VOTING Stimmunterlagen Offline

Trusted Build

Author	Abraxas Informatik AG
Classification	public
Version	1.1
Date	August, 24th 2023

For digital Switzerland. For Sure.

abraxas

Contents

1. Introduction

The release process for VOTING Stimmunterlagen Offline-Client follows the "trusted-build" procedure. The trusted build is defined on a process and technical level. Both levels include adequate security measures to ensure that the build output (compilation / executable) matches what the source code defines in uncompiled form. The build procedure ensures that all referenced hosted libraries and the application itself can be built at any time in the same version via a deterministic build. External libraries are referenced by a hardened version and thus fulfill their deterministic purpose.

2. Trusted Build

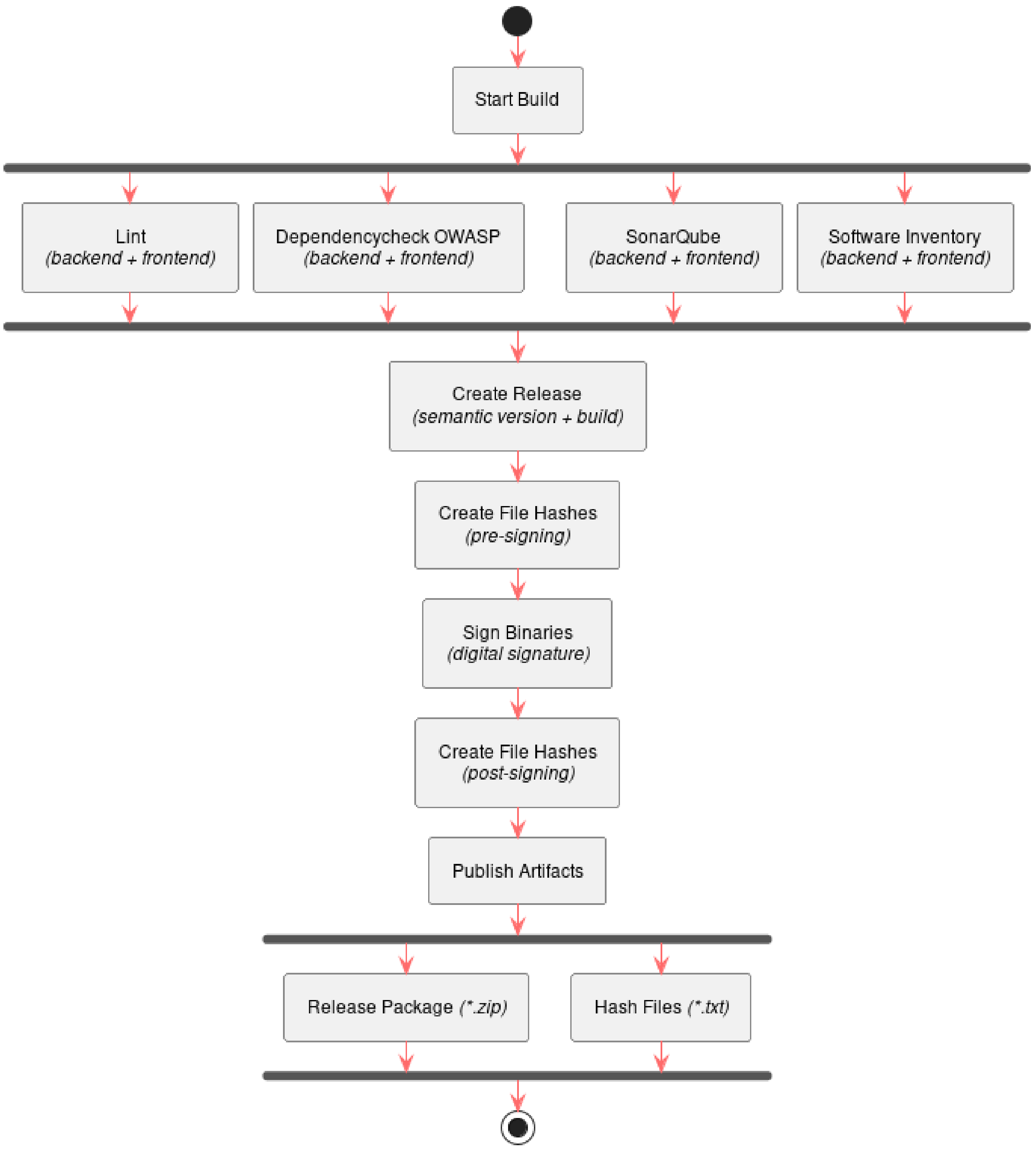
The trusted build is defined by a sequence of defined process steps:

1. At the time of creating a new software release, the client can observe the library and application build process (code compilation) on-site from start to the end until the release artifacts are published.
2. At the end of the build process the following verification is applied:
    a. Library
        - Verify detailed log messages from the build system
        - Verify git commit hash value against the created release
        - Verify hash values from the NuGet Packages with the ones from the production NuGet registry.
        - Verify the semantic build version from the NuGet publish job against the registry NuGet version.

    b. Application
        - Verify detailed log messages from the build system
        - Verify git commit hash value against the created release
        - Verify digital signatures generated for all built binaries and executables during the build process
        - Verify generated SHA-256 hash values for every single file against the release artifacts
        - Verify the semantic build version from the release job against the application's embedded version from the settings page
3. At any point in time, the client can request to re-create a specific release for a specific git commit reference.
    - Verify repeated build with previous build
    - Verify deterministic build which results in exact same binary output.
4. At the time when the new release is installed on the client side, the hash values can be verified to ensure that no manipulations have been made between the creation of the release and the installation.

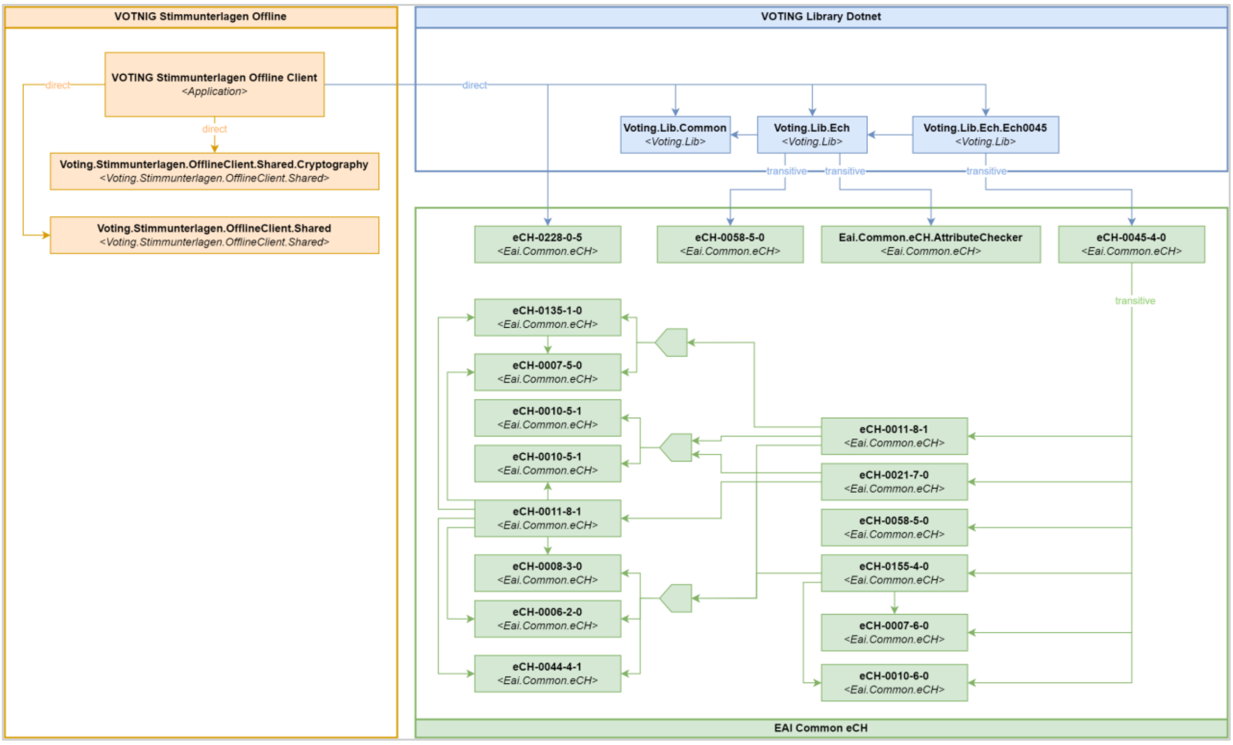
Based on the steps performed by the trusted build, the software's immutability is always ensured and can be verified at any point in time.

