Installation

Laying the foundation

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

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
`--age n`	Only return mirrors that have synchronized in the last *n* hours.
`--country NAME`	Restrict mirrors to selected countries, e.g. `France,Germany` (check available with `--list-countries`)
`--fastest n`	Return the *n* fastest mirrors that meet the other criteria. Do not use without filters!
`--latest n`	Limit the list to the *n* most recently synchronized servers.
`--score n`	Limit the list to the *n* servers with the highest score.
`--number n`	Return at most *n* mirrors.
`--protocol PROTO`	Restrict protocol used by mirrors. Either `https`, `http`, `ftp` 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
`base`	Absolute essentials **(required)**
`linux`	Vanilla Linux kernel and modules, with a few patches applied **(required)**
`linux-hardened`	A security-focused Linux kernel applying a set of hardening patches to mitigate kernel and userspace exploits
`linux-lts`	Long-term support (LTS) Linux kernel and modules
`linux-zen`	Result of a collaborative effort of kernel hackers to provide the best Linux kernel possible for everyday systems

Package	Description
`linux-firmware`	Device firmware files, e.g. WiFi **(required)**
`intel-ucode`	Intel CPU microcode **(required, if on Intel)**
`amd-ucode`	AMD CPU microcode **(required, if on AMD)**
`btrfs-progs`	Userspace tools to manage btrfs filesystems
`dosfstools`	Userspace tools to manage FAT filesystems
`exfatprogs`	Userspace tools to manage exFAT filesystems
`f2fs-tools`	Userspace tools to manage F2FS filesystems
`e2fsprogs`	Userspace tools to manage ext2/3/4 filesystems
`jfsutils`	Userspace tools to manage JFS filesystems
`nilfs-utils`	Userspace tools to manage NILFS2 filesystems
`ntfs-3g`	Userspace tools to manage NTFS filesystems
`udftools`	Userspace tools to manage UDF filesystems
`xfsprogs`	Userspace tools to manage XFS filesystems
`lvm2`	Userspace tools for Logical Volume Management
`cryptsetup`	Userspace tools for encrypting storage devices (LUKS)
`networkmanager`	Comprehensive network management and configuration suite
`nano`	Console text editor
`man`	Read documentation (**man**uals)
`sudo`	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 desireable 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 -K /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 assigned labels of devices/partitions
genfstab -L /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

Use `timedatectl` to check which time zone your system is currently set to:

```
               Local time: Tue 2025-09-23 20:04:39 UTC
           Universal time: Tue 2025-09-23 20:04:39 UTC
                 RTC time: Tue 2025-09-23 20:04:39
                Time zone: UTC (UTC, +0000)
System clock synchronized: yes
              NTP service: active
          RTC in local TZ: no
```

If the time zone doesn't match with the one you live in (e.g. if it says UTC), use the `set-timezone` command to change it:

> **NOTE:** To list all available time zones, use the `list-timezones` command. Search for town names with `/` (search is case-sensitive). Alternatively, there's a website to help you pick the correct one by country.

```
timedatectl set-timezone Europe/Berlin
```

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.

Alternatively, uncomment the locales via `sed` in one command, e.g. German and US English:

```
sed -i -E 's/^# *(de_DE\.UTF-8|en_US\.UTF-8)/\1/' /etc/locale.gen
```

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
```

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, which will likely under-utilize the Wi-Fi bandwidth on your device.

Install `wireless-regdb` for utilities:

```
pacman -S wireless-regdb
```

To set your region temporarily, e.g. Germany:

```
iw reg set DE
```

To set it permanently, uncomment the line with your country in the file `/etc/conf.d/wireless-regdom` with `nano` or use a single `sed` command:

```
sed -i 's/^#WIRELESS_REGDOM="DE"/WIRELESS_REGDOM="DE"/' /etc/conf.d/wireless-regdom
```

The file will be picked up into the initramfs during rebuilds, to set the regulatory domain during boot. Remember to run `mkinitcpio -P` if you decide to change it after installation.

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 a configuration file:

```
pacman -S iwd
nano /etc/NetworkManager/conf.d/wifi_backend.conf
```

Set the following in the configuration file:

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

When NetworkManager starts, it will now use `iwd` for managing wireless connections.

IPv6 Privacy Extensions

By default, Arch enables IPv6, but with the actual public IP address exposed. IPv6 includes the MAC address of the network interface. IPv6 Privacy Extensions mangle the public IP address in a way that prevents the actual address from being known publicly.

Enabling IPv6 Privacy Extensions can be done in different ways:

1. via `sysctl` parameters, setting it at the lowest level
2. via NetworkManager, either globally or per connection profile

If not set via the global NetworkManager config or a connection profile, NetworkManager uses sysfs to determine if IPv6 Privacy Extensions should be enabled.

`sysctl`

To enable IPv6 Privacy Extensions via sysfs during boot, `sysctl` parameters in a config file can be used.

There are 3 parameters which control behavior:

> **NOTE:** The spelling for the parameter `temp_prefered_lft` is not a typo!

Name	Value	Description
`use_tempaddr`	2	0 = disabled, 1 = enable, prefer real IP, 2 = enable, prefer temporary IP
`temp_prefered_lft`	86400	Preferred life time of temporary IP in seconds (default = 1 day)
`temp_valid_lft`	604800	Maximum life time of temporary IP in seconds (default = 7 days)

These parameters can be applied to:

1. set the parameter on `all` connections
2. set the parameter on the `default` connection
3. set the parameter on a specific network interface (`nic`)

Create a config file such as `/etc/sysctl.d/40-ipv6.conf` and choose your parameter values for **one** of the three ways of setting up IPv6 Privacy extensions.

For **all** network interfaces:

```
# Enable IPv6 Privacy Extensions
net.ipv6.conf.all.use_tempaddr = 2
net.ipv6.conf.all.temp_prefered_lft = 86400
net.ipv6.conf.all.temp_valid_lft = 604800
```

For the **default** network interface:

```
# Enable IPv6 Privacy Extensions
net.ipv6.conf.default.use_tempaddr = 2
net.ipv6.conf.default.temp_prefered_lft = 86400
net.ipv6.conf.default.temp_valid_lft = 604800
```

For a specific network interface, e.g. the first Wi-Fi adapter called **wlan0**:

```
# Enable IPv6 Privacy Extensions
net.ipv6.conf.wlan0.use_tempaddr = 2
net.ipv6.conf.wlan0.temp_prefered_lft = 86400
net.ipv6.conf.wlan0.temp_valid_lft = 604800
```

NetworkManager

> **NOTE:** If you set up IPv6 Privacy Extensions via `sysctl` config, NetworkManager will use it automatically.

NetworkManager can be set up to enable IPv6 Privacy Extensions. This can either be done globally or per connection profile.

To enable it **globally** create the config file `/etc/NetworkManager/conf.d/ip6-privacy.conf` with the following contents:

```
[connection]
ipv6.ip6-privacy=2
```

This will apply the setting across all current and future connections.

To enable it only **for specific connections**, open the connection profile, e.g. `/etc/NetworkManager/system-connections/<connection name>.nmconnection`, look for the `[ipv6]` section in the file and add the following:

```
...
[ipv6]
...
ip6-privacy=2
...
```

Connection profile files are named the same as their corresponding network, so `Wired Connection 1.nmconnection` or the name of any Wi-Fi network you ever connected to. When you connect to a

new network, you will have to apply these settings again for the new connection.

`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 explenation 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 authentification 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
`allow-downgrade`	Validate DNSSEC only if the upstream DNS server supports it
`true`	Always validate DNSSEC, breaking DNS resolution if the server does not support it
`false`	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
`opportunistic`	Attempt DNS over TLS when possible and fall back to unencrypted DNS if the server does not support it

Setting	Description
`true`	Always use DNS over TLS, breaking resolution if the server does not support it
`false`	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 synthesizinig 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]
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
`resolve`	Only enables resolution support, but responding is disabled
`true`	Enables full mDNS responder and resolver support
`false`	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 packages pull in the `avahi` package as a dependency, e.g. `cups`. It enables desktop file managers 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.

Exposing services via mDNS

Avahi exposes services on the network via `.service` files. These are bascially just XML files describing the type and port on which the service is reachable.

The `avahi` package provides example files in `/usr/share/doc/avahi/`, e.g. for exposing SSH and SFTP services on the local network:

> **NOTE:** See avahi.service(5) for more information on how to write your own service files.

> **WARNING:** Exposing sensitive services on untrusted networks (i.e. public Wi-Fi) is a security risk. Your machine will broadcast its services to every computer on the same network, making it easier to identify how to attack it. Use of a filewall with zoning capabilities per connection (such as `firewalld`, see below) is *strongly recommended* if you want to use this!

