The Notary Project Security Audit

Technical report focused on new cryptographic features

Reference	24-10-1825-LIV
Version	1.5
Date	2025-01-12

Quarkslab SAS
10 boulevard Haussmann
75009 Paris
France

1. Project Information

1.1. Document history

Version	Date	Details	Authors
1.0	2024/09/27	Initial version	Dahmun Goudarzi Sebastien Rolland
1.1	2024/11/12	Updated version after applied fixes	Dahmun Goudarzi Sebastien Rolland
1.2	2024/11/22	Updated version after more applied fixes	Dahmun Goudarzi Sebastien Rolland
1.3	2024/12/11	Updated version after more applied fixes	Dahmun Goudarzi Sebastien Rolland
1.4	2024/12/20	Updated version after more applied fixes	Dahmun Goudarzi Sebastien Rolland
1.5	2025/01/12	Added CVEs	Dahmun Goudarzi Sebastien Rolland

1.2. Contacts

1.2.1. Quarkslab

Contact	Role	Email
Frédéric Raynal	CEO	fraynal@quarkslab.com
Ramtine Tofighi Shirazi	Project Manager	mrtofighishirazi@quarkslab.com
Dahmun Goudarzi	R&D Engineer	dgoudarzi@quarkslab.com
Sébastien Rolland	R&D Engineer	srolland@quarkslab.com

1.2.2. Client

Contact	Role	Email
Derek Zimmer	Executive Director (OSTIF)	derek@ostif.org
Amir Montazery	Managing Director (OSTIF)	amir@ostif.org
Helen Woeste	Communications Manager (OSTIF)	helen@ostif.org
Shiwei Zhang	Notary Project maintainer	shizh@microsoft.com
Yi Zha	Notary Project maintainer	yizha1@microsoft.com
Feynman Zhou	Notary Project maintainer	feynmanzhou@microsoft.com
Pritesh Bandi	Notary Project maintainer	pritesb@amazon.com
Vani Rao	Notary Project maintainer	vaninrao@amazon.com

Contents

2. Executive Summary

2.1. Context

The Open Source Technology Improvement Fund, Inc. (OSTIF) engaged with Quarkslab to perform a security audit of the Notary project, focused on three new features.

The Notary Project is a set of specifications and tools intended to provide a cross-industry standard for securing software supply chains by using authentic container images and other OCI artifacts. Notation Project specification and tooling provide signing and verification workflows for OCI artifacts, signature portability across OCI compliant registries, and integration with 3rd party key management solutions through a plugin model.

The Notary Project is part of the Cloud Native Computing Foundation (CNCF) projects.

The OSTIF and Quarkslab have collaborated on several security assessments through the years, in the context of securing widely used and crucial open-source projects, such as

- Audit of Operator Fabric, 2024
- Cloud Native Buildpacks security audit, 2024
- Kuksa security audit, 2024
- Falco security audit, 2023

This report presents the results of the security assessment, performed in 25 days.

Info

This report reflects the work and results obtained within the duration of the audit and on the specified scope, as agreed between the OSTIF, The Notary Project, and Quarkslab.

Tests are not guaranteed to be exhaustive and the report does not ensure that the code is bug or vulnerability free.

2.2. Objectives

The defined objectives for this collaboration were to perform a security review focused on new enhancements applied on The Notary Project version `v1.2.0` and `v1.3.0`, namely

- Timestamping support (involving notation, notation-go, notation-core-go, and tspclient-go)
- Revocation checking with CRL (involving notation-go and notation-core-go)

2.3. Methodology

To assess the security of The Notary Project's new features, Quarkslab's team first needed to familiarize themselves with the structure of the project and understand the key tasks outlined in the audit's scope. To achieve this, Quarkslab experts gathered and reviewed the available documentation and project resources. With a clear understanding of the features to be evaluated, Quarkslab performed tests to evaluate all requested features.

The evaluation employed a combination of dynamic and static analysis. The static analysis focused on scrutinizing the source code to identify vulnerabilities related to the implementation and logic of the specified assessment targets. Dynamic analysis was used to complement the static review by speeding up the process through fuzzing and validating or refuting the hypotheses generated during the static analysis.

Overall, the following steps were defined for the security audit:

1. **Step 1: Discovery**
   - Focus on the Notary Project new enhancements and associated documentation;
   - Gain an understanding of the security guarantees imparted to the newly implemented enhancements.
2. **Step 2: Threat model**
   - Given the precise scope, definition of tests to be applied based on Step 1 based on relevant threats.
3. **Step 3: Manual code review**
   - Manual code review to find potential bad practices, bug, and/or vulnerabilities in the new enhancements implementation.
4. **Step 4: Cryptographic review**
   - Review of cryptographic primitive usage.
   - Assessment of compliancy to best practices and of potential security issues.
5. **Step 5: Dynamic testing**
   - Application of dynamic tests to assess the resiliency of newly implemented enhancements or to validate potential findings from Steps 3 and 4.

2.4. Findings Summary

During the time frame of the security audit, Quarkslab has discovered several security issues and vulnerabilities, among which:

- 1 security issues considered as medium severity;
- 1 security issues considered as low severity;
- 9 issues considered informative.

Security issues were reported directly using the project GitHub security advisories.

Info

Two notable findings were reported and fixed by The Notary Project maintainers:

- MEDIUM-5: GHSA-45v3-38pc-874v / CVE-ID: CVE-2024-56138
- LOW-11: GHSA-qjh3-4j3h-vmwp / CVE-ID: CVE-2024-51491

Informative findings are only provided in this report.

ID	Name	Perimeter
MEDIUM-5	Revocation in certificate chain unchecked while signing	Time-stamping verification
LOW-11	Non-portable way of creating temporary files for CRL's cache	CRL Cache Creation
INFO-1	Unused risky flag `NoNonce`	Time-stamping request
INFO-2	Option for user to choose the `Nonce`	Time-stamping request
INFO-3	Lack of check after URL parsing	CLI signing with time-stamping
INFO-4	Abort TSP HTTP Request Response Validation if invalid signature	Time-stamping verification
INFO-6	Non-compliant to the RFC for verification of signed attributes	Time-stamping verification
INFO-7	Abort Counter-signature verification if invalid signature	Verification with time-stamping
INFO-8	Shallow Verification of TSA trust store certificates	Certificate verification
INFO-9	No proper error handling when OCSP or CRL are not available	Revocation status verification
INFO-10	Non-compliant use of HTTP Status Code	Fetch of CRL

Info

This report was updated to indicate The Notary Project maitainers actions to handle and fix all provided issues. To that purpose, a note has been added to each issues for which a pull request has been made.

2.5. Recommendation and Action Plan

For each of the security issues and informative findings, Quarkslab suggests different ways to tackle them as action plans for *quick wins*.

ID	Name
MEDIUM-5	Add a check to certification chain to verify their status while signing.
LOW-11	The file should be copied instead of being moved, or, directly created in the user cache directory and then renamed. First solution can be implemented thanks `os.Open`, `os.Create`, `io.Copy` and `os.Remove` from standard Go library.
INFO-1	Remove this field since it does not bring anything relevant to the protocol
INFO-2	Remove the possibility to use a user-provided `Nonce` (if statement of the code snippet).

INFO-3	Add a test after parsing the URL, such as the following one:
	`if (u.Scheme == ""  || u.Host == "") { // where u is the return value of url.Parse`
	`  // return error`
	`}`

INFO-4	Verify now the signature before further processing.
INFO-6	Either comply to RFC 5652 at this stage and handle the case in the caller method (e.g. `timestamp.Timestamp`), or handle the error at this stage.
INFO-7	Verify the signature of the time-stamp token before continuing to extract and parse the rest of the information in the TST.
INFO-8	Have the same level of verification as in init phase or move the later verification to this stage.
INFO-9	Another level of verification could be implemented, so that `strict` mode raises an error, or at least prints warning logs if the revocation checks are not available, especially when the certificate chain contains more than one certificate.
INFO-10	Reject any response containing anything but HTTP 200 as status code.

2.6. Conclusions

Quarkslab identified several issues or bugs in Notary projects, however only one of them may involve an immediate safety risk. Quarkslab recognizes the considerable security efforts made by Notary developers to safeguard the tool, mainly thanks to conscientious implementation of the different related RFCs. Additionally, Quarkslab provided recommendations and strategies for addressing the issues, helping to strengthen the open-source tool and enhance its security moving forward.

3. Reading Guide

This reading guide describes the different sections present in this report and gives some insights about the information contained in each of them and how to interpret it.

3.1. Executive summary

The executive summary Section 2. presents the results of the assessment in a non-technical way, summarizing all the findings and explaining the associated risks. For each vulnerability, a severity level is provided as well as a name or short description, and one or more mitigation, as shown below.

ID	Name	Category
CRITICAL	Vulnerability Name #1	Injection
HIGH	Vulnerability Name #2	Remote code injection
MEDIUM	Vulnerability Name #3	Denial of Service
LOW	Vulnerability Name #4	Information leak

Each vulnerability is identified throughout this document by a unique identifier `<LEVEL>-<ID>` , where `ID` is a number and `LEVEL` the severity (`INFO`, `LOW`, `MEDIUM`, `HIGH` or `CRITICAL`). Every vulnerability identifier present in the vulnerabilities summary table is a clickable link that leads to the corresponding technical analysis that details how it was found (and exploited if it was the case). Severity levels are explained in Section 3.2. .

The executive summary also provides an action plan with a focus on the identified *quick wins*, some specific mitigation that would drastically improve the security of the assessed system.

3.2. Metric definition

This report uses specific metrics to rate the severity, impact and likelihood of each identified vulnerability.

3.2.1. Impact

The impact is assessed regarding the information an attacker can access by exploiting a vulnerability but also the operational impact such an attack can have. The following table summarizes the different levels of impact we are using in this report and their meanings in terms of information access and availability.

CRITICAL	Allows a total compromise of the assessed system, allowing an attacker to read or modify the data stored in the system as well as altering its behavior.
HIGH	Allows an attacker to impact significantly one or more components, giving access to sensitive data or offering the attacker a possibility to pivot and attack other connected assets.

MEDIUM	Allows an attacker to access some information, or to alter the behavior of the assessed system with restricted permissions.
LOW	Allows an attacker to access non-sensitive information, or to alter the behavior of the assessed system and impact a limited number of users.

3.2.2. Likelihood

The vulnerability likelihood is evaluated by taking the following criteria in consideration:

- **Access conditions**: the vulnerability may require the attacker to have physical access to the targeted asset or to be present in the same network for instance, or can be directly exploited from the Internet.
- **Required skills**: an attacker may need specific skills to exploit the vulnerability.
- **Known available exploit**: when a vulnerability has been published and an exploit is available, the probability a non-skilled attacker would find it and use it is pretty high.

The following table summarizes the different level of vulnerability likelihood:

CRITICAL	The vulnerability is easy to exploit even from an unskilled attacker and has no specific access conditions.
HIGH	The vulnerability is easy to exploit but requires some specific conditions to be met (specific skills or access).
MEDIUM	The vulnerability is not trivial to discover and exploit, requires very specific knowledge or specific access (internal network, physical access to an asset).
LOW	The vulnerability is very difficult to discover and exploit, requires highly specific knowledge or authorized access

3.2.3. Severity

The severity of a vulnerability is defined by its impact and its likelihood, following the following table:

		Impact			
		●●●●	●●●○	●●○○	●○○○
Likelihood	●●●●	CRITICAL	CRITICAL	HIGH	MEDIUM
	●●●○	CRITICAL	HIGH	HIGH	MEDIUM
	●●○○	HIGH	HIGH	MEDIUM	LOW
	●○○○	MEDIUM	MEDIUM	LOW	LOW

4. Discovery

4.1. Projects Informations

The Notary project is an initiative incubated by the Cloud Native Computing Foundation in 2017 that offers a collection of libraries for supporting signing and verification of OCI artifacts. The main developers are from Microsoft and Amazon and the project is supported by main actors such as Amazon Web Services, Microsoft, Zot registry, or Harbor, among others.

For this specific audit, the scope was reduced to two specific features:

- Time-stamping protocol (RFC 3161) and its implementation,
- Revocation check with CRL (RFC 5280) and its implementation.

4.1.1. Projects Breakdown

The Notation project source code is divided into four projects: notation, notation-go, notation-core-go, and tspclient-go.

Project Name	Description	URL	Git Commit Hash
notation	Source code of the convenient CLI implementation of new Notary Project specifications.	https://github.com/notaryproject/notation	`1AF69FC9E184F2EB9E1928F2D66DC0471793491`
notation-go	Source code for Go library of the new Notary Project signing and verification flow.	https://github.com/notaryproject/notation-go	`694E3A0314B5ECB7F04D100BAE6249C527ABFD47`
notation-core-go	Source code for Go library of the Notary Project signature specification and wrapping (COSE and JWS).	https://github.com/notaryproject/notation-core-go	`55E35686875491A8D6AD8A47DE9C566383FB1B42`
tspclient-go	Source code for Go implementation of the TSP client.	https://github.com/notaryproject/tspclient-go	`DF25Ef8D21722B66F866CDC13F218473648126DC`

Warning

The above commit hashes correspond to the most up-to-date ones when the audit has started. However, as the certificate revocation check based on CRL feature was implemented later, Quarkslab's engineers had to update them in order to audit it. More details are provided in the relevant CRL Analysis section 8.

The three projects, `notation-go`, `notation-core-go`, and `tscpclient-go`, serve as backend for the command line tool implemented by the `notation` project. The implementation is mainly written in Golang and the four projects represent around 12.000 lines of code (including the numerous test functions).

Please note that the Notary project (excluding `tspclient-go`) has been already audited in 2018 by Cure53 and 2023 by ADA Logics.

4.1.2. Specification and Guides

A detailed specification is provided on GitHub. It includes a presentation of the project, the requirements, security documents such as the previous audits, the specifications, a threat model, and how to join their public channels and meetings. Notably, the specifications and the threat model allow them to have a very clear understanding on how the command line tool `notation` and its different features work to obtain and verify signatures for container registry.

https://notaryproject.dev/docs/user-guides/: The Notary project proposes a very thorough and complete guide on how to use `notation`. It is composed of:

- Installation guide
- Tutorials
- How-to guides
- CLI reference
- Best practice for deploying Notation
- Experimental features guide
- Common problems troubleshooting

The different examples and best practices seem up-to-date with the version of Notation audited.

Another notable documentation is the Microsoft tutorial on signing container images with notation that you can find here.

4.1.3. Previous Audits

Year	Audit Scope	Auditors	Report
2018	TUF, Notary	Cure53	PDF
2023	notation, notation-go, notation-core-go (security audit)	ADA Logics	PDF
2023	notation, notation-go, notation-core-go (fuzzing)	ADA Logics	PDF

Please note that TUF stands for “The Update Framework”, a client-server software for interaction with trusted collections. `notary` is the very first TUF-based implementation circa 2016.

4.2. Installation and Debug

4.2.1. Build

To build the `notation` Command Line Interface (CLI), one has to clone the GitHub repository, and run `make build` or directly start the build of the `notation` target using the `go` command. The commands below allow to download, build and install the CLI binary:

```
git clone https://github.com/notaryproject/notation
cd notation
go build ./cmd/notation
mv ./notation /usr/local/bin/
```

4.2.2. Configuration

4.2.2.1. Start Local Registry

A local registry can be started using docker and the following command:

```
docker run -d -p 5001:5000 -e REGISTRY_STORAGE_DELETE_ENABLED=true --name registry
registry
```

This is useful for testing and debugging purposes.

4.2.2.2. Sign and Verify Artifacts

In order to sign and verify artifacts, `notation` CLI requires signing keys, certificates and a trust policy.

`notation` can either use locally configured data, or use Azure Key Vault / AWS Signer to securely store and use them.

As this part is not included in the audit scope and requires extra configuration steps, we chose the easiest way to set up `notation` and therefore installed self-signed certificates.

4.2.2.3. Certificate and Key generation

This can simply be done by running the following command:

```
notation certificate generate-test test_cert
```

4.2.2.4. Trust Policy

To configure the trust policies, one needs to create a `JSON` file in `.config` (or directly create it using VSCode):

```
touch ~/.config/notation/trustpolicy.json
```

and fill it with the following data:

```
{
  "version": "1.0",
  "trustPolicies": [
    {
      "name": "quarkslab-test",
      "registryScopes": [ "*" ],
      "signatureVerification": {
          "level" : "strict"
      },
      "trustStores": [ "ca:test_cert" ],
      "trustedIdentities": [
          "*"
      ]
    }
  ]
}
```

This default and simple trust policy specifies how signature verification should be handled by `notation`:

- `registryScopes`: specifies which trust policy is applicable for a given artifact.
- `signatureVerification/level`: specifies the verification level. **strict** enforces all checks except Time-Stamp Authority (TSA) if no truststore has been configured. More data can be found here.
- `trustStores`: specifies the trust stores applications for this trust policy.
- `trustedIdentities`: specifies which identities, defined by their Distinguished Name (DN), can be used from the trusted stores signed certificates.

4.2.3. Verify TSA Signatures

`notation` allows to add time-stamping as defined by the Time-Stamp Protocol (see RFC 3161) during the signing process, with a trusted third party called the TSA.

In order to verify time-stamp signatures, the root certificate of the chosen TSA must be installed and the trust policy updated.

4.2.3.1. Add a TSA Root Certificate

For testing purposes, we choose to set up DigiCert TSA, as documented by Notary's documentation. The root certificate is available here. Once downloaded, the certificate can simply be added with the CLI as follows:

```
notation cert add --type tsa --store tsa_test_cert DigiCertTrustedRootG4.crt
```

4.2.3.2. Update the Trusted Policies

In order to use this new trusted store, one needs to update the `trustStore` property of the trusted policy in order to add it:

```
"trustStores": [ "ca:test_cert", "tsa:tsa_test_cert" ],
```

4.2.4. Debugging with VSCode

In order to have proper debugging accesses and interfaces, VSCode can be configured. To do so, the following setup needs to be done:

- Install VSCode Go extension.
- Build `notation` CLI.
- Create `.vscode/launch.json` in root `notation` repository.
- Paste the following code in the JSON file:

```json
{
  "version": "0.2.0",
  "configurations": [
    {
        "name": "Launch Package",
        "type": "go",
        "request": "launch",
        "mode": "auto",
        "program": "${cwd}/cmd/notation",
        "args": ["sign",
            "--timestamp-url",
            "http://timestamp.digicert.com",
            "--timestamp-root-cert",
            "${cwd}/DigiCertTrustedRootG4.crt",
            "--signature-format",
            "jws",
            "localhost:5001/ubuntu:latest"
        ]
    }
  ]
}
```

Info

Modify `args` according to the location of different files and configuration you would like to test.

4.3. Scope Exploration

Even though the scope is limited to the time-stamp feature of `notation` and the CRL verification, the discovery was made on a larger scope.

To explore the different features brought by the `notation` command line and get a better understanding of the target project scope, we used the different commands available on some use cases and perform some basic dynamic tests. We noticed that in the initial target scope the signing of arbitrary blob was asked to be audited, but the feature is not implemented as of the time of the audit.

4.3.1. Signing Blobs

Scope:

- `notation-core-go/signature`
- `notation sign *`

The `notation` CLI allows signing either Open Container Initiative (OCI) (Open Container Initiative) or blob artifacts. Difference between the two payload signature processes are described in the specification (see here).

While OCI signatures are embedded in the target image and associated to it in the registry, blob signatures are separated as explained here. Signing **OCI artifacts** is not in the scope of this audit.

In order to sign blobs, `notation blob` can be used as described in the specification (see here). However, at the time of this audit, the feature is not implemented by the Notary project.

4.3.2. Time Stamping Support

Source: https://notaryproject.dev/docs/user-guides/how-to/timestamping/

Scope:

- `notation-core-go/internal/timestamp/`
- `notation-core-go/x509/timestamp_cert_validations.go`
- `tspclient-go`
- `notation sign --timestamp-url <TSA_URL> --timestamp-root-cert <TSA_ROOT_CERT> <ref_to_artifact>`
- `notation inspect`

The goal is to obtain an **authentic timestamp**, considered as a **timestamp countersignature** that aims to guarantee that the signature was generated when the certificate was valid. In the absence of a such countersignature, a signature is considered invalid if the signing certificate or chain is either expired or revoked.

In order to verify such data in signatures, one has to create a certificate store of type `tsa` and update the property `trustStores` (refer to Section 4.2.3.).

It relies on `tspclient-go`.

4.3.3. Playing with `notation sign`

To get familiar with the tool and its different behaviors, we toyed a bit with JWS signatures and interaction with OCI registry using an Ubuntu docker image.

4.3.3.1. Sign process - registry

One that has write privileges on a container registry can produce a signature for an OCI artifact and push it leveraging the `notation`.

Info

Beforehand, an Ubuntu docker image was pushed to the deployed local container registry on `localhost:5001`.

An example of a `notation` usage could be the following: `notation sign "localhost:5001/ubuntu:latest"`

When `notation` has to sign an image, it:

1. Pushes a new blob layer to the registry for the target image with its associated content hash, containing the signature. The `content-type` is set to "application/octet-stream" and `media-type` (for JWS) to "application/jose+json".
2. Pushes an OCI manifest containing:
   - Reference to an empty `application/vnd.cncf.notary.signature` (that seems to stay empty forever, but since it is not in the direct scope of the audit we did not pursue more details about it). It seems to be used to indicate that a signature already exists or not;
   - Reference to a signature layer;
   - Subject field referring to the Docker manifest of our target image;
   - `io.cncf.notary.x509chain.thumbprint#S256` annotation containing the SHA-256 fingerprint.
3. Gather previous manifest index and push the updated one.
4. Delete previous manifest.

4.3.3.2. Registry interaction flow & content

The following section contains raw HTTP communications between `notation` and a local container registry. This was mainly done in order to get a proper understanding of the behavior of the tool against a pseudo-real world usage. As this is not directly in the scope, the different data flows are not explained nor detailed.

