# Contents



# NixOS Android Builder

## Design principles

With the aim of enabling offline SLSA-compliant builds for custom distributions of Android AOSP in mind, we set out to build minimal Linux system with the following properties:

- **Portable** - Runs on arbitrary `x86_64` hardware with `UEFI` boot as well as sufficient disk (>250G) and memory (>64G) 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 traces of neither build inputs or artifacts should remain after a build.
- **Declarative** - All aspects of the build systems are described in nix expression to ensures identical behavior regardless of the build system or time of build.
- **Trusted** - All deployed artifacts such as disk images must be cryptographically signed for tamper prevention and provenance.

We succeeded in quickly creating a modular Proof-of-Concept, based on NixOS, that fullfills most of the properties, with the remaining limitations and plans listed below.

A test-suite based on `qemu` virtual machines was created to enable faster iteration due to accelerated testing cycles.

### Limitations and Further Work

- **aarch64 support** could be added for both, build and target platforms , if there is demand but has neither been implemented yet.
- **unattended mode**: we plan to disable all interactive access in production images, while providing an interactive variant for debugging.
- **artifact uploads**: build artifacts are currently not automatically uploaded anywhere.
- **credential handling** we do not currently implement any measures to handle secrets other than what NixOS ships out of the box. 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

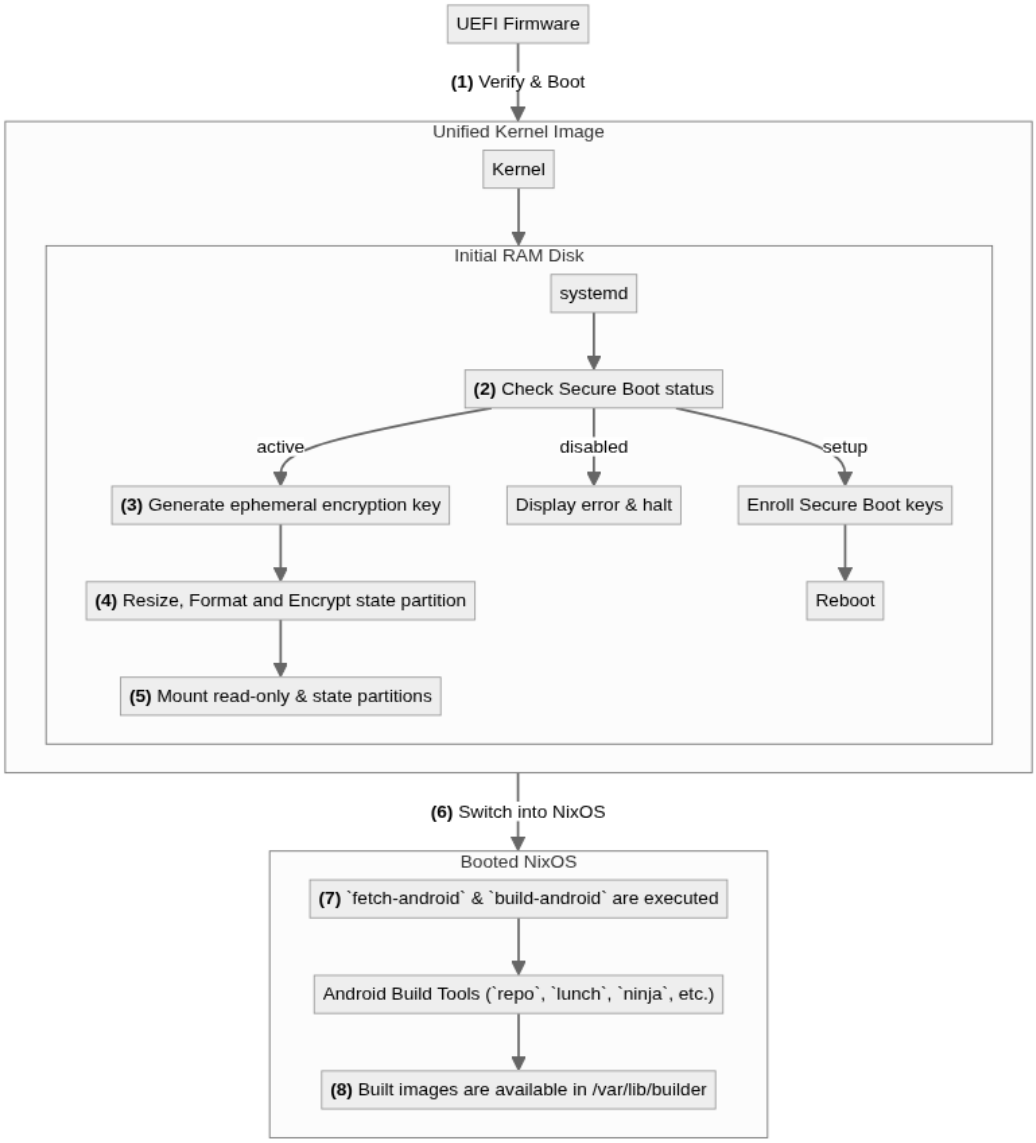
- `NixOS` as Linux Distribution for its declarative module system and flexible boot process.
- `nixpkgs` as software repository for its reproducible approach of building up-to-date Open Source packages.
- `qemu` to emulate generic virtual machines during testing.
- `systemd` to orchestrate upstream components and custom ones while handling credentials and persistent state.
- `systemd-repart` to prepare sign-able read-only disk images for the builder. And to resize, and re-encrypt the state partition on each boot.
- **Linux Unified Key Setup (`LUKS`)** - to encrypt the state partition with an ephemeral key on each boot.
- Various **build requirements** of Android, such as Python 3 and OpenJDK. See definition of `packages` in `android-build-env.nix` for a complete list.
- A complete **Software Bill of Materials** (SBOM) for the builders NixOS closure can be acquired by running the following command from the repository top-level: `nix run github:tiiuae/sbomnix#sbomnix -- .#nixosConfigurations.vm.toplevel`.

