VOTING Stimmunterlagen Offline

Trusted Build

Author	Abraxas Informatik AG
Classification	public
Version	1.0
Date	March, 31st 2023

For digital Switzerland. For Sure.

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.

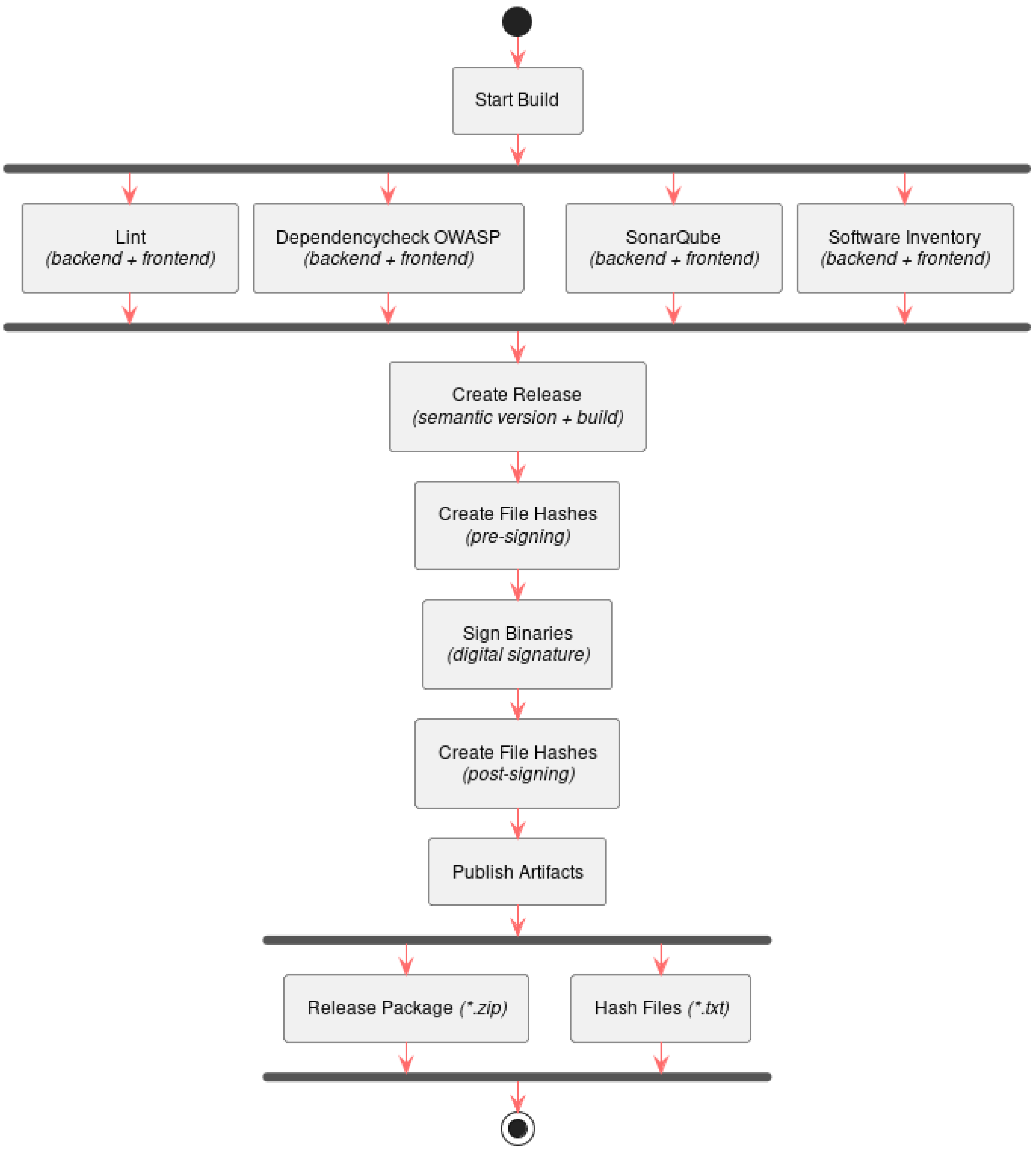
1.1 Trusted Build

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 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:
    - 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.