4.3.3.2.1. Check the existence of the manifest for the asked tag

Request:

```
HEAD /v2/ubuntu/manifests/latest HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0
Accept: application/vnd.docker.distribution.manifest.v2+json, application/
vnd.docker.distribution.manifest.list.v2+json, application/
vnd.oci.image.manifest.v1+json, application/vnd.oci.image.index.v1+json,
application/vnd.oci.artifact.manifest.v1+json
```

Answer:

```
HTTP/1.1 200 OK
Content-Length: 529
Content-Type: application/vnd.docker.distribution.manifest.v2+json
Docker-Content-Digest:
sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9
Docker-Distribution-Api-Version: registry/2.0
Etag: "sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9"
X-Content-Type-Options: nosniff
Date: Tue, 17 Sep 2024 17:40:47 GMT
```

Images tags are not used, the corresponding SHA-256 hash is gathered and we repeat the procedure:

Request:

```
HEAD /v2/ubuntu/manifests/
sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9
HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0
Accept: application/vnd.docker.distribution.manifest.v2+json, application/
vnd.docker.distribution.manifest.list.v2+json, application/
vnd.oci.image.manifest.v1+json, application/vnd.oci.image.index.v1+json,
application/vnd.oci.artifact.manifest.v1+json
```

Answer:

```
HTTP/1.1 200 OK
Content-Length: 529
Content-Type: application/vnd.docker.distribution.manifest.v2+json
Docker-Content-Digest:
sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9
Docker-Distribution-Api-Version: registry/2.0
Etag: "sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9"
X-Content-Type-Options: nosniff Date: Tue, 17 Sep 2024 17:40:47 GMT
```

4.3.3.2.2. New blob upload init

Request:

```
POST /v2/ubuntu/blobs/uploads/ HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0 Content-Length: 0 Accept-Encoding: gzip  ", 135) = 135
```

Answer:

```
HTTP/1.1 202 Accepted
```

```
Content-Length: 0
Docker-Distribution-Api-Version: registry/2.0
Docker-Upload-Uuid: 0dc1dfc0-c1c4-44f2-891c-1f146af1781d
Location: http://localhost:5001/v2/ubuntu/blobs/uploads/0dc1dfc0-c1c4-44f2-891c-1
f146af1781d?_state=QeiQutSjU-rXUWJLW51nIClJCsxHEHRkJBAmcnS7kwR7Ik5hbWUiOiJ1YnVudH
UiLCJVVUlEIjoiMGRjMWRmYzAtYzFjNC00NGYyLTg5MWMtMWYxNDZhZjE3ODFkIiwiT2Zmc2V0IjowLCJ
TdGFydGVkQXQiOiIyMDI0LTA5LTE3VDE3OjQwOjQ3LjI1MzU0NDExNFoifQ%3D%3D
Range: 0-0 X-
Content-Type-Options: nosniff Date: Tue, 17 Sep 2024 17:40:47 GMT
```

4.3.3.2.3. Push the signature blob (format JWS here)

Request:

```
PUT /v2/ubuntu/blobs/uploads/0dc1dfc0-c1c4-44f2-891c-1f146af1781d?
_state=QeiQutSjU-
rXUWJLW51nIClJCsxHEHRkJBAmcnS7kwR7Ik5hbWUiOiJ1YnVudHUiLCJVVUlEIjoiMGRjMWRmYzAtYzFj
HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0
Content-Length: 2095
Content-Type: application/octet-stream
Accept-Encoding: gzip
{"payload":"eyJ0YXJnZXRBcnRpZmFjdCI6eyJkaWdlc3QiOiJzaGEyNTY6MDRiNWFkYTRjZGI1MDM0YT
{"x5c":
["MIIDQTCCAimgAwIBAgICAJUwDQYJKoZIhvcNAQELBQAwTzELMAkGA1UEBhMCVVMxCzAJBgNVBAgTAldB
DydFNBDm/
iK+Dpoggx4b+599fRvK2aJ2DG5HKNyqrYajbb2JBSWqopVrObYqZIIPVoKz1ebtcwSrZr4UtNoW7lWheTq
b4W8VZd/r/5z9sRLVlKs/
SPmzk73YacHKZHUzDqfOXsT1xx7PSzou3XjNEbBuD3BNlFlDAyqAawoHDBIe92JVuRo+z1J/qKRt/
gcVYYuTM5M8CFZn2AJKL4bGNxWeYkxnE9wj3r479o4s1QgU1dhGbgyI0SAw0d5eV/
XcCsCAwEAAaMnMCUwDgYDVR0PAQH/
BAQDAgeAMBMGA1UdJQQMMAoGCCsGAQUFBwMDMA0GCSqGSIb3DQEBCwUAA4IBAQAxMtgh57J8BC2k7Pkni3
OsSLQh1As2fj0i0oInod5BHJWqyRG1NYJoqVRxkHW8ddJOEeo4QTr2s28FxFWeZ6ZGTOZjX4ClBc0Pp/
FFfc/N+T3qazt3tEm5JGAyleXPfoXJq/
s6HKptoYa47tjAGRBALzrlbN2wbzxF3q2URV5KaxKMO04gg81KEoUsrZn7eArD0oPGzBWSkOsPKwML4q+F
DTbxIGRxdkEibgGtPDLjOLBX3nj+sck0nPP2uGxXhEKqF9"],"io.cncf.notary.signingAgent":"no
go/1.2.0"},"signature":"EucQbRfC-
hQgZVGnyODxes9bIsGGMdOtxvmXFPbVR3iU0dt2zfp1jTXCXXMkBOTkwq0gy9IOCVd3iyujKpXwQAxEalX
PRNRV3JVeInNRTtHiuVNrg8cxArCk1Ke-Wy2Vr9aLH7RwIOp0AgtE_VRP_8TCtune8CigAlvUhN-
Zb2Y6xB-afDXKs-
rSkzNlFpdr6aa8M4V-174Nle2_ZUMXYSlNxR_MsoWiFXqalmhwe91g7PdGm8GUyFzCIq5fxmtxu3MCxk51
```

Answer:

```
HTTP/1.1 201 Created
Content-Length: 0
Docker-Content-Digest:
sha256:58069de41a1b482d8c375bd5d5e8b3d1fd0ff9a92c065691050c0baaff0f4b47
Docker-Distribution-Api-Version: registry/2.0
Location: http://localhost:5001/v2/ubuntu/blobs/sha256:58069de41a1b482d8c375bd5d5
e8b3d1fd0ff9a92c065691050c0baaff0f4b47
```

```
X-Content-Type-Options: nosniff
Date: Tue, 17 Sep 2024 17:40:47 GMT
```

4.3.3.2.4. Check if existing signature exists

Info

A GET request on `blobs/sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a` returns `{}` and `sha256sum` of `{}` indeed gives the following hash.

The layer seems to exist only to tell notary there is already a signature for the image but not for this specific tag.

If this layer does not exist, it is created right after that.

Request:

```
HEAD /v2/ubuntu/blobs/
sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a
HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0
```

Answer:

```
HTTP/1.1 200 OK
Accept-Ranges: bytes
Cache-Control: max-age=31536000
Content-Length: 2
Content-Type: application/octet-stream
Docker-Content-Digest:
sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a
Docker-Distribution-Api-Version: registry/2.0
Etag: "sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a"
X-Content-Type-Options: nosniff Date: Tue, 17 Sep 2024 17:40:47 GMT
```

4.3.3.2.5. Push image manifest for signature

Note: We usually push manifest for a dedicated new tag (`/manifests/<new_tag>`), whereas here it is using the hash value of the manifests itself.

Request:

```
PUT /v2/ubuntu/manifests/
sha256:f401f5cdea4782008d231f29839f111609295d175a8b2e083feb41de752b19c1
HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0
Content-Length: 738
Content-Type: application/vnd.oci.image.manifest.v1+json
Accept-Encoding: gzip
```

```
 {"schemaVersion":2,
 "mediaType":"application/vnd.oci.image.manifest.v1+json",
 "config":
    {"mediaType":"application/vnd.cncf.notary.signature",
"digest":"sha256:44136fa355b3678a1146ad16f7e8649e94fb4fc21fe77e8310c060f61caaff8a
  "size":2},
    "layers":[
    {"mediaType":"application/jose+json",
"digest":"sha256:58069de41a1b482d8c375bd5d5e8b3d1fd0ff9a92c065691050c0baaff0f4b47
    "size":2095
    }
    ],
    "subject":
{"mediaType":"application/vnd.docker.distribution.manifest.v2+json",
"digest":"sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9
 "size":529},
 "annotations":{
"io.cncf.notary.x509chain.thumbprint#S256":"[\"9afd9f36f450beff81331e5a78685a1ba8
 "org.opencontainers.image.created":"2024-09-17T17:40:47Z"
 }
 }
```

Answer:

```
HTTP/1.1 201 Created
Docker-Content-Digest:
sha256:f401f5cdea4782008d231f29839f111609295d175a8b2e083feb41de752b19c1
Docker-Distribution-Api-Version: registry/2.0
Location: http://localhost:5001/v2/ubuntu/manifests/sha256:f401f5cdea4782008d231f
29839f111609295d175a8b2e083feb41de752b19c1
X-Content-Type-Options: nosniff
Date: Tue, 17 Sep 2024 17:40:47 GMT
Content-Length: 0
```

4.3.3.2.6. Get OCI manifests index for the concerned image

Note: If this layer doesn't exist, it is created and last step (deletion step) is not executed.

Request:

```
GET /v2/ubuntu/manifests/
sha256-04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9
HTTP/1.1
Host: localhost:5001
User-Agent: notation/1.2.0
Accept: application/vnd.docker.distribution.manifest.v2+json, application/
vnd.docker.distribution.manifest.list.v2+json, application/
vnd.oci.image.manifest.v1+json, application/vnd.oci.image.index.v1+json,
application/vnd.oci.artifact.manifest.v1+json
Accept-Encoding: gzip
```

Answer:

```
HTTP/1.1 200 OK
Content-Length: 883
Content-Type: application/vnd.oci.image.index.v1+json
Docker-Content-Digest:
sha256:96569f73e32cf3907973604bde69b7a5d926ec1dcab2533ff97e04e8c01ab64d
Docker-Distribution-Api-Version: registry/2.0
Etag: "sha256:96569f73e32cf3907973604bde69b7a5d926ec1dcab2533ff97e04e8c01ab64d"
X-Content-Type-Options: nosniff
Date: Tue, 17 Sep 2024 17:40:47 GMT
{"schemaVersion":2,"mediaType":"application/
vnd.oci.image.index.v1+json","manifests":[{"mediaType":"application/
vnd.oci.image.manifest.v1+json","digest":"sha256:f3c79a9b3f2ac3e8e773207f29c0935ae
{"io.cncf.notary.x509chain.thumbprint#S256":"["cfbc6a1df39a6dc3c820064abb826677851
vnd.cncf.notary.signature"},{"mediaType":"application/
vnd.oci.image.manifest.v1+json","digest":"sha256:587be6b687345d519aea0900f0e5bd5fd
{"io.cncf.notary.x509chain.thumbprint#S256":"["cfbc6a1df39a6dc3c820064abb826677851
vnd.cncf.notary.signature"}]}
```

4.3.3.2.7. Push updated OCI manifest index with previous ones

It is appending:

```
 {
   "mediaType":"application/vnd.oci.image.manifest.v1+json",

 "digest":"sha256:f401f5cdea4782008d231f29839f111609295d175a8b2e083feb41de752b19c1
    "size":738,
    "annotations":{

 "io.cncf.notary.x509chain.thumbprint#S256":" [\"9afd9f36f450beff81331e5a78685a1ba
      "org.opencontainers.image.created":"2024-09-17T17:40:47Z"
      },
    "artifactType":"application/vnd.cncf.notary.signature"
 }
```

Request:

```
PUT /v2/ubuntu/manifests/
sha256-04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9 HTTP/1.1
Host: localhost:5001 User-Agent: notation/1.2.0 Content-Length: 1281 Content-
Type: application/vnd.oci.image.index.v1+json Accept-Encoding: gzip
{"schemaVersion":2,"mediaType":"application/
vnd.oci.image.index.v1+json","manifests":[{"mediaType":"application/
vnd.oci.image.manifest.v1+json","digest":"sha256:f3c79a9b3f2ac3e8e773207f29c0935aef
{"io.cncf.notary.x509chain.thumbprint#S256":"["cfbc6a1df39a6dc3c820064abb826677851c
vnd.cncf.notary.signature"},{"mediaType":"application/
vnd.oci.image.manifest.v1+json","digest":"sha256:587be6b687345d519aea0900f0e5bd5fdd
{"io.cncf.notary.x509chain.thumbprint#S256":["cfbc6a1df39a6dc3c820064abb826677851c
vnd.cncf.notary.signature"},{"mediaType":"application/
vnd.oci.image.manifest.v1+json","digest":"sha256:f401f5cdea4782008d231f29839f111609
{"io.cncf.notary.x509chain.thumbprint#S256":["9afd9f36f450beff81331e5a78685a1ba831
vnd.cncf.notary.signature"}]}
```

Answer:

```
HTTP/1.1 201 Created Docker-Content-Digest:
sha256:602d9b8f51fe06e8c170a934061a50437f5f359331b0fb6d065e9665412830fe Docker-
Distribution-Api-Version: registry/2.0 Location: http://localhost:5001/v2/ubuntu/
manifests/sha256:602d9b8f51fe06e8c170a934061a50437f5f359331b0fb6d065e9665412830fe
X-Content-Type-Options: nosniff Date: Tue, 17 Sep 2024 17:40:47 GMT Content-
Length: 0  ", 4096) = 381
```

4.3.3.2.8. Delete previous manifest for our target image

Request:

```
DELETE /v2/ubuntu/manifests/
sha256:96569f73e32cf3907973604bde69b7a5d926ec1dcab2533ff97e04e8c01ab64d HTTP/1.1
Host: localhost:5001 User-Agent: notation/1.2.0 Accept-Encoding: gzip  ", 185) =
185
HTTP/1.1 202 Accepted Docker-Distribution-Api-Version: registry/2.0 X-Content-
Type-Options: nosniff Date: Tue, 17 Sep 2024 17:40:47 GMT Content-Length: 0  ",
4096) = 161
```

4.4. Source Discovery

4.4.1. Code Structure

The source material is splitted into 4 main repositories. Each of those repositories implement different features, based on the respective RFCs. We detailed the different folders and RFC that are implemented by each one in Figure 1.

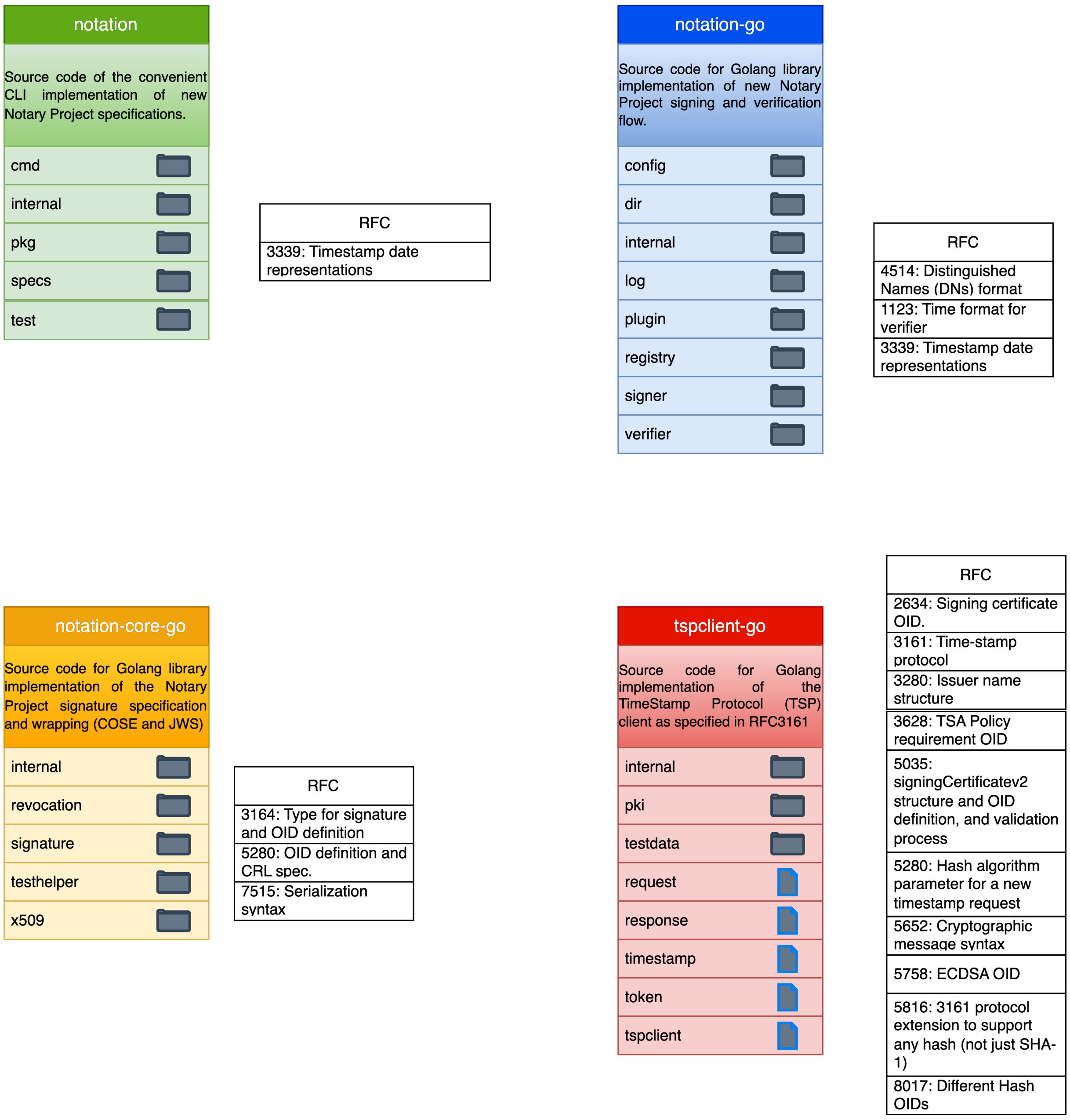

notation
Source code of the convenient CLI implementation of new Notary Project specifications.
cmd
internal
pkg
specs
test
RFC
3339: Timestamp date representations
notation-go
Source code for Golang library implementation of new Notary Project signing and verification flow.
config
dir
internal
log
plugin
registry
signer
verifier
RFC
4514: Distinguished Names (DNs) format
1123: Time format for verifier
3339: Timestamp date representations
notation-core-go
Source code for Golang library implementation of the Notary Project signature specification and wrapping (COSE and JWS)
internal
revocation
signature
testhelper
x509
RFC
3164: Type for signature and OID definition
5280: OID definition and CRL spec.
7515: Serialization syntax
tspclient-go
Source code for Golang implementation of the TimeStamp Protocol (TSP) client as specified in RFC3161
internal
pki
testdata
request
response
timestamp
token
tspclient
RFC
2634: Signing certificate OID.
3161: Time-stamp protocol
3280: Issuer name structure
3628: TSA Policy requirement OID
5035: signingCertificatev2 structure and OID definition, and validation process
5280: Hash algorithm parameter for a new timestamp request
5652: Cryptographic message syntax
5758: ECDSA OID
5816: 3161 protocol extension to support any hash (not just SHA-1)
8017: Different Hash OIDs

Figure 1: *Notary project cartography*

4.4.2. Code Quality

The code quality is overall good. We did not find any dependencies that were not up-to-date or subject to security issues, the code is well written and commented, and regularly maintained.

On top of that, every major new set of features is associated with a security audit and a fuzzing campaign, which shows a high level of maturity in the practice of security.

The only issue is that numerous instances of redundant code were found and are highlighted during the analysis sections. Additionally, Quarkslab's engineers used an open source tool to display the different duplicated code that could be found. The output is provided in Appendix A.

5. Time-Stamp Protocol Compliance

Warning

Please note that the compliance checks were made on an earlier version of the code as specified in Section 4.1.1. . This means that the code lines given in this section for the compliance checks are valid for those commit hashes only. In fact, the new code release (for instance of the CRL checks) induced a shift in those line numbers.

5.1. Description

The Time-Stamp Protocol, as described in RFC3161, is a protocol to receive a cryptographic time-stamp from a trusted third party, called the TSA. The time-stamp acts as proof that a certificate, even after its expiration date, existed and was valid at a particular time, usually its validity period.

The protocol is based on a request-response communication, where the request is formed following a specific format defined in RFC3161. The important field are the message imprint, the nonce used to thwart replay attacks, and the policy. Upon reception, the TSA creates a Time-Stamp Token (TST) containing the message imprint, a unique serial number, the time-stamp and a digital signature using the TSA private key. One can then verify the validity of the TST using the TSA's root certificate to verity the signature and the TST different fields.

Notary implementation of the protocol is done over HTTP only (the RFC also includes a way to operate it over TCP sockets or emails).

Note that one of the main features of the time-stamp relies on the validity of a trusted local time on the machine on which a user operates. In order to avoid replay attacks, the RFC includes the description of the use of a nonce to thwart such attacks. This nonce should be only used and checked in a setting where the user cannot trust its local time, according to the RFC. Notary chooses to always use such nonce instead of verifying if a trusted local time is accessible.

We detailed hereafter the conformity of the source material to the RFC for the different building blocks of the protocols and the numerous types involved.

5.2. Request Format

```
TimeStampReq ::= SEQUENCE  {
   version                INTEGER  { v1(1) },
   messageImprint        MessageImprint, --a hash algorithm OID and the hash value
of the data to be time-stamped
   reqPolicy              TSAPolicyId              OPTIONAL,
   nonce                  INTEGER                  OPTIONAL,
   certReq                BOOLEAN                  DEFAULT FALSE,
   extensions             [0] IMPLICIT Extensions  OPTIONAL
}
```

request.go:59-75

Defined in a structure called `Request` containing all the appropriate fields.

Message format:

```
MessageImprint ::= SEQUENCE  {
        hashAlgorithm                AlgorithmIdentifier,
        hashedMessage                OCTET STRING
}
```

request.go:31-39

Defined in a structure called MessageImprint containing all the appropriate fields.

Note that the request should not identify the requester. If the TSA requires the identity of the requester, identification/authentication means have to be used such as CMS encapsulation or TLS authentication.

5.3. Response Format

```
TimeStampResp ::= SEQUENCE  {
      status                  PKIStatusInfo,
      timeStampToken          TimeStampToken     OPTIONAL
}
```

response.go:103-111

Defined in a structure called `Response` containing all the appropriate fields.

