ADA LOGICS

PRESENTS

Notary Fuzzing Audit

In collaboration with the Notary project maintainers and The Linux Foundation

Authors

Adam Korczynski <adam@adalogics.com>
David Korczynski <david@adalogics.com>
Date: 21st March, 2023

CNCF security and fuzzing audits

This report details a fuzzing audit commissioned by the CNCF and the engagement is part of the broader efforts carried out by CNCF in securing the software in the CNCF landscape. Demonstrating and ensuring the security of these software packages is vital for the CNCF ecosystem and the CNCF continues to use state of the art techniques to secure its projects as well as carrying out manual audits. Over the last handful of years, the CNCF has been investing in security audits, fuzzing and software supply chain security that has helped proactively discover and fix hundreds of issues.

Fuzzing is a proven technique for finding security and reliability issues in software and the efforts so far have enabled fuzzing integration into more than twenty CNCF projects through a series of dedicated fuzzing audits. In total, more than 350 bugs have been found through fuzzing of CNCF projects. The fuzzing efforts of the CNCF have focused on enabling continuous fuzzing of projects to ensure continued security analysis, which is done by way of the open source fuzzing project OSS-Fuzz[1].

The CNCF continues work in this space and will further increase investment to improve security across its projects and community. The focus for future work is integrating fuzzing into more projects, enabling sustainable fuzzer maintenance, increasing maintainer involvement and enabling fuzzing to find more vulnerabilities in memory safe languages. Maintainers who are interested in getting fuzzing integrated into their projects or have questions about fuzzing are encouraged to visit the dedicated cncf-fuzzing repository https://github.com/cncf/cncf-fuzzing where questions and queries are welcome.

[1] https://github.com/google/oss-fuzz

Executive summary

In this fuzzing audit, Ada Logics worked on setting up a fuzzing suite for the Notary Project. At the time of this engagement, the Notary Project was not integrated into OSS-Fuzz, and the goal of the audit was to first integrate it and build upon this integration and improve the fuzzing efforts in a continuous manner.

The fuzzing audit was carried out in collaboration with the maintainers of all code assets in scope. The audit spanned several months at the end of 2022 and early 2023.

The fuzzing audit focused on improving coverage for three projects:

1. Notary: https://github.com/notaryproject/notary
2. Notation-go: https://github.com/notaryproject/notation-go
3. Notation-core-go: https://github.com/notaryproject/notation-core-go

Most development of the fuzzers was carried out in the CNCF-Fuzzing repository, https://github.com/cncf/cncf-fuzzing/tree/main/projects/notary. This allowed the Ada Logics team to make smaller iterations of the fuzzers throughout the audit and avoid imposing the overhead of having the Notary maintainers review trivial changes to the fuzzers. OSS-Fuzz was configured to pull the fuzzers from CNCF-Fuzzing in addition to the fuzzers from the Notary Projects repositories.

Results summarised
19 fuzzers developed
All of the Notary Projects fuzzers added to its OSS-Fuzz integration
All of Notations fuzzers supported by CIFuzz
Two crashes found: ● 1 slice bounds out of range panic ● 1 memory exhaustion vulnerability (CVE-2023-25656)

Table of Contents

Project Summary

Ada Logics auditors

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

Notary maintainers involved in the audit

Name	Title	Email
Shiwei Zhang	Notary maintainer	shizh@microsoft.com
Yi Zha	Notary maintainer	yizha1@microsoft.com
Pritesh Bandi	Notary maintainer	pritesb@amazon.com
Toddy Mladenov	Notary maintainer	metodi.mladenov@microsoft.com
Vani Rao	Notary maintainer	vaninrao@amazon.com
Feynman Zhou	Notary maintainer	feynmanzhou@microsoft.com
Samir Kakkar	Notary maintainer	iamsamir@amazon.com
Patrick Zheng	Notary maintainer	patrickzheng@microsoft.com

Assets

Url	Branch
https://github.com/notaryproject/notary	`master`
https://github.com/notaryproject/notation-go	`main`
https://github.com/notaryproject/notation-core-go	`main`

Fuzzing the Notary Project

In this section we present details on the Notary Projects fuzzing set up, and in particular the overall fuzzing architecture as well as the specific fuzzers developed.