2.1 Build Steps


Start Build
Lint (backend + frontend)
Dependencycheck OWASP (backend + frontend)
SonarQube (backend + frontend)
Software Inventory (backend + frontend)
Create Release (semantic version + build)
Create File Hashes (pre-signing)
Sign Binaries (digital signature)
Create File Hashes (post-signing)
Publish Artifacts
Release Package (*.zip)
Hash Files (*.txt)

3. Dependency Tree

The application VOTING Stimmunterlagen Offline Client depends on a subset of direct and transitive dependencies from different library projects. The deterministic build procedure requires each of the packages to be built and packaged in advance to successfully create the final application artifacts.



4. Deterministic Build

The application and the build process is configured to be deterministic and allows reproducible builds as part of the overall chain of trust. The produced release artifacts are identical byte-for-byte across builds for the same code base. The deterministic build is a prerequisite for the trusted build process and proves that the release artifacts are compiled from a trusted source code. The following chapters describe various approaches that make it possible to support deterministic builds.

4.1 Software Version Hardening

- npm dependencies are configured to not accept any update types (~ | ^ | *). Dependencies are always pinned to an explicit semantic version.
- npm dependency tree package-lock.json is added under source control.
- npm packages are installed from an internal trusted registry endpoint.
- nuget package dependencies are always pinned to an explicit semantic version.
- nuget packages are installed from an internal trusted registry endpoint.
- repetitive referencing nuget packages are collected through central Directory.Build.props build configuration file for better maintainability and clarity.

4.2 Build Infrastructure Hardening

- build jobs for CI/CD use hardened and pinned templates.
- build jobs are based on docker images referenced by pinned digest (sha256).
- dependencies and versions are stored within a software inventory system for every build.
- static code analysis and dependency vulnerability reports are stored centrally on SonarQube.

4.3 Prove of Integrity

To ensure that the integrity and determinism of the release artifacts can be verified at any time, hash values are created for each file at the time of building the application. The hash values can be verified at any time in production. Because the hash values change when signing the artifacts using a digital certificate, the hash values are generated before and after signing.

- Create hash values for all release artifacts before and after applying the digital signatures using the core util sha256sum
    - Hash file before signing: release_artifacts_before_signing.sha256
    - Hash file after signing: release_artifacts.sha256
- Publish both generated hash files as part of the release to prove integrity.
- Verify hashes at any point in time, e.g *sha256sum -c release_artifacts.sha256*

5. Deterministic Build - Runbook

In order to reproduce the deterministic build procedure on a local environment, the underlying build infrastructure must be fully hardened and equal to the automated CI/CD release process. To achieve the deterministic build all libraries and the application itself are built based on predefined and hardened docker images. From a dependency tree perspective, all dependencies must be built bottom-up to successfully build the final application.
All prerequisites required to build the libraries and application are available from GitHub | Trusted-Build. The artifacts can be built from a single script which in turn calls individual deployment scripts (bottom-up principle). Every build is based on a pre-defined set of configurations (*.env) that describes the target to be built. The configurations can be customized depending on the desired target versions.
Prerequisites:

- Docker Engine (e.g Docke for Windows)
- Docker Compose (part of Docker Desktop)
- PowerShell (v7 for cross-platform purposes)

Build instruction:

- Download build scrips from GitHub | Trusted-Build.
- Open PoweShell and navigate to the script folder.
- Run *deploy-all.ps1* to build all libraries and the application:
    a. Application artifacts (docker mount): *.\deplyoment*
    b. Library artifacts (docker mount): *.\packages*

The application can be started by running the executable from *.\deplyoment\package\voting-stimmunterlagen-offline-win32-x64\voting-stimmunterlagen-offline.exe*

5.1 Deterministic Build Exclusions

It is important to note that all *runtimeconfig.json and *deps.json files from the file *release_artifacts_before_signing.sha256* do not match with the published hash values on GitHub. All projects that have direct or transitive dependencies on locally created NuGet packages are affected. While the hash values of the locally created binaries (*.dll) match, the hash values of the generated NuGet packages differ due to timestamps and other random metadata added to the NuGet package. For the build process the hash values of the binaries are relevant and not the NuGet package itself. Since the .NET Framework specifies the hash values of the NuGet packages in the *.deps files, these discrepancies consequently occur.
In summary: NuGet packages cannot be created deterministically via the .NET CLI (except with manual hacks), while determinism is ensured at all times for the binaries they contain.

5.2 Deterministic Build Scripts / Configs

File(s)	Purpose
*.env	Library deployment base configuration files
*.overwrite.env	Library deployment custom configuration files (secrets)
docker-compose-deploy-library.yml	Application specific docker compose file used for processing unattended deployment.
docker-compose-deploy-client.yml	Generic Library docker compose file used for processing unattended deployment.
deploy-*.ps1	Individual deployment scripts for library and application. Initializes base and custom *.env configurations used by docker compose.
deploy-all.ps1	Overall deployment. Calls all individual deployment scripts in correct order and unattended mode.
deploy-library.sh	Generic library shell script responsible to clone target repository, restore and package projects as NuGet package output.
deploy-client.sh	Specific application shell script responsible to build final application artifact based on direct and transitive dependency artifacts.
.npmrc	NPM Registry configuration. Uses public only registry.
nuget.config	NuGet Registry configuration. Uses public and local directory-based registries.
stimmunterlagen-ofline-client.dockerfile	The application binaries are built based on a hardened Dockerfile base image. All referenced and installed software and library components are fully pinned to the exact version. The image that is built out of the Dockerfile specification is stored on an internal

File(s)	Purpose
	docker registry and uniquely tagged using semantic versioning pattern. Every change to the Dockerfile generates a new image with an increased version. Since the base image is always referenced by the build system with its image digest (sha256), reproducible builds are guaranteed by the underlying host.

6. Code Signing

All self-managed binaries (dll, exe) are protected with a code signing certificate. A full list of the target binaries is available from the create-signature.sh script. For signing osslsigncode is used. The tool is an alternative to Microsoft's signtool.exe and can be fully integrated into the Linux Docker Image based CI/CD process. A code signing certificate from Abraxas Informatik AG is used for signing (fingerprint: c023304bcd2d3d337abfdee1f209833b26f2420c). The signature is used to ensure authenticity and integrity. A static timestamp is used to preserve deterministic from signed binaries when re-building the solution during a trusted build procedure.

7. Quality Assurance

The trusted build is not limited to ensuring integrity and authorship, but also requires quality assurance at the time of the build. Each build of the VOTING Stimmunterlagen Offline-Client is based on the following quality measures and gates:

- using hardened build templates
- using git as a source control system with hashed commits
- using signed commits with committer validation based on gpg cryptography
- using strict merge request policies
  - pipeline must succeed
  - prevent self-approval, remove approvals on changes
- using strict merge request quality gates
  - treat all warnings as errors in build step
  - apply linter for backend and frontend
  - apply dependency-check for backend and frontend
  - apply SonarQube static code analysis for backend and frontend