5.3.1. Status Format

PKIStatusInfo format, following RFC2510:

```
PKIStatusInfo ::= SEQUENCE {
      status         PKIStatus,
      statusString   PKIFreeText     OPTIONAL,
      failInfo       PKIFailureInfo  OPTIONAL
}
```

pki.go:113-129

Defined in a structure called `StatusInfo` containing all the appropriate fields.

The different status values are the following:

- granted: 0
- granted with mods: 1
- rejection: 2
- waiting: 3
- revocation warning: 4
- revocation notification: 5

Client must generate an error if status values are not in that list.

pki.go:31-61

Type properly defined with all the different possible status and the string conversion function handle the error for unknown status.

The different reasons for failing (also known as failInfo values) are the following:

- bad algorithm: 0
- bad request: 2
- bad data format: 5
- time not available (for TSA): 14
- unaccepted policy: 15

- unaccepted extension: 16
- additional information not available: 17
- system failure: 25

Client must generate an error if status values are not in that list.

pki.go:77-111

Type properly defined with all the different possible status and the error conversion function handle the error for unknown failure info.

5.3.2. Time-stamp Token Format

```
TimeStampToken ::= ContentInfo
     -- contentType is id-signedData ([CMS])
     -- content is SignedData ([CMS])
```

The content is composed of the following fields:

- eContentType: OID for a time-stamp token (1.2.840.113549.1.9.16.1.4)
- eContent: DER-encoded value of TSTInfo
- SigningCertificate:
  - ESSCertID as signerInfos: certificate identifier of the TSA certificate
- Signature: signature of the TSA

The token should not contain any signatures other than the TSA signature.

cms.go

Generic type such as ContentInfo, SignedData, etc, are properly defined there according to their corresponding RFC (RFC 5652).

TSTInfo format:

```
TSTInfo ::= SEQUENCE  {
   version                      INTEGER  { v1(1) },
   policy                       TSAPolicyId,
   messageImprint               MessageImprint,
     -- MUST have the same value as the similar field in
     -- TimeStampReq
   serialNumber                 INTEGER,
    -- Time-Stamping users MUST be ready to accommodate integers
    -- up to 160 bits.
   genTime                      GeneralizedTime,
   accuracy                     Accuracy                 OPTIONAL,
   ordering                     BOOLEAN             DEFAULT FALSE,
   nonce                        INTEGER                  OPTIONAL,
     -- MUST be present if the similar field was present
     -- in TimeStampReq.  In that case it MUST have the same value.
   tsa                          [0] GeneralName          OPTIONAL,
   extensions                   [1] IMPLICIT Extensions   OPTIONAL
}
```

Among the optional fields, only the nonce field must be supported.

token.go:193-215

Defined in a structure called `TSTInfo` containing all the appropriate fields.

5.4. Response Verification

5.4.1. TST Reception Checks

Upon the reception of the token, the client will perform a certain number of checks to validate the content of the TST, the TSA signature, and other different checks to assess the validity and soundness of the produced time-stamp. The different verification steps are the following:

1. Verify status error.
2. Verify various fields contained in the Time-Stamp Protocol (TSP).
3. Verify validity of the signature of the TSP.
4. Verify that what was timestamped is what was asked to be timestamped.
5. Verify that the token contains the correct: certificate identifier, data imprint, hash algorithm OID.
6. a. Verify the time included in the response against local trusted time.

   b. (if no local trusted time) Verify the nonce against the value included in the request.
7. Check the status of the TSA's certificate (not expired).
8. Check the policy field for compliance with the application's need.

TST Reception Checks

1. 154-156 by calling the `validateStatus` function, which calls the Err() properly defined as stated in the format section.
2. - Version : `tspclient-go/response.go:165` + `tspclient-go/token.go:250`
   - Content-Type : `tspclient-go/token.go:47`
   - Certificates for presence and well-formed: `tspclient-go/internal/cms/signed.go:76`
3. - `tspclient-go/internal/cms/signed.go:201` (signature by itself)
   - `tspclient-go/internal/cms/signed.go:203` (check that hash sum of signed content in SignedAttributes equals to hash sum of the TSTInfo)
   - `tspclient-go/internal/timestamp/timestamp.go:54` (check issuance of the certification chain + validation of leaf certificate and root certificate according to their spec)
4. `tspclient-go/response.go:173`
5. - correct certificate identifier of the TSA : `tspclient-go/internal/cms/signed.go:195`
   - correct data imprint : `tspclient-go/token.go:262`
   - correct hash algorithm OID : `tspclient-go/token.go:254`
6. - Check that timezone is UTC: `tspclient-go/response.go:178`
   - Nonce : `tspclient-go/response.go:183`
7. `tspclient-go/internal/cms/signed.go:l295` when calling isSigningTimeValid
8. `tspclient-go/response.go:169`

Please note that the client never verifies that the local time is coming from trusted sources. The tspclient's protocol always uses the option with a unique nonce to avoid such test and guarantee replay attack protection.

However, we noticed that in the source material (and only there, this part of the code is not reachable by the CLI as far as we know), the Notary project implemented two features that could be problematic in the current state of the code.

The first one is a flag called `NoNonce`, which disables the use of a Nonce in the request. Since the local time trust is never checked, this could be problematic in case the flag is set to `True`. Fortunately, this flag is never checked nor used in practice.

INFO	**QB-1** Unused risky flag `NoNonce`
Perimeter	Time-stamping request
Description	
Definition of flag that could lead to potential security issues and which is never used in practice.	
Recommendation	
Remove this field since it does not bring anything relevant to the protocol	

The second one is the value of the `Nonce`, which can be set by the user in the request. If empty, the client uses a proper random generator to pick a unique nonce value. On top of that, the comment describing the `Nonce` can be misinterpreted as it states that the `Nonce` is a number that the client generates once, whereas it should be generated once per request. Fortunately again, there is currently no possible ways to provide a `Nonce` to the CLI in practice.

```go
if !opts.NoNonce {
  if opts.Nonce != nil { // user provided Nonce, use it
    nonce = opts.Nonce
  } else { // user ignored Nonce, use built-in Nonce
    var err error
    nonce, err = generateNonce()
    if err != nil {
      return nil, &MalformedRequestError{Msg: err.Error()}
    }
  }
}
```

We believe these two features were mostly introduced by the developers for testing and debugging purposes and should be removed from the code to avoid any misbehavior or misuse of them, leading to potential security issues, such being vulnerable to replay attacks when the request is made on an environment where the local time cannot be trusted.

INFO	**QB-2** Option for user to choose the `Nonce`
Perimeter	Time-stamping request
Description	
There is an option in the code that states that the user could choose the Nonce to be used in the request. If not properly chosen, this could lead to potential issue as the nonce needs to be unique for each request.	
Recommendation	
Remove the possibility to use a user-provided `Nonce` (if statement of the code snippet).	

Info

The associated pull request for these two issues is: https://github.com/notaryproject/tspclient-go/pull/34

5.4.2. TST Fields Checks

Requesters must be able to recognize all optional fields present but are not mandated to understand the semantics of any extension, if present.

- Version: TSA must be able to provide version 1 time-stamp tokens.
- Policy: TSA's policy under which the response was produced. If present in the request, it must match. The policy contains two information:
  - Time-stamp usage condition.
  - Availability of a time-stamp token log for later verification of authenticity.
- Message Imprint: verify that the length corresponds to the hash function used and that it is the same value as in the request.
- Serial Number: Unique number (w.r.t. to TSA) for the TSP.
- genTime: Time at which the token has been created in UTC time (RFC 2459 should be followed).
- accuracy: Time deviation around the UTC time. Allows to create an upper- and lower-bound at which the token has been created. If set to false or missing, only take genTime into account.
- nonce: must be present if it was in the request and equal to it.
- TSA: name of the TSA.

Valid TST Fields Checks

- Version: `tspclient-go/response.go:165` and `tspclient-go/token.go:250`
- Policy:
  - `tspclient-go/response.go:169`: Verifies that policy in request matches with the one in the response;
  - `tspclient-go/token.go:230`: Compares policy to `oid.BaselineTimestampPolicy` in order to know how to set `accuracy`;
  - Policy information are not handled;
- Message Imprint:
  - `tspclient-go/response.go:173`: Verifies that the Message Imprint of the request is the same as in the response;
- Serial Number: Present in `TSTInfo` structure but never mentioned;
- genTime: `tspclient-go/response:179`
- accuracy:
  - `tspclient-go/timestamp.go:36` & `tspclient-go/timestamp.go:39`: Lower and upper bound created out of `genTime` and `accuracy`;
  - `tspclient-go/token.go:238`: Time-stamp structure is created from `genTime` and `accuracy`.
- nonce: `tspclient-go/response.go:183`
- TSA: Field TSA is present in `TSTInfo` structure but never mentioned.

Erroneous TST Fields Checks

- Message Imprint:
  - `tspclient-go/request:198`: Verifies that the Message Imprint length corresponds to the hash function used. However it is done during request validation. Not for the response.

5.4.3. Signature Verification

To verify the TSA's signature of the time-stamp, the following routine is used:

1. Time-stamping information needs to be obtained soon after the signature has been produced (e.g. within a few minutes or hours).
   1. The signature is presented to the Time Stamping Authority (TSA). The TSA then returns a TimeStampToken (TST) upon that signature.
   2. The invoker of the service MUST then verify that the TimeStampToken is correct.
2. The validity of the digital signature may then be verified in the following way:
   1. The time-stamp token itself MUST be verified and it MUST be verified that it applies to the signature of the signer.
   2. The date/time indicated by the TSA in the TimeStampToken MUST be retrieved.
   3. The certificate used by the signer MUST be identified and retrieved.
   4. The date/time indicated by the TSA MUST be within the validity period of the signer's certificate.

5. The revocation information about that certificate, at the date/time of the Time-Stamping operation, MUST be retrieved.
6. Should the certificate be revoked, then the date/time of revocation shall be later than the date/time indicated by the TSA.

If all these conditions are successful, then the digital signature is valid.

Signature verification

- Time-stamping is part of the signing process and is performed right after the signature to be timestamped has been generated
  - `notation-core-go/signature/jws/enveloppe.go:97` or `notation-core-go/signature/cose/enveloppe.go:252`;
- `tspclient-go/request:128`: Signature digest is embedded in `Request` structure;
- `tspclient-go/http.go:86`: Signature is present to the TSA
- `tspclient-go/http.go:117`: Verifies that the received data is a TimeStampToken and is correct. (Integrity and Authenticity not verified)

Validity of the digital signature

- The time-stamp token itself MUST be verified and it MUST be verified that it applies to the signature of the signer.
  - `tspclient-go/token.go:246`: It applies to the signature of the signer;
  - `tspclient-go/internal/cms/signed.go:214`: Time-stamp token signature is verified.
- `tspclient-go/token.go:177`: The date/time indicated by the TSA in the TimeStamp-Token MUST be retrieved.
- `tspclient-go/token.go:76`: The certificate used by the signer MUST be identified and retrieved.
- `notation-go@1.3.0-rc.1/verifier/verifier.go:938`: The date/time indicated by the TSA MUST be within the validity period of the signer's certificate.
- `notation-go@v1.3.0-rc.1/verifier/verifier.go:697`: The revocation information about that certificate, at the date/time of the Time-Stamping operation, MUST be retrieved.

Validity of the digital signature

- `notation-go@v1.3.0-rc.1`: Should the certificate be revoked, then the date/time of revocation shall be later than the date/time indicated by the TSA
  - The date/time indicated by the TSA is not passed to the revocation check method.

5.5. TSP via HTTP

Requests are made with ASN.1-encoded messages. Two MIME objects are specified as follows.

Content-Type: application/timestamp-query

```
<<the ASN.1 DER-encoded Time-Stamp Request message>>
```

Content-Type: application/timestamp-reply

```
<<the ASN.1 DER-encoded Time-Stamp Response message>>
```

6. Time Stamp Analysis in `Notation`

As detailed in previous sections, `Notation v1.2.0` implements the Time-Stamp Protocol, following RFC3161 standard, which allows to add a time-stamp composed of contents proving the validity of a certificate at a given date and the signature authenticating those data, called a *countersignature*. Issued by a TSA, the time-stamp allows to extend the trust of signatures created within certificates' validity even after they are expired. In fact, if one certificate of the trusted certificate chain has expired, the time-stamp can be used to verify that the signature was issued when the certificate chain was still valid.

In this section, we describe the test performed playing with the time-stamping feature and the corresponding findings discovered while testing.

The tests of this section were performed using :
- `Fedora Linux 38 aarch64` as operating system;
- `Docker v24` as container engine;
- `docker.io/library/registry@sha256:ac0192b549007e22998eb74e8d8488dcfe70f1489520c3b144a6047ac5efbe90` image as a Docker Registry, listening on `5001` TCP port on `localhost`;
- DigiCert as TSA.

Note: Since the OCI signatures of `notation` are out of the scope of the audit and to make reading easier, related configurations and corresponding control-flows are ignored when not relevant with respect to the time-stamping functionality.

In the following, we use diagrams to illustrate how the code processes with respect to the test we do. The block represents Golang methods, formatted using the following nomenclature: `<Go STD package>/package.method name`. If the method comes from Golang standard library, it is written preceded by a slash. Content of such methods are not part of this audit, however, how they are used in practice in `notation` is.

The overall interaction when requesting a time-stamp is described in Figure 2.

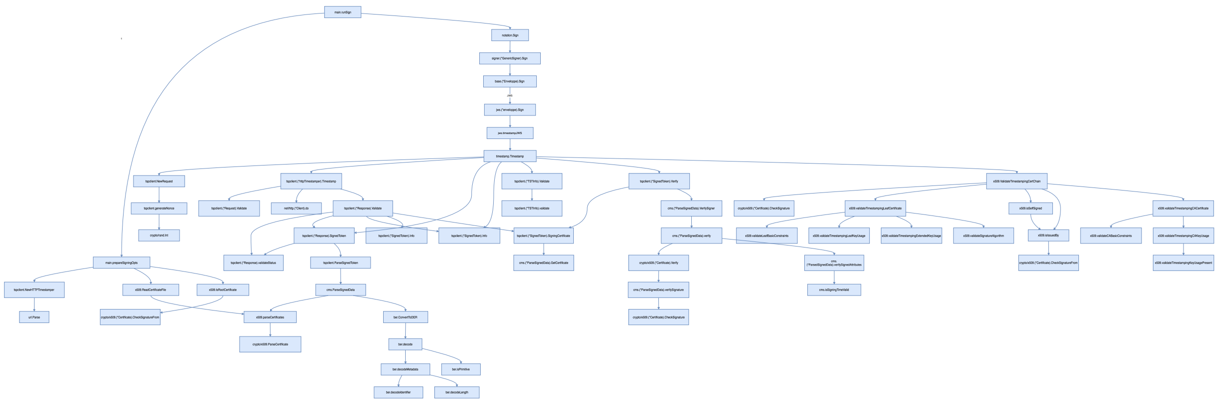


Figure 2: *Complete view of time-stamp control flow graph*[1]

[1]Full page scaled picture can be found in Appendix C.A.

6.1. Signing with a Countersignature

In order to add a countersignature to the signature envelope of an artifact, one can use the `sign` functionality of the `notation` CLI and, in addition to the target, specify:

- The time-stamp server URL of the TSA (`--timestamp-url`);
- The path to the root certificate of the specified TSA (`--timestamp-root-certificate`).

```
notation sign  --timestamp-url "http://timestamp.digicert.com" --timestamp-root-
cert "DigiCertTrustedRootG4.crt" "localhost:5001/ubuntu:latest"
```

6.1.1. Initialization phase

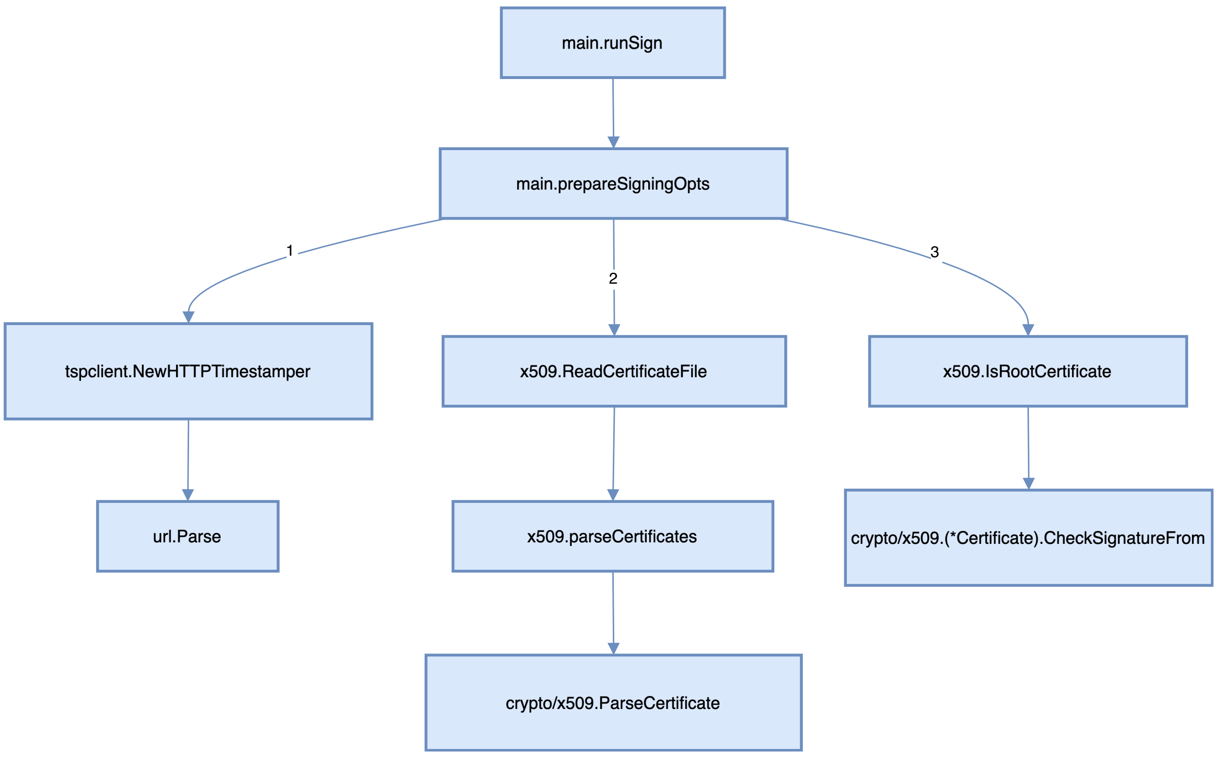


Figure 3: *Time-stamp initialisation phase control flow graph*

The function `main.runSign` from `cmd/notation/sign.go` is in charge of the signing functionality of the `notation` project.

The raw arguments from the user inputs are sent to `main.prepareSigningOpts` in order to get the signing options. If the field `tsaServerURL`, corresponding to the CLI option `--timestamp-url` is not empty, then all the required configurations to request a TST are checked and added to the signing options.

Following Figure 3 illustration, the `main.runSign` runs the following functions and checks:

1. Parsing of TSA' URL using Go's `net/url.Parse` as follows:

```
if _, err := url.Parse(endpoint); err != nil {
  return nil, err
}
```

We noticed at this step a confusing check around the URL parsing. In fact, there are checks that the URL is properly parsed, but not of its contents. This means that empty URL or schema would lead to a proper parsing result (when it should not).

Note that from a security point of view, this is not an issue as the validity of the schema or URL is checked later on in the process, when a request is sent. However this could save some time and a HTTP request.

INFO	**QB-3** Lack of check after URL parsing
Perimeter	CLI signing with time-stamping

Description

No check after parsing the TSA's URL of content of the host or the scheme. This means that an empty host or schema is accepted and processed by the CLI. While this does not directly lead to an issue, since those fields are checked later on by third-party code, this, however, requires that this outside check stays.

Recommendation

Add a test after parsing the URL, such as the following one:

```
if (u.Scheme == ""  || u.Host == "") { // where u is the return value of url.Parse
  // return error
}
```

2. Reading and parsing of the submitted root certificate encoded either DER or PEM using `crypto/x509.parseCertificates` function.
3. Verification that only one certificate is returned and that the returned certificate is a root certificate by calling `x509.IsRootCertificate`. This function checks the certificate's signature using `crypto/x509.CheckSignatureFrom` and verifies that the `RawSubject` is equal to the `RawIssuer`.

While it does not seem possible to specify a malformed certificate or anything different from a valid root certificate to the CLI, the size of the keys is not verified. This can be concerning for cases using RSA as the signature scheme as keys smaller than 2048 are considered insecure. The validity of the certificate is not checked either.

Note: From a security point of view, this is not an issue as the validity of the whole certificate chain is verified at the very end of the process (but not by `notation` *code). However this could save some time and a HTTP request.*

Info

The associated pull request for this issue is: https://github.com/notaryproject/tspclient-go/pull/37

6.1.2. TST Request and Validation

Depending on the requested signature format, the created envelope (a structure used to create and verify signature) is either handled by the `JWS` or `COSE` package, by calling the method `(e *envelope) Sign(req *signature.SignRequest)` from either `notation-core-go/signature/jws` or from `notation-core-go/signature/cose`.
Using different techniques, both packages build a `tspclient.RequestOptions` containing the raw signature content as well as the hash method used. The overall process is illustrated in Figure 4 (using JWS).