Architecture

A central component in the Notary Projects approach to fuzzing is continuous fuzzing by way of OSS-Fuzz. The code repositories of Notary, Notation-go and Notation-core-go and the source code for the Notary Projects fuzzers are the two key software packages that OSS-Fuzz uses to fuzz the Notary Project. The following figure gives an overview of how OSS-Fuzz uses these packages and what happens when an issue is found/fixed.

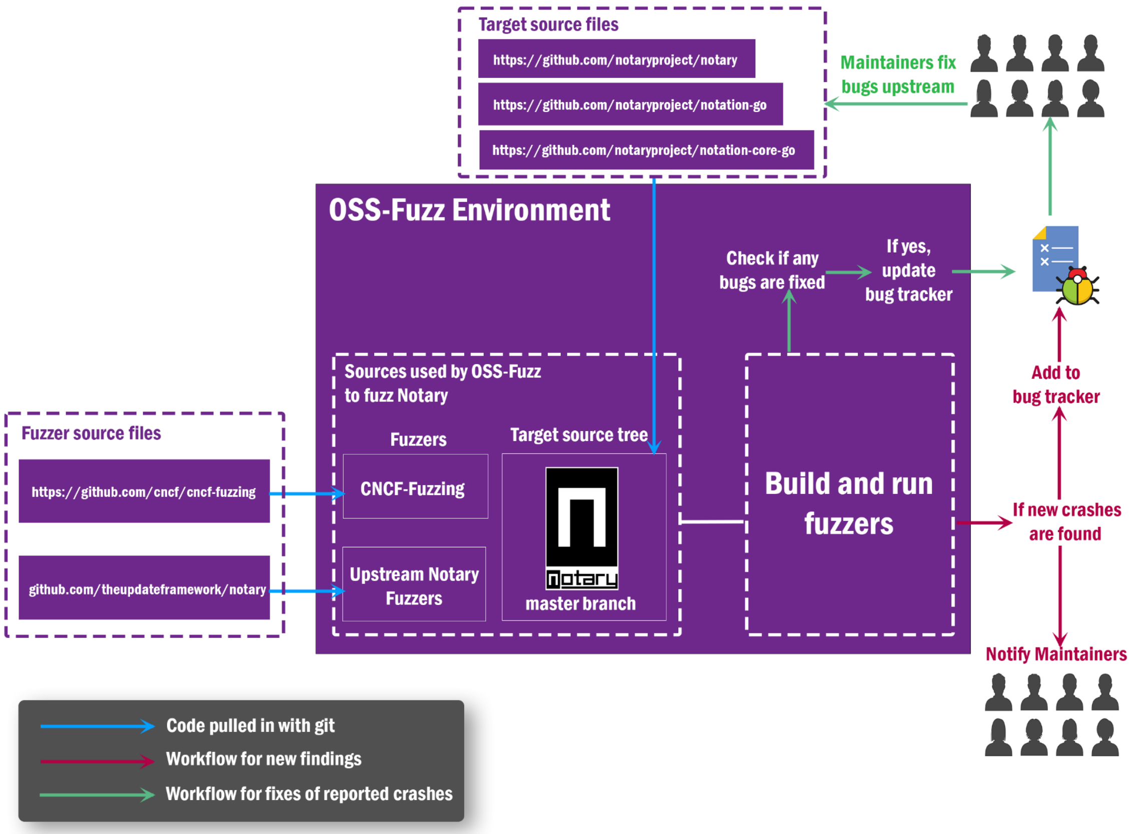
Target source files
https://github.com/notaryproject/notary
https://github.com/notaryproject/notation-go
https://github.com/notaryproject/notation-core-go
Maintainers fix bugs upstream
OSS-Fuzz Environment
Check if any bugs are fixed
If yes, update bug tracker
Sources used by OSS-Fuzz to fuzz Notary
Fuzzers
Target source tree
CNCF-Fuzzing
Upstream Notary Fuzzers
Notary
master branch
Build and run fuzzers
Fuzzer source files
https://github.com/cncf/cncf-fuzzing
github.com/theupdateframework/notary
Add to bug tracker
If new crashes are found
Notify Maintainers
Code pulled in with git
Workflow for new findings
Workflow for fixes of reported crashes

Figure 1.1: Notarys fuzzing architecture

The current OSS-Fuzz set up builds the fuzzers by cloning the upstream Notary, Notation-go and Notation-core-go Github repositories to get the latest Notary source code and the CNCF-Fuzzing Github repository to get the latest set of fuzzers, and then builds the fuzzers against the cloned Notary code. As such, the fuzzers are always run against the latest commit of the three Notary Project repositories. The reason for pulling in three repositories is that Notary organises its project in such a manner, where each main component is located in its own Github repository.

