systems
<code>linux-firmware</code>	Device firmware files, e.g. WiFi (required)

Package	Description
<code>intel-ucode</code>	Intel CPU microcode (required, if on Intel)
<code>amd-ucode</code>	AMD CPU microcode (required, if on AMD)
<code>btrfs-progs</code>	Userspace tools to manage btrfs filesystems
<code>dosfstools</code>	Userspace tools to manage FAT filesystems
<code>exfatprogs</code>	Userspace tools to manage exFAT filesystems
<code>f2fs-tools</code>	Userspace tools to manage F2FS filesystems
<code>e2fsprogs</code>	Userspace tools to manage ext2/3/4 filesystems
<code>jfsutils</code>	Userspace tools to manage JFS filesystems
<code>nilfs-utils</code>	Userspace tools to manage NILFS2 filesystems
<code>ntfs-3g</code>	Userspace tools to manage NTFS filesystems
<code>udftools</code>	Userspace tools to manage UDF filesystems
<code>xfsprogs</code>	Userspace tools to manage XFS filesystems
<code>lvm2</code>	Userspace tools for Logical Volume Management
<code>cryptsetup</code>	Userspace tools for encrypting storage devices (LUKS)
<code>networkmanager</code>	Comprehensive network management and configuration suite
<code>nano</code>	Console text editor
<code>man</code>	Read documentation (man uals)
<code>sudo</code>	Execute commands with elevated privileges

CAUTION: If you have an AMD CPU, include the `amd-ucode` package. If you have an Intel CPU, include the `intel-ucode` package!

ATTENTION: Include the `cryptsetup` package if you've encrypted your disks!

A desirable selection of packages for a base system with an AMD CPU, btrfs filesystem, UEFI ESP, LUKS disk encryption, a basic text editor, a network manager and tools for system maintenance as regular user would look something like this:

```
pacstrap /mnt base linux linux-firmware amd-ucode btrfs-progs dosfstools cryptsetup nano
networkmanager sudo
```

Generate the `fstab` containing information about which storage devices should be mounted at boot:

```
# Generate fstab referencing UUIDs of devices/partitions
genfstab -U /mnt >> /mnt/etc/fstab
```

Switch into the newly installed system with `arch-chroot` and continue setting it up:

```
arch-chroot /mnt
```

Time Zone & Locale

Time zone

Create a symbolic link to your local time zone at `/etc/localtime` and sync the time with the local hardware clock:

```
ln -sf /usr/share/zoneinfo/Europe/Berlin /etc/localtime  
hwclock --systohc
```

Localization

Edit `/etc/locale.gen` and uncomment `en_US.UTF-8 UTF-8` and other desired locales (prefer UTF-8):

```
nano /etc/locale.gen
```

NOTE: You can search in `nano` using CTRL + W.

Generate the locales by running:

```
locale-gen
```

Set which locales and keyboard layout the system should use for messages and documentation (`man` pages):

```
echo "LANG=de_DE.UTF-8" > /etc/locale.conf  
echo "KEYMAP=de-latin1" > /etc/vconsole.conf
```

Network

Set up the default host name of the machine as well as `localhost`:

NOTE: `sebin-desktop` is used as an example here. Set `$HOSTNAME` to whatever you like.

```
# Define an environment variable containing the desired hostname
export HOSTNAME='sebin-desktop'

# Set the hostname of the machine
echo "$HOSTNAME" > /etc/hostname

# Set localhost to resolve to the machine's loopback address
echo "127.0.0.1 localhost" >> /etc/hosts
echo "::1 localhost" >> /etc/hosts
echo "127.0.1.1 $HOSTNAME.localdomain $HOSTNAME" >> /etc/hosts
```

Set wireless region

If your machine has Wi-Fi it is advisable to set the region for wireless radio waves to comply with local regulations. Not doing this will limit you to 2,4 GHz Wi-Fi.

To set your region temporarily:

```
pacman -S iw
iw reg set DE # Set region to e.g. Germany
```

To set it permanently, install `wireless-regdb` and uncomment the line with your country in the file `/etc/conf.d/wireless-regdom`.

Network manager

Previously we installed NetworkManager as our default network mangaging software. GNOME and KDE have out of the box support for managing network connections in their settings dialogs in a graphical manner. Both rely on NetworkManager.

Enable NetworkManager to start at boot:

```
systemctl enable NetworkManager
```

Using `iwd` as the Wi-Fi backend (optional)

By default NetworkManager uses `wpa_supplicant` for managing Wi-Fi connections.

`iwd` (iNet wireless daemon) is a wireless daemon for Linux written by Intel. The core goal of the project is to optimize resource utilization by not depending on any external libraries and instead utilizing features provided by the Linux Kernel to the maximum extent possible.

To enable the [experimental iwd backend](#), first install `iwd` and then create the following configuration file:

```
mkdir /etc/NetworkManager/conf.d
nano /etc/NetworkManager/conf.d/wifi_backend.conf
```

With the following contents:

```
[device]
wifi.backend=iwd
```

`systemd-resolved` for DNS name resolution

`systemd-resolved` is a `systemd` service that provides network name resolution to local applications via a D-Bus interface, the `resolve` NSS service, and a local DNS stub listener on `127.0.0.53`.