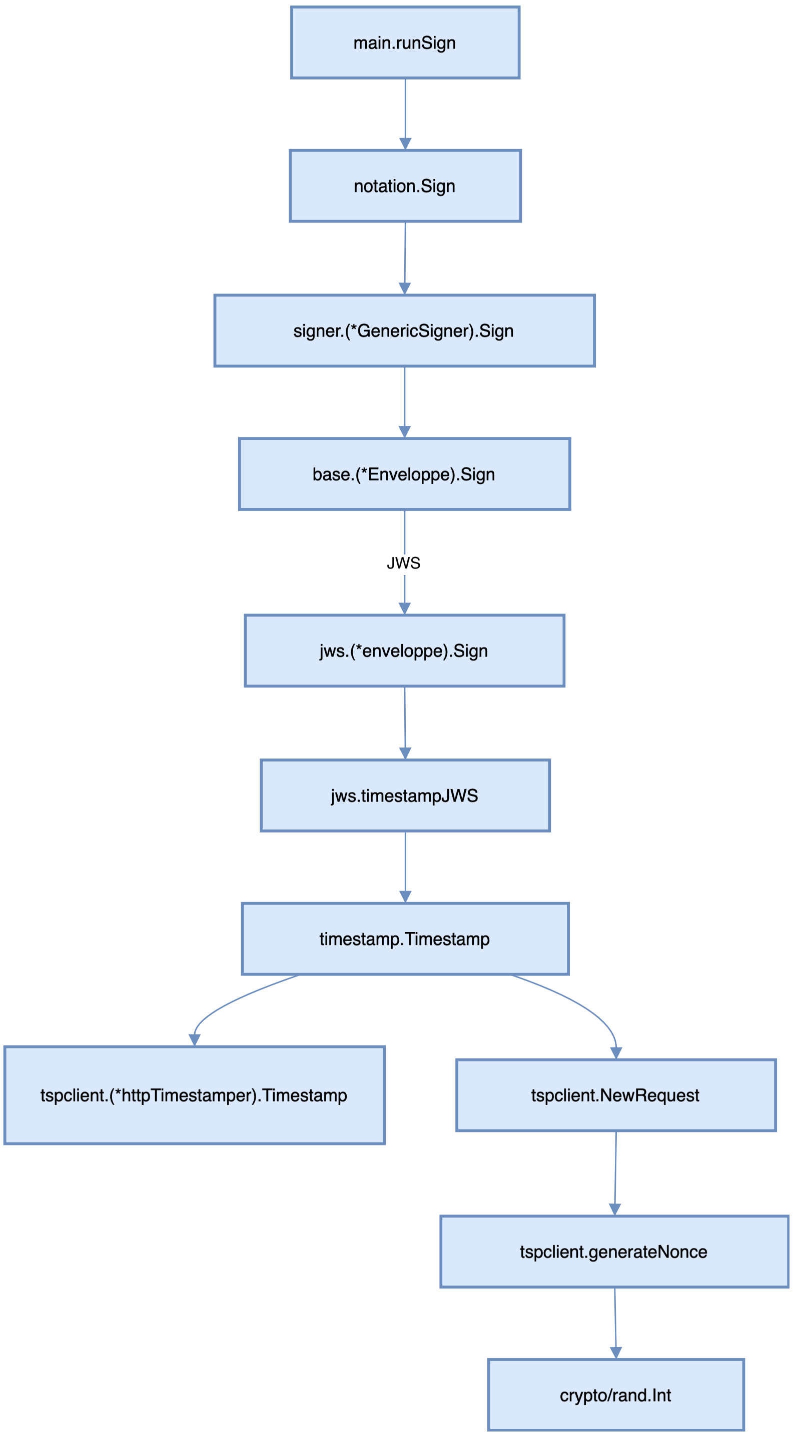


Figure 4: *Time-stamping request validation control flow graph*

The method `func Timestamp(req *signature.SignRequest, opts tspclient.RequestOptions)` from `notation-core-go/internal/timestamp` is then called, containing the `SignRequest` structure as well as the `RequestOptions`. The whole time-stamping functionality logic is handled by this method.

The first argument, a `tspclient-go.Timestamper` structure contains a `net/http.Client` client and the URL of the TSA, and a `crypto/x509.CertPool` structure built out of the previously submitted root certificate.

The method is defined as follows:

```go
func Timestamp(req *signature.SignRequest, opts tspclient.RequestOptions)
([]byte, error) {
  // Build the TSP request
  tsaRequest, err := tspclient.NewRequest(opts)
  if err != nil {
    return nil, err
  }
  ctx := req.Context()
  // Send the TSP request, read and validate TSP Response
  resp, err := req.Timestamper.Timestamp(ctx, tsaRequest)
  if err != nil {
    return nil, err
  }
  // Parse the SignedToken (CMS SignedData) part of the TimeStampToken
  token, err := resp.SignedToken()
  if err != nil {
    return nil, err
  }
  // Extract the TSTInfo ( CMS SignedData.EncapsulatedContentInfo)
  info, err := token.Info()
  if err != nil {
    return nil, err
  }
  // Validation of the TSTInfo content
  timestamp, err := info.Validate(opts.Content)
  if err != nil {
    return nil, err
  }
  // Build certification chain and verify signatures
  tsaCertChain, err := token.Verify(ctx, x509.VerifyOptions{
    CurrentTime: timestamp.Value,
    Roots:       req.TSARootCAs,
  })
  if err != nil {
    return nil, err
  }
  // Verify validity and compliances of certificates from the certificate chain
  if err := nx509.ValidateTimestampingCertChain(tsaCertChain); err != nil {
    return nil, err
  }
  return resp.TimestampToken.FullBytes, nil
}
```

6.1.3. Building TSP request

`tspclient.NewRequest` is called in order to compute a new TSP request `Request` defined as follows:

```go
type Request struct {
  Version        int // fixed to 1 as defined in RFC 3161 2.4.1 Request Format
  MessageImprint MessageImprint
  ReqPolicy      asn1.ObjectIdentifier `asn1:"optional"`
  Nonce          *big.Int              `asn1:"optional"`
  CertReq        bool                  `asn1:"optional,default:false"`
  Extensions     []pkix.Extension      `asn1:"optional,tag:0"`
}
type MessageImprint struct {
  HashAlgorithm pkix.AlgorithmIdentifier
  HashedMessage []byte
}
// pkix.AlgorithmIdentifier
type AlgorithmIdentifier struct {
  Algorithm  asn1.ObjectIdentifier
  Parameters asn1.RawValue `asn1:"optional"`
}
```

While the code below may suggest that the `Nonce` can be empty or user submitted, it has been found that no such option in the `notation` CLI allows to set them, or anywhere else. It is therefore always included, and generated thanks to Go's cryptographically secure pseudo-random number generator (see Section 5.4.1.).

```go
  var nonce *big.Int
  if !opts.NoNonce {
    if opts.Nonce != nil { // user provided Nonce, use it
      nonce = opts.Nonce
    } else { // user ignored Nonce, use built-in Nonce
      var err error
      nonce, err = generateNonce()
      if err != nil {
        return nil, &MalformedRequestError{Msg: err.Error()}
      }
    }
  }
```

The CertReq field is always set to True, while the remaining fields are always empty. The returned structure is the following:

```go
  return &Request{
    Version: 1,
    MessageImprint: MessageImprint{
      HashAlgorithm: pkix.AlgorithmIdentifier{
        Algorithm:  hashAlg, // OID of hash algorithm
        Parameters: hashAlgParameter, // Empty
      },
      HashedMessage: digest,
    },
    ReqPolicy:  opts.ReqPolicy, // Empty
    Nonce:      nonce, // 20 random bytes
    CertReq:    !opts.NoCert, // True
    Extensions: opts.Extensions, // Empty
  }, nil

```

6.1.4. TSP HTTP Request

The method (ts *httpTimestamper) Timestamp(ctx context.Context, req *Request) from tspclient.Timestamp validates the content of the Request req, sends a POST request to the TSA, reads its response and validates it.

The validation of the request consists in verifying that the Request.Version is set to 1 and if there is no mistakes related to the hash algorithm OID and the length of the hashed message.

The HTTP POST request is performed and the response content is read, up to 1 MB, if the HTTP Status is 200 and if the Content-Type header is set to application/timestamp-reply.

Success

No issue was found during this part.

6.1.5. TSP HTTP request response validation

The method func (r *Response) Validate(req *Request) from tspclient.Response verifies that the response is valid regarding the corresponding request and RFC3161.

The Response structure is defined as follows:

```go
type Response struct {
  Status         pki.StatusInfo
  TimestampToken asn1.RawValue `asn1:"optional"`
}
// pki.StatusInfo
type StatusInfo struct {
  Status       Status
  StatusString []string       `asn1:"optional,utf8"`
```

```go
    FailInfo     asn1.BitString `asn1:"optional"`
}
```

It is verified that:

- The response is not empty;
- The status `Status` is either `StatusGranted` (0) or `StatusGrantedWithMods` (1).

The `SignedData` (`SignedToken`) part of the CMS envelope is extracted by calling `func (r *Response) SignedToken` from `tspclient.Response`. The `TimestampToken` value is sent to `tspclient.ParseSignedToken` which sends it to `cms.ParseSignedData`.

The content is converted from ASN.1 BER encoding to DER thanks to the `tspclient-go/internal/encoding/ber` package. It is verified that the content is a CMS formatted package containing a `SignedData` version 3 envelope.

Success

No issue was found during the parsing and extraction.

The returned structure `cms.ParsedSignedData`, also called `SignedToken`, contains the content, content type, the certificate chained used to sign the token, the CRLs, and the signature. The structure content is specified as follows:

```go
type ParsedSignedData struct {
    // Content is the content of the EncapsulatedContentInfo.
    Content []byte
    // ContentType is the content type of the EncapsulatedContentInfo.
    ContentType asn1.ObjectIdentifier
    // Certificates is the list of certificates in the SignedData.
    Certificates []*x509.Certificate
    // CRLs is the list of certificate revocation lists in the SignedData.
    CRLs []x509.RevocationList
    // SignerInfos is the list of signer information in the SignedData.
    SignerInfos []SignerInfo
}
type SignerInfo struct {
    // Version field specifies the syntax version number of the SignerInfo.
    Version int
        // SignerIdentifier field specifies the signer's certificate. Only
IssuerAndSerialNumber
    // is supported currently.
    SignerIdentifier IssuerAndSerialNumber
    // DigestAlgorithm field specifies the digest algorithm used by the signer.
    DigestAlgorithm pkix.AlgorithmIdentifier
    // SignedAttributes field contains a collection of attributes that are
    // signed.
    SignedAttributes Attributes `asn1:"optional,tag:0"`
    // SignatureAlgorithm field specifies the signature algorithm used by the
    // signer.
    SignatureAlgorithm pkix.AlgorithmIdentifier
    // Signature field contains the actual signature.
    Signature []byte
    // UnsignedAttributes field contains a collection of attributes that are
    // not signed.
    UnsignedAttributes Attributes `asn1:"optional,tag:1"`
}
```

INFO	**QB-4** Abort TSP HTTP Request Response Validation if invalid signature
Perimeter	Time-stamping verification

Description

At this stage of the process, the signature should be verified before processing anything further to follow the principle of defense-in-depth and execute as few as possible instructions with potential malicious data.. The signature is actually verified only at a later stage (see Section 6.1.8.).

Recommendation

Verify now the signature before further processing.

Instead, the `Validate` method continues to verify the following fields, extracted from the TST:

- The version number;
- The policy;
- The `MessageImprint` structure value to be the same as the one in the request;
- The timezone to be `UTC`;
- The `Nonce` to be the same as in the request;
- The signing certificate specified by the `signerInfo` is present in the `SignedToken Certificates` slice thanks `func (t *SignedToken) SigningCertificate` from `tspclient` where :
  - The signing certificate references `issuerSerial` are extracted from `signerInfo.SignedAttributes.Certificates[0]`;
  - If present, the `issuerSerial.SerialNumber` and `issuerSerial.IssuerName` are extracted in order to be compared to those contained in the submitted certificate chain and the corresponding certificate is returned.
  - If not present, the `DER` encoded Issuer and certificate serial number are used from the `signerInfo.SignerIdentifier` to search and find the corresponding certificates.
  - The hash sum of the selected certificates is then computed and compared to the one extracted from the `SignedAttributes` of the specified signing certificate.

The overall request validation process in illustrated in Figure 5.

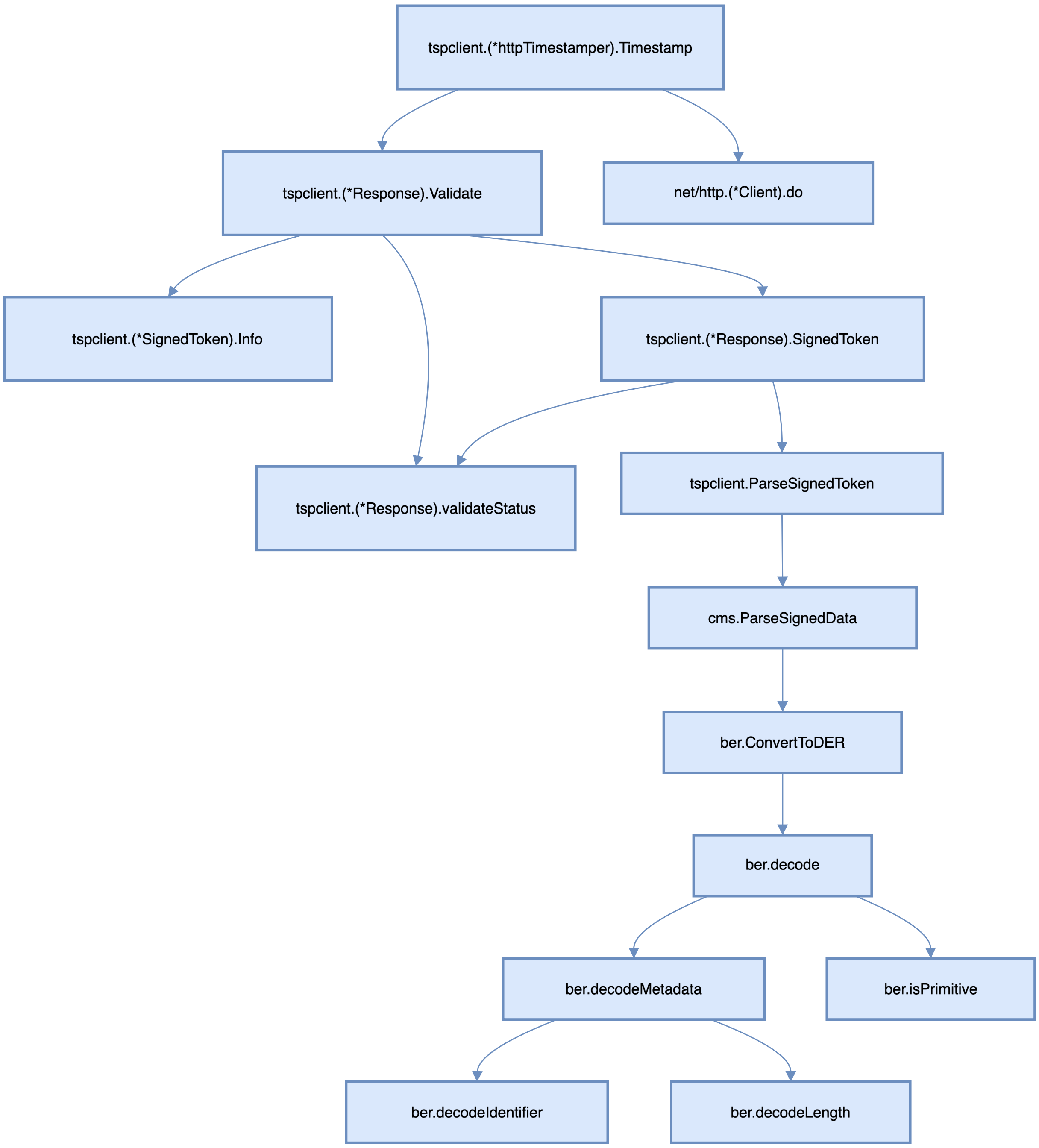

tspclient.(*httpTimestamper).Timestamp
tspclient.(*Response).Validate
net/http.(*Client).do
tspclient.(*SignedToken).Info
tspclient.(*Response).SignedToken
tspclient.(*Response).validateStatus
tspclient.ParseSignedToken
cms.ParseSignedData
ber.ConvertToDER
ber.decode
ber.decodeMetadata
ber.isPrimitive
ber.decodeIdentifier
ber.decodeLength

Figure 5: *Time-stamp response validation control flow graph*

6.1.6. TST Validation

The method `SignedToken` is again called and the whole BER to DER conversion process is repeated, as well as response parsing in order to get the `SignedToken` structure.

The `TSTInfo` is then validated against the original sent message through `tspclient.(*TSTInfo).Validate`. Then, `tspclient.(*TSTInfo).validate` (mind the minus `v`) verifies again that the `TSTInfo.Version` is equals to 1 (see Figure 6).

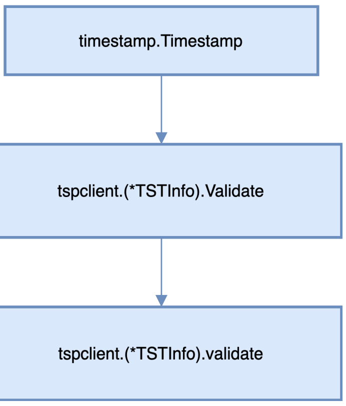


Figure 6: *Time-stamp token validation control flow graph*

It also reads the hash algorithm used out of the token, computes the hash sum of the original message (signature to be time-stamped) and compares it to the `messageImprint.HashedMessage` of the received token.

This operation seems redundant as the `messageImprint` structure was verified to be the same as the one sent in the request earlier by `func (r *Response) Validate(req *Request)` from `tspclient` when verifying the response of the TSA.

The `accuracy` is then computed according to the TSA policy and a `Timestamp` structure is returned:

```
Timestamp{
    Value:    tst.GenTime, // Time of generation
    Accuracy: accuracy,    // Accuracy in seconds, milliseconds and microseconds
}
```

This field is never used in the countersignature creation part. It is, however, used when verifying the countersignature.

6.1.7. Token certificate chain identification and signature verification (signing side)

Method `(t *SignedToken) Verify(ctx context.Context, opts x509.VerifyOptions)` from `tspclient.Token` is called in order to verify the signature of the received token. The full verification process is illustrated in Figure 7.

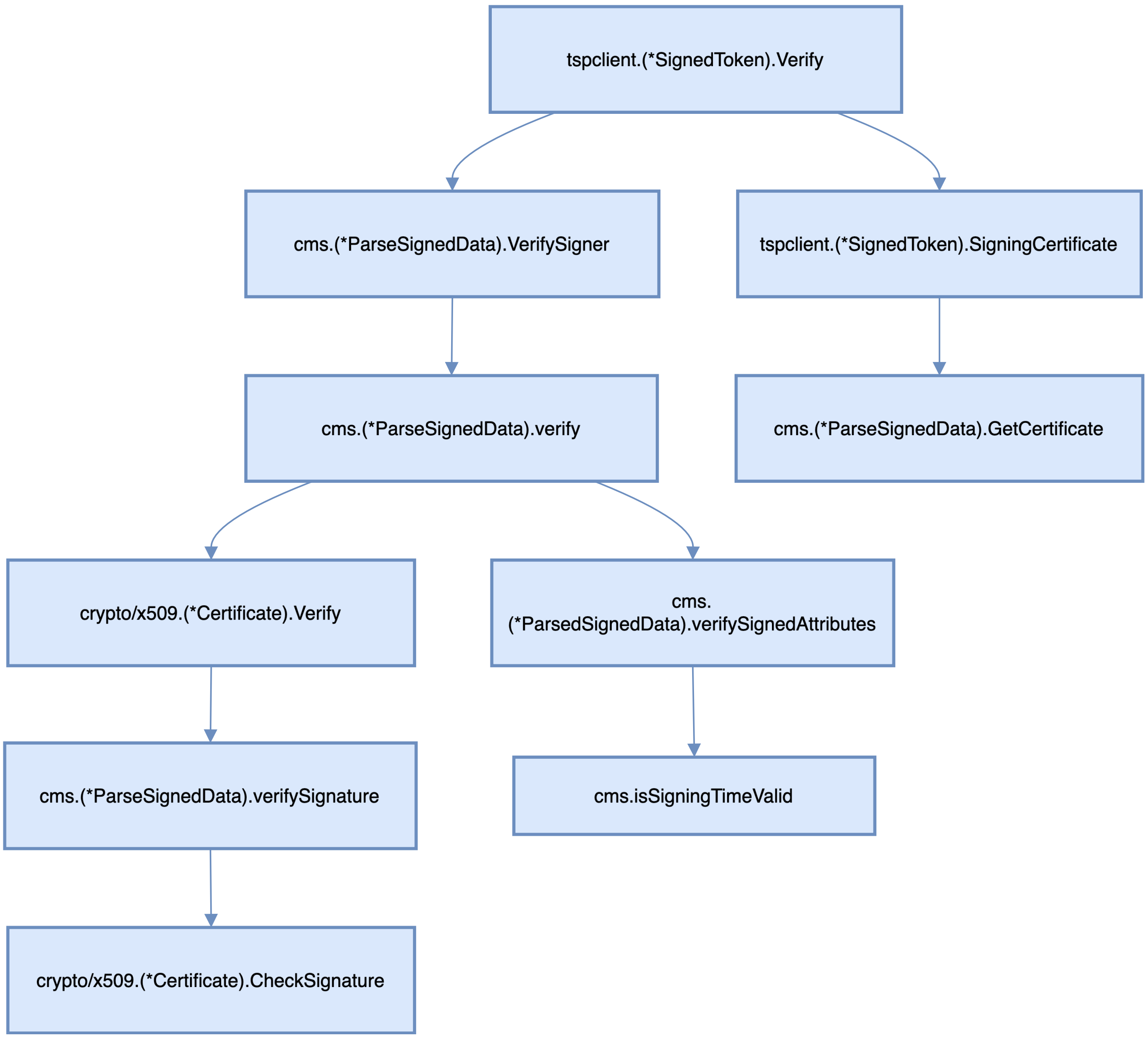

tspclient.(*SignedToken).Verify
cms.(*ParseSignedData).VerifySigner
tspclient.(*SignedToken).SigningCertificate
cms.(*ParseSignedData).verify
cms.(*ParseSignedData).GetCertificate
crypto/x509.(*Certificate).Verify
cms.
(*ParsedSignedData).verifySignedAttributes
cms.(*ParseSignedData).verifySignature
cms.isSigningTimeValid
crypto/x509.(*Certificate).CheckSignature

Figure 7: *Time-stamp signature verification control flow graph*

For each `SignerInfo`, the actual signing certificate is identified and returned thanks to `(t *SignedToken) SigningCertificate` from `tspclient.Token`. This method behavior was described above in Section 6.1.5. .