This build cycle happens daily and OSS-Fuzz will verify if any existing bugs have been fixed. If OSS-fuzz finds that any bugs have been fixed OSS-Fuzz marks the crashes as fixed in the Monorail bug tracker and notifies maintainers.

In each fuzzing iteration, OSS-Fuzz uses its corpus accumulated from previous fuzz runs. If OSS-Fuzz detects any crashes when running the fuzzers, OSS-Fuzz performs the following actions:

1. A detailed crash report is created.
2. An issue in the Monorail bug tracker is created.
3. An email is sent to maintainers with links to the report and relevant entry in the bug tracker.

OSS-Fuzz has a 90 day disclosure policy, meaning that a bug becomes public in the bug tracker if it has not been fixed. The detailed report is never made public. The maintainers of each Notary Project repository will fix issues upstream, and OSS-Fuzz will pull the latest master branch the next time it performs a fuzz run and verify that a given issue has been fixed.

Notary Fuzzers

In this section we present a highlight of the Notary Projects fuzzers and which parts of Notary they test. In total, 19 fuzzers were written during the fuzzing audit.

The audit had three code repositories in scope:

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

Notary
This is the repository for Notary 1.0. The notary repository has a server and a client. In this audit, Ada Logics focused more on the server of the two as well as the trustmanager and the cryptoservice. For the server, the handlers were given increased focus.

Notation-go
Along with Notation-core-go, Notation-go is part of the ongoing work for the Notary standard. An extensive documentation for Notary project can be found here: https://github.com/notaryproject/notaryproject.
Notation-go has two main high level API's:

1. `Sign`: Signs an artifact in a remote registry and pushes the signature to the registry. Such a registry can be Distribution[2].
2. `Verify`: Verifies the signature of each of Notarys supported verification types and returns the verification outcome.

[2] https://github.com/distribution/distribution

Notation-core-go

Notation-core-go implements support for Notation signature envelope. In this audit, Ada Logics wrote two fuzzers for the cose and jws types that test each type's parsing routines and subsequently make calls to either the created envelopes `Content()` or `Verify()` methods.

Fuzzers

#	Name	Package
1	`FuzzVerify`	`github.com/notaryproject/notation-go/verifier`
2	`FuzzArtifactReferenceParsing`	`github.com/notaryproject/notation-go`
3	`FuzzDocumentValidate`	`github.com/notaryproject/notation-go/verifier/trustpolicy`
4	`FuzzImportKeysSimple`	`github.com/theupdateframework/notary/trustmanager`
5	`FuzzImportKeysStructured`	`github.com/theupdateframework/notary/trustmanager`
6	`FuzzParsePEMPrivateKey`	`github.com/theupdateframework/notary/tuf/utils`
7	`FuzzAtomicUpdateHandler`	`github.com/theupdateframework/notary/server/handlers`
8	`FuzzAtomicUpdateHandlerMultipart`	`github.com/theupdateframework/notary/server/handlers`
9	`FuzzGetKeyHandler`	`github.com/theupdateframework/notary/server/handlers`
10	`FuzzChangefeed`	`github.com/theupdateframework/notary/server/handlers`
11	`FuzzRotateKeyHandler`	`github.com/theupdateframework/notary/server/handlers`
12	`FuzzDeleteHandler`	`github.com/theupdateframework/notary/server/handlers`
13	`FuzzKeyDBStore`	`github.com/theupdateframework/notary/signer/keydbstore`
14	`FuzzServerStorageSQL`	`github.com/theupdateframework/notary/server/storage`
15	`FuzzServerStorageMemStorage`	`github.com/theupdateframework/notary/server/storage`
16	`FuzzServerStorageTufStorage`	`github.com/theupdateframework/notary/server/storage`
17	`FuzzSignatureCose`	`github.com/notaryproject/notation-core-go/signature/cose`
18	`FuzzSignatureJws`	`github.com/notaryproject/notation-core-go/signature/jws`