Benefits of using `systemd-resolved` include:

- `resolvectl` as the primary single command for interfacing with the network name resolver service
- A system-wide DNS cache for speeding up subsequent name resolution requests
- Split DNS when using VPNs, which can help in preventing DNS leaks when connecting to multiple VPNs (See [Fedora Wiki](#) for a detailed explanation why this is important)
- Integrated DNSSEC capabilities to verify the authenticity and integrity of name resolution requests, e.g. to prevent [cache poisoning/DNS hijacking](#)
- DNS over TLS for further securing name resolution requests by encrypting them, improving privacy (not to be confused with DNS over HTTPS)

To use `systemd-resolved` enable the respective unit:


```
systemctl enable systemd-resolved
```

To provide domain name resolution for software that reads `/etc/resolv.conf` directly, such as web browsers and GnuPG, `systemd-resolved` has four different modes for handling the file

- **stub:** a symlink to the `systemd-resolved` managed file `/run/systemd/resolve/stub-resolv.conf` containing only the stub resolver and search domains
- **static:** a symlink to the static `systemd-resolved` owned file `/usr/lib/systemd/resolv.conf` containing only the stub resolver, but no search domains
- **uplink:** a symlink to the `systemd-resolved` managed file `/run/systemd/resolve/resolv.conf` containing all upstream DNS servers known to `systemd-resolved`, effectively bypassing the stub resolver
- **foreign:** an external tool managing system-wide DNS entries for `systemd-resolved` to derive its DNS configuration from

The recommended mode is *stub*.

ATTENTION: A few notes about setting this up:

- Failure to properly configure `/etc/resolv.conf` will result in broken DNS resolution!
- Attempting to symlink `/etc/resolv.conf` whilst inside `arch-chroot` will not be possible, since the file is bind-mounted from the archiso live system. In this case, create the symlink from *outside* `arch-chroot`:

```
ln -sf ../run/systemd/resolve/stub-resolv.conf /mnt/etc/resolv.conf
```

- Some DHCP and VPN clients use the `resolvconf` program to set name server and search domains (see [this list](#)). For these, you also need to install the `systemd-resolvconf` package to provide a `/usr/bin/resolvconf` symlink.

This propagates the `systemd-resolved` managed configuration to all clients. To use it, replace `/etc/resolv.conf` with a symbolic link to it:

```
ln -sf ../run/systemd/resolve/stub-resolv.conf /etc/resolv.conf
```

When set up this way, NetworkManager automatically picks up `systemd-resolved` for network name resolution.

Fallback DNS servers

If `systemd-resolved` does not receive DNS server addresses from the network manager and no DNS servers are configured manually, then `systemd-resolved` falls back to a hardcoded list of DNS servers.

The fallback order is:

1. Cloudflare
2. Quad9 (without filtering and without DNSSEC)
3. Google

ATTENTION: Depending on your use-case, you might not want to route all your DNS traffic through the pre-determined fallback servers for privacy reasons. Do your own research on fallback DNS servers that you want to trust.

Fallback addresses can be manually set in a drop-in config file, e.g.

`/etc/systemd/resolved.conf.d/fallback_dns.conf`:

```
[Resolve]
FallbackDNS=127.0.0.1 ::1
```

To disable the fallback DNS functionality set the `FallbackDNS` option without specifying any addresses:

```
[Resolve]
FallbackDNS=
```

DNSSEC

WARNING: DNSSEC support in `systemd-resolved` is considered experimental and incomplete.

DNSSEC is an extension to the DNS system that verifies DNS entries via authentication and data integrity checks to prevent DNS cache poisoning, but *does not **encrypt** DNS queries*. For actually encrypting your DNS traffic, see the section below.

`systemd-resolved` can be configured to use DNSSEC for validation of DNS requests. It can be configured in three modes:

Setting	Description
<code>allow-downgrade</code>	Validate DNSSEC only if the upstream DNS server supports it
<code>true</code>	Always validate DNSSEC, breaking DNS resolution if the server does not support it
<code>false</code>	Disable DNSSEC validation entirely

Set up DNSSEC in a drop-in config file, e.g. `/etc/systemd/resolved.conf.d/dnssec.conf`:

```
[Resolve]
DNSSEC=allow-downgrade
```

DNS over TLS

DNS over TLS (DoT) is a security protocol for encrypting DNS queries and responses via Transport Layer Security (TLS), thereby increasing privacy and security by preventing eavesdropping on DNS requests by internet service providers and malicious actors in man-in-the-middle attack scenarios.

DNS over TLS in `systemd-resolved` is disabled by default. To enable validation of your DNS provider's server certificate, include their hostname in the `DNS` setting in the format `ip_address#hostname` and set `DNSoverTLS` to one of three modes:

Setting	Description
<code>opportunistic</code>	Attempt DNS over TLS when possible and fall back to unencrypted DNS if the server does not support it
<code>true</code>	Always use DNS over TLS, breaking resolution if the server does not support it
<code>false</code>	Disable DNS over TLS entirely

ATTENTION: When setting `DNSoverTLS=opportunistic` `systemd-resolved` will try to use DNS over TLS and if the server does not support it fall back to regular DNS. Note, however, that this opens you to "downgrade" attacks, where an attacker might be able to trigger a downgrade to non-encrypted mode by synthesizing a response that suggests DNS over TLS was not supported.

