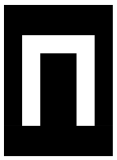
PRESENTS

Notation security audit

In collaboration with the Notation maintainers, Open Source Technology Improvement Fund and The Linux Foundation



Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 6th July 2023

Table of contents

Executive summary

In March and April 2023 Ada Logics carried out a security audit for Notation. The audit was a time-based engagement in collaboration with the Notation maintainers, OSTIF and the CNCF. The audit had four goals:

1. Formalize a threat model for the projects in scope.
2. Manually audit the code base in scope.
3. Assess the completeness of the fuzzing suite and improve it if necessary.
4. Carry out a SLSA review of the projects in scope.

These four goals each take a different approach to assessing Notations security posture; The threat model emphasized the importance of supply-chain specific threat vectors as well as clarifying the untrusted input sources. With these established, we assessed the code pragmatically against supply-chain attacks and security vulnerabilities which led us to finding 3 vulnerabilities of Low, Moderate and High severity. We also found a number of areas in Notations documentation that was lacking a security context, and we made a number of recommendations to Notations documentation to help users avoid using Notation insecurely.

Ada Logics made a number of improvements to Notations fuzzing suite, mostly related to the performance of its verification fuzzer. We improved its speed more than 10 times and pushed all improvements to its OSS-Fuzz integration, so that the fuzzer runs continuously with improved performance.

Notations SLSA review demonstrated compliance in most areas besides provenance generation. Since our review, SLSA v1.0 has been released, and we recommend the Notation team to look into working continuously on improving users' experience and security by following the SLSA standards.

Notable findings

The audit found 3 CVE's of low, moderate and high severity. These are included in the findings section of the report with the following issue ID's:

Issue ID	CVE	Advisory	Severity
ADA-NOT-23-2	CVE-2023-33958	GHSA-rvrx-rrwh-r9p6	Moderate
ADA-NOT-23-6	CVE-2023-33959	GHSA-xhg5-42rf-296r	High
ADA-NOT-23-7	CVE-2023-33957	GHSA-9m3v-v4r5-ppx7	Low

Project Summary

The auditors of Ada Logics were:

Name	Title	Email
Adam Korczynski	Security Engineer, Ada Logics	Adam@adalogics.com
David Korczynski	Security Researcher, Ada Logics	David@adalogics.com

The Notation community members involved in audit were:

Name	Title	Email
Feynman Zhou	Product Manager, Microsoft	feynmanzhou@microsoft.com
Junjie Gao	Engineer, Microsoft	junjiegao@microsoft.com
Pritesh Bandi	Engineer, Amazon	pritesb@amazon.com
Patrick Zheng	Engineer, Microsoft	patrickzheng@microsoft.com
Samir Kakkar	Sr. Product Manager Tech, AWS	iamsamir@amazon.com
Shiwei Zhang	Engineering Manager, Microsoft	shizh@microsoft.com
Toddy Mladenov	Product Manager, Microsoft	metodi.mladenov@microsoft.com
Vani Rao	Engineering Leader, AWS	vaninrao@amazon.com
Yi Zha	Product Manager, Microsoft	yizha1@microsoft.com

The following facilitators of OSTIF were engaged in the audit:

Name	Title	Email
Derek Zimmer	Executive Director, OSTIF	Derek@ostif.org
Amir Montazery	Managing Director, OSTIF	Amir@ostif.org

Audit Scope

The following assets were in scope of the audit.

Repository	https://github.com/notaryproject/notation
Language	Go

Repository	https://github.com/notaryproject/notation-go
Language	Go

Repository	https://github.com/notaryproject/notation-core-go
Language	Go

Threat model formalisation

In this section we outline our threat modelling of Notation. We first cover the dataflow of Notation where we draw on the ongoing threat modelling that Notation is doing. Next, we specify the threat actors that could have a harmful impact on a Notation use case. Finally, we exemplify several threat scenarios based on the observations we made in the architecture overview and the specified threat actors.

The threat modelling has been conducted based on the following public resources:

- Notations documentation including README files from the Notation repositories
- Notations source code
- Feedback from the Notation maintainers
- Open pull requests in all Notation repositories

We expect that the threat model evolves over time based on both how Notation and adoption evolves. As such, threat modelling should be seen as an ongoing effort. Future security disclosures to the Notation security team are opportunities to evaluate the threat model of the affected components.

The threat model in this report is meant to be helpful for different readers. First, the threat model is written as a reference for the Notation maintainers to reason about the security of Notation. This is particularly helpful when Notation receives vulnerability disclosures from the community, and the security team needs to evaluate the security criticality of reported issues. The second type of reader we address the threat model to are the adopters and contributors of Notation. Adopters and contributors experience bugs in their own deployments that they report upstream. Some of these bugs may have security relevance, and the threat model serves as a reference to consider the security criticality before reporting bugs. Finally, we address the threat model to security researchers that wish to contribute to the security of Notation by auditing the code and reporting vulnerabilities. For this group, the threat model aims to lower the bar of entry for researchers that are specialized in security research, but may not have the specific knowledge of the internals and use cases of Notation.

Architecture

Notation is currently working actively to develop the project's threat model. This is a community-driven effort that documents the core components of Notation and the dataflow between the core components and remote services. Notation’s own threat modelling also includes known use cases and the threat vectors that Notation faces. This is

a positive sign for the security of the project; Involving project maintainers in the threat modelling is a good security practice that allows the project to take advantage of the experience and field knowledge from developing the project in the threat modelling. It is also positive from the point of view that the process is ongoing which helps develop the threat model as the project matures.

At the time of the audit, Notations own threat modelling was taking place in this file: https://github.com/notaryproject/notaryproject/blob/7e4d84e3a7f52678fb42531c506d1cbdab32a736/threatmodels/notation-theatmodel.md[1]. The file includes dataflow charts and threat vectors. This is a good reference for someone unfamiliar with the Notation ecosystem to get an understanding of the architecture.

In this audit, Ada Logics used the threat model as outlined in notation-threatmodel.md as the starting point for our own threat modelling and code auditing. As such, Notations own threat model was tested, and this report includes the steps we took in auditing Notation from Notations own threat modelling efforts. It is our goal that by including the threat modelling we did as part of this audit, we offer feedback on how Notations threat model is perceived and used in the community.

To get an understanding of Notations architecture and dataflow, we refer to https://github.com/notaryproject/notaryproject/blob/7e4d84e3a7f52678fb42531c506d1cbdab32a736/threatmodels/notation-theatmodel.md.

[1] At the time of the audit being updated in this PR: https://github.com/notaryproject/notaryproject/pull/242

Threat actors

Threat actors are personas that could seek to cause harm to Notation users. When threat modelling, threat actors answer the question, “who?”, in different scenarios. For example, when a security vulnerability is identified, who could benefit from exploiting it? Or who could have motives to exploit it?
Threat actors are not always a threat to the system under analysis. They only become a threat if they assume a malicious role. For example, one threat actor we identify is the open source contributor. This is a valid actor in the Notation ecosystem, since Notation is an open source project; Not all open source contributors have malicious intentions, but they can have.