```
<?xml version="1.0" standalone='no'?><!--*-nxml-*-->
<!DOCTYPE service-group SYSTEM "avahi-service.dtd">

<service-group>

  <name replace-wildcards="yes">%h</name>

  <service>
    <type>_ssh._tcp</type>
    <port>22</port>
  </service>

</service-group>
```

Simply copy the example files from `/usr/share/doc/avahi/` to `/etc/avahi/services/`:

```
cp /usr/share/doc/avahi/*.service /etc/avahi/services/
```

After that, restart Avahi (or reboot after finishing installation).

Firewall

`firewalld` provides a dynamically managed firewall with support for network "zones" to define the trust level of network connections or interfaces. It acts as a powerful frontend to `nftables` (the modern Linux kernel packet classification framework) and integrates seamlessly with NetworkManager.

Install `firewalld` from the Arch repos:

```
pacman -S firewalld
```

There is also a graphical configuration utility as well as an applet available as optional dependencies:

```
pacman -S --asdeps firewall-config firewall-applet
```

Enable `firewalld` to launch when the system boots:

```
systemctl enable firewalld
```

Configuration

To configure `firewalld` use the `firewall-cmd` command.

`firewalld` comes with pre-defined service configurations for many network services, for example for allowing incoming mDNS requests to pass through the firewall. These services are enabled per "zone". Different zones can be applied to different network interfaces and connection profiles. This allows for quickly switching out the set of allowed services for a connection based on its zone, e.g. "home" or "work" for trusted networks, "public" for untrusted networks in airports, cafés, etc.

To query the current default zone applied to new connections:

> **ATTENTION:** If you're still setting up Arch, use `firewall-offline-cmd` instead. The command syntax is mostly identical.

```
firewall-cmd --get-default-zone
```

To list the zones currently applied to active network connections:

```
firewall-cmd --get-active-zones
```

To allow a service on a specific zone:

```
firewall-cmd --zone=home --add-service=mdns --permanent
```

You can also allow a specific port and protocol combination:

```
firewall-cmd --zone=work --add-port=8080/tcp --permanent
```

However, it's much more elegant to create a service profile to enable/disable a range of ports and protocols (using OpenTTD as an example, which is not included):

```
firewall-cmd --new-service=openttd --permanent
firewall-cmd --service=openttd --add-port={3979/tcp,3979/udp,3978/udp} --permanent
```

This creates a new service with the ID `openttd` and adds the set of specified port/protocol combinations. Changes are written to disk at `/etc/firewalld/` but don't take effect immediately. A reload is necessary to load changes into memory:

```
firewall-cmd --reload
```

Now you can query info about the newly created service:

```
firewall-cmd --info-service=openttd
```

Which should get you:

```
openttd
  ports: 3979/tcp 3979/udp 3978/udp
  protocols:
  source-ports:
  modules:
  destination:
  includes:
  helpers:
```

The newly created service can then be enabled for a zone like any other, as shown in the example above.

Root Password

Set the password for the `root` user:

```
passwd
```

This password schould 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
`zsh-autosuggestions`	Suggests commands as you type based on history and completions
`zsh-completions`	Additional completion definitions for `zsh`
`zsh-history-substring-search`	Type any part of any command from history and cycle through matches
`zsh-syntax-highlighting`	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 is needs and all the usual metadata associated with every other Arch Linux package.

Arch Wiki has a list of AUR helpers with comparison tables

> **WARNING:** The AUR is entirely user-maintained with no formal security audit by the Arch Linux maintainers. If you install ***anything*** from the AUR, thoroughly inspect the `PKGBUILD` files and verify they contain no malicious instructions!

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'rere 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. `paru`:

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

`makepkg -si` will prompt you to install any missing dependencies for your chosen AUR helper 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 with a drop-in config file `/etc/makepkg.conf.d/options.conf` and either remove the `debug` option entirely or disable 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` in `/etc/makepkg.conf.d/cflags.conf`:

```
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.