WARNING: If setting `DNSoverTLS=yes` and the server provided in `DNS=` does not support DNS over TLS *all DNS requests will fail!*

To enable DNS over TLS system-wide for all connections, add your DNS over TLS capable servers in a drop-in config file, e.g. `/etc/systemd/resolved.conf.d/dns_over_tls.conf`:

```
[Resolve]
DNS=9.9.9.9#dns.quad9.net 149.112.112.112#dns.quad9.net [2620:fe::fe]#dns.quad9.net
[2620:fe::9]#dns.quad9.net
DNSoverTLS=yes
```

Alternatively, you can use drop-in configuration files for NetworkManager to instruct it to use DNS over TLS per connection. You can save this as a drop-in configuration file under

`/etc/NetworkManager/conf.d/dns_over_tls.conf` to apply it to current and future connections or on a per-connection basis to an existing connection profile under `/etc/NetworkManager/system-connections/*.nmconnection` (as `root`).

There's three possible values:

- 2 = DNS over TLS always on (fail if DoT is unavailable)
- 1 = opportunistic DNS over TLS (downgrades to unencrypted DNS if DoT is unavailable)
- 0 = never use DNS over TLS

Add or modify

```
[connection]
dns-over-tls=2
```

Multicast DNS

`systemd-resolved` is capable of working as a [multicast DNS](#) (mDNS) resolver and responder. The resolver provides hostname resolution using a "*hostname.local*" naming scheme.

mDNS support in `systemd-resolved` is enabled by default. For a given connection, mDNS will only be activated if both mDNS in `systemd-resolved` is enabled, and if the configuration for the currently active network manager enables mDNS for the connection.

The `MulticastDNS` setting in `systemd-resolved` can be set to one of the following:

Setting	Description
<code>resolve</code>	Only enables resolution support, but responding is disabled
<code>true</code>	Enables full mDNS responder and resolver support
<code>false</code>	Disables both mDNS responder and resolver

ATTENTION: If you plan on using `systemd-resolved` as mDNS resolver and responder consider the following:

- Some desktop environments have the `avahi` package as a dependency. To prevent conflicts, `disable` or `mask` both `avahi-daemon.service` and `avahi-daemon.socket`
- If you plan on using a firewall, make sure UDP port `5353` is open

To enable mDNS for a connection managed by NetworkManager tell `nmcli` to modify an existing connection:

```
nmcli connection modify CONNECTION_NAME connection.mdns yes
```

TIP: The default for all NetworkManager connections can be set by creating a configuration file in `/etc/NetworkManager/conf.d/` and setting `connection.mdns=2` (equivalent to "yes") in the `[connection]` section.

```
[connection]
connection.mdns=2
```

Avahi

Avahi implements zero-configuration networking (zeroconf), allowing for multicast DNS/DNS-SD service discovery. This enables programs to publish and discover services and hosts running on a local network, e.g. network file sharing servers, remote audio devices, network printers, etc.

Some desktop environments pull in the `avahi` package as a dependency. It enables their file manager to scan the network for services and make them easily accessible.

ATTENTION: If you plan on using `avahi` as mDNS resolver and responder consider the following:

- You need to disable mDNS in `systemd-resolved`. You can do so in a drop-in config file, e.g. `/etc/systemd/resolved.conf.d/mdns.conf`:

```
[Resolve]
MulticastDNS=false
```

- If you plan on using a firewall, make sure UDP port `5353` is open

Avahi provides local hostname resolution using a "*hostname.local*" naming scheme. To use it, install the `avahi` and `nss-mdns` package and enable Avahi:

```
pacman -S avahi nss-mdns
systemctl enable avahi-daemon
```

Then, edit the file `/etc/nsswitch.conf` and change the `hosts` line to include `mdns_minimal` `[NOTFOUND=return]` before `resolve` and `dns`:

```
hosts: mymachines mdns_minimal [NOTFOUND=return] resolve [!UNAVAIL=return] files myhostname  
dns
```

To discover services running in your local network:

```
avahi-browse --all --ignore-local --resolve --terminate
```

To query a specific host for the services it advertises:

```
avahi-resolve-host-name hostname.local
```

Avahi also includes the `avahi-discover` graphical utility that lists the various services on your network.

Root Password

Set the password for the `root` user:

This password should differ from the regular user password for security reasons.

In the case of system recovery operations the `root` user comes into play, e.g. when the kernel fails to mount the root file system or system maintenance via `chroot` is needed.

sudo

`sudo` is the standard tool for gaining temporary system administrator privileges on Linux to perform administrative tasks. This eliminates the need to change the current user to `root` to perform these tasks.

To allow regular users to execute commands with elevated privileges, the configuration for `sudo` needs to be modified to allow this.

`sudo` supports configuration drop-in files in `/etc/sudoers.d/`. Using these makes it easy to modularize the configuration and remove offending files, if something goes wrong.

TIP: File names starting with `.` or `~` will get ignored. Use this to turn off certain configuration settings if you need to.

WARNING: Drop-in files are just as fragile as `/etc/sudoers`! It is therefore strongly advised to always use `visudo` when creating or editing `sudo` config files, as it will check for syntax errors. Failing to do so will risk rendering `sudo` inoperable!