Actor #	Name	Description
1	Fully untrusted users	A user whose identity, motives and resources that a Notation user does not know.
2	Artifact vendor	A vendor of software artifacts that Notation users consume.
3	State-backed criminal organization	An organization backed by governmental agencies. Typically have either financial or political goals.
4	Open source contributor	A member of the open source community contributing to Notation or Notations 3rd party dependencies.
5	Users with limited access to the registry	A user who can perform one or several tasks against a registry but is not full admin of the registry.

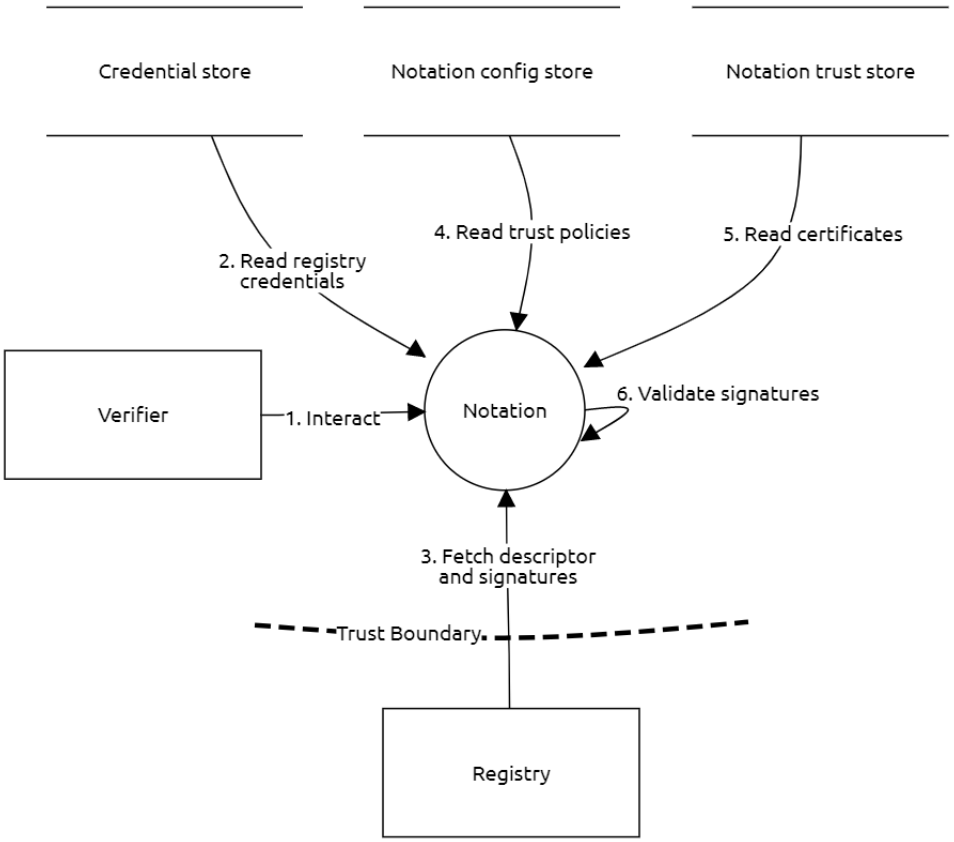
Trust boundaries

A trust boundary is a point in the architecture where trust changes vertically ie. increases or decreases. When a request or process enters a trusted system, its level of trust increases, whereas when it leaves a trusted system to a remote service, its level of trust decreases.
Trust boundaries are an important part of threat modelling a system, as they expose entry points that threat actors will look for vulnerabilities in to exploit. When the trust level is elevated - for example when an untrusted user sends a request to a trusted system - it is important that excessive privileges are not granted.

Notations own data flow charts include trust boundaries for several use cases. We include the following dataflow charts from https://github.com/notaryproject/notaryproject/blob/7e4d84e3a7f52678fb42531c506d1cbdab32a736/threatmodels/notation-theatmodel.md:

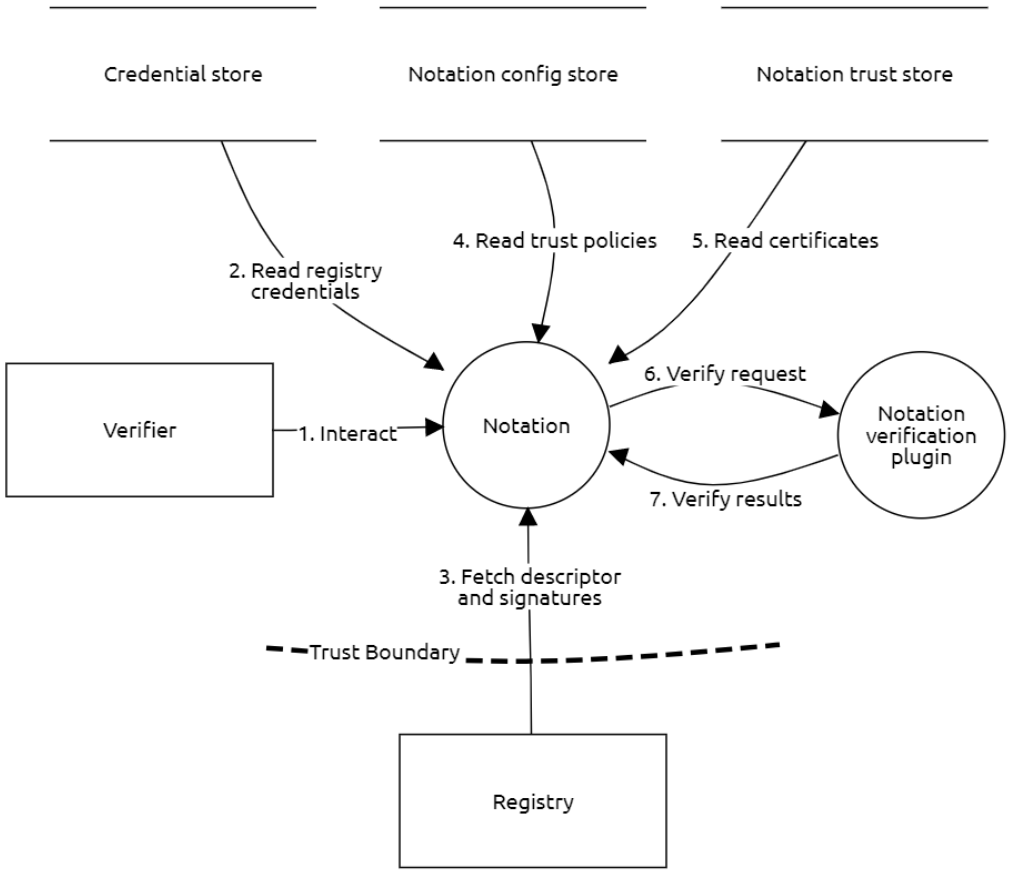
Verifying a remote artifact

In this scenario, the trust boundary is between the Notation deployment and the remote registry. This is a widely-used use case of Notation. The level of trust increases from the registry to Notation.