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.

Example config

```
[zram0]
# Disable limit beyond which zram-generator will skip device creation
host-memory-limit = none

# Size of the zram device, half of total RAM
zram-size = ram / 2

# Compress memory pages with zstd right away
compression-algorithm = zstd

# Enable discard of freed blocks on zram device to keep it as small as possible
options = discard
```

Boot Loader

`systemd-boot`

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

Install

> **NOTE:** For Secure Boot setups, please see the Secure Boot page of this chapter on how to generate keys with `systemd-ukify` and enroll them into the firmware with `systemd-boot`.

> **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 localtion with the `--esp-path` parameter. `$ESP` refers to this location. Adjust paths accordingly!

> **WARNING:** `bootctl` will not operate on UEFI variables which store boot entries when running in regular `arch-chroot`, which could leave your machine unbootable. Enter a chroot environement with `arch-chroot -S` instead.

Install `systemd-boot` by simply invoking `bootctl` with the `install` command:

```
bootctl install
```

This will do the following:

- Create the directory `$ESP/EFI/Linux`
- Copy `/usr/lib/systemd/boot/efi/systemd-bootx64.efi` to `$ESP/EFI/systemd/systemd-bootx64.efi`
- Copy `/usr/lib/systemd/boot/efi/systemd-bootx64.efi` to `$ESP/EFI/BOOT/BOOTX64.EFI`
- Create a 32 byte random seed file at `$ESP/loader/random-seed`
- Create an EFI boot entry named *Linux Boot Manager* at the top of firmware boot entries

> **NOTE:** If a signed version of `systemd-bootx64.efi` exists as `systemd-bootx64.efi.signed` in the source directory (i.e. for Secure Boot), the signed file is copied 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 lenient. 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`. If you plan on using `systemd`'s GPT auto-mounting feature, it will set the appropriate file system permissions for you.

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`

The most important options for the boot loader are as follows:

Setting	Type	Description
`default`	string	The pre-selected default boot entry. Can be pre-determined value, file name or glob pattern
`timeout`	number	Time in seconds until the default entry is automatically booted
`console-mode`	number/string	Display resolution mode (`0`, `1`, `2`, `auto`, `max`, `keep`)
`auto-entries`	number/boolean	Show/hide other boot entries found by scanning the boot partition
`auto-firmware`	number/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
`title`	string	The name of the entry in the boot menu (optional)
`version`	string	Human readable version of the entry (optional)
`machine-id`	string	The unique machine ID of the computer (optional)
`sort-key`	string	Used for sorting entries (optional)
`linux`	path	Location of the Linux kernel (relative to ESP)
`initrd`	path	Location of the Linux initrd image (relative to ESP)
`efi`	path	Location of an EFI executable, hidden on non-EFI systems
`options`	string	Kernel command line parameters
`devicetree`	path	Binary device tree to use when executing the kernel (optional)
`devicetree-overlay`	paths	List of device tree overlays. If multiple, separate by space, applied in order
`architecture`	string	Architecture the entry is intended for (`IA32` , `x64` , `ARM` , `AA64`)

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 for 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 skip this step and 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. For example, if `/dev/nvme0n1p2` is the root file system:

```
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
```

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
`MODULES`	Array	Kernel modules to be loaded before any boot hooks are run.
`BINARIES`	Array	Additional binaries you want included in the initramfs image.
`FILES`	Array	Additional files you want included in the initramfs image.
`HOOKS`	Array	Hooks are scripts that execute in the initial ramdisk.
`COMPRESSION`	String	Which tool to use for compressing the image.
`COMPRESSION_OPTIONS`	Array	Extra arguments to pass to the `COMPRESSION` 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
    HERE ->  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` evironment 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, containting 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
> ```
>
> Alternatively, refer to Arch Wiki for a complete rundown of all the different hooks and their recommended order.