Create a new drop-in file at:

```
EDITOR=nano visudo /etc/sudoers.d/01_wheel
```

The contents of the drop-in file are as follows:

```
## Allow members of group wheel to execute any command
%wheel ALL=(ALL:ALL) ALL
```

Save and exit.

Now every user who is in the `wheel` user group is allowed to run any command as `root`.

zsh

`zsh` is a modern shell with lots of customizability and features. Install the following packages:

```
pacman -S zsh zsh-autosuggestions zsh-completions zsh-history-substring-search zsh-syntax-highlighting
```

Package	Description
<code>zsh-autosuggestions</code>	Suggests commands as you type based on history and completions
<code>zsh-completions</code>	Additional completion definitions for <code>zsh</code>
<code>zsh-history-substring-search</code>	Type any part of any command from history and cycle through matches
<code>zsh-syntax-highlighting</code>	Highlights commands whilst they are typed, helping in reviewing commands before running them

Add User

It is advised to add a regular user account for day to day usage.

Add a new user, create a home directory, add them to the wheel group, set their default shell to zsh:

```
useradd -mG wheel -s /bin/zsh sebin
```

Set a password for the new user:

```
passwd sebin
```

AUR Helper

An AUR helper is a tool that automates the process of installing packages from the [Arch User Repository](#).

It does this by automating the following tasks:

- search the AUR for published packages
- resolve dependencies for AUR packages
- retrieval and build of AUR packages
- show user comments
- submission of AUR packages

AUR packages are distributed in the form of `PKGBUILD`s that contain information on how the package needs to be built, what dependencies it needs and all the usual metadata associated with every other Arch Linux package.

[Arch Wiki](#) has a list of AUR helpers with comparison tables

Installation

The installation procedure for any AUR helper is largely the same, as they are all published on the AUR itself.

Building packages from the AUR manually will at minimum require the `base-devel` and `git` packages:

```
pacman -S base-devel git
```

ATTENTION: If you're currently logged in as the `root` user, you need to switch to a regular user profile with `su username`, as `makepkg` will not allow you to run it as `root`.

Change to a temporary directory, clone the AUR helper of your choice with `git`, change into the newly created directory and call `makepkg` to build and install it, e.g. `yay`:

```
cd /tmp
git clone https://aur.archlinux.org/yay
cd yay
makepkg -si
```

`makepkg -si` will prompt you to install any missing dependencies for your chosen AUR helper, i.e. `go` for `yay`, `rust` for `paru`, etc. and call `pacman` to install the helper for you after the build has finished.

Configuration

`makepkg` can be configured to make better use of available system resources, improving build times and efficiency.

One of these optimizations is instructing `makepkg` to pass specific options to compilers. You can either edit the main configuration file of `makepkg` at `/etc/makepkg.conf` or supply a drop-in config file in `/etc/makepkg.conf.d/*.conf` — the latter is recommended in case building starts to act strangely and you want to quickly be able to revert changes by deleting drop-in config files.

Optimizing builds

By default, `makepkg` is configured to produce generic builds of software packages. Since `makepkg` will mostly be used to build packages for your own personal machine, compiler options can be tweaked to produce optimized builds for the machine they're getting built on.

For example, create a drop-in config file `/etc/makepkg.conf.d/cflags.conf` with the following contents:

```
CFLAGS="-march=native -O2 -pipe -fno-plt -fexceptions \  
-Wp,-D_FORTIFY_SOURCE=3 -Wformat -Werror=format-security \  
-fstack-clash-protection -fcf-protection \  
-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"
```

This will cause GCC to automatically detect and enable safe architecture-specific optimizations.

The same thing can be applied to the Rust compiler. There is already a drop-in config file at `/etc/makepkg.conf.d/rust.conf` that can be edited:

```
RUSTFLAGS="-C opt-level=2 -C target-cpu=native"
```

The `opt-level` parameter can be set to different values ranging in different levels of optimizations that will have an impact on build time. See the [Rust docs](#) for details.

Additionally, the `make` build system can also be optimized with the `MAKEFLAGS` variable. One such optimization is to increase the number of jobs that can run simultaneously.

Create a drop-in config file `/etc/makepkg.conf.d/make.conf` with the following contents:

```
MAKEFLAGS="--jobs=$(nproc)"
```

This will prompt `make` to utilize the maximum number of CPU cores to run build jobs.

ATTENTION: Some `PKGBUILD`s specifically override this with `-j1`, because of race conditions in certain versions or simply because it is not supported in the first place. If a package fails to build you should report this to the package maintainer.

Prevent build of `-debug` packages

By default, `makepkg` is configured to also generate debug symbol packages. This affects all AUR helpers. To turn this behavior off, modify the `OPTIONS` array by either removing the `debug` option or disabling it with a `!` in front of it:

```
OPTIONS=(strip docs !libtool !staticlibs emptydirs zipman purge !debug lto)
```

Using the mold linker

`mold` is a drop-in replacement for `ld`/`lld` linkers, which claims to be significantly faster.