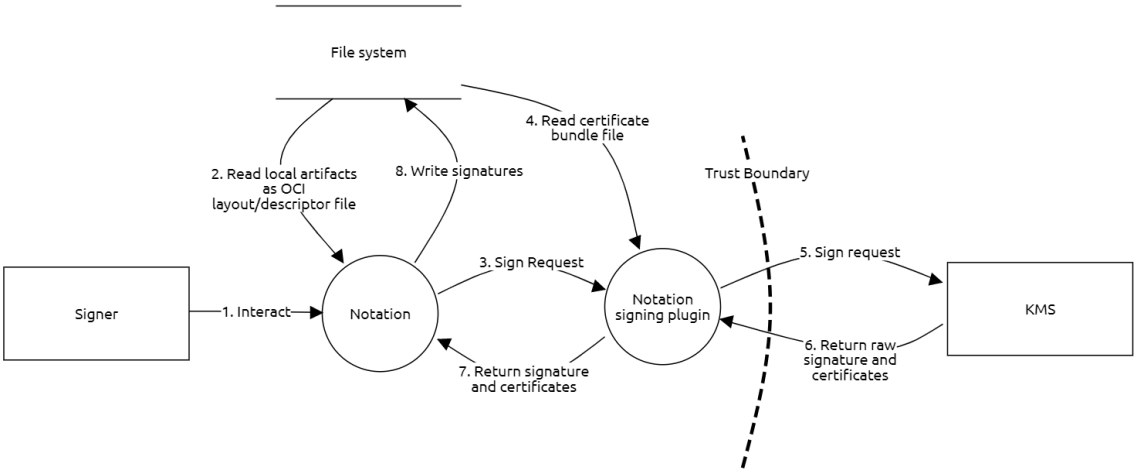
Verifying a remote artifact using a plugin

In this scenario, the trust boundary is between the Notation deployment and the remote registry. The level of trust increases in the direction from the registry to Notation.



Signing a local artifact using a remote KMS

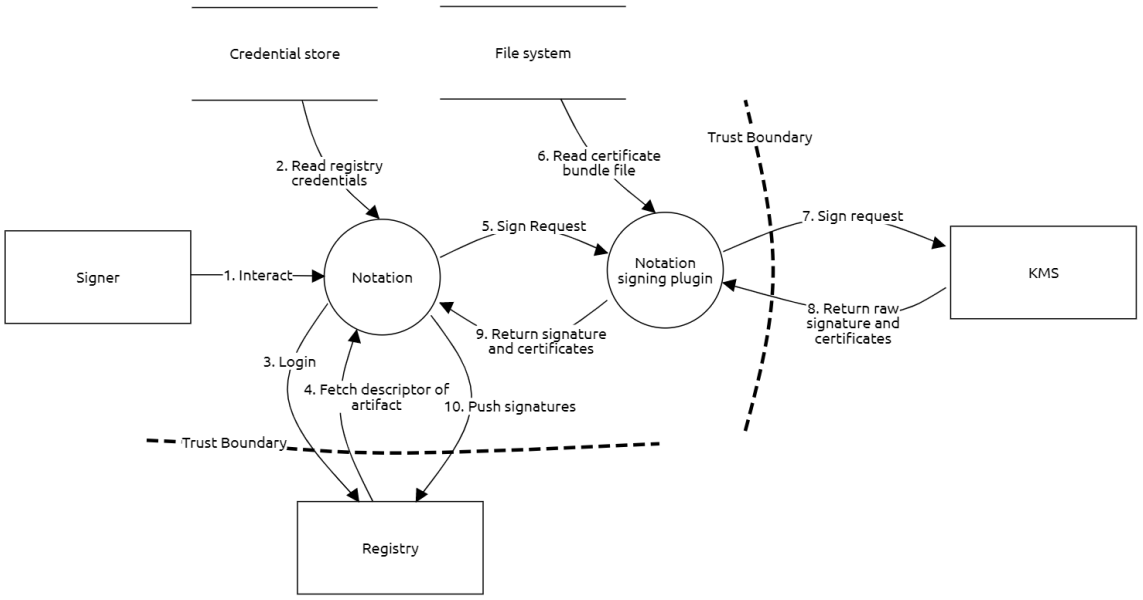
In this scenario, the trust boundary is between the Notation deployment and the remote key management service. The level of trust increases from the KMS to Notation.



Signing a remote artifact using a remote KMS

This scenario has two trust boundaries:

1. Between the remote registry and Notation. The trust flows from low to high in the direction from the remote registry to Notation.
2. Between the remote KMS and notation. The trust flows from low to high in the direction from the KMS to Notation.



Threat vectors

Notation is a tool to secure the software supply chain by allowing users to assign trust to software artifacts. As such, an important class of vulnerabilities of Notations attack vector is the class of software supply chain vulnerabilities. Notation-go and Notation are both exposed to these attacks. In particular, when Notation interacts with the registry, Notation is prone to supply-chain attacks. Notations verification routine of artifacts processes untrusted data and is particularly exposed. Below we enumerate some important supply-chain attack vectors for Notation.

Endless data attack

This is an attack where an attack tricks a client into downloading an endless stream of data when the client requests it. A successful endless data attack on Notation keeps Notation occupied on a task without ever finishing it which has the result of preventing Notation

from performing other tasks. This could for example prevent Notation from verifying an artifact; This, in turn, could prevent the user from deploying an artifact containing security fixes, and an attacker could keep the Notation user in a vulnerable state.

Rollback attacks
A rollback attack is where a threat actor is able to trick a client into installing older versions of a given software artifact. An attacker could launch a rollback attack on a Notation user to trick them into installing an artifact containing known security vulnerabilities and then exploit these vulnerabilities. Notation itself does not handle the installation of artifacts, and the attacker would attempt to trick the user into making wrong assumptions about an artifact that would cause the user to consume the wrong artifact.

Freeze attacks
This is an attack where an actor prevents a client from updating software artifacts. The threat actor does this by presenting files to the client that the client is already aware of and is tricked into concluding that there is no update to the artifact. A freeze attack can be enabled by a vulnerability that allows the actor to carry out a Man-in-the-Middle attack, where the actor intercepts requests between the client and the central repository, when the client checks for updates. Similarly to rollback attacks, the attack vector here against Notation users is that an attacker could trick the user into making wrong assumptions about an artifact and consume a different artifact than they should; For example, an attacker could prevent users from consuming artifacts with the latest security updates.

Arbitrary package attacks
An arbitrary package attack is where an attacker can trick a client into downloading a software artifact of the attackers choice. In a successful arbitrary package attack, the client will not be aware that it downloaded the wrong artifact. Notation does not install packages and the attacker would attempt to trick the user into installing a different package based on Notations response to the users.