By default, systemd will 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 like 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).

The default `HOOKS` array should be enough to bring up most systems. However, if you have special use cases, additional hooks will be needed:

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

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

> **ATTENTION:** The order of the hooks in the array is important! Any hook placed *after* `autodetect` is stripped down to match only the hardware present during initramfs generation. While this minimizes the image size, it removes the driver redundancy needed for hardware changes. For example, if you build the image using a built-in laptop keyboard, `autodetect` might omit drivers for external USB keyboards, leaving you unable to type a decryption password or boot prompt if you switch inputs later. To ensure device compatibility across changing environments, place critical input and hardware hooks *before* the `autodetect` hook.

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

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
`cat`	Uncompressed
`zstd`	Best tradeoff between de-/compression time and image size (default)
`gzip`	Balanced between speed and size, acceptable performance
`bzip2`	Rarely used, decent compression, resource conservative
`lzma`	Very small size, slow to compress
`xz`	Smallest size at longer compression time, RAM intensive compression

Algorithm	Description
`lzop`	Slightly better compression than lz4, still fast to decompress
`lz4`	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 resulting 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.

Additionally, `MODULES_DECOMPRESS` instructs `mkinitcpio` to decompress kernel modules prior to inclusion in the initramfs. This can further increase compression efficiency and bring down the initramfs size further. When this option is not set, compressed kernel modules are included as-is.

For example, to use the maximum zstd compression level, using all available CPU cores and show verbose output during compression:

```
COMPRESSION="zstd"
COMPRESSION_OPTIONS=(-T0 -19 --long --auto-threads=logical -v)
MODULES_DECOMPRESS="yes"
```

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, thus making a boot manager potentially redundant.

The main benefit of UKIs is streamlining the signing process for Secure Boot, as there is only a single file to sign. By keeping this executable on a separately mounted EFI System Partition, the standard `/boot` directory can safely reside inside an encrypted LUKS partition, ensuring all

underlying system files are fully encrypted while the boot sequence itself is protected against tampering.

Version 31 of `mkinitcpio` introduced support for building UKIs out of the box. Starting with v39, `systemd-ukify` is the recommended tool by which to generate UKIs. As `systemd-ukify` is *not* part of the `systemd` package, you'll have to install it manually:

```
pacman -S --asdeps 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 from `/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.

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"
#ALL_kerneldest="/boot/vmlinuz-linux"

#PRESETS=('default')
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,plymouth --cmdline /etc/kernel/cmdline_fallback"
```

This `*.preset` file instructs `mkinitcpio` to generate a UKI and enables the fallback initramfs. It skips the `autodetect` and `plymouth` hooks for the fallback initramfs and passes a different set of kernel command line parameters, e.g. displaying boot logs instead of showing a splash screen.

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 `mkinitcpio` does not find command line parameters in either of the above locations, it will fall back to reading the command line of 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 parameter in one of the above locations!**

Create the directory for command line parameter drop-in files and start with specifying parameters for the root file system:

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

Continue by specifying the root file system via persistent block device naming and mounting it writable:

```
root=UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX rw
```

You can add as many `*.conf` files as you need to logically split up kernel parameters. All of the parameters from all files will be included in the UKI.

GPT auto mounting

> **ATTENTION:** This requires that the correct GPT partition types were set during partitioning and that the file systems to be auto mounted are located on the same disk as the EFI system partition. If other important file systems are located on other disks, i.e. the home directory is on another physical storage device, they must still be specified via `/etc/crypttab` and

`/etc/fstab`.

IMPORTANT: Be aware of the specifics of your chosen root file system. For example, when using btrfs, you will still need to specify the subvolume and any other file system options as kernel command line parameters (for example `rootflags=noatime,compress=zstd,subvol=@`), as automatic discovery and mounting will use the default options for mounting file systems.

Since the default initramfs type is systemd-based, it's possible to rely on `systemd-gpt-auto-generator(8)` for automatic discovery and mounting of file systems during boot. It specifically looks for a partition of type "Linux root" (specifically, the pre-determined GPT partition GUID). If it finds an encrypted LUKS container, it will ask for the passphrase for it and subsequently mount the file system within. This eliminates the need for `root` and `rd.luks` kernel command line parameters.

