

NixOS Android Builder

Contents

Design Principles	2
Limitations and Further Work	2
Used Technologies	2
Major Components	3
Disk Image	3
Build Process	3
Disk Layout	3
Ephemeral State Partition	4
Secure Boot Support	4
Custom FHS Environment	4
Android Build Environment	4
YubiKey Authentication	5
Unattended Mode	5
Fatal Error Handling	6
Debug Mode	6
Sequence Chart	7
Build-time	7
Description	7
Run-time	9
Description	9
Glossary	11

Design Principles

With the goal of enabling offline, SLSA-compliant builds for custom distributions of Android AOSP, we set out to create a minimal Linux system with the following properties:

- **Portable** – Runs on arbitrary `x86_64` hardware with UEFI boot, that provides sufficient disk (≥ 250 GB) and memory (≥ 64 GB) to build Android.
- **Offline** – Requires no network connectivity other than to internal source-code and artifact repositories.
- **Ephemeral** – Each boot of the builder should result in a pristine environment; no trace of build inputs or artifacts should remain after a build.
- **Declarative** – All aspects of the build system are described in Nix expressions, ensuring identical behavior regardless of the build environment or the time of build.
- **Trusted** – All deployed artifacts, such as disk images, are cryptographically signed for tamper prevention and provenance.

We created a modular proof-of-concept based on NixOS that fulfills most of these properties, with the remaining limitations and future plans detailed below. Usage instructions can be found in [./user-guide.pdf](#).

Limitations and Further Work

- **aarch64 support** could be added if needed. Only `x86_64` with `UEFI` is implemented at the moment.
- **artifact uploads**: build artifacts are currently not automatically uploaded anywhere, but stay on the build machine. Integration of a Trusted Platform Module (TPM) could be useful here, to ease authentication to private repositories as well as destinations for artifact upload.
- **measured boot**: while we use Secure Boot with a platform custom key, we do not measure involved components via a TPM yet. Doing so would improve existing Secure Boot measures as well as help with implementing attestation capabilities later on.
- **higher-level configuration**: Adapting the build environment to the needs of custom AOSP distributions might need extra work. Depending on the nature of those customizations, a good understanding of `nix` might be needed. We will ease those as far as possible, as we learn more about users customization needs.

Used Technologies

- **NixOS** - the Linux distribution chosen for its declarative module system and flexible boot process.
- **nixpkgs** - the software repository that enables reproducible builds of up-to-date open-source packages.
- **qemu** - used to run virtual machines during interactive, as well as automated testing. Both help to decrease testing & verification cycles during development & customization.
- **systemd** - orchestrates both upstream and custom components while managing credentials and persistent state.
- **systemd-repart** - prepares signable read-only disk images for the builder and resizes and re-encrypts the state partition at each boot.
- **Linux Unified Key Setup (LUKS)** - encrypts the state partition with an ephemeral key on each boot.
- Various **build requirements** for Android, such as Python 3 and OpenJDK. The complete list is in the `packages` section of `android-build-env.nix`.

A complete **Software Bill of Materials (SBOM)** for the builder's NixOS closure can be generated from the repository root by running, e.g.:

```
nix run github:tiiuae/sbomnix#sbomnix -- .#nixosConfigurations.nixos.toplevel
```

Major Components

The **NixOS Android Builder** is a collection of Nix expressions (a “nix flake”) and helper scripts that produce a reproducible¹, ready-to-flash Linux system capable of compiling Android Open Source Project (AOSP) code. The flake pins `nixpkgs` to a specific commit, ensuring that the same versions of compilers, libraries, and build tools are used on every build. Inside the flake, a NixOS module describes the system layout, the `android-build-env` package, and the custom `fhsenv` derivation that provides conventional Linux file system hierarchy. This approach guarantees that the same inputs always generate the same output, making the build process deterministic and auditable.

Users with `nix` installed can clone this repository, download all dependencies and build a signed disk image, ready to flash & boot on the build machine, in a few simple steps outlined in [README.md](#).