Typosquatting attacks
A typosquatting attack is when an attacker tricks a client into downloading malicious software artifacts by typing a name that closely resembles the safe artifacts name but is still different.

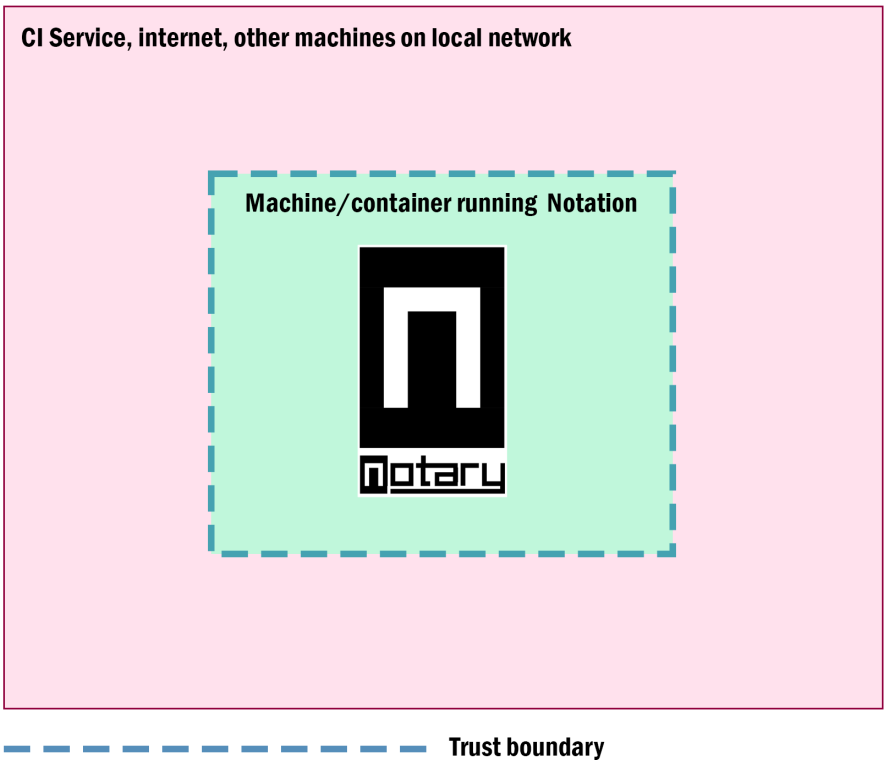
Other important attack vectors

Below we list other attack vectors that surfaced during the manual auditing of Notation. These were exposed from a bottom-up approach during the code auditing.

1. The file system: Notation grants a high level of trust to the file system, and obtaining control of the file system would allow an attacker to escalate privileges to the highest level and achieve remote code execution.
2. The trust policies: The trust policies are an important part of Notation to manage trust for artifacts, registries and vendors. They have a number of threat vectors that we include below.

Compromising the file system to obtain RCE privileges

If an attacker compromises the file system running Notation, they have the potential to escalate their privileges to running arbitrary code on the machine. This is for example enabled by Notations plug-in feature, that invokes binaries found in the `~/.config/notation/plugins directory`. Prior to invoking a plugin, Notation does not check the integrity of this plugin, and an attacker could replace the file with a malicious binary leading to remote code execution.



This is important for the Notation user and Notation itself. If users deploy Notation in an environment in which lower privileged users have access to modify the file system, then Notation is deployed insecurely. If Notation itself allows users to gain control over the file system, then Notation is allowing users to escalate privileges.

A malicious user can launch other attacks with control over the file system. This could be:

- Replacing binaries being invoked on the file system, for example plugins.
- Adding new plugins to the plugin manager
- Replace files in the user configuration directory.
- Read secrets such as the Docker credentials file.

Trust policies

A trust policy is a policy document in JSON format with a set of rules and actions Notation Should take based on the properties extracted about an artifact. A trust policy can for example specify that a consumer of a software artifact only wishes to consume artifacts signed by a certain entity.

A trust policy is required to specify a signature verification level which defines the level of validation. The levels are specified in detail here: https://github.com/notaryproject/notaryproject/blob/v1.0.0-rc.2/specs/trust-store-trust-policy.md#signature-verification, and for ease of access we enumerate them below:

- Strict level: Enforces all validation steps.
- Permissive: Enforces most validation steps.
- Audit: Enforces signature integrity validation if a signature exists.
- Skip: Skips signature verification.

Trust policy validation
Notation carry out a series of validations based on the level specified in the trust policy. The validations are described in detail here: https://github.com/notaryproject/notaryproject/blob/v1.0.0-rc.2/specs/trust-store-trust-policy.md#signature-verification.

The following table presents the validation steps of each verification level:

Signature Verification Level	Recommended Usage	Validations				
		Integrity	*Authenticity*	*Authentic timestamp*	*Expiry*	*Revocation check*
strict	Use at development, build and deploy time	enforced	enforced	enforced	enforced	enforced
permissive	Use at deploy time or runtime	enforced	enforced	logged	logged	logged
audit	Use when adopting signed images, without breaking existing workflows	enforced	logged	logged	logged	logged
skip	Use to exclude verification for unsigned images	skipped	skipped	skipped	skipped	skipped

The Notation documentation specifies each level as such:

Integrity: Guarantees that the artifact wasn't altered after it was signed, or the signature isn't corrupted.
Authenticity: Guarantees that the artifact was signed by an identity trusted by the verifier.
Authentic timestamp: Guarantees that the signature was generated when the certificate was valid.
Expiry: This is an optional feature that guarantees that the artifact is within "best by use" date indicated in the signature.
Revocation check: Guarantees that the signing identity is still trusted at signature verification time.

We note that the Notation documentation says "Guarantees" in each validation level. From the perspective of Notations goal - securing the software supply chain - Notation must guarantee the goal of each validation. Each validation step protects consumers of software artifacts from different attack vectors in the software supply chain.

A malicious actor could seek to bypass a trust policy and trick the consumer into an assumption that is incorrect and that puts the consumer at risk. The most dangerous one is tricking the consumer into trusting a unapproved artifact by making it appear as a trusted artifact. This could allow a malicious actor to trick consumers into consuming a malicious artifact.

It is important that Notation correctly implements the validation enforcement for each level. A mistake in the validation routines in Notation that allows a malicious actor to circumvent any of the guaranteed qualities of the validation processes is a security issue. A malicious actor would ultimately achieve this by putting Notation in a state where a validation routine should disallow a given artifact based on a trust policy, but where the validation routine does not do that.

When running `notation verify`, notation retrieves a trust policy document from the local file system. This document is used to determine the skip level, and if an attacker could compromise the file system and replace the trust policy document, they could ultimately control the skip level Should an attacker be able to control the verifier, then they could set the level to the most permissive and thus skip all validation.