Once autommatically discovered file systems have been mounted, `/etc/crypttab` and `/etc/fstab` are processed and the boot process continues like normal.

Manually

In cases where GPT auto mounting is not possible or undesired, the manual way of specifying encrypted devices remains available.

In a systemd-based initramfs, `rd.luks.name` is used to specify the encrypted partition by its UUID and a mapper name by which the decrypted file system is made available, resulting in a kernel command line that looks like this:

NOTE: By default, dm-crypt does not allow TRIM for SSDs for security reasons (information leak). To override this behavior, either specify `rd.luks.options=discard` as an additional kernel command line parameter or add the `discard` option in `/etc/crypttab` in the options field.

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

The UUID of the encrypted file system can easily be determined using `cryptsetup luksUUID`:

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

```
cryptsetup luksUUID /dev/disk/by-label/cryptroot
```

If you prefer a config file approach, or want/need to mount multiple encrypted file systems during boot, add `x-initrd.attach` to the options column in your installed system's `/etc/crypttab` file. This will indicate to the `sd-encrypt` hook that the device is required during early boot and add it to

`/etc/crypttab` in the initramfs. Using persistent block device naming, the file could look like this (see `crypttab(5)` for details on the syntax):

> **NOTE:** Instead of the UUID, you can also pass the assigned `LABEL` of the LUKS container, e.g. `LABEL=cryptroot`.

```
# <name>    <device>                                    <passphrase>    <options>
root        UUID=XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX                  x-initrd.attach
```

This allows for omitting any `rd.luks` parameters entirely:

> **ATTENTION:** Be aware of the specifics of your chosen root file system. For example, when using btrfs, you will still need to specify the subvolume and any other file system options as kernel command line parameters (for example `rootflags=noatime,compress=zstd,subvol=@`), as automatic discovery and mounting will use the default options for mounting file systems.

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

Secure Boot

Secure Boot

Secure Boot is a UEFI security feature designed to protect the pre-boot environment from malware like bootkits and rootkits, or to strictly enforce which operating systems are permitted to load. By validating bootloaders, kernels, and drivers against a user-controlled database of authorized cryptographic signatures, it blocks both unauthorized code and unapproved operating systems from executing.

> **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

Firmware

To determine the current state of Secure Boot, execute:

```
bootctl status
```

This prints information about the current boot environment:

```
System:
      Firmware: UEFI 2.70 (EDK II 1.00)
 Firmware Arch: x64
   Secure Boot: disabled (user) <-- SEE THIS LINE
  TPM2 Support: yes
  Measured UKI: no
   Measured OS: no
  Boot into FW: supported
```

```
  Platform Lang: n/a


...
```

To proceed, you must set your firmware's Secure Boot mode to **Setup Mode** (this is usually done by clearing/wiping the default factory keys inside your UEFI settings). Refer to your motherboard's manual for specific instructions.

> **WARNING:** Purging default Secure Boot keys to enroll custom ones can **soft-brick your machine!** Hardware Option ROMs (OpROMs)—such as your GPU's VBIOS—are typically signed using the **Microsoft 3rd Party UEFI CA** certificate. If this specific certificate is missing from the signature database (`db`) when Secure Boot is active, the firmware will block the GPU's OpROM from executing. This results in a black screen during POST, completely disabling display output and locking you out of the UEFI settings needed to revert the change!

Unified Kernel Image

The most practical way to sign images is a Unified Kernel Image, which combines the kernel, initramfs, and kernel command line into a single EFI executable.

`mkinitcpio` recommends using `ukify` for UKI generation. Install it as an optional dependency:

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

If `mkinitcpio` detects `ukify` on the system, it will use it automatically. Refer to your local UKI configuration on how to set up UKIs with `mkinitcpio`.

Refer to the Unified Kernel Image section of initramfs on how to set up UKIs with `mkinitcpio`.