The resulting disk image boots on generic `x86_64` hardware with UEFI as well as Secure Boot, and provides an isolated build environment. It contains scripts for secure boot enrollment, a verified filesystem, and an ephemeral, encrypted state partition that holds build artifacts that cannot fit into memory.

Disk Image

A ready-made disk image to run NixOS Android Builder on a target host can be built from any existing `x86_64-linux` system with `nix` installed. Under the hood, the image itself is built by `systemd-repart`, using NixOS module definitions from `nixpkgs` as well as custom enhancements shipped in this repository.

Build Process

`systemd-repart` is called twice during build-time:

1. While building `system.build.intermediateImage`: A first image is built, it contains the `store` partition, populated with our NixOS closure as well as minimal `var-lib-build` partition. `boot` and `store-verity` remain empty during this step.
2. While building `system.build.finalImage`: Take the populated `store` partition from the first step, derive `dm-verity` hashes from them and write them into `store-verity`. The resulting `usrhash` is added to a newly built UKI, which is then copied to `boot`, to a path where the firmware finds it (`/EFI/BOOT/BOOTX86.EFI`).
3. The image then needs to be signed with a script outside a `nix` build process (to avoid leaking keys into the world-readable `/nix/store`. No `systemd-repart` is involved in this step. Instead we use `mtools` to read the UKI from the image, sign it and - together with Secure Boot update bundles, write it back to `boot` inside the image.
4. Finally, `systemd-repart` is called once more during run-time, in early boot at the start of `initrd`: The minimal `var-lib-build` partition, created in the first step above, is resized and encrypted with a new random key on each boot. That key is generated just before `systemd-repart` in our custom `generate-disk-key.service`.

Disk Layout

Partition	Label	Format	Mountpoint
<code>00-esp</code>	<code>boot</code>	<code>vfat</code>	<code>/boot</code>
<code>10-store-verity</code>	<code>store-verity</code>	<code>dm-verity hash</code>	<code>n/a</code>
<code>20-store</code>	<code>store</code>	<code>erofs</code>	<code>/usr</code>

¹Reproducible in functionality. The final disk images are not yet expected to be *fully* bit-by-bit reproducible. That could be done, but would require a long-tail of removing additional sources of indeterminism, such as date & time of build. See [reproducible.nixos.org](#)

Partition	Label	Format	Mountpoint
30-var-lib-build	<code>var-lib-build</code>	<code>ext4</code>	<code>/var/lib/build</code>

- **boot** – Holds the signed Unified Kernel Image (UKI) as an EFI application, as well as Secure Boot update bundles for enrollment. The partition itself is unsigned and mounted read-only during boot.
- **store-verity** – Stores the `dm-verity` hash for the `/usr` partition. The hash is passed as `usrhash` in the kernel command line, which is signed as part of the UKI.
- **store** – Contains the read-only Nix store, bind-mounted into `/nix/store` in the running system. The integrity of `/usr` is verified at runtime using `dm-verity`.
- **var-lib-build** – A minimal, ephemeral state partition. See next section below.

Notably, the root filesystem (`/`) is, along with an optional writable overlay of the Nix store, kept entirely in RAM (`tmpfs`) and therefore not present in the image. There's also no boot loader, because the UKI acts as an EFI application and is directly loaded by the host's firmware.

Ephemeral State Partition

The `/var/lib/build` partition is deliberately designed to be temporary and encrypted. Each time the system boots, a fresh key is generated and the partition is resized to match the current disk size. This ensures that sensitive build artifacts never persist beyond a single session, reducing the risk of leaking proprietary information or to introduce impurities between different builds.

Secure Boot Support

Secure Boot is enabled by generating a set of keys that are stored unencrypted in a local `keys/` directory within the repository. Users must protect these keys and back them up. When a new image is signed, Secure Boot update bundles (`*.auth` files) are created for each target machine. These bundles are stored unsigned and unencrypted on the `/boot` partition. On boot, we check whether we are in Secure Boot setup mode and, if so, enroll our keys. If Secure Boot is disabled, we display an error and fail early during boot.

Custom FHS Environment

The builder image includes a custom builder for File Hierarchy Standard (FHS) environments.