Fuzzing

Ada Logics assessed Notations fuzzing suite to find areas to be improved. We looked at OSS-Fuzz's dashboard[2] which contains in-depth performance details of Notations fuzzers. Notations verification fuzzer was running at a low execution speed with an average of 164.9 executions per second[3]:

libFuzzer_notary_FuzzSignatureCose	Performance	249,967,761	0	--	--	4196 (17 MB)	6,206.7
libFuzzer_notary_FuzzSignatureJws	Performance	248,097,156	0	--	--	2656 (21 MB)	5,885.1
libFuzzer_notary_FuzzVerify	Performance	7,955,871	0	--	--	2350 (82 MB)	164.9
libFuzzer_notary_fuzz	Performance	284,982,945	0	--	--	587 (30 MB)	3,452.1

This fuzzer targets `github.com/notaryproject/notation-go.Verify()` which is exposed to untrusted input. As such, it is important that this fuzzer runs efficiently and at a high speed. Before it calls the target API, `github.com/notaryproject/notation-go.Verify()`, the fuzzer creates an artifact reference, a trustpolicy document and verification options. Ada Logics rewrote the fuzzer to do each of these things in a separate goroutine instead of one after the other. That made the fuzzer 10 times faster[4]:

libFuzzer_notary_FuzzSignatureCose	Performance	540,547,080	0	--	--	4196 (17 MB)	6,478.4
libFuzzer_notary_FuzzSignatureJws	Performance	902,379,084	0	--	--	2656 (21 MB)	6,157.3
libFuzzer_notary_FuzzVerify	Performance	272,266,649	0	--	--	2350 (82 MB)	1,751.3
libFuzzer_notary_fuzz	Performance	444,488,929	0	--	--	587 (30 MB)	2,763.3

The work on improving execution speed spanned the following pull requests:

1. https://github.com/cncf/cncf-fuzzing/pull/332
2. https://github.com/cncf/cncf-fuzzing/pull/335

Ada Logics also rewrote the fuzzer to avoid using a skip verifier. A skip verifier skips verification of an artifact and is unnecessary to waste execution cycles on. While this does not improve execution speed, it does make the fuzzer progress to verification after its expensive initialization. As such, it improves execution speed of meaningful code paths. The work on removing the skip verifier was made in the following pull request:

---

2 https://oss-fuzz.com/fuzzer-stats?project=notary&fuzzer=libFuzzer&job=libfuzzer_asan_notary&group_by=by-fuzzer

3 https://oss-fuzz.com/fuzzer-stats?group_by=by-fuzzer&date_start=2023-03-23&date_end=2023-03-23&fuzzer=libFuzzer&job=libfuzzer_asan_notary&project=notary

4 https://oss-fuzz.com/fuzzer-stats?group_by=by-fuzzer&date_start=2023-03-26&date_end=2023-03-27&fuzzer=libFuzzer&job=libfuzzer_asan_notary&project=notary

1. https://github.com/cncf/cncf-fuzzing/pull/334

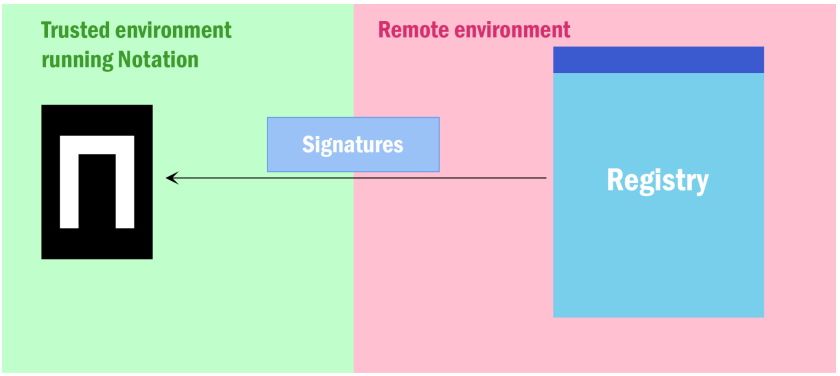
Issues found

In this section we present the security issues found during the manual auditing part of the security audit.

We found multiple attack vectors in the scenario where an attacker compromises a registry and can control the number of signatures of a given artifact. The registry is untrusted in Notations threat model, and Notation should be guarded against any attack vector from it.

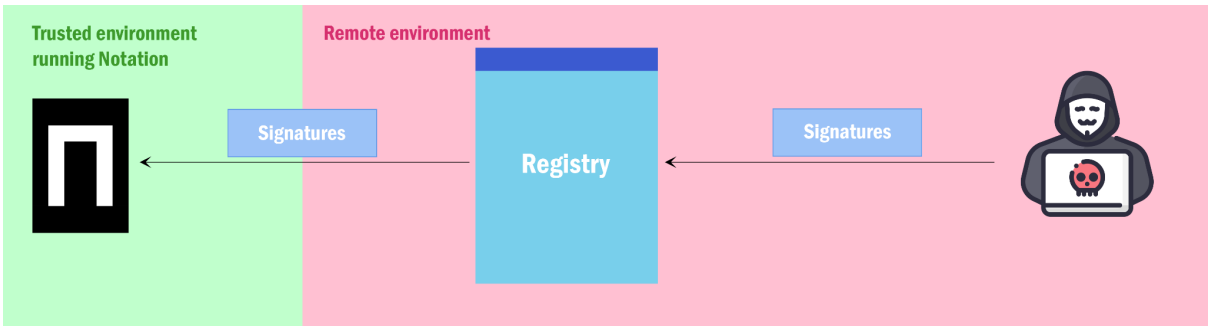
We can illustrate this attack vector as such:

This diagram represents a valid use case for Notation:



Notation retrieves the signatures of an artifact from a registry to verify the artifact or inspect or list the signatures manually. Notation does this under the hood during operations such as verifying an artifact.

In case an attacker compromises the registry, they can launch a number of attacks against Notation that Notation was not guarded against. Notation has a wide attack surface in case an attacker is able to add unlimited signatures to the registry:



The scenario here is that an attacker compromises a registry in a manner that they are able to add unlimited signatures to the registry. This scenario gives an attacker control over the Notation users supply-chain in the following ways:

- Endless data attack: The attacker can cause Notation to loop through signatures endlessly.
- Freeze attack: In certain operations, Notation sets a limit on the number of artifacts it will process. By adding a large amount of signatures to the registry, the attacker can cause Notation to stop looping through the signatures at the maximum allowed number, and block Notation from verifying the correct signature, thus blocking Notation from verifying a verifiable artifact.
- Blocking fetching of signatures. In some deployments, a large amount of signatures from the registries cannot be fetched by Notation from HTTP timeouts or from sending too large a body. As such, an attacker can block Notation from fetching any of the signatures which can result in another freeze attack vector.
- The above freeze attacks can result in a rollback attack, if the Notation user sets up their deployment to fetch the latest verifiable version of a given software artifact.