Install `mold` from the repositories:

```
pacman -S mold
```

To use `mold`, append `-fuse-ld=mold` to `LDFLAGS`:

```
LDFLAGS="-Wl,-O1 -Wl,--sort-common -Wl,--as-needed -Wl,-z,relro -Wl,-z,now \
-Wl,-z,pack-relative-relocs -fuse-ld=mold"
```

This also needs to be passed to `RUSTFLAGS`:

```
RUSTFLAGS="-C opt-level=2 -C target-cpu=native -C link-arg=-fuse-ld=mold"
```

Compression options

By default, `makepkg` will compress built packages with `zstd`. This is controlled by the `PKGEXT` variable. The compression algorithm used is inferred from the archive extension. To speed up the packaging process, you might consider turning off the compression at the expense of increased storage usage in the package cache:

```
PKGEXT='.pkg.tar'
```

If you need to conserve space, consider keeping compression enabled, but increasing the number of utilized cores by telling `zstd` to count logical cores instead of physical ones with `--auto-threads=logical`:

```
COMPRESSZST=(zstd -c -T0 --auto-threads=logical -)
```

You can also increase the level of compression applied at the expense of longer packaging time, ranging from 1 (weakest) to 19 (strongest), default is 3:

```
COMPRESSZST=(zstd -c -T0 -19 --auto-threads=logical -)
```

Or use the LZ4 algorithm, which is optimized for speed:

```
PKGEXT='.pkg.tar.lz4'
```

Build entirely in RAM

You can pass `makepkg` a different directory for building packages. Since building causes a lot of rapid small file access, performance could be improved by moving this process to a `tmpfs` location that is held entirely in RAM. The variable `BUILDDIR` can be used to instruct `makepkg` to build packages in another location:

```
BUILDDIR=/tmp/makepkg
```

Since `/tmp` is such a `tmpfs` files in this directory are held in RAM. Building packages completely in RAM can therefore speed up data access and help preserve the durability of flash-based storage mediums like SSDs.

initramfs

The initramfs contains all the necessary programs and config files needed to bring up the machine, mount the root file system and hand off the rest of the boot process to the installed system. It can be further customized with additional modules, binaries, files and hooks for special use cases and hardware.

Usage

Automated image generation

Every kernel in Arch Linux comes with its own *.preset* file stored in `/etc/mkinitcpio.d/` with configuration presets for `mkinitcpio`. Pacman hooks build a new image after every kernel upgrade or installation of a new kernel.

Manual image generation

To manually generate a Linux kernel image issue the following command:

```
mkinitcpio -p linux
```

This will generate a new kernel image with the settings of the preset file

```
/etc/mkinitcpio.d/linux.preset
```

.

To generate kernel images with every preset available, pass the `-P` argument:

```
mkinitcpio -P
```

Configuration

To customize your initramfs, place drop-in configuration files into `/etc/mkinitcpio.conf.d/`. They will override the settings in the main configuration file at `/etc/mkinitcpio.conf`.

An overview of the settings you can customize:

Setting	Type	Description
---------	------	-------------

<code>MODULES</code>	Array	Kernel modules to be loaded before any boot hooks are run.
<code>BINARIES</code>	Array	Additional binaries you want included in the initramfs image.
<code>FILES</code>	Array	Additional files you want included in the initramfs image.
<code>HOOKS</code>	Array	Hooks are scripts that execute in the initial ramdisk.
<code>COMPRESSION</code>	String	Which tool to use for compressing the image.
<code>COMPRESSION_OPTIONS</code>	Array	Extra arguments to pass to the <code>COMPRESSION</code> tool.

WARNING: Do not use the `COMPRESSION_OPTIONS` setting, unless you know exactly what you are doing. Misuse can produce unbootable images!

MODULES

The `MODULES` array is used to specify modules to load before anything else is done.

Here you can specify additional kernel modules needed in early userspace, e.g. file system modules (`ext2`, `reiser4`, `btrfs`), keyboard drivers (`usbhid`, `hid_apple`, etc.), USB 3 hubs (`xhci_hcd`) or "out-of-tree" modules which are not part of the Linux kernel (mainly NVIDIA GPU drivers). It is also needed to add modules for hardware devices that are not always connected but you would like to be operational from the very start if they are connected during boot.

HINT: If you don't know the name of the driver of a device, `lshw` can tell you what hardware uses which driver, e.g.:

```
*-usb:2
    description: USB controller
    product: Tiger Lake-LP USB 3.2 Gen 2x1 xHCI Host Controller
    vendor: Intel Corporation
    physical id: 14
    bus info: pci@0000:00:14.0
    version: 20
    width: 64 bits
    clock: 33MHz
    capabilities: xhci bus_master cap_list
-> configuration: driver=xhci_hcd latency=0
```



```
resources: iomemory:600-5ff irq:163 memory:603f260000-603f26ffff
```

The second to last line starting with `configuration` shows the driver being used.

Example of a `MODULES` array that adds two modules to the generated image needed for keyboard input, if the keyboard is connected to a USB 3 hub, e.g. a docking station:

```
MODULES=(xhci_hcd usbhid)
```

CAUTION: Keep in mind that adding to the initramfs increases the size of the resulting image on disk. Unless you have created your boot partition (more specifically the EFI System partition at either `/efi`, `/boot` or `/boot/efi`) with generous space, you should limit yourself to modules strictly needed for your system. The `autodetect` hook tries to detect all currently loaded modules of the running system to determine the needed modules to include by default. Only include additional modules if something doesn't work as expected.