It consists of a derivation that runs a python script, `fhsenv.py` to bundle together all libraries and binaries of declared packages (`nixosAndroidBuilder.fhsEnv.packages`), arranging them in one big FHS layout with `/bin` & `/lib` directories in the derivations output.

A mechanism to pin specific instances of packages which might be included multiple times inside the transitive dependency tree. See `nixosAndroidBuilder.fhsEnv.pins`.

The `fhsenv.nix` NixOS Module bind-mounts `/lib` and `bin` from the derivations output during runtime, while also setting default pins / packages, `$PATH` and adding a custom build of `glibc` for its dynamic linker, and a FHS-compatible build of `bash`.

That dynamic linker is configured to `/lib` instead of the standard Nix store paths. This setup mimics a conventional Linux environment, allowing the Android build system to function without modification.

Alternative approaches, such as `pkgs.buildFHSEnv`, `nix-ld` or `envfs`, were evaluated but found insufficient because they rely on individual symlinks that break when sandboxed bind-mounts are applied to `/bin` and `/lib` only, without having `/nix/store` in the sandbox.

Android Build Environment

The `android-build-env.nix` NixOS module uses the `fhsenv.nix` module described in the section above, to add all tools required for an AOSP build. By using this module, developers can compile

Android in a clean, reproducible environment that mimics a standard Linux installation.

It also adds 4 scripts, added for convenience:

- `fetch-android` checks out the configured `repo` repository & branch, upstream AOSP's `android-latest-release` by default. If multiple branches are configured via `nixosAndroidBuilder.build.branches`, `fetch-android` will use the branch selected by the `select-branch` script (see below).
- `build-android` loads the shell setup, sets the configured `lunch` target and builds a given `target`.
- `android-sbom` is a thin wrapper around `build-android` to run upstream's Software Bill Of Materials facilities.
- `android-measure-source` hashes all files across all git repositories in the checkout to produce a source measurement in `out/source_measurement.txt`.

Please refer to the options reference in [user-guide.pdf](#).

YubiKey Authentication

The `yubikey-auth.nix` module enforces hardware-token-based authentication using YubiKeys via `pam_u2f`. Two groups of keys can be configured:

- `nixosAndroidBuilder.yubikeys.groupA` – The first set of U2F public keys.
- `nixosAndroidBuilder.yubikeys.groupB` – An optional second set. If configured, both groups are required for login, enforcing dual-approval (e.g. two different people must each touch their YubiKey).

Public keys are generated with `pamu2fcfg` and stored in the NixOS configuration:

```
$ pamu2fcfg -N -i "pam://nixos-android-builder" -o "pam://nixos-android-builder" -u "user"
```

Password-based authentication is disabled entirely — `login` and `su` require U2F only. The module also provides a `start-shell-if-yubikey-found` script that polls for a YubiKey for 30 seconds and, if one is detected, opens an interactive login shell. This is typically used as the first step in an unattended build pipeline (see below) to allow operators to interrupt automated execution.

To test YubiKey authentication in a VM, pass through the USB device:

```
$ nix run .#run-vm -- -usb -device usb-host,vendorid=0x1050,productid=0x0407
```

Unattended Mode

The `unattended.nix` module enables fully automated build pipelines via `nixosAndroidBuilder.unattended.enable`. When enabled, a `nixos-android-builder` systemd service runs on `tty2` after boot and executes a configurable list of steps sequentially.

Steps are defined in `nixosAndroidBuilder.unattended.steps` as a list of command names. Steps prefixed with `root:` are executed with super-user privileges. The default step sequence is:

```
[  
  "root:start-shell-if-yubikey-found"  
  "select-branch"  
  "fetch-android"  
  "build-android"  
  "android-sbom"  
  "android-measure-source"  
  "copy-android-outputs"  
  "root:lock-var-lib-build"  
  "root:disable-usb-guard"  
  "root:start-shell-and-shutdown"  
]
```

The module also enables **USBGuard**, which blocks USB mass storage devices to prevent unauthorized data exfiltration during builds. The `disable-usb-guard` step stops USBGuard and re-authorizes all USB devices, typically run near the end of the pipeline to allow copying artifacts to USB storage.