This issue is not isolated to Notation but also affects Notations own supply-chain. From a high level, Notation depends on an ecosystem that can be compromised or might not have security measures that Notation users can depend on.

Ada Logics found 3 security issues related to the scenario, where an attacker controls the registry. These are listed in the table below and are detailed in the “Issues found” section later in the report.

Issues related to signature processing

ID	Title	Attack vector
ADA-NOT-23-1	Potential endless data attack in `notation ls`	Endless data attack
ADA-NOT-23-2	Max number signatures allowed enables an endless data attack	Endless data attack
ADA-NOT-23-7	Denial of Service From Resource Exhaustion in `notation inspect`	Resource exhaustion

All issues found

#	ID	Title	Severity	Fixed
1	ADA-NOT-23-1	Potential endless data attack in `notation ls`	Moderate	Yes
2	ADA-NOT-23-2	Max number signatures allowed enables an endless data attack	Moderate	Yes
3	ADA-NOT-23-3	Overwriting global variable	Informational	Yes
4	ADA-NOT-23-4	Insufficient security-relevant documentation	Low	Yes
5	ADA-NOT-23-5	Cleartext Storage of Sensitive Information in an Environment Variable	Low	Yes
6	ADA-NOT-23-6	Insufficient Verification of Fetched Artifact Descriptor	High	Yes
7	ADA-NOT-23-7	Denial of Service From Resource Exhaustion in `notation inspect`	Low	Yes

ADA-NOT-23-1: Potential endless data attack in `notation ls`

ID	ADA-NOT-23-1
Component	Notation ls
Severity	Moderate
Fixed in: https://github.com/notaryproject/notation/commit/ed22fde52f6d70ae0b53521bd28c9ccafa868c24	

Notation is susceptible to a potential endless data attack in the `notation ls` command if the artifact has a large amount of signatures. The issue exists in the following lines of code:

https://github.com/notaryproject/notation/blob/2e56dd42e385ee1568c5e13e6ef38edb2a549500/cmd/notation/list.go#L90-L102

```
	err := sigRepo.ListSignatures(ctx, targetDesc, func(signatureManifests
[]ocispec.Descriptor) error {
		for _, sigManifestDesc := range signatureManifests {
			if prevDigest != "" {
				// check and print title
				printTitle()

				// print each signature digest
				fmt.Printf("    ├── %s\n", prevDigest)
			}
			prevDigest = sigManifestDesc.Digest
		}
		return nil
	})
```

If `sigRepo.ListSignatures()` returns a high number of signatures, notation could be forced to loop for a long time resulting in an endless data attack.

ADA-NOT-23-2: Max number signatures allowed enables an endless data attack

ID	ADA-NOT-23-2
Component	Notation verify
Severity	Moderate
Fixed in: https://github.com/notaryproject/notation/commit/ed22fde52f6d70ae0b53521bd28c9ccafa868c24	

This issue has been assigned CVE-2023-33958. Notations advisory can be found here: GHSA-rvrx-rrwh-r9p6

Notation uses a high number for the maximum number of allowed signatures that it will verify. This exposes Notation to an endless data attack, since Notation loops through all signatures until it reaches the limit.

Notation specifies the max signature attempts here: https://github.com/notaryproject/notation/blob/4589c835eeabed221f6118db7eb69a769f529496/cmd/notation/verify.go#L21

```
const maxSignatureAttempts = math.MaxInt64
```

This number is equal to 9223372036854775807

Local testing shows that to execute a simple Golang loop 9223372036854775807 times will take 310 years on an 8-core machine.

If an attacker is able to flood a registry with signatures, they could launch an endless data attack on Notation which in turn could prevent upgrading a given artifact.

ADA-NOT-23-3: Overwriting global variable

ID	ADA-NOT-23-3
Component	Notation verify
Severity	Informational
Fixed in: https://github.com/notaryproject/notation/pull/676	

Notation overwrites a global import identifier in the verification command. There is no current way to exploit this issue, but it could lead to undefined behavior of Notation in the future, if a contributor adds code that allows an attacker to trigger an issue. The issue is flagged informational since we have found no attack vector.

```
	ref, err := resolveReference(command.Context(), &opts.SecureFlagOpts,
reference, sigRepo, func(ref registry.Reference, manifestDesc
ocispec.Descriptor) {
		fmt.Fprintf(os.Stderr, "Warning: Always verify the artifact using
digest(@sha256:...) rather than a tag(:%s) because resolved digest may not point
to the same signed artifact, as tags are mutable.\n", ref.Reference)
	})
	if err != nil {
		return err
	}

	// initialize verifier
	verifier, err := verifier.NewFromConfig()
	if err != nil {
		return err
	}
```

… with Notation declaring the following import in the same file:

```
import (
	"context"
	"errors"
	"fmt"
	"math"
	"os"
	"reflect"

	"github.com/notaryproject/notation-go"
	notationregistry "github.com/notaryproject/notation-go/registry"
	"github.com/notaryproject/notation-go/verifier"
	"github.com/notaryproject/notation-go/verifier/trustpolicy"
```

```
        "github.com/notaryproject/notation/internal/cmd"
        "github.com/notaryproject/notation/internal/ioutil"
        ocispec "github.com/opencontainers/image-spec/specs-go/v1"

        "github.com/spf13/cobra"
        "oras.land/oras-go/v2/registry"
)
```

ADA-NOT-23-4: Insufficient security-relevant documentation

ID	ADA-NOT-23-4
Component	Notation
Severity	Low
Fixed in: • https://github.com/notaryproject/notaryproject.dev/pull/219 • https://github.com/notaryproject/notaryproject.dev/pull/228 • https://github.com/notaryproject/notaryproject.dev/pull/270	

The security-relevant documentation for Notation is lacking. Security-relevant documentation helps adopters use a product securely and understand the trade-offs when using it insecurely. We recommend adding a security best practices page to the Notation documentation that outlines details on securely using notation. In particular, we suggest documenting the following point:

1: What is a secure deployment?
The audit found several threats against a Notation deployment if notation was to be deployed carelessly. For example, the threat model in this report highlights that a compromise of the file system in which Notation is deployed is a major security risk. As such, users should be aware of the risks associated with slacking on the security of the file system in their usage.

2: Writing secure verification plugins
Notation invokes verification plugins as binaries which is a design susceptible to numerous security risks. Untrusted data is passed to the plugins which means their attack surface is exposed to a potential attacker. The plugins should be documented such that the security trade-offs are clear to plugin developers.

3: Security-critical flag
Ada Logics found that the documentation for the PlainHTTP setting is lacking:
https://github.com/notaryproject/notation/blob/a08dc9e6459bf8e5cd80d550e4797904937dcab6/cmd/notation/common.go#L47

```
type SecureFlagOpts struct {
	Username  string
	Password  string
	PlainHTTP bool
}
```

This setting is security critical but is neither documented or warned against. PlainHTTP causes Notation to communicate over http instead of https which makes the Notation user susceptible to a MitM attack. As such, it should only be used when the usage and deployment of Notation mitigates this kind of attack. Notation should make this clear in the documentation and if possible inline in the code as well.