## Major Components

- **`AOSP` source distribution**: the only component not shipped with the builders images. Android sources must be cloned into `/var/lib/builder/sources`, e.g. with `fetch-android` described below.
- **`android-build-env`**: a NixOS module to emulate a conventional Linux environment with paths adhering to the File Hierachy Standard (FHS; i.e. `/bin`, `/lib`, ...). We implement this by creating a custom `fhsenv` derivation, containing libraries and binaries from all of Androids build-time dependencies, as well as a custom dynamic linker that searches `/lib` instead of nix store paths. Alternative approaches such as `nix-ld` and `envfs` where considered, but deemed insufficient as they work with individual symlinks to store paths and those links break with sandboxes bind-mounting `/bin` and `/lib`.
- **`nixos-android-builder` source repository**: reproducibly defines all software dependencies of the builder image in a `nix flake`. It pins `nixpkgs` to a specific revision and declaratively describes the builders NixOS system, including `android-build-env`.
- **disk image**: Built from `nixos-android-builder`, ready to be booted on generic `x86_64` hardware. Contains 4 partitions:
  - **`/boot`**: contains the signed Unified Kernel Image as an EFI executable. Mounted read-only at run-time.
  - **`/usr`**: contains `/nix/store`, which is bind-mounted during boot. Verified via `dm-verity` at run-time.
  - **`/usr hash`**: contains `dm-verity` hashes for `/usr` above. Generated during build-time. Its hash is added to kernel parameters in the `UKI` as `usrhash`.
  - **`/var/lib` state partition**: While a minimal, empty partition is built into the image, this partition is meant to be resized & formatted during each boot as described below. It is encrypted with an ephemeral key to prevent data leaks. Its purpose is to store build artifacts that would not fit into memory.

  - No `/` is included here, because the root file system as well as a writable `/nix/store` overlay are kept in `tmpfs` only.
- **Secure Boot keys** those are currently generated manually and stored unencrypted in a local, untracked `keys/` directory in the source repository. It's a users responsibility to keep them safe and do backups. Secure Boot update bundles `*.auth` for enrollment of individual machines are stored unsigned and unencrypted on the `/boot` partition when signing an image.

## Sequence 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.services`, asks EFI firmware about the current Secure Boot status.

- If it is **active** and our image is booting succesfully, 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` 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. The current implementation still includes an inteactive shell and 2 demo scripts which can be used as a starting point:
   - `fetch-android` uses Androids `repo` utility to clone the latest `AOSP` release from `android.googlesource.com` to `/var/lib/build/source`.
   - `build-android` sources required environment variables before building a minimal `x86_64 AOSP` image.
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`. Those are currently not persisted on the builder, so manual copying is required if build outputs should be kept.