Additional helper scripts provided by the module:

- `lock-var-lib-build` – Unmounts `/var/lib/build`, closes the LUKS device, and destroys the encryption key slot, rendering build data permanently inaccessible even before power-off.
- `start-shell-and-shutdown` – Opens an interactive login shell and powers off the system when the user exits.

Fatal Error Handling

The `fatal-error.nix` module provides user-friendly error reporting for critical failures. It overrides systemd's default emergency target behavior with a dialog-based message box on `tty2`.

When a service writes an error message to `/run/fatal-error` and fails, the `fatal-error.service` displays that message in a dialog box with a "Shutdown" button. If no error file is present, a generic message directs the user to check logs on `tty1`.

Debug Mode

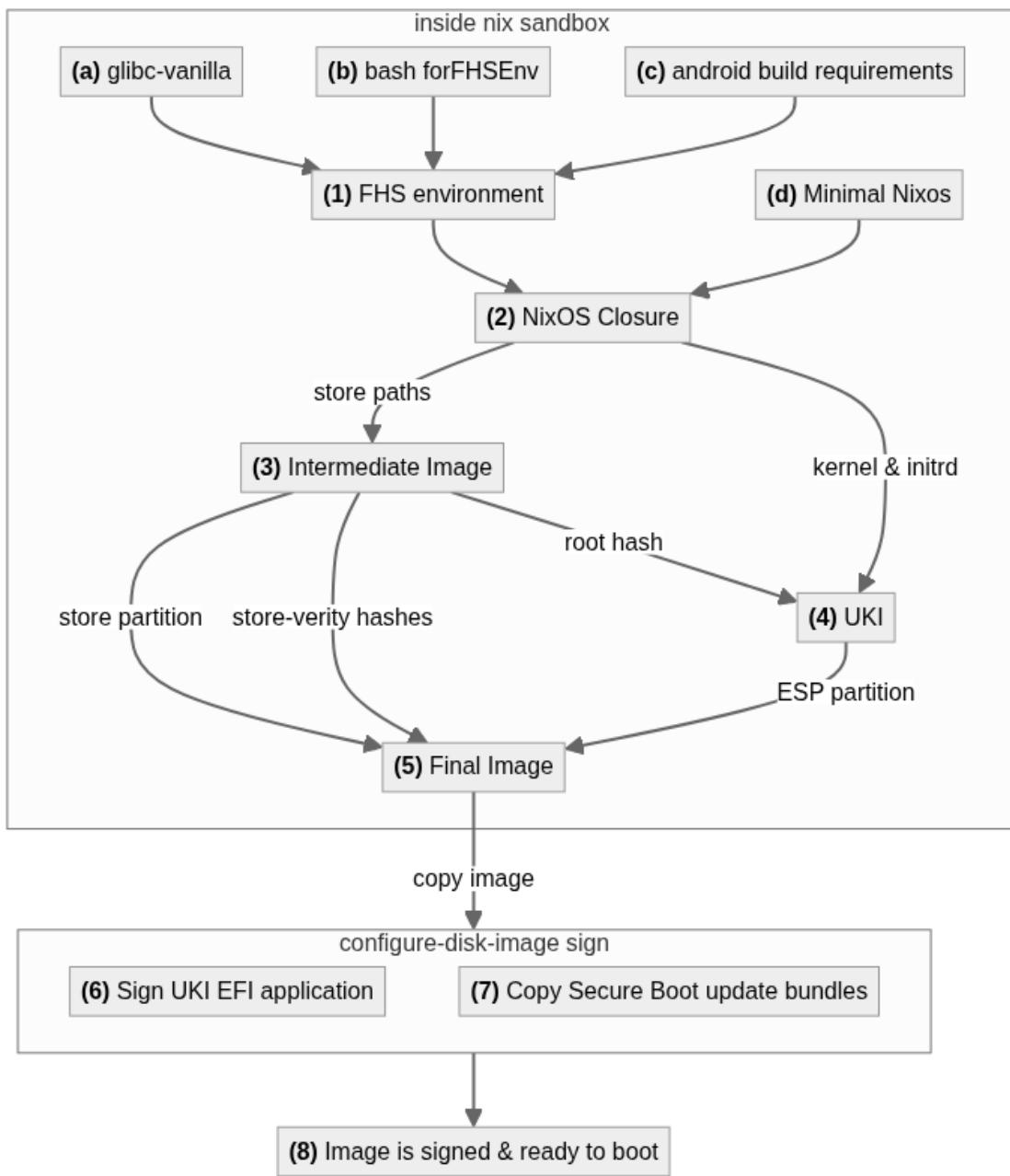
The `debug.nix` module, activated by `nixosAndroidBuilder.debug`, adds conveniences for interactive development and troubleshooting. It is not intended for production use. When enabled, it:

- Grants unauthenticated access to the emergency shell (both in initrd and the main system).
- Adds extra packages: `vim`, `htop`, `tmux`, and `git`.
- Enables `nix` with flake support inside the running system.
- Sets an empty password for the `user` account and enables auto-login.
- Enables password-less `sudo` for `wheel` group members.
- Adds verbose boot logging and a debug shell on `tty3`.

Sequence Chart

Build-time

The following chart depicts a high-level overview on how the different components are assembled into the final disk image at build-time. A detailed description of the steps follows after the chart.



Description

- (1) We start by building an **FHS environment** in a derivation, as outlined above. Main components are:
 - (a) **glibc-vanilla** - NixOS glibc, but with a dynamic linker configured to search FHS paths, such as `/lib`, `/bin`, ...
 - (b) **bash** with `forFHSEnv` set to `true`. NixOS bash does not include `bin` in `PATH` in empty environments. Built with `forFHSEnv` it does.

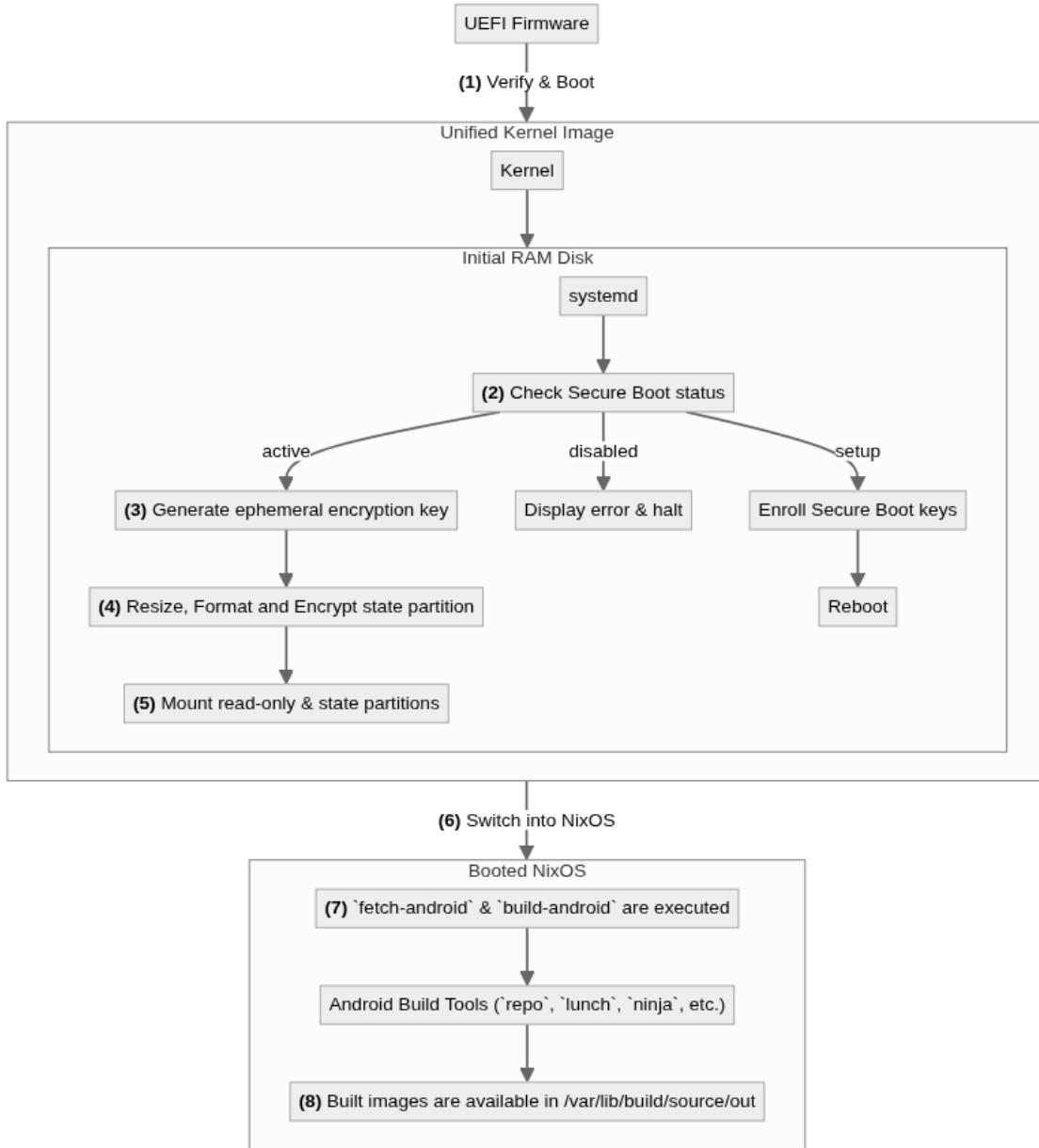
- (c) Android build dependencies that are not shipped in-tree. `repo`, etc.
- (2) The NixOS closure (`system.build.toplevel`) is built, including (d) boot & system services as well as, the `fhsenv` derivation from the previous step.
- (3) First run of `systemd-repart` (`system.build.intermediateImage`):
 - Starts from a blank disk image.
 - Store paths from the NixOS closure are copied into the newly `store` partition.
 - `esp`, `store-verity` and `var-lib-build` are created but stay empty for the moment.
- (4) With a filled store partition, `dm-verity` hashes can be calculated. So we build a new UKI, taking kernel & initrd from the NixOS closure and add the root hash of the `dm-verity` merkle tree to the kernels command line as `usrhash`.
- (5) Second run of `systemd-repart` (`system.build.finalImage`):
 - Starts from the intermediate image from step (3).