ATTENTION: If you use an NVIDIA graphics card, the following modules are **required** in the `MODULES` array for early KMS:

```
MODULES=(nvidia nvidia_modeset nvidia_uvm nvidia_drm)
```

BINARIES

The `BINARIES` array holds the name of extra executables needed to boot the system. It can also be used to replace binaries provided by `HOOKS`. The executable names are sourced from the `PATH` environment variable, associated libraries are added as well.

Example of a `BINARIES` array that adds the `kexec` binary:

```
BINARIES=(kexec)
```

This option usually only needs to be set for special use cases, e.g. when there's a binary you need included that is not already part of a member in the `HOOKS` array.

FILES

The `FILES` array holds the full path to arbitrary files for inclusion in the image.

Example of a module configuration file to be included in the image, containing the names of modules to auto-load and optional module parameters:

```
FILES=(/etc/modprobe.d/modprobe.conf)
```

This option usually only needs to be set for special use cases.

HOOKS

The `HOOKS` array is the most important setting in the file. Hooks are small scripts which describe what will be added to the image. Hooks are referred to by their name, and executed in the order they are listed in the `HOOKS` array.

HINT: For a full list of availble hooks run:

```
mkinitcpio -L
```

See the help text for a hook with:

```
mkinitcpio -H hook_name
```

The default `HOOKS` line in `/etc/mkinitcpio.conf` is as follows:

```
HOOKS=(base udev autodetect microcode modconf kms keyboard keymap consolefont block filesystems fsck)
```

This creates a basic image suitable for most single disk systems.

A quick overview of the hooks and their meaning:

Hook	Description
base	Sets up all initial directories and installs base utilities and libraries.
udev	Adds the udev device manager to scan and set up devices. Recommended for simple boot process.
autodetect	Trims hooks after that come after to only include modules that are needed for the current system. Keeps image slim.
microcode	Includes CPU microcode updates in the image.
modconf	Includes module configuration files from <code>/etc/modprobe.d/</code> and <code>/usr/lib/modprobe.d/</code> .

Hook	Description
<code>kms</code>	Adds modules to bring up graphics cards as early as possible in the boot process.
<code>keyboard</code>	Adds modules for keyboards. Required for keyboard input in early userspace.
<code>keymap</code>	Adds the specified keymap(s) from <code>/etc/vconsole.conf</code> .
<code>consolefont</code>	Adds the specified console font from <code>/etc/vconsole.conf</code> .
<code>block</code>	Adds block device modules needed to bring up different kinds of storage devices.
<code>filesystems</code>	Adds file system modules. Required unless file system modules are specified in <code>MODULES</code> .
<code>fsck</code>	Adds tools for checking file systems before they are mounted. Strongly recommended!

busybox

By default, `mkinitcpio` will generate a busybox-based initramfs. It starts an init script that scans the filesystem of the initramfs for scripts to execute and bring up the system and hand over the remaining boot process to systemd once the root file system is mounted. This is fine for most use-cases.

For special cases some additional hooks may be required for busybox to bring up the machine properly:

Hook	Description
<code>usr</code>	Needed for when you have <code>/usr</code> on a separate partition
<code>resume</code>	Needed for suspend-to-disk (hibernation) support
<code>btrfs</code>	Needed for btrfs file systems that span multiple drives, needs the <code>btrfs-progs</code> package installed
<code>net</code>	Needed for booting from a network drive, needs the <code>mkinitcpio-nfs-utils</code> package installed
<code>dmraid</code>	Needed for fakeRAID (BIOS RAID) root devices, needs the <code>dmraid</code> package installed
<code>mdadm_udev</code>	Needed for assembling RAID arrays via udev (software RAID), needs the <code>mdadm</code> package installed
<code>encrypt</code>	Needed for booting from an encrypted file system, needs the <code>cryptsetup</code> package installed
<code>lvm2</code>	Needed for booting a system that is on LVM, needs the <code>lvm2</code> package installed

One such special case is encryption, which would result in a `HOOKS` array that looks like this:



ATTENTION: The order in which hooks are placed in the array is important!

```
HOOKS=(base udev autodetect microcode modconf kms keyboard keymap consolefont block encrypt filesystems fsck)
```

ATTENTION: In some cases it might be necessary to place the `keyboard` hook before the `autodetect` hook to be able to enter the passphrase to unlock the encrypted file systems, e.g. when using different keyboards requiring a different module from the one in use at the time of building the initramfs.

systemd

If you wish, you can also make systemd bring the whole system up start to finish. In this case bootup will be handled by systemd unit files instead of scripts.

The benefit of this is faster boot times and some additional features not available to a busybox-based initramfs, e.g. unlocking LUKS encrypted file systems with a TPM or FIDO2 token and automatic detection and mounting of partitions with the appropriate GUID Partition Table (GPT) UUIDs (see: [Discoverable Partition Specification](#)).