The full signing certificate chain is then recovered using `(d *ParsedSignedData) verify(signerInfo *SignerInfo, cert *x509.Certificate, opts *x509.VerifyOptions)` from `cms.Signed` which calls `crypto/x509.(*Certificate).Verify`. Using the configured root certificate and the intermediate certificates provided by the TSA, the certificate chained is returned, if valid.

Please note however, that during this full verification process, no check on the validity of the certificates is done. In fact, revoked certificates can be used in the certification chain, leading to a countersignature signed by a revoke entity being accepted by `notation`.

MEDIUM	QB-5 Revocation in certificate chain unchecked while signing		
Likelihood	●○○○	**Impact**	●●●○
Perimeter	Time-stamping verification		

Description

There is currently no check in the certification chain to see if revoked certificates are used when verifying the time-stamp countersignature received from TSA.

Recommendation

Add a check to certification chain to verify their status while signing.

Info

The associated pull requests for this issue are:

- https://github.com/notaryproject/notation-core-go/pull/246
- https://github.com/notaryproject/notation-go/pull/482
- https://github.com/notaryproject/notation/pull/1094

This security issue has been assigned a CVE-ID, see GHSA-45v3-38pc-874v / CVE-ID: CVE-2024-56138.

The signature is then checked by `(d *ParsedSignedData) verifySignature` from `cms.Signed`. If present, `SignedAttributes` of the `signerInfo` are the contents that are actually signed. Their contents are therefore used to be verified with the signature. Otherwise, the signature is generated from the content coming from `TSTInfo`.
The signature verification is performed by `crypto/x509.(*Certificate).CheckSignature`.

Then, method `(d *ParsedSignedData) verifySignedAttributes ([]*x509.Certificate, error)` from `tspclient-go/internal/cms.cms` verifies the `SignedAttributes`.

If the `signerInfo.SignedAttributes` slice is empty and the content type specifies `id-data`, this function returns `nil, nil`:

```go
func (d *ParsedSignedData) verifySignedAttributes(signerInfo *SignerInfo, chains
[][]*x509.Certificate) ([]*x509.Certificate, error) {
  if len(chains) == 0 {
     return nil, VerificationError{Message: "Failed to verify signed attributes
because the certificate chain is empty."}
  }

  if len(signerInfo.SignedAttributes) == 0 {
    if d.ContentType.Equal(oid.Data) {
      return nil, nil
    }

```

```
        ...
```

INFO	**QB-6** Non-compliant to the RFC for verification of signed attributes
Perimeter	Time-stamping verification

Description

In `cms.(*ParsedSignedData).verifySignedAttributes` method, if the `signerInfo. SignedAttributes` slice is empty and the content type specifies `id-data`, returned values are `nil, nil`. While following RFC 5652 no error is returned, this case should never happen for a TST because, according to RFC 3161, the content type has to be `id-ct-TSTInfo`. Returning `nil` here triggers a generic error later during certificate chain verification because the slice is empty.

Recommendation

Either comply to RFC 5652 at this stage and handle the case in the caller method (e.g. `timestamp.Timestamp`), or handle the error at this stage.

If the content type is different than `id-data` and the field is present, the following fields are checked:

- The content type specified in the `ContentType` and the one of the TST
- The `signerInfo.SignedAttributes.MessageDigest` and the hash sum computed out of the `TSTInfo`: added to the signature verification, this ensures that the TSA has signed this specific time-stamp token;
- The certificates of the chain to be valid compared to the `signerInfo.SignedAttributes.signingTime` field.

Finally, the time-stamping key usage from the extension key usage field of the signing certificate is checked and ensuring that it is marked as critical.

Info

The associated pull request for this issue is: https://github.com/notaryproject/tspclient-go/pull/35

6.1.8. Verification of the certificate chain (signing side)

Finally, the certificate chain used until then is verified to be a valid one according to the specification (see here). The overall verification of the certificate chain process is illustrated in Figure 8.

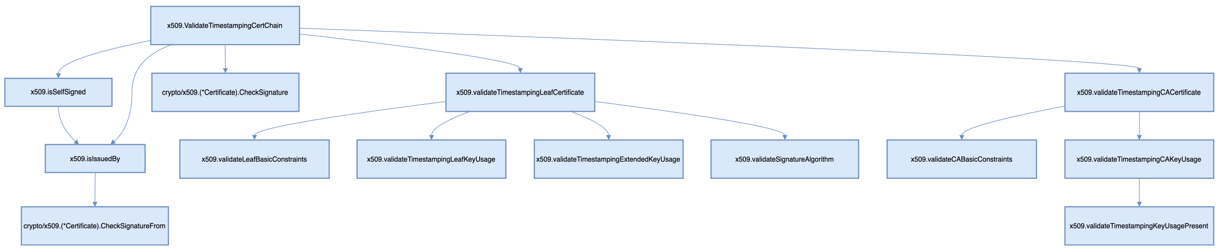

x509.ValidateTimestampingCertChain
x509.isSelfSigned
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafCertificate
x509.validateTimestampingCACertificate
x509.isIssuedBy
x509.validateLeafBasicConstraints
x509.validateTimestampingLeafKeyUsage
x509.validateTimestampingExtendedKeyUsage
x509.validateSignatureAlgorithm
x509.validateCABasicConstraints
x509.validateTimestampingCAKeyUsage
crypto/x509.(*Certificate).CheckSignatureFrom
x509.validateTimestampingKeyUsagePresent

Figure 8: *Time-stamp certificate chain verification control flow graph*

If the chain contains only one certificate, `validateTimestampingLeafCertificate(cert *x509.Certificate)` from `x509` verifies that the certificate is a self-signed certificate, and a valid leaf certificate as per the specification:

- The `cA` extended attribute must be set to `false` if `BasicConstraints` field is present;
- The `KeyUsage` extension is set and `Digital Signature` is set;
- The `ExtendedKeyUsage` extension is set, marked as critical and `Time Stamping Usage` is set.

If the chain contains more than one certificate, each of them is verified to not be a self-signed certificate and to have been issued by its parent in the certificate chain. The first certificate is verified to be a valid leaf certificate as per the specification, and the last certificate is verified to be a valid CA root certificate through `validateTimestampingCACertificate(cert *x509.Certificate, expectedPathLen int)` from `x509`:

- The `cA` extended attribute must be set and `BasicConstraints` field must be present;
- The `KeyUsage` extension is set and `Digital Signature` is set.

We noticed that in the verification process, several of the verification are redundant like:

- Verifying the issuance chain, as this is already checked by `crypto/x509.(*Certificate).Verify` (see Section 6.1.7.);
- Calling `validateTimestampingCACertificate` against the last certificate of the chain, as `crypto/x509.(*Certificate).Verify` makes sure the last certificate of the chain is the end user submitted root certificate. This certificate was verified to be a root certificate thanks to the method `x509.IsRootCertificate` which is even stricter than `validateTimestampingCACertificate`.

Success

No security issue was found during this part.

Danger

Even though no security issue was found here, the process of creating the countersignature does not follow security defense-in-depth principles. When retrieving the TSA response, first action is to manage to extract enough data to verify the signature and the integrity the data, before processing to further steps. As the data has to be considered insecure or malicious, any work on this data should be done after authenticity and integrity checks.

This means that after converting the BER data to DER and extracting the `CMS SignedData`, the certificate chain should be built and verified, the signature verified and the `TSTInfo` hash sum compared to the one specified in the

`SignedData.SignerInfo.SignedAttributes.MessageImprint` field . After this, the remaining verification work can be securely done.

6.2. Verifying a Signature and its Time-stamp Countersignature

In order to verify a signature of an OCI artifact, potentially also containing a time-stamp countersignature, one can leverage the `notation` CLI this way:

```
notation verify <artifact>
```

In our case, and as using `tag` is not recommended, we are using the manifest hash sum of the target container image:

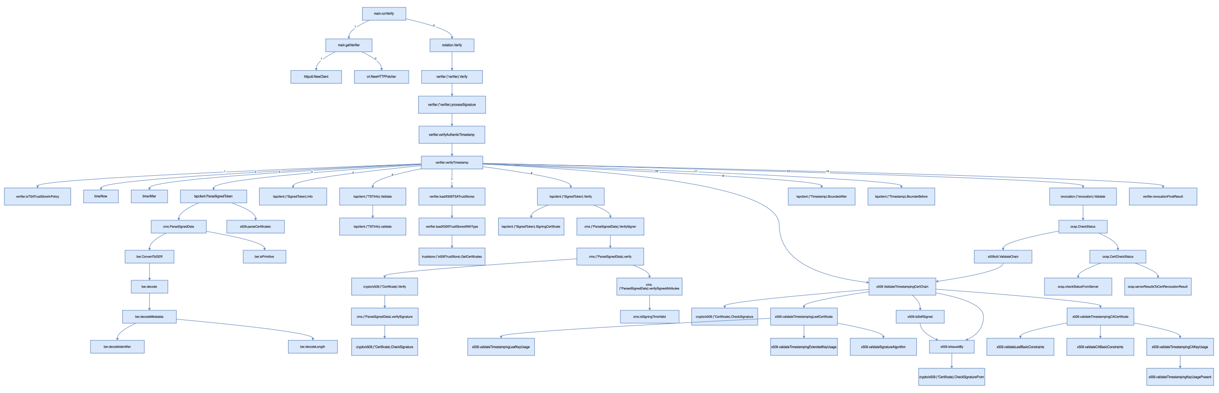
```
notation verify localhost:5001/
ubuntu@sha256:04b5ada4cdb5034a879599d9af5711687357fc25efb63ffdf2b44cfab69affb9
```

The signatures for the target artifact are then retrieved and each of them is verified until one is valid.

If the signature has not expired and is valid, the process of verifying the time-stamp countersignature starts by calling `verifyAuthenticTimestamp(ctx context.Context, policyName string, trustStores []string, signatureVerification trustpolicy.SignatureVerification, x509TrustStore truststore.X509TrustStore, r revocation.Validator, outcome *notation.VerificationOutcome) *notation.ValidationResult` from the package `verifier`.

If the signingScheme specified in the `SignedAttributes` of the `signerInfo` corresponds to `notary.x509`, then the method `verifyTimestamp(ctx context.Context, policyName string, trustStores []string, signatureVerification trustpolicy.SignatureVerification, x509TrustStore truststore.X509TrustStore, r revocation.Validator, outcome *notation.VerificationOutcome)` is called, which is ultimately in charge for the Time-stamp countersignature validation.

The overall process is illustrated in the following figure:



main.runVerify
1
2
main.getVerifier
notation.Verify
httputil.NewClient
crl.NewHTTPFetcher
verifier.(*verifier).Verify
verifier.(*verifier).processSignature
verifier.verifyAuthenticTimestamp
verifier.verifyTimestamp
verifier.isTSATrustStoreInPolicy
time/Now
time/After
tspclient.ParseSignedToken
tspclient.(*SignedToken).Info
tspclient.(*TSTInfo).Validate
verifier.loadX509TSATrustStores
tspclient.(*SignedToken).Verify
tspclient.(*Timestamp).BoundedAfter
tspclient.(*Timestamp).BounderBefore
revocation.(*revocation).Validate
verifier.revocationFinalResult
cms.ParseSignedData
x509.parseCertificates
tspclient.(*TSTInfo).validate
verifier.loadX509TrustStoresWithType
tspclient.(*SignedToken).SigningCertificate
cms.(*ParseSignedData).VerifySigner
ocsp.CheckStatus
ber.ConvertToDER
ber.isPrimitive
truststore.(*x509TrustStore).GetCertificates
cms.(*ParseSignedData).verify
x509util.ValidateChain
ocsp.CertCheckStatus
ber.decode
crypto/x509.(*Certificate).Verify
cms.(*ParsedSignedData).verifySignedAttributes
x509.ValidateTimestampingCertChain
ocsp.checkStatusFromServer
ocsp.serverResultsToCertRevocationResult
ber.decodeMetadata
cms.(*ParseSignedData).verifySignature
cms.isSigningTimeValid
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafCertificate
x509.isSelfSigned
x509.validateTimestampingCACertificate
ber.decodeIdentifier
ber.decodeLength
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafKeyUsage
x509.validateTimestampingExtendedKeyUsage
x509.validateSignatureAlgorithm
x509.isIssuedBy
x509.validateLeafBasicConstraints
x509.validateCABasicConstraints
x509.validateTimestampingCAKeyUsage
crypto/x509.(*Certificate).CheckSignatureFrom
x509.validateTimestampingKeyUsagePresent

Figure 9: *Overall view of verifying a time-stamp countersignature*[2]

6.2.1. Verifying the TSA policy

It is firstly verified that a trust store configured for a `TSA` exists in the policy using `isTSATrustStoreInPolicy(policyName string, trustStores []string)`. The method iterates

[2]Full page scaled picture can be found in Appendix C.B.

over the `trustStores` searching for entries that begin with `tsa:`. If there is no entry or no such trust store configured, the verification of the countersignature is disabled and it is just verified that the current local time is in the range of the certificate chain validity period.

If a `TSA` trust store is configured, it is then verified if the trust policy enforces the countersignature verification every time (corresponds to an empty `VerififyTimestamp` option field in the `trustpolicy.SignatureVerification` structure), or only if the certificate chain of the regular signature has expired, as per the specification (see here).

The behavior of the control-flow corresponds to the specification and what the policy specifies.

6.2.2. Time-stamp countersignature extraction and validation

At this point, if the `UnsignedAttributes` of the `signerInfo` structure does not contain a time-stamp countersignature, the process returns an error.

The `signedToken` is extracted by `ParseSignedToken(berData []byte)` from `tspclient` as detailed in Section 6.1.5. .

INFO	**QB-7** Abort Counter-signature verification if invalid signature
Perimeter	Verification with time-stamping
Description	
When enough data are extracted to verify the signature in the TST itself, the verification process continues to extract and parse the remaining content before checking the validity of the signature.	
Recommendation	
Verify the signature of the time-stamp token before continuing to extract and parse the rest of the information in the TST.	

Warning

At this point, enough data have been extracted in order to verify the signature of the time-stamp token, however, it continues to extract and parse the remaining content.

Info

The associated pull requests for this issue are https://github.com/notaryproject/notation-core-go/pull/243 and https://github.com/notaryproject/notation-go/pull/478

The `TSTInfo` structure is extracted and validated by `(tst *TSTInfo) Validate(message []byte)` from `tspclient`, which ensures that the countersignature is issued for the signature that is currently verified. This process is also detailed in Section 6.1.6. and illustrated in Figure 10.

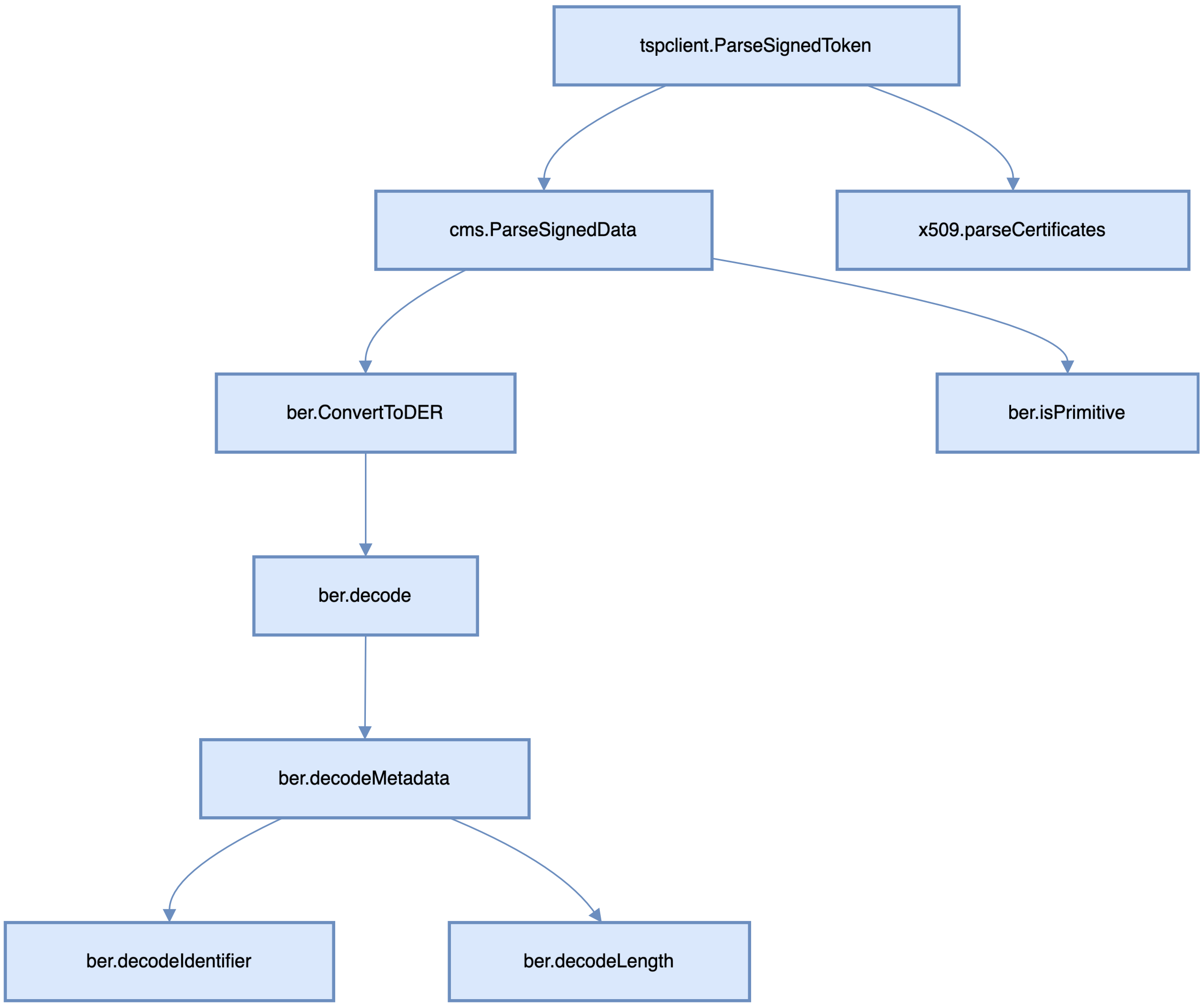


Figure 10: *Timestamp countersignature extraction and validation control flow graph*

6.2.3. Loading the trusted certificates

The trusted certificates contained in the trust store are loaded by `loadX509TSATrustStores(ctx context.Context, scheme signature.SigningScheme, policyName string, trustStores []string, x509TrustStore truststore.X509TrustStore)` from `verifier`.

If the signing scheme is `notary.x509`, which at this point of the control-flow cannot be something else, the method `loadX509TrustStoresWithType(ctx context.Context, trustStoreType truststore.Type, policyName string, trustStores []string, x509TrustStore truststore.X509TrustStore)` is called in order to retrieve a `TSA` trust store, specified in the `trustStoreType` argument.

For each trust store that is of type `TSA`, the method `(trustStore *x509TrustStore) GetCertificates(ctx context.Context, storeType Type, namedStore string)` from `verifier` is called in order to retrieve the corresponding certificates, as depicted in Figure 11.

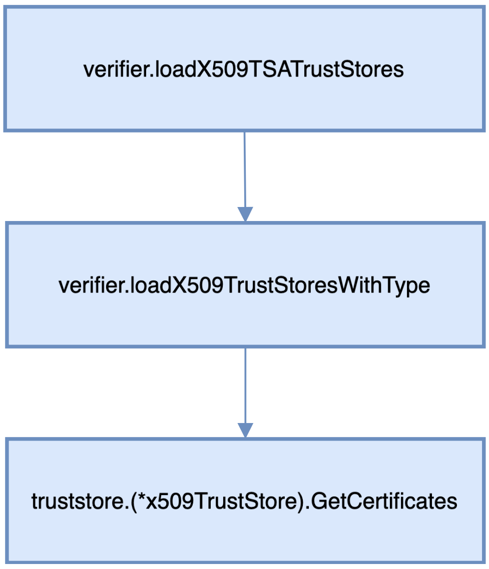

verifier.loadX509TSATrustStores
verifier.loadX509TrustStoresWithType
truststore.(*x509TrustStore).GetCertificates

Figure 11: *Load of the trusted certificates control flow graph*

It is then verified that:

- the store type is valid (either `TSA` or `CA`) ;
- The name of the store is also valid (correspond to the regular expression `[a-zA-Z0-9_.-]+`.);
- The path to the trust store is a regular directory, and not a symbolic link.

The certificate is then read and parsed by `ReadCertificateFile(path string) ([]*x509.Certificate, error)`, as described in Section 6.1.1. .

The certificates from the trust store are then verified to be CA certificates or self-signed certificate by `ValidateCertificates(certs []*x509.Certificate)` from `verifier`.

INFO	**QB-8** Shallow Verification of TSA trust store certificates
Perimeter	Certificate verification

Description

Verifications performed on the certificates from the TSA trust store are shallow and not sufficient to ensure the certificates are either Root CA certificate or self-signed. This is incoherent behavior since during Initialization phase 6.1.1. , the user-specified root certificate is verified to be a valid root CA certificate against `crypto/x509.IsRootCertificate`. Here, it is only verified that the `CA` attribute is set, and if not, the self-signature is. An intermediate CA certificate could be specified by the trust store, and an error will be triggered later during the certificate chain verification. This weakness is, however, not a vulnerability because stricter verification will occur later.

Recommendation

Have the same level of verification as in init phase or move the later verification to this stage.

Shallow Verification Code

```go
func ValidateCertificates(certs []*x509.Certificate) error {
  if len(certs) < 1 {
    return errors.New("input certs cannot be empty")
  }
  for _, cert := range certs {
    if !cert.IsCA { // If set, it is considered as CA certificate
      if err := cert.CheckSignature(cert.SignatureAlgorithm, cert.RawTBSCertificate, cert.Signature); err != nil {
        return fmt.Errorf(
          "certificate with subject %q is not a CA certificate or self-signed signing certificate",
          cert.Subject,
        )
      }
    }
  }
  return nil
}
```

Info

The associated pull request for this issue is: https://github.com/notaryproject/tspclient-go/pull/471

6.2.4. Token certificate chain identification and signature verification (verif side)

The certificate chain retrieval and the signature verification are then handled by the method `(t *SignedToken) Verify(ctx context.Context, opts x509.VerifyOptions)` from `tspclient` as detailed in Section 6.1.7. and illustrated in Figure 12.