 - The `store` and `var-lib-build` partitions are copied as-is.
 - `dm-verity` hashes are written to the `store-verity` partition.
 - The unsigned UKI from step (4) is copied into the `esp` partition.
 - With that being done, the image is built and contains our entire NixOS closure, including the `fhsenv`, in a `dm-verity`-checked store partition, as well as the UKI including `usrhash`.

All that's left to do, is to sign it and prepare it for Secure Boot. The UKI is not yet signed, as doing so inside the nix sandbox, might expose the signing keys. So the user is asked to copy the built image from the nix store to a writable location and execute `configure-disk-image sign` on it. Usage is documented in [user-guide.pdf](#). `configure-disk-image` manipulates the `vfat` partition inside the disk image directly, in order to:

- (6) The UKI is copied to a temporary file, signed, and copied back into the `esp` again.
- (7) Secure Boot update bundles (`*.auth` files) are copied to the `esp` to ensure that `ensure-secure-boot-enrollment.service` can find them during boot.
- (8) We finally have a signed image, ready to flash & boot on a target machine.

Run-time

The following chart depicts a high-level overview on steps that run after the disk image has been booted on target hardware. A detailed description of the steps follows after the chart.



Description

1. The hosts EFI firmware boots into the Unified Kernel Image (UKI), verifying its cryptographic signature if secure boot is active. A service to check that Secure Boot is active runs early in the UKI's initial RAM disk (`initrd`).
2. `ensure-secure-boot-enrollment.service`, asks EFI firmware about the current Secure Boot status.
 - If it is **active** and our image is booting successfully, we trust the firmware here and continue to boot normally.
 - If it is in **setup** mode, we enroll certificates stored on our ESP. Setting the platform key disables setup mode automatically and reboot the machine right after.