To instruct `mkinitcpio` to build a systemd-based initramfs:

- replace the `udev` hook with the `systemd` hook
- replace the `keymap` and `consolefont` hooks with the `sd-vconsole` hook

The resulting `HOOKS` array should look something like this:

```
HOOKS=(base systemd autodetect microcode modconf kms keyboard sd-vconsole block filesystems fsck)
```

For special cases some additional hooks may be required for systemd to bring up the machine properly:

Hook	Description
<code>mdadm_udev</code>	Needed for assembling RAID arrays via udev (software RAID), needs the <code>mdadm</code> package installed
<code>sd-encrypt</code>	Needed for booting from an encrypted file system, needs the <code>cryptsetup</code> package installed
<code>lvm2</code>	Needed for booting a system that is on LVM, needs the <code>lvm2</code> package installed

One such special case is encryption, which would result in a `HOOKS` array that looks like this:

ATTENTION: The order in which hooks are placed in the array is important!

```
HOOKS=(base systemd autodetect microcode modconf kms keyboard sd-vconsole block sd-encrypt filesystems fsck)
```

ATTENTION: In some cases it might be necessary to place the `keyboard` hook before the `autodetect` hook to be able to enter the passphrase to unlock the encrypted file systems, e.g. when using different keyboards requiring a different module from the one in use at the time of building the initramfs.

COMPRESSION

The `COMPRESSION` option instructs `mkinitcpio` to compress the resulting images to save on space on the EFI System Partition or `/boot` partition. This can be especially important if you include a lot of modules and hooks and the size of the image grows.

Compressing the initramfs is a tradeoff between:

- time it takes to compress the image
- space saved
- time it takes the kernel to decompress the image during boot

Which one you choose is something you have to decide on the constraints you're working with (slow/fast CPU, available cores, RAM usage, disk space), but generally speaking the default `zstd` compression strikes a good balance.

Algorithm	Description
<code>cat</code>	Uncompressed
<code>zstd</code>	Best tradeoff between de-/compression time and image size (default)
<code>gzip</code>	Balanced between speed and size, acceptable performance
<code>bzip2</code>	Rarely used, decent compression, resource conservative
<code>lzma</code>	Very small size, slow to compress
<code>xz</code>	Smallest size at longer compression time, RAM intensive compression
<code>lzop</code>	Slightly better compression than lz4, still fast to decompress
<code>lz4</code>	Fast decompression, slow compression, "largest" compressed output

NOTE: See [this article](#) for a comprehensive comparison between compression algorithms.

COMPRESSION_OPTIONS

WARNING: Misuse of this option may lead to an **unbootable system** if the kernel is unable to unpack the resultant archive. **Do not set** this option unless you're *absolutely* sure that you have to!

The `COMPRESSION_OPTIONS` setting allows you to pass additional parameters for the compression tool. Available parameters depend on the algorithm chosen for the `COMPRESSION` option. Refer to the tool's manual for available options. If left empty `mkinitcpio` will make sure it always produces a working image.

zram

The zram kernel module provides a compressed block device in RAM. If you use it as swap device, the RAM can hold much more information but uses more CPU. Still, it is much quicker than swapping to a hard drive. If a system often falls back to swap, this could improve responsiveness. Using zram is also a good way to reduce disk read/write cycles due to swap on SSDs.

Install the `zram-generator` package and copy the example configuration:

```
pacman -S zram-generator  
cp /usr/share/doc/zram-generator/zram-generator.conf.example /etc/systemd/zram-generator.conf
```

Edit the copy of the example configuration to your liking. Comments explain what each setting does.

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` bootloader will be copied to the ESP after a reboot, completely automating the bootloader update and signing process!

Boot Loader

systemd-boot

`systemd` comes with `systemd-boot` already, so no additional packages need to be installed.

Install

ATTENTION: By default, `systemd-boot` will install itself to either of the well-known ESP locations, e.g. `/efi`, `/boot`, or (discouraged) `/boot/efi`. If your ESP is mounted somewhere else pass the location with the `--esp-path` parameter.

To install `systemd-boot` to your EFI System Partition and create a boot loader entry named "Linux Boot Manager" in your firmware:

```
bootctl install
```

This will copy `/usr/lib/systemd/boot/efi/systemd-bootx64.efi` to `$ESP/EFI/systemd/systemd-bootx64.efi` and `$ESP/EFI/BOOT/BOOTX64.EFI`.

NOTE: If a signed version of `systemd-bootx64.efi` exists as `systemd-bootx64.efi.signed` in the same directory, `bootctl` copies the signed file instead.

NOTE: `bootctl` may complain about your ESP's mount point and the random seed file as being "world accessible". This is to let you know your ESP's current file system permissions are too open. To solve this, change the `fmask` and `dmask` mount options for your ESP in `/etc/fstab` from `0022` to `0077`. Changes apply on next boot. See also: [mount\(8\) \\$ Mount options for fat](#)

Configure

`systemd-boot` has two kinds of configs:

- `$ESP/loader/loader.conf`: Configuration file for the boot loader itself
- `$ESP/loader/entries/*.conf`: Configuration files for individual boot entries

Boot loader config

NOTE: For a full list of options and their explanation refer to `loader.conf(5) § OPTIONS`

Setting	Type	Description
<code>default</code>	string	The pre-selected default boot entry. Can be pre-determined value, file name or glob pattern
<code>timeout</code>	number	Time in seconds until the default entry is automatically booted
<code>console-mode</code>	number/string	Display resolution mode (<code>0</code> , <code>1</code> , <code>2</code> , <code>auto</code> , <code>max</code> , <code>keep</code>)
<code>auto-entries</code>	boolean	Show/hide other boot entries found by scanning the boot partition
<code>auto-firmware</code>	boolean	Show/hide "Reboot into firmware" entry

An example loader configuration could look something like this:

ATTENTION: Only spaces are accepted as white-space characters for indentation, do not use tabs!

```
default      arch    # pre-selects entry from $ESP/loader/entries/arch.conf
timeout      3      # 3 seconds before the default entry is booted
auto-entries  1      # shows boot entries which were auto-detected
auto-firmware 1      # shows entry "Reboot into firmware"
console-mode max    # picks the highest-numbered mode available
```

Boot entry config

SEE ALSO: [The Boot Loader Specification](#) for a comprehensive overview of what `systemd-boot` implements.