The difference is that in this case, the `opts.Roots` might contain a CA certificate which is not a root certificate. The returned certificate chain therefore might end with an intermediate CA certificate.

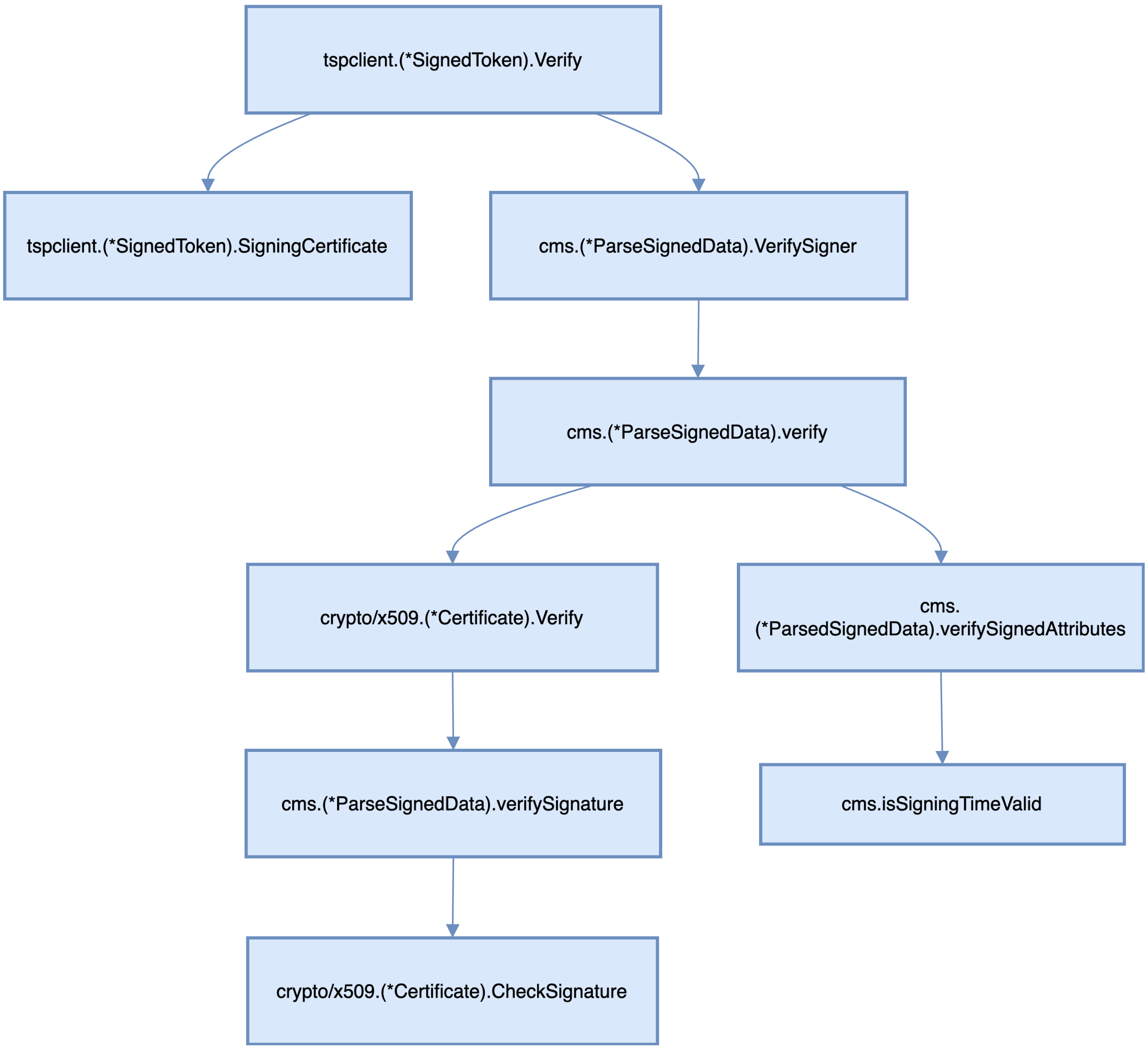

tspclient.(*SignedToken).Verify
tspclient.(*SignedToken).SigningCertificate
cms.(*ParseSignedData).VerifySigner
cms.(*ParseSignedData).verify
crypto/x509.(*Certificate).Verify
cms.(*ParsedSignedData).verifySignedAttributes
cms.(*ParseSignedData).verifySignature
cms.isSigningTimeValid
crypto/x509.(*Certificate).CheckSignature

Figure 12: *Certificate chain identification and signature verification control flow graph*

6.2.5. Verification of the certificate chain (verification side)

The retrieved certificate chain is verified by `ValidateTimestampingCertChain(certChain []*x509.Certificate)` from `x509` as detailed in Section 6.1.8. and depicted in Figure 13.

Here, if the last certificate of the identified certificate chain is an intermediate CA certificate, an error will be triggered because it is verified that it is a valid, self-signed certificate.

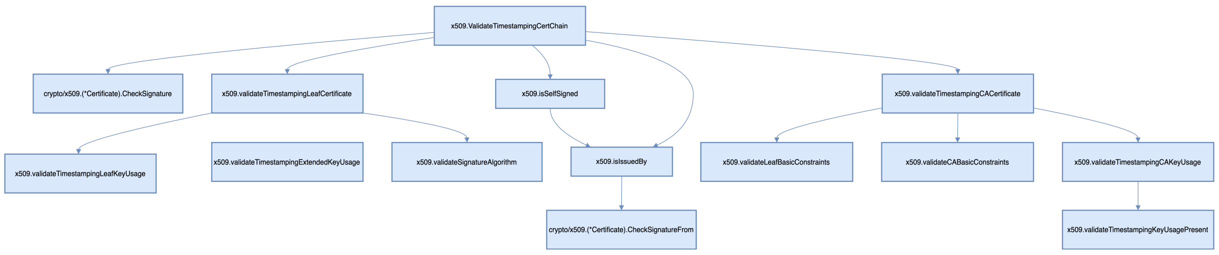

x509.ValidateTimestampingCertChain
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafCertificate
x509.isSelfSigned
x509.validateTimestampingCACertificate
x509.validateTimestampingLeafKeyUsage
x509.validateTimestampingExtendedKeyUsage
x509.validateSignatureAlgorithm
x509.isIssuedBy
x509.validateLeafBasicConstraints
x509.validateCABasicConstraints
x509.validateTimestampingCAKeyUsage
crypto/x509.(*Certificate).CheckSignatureFrom
x509.validateTimestampingKeyUsagePresent

Figure 13: *Certificate chain verification*

6.2.6. Verifying the time-stamp against the signing certificate chain

For each certificate contained in the certificate chain, it is verified that the time-stamp was not issued before or after the certificate validity period with the methods `(t *Timestamp) BoundedBefore(u time.Time)` and `(t *Timestamp) BoundedAfter(u time.Time)`. The `Accuracy` of the time-stamp is also taken into account, computed following the TSA policy.

6.2.7. Verifying the revocation status

The revocation status of the certificate chain is then verified by calling `(r *revocation) ValidateContext(ctx context.Context, validateContextOpts ValidateContextOptions)`

Warning

The version tested until now does not implement the CRL support. Additionally, CRL support audit is in the scope of Quarkslab audit and is therefore dedicated in Section 7. and Section 8. , as it does not only concern time-stamping countersignature, but also regular signatures.

The current version, however, implements OCSP. It is handled by `CheckStatus(opts Options)` from the `ocsp` package.

The certificate chain is again verified thanks to `ValidateTimestampingCertChain(certChain []*x509.Certificate)` from `x509`. Then for each certificate of the certificate chain except the last one, which is the root certificate, the method `certCheckStatus(cert, issuer *x509.Certificate, opts Options)` from `ocsp` is called.

Due to the time constraint of the audit and since the rest of the process is not only related to the time-stamping functionality (out of scope), the OCSP revocation process checking was not audited.

Finally, `revocationFinalResult(certResults []*revocationresult.CertRevocationResult, certChain []*x509.Certificate, logger log.Logger)` from `verifier` iterates over the `certResults` slice, and returns an error if one or more certificate is either revoked or an error occurred during the revocation check.

The full revocation status process is illustrated in Figure 14.

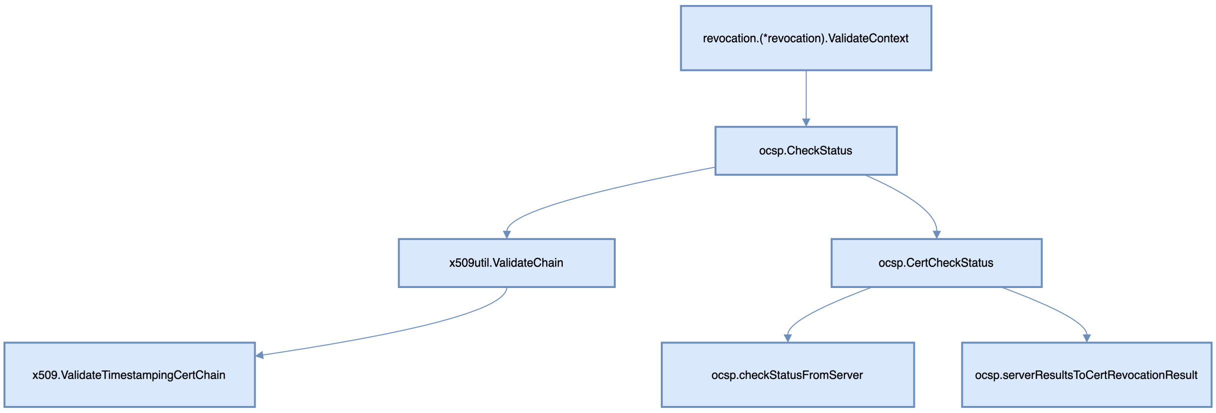

revocation.(*revocation).ValidateContext
ocsp.CheckStatus
x509util.ValidateChain
ocsp.CertCheckStatus
x509.ValidateTimestampingCertChain
ocsp.checkStatusFromServer
ocsp.serverResultsToCertRevocationResult

Figure 14: *Verification of the revocation status control flow graph*

Info

If more than one certificate is revoked, the certificate subject of the last one only is returned (i.e., the last one of the slices which is sorted from leaf to root).

7. Certificate Revocation List Compliance

7.1. Description

In public key infrastructure based on X.509 certificates (PKIX), the certificate authority (CA), namely the authority that is used to issue and sign certificates is also responsible for handling the revocation status of each of the certificates that they issued.

The two main ways to handle the revocation status are:

- Online Certificate Status Protocol (OCSP) (see RFC2560);
- Certificate Revocation List (CRL) (see RFC5280).

The CRL corresponds to a publicly available time-stamped list of revoked certificates, signed by a CA or CRL issuer. To identify the certificates in the list, their serial number is used. Using CRLs, hence means that the verification process of the certificate is enhanced by also including the retrieval of the corresponding CRL and checking if the serial number of the certificate is in the list.

Due to CRLs being signed by trusted authorities, they can be freely distributed over public (unsecured) channels.

However, this means that the CRL must be regularly updated with a time granularity that should be relevant to the use cases.

We detail hereafter the conformity of the source material to the RFC. Since most of the RFC is describing formatting and structure for CAs and CRL issuers, we focus here mainly on the compliance to the validation process detailed in the RFC.

Info

Please note that RFC5280 also describes the delta CRL features, which as of the time of the audit, was not implemented by `Notation` but most likely planned for later releases.

7.2. CRL Validation

7.2.1. CRL Cache

RFC2580 Section 6 states that for a proper validation process to work, the CRLs need to be available in a local cache and if the update time is reached, a mechanism needs to be implemented to fetch the most recent CRL and place it in the local CRL cache.

CRL Cache presence and update

The local cache for CRLs is available, where each CRLs expiry date is properly checked in `notation-go/verifier/crl/crl.go`. CRLs are properly fetched when expired in `notation-core-go/revocation/crl/fetcher.go`.

7.2.2. Revocation State

To support CRL processing, the algorithm requires the following state variables:

1. **reasons_mask**: This variable contains the set of revocation reasons supported by the CRLs and delta CRLs processed so far. The legal members of the set are the possible revocation reason values minus unspecified: keyCompromise, cACompromise, affiliationChanged, superseded, cessationOfOperation, certificateHold, privilegeWithdrawn, and aACompromise. The special value all-reasons is used to denote the set of all legal members. This variable is initialized to the empty set.
2. **cert_status**: This variable contains the status of the certificate. This variable may be assigned one of the following values: unspecified, keyCompromise, cACompromise, affiliationChanged, superseded, cessationOfOperation, certificateHold, removeFromCRL, privilegeWithdrawn, aACompromise, the special value UNREVOKED, or the special value UNDETERMINED. This variable is initialized to the special value UNREVOKED.
3. **interim_reasons_mask**: This contains the set of revocation reasons supported by the CRL or delta CRL currently being processed.

Presence of revocation state variable

They are defined in here, which is imported by `Notation` but the variables are never used.

7.2.3. CRL Processing

In order to check the revocation status of a certificate, one must perform the following for each corresponding CRL in the local CRL cache:

1. Update the local CRL cache by obtaining a complete CRL as required.
2. Verify the issuer and scope of the complete CRL as follows.
3. Obtain and validate the certification path for the issuer of the complete CRL. The trust anchor for the certification path MUST be the same as the trust anchor used to validate the target certificate. If a key usage extension is present in the CRL issuer's certificate, verify that the cRLSign bit is set.

4. Validate the signature on the complete CRL using the public key validated in step (3).
5. Search for the certificate on the complete CRL. If an entry is found that matches the certificate issuer and serial number then indicate the revocation reason.

CRL Processing

1. Update CRL: `notation-core-go/revocation/crl/fetcher.go`
2. Verify issuer and scope: `notation-core-go/revocation/internal/crl/crl.go`
3. Obtain and validate certification path: handled by `crypto/x509` with `(rl *RevocationList) checkSignatureFrom`
4. Validate the signature on the complete CRL: handled by `crypto/x509` with `(rl *RevocationList) checkSignatureFrom`
5. Search for the certificate: `notation-core-go/revocation/internal/crl/crl.go`

Warning

The RFC also includes other steps about the state variable `interim_reason_mask` that we did not find in `Notation` code. Upon quick investigation, it seems, however, that these checks are only relevant when a CRL does not support all the revocation reasons possible.

Info

As state by the RFC, a validation process with CRL does not require to fully follow the CRL processing specification but the implementation should follow the described logic. This is the case with `Notation`.

8. CRL Analysis in `Notation`

As explained in the previous section, part of the protocol to verify a certificate includes the check of the revocation status. These revocation checks can occur when:

- Verifying regular signatures;
- Verifying time-stamp counter signatures.

The usual way to perform such technique is via OCSP.

When revocation method with OCSP is not available or fails for checking the revocation status of a certificate, `notation` implements a CRL verification fallback.

The overall CRL verification overview is depicted in Figure 15.

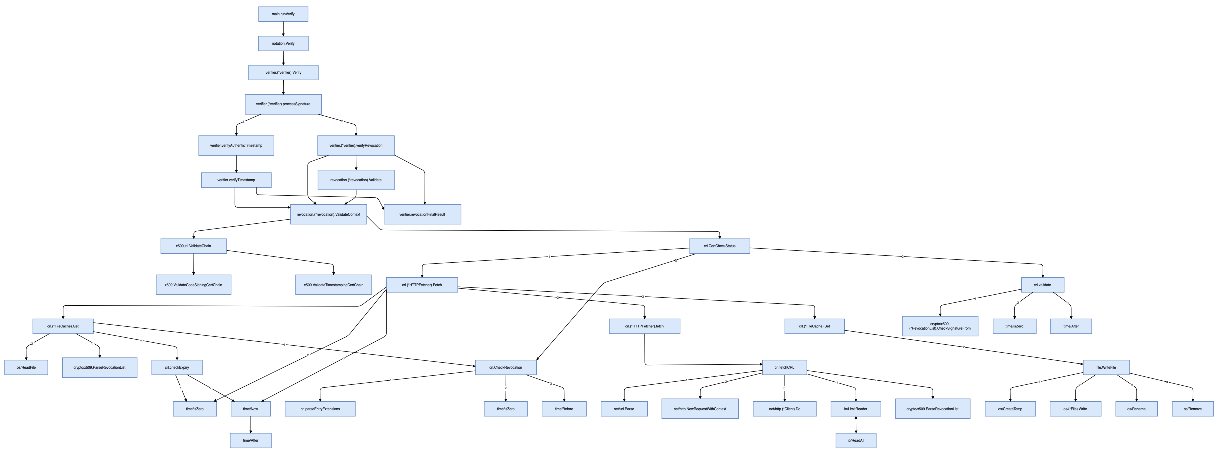


Figure 15: *Global overview of the CRL verification*[3]

8.1. Set up the updated source code

The CRL revocation functionality was released during Quarkslab audit. Quarkslab's engineers therefore needed to update the source code of the projects in order to test and audit the up-to-date ones.

```
notation: 0d9ceacde56c4b61dbbd7d83a7875986195781f8 (Tag v1.3.0-rc.1)
notation-go: a86f8da6ea2dcd5764ce026640cc4dfbeaa1c613 (Tag v1.3.0-rc.1)
notation-core-go: e90546bd90a8074357dba2cbf19f2e755bd4be2a (Tag v1.2.0-rc.1)
```

Info

As the updated CLI, `notation` is not using `notation-go` version `v1.3.0-rc.1` at the time of the audit, but version `v1.2.0-beta.1.0.20240926015724-84c2ec076201` instead, a patch has

[3]Full page scaled picture can be found in Appendix C.C.

been applied in order to have a working environment that included the up-to-date CRL revocation check feature. You can find the way we applied the patch in the Appendix B.

8.2. Revocation verification

The method `(r *revocation) ValidateContext(ctx context.Context, validateContextOpts ValidateContextOptions)` from `revocation` handles the verification of the revocation status of the certificate chains, contained in its `validateContextOpts` argument.

This method is either directly called when verifying the revocation status of the certificate chain that has signed a time-stamp countersignature, by `verifyTimestamp(ctx context.Context, policyName string, trustStores []string, signatureVerification trustpolicy.SignatureVerification, x509TrustStore truststore.X509TrustStore, r revocation.Validator, outcome *notation.VerificationOutcome)` , or by `(v *verifier) verifyRevocation(ctx context.Context, outcome *notation.VerificationOutcome)` from `verifier` when the regular signature verification is processed.

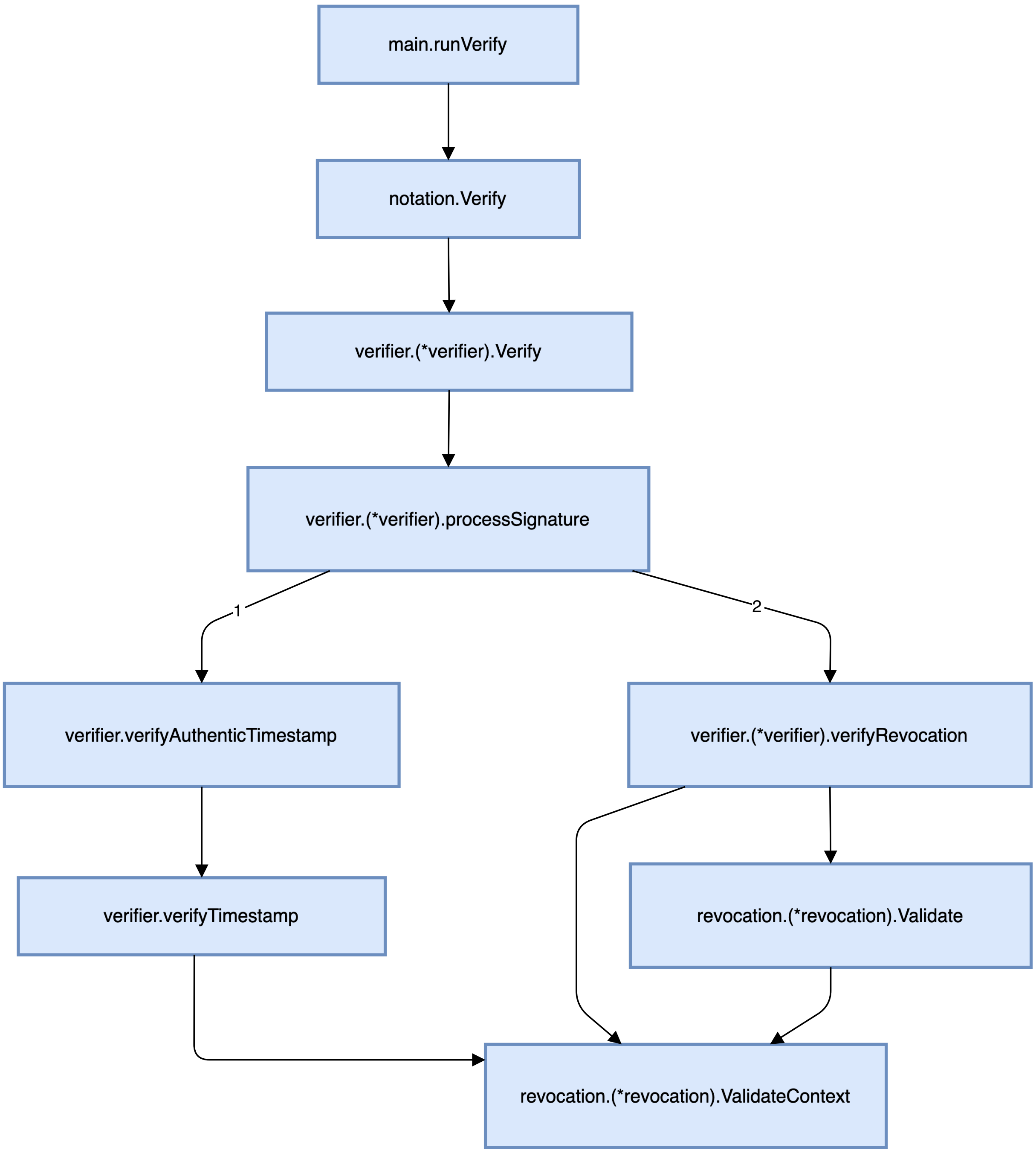

main.runVerify
notation.Verify
verifier.(*verifier).Verify
verifier.(*verifier).processSignature
1
2
verifier.verifyAuthenticTimestamp
verifier.(*verifier).verifyRevocation
verifier.verifyTimestamp
revocation.(*revocation).Validate
revocation.(*revocation).ValidateContext

Figure 16: *Paths leading to revocation verification*

8.2.1. Certificate chain verification

First, it is verified that the certificate chain is not empty and is valid by calling `ValidateChain(certChain []*x509.Certificate, certChainPurpose purpose.Purpose)` from `x509util`. Depending on the `certChainPurpose` value which can be either `CodeSigning` or `Timestamping` , either `ValidateCodeSigningCertChain(certChain []*x509.Certificate, signingTime *time.Time)` or `ValidateTimestampingCertChain(certChain []*x509.Certificate)` from `x509` is called, again, as already described in Section 6. .