- If it is **disabled** or in any unknown mode, we halt the machine but don't power it off to keep the error message readable.
3. Before encrypting the disks, we run `generate-disk-key.service`. A simple script that reads 64 bytes from `/dev/urandom` without ever storing it on disk. All state is encrypted with that key, so that if the host shuts down for whatever reason - including sudden power loss - the encrypted data ends up unusable.
 4. `systemd-repart` searches for the small, empty state partition on its boot media and resizes it before using LUKS to encrypt it with the ephemeral key from (2).
 5. We proceed to mount required file systems:
 - A read-only `/usr` partition, containing our `/nix/store` and all software in the image, checked by `dm-verity`.
 - Bind-mounts for `/bin` and `/lib` to simulate a conventional, FHS-based Linux for the build.
 - An ephemeral `/` file system (`tmpfs`)
 - `/var/lib/build` from the encrypted partition created in (3).
 6. With all mounts in place, we are ready to finish the boot process by switching into Stage 2 of NixOS.
 7. With the system fully booted, we can start the build in various ways. In unattended mode (`nixosAndroidBuilder.unattended.enable`), a configurable sequence of steps is executed automatically. In interactive mode, the following scripts are available:
 - `select-branch` presents a dialog to choose from configured branches (auto-selects if only one is configured).
 - `fetch-android` uses Android's `repo` utility to clone the selected branch from the configured manifest URL to `/var/lib/build/source`.
 - `build-android` sources required environment variables before building the configured `lunch` target.
 - `android-sbom` generates a Software Bill of Materials using upstream AOSP facilities.
 - `android-measure-source` produces a hash over all files in the source checkout.
 - `copy-android-outputs` copies build outputs to `/var/lib/artifacts` (requires artifact storage to be enabled).
 8. Finally, build outputs can be found in-tree, depending on the targets built. E.g. `/var/lib/build/source/out/target/product/vsoc_x86_64_only`.
If `nixosAndroidBuilder.artifactStorage.enable` is set, outputs can be persisted to a second disk via `copy-android-outputs`.

Glossary

AOSP – Android Open Source Project. The publicly available source code for Android maintained by Google.

dm-verity – A Linux kernel feature that provides transparent integrity checking of block devices using a Merkle tree.

EFI/UEFI – Unified Extensible Firmware Interface. The modern firmware interface between the operating system and hardware, replacing legacy BIOS.

ESP – EFI System Partition. A FAT-formatted partition that contains files needed to boot.

FHS – Filesystem Hierarchy Standard. A standard defining the directory structure and contents of traditional Linux systems (e.g., `/bin`, `/lib`, `/usr`).

Flake – A Nix feature providing a standardized way to define reproducible Nix projects with locked dependencies.

initrd – Initial RAM Disk. A temporary root filesystem loaded into memory during boot, used to prepare the real root filesystem.

LUKS – Linux Unified Key Setup. The standard system for Linux disk encryption.

Nix – A purely functional package manager and build system that enables reproducible, declarative builds.

NixOS – A Linux distribution built on Nix, where the entire system configuration is declared in Nix expressions.

nixpkgs – The main repository of Nix packages, containing build instructions for tens of thousands of software packages.

PK/KEK/DB – Platform Key, Key Exchange Key, and Signature Database. Keys used by UEFI Secure Boot to verify boot components.

repo – Google’s tool for managing Git repositories, used extensively in Android development.

SBOM – Software Bill of Materials. A formal inventory of all components and dependencies in a piece of software.

Secure Boot – A UEFI feature that ensures only cryptographically signed software can be booted.

Setup Mode – A Secure Boot state where custom keys can be enrolled. The firmware accepts new keys without signature verification.

SLSA – Supply-chain Levels for Software Artifacts. A security framework for ensuring the integrity of software artifacts throughout the supply chain.

TPM – Trusted Platform Module. A dedicated security chip that provides hardware-based cryptographic functions and key storage.

UKI – Unified Kernel Image. A single EFI executable containing the Linux kernel, initrd, and boot parameters, simplifying Secure Boot signing.