19	`FuzzParseDistinguishedName`	`github.com/notaryproject/notation-go/internal/pkix`

Descriptions

1: FuzzVerify

`FuzzVerify` implements a fuzzer that tests notation-go's `Verify` API: https://github.com/notaryproject/notation-go/blob/6c88d3de1283d0d573bafae9d1567026367bfcb6/notation.go#L196. The fuzzer creates a trustpolicy Document with pseudo-random trust policies and validates it. It then creates pseudo-random options. The fuzzer then creates a verifier and invokes `github.com/notaryproject/notation-go.Verify()`, passing all the created parameters.

`github.com/notaryproject/notation-go.Verify()` is a function with a large call tree and multiple parsing routines. Because of Verify's wide reach in the Notation-go project, Ada Logics has optimized this fuzzer to generate structured input data to `Verify()`. This increases the frequency of creating valid trust policies and therefore increases the times the trust policy passes the validation.

2: FuzzArtifactReferenceParsing

`FuzzArtifactReferenceParsing` tests the parsing routine for artifact references in `github.com/notaryproject/notation-go.Sign()`: `oras.land/oras-go/v2/registry.ParseReference()`. `ParseReference()` parses a string into a `oras.land/oras-go/v2/registry.Reference` which `github.com/notaryproject/notation-go.Sign()` subsequently validates:

https://github.com/notaryproject/notation-go/blob/6c88d3de1283d0d573bafae9d1567026367bfcb6/notation.go#L59-L82

```go
func Sign(ctx context.Context, signer Signer, repo registry.Repository, opts
SignOptions) (ocispec.Descriptor, error) {
	// Input validation for expiry duration
	...

	logger := log.GetLogger(ctx)
	artifactRef := opts.ArtifactReference
	ref, err := orasRegistry.ParseReference(artifactRef)
	if err != nil {
		return ocispec.Descriptor{}, err
	}

	...
	if ref.ValidateReferenceAsDigest() != nil {
```

The fuzzer mimics this behavior and passes the unmodified testcase to `ParseReference()` and then validates it.

3: FuzzDocumentValidate
This fuzzer tests the validation of policy documents. Where `FuzzVerify` structures its policy document such that it passes validation more often, `FuzzDocumentValidate` tests policy document validation in a more focused manner to see if the validation routine itself has issues.

4: FuzzImportKeysSimple
`FuzzImportKeysSimple` is one of two fuzzers for `github.com/notaryproject/notary/trustmanager.ImportKeys()`. This fuzzer passes unstructured bytes into the reader.

5: FuzzImportKeysStructured
`FuzzImportKeysStructured` is the second of two fuzzers for `github.com/notaryproject/notary/trustmanager.ImportKeys()`. `FuzzImportKeysStructured` structures the bytes in the reader that are passed to `ImportKeys()`; The fuzzer creates two pem blocks with pseudo-randomized header maps and encodes them. The encoded bytes are added to a reader that is then passed to `ImportKeys()`.

6: FuzzParsePEMPrivateKey
`FuzzParsePEMPrivateKey` tests `github.com/theupdateframework/notary/tuf/utils.ParsePEMPrivateKey()` by passing unstructured bytes and a passphrase.

7: FuzzAtomicUpdateHandler
`FuzzAtomicUpdateHandler` tests the `AtomicUpdateHandler` of `github.com/notaryproject/notary` implemented here: https://github.com/notaryproject/notary/blob/efc35b02698644af16f6049c7b585697352451b8/server/handlers/default.go#L43. The fuzzer creates an http request and sets its body. It then proceeds to add pseudo-random key/value pairs to the requests header. The fuzzer then makes a call to `AtomicUpdateHandler()` passing the http request and a mock writer.