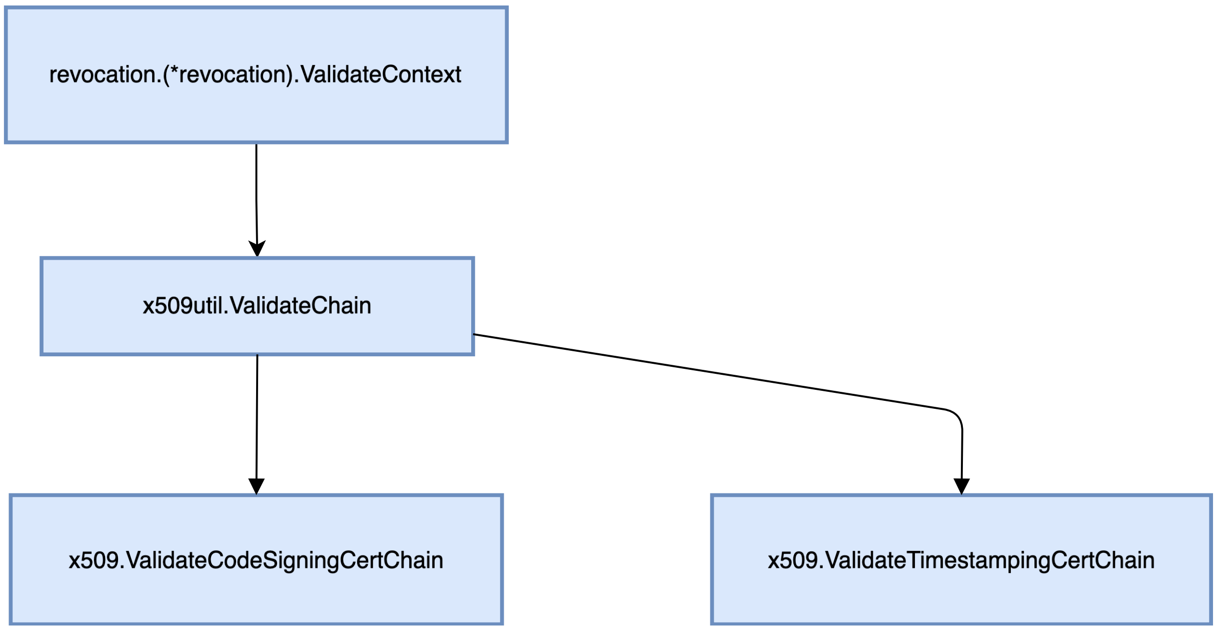

revocation.(*revocation).ValidateContext
x509util.ValidateChain
x509.ValidateCodeSigningCertChain
x509.ValidateTimestampingCertChain

Figure 17: *Certificate chain verification*

8.2.2. Revocation checking methods

Info

As OCSP is out of scope and is used each time it is available, we commented out the code corresponding to OCSP in order to only rely on CRL.

If the certificate chain only contains one certificate, it is assumed that it is a root certificate which is not revokable. The following result structure is set for the certificate that is currently:

```
certResults[len(certChain)-1] = &result.CertRevocationResult{
  Result: result.ResultNonRevokable,
  ServerResults: []*result.ServerResult{{
    Result:           result.ResultNonRevokable,
    RevocationMethod: result.RevocationMethodUnknown,
  }},
  RevocationMethod: result.RevocationMethodUnknown,
}
```

Otherwise, for each certificate except the root one, from leaf to root order:

- It is normally checked that `OCSP` method is available, and, if it is, tries to verify the revocation status using this method. If verification process fails for unknown reason, then it fallbacks to CRL;
- It is checked that `CRL` is available, and if it is, verifies the revocation status using this method;
- By default, the same structure as for a root certificate is set.

INFO	QB-9	No proper error handling when OCSP or CRL are not available
Perimeter	Revocation status verification	

Description

If none of the revocation check methods are available, the returned structure only tells that the status of the certificate is not revokable. This will not trigger any error, even if the revocation verification policy is set to `strict`.

Recommendation

Another level of verification could be implemented, so that `strict` mode raises an error, or at least prints warning logs if the revocation checks are not available, especially when the certificate chain contains more than one certificate.

Info

The associated pull request for this issue is: https://github.com/notaryproject/notation-go/pull/479

The method `CertCheckStatus(ctx context.Context, cert, issuer *x509.Certificate, opts CertCheckStatusOptions)` from `crl` handles the revocation status check using CRL.

8.2.3. Fetching CRL

Fetching CRL is handled by `(f *HTTPFetcher) Fetch(ctx context.Context, url string)` from `crl`. If a cache exists, it will try to extract the CRL from it.

8.2.3.1. Extracting CRL from the cache

Info

The root path to the cache is located in the user default location for cache, according to the operating system, followed by `/notation/crl`.

The method `(c *FileCache) Get(ctx context.Context, url string)` is called, it gets the `sha256` sum of the `url` argument, and tries to read the content of the file with the corresponding name in the cache repository. If it succeeds, the content is deserialized into a `fileCacheContent` structure which is then parsed by `crypto/x509.ParseRevocationList`.

Expiry time is then checked by `checkExpiry(ctx context.Context, nextUpdate time.Time)` from `crl`.

Back to `crl.(*HTTPFetcher).Fetch`, expiry time is again checked, exactly the same way `crl.checkExpiry` does.

At this point, if no error is encountered, the bundle is returned. Otherwise, if a non-critical error was encountered like absent or expired cache, it is fetched.

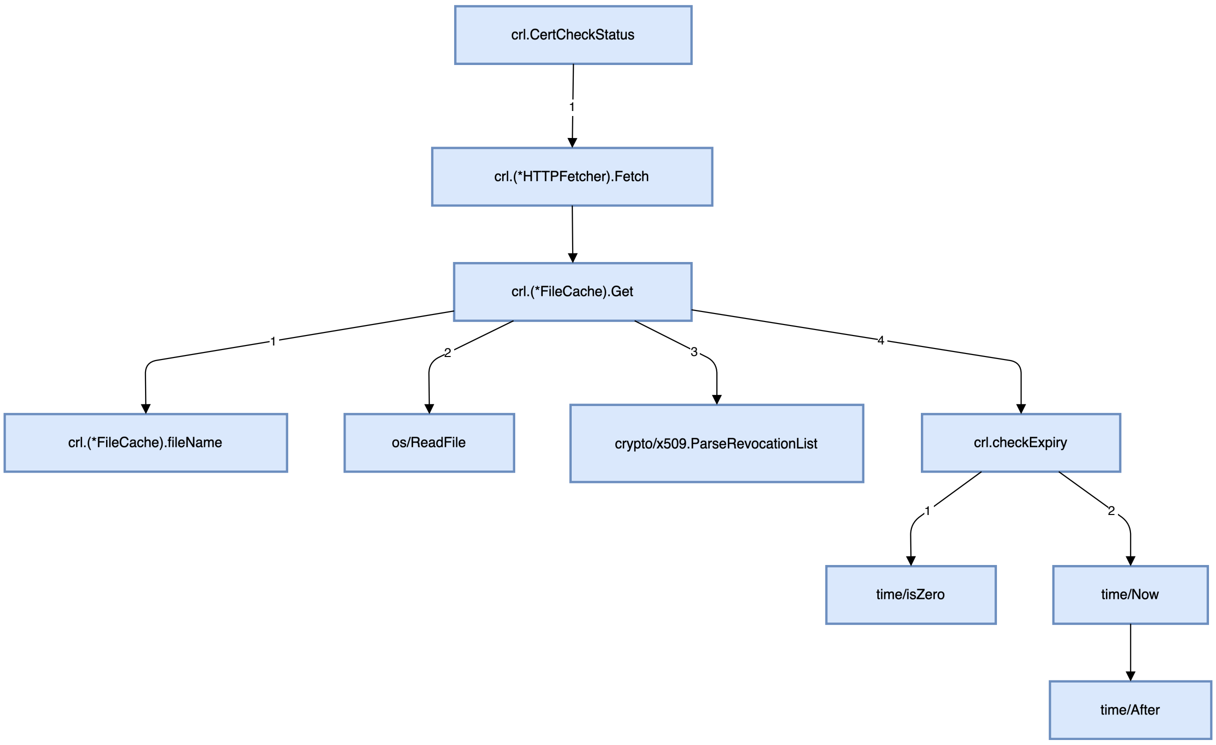


Figure 18: *Fetching CRL from cache*

Success

No security issue was discovered during this section.

8.2.3.2. Fetching and extracting CRL from URL

The method `(f *HTTPFetcher) fetch(ctx context.Context, url string)` from `crl` is called, which calls `fetchCRL(ctx context.Context, crlURL string, client *http.Client)`. `crlURL` is parsed by `net/url.Parse` and the url scheme is verified to be `http`.

The CRL is then downloaded, during up to 2 seconds. The response content is read up to 32MB if the HTTP response status code is between 200 and 299.

INFO	**QB-10** Non-compliant use of HTTP Status Code
Perimeter	Fetch of CRL

Description

When fetching CRL from specified url, using HTTP Get method, any HTTP status code between 200 and 299 are accepted. Even if an HTTP Status code starting with 2 is usually related to a successful operation, anything but 200 should be rejected here and considered as an error, as it would not make sense to receive something else considering the sent HTTP `GET` request.

Recommendation

Reject any response containing anything but HTTP 200 as status code.

The downloaded content is then parsed by `crypto/x509.ParseRevocationlist` and returned.

Info

This security issue has been taken into account and is fixed by the Notary Project maintainers. The PR is currently private.

8.2.3.3. Setting the CRL cache

Finally, the method `(c *FileCache) Set(ctx context.Context, url string, bundle *corecrl.Bundle)` from `crl` is called to try to fill the cache with the downloaded content, if the cache exists.

The raw content is `JSON` serialized and passed to `WriteFile(path string, content []byte)` from `file`. The `path` argument is built out of the path to the cache content, and the `sha256` sum of the filename.

A temporary file is created using `os/CreateTemp`, in the operating system dedicated area for temporary files. The file is written with the `content`, closed, and then moved and renamed thanks to `os/Rename`.

LOW	**QB-11**	Non-portable way of creating temporary files for CRL's cache	
Likelihood	●○○○	**Impact**	●○○○
Perimeter	CRL Cache Creation		

Description

Method `os/Rename` is used to rename and move a temporary file to the user cache folder. As detailed in the `Rename` method description, OS specific restrictions may apply. On Linux, it is relying on `libc`, `rename` function. As per rename documentation, renaming cannot be applied when the source and destination are located on two different mountpoints. On modern Linux operating systems (tested with Fedora 38), dedicated temporary data area is often a dedicated partition of type `tmpfs`, located in `/tmp`. An `EXDEV` error, detailed as "`Cross device link not permitted`" is therefore raised during the execution of `file.WriteFile` on such operating systems.

Recommendation

The file should be copied instead of being moved, or, directly created in the user cache directory and then renamed. First solution can be implemented thanks `os.Open`, `os.Create`, `io.Copy` and `os.Remove` from standard Go library.

Info

This security issue has been fixed by the Notary Project maintainers and has been assigned a CVE-ID, see GHSA-qjh3-4j3h-vmwp / CVE-ID: CVE-2024-51491.

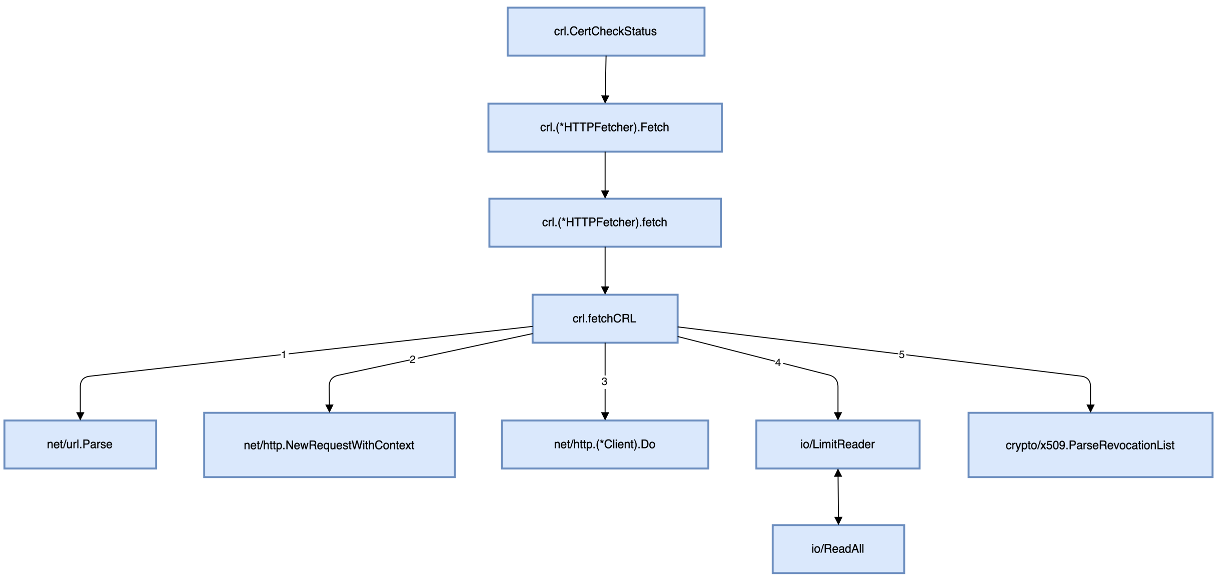


Figure 19: *Fetching CRL from URL*

8.2.4. Validation of the CRL bundle

The method `validate(crl *x509.RevocationList, issuer *x509.Certificate)` is then called in order to verify the signature of the bundle. `crypto/x509.(*RevocationList).CheckSignatureFrom` is leveraged in order to check it. The expiry time is checked and the extensions of the revocation list are verified to not contain a critical one, or a delta CRL.

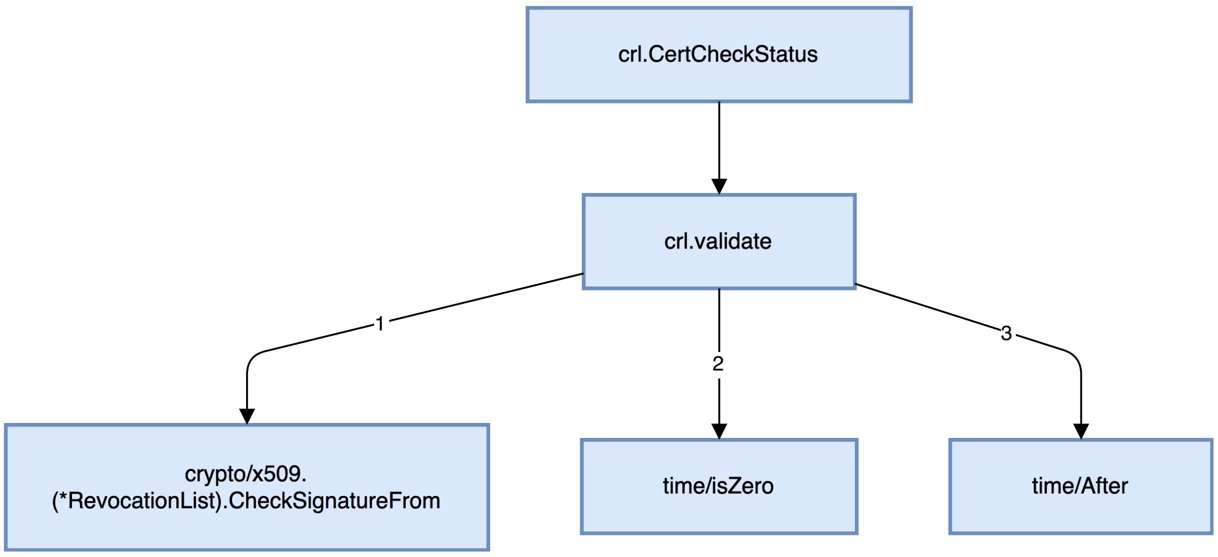


Figure 20: *Validation of the CRL bundle*

Success

No security issue was discovered during this section.

8.2.5. Check the revocation status

The revocation list is iterated by `checkRevocation(cert *x509.Certificate, baseCRL *x509.RevocationList, signingTime time.Time, crlURL string)` from `crl`, comparing the serial number of each entry against the currently tested certificate.

If the certificate is in the revocation list:

- Its extensions are extracted by `parseEntryExtensions(entry x509.RevocationListEntry)` from `crl`, where it is verified that the `InvalidityDate` extension is present and valid, and that there is no critical extension;
- It is verified that the invalidity date is after the signing time, if both of them are specified. If it is the case, the certificate was revoked after the time of signing and is not considered revoked. Otherwise, it is considered as revoked and the method immediately returns a `ServerResult` containing a `ResultRevoked` Result.

If the end of the loop is reached, the method returns a `ServerResult` with a `ResultOK` Result.

Jumping back to `crl.CertCheckStatus`, if the returned structure `Result` fields are equal to the `result.ResultRevoked`, a `CertRevocationResult` is returned containing the same `Result` value. If the end of the loop over the CRL URL is reached, and an error was encountered, the set value for `Result` is `ResultUnknown`. Otherwise, it is a success and the set value is `ResultOK`.

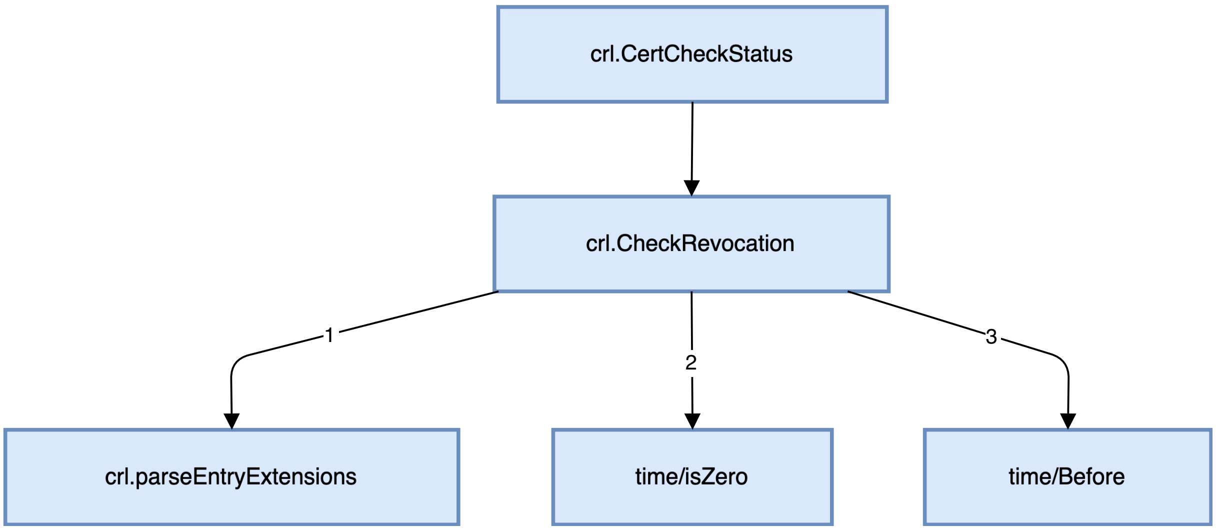


Figure 21: *Revocation checking*

Success

No security issue was discovered during this section.

8.2.6. Interpreting the results

The final result of the revocation status is handled by `revocationFinalResult(certResults []*revocationresult.CertRevocationResult, certChain []*x509.Certificate, logger log.Logger)` from `verifier`.

It iterates over `certResults`, logs any errors that occur during the revocation checking process and then either returns:

- `result.ResultOK`, `''` if no fatal error were encountered and no certificate were revoked;
- `result.ResultRevoked`, `"<Subject of the last revoked certificate of the chain>"` otherwise

Success

No security issue was discovered during this section.

9. Conclusions

Quarkslab identified several issues or bugs in Notary projects. However, only one of them may involve an immediate safety risk.

Quarkslab recognizes the considerable security efforts made by Notary developers to safeguard the tool, mainly thanks to the conscientious implementation of the different related RFCs.

Additionally, Quarkslab provided recommendations and strategies for addressing the issues, helping to strengthen the open-source tool and enhance its security moving forward.

