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

- **disk signing**: While we do have key generation and signing of the Unified Kernel Image (UKI) for Secure Boot in place. DM-Verity signing of the partition is not yet implemented.
- **`nsjail` patch**: We currently require a small patch in order to build AOSP. Without it, Androids `nsjail` can't currently find dynamic libraries. A fix might require a need to present real binaries to the sandbox in `/bin` instead of symlinks into `/nix/store`.
- **factory reset** does not work as intended yet, due to an upstream issue. This needs to be re-evaluated after upcoming changes in systemd v258.
- `aarch64` support could be added for both, build and target platforms, but has neither been implemented nor tested yet.

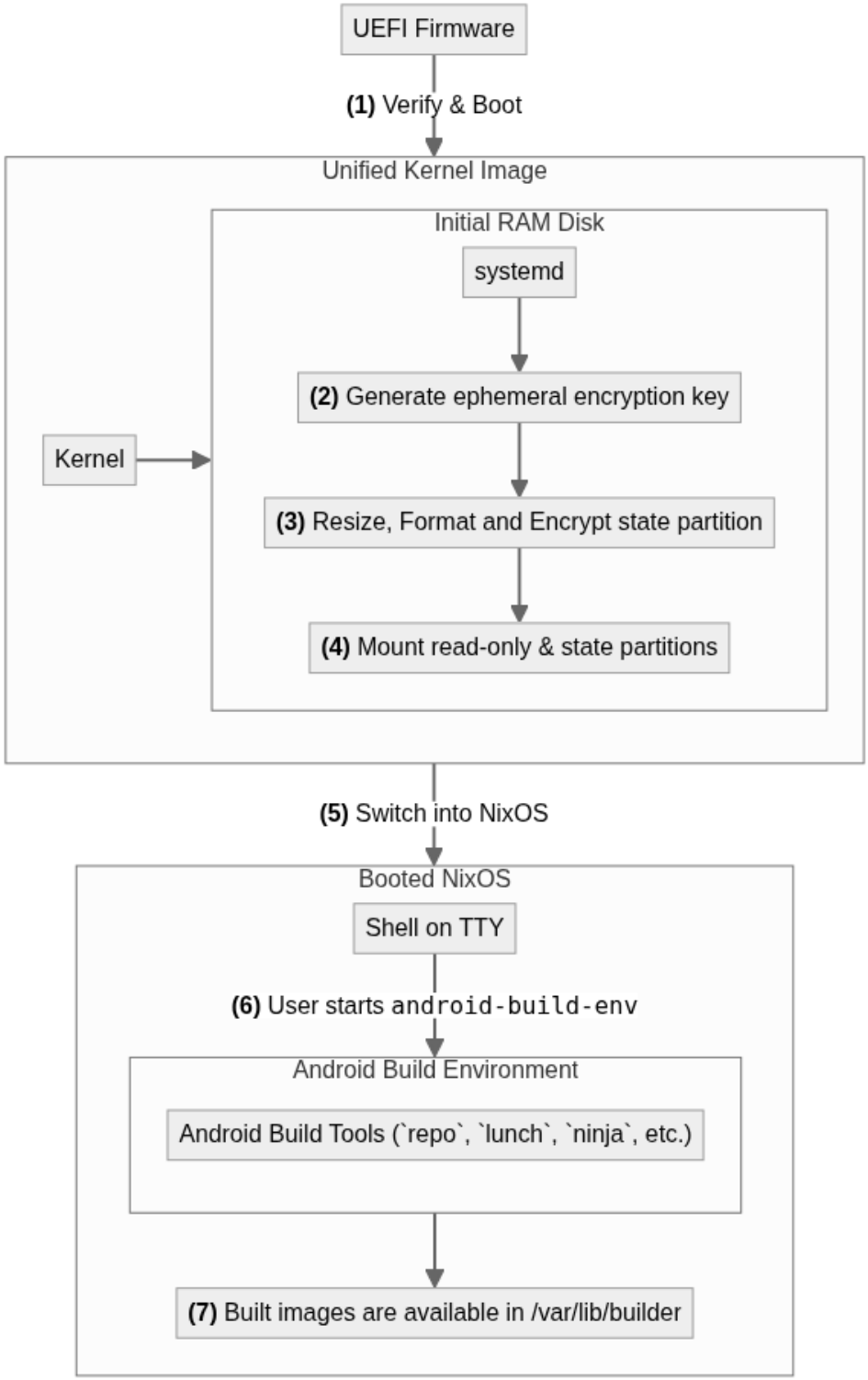
Used technologies

- `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.
- `pkgs.buildFHSEnv` - a helper from `nixpkgs` to run the build inside an enviroment adhering to the Linux Filesystem Hierachy Standard (FHS). Sandboxed via bubblewrap.
- Various **build requirements** of Android, such as Python 3 and OpenJDK. See definition of `targetPkgs` 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`, as in i.e. `fetch-android` described below.
- `android-build-env`: a sandboxed shell containing all required dependencies to build Android that are not included in `AOSP` source distributions in conventional FHS-paths (i.e. `/bin`, `/usr/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 3 partitions:
  - **read-only `/boot` & `/nix/store` partitions**: Those are mounted read-only at run-time. `/boot` only contains the signed Unified Kernel Image as an EFI executable, while `/nix/store` must be signed with `dm-verity` hashes to ensure integrity.
  - **`/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.

Sequence Chart



Description

1. The hosts UEFI firmware boots into the Unified Kernel Image (`UKI`), after verifying its cryptographic signature.
2. One of the first services in the `UKI`s initial RAM disk (`initrd`) is `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.
3. `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).
4. We proceed to mount required file systems:
   - The read-only `/nix/store` from our disk image
   - A writable overlay for that store (`tmpfs`)
   - An ephemeral `/` file system (`tmpfs`)
   - `/var/lib` from the encrypted partition created in (3).
5. With all mounts in place, we are ready to finish the boot process by switching into Stage 2 of NixOS.
6. With the system fully booted, it presents a shell to local user. Users can start a sandboxed shell with `android-build-env` that contains all out-of-tree dependencies required to build `AOSP`, a `FHS` filesystem layout, as well as two scripts for demonstration purposes and as reference:
   - `fetch-android` uses Androids `repo` utility to clone the latest `AOSP` release from `android.googlesource.com` to `/var/lib/build/source`. (and patch it, as mentioned under "Limitations" above).
   - `build-android` sources required environment variables before building a minimal `x86_64` `AOSP` image.
7. 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.