8: FuzzAtomicUpdateHandlerMultipart
This fuzzer is similar to "7: FuzzAtomicUpdateHandler" except that it uses Notarys helper `github.com/theupdateframework/notary/storage.NewMultiPartMetaRequest()` to create a multipart request. The metadata passed to `NewMultiPartMetaRequest()` is created by the fuzzer.

9: FuzzGetKeyHandler
`FuzzGetKeyHandler` sends pseudo-randomized requests to `github.com/theupdateframework/notary/server/handlers.GetKeyHandler()`. The request can either be `GET` or `POST` and has the testcase from the fuzzer as its body.

10: FuzzChangefeed

`FuzzChangeFeed` sends pseudo-randomized requests to `github.com/theupdateframework/notary/server/handlers.ChangeFeed()`. The request can either be `GET` or `POST` and has the testcase from the fuzzer as its body.

11: FuzzRotateKeyHandler

`FuzzRotateKeyHandler` sends pseudo-randomized requests to `github.com/theupdateframework/notary/server/handlers.RotateKeyHandler()`. The request can either be `GET` or `POST` and has the testcase from the fuzzer as its body.

12: FuzzDeleteHandler

`FuzzDeleteHandler` makes pseudo-randomized requests to `github.com/theupdateframework/notary/server/handlers.DeleteHandler()`. The request can either be `GET` or `POST` and has the testcase from the fuzzer as its body.

13: FuzzKeyDBStore

This fuzzer sets up an sqlite database and makes pseudo-random calls in pseudo-random order to the following APIs:

#	API name	API url
1	`AddKey`	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/signer/keydbstore/sql_keydbstore.go#L74
2	`RemoveKey`	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/signer/keydbstore/sql_keydbstore.go#L159
3	`GetPrivateKey`	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/signer/keydbstore/sql_keydbstore.go#L131
4	`RotateKeyPassphrase`	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/signer/keydbstore/sql_keydbstore.go#L167
5	`HealthCheck`	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/signer/keydbstore/sql_keydbstore.go#L232

14: FuzzServerStorageSQL

This fuzzer sets up an sqlite database and makes pseudo-random calls in pseudo-random order to the following APIs:

#	API name	API url
1	`UpdateCurrent`	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L56

2	Delete	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L266
3	CheckHealth	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L293
4	GetChanges	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L312
5	UpdateMany	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L124
6	GetChecksum	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L224
7	GetCurrent	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/sqldb.go#L213

15: FuzzServerStorageMemStorage

`FuzzServerStorageMemStorage` tests methods of the servers `memStorage`. It does so by setting up a new `memStorage` and making pseudo-random calls to the following APIs:

#	API name	API url
1	UpdateCurrent	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/memory.go#L59
2	Delete	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/memory.go#L190
3	UpdateMany	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/memory.go#L101
4	GetChanges	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/memory.go#L219
5	GetChecksum	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/memory.go#L164
6	GetVersion	https://github.com/notaryproject/notary/blob/b55ecf723a77e9834d96617e6ca9a553602024b9/server/storage/memory.go#L175

16: FuzzServerStorageTufStorage

`FuzzServerStorageTufStorage` sets up a new `TUFMetaStore` instance and makes a call to its `GetCurrent()` method.

17: FuzzSignatureCose

`FuzzSignatureCose` creates an envelope for the cose type by parsing raw bytes that are the raw testcase provided by the fuzzer. The fuzzer then either invokes `Verify()` or `Content()` on the created envelope.

18: FuzzSignatureJws

`FuzzSignatureJws` creates an envelope for the jws type by parsing raw bytes that are the raw testcase provided by the fuzzer. The fuzzer then either invokes `Verify()` or `Content()` on the created envelope.

19: FuzzParseDistinguishedName

`FuzzParseDistinguishedName` passes the raw string provided by the fuzzing engine to `github.com/notaryproject/notation-go/internal/pkix.FuzzParseDistinguishedName()`.

Issues found by fuzzers

The audit resulted in 2 unique issues. The most interesting finding is ADA-NOT-FUZZ-1 which revealed a way for untrusted input to control the memory that certain execution paths of Notation-go would allocate thus allowing an untrusted user to allocate excessive memory and cause denial of service. The issue was reported to the Notary security team via OSS-Fuzz bug tracker. Ada Logics and the Notary security team collaboratively triaged the issues, and the Notary community implemented a fix.