A. Code Duplicate

```
term ~/2/sources (main)> dupl -t 30 $(find . | grep -v "test" | grep "\\.go\$")
found 2 clones:
  ./notation-core-go/revocation/internal/ocsp/errors.go:45,51
  ./notation-core-go/revocation/result/errors.go:27,33
found 2 clones:
  ./tspclient-go/internal/oid/algorithm.go:25,33
  ./tspclient-go/internal/oid/algorithm.go:34,42
found 3 clones:
  ./notation-go/plugin/plugin.go:108,129
  ./notation-go/plugin/plugin.go:121,142
  ./notation-go/plugin/plugin.go:134,155
found 2 clones:
  ./notation-go/dir/fs.go:36,40
  ./notation-go/internal/mock/mockfs/fs.go:29,33
found 2 clones:
  ./notation-go/verifier/verifier.go:768,768
  ./notation-go/verifier/verifier.go:970,970
found 2 clones:
  ./notation-go/plugin/proto/sign.go:14,54
  ./notation-go/plugin/proto/verify.go:14,56
found 2 clones:
  ./notation-go/verifier/truststore/errors.go:22,30
  ./notation-go/verifier/truststore/errors.go:42,50
found 6 clones:
  ./tspclient-go/errors.go:27,36
  ./tspclient-go/errors.go:50,59
  ./tspclient-go/errors.go:73,82
  ./tspclient-go/errors.go:96,105
  ./tspclient-go/internal/cms/errors.go:38,47
  ./tspclient-go/internal/cms/errors.go:61,70
found 2 clones:
  ./notation/cmd/notation/cert/generateTest.go:34,42
  ./notation/cmd/notation/key.go:31,39
found 2 clones:
  ./notation-go/verifier/verifier.go:185,188
  ./notation-go/verifier/verifier.go:208,211
found 2 clones:
  ./notation-core-go/x509/codesigning_cert_validations.go:98,133
  ./notation-core-go/x509/timestamp_cert_validations.go:91,126
found 3 clones:
  ./notation-go/plugin/proto/algorithm.go:36,43
  ./notation-go/plugin/proto/algorithm.go:153,160
  ./notation-go/plugin/proto/errors.go:28,65
found 2 clones:
  ./notation-go/plugin/errors.go:61,66
  ./notation-go/plugin/errors.go:86,91
found 2 clones:
  ./notation-go/verifier/verifier.go:92,102
  ./notation-go/verifier/verifier.go:105,115
found 2 clones:
  ./notation-core-go/revocation/revocation.go:241,248
  ./notation-core-go/revocation/revocation.go:253,260
found 2 clones:
  ./notation/cmd/notation/cert/list.go:128,131
  ./notation/cmd/notation/cert/list.go:144,147
found 4 clones:
  ./notation-go/plugin/plugin.go:108,116
  ./notation-go/plugin/plugin.go:121,129
  ./notation-go/plugin/plugin.go:134,142
  ./notation-go/plugin/plugin.go:147,155
found 2 clones:
  ./notation-core-go/signature/algorithm.go:73,83
  ./notation-core-go/signature/algorithm.go:85,95
found 3 clones:
  ./notation-go/internal/mock/mocks.go:166,166
  ./notation-go/registry/interface.go:39,39
  ./notation-go/registry/repository.go:144,144
found 4 clones:
  ./tspclient-go/errors.go:27,36
  ./tspclient-go/errors.go:50,59
  ./tspclient-go/errors.go:73,82
  ./tspclient-go/errors.go:96,105
found 2 clones:
  ./notation-core-go/signature/cose/envelope.go:411,421
  ./notation-core-go/signature/cose/envelope.go:422,432
found 3 clones:
  ./notation-go/internal/mock/mocks.go:191,193
  ./notation-go/internal/mock/mocks.go:195,197
  ./notation-go/internal/mock/mocks.go:199,201
found 3 clones:
  ./notation-go/verifier/helpers.go:50,50
  ./notation-go/verifier/helpers.go:134,134
  ./notation-go/verifier/helpers.go:145,145
found 2 clones:
  ./notation/cmd/notation/verify.go:176,182
  ./notation/cmd/notation/verify.go:185,191
found 9 clones:
  ./notation/cmd/notation/cert/generateTest.go:57,63
  ./notation/cmd/notation/inspect.go:98,104
  ./notation/cmd/notation/key.go:92,98
  ./notation/cmd/notation/key.go:123,129
  ./notation/cmd/notation/list.go:67,73
  ./notation/cmd/notation/login.go:56,62
  ./notation/cmd/notation/logout.go:39,45
  ./notation/cmd/notation/sign.go:107,113
  ./notation/cmd/notation/verify.go:84,90
found 2 clones:
  ./notation/cmd/notation/key.go:89,102
  ./notation/cmd/notation/logout.go:36,49
found 2 clones:
  ./notation-go/internal/envelope/envelope.go:37,44
  ./notation/internal/envelope/envelope.go:53,60
found 4 clones:
  ./notation/cmd/notation/common.go:35,37
  ./notation/cmd/notation/common.go:44,46
  ./notation/internal/cmd/flags.go:38,40
  ./notation/internal/cmd/flags.go:87,89
found 2 clones:
  ./notation-core-go/signature/internal/base/envelope.go:76,94
  ./notation-core-go/signature/internal/base/envelope.go:97,112
found 2 clones:
  ./notation-core-go/signature/cose/envelope.go:440,445
  ./notation-core-go/signature/jws/jwt.go:124,131
found 2 clones:
  ./notation-go/plugin/proto/algorithm.go:46,68
  ./notation-go/plugin/proto/algorithm.go:117,139
found 2 clones:
  ./tspclient-go/request.go:168,173
  ./tspclient-go/response.go:117,122
found 2 clones:
  ./notation-go/plugin/plugin.go:108,142
  ./notation-go/plugin/plugin.go:121,155
found 2 clones:
  ./notation-go/registry/repository.go:183,189
  ./notation-go/registry/repository.go:189,195
found 2 clones:
  ./tspclient-go/internal/cms/errors.go:38,47
  ./tspclient-go/internal/cms/errors.go:61,70
found 2 clones:
  ./notation/cmd/notation/inspect.go:108,116
  ./notation/cmd/notation/verify.go:97,105
found 2 clones:
  ./notation-core-go/revocation/internal/crl/crl.go:63,74
  ./notation-core-go/revocation/internal/crl/crl.go:76,86
```

```
found 2 clones:
  ./notation/cmd/notation/cert/add.go:37,43
  ./notation/cmd/notation/key.go:160,166
found 2 clones:
  ./notation/cmd/notation/cert/show.go:43,52
  ./notation/cmd/notation/plugin/uninstall.go:48,57
found 2 clones:
  ./notation-core-go/x509/codesigning_cert_validations.go:40,42
  ./notation-core-go/x509/timestamp_cert_validations.go:36,38
found 2 clones:
  ./notation/cmd/notation/list.go:67,79
  ./notation/cmd/notation/sign.go:107,119

Found total 41 clone groups.
```

B. Patch to `notation` CLI for CRL support

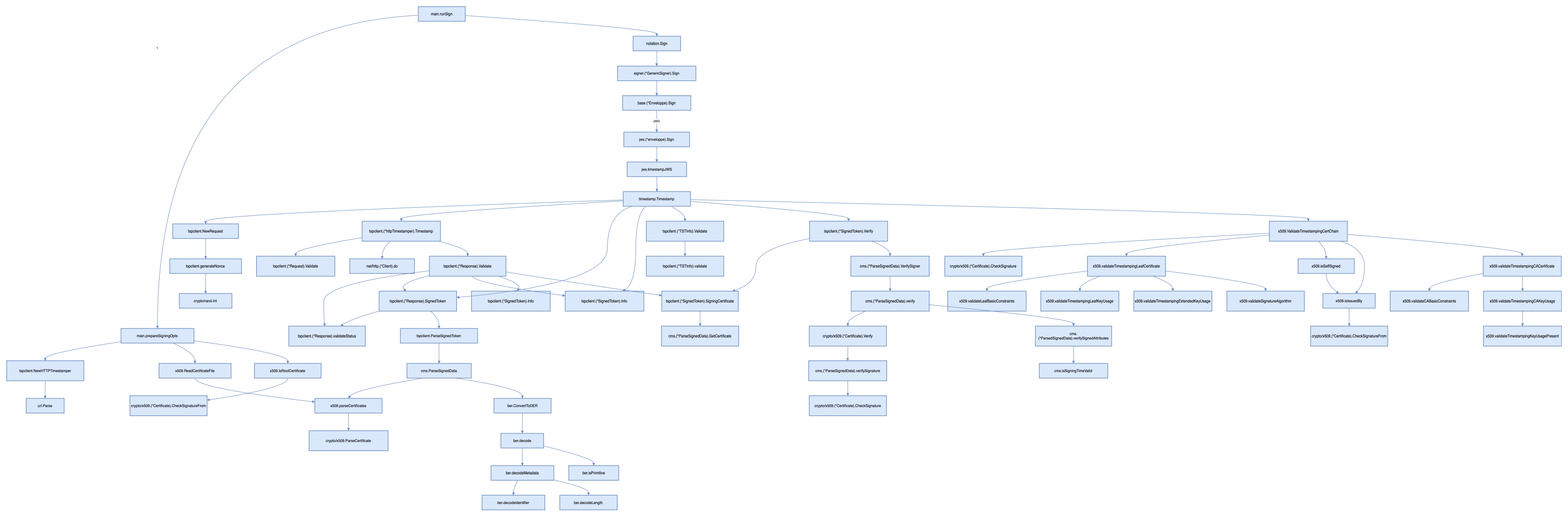
Patch to include `notation-go@v1.3.0-rc.1` in notation CLI.

```diff
diff --git a/cmd/notation/verify.go b/cmd/notation/verify.go
index 8943c08..96b1ee1 100644
--- a/cmd/notation/verify.go
+++ b/cmd/notation/verify.go
@@ -264,13 +264,13 @@ func getVerifier(ctx context.Context) (notation.Verifier, error) {
  }

  // trust policy and trust store
- policyDocument, err := trustpolicy.LoadOCIDocument()
+ policyDocument, err := trustpolicy.LoadDocument()
  if err != nil {
    return nil, err
  }
  x509TrustStore := truststore.NewX509TrustStore(dir.ConfigFS())

-  return  verifier.NewVerifierWithOptions(policyDocument,  nil,  x509TrustStore, plugin.NewCLIManager(dir.PluginFS()), verifier.VerifierOptions{
+       return       verifier.NewWithOptions(policyDocument,      x509TrustStore, plugin.NewCLIManager(dir.PluginFS()), verifier.VerifierOptions{
    RevocationCodeSigningValidator:  revocationCodeSigningValidator,
    RevocationTimestampingValidator: revocationTimestampingValidator,
  })
diff --git a/go.mod b/go.mod
index 1bd05a0..7da3817 100644
--- a/go.mod
+++ b/go.mod
@@ -4,7 +4,7 @@ go 1.23

 require (
  github.com/notaryproject/notation-core-go v1.2.0-rc.1
-                                            github.com/notaryproject/notation-go v1.2.0-beta.1.0.20240926015724-84c2ec076201
+ github.com/notaryproject/notation-go v1.3.0-rc.1
  github.com/notaryproject/tspclient-go v0.2.0
  github.com/opencontainers/go-digest v1.0.0
  github.com/opencontainers/image-spec v1.1.0
diff --git a/go.sum b/go.sum
index 9eade9e..96fa332 100644
--- a/go.sum
+++ b/go.sum
```

```
@@ -39,6 +39,8 @@ github.com/notaryproject/notation-core-go v1.2.0-rc.1 h1:VMFlG+9a1JoNAQ3M96g8iqC
 github.com/notaryproject/notation-core-go v1.2.0-rc.1/go.mod h1:b/70rA4OgOHlg0A7pb8zTWKJadF06781zS3a37KHEJQ=
 github.com/notaryproject/notation-go v1.2.0-beta.1.0.20240926015724-84c2ec076201 h1:2QBYa9Df+vMwMiaHaFqPoUiwfx5vcPEgM7KbusivTpw=
 github.com/notaryproject/notation-go v1.2.0-beta.1.0.20240926015724-84c2ec076201/go.mod h1:F6zMQl3PhVdCsI1xlIjK66kCorUQhWkoMtlZdvJWxFI=
+github.com/notaryproject/notation-go v1.3.0-rc.1 h1:pm9tdUy2tWYqlwyRDZyKXgLwAscDATPUYv0ul2RK/Iw=
+github.com/notaryproject/notation-go v1.3.0-rc.1/go.mod h1:W4o45yolX4Q+3PKlcpGleLLXEKWHa3BshEqw/JX5c6I=
 github.com/notaryproject/notation-plugin-framework-go v1.0.0 h1:6Qzr7DGXoCgXEQN+1gTZWuJAZvxh3p8Lryjn5FaLzi4=
 github.com/notaryproject/notation-plugin-framework-go v1.0.0/go.mod h1:RqWSrTOtEASCrGOEffq0n8pSg2KOgKYiWqFWczRSics=
 github.com/notaryproject/tspclient-go v0.2.0 h1:g/KpQGmyk/h7j60irIRG1mfWnibNOzJ8WhLqAzuiQAQ=
```

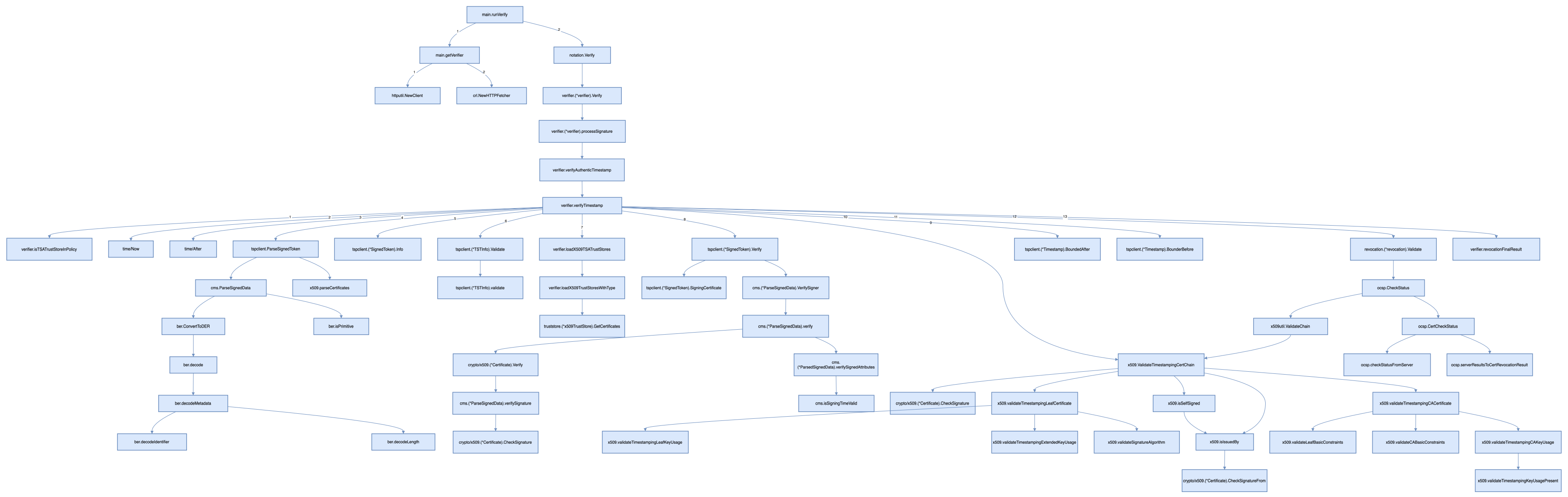
C. Figures

C.A. Time-stamp Control Flow Graph



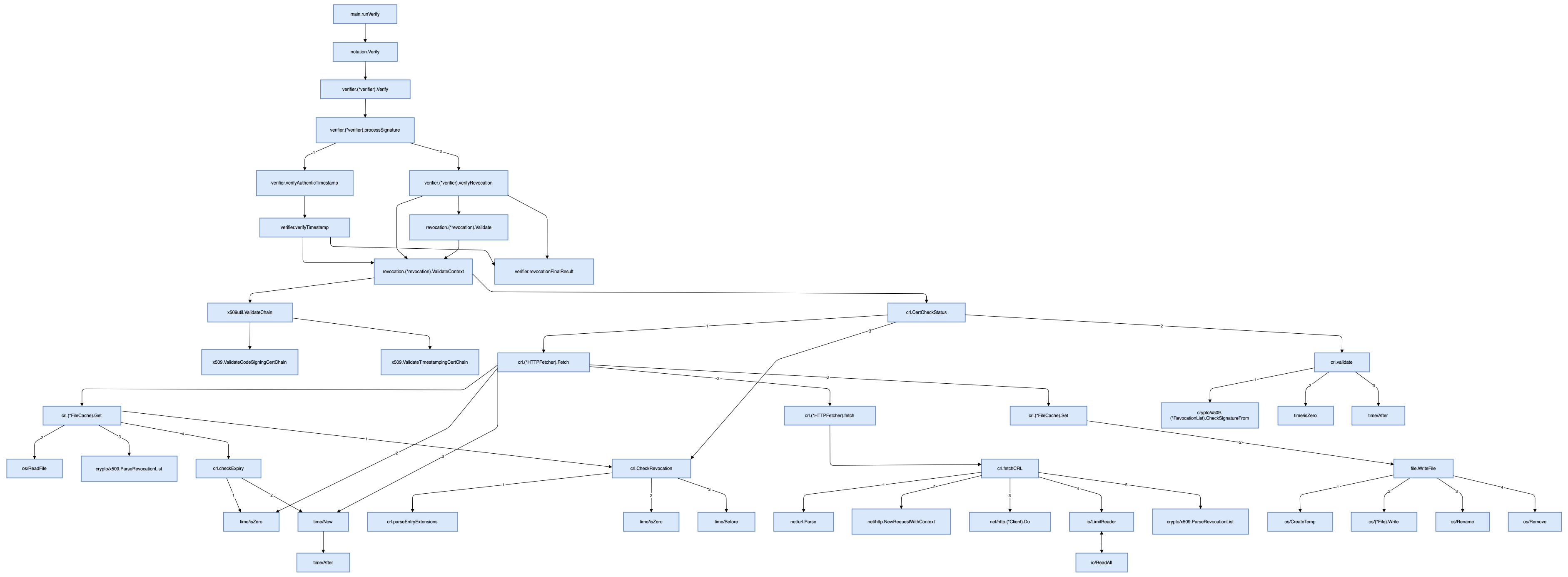
main.runSign
notation.Sign
signer.(*GenericSigner).Sign
base.(*Enveloppe).Sign
JWS
jws.(*enveloppe).Sign
jws.timestampJWS
timestamp.Timestamp
tspclient.NewRequest
tspclient.(*httpTimestamper).Timestamp
tspclient.(*TSTInfo).Validate
tspclient.(*SignedToken).Verify
x509.ValidateTimestampingCertChain
tspclient.generateNonce
tspclient.(*Request).Validate
net/http.(*Client).do
tspclient.(*Response).Validate
tspclient.(*TSTInfo).validate
cms.(*ParseSignedData).VerifySigner
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafCertificate
x509.isSelfSigned
x509.validateTimestampingCACertificate
crypto/rand.Int
tspclient.(*Response).SignedToken
tspclient.(*SignedToken).Info
tspclient.(*SignedToken).Info
tspclient.(*SignedToken).SigningCertificate
cms.(*ParseSignedData).verify
x509.validateLeafBasicConstraints
x509.validateTimestampingLeafKeyUsage
x509.validateTimestampingExtendedKeyUsage
x509.validateSignatureAlgorithm
x509.isIssuedBy
x509.validateCABasicConstraints
x509.validateTimestampingCAKeyUsage
main.prepareSigningOpts
tspclient.(*Response).validateStatus
tspclient.ParseSignedToken
cms.(*ParseSignedData).GetCertificate
crypto/x509.(*Certificate).Verify
cms.(*ParsedSignedData).verifySignedAttributes
crypto/x509.(*Certificate).CheckSignatureFrom
x509.validateTimestampingKeyUsagePresent
tspclient.NewHTTPTimestamper
x509.ReadCertificateFile
x509.IsRootCertificate
cms.ParseSignedData
cms.(*ParseSignedData).verifySignature
cms.isSigningTimeValid
url.Parse
crypto/x509.(*Certificate).CheckSignatureFrom
x509.parseCertificates
ber.ConvertToDER
crypto/x509.(*Certificate).CheckSignature
crypto/x509.ParseCertificate
ber.decode
ber.decodeMetadata
ber.isPrimitive
ber.decodeIdentifier
ber.decodeLength

C.B. Time-stamp Countersignature Verification



main.runVerify
1
2
main.getVerifier
notation.Verify
1
2
httputil.NewClient
crl.NewHTTPFetcher
verifier.(*verifier).Verify
verifier.(*verifier).processSignature
verifier.verifyAuthenticTimestamp
verifier.verifyTimestamp
1
2
3
4
5
6
7
8
9
10
11
12
13
verifier.isTSATrustStoreInPolicy
time/Now
time/After
tspclient.ParseSignedToken
tspclient.(*SignedToken).Info
tspclient.(*TSTInfo).Validate
verifier.loadX509TSATrustStores
tspclient.(*SignedToken).Verify
tspclient.(*Timestamp).BoundedAfter
tspclient.(*Timestamp).BounderBefore
revocation.(*revocation).Validate
verifier.revocationFinalResult
cms.ParseSignedData
x509.parseCertificates
tspclient.(*TSTInfo).validate
verifier.loadX509TrustStoresWithType
tspclient.(*SignedToken).SigningCertificate
cms.(*ParseSignedData).VerifySigner
ocsp.CheckStatus
ber.ConvertToDER
ber.isPrimitive
truststore.(*x509TrustStore).GetCertificates
cms.(*ParseSignedData).verify
x509util.ValidateChain
ocsp.CertCheckStatus
ber.decode
crypto/x509.(*Certificate).Verify
cms.
(*ParsedSignedData).verifySignedAttributes
x509.ValidateTimestampingCertChain
ocsp.checkStatusFromServer
ocsp.serverResultsToCertRevocationResult
ber.decodeMetadata
cms.(*ParseSignedData).verifySignature
cms.isSigningTimeValid
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafCertificate
x509.isSelfSigned
x509.validateTimestampingCACertificate
ber.decodeIdentifier
ber.decodeLength
crypto/x509.(*Certificate).CheckSignature
x509.validateTimestampingLeafKeyUsage
x509.validateTimestampingExtendedKeyUsage
x509.validateSignatureAlgorithm
x509.isIssuedBy
x509.validateLeafBasicConstraints
x509.validateCABasicConstraints
x509.validateTimestampingCAKeyUsage
crypto/x509.(*Certificate).CheckSignatureFrom
x509.validateTimestampingKeyUsagePresent

C.C. CRL Verification



Acronyms Index

CLI:	Command Line Interface
CNCF:	Cloud Native Computing Foundation
CRL:	Certificate Revocation List
OCI:	Open Container Initiative
OCSP:	Online Certificate Status Protocol
OSTIF:	Open Source Technology Improvement Fund, Inc.
TSA:	Time-Stamp Authority
TSP:	Time-Stamp Protocol
TST:	Time-Stamp Token