1.1.1 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)

1.2 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.

1.2.1 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

1.2.2 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

1.2.3 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. Since the deterministic is lost by signing the artifacts using a digital certificate and timestamp in favor of security, 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 script *create-signatures.sh*
- Publish script *Compare-FileHashes.ps1* for verifying hashes as part of the release artifacts. The script is hashed and part of the reference hash file to prevent manipulation.
- Publish hash files *filehashes_build.txt* and *filehashes_build_before_signing.txt* separately with the release to prove integrity and/or deterministic

1.3 Build Scripts

1.3.1 Docker Base Image

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 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.

The following is an exemplary example:

```
FROM node:16.13.0-bullseye

ENV \
    # Enable detection of running in a container
    DOTNET_RUNNING_IN_CONTAINER=true \
    # Enable correct mode for dotnet watch (only mode supported in a container)
    DOTNET_USE_POLLING_FILE_WATCHER=true \
    # Skip extraction of XML docs - generally not useful within an image/container - helps
performance
    NUGET_XMLDOC_MODE=skip \
    # disable telemtry
    DOTNET_CLI_TELEMETRY_OPTOUT=1 \
    # skip first time
    DOTNET_SKIP_FIRST_TIME_EXPERIENCE=1

ARG DOTNET_VERSION=6.0.406
ARG BIN=/usr/bin
ARG DOTNET_BIN=/usr/share/dotnet
ARG WINE_VERSION=8.0.0.0~bullseye-1

# install utils
RUN apt-get update && \
    apt-get install -y git-lfs=2.13.2-1+b5 osslsigncode=2.1-1 && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

# install dotnet
RUN curl --fail -L "https://dot.net/v1/dotnet-install.sh" -o ./dotnet-install.sh && \
    chmod +x ./dotnet-install.sh && \
    ./dotnet-install.sh -v "${DOTNET_VERSION}" --install-dir ${DOTNET_BIN} && \
    rm ./dotnet-install.sh && \
    ln -s ${DOTNET_BIN}/dotnet ${BIN}/dotnet

# install wine (to build electron apps for windows)
RUN dpkg --add-architecture i386 && \
    mkdir -pm755 /etc/apt/keyrings && \
    wget -O /etc/apt/keyrings/winehq-archive.key https://dl.winehq.org/wine-
builds/winehq.key && \
    wget -nc https://dl.winehq.org/wine-builds/debian/dists/bullseye/winehq-
bullseye.sources && \
    mv winehq-bullseye.sources /etc/apt/sources.list.d && \
    apt-get update && \
    apt-get install -y --no-install-recommends \
        winehq-stable=${WINE_VERSION} \
        wine-stable=${WINE_VERSION} \
        wine-stable-amd64=${WINE_VERSION} \
        wine-stable-i386=${WINE_VERSION} && \
    apt-get clean && rm -rf /var/lib/apt/lists/*

# install semantic release
RUN npm i -g \
    semantic-release@19.0.2 \
    @semantic-release/gitlab@6.2.2 \
    @semantic-release/exec@5.0.0

ENTRYPOINT []
```

1.3.2 Offline Client Release Job

```
release:
  image: $DOCKER_REGISTRY/stimmunterlagen-offline-client:$DIGEST
  stage: release
  rules:
    - if: $CI_COMMIT_BRANCH == $CI_DEFAULT_BRANCH
  script:
    - ...
    - semantic-release
```

1.3.3 Offline Client Build Job

```
build:
  image: $DOCKER_REGISTRY/stimmunterlagen-offline-client:$DIGEST
  stage: build
  rules:
    - if: $CI_COMMIT_BRANCH && $CI_COMMIT_BRANCH != $CI_DEFAULT_BRANCH ||
$CI_MERGE_REQUEST_IID
  script:
    - ...
    - sh ./build.sh
```

1.4 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