ADA-NOT-23-5: Cleartext Storage of Sensitive Information in an Environment Variable

ID	ADA-NOT-23-5
Component	Notation
Severity	Moderate
Fixed in: https://github.com/notaryproject/notaryproject.dev/pull/223	

Notation stores username and password in plaintext format in environment variables. This means the credentials are exposed to a wider audience than intended, e.g. an internal attacker with a host position will be able to watch credentials through various utilities such as via `ps` or by printing out the environment variables. This could lead to credentials being stolen if someone has access to the host at which the command line was entered but not the access to the token of a given Github, as then the details can be leaked.

In the Kubernetes Security audit of 2019 this type of issue was classified Medium in severity (https://github.com/kubernetes/sig-security/blob/main/sig-security-external-audit/security-audit-2019/findings/Kubernetes%20Final%20Report.pdf finding ID: TOB-K8S-005) and to keep consistency we follow this severity.

Username and password stored in ENV variables here: https://github.com/notaryproject/notation/blob/a08dc9e6459bf8e5cd80d550e4797904937dcab6/cmd/notation/common.go#L56

Username and password are passed in plain text here: https://github.com/notaryproject/notation/blob/a08dc9e6459bf8e5cd80d550e4797904937dcab6/cmd/notation/common.go#L56

ADA-NOT-23-6: Insufficient Verification of Fetched Artifact Descriptor

ID	ADA-NOT-23-6
Component	Notation-go Verify
Severity	High
Fixed in: • https://github.com/notaryproject/notation-go/pull/313 • https://github.com/notaryproject/notation-go/pull/317 • https://github.com/notaryproject/notation/pull/689	

This issue has been assigned CVE-2023-33959. Notation-go's security advisory can be found here: GHSA-xhg5-42rf-296r

Notation-go is lacking sufficient verification for descriptors fetched from a remote registry. An attacker that controls the registry could trick Notation into providing the wrong verification outcome. The attacker could leverage this to trick the user into believing that the artifact is trusted when it is not, which in turn could cause the user the consume a malicious artifact.

The root cause of the issue is that Notation-go lacks verification for the descriptors. Notation validates the digest of the fetched descriptor during verification on the highlighted line below (green).

https://github.com/notaryproject/notation-go/blob/67a477f0c6d0054cb54f3062fe62a76de6882a63/notation.go#L319-L327

```
	artifactDescriptor, err := repo.Resolve(ctx, ref.Reference)
	if err != nil {
		return ocispec.Descriptor{}, nil,
ErrorSignatureRetrievalFailed{Msg: err.Error()}
	}
	if ref.ValidateReferenceAsDigest() != nil {
		// artifactRef is not a digest reference
		logger.Infof("Resolved artifact tag `%s` to digest `%s` before
verification", ref.Reference, artifactDescriptor.Digest.String())
		logger.Warn("The resolved digest may not point to the same signed
artifact, since tags are mutable")
	}
```

This validation checks whether the fetched descriptor has the same digest as the requested reference. Notation makes the assumption here that if digest validation fails, then there is a security issue, and Notation logs the reference and the descriptors digest. This is done on the highlighted lines above (purple).
Notation performs a similar check in the signing routine:
https://github.com/notaryproject/notation-go/blob/67a477f0c6d0054cb54f3062fe62a76de6882a63/notation.go#L101-L105

```
	targetDesc, err := repo.Resolve(ctx, artifactRef)
	if err != nil {
		return ocispec.Descriptor{}, fmt.Errorf("failed to resolve reference: %w", err)
	}
	if artifactRef != targetDesc.Digest.String() {
		// artifactRef is not a digest reference
		logger.Warnf("Always sign the artifact using digest(`@sha256:...`) rather than a tag(`:%s`) because tags are mutable and a tag reference can point to a different artifact than the one signed", artifactRef)
		logger.Infof("Resolved artifact tag `%s` to digest `%s` before signing", artifactRef, targetDesc.Digest.String())
	}
```

The condition, `if ref.ValidateReferenceAsDigest() != nil {` only checks if a digest is valid, and based on the result, Notation makes the assumption that Notation might have fetched a descriptor that is different from the requested one. However, Notation only checks if the digest of the descriptor is a valid digest, not if it is the same digest that the user is verifying. The allows an attacker to cause Notation to successfully verify a malicious artifact using the following steps:

1. Notation initiates the verification of a malicious artifact.
2. The registry does not return the malicious descriptor which is the one Notation should receive. Instead, the registry returns a descriptor containing valid signatures.
3. Notation validates the descriptor with a successful outcome.

We recommend:

1. Checking the equality between the reference and the descriptors digest. This does not prevent an attack but limits the threat vector.
2. Do byte content verification against the fetched descriptors digest.

ADA-NOT-23-7: Denial of Service From Resource Exhaustion in `notation inspect`

ID	ADA-NOT-23-7
Component	Notation verify
Severity	Low
Fixed in: https://github.com/notaryproject/notation/commit/ed22fde52f6d70ae0b53521bd28c9ccafa868c24	

This issue has been assigned CVE-2023-33957. Notations advisory can be found here: GHSA-9m3v-v4r5-ppx7

When inspecting the signatures of an artifact with `notation inspect`, Notation creates an `inspectOutput{}`. If Notation does not skip a signature it adds it to the `inspectOutput{}`, and once Notation has considered all signatures, it returns the `inspectOutput{}` to the user.

Notation creates the inspectOutput{} before the `err = sigRepo.ListSignatures()` loop: https://github.com/notaryproject/notation/blob/d62fd58434aa299b562cf6279b243364c253b10c/cmd/notation/inspect.go#L117

```
	if err := ref.ValidateReferenceAsDigest(); err != nil {
		fmt.Fprintf(os.Stderr, "Warning: Always inspect the artifact using digest(@sha256:...) rather than a tag(:%s) because resolved digest may not point to the same signed artifact, as tags are mutable.\n", ref.Reference)
		ref.Reference = manifestDesc.Digest.String()
	}

	output := inspectOutput{MediaType: manifestDesc.MediaType, Signatures: []signatureOutput{}}
	skippedSignatures := false
	err = sigRepo.ListSignatures(ctx, manifestDesc, func(signatureManifests []ocispec.Descriptor) error {
		for _, sigManifestDesc := range signatureManifests {
			sigBlob, sigDesc, err := sigRepo.FetchSignatureBlob(ctx, sigManifestDesc)
			if err != nil {
				fmt.Fprintf(os.Stderr, "Warning: unable to fetch signature %s due to error: %v\n", sigManifestDesc.Digest.String(), err)
				skippedSignatures = true
				continue
```

```
                }
```

And appends the signatures inside the `err = sigRepo.ListSignatures()` loop:

```
      err = sigRepo.ListSignatures(ctx, manifestDesc, func(signatureManifests
[]ocispec.Descriptor) error {
            for _, sigManifestDesc := range signatureManifests {
                  sigBlob, sigDesc, err := sigRepo.FetchSignatureBlob(ctx,
sigManifestDesc)
                  ...
                  output.Signatures = append(output.Signatures, sig)
            }
            return nil
      })
```

This can cause Notation to exhaust memory if `output.Signatures` grows sufficiently large. In such an event, Go will kill Notation when it exhausts the memory of the machine which could cause other running Go services on the machines to experience a short denial of service.