Generating keys

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

Secure Boot implementations use these keys:

Key Type	Description

Platform Key (PK)	Establishes platform ownership and toggles Setup Mode. The private key signs updates to the `KEK` database.
Key Exchange Key (KEK)	Database of certificates authorized to sign updates to `db` and `dbx`.
Signature Database (db)	Database containing certificates or SHA-256 hashes of allowed EFI binaries, kernels, and Option ROMs.
Forbidden Signatures Database (dbx)	Database containing revoked certificates or hashes of banned binaries. Checked first; overrides matches in `db`.

`systemd-ukify` has built-in support for generating Secure Boot keys and will use `systemd-sbsign` to sign the generated UKIs.

To generate the keys, supply `ukify` with a configuration file. Create `/etc/kernel/uki.conf` with the following contents:

> **NOTE:** The `[PCRSignature]` sections are optional. They allow a TPM to unlock a LUKS volume with signed PCR Policies during boot. If you don't intend to use TPM unlocking, you can leave them out.

```
[UKI]
SecureBootSigningTool=systemd-sbsign
SignKernel=true
SecureBootPrivateKey=/etc/kernel/secure-boot-private-key.pem
SecureBootCertificate=/etc/kernel/secure-boot-certificate.pem

[PCRSignature:all]
PCRPrivateKey=/etc/systemd/tpm2-pcr-private-key.pem
PCRPublicKey=/etc/systemd/tpm2-pcr-public-key.pem

[PCRSignature:initrd]
Phases=enter-initrd
PCRPrivateKey=/etc/systemd/tpm2-pcr-private-key-initrd.pem
PCRPublicKey=/etc/systemd/tpm2-pcr-public-key-initrd.pem
```

Instruct `ukify` to generate the keys listed in the config file:

```
ukify genkey --config=/etc/kernel/uki.conf
```

When `mkinitcpio` rebuilds the initramfs during system upgrades it calls `ukify` with `/etc/kernel/uki.conf` and will automatically sign the resulting UKIs with the keys listed in that config.

Signing `systemd-boot`

The bootloader itself must be signed, or the firmware will refuse to execute it.

Create a pacman hook at `/etc/pacman.d/hooks/90-systemd-boot-sign.hook` to automatically sign `systemd-boot` during installation and upgrades:

```
[Trigger]
Operation = Install
Operation = Upgrade
Type = Path
Target = usr/lib/systemd/boot/efi/systemd-boot*.efi

[Action]
Description = Signing systemd-boot EFI binary for Secure Boot
When = PostTransaction
Exec = /bin/sh -c 'while read -r f; do /usr/lib/systemd/systemd-sbsign sign --private-key
/etc/kernel/secure-boot-private-key.pem --certificate /etc/kernel/secure-boot-certificate.pem
--output "${f}.signed" "$f"; done;'
Depends = sh
NeedsTargets
```

This takes all `.efi` binaries in the target path and creates signed `.efi.signed` copies of them.

Reinstall `systemd` to invoke the bootloader signing hook and build a signed UKI:

```
pacman -S --asdeps systemd
```

To keep the bootloader up-to-date, enable `systemd-boot-update.service`:

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

It invokes `bootctl update` on every boot, checking if a newer version is available for installation, and copies the updated, signed version of the bootloader to the ESP.

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 supports two levels:

- `integrity`: Disables kernel features that allow userland to modify the running kernel (e.g., kexec, certain eBPF modifications, and raw port access).
- `confidentiality`: Additionally blocks userland from extracting confidential information from the kernel space.

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

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

Enrolling keys in firmware

> **WARNING:** If your system loads OpROMs (e.g. GPU firmware) you need to make sure to either include Microsoft's keys or the OpROM's digest value from the TPM Event Log into your firmware signature database. Check for OpROMs with `find /sys/devices/ -name rom`. If there is output, your system has OpROMs and you need to take extra steps to not soft-brick 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.

`systemd-boot` is capable of enrolling Secure Boot keys. To instruct it to do so automatically on next boot:

```
bootctl install \
  --secure-boot-auto-enroll yes \
  --certificate /etc/kernel/secure-boot-certificate.pem \
  --private-key /etc/kernel/secure-boot-private-key.pem
```

This creates the necessary signature lists inside `$ESP/loader/keys/auto/` and triggers native auto-enrollment on reboot.