Available parameters in boot entry config files:

Key	Value	Description
<code>title</code>	string	The name of the entry in the boot menu (optional)
<code>version</code>	string	Human readable version of the entry (optional)
<code>machine-id</code>	string	The unique machine ID of the computer (optional)

Key	Value	Description
<code>sort-key</code>	string	Used for sorting entries (optional)
<code>linux</code>	path	Location of the Linux kernel (relative to ESP)
<code>initrd</code>	path	Location of the Linux initrd image (relative to ESP)
<code>efi</code>	path	Location of an EFI executable, hidden on non-EFI systems
<code>options</code>	string	Kernel command line parameters
<code>devicetree</code>	path	Binary device tree to use when executing the kernel (optional)
<code>devicetree-overlay</code>	paths	List of device tree overlays. If multiple, separate by space, applied in order
<code>architecture</code>	string	Architecture the entry is intended for (<code>IA32</code> , <code>x64</code> , <code>ARM</code> , <code>AA64</code>)

Type 1 (text file based)

NOTE: As of `mkinitramfs v38`, the CPU microcode is embedded in the `initramfs` and it is no longer necessary to specify CPU microcode images on a separate `initrd` line before the actual `initramfs`.

Type 1 entries specify their parameters in `*.conf` files under `$ESP/loader/entries/`.

All paths in these configs are relative to the ESP, e.g. if the ESP is mounted at `/boot` a boot loader entry located at `$ESP/loader/entries/arch.conf` would look like this:

```
title Arch Linux
linux vmlinuz-linux
initrd /initramfs-linux.img
options rd.luks.name=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX=cryptroot root=/dev/mapper/cryptroot
rw
```

Type 2 (EFI executable)

When using a unified kernel image, any image ending with `*.efi` placed under `$ESP/EFI/Linux/` will be automatically picked up by `systemd-boot` along with the metadata embedded in that image (e.g. title, version, etc.)

If your UKIs are stored somewhere else, you will need a loader entry `*.conf` file with an `efi` key pointing `systemd-boot` to the location of the `*.efi` file on the ESP:

```
title Arch Linux
efi /EFI/Arch/linux.efi
```

EFISTUB

EFISTUB is a method of booting the kernel directly as an EFI executable by the firmware without the need to use a boot loader. This can be useful in cases where you want to reduce the attack surface a boot loader can introduce, or you intend to only ever boot one image. However, some UEFI firmware implementations can be flaky, so this isn't always practical.

Install

To be able to manipulate EFI boot variables install `efibootmgr`:

```
pacman -S efibootmgr
```

Configure

ATTENTION: `efibootmgr` cannot overwrite existing boot entries and will disregard the creation of a boot entry if one with the same label already exists. If you need to overwrite an existing entry you will need to delete it first. Call `efibootmgr` without any arguments to list all current boot entries:

```
efibootmgr
```

To delete an entry, note its 4-digit boot entry order and instruct `efibootmgr` to delete it:

```
efibootmgr -Bb XXXX
```

To create a new entry `efibootmgr` needs to know the disk and partition where the kernel image resides on the ESP.

In this example, the ESP is the first partition of the block device `/dev/nvme0n1`. Kernel parameters are part of the `-u` option. The partition that holds your root file system needs to be passed as a [persistent block device name](#).

NOTE: If you use LVM or LUKS, you can supply the device mapper name since that already is persistent.

You can get the persistent block device identifier of a file system with the `blkid` command, i.e. to get the UUID of the root file system:

```
# /dev/nvme0n1p1 is the ESP, hence /dev/nvme0n1p2 is the root fs
blkid -s UUID -o value /dev/nvme0n1p2
```

For ease of scriptability, save the values to environment variables:

```
export ROOT=$(blkid -s UUID -o value /dev/nvme0n1p2)
export CMDL="root=UUID=$ROOT rw add_efi_memmap initrd=\\initramfs-linux.img"
```

Then create the boot entry using `efibootmgr`:

```
efibootmgr -c -L "Arch Linux" -d /dev/nvme0n1 -p 1 -l /vmlinuz-linux -u $CMDL -v
```

Unified kernel image

When using a **unified kernel image** you can instead just point to the UKI without needing to specify any kernel parameters via the `-u` option (as these will be part of the UKI already):

ATTENTION: If Secure Boot is enabled and the command line parameters are embedded in the UKI, the embedded command line parameters will always take precedence, even if you pass additional parameters with the `-u` option.

```
efibootmgr -c -L "Arch Linux" -d /dev/nvme0n1 -p 1 -l "EFI\Linux\archlinux-linux.efi" -v
```