SLSA review

In this section we present our findings from our SLSA compliance review of Notation.

SLSA is a framework for assessing artifact integrity and ensure a secure supply chain for downstream users.

In this part of the audit, we assessed Notations SLSA compliance by following SLSA's v0.1 requirements[5]. This version of the SLSA standard is currently in alpha and is likely to change.

Our assessment shows Notations current level of compliance. We include our assessment for each of the three repositories in scope in this audit.

Overview

Notation

https://github.com/notaryproject/notation

Notation manages its source code on Github which makes it version controlled and possible to verify the commit history. The source code is retained indefinitely and all commits are verified by two different maintainers.
The build is fully scripted and is invoked via Notations Makefile. The build runs in Github Actions which provisions the build environment for building Notation and does not reuse it for other purposes. Github actions are not fully isolated, in that the build can access env var mounted secrets. The build is also not fully hermetic, since it runs with network access, which Notation needs to pull in dependencies at build time.
Notation lacks the provenance statement, and this is the area where Notation can improve the most. Notation can achieve level 1 SLSA compliance by:

1. Making the provenance statement available with releases.
2. Including the builder, artifacts and build instructions in the provenance.

The build instructions are the highest level of entry, which in Notation case is the command that invokes the Makefile.

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Source - Version controlled		✓	✓	✓

[5] https://slsa.dev/spec/v0.1/requirements

Source - Verified history			✓	✓
Source - Retained indefinitely				✓
Source - Two-person reviewed				✓
Build - Scripted build	✓	✓	✓	✓
Build - Build service		✓	✓	✓
Build - Build as code			✓	✓
Build - Ephemeral environment			✓	✓
Build - Isolated			⛔	⛔
Build - Parameterless				✓
Build - Hermetic				⛔
Build - Reproducible				✓
Provenance - Available	⛔	⛔	⛔	⛔
Provenance - Authenticated		⛔	⛔	⛔
Provenance - Service generated		⛔	⛔	⛔
Provenance - Non-falsifiable			⛔	⛔
Provenance - Dependencies complete				⛔
Provenance - Identifies artifact	⛔	⛔	⛔	⛔
Provenance - Identifies builder	⛔	⛔	⛔	⛔
Provenance - Identifies build instructions	⛔	⛔	⛔	⛔
Provenance - Identifies source code		⛔	⛔	⛔
Provenance - Identifies entry point			⛔	⛔
Provenance - Includes all build parameters			⛔	⛔
Provenance - Includes all transitive dependencies				⛔
Provenance - Includes reproducible info				⛔
Provenance - Includes metadata	⛔	⛔	⛔	⛔
Common - Security	Not defined by SLSA requirements			
Common - Access				✓
Common - Superusers				✓

Notation-go
https://github.com/notaryproject/notation-go

Notation-go manages its source code on Github which makes it version controlled and possible to verify the commit history. The source code is retained indefinitely and all commits are verified by two different maintainers.
Notation-go does not invoke a scripted build or a builder for releases.
Releases only include source codes and no provenance statement.

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Source - Version controlled		✓	✓	✓
Source - Verified history			✓	✓
Source - Retained indefinitely				✓
Source - Two-person reviewed				✓
Build - Scripted build	⛔	⛔	⛔	⛔
Build - Build service		⛔	⛔	⛔
Build - Build as code			⛔	⛔
Build - Ephemeral environment			⛔	⛔
Build - Isolated			⛔	⛔
Build - Parameterless				⛔
Build - Hermetic				⛔
Build - Reproducible				⛔
Provenance - Available	⛔	⛔	⛔	⛔
Provenance - Authenticated		⛔	⛔	⛔
Provenance - Service generated		⛔	⛔	⛔
Provenance - Non-falsifiable			⛔	⛔
Provenance - Dependencies complete				⛔
Provenance - Identifies artifact	⛔	⛔	⛔	⛔
Provenance - Identifies builder	⛔	⛔	⛔	⛔
Provenance - Identifies build instructions	⛔	⛔	⛔	⛔

Provenance - Identifies source code		⛔	⛔	⛔
Provenance - Identifies entry point			⛔	⛔
Provenance - Includes all build parameters			⛔	⛔
Provenance - Includes all transitive dependencies				⛔
Provenance - Includes reproducible info				⛔
Provenance - Includes metadata	⛔	⛔	⛔	⛔
Common - Security	Not defined by SLSA requirements			
Common - Access				✓
Common - Superusers				✓

Notation-core-go

https://github.com/notaryproject/notation-core-go

Notation-core-go manages its source code on Github which makes it version controlled and possible to verify the commit history. The source code is retained indefinitely and all commits are verified by two different maintainers.
Notation-core-go does not invoke a scripted build or a builder for releases.
Releases only include source codes and no provenance statement.

Requirement	SLSA 1	SLSA 2	SLSA 3	SLSA 4
Source - Version controlled		✓	✓	✓
Source - Verified history			✓	✓
Source - Retained indefinitely				✓
Source - Two-person reviewed				✓
Build - Scripted build	⛔	⛔	⛔	⛔
Build - Build service		⛔	⛔	⛔
Build - Build as code			⛔	⛔
Build - Ephemeral environment			⛔	⛔
Build - Isolated			⛔	⛔
Build - Parameterless				⛔

Build - Hermetic				⛔
Build - Reproducible				⛔
Provenance - Available	⛔	⛔	⛔	⛔
Provenance - Authenticated		⛔	⛔	⛔
Provenance - Service generated		⛔	⛔	⛔
Provenance - Non-falsifiable			⛔	⛔
Provenance - Dependencies complete				⛔
Provenance - Identifies artifact	⛔	⛔	⛔	⛔
Provenance - Identifies builder	⛔	⛔	⛔	⛔
Provenance - Identifies build instructions	⛔	⛔	⛔	⛔
Provenance - Identifies source code		⛔	⛔	⛔
Provenance - Identifies entry point			⛔	⛔
Provenance - Includes all build parameters			⛔	⛔
Provenance - Includes all transitive dependencies				⛔
Provenance - Includes reproducible info				⛔
Provenance - Includes metadata	⛔	⛔	⛔	⛔
Common - Security	Not defined by SLSA requirements			
Common - Access				✓
Common - Superusers				✓