#	ID	Title	Severity	Fixed
1	ADA-NOT-FUZZ-1	Excessive memory allocation on verification	High	Yes
2	ADA-NOT-FUZZ-2	Slice bounds out of range in 3rd-party library	Low	Yes

All found issues during this audit have been fixed.

1: Excessive memory allocation on verification

ID	ADA-NOT-FUZZ-1
Severity	High
Fixed	Yes
Fuzzer	`FuzzVerify`
OSS-Fuzz bug tracker: ● https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=55278	

Description

The `FuzzVerify` fuzzer found that a well-crafted payload passed to `github.com/notaryproject/notation-go.Verify()` would allocate excessive memory resulting in a Denial of Service. The issue was assigned CVE-2023-25656. The issue was quickly triaged and fixed by the Notation-go team and users of Notation-go should make sure to patch to `v1.0.0-rc.3` or later.

Github advisory:
https://github.com/notaryproject/notation-go/security/advisories/GHSA-87x9-7grx-m28v

2: Slice bounds out of range in 3rd-party library

ID	ADA-NOT-FUZZ-2
Severity	Low
Fixed	Yes
Fuzzer	`FuzzVerify`
OSS-Fuzz bug tracker: ● https://bugs.chromium.org/p/oss-fuzz/issues/detail?id=55271	

Description

`FuzzVerify` uncovered a slice-bounds-out-of-range panic from a well-crafted payload passed `github.com/notaryproject/notation-go.Verify()`. The root cause was in a 3rd-party dependency and the crash is recoverable.

Stacktrace

```
panic: runtime error: slice bounds out of range [:2] with capacity 0 [recovered]
    panic: runtime error: slice bounds out of range [:2] with capacity 0
goroutine 17 [running, locked to thread]:
main.catchPanics()
    ./main.242133646.go:49 +0x35c
panic({0xa3fc80, 0x10c00016d560})
    runtime/panic.go:884 +0x212
github.com/go-asn1-ber/asn1-ber.parseBinaryFloat({0x10c0001c3dfa, 0x1, 0x1})
    github.com/go-asn1-ber/asn1-ber@v1.5.4/real.go:98 +0x971
github.com/go-asn1-ber/asn1-ber.ParseReal({0x10c0001c3dfa, 0x1, 0x1})
    github.com/go-asn1-ber/asn1-ber@v1.5.4/real.go:45 +0x15e
github.com/go-asn1-ber/asn1-ber.readPacket({0xa51620, 0x10c000101a40})
    github.com/go-asn1-ber/asn1-ber@v1.5.4/ber.go:382 +0xb17
github.com/go-asn1-ber/asn1-ber.DecodePacketErr({0x10c0001c3e00, 0x8, 0x10})
    github.com/go-asn1-ber/asn1-ber@v1.5.4/ber.go:278 +0xaa
github.com/go-ldap/ldap/v3.ParseDN({0x10c0001c780d, 0x30})
    github.com/go-ldap/ldap/v3@v3.4.4/dn.go:177 +0xa99
github.com/notaryproject/notation-go/internal/pkix.ParseDistinguishedName({0x10c0001c780d, 0x30})
    github.com/notaryproject/notation-go/internal/pkix/pkix.go:13 +0xff
github.com/notaryproject/notation-go/verifier/trustpolicy.validateTrustedIdentities({{0x10c00016d0f8, 0x14},
{0x10c0000fbb70, 0x1, 0x1}, {{0x948e5f, 0x5}, 0x10c000101920}, {0x10c0000fbbb0, 0x1, ...}, ...})
    github.com/notaryproject/notation-go/verifier/trustpolicy/trustpolicy.go:410 +0x6f9
github.com/notaryproject/notation-go/verifier/trustpolicy.(*Document).Validate(0x10c000101950)
    github.com/notaryproject/notation-go/verifier/trustpolicy/trustpolicy.go:207 +0xbc5
github.com/notaryproject/notation-go/verifier.FuzzVerify.func1(0x577e0c?, {0x612000000944, 0x12c, 0x12c})
    github.com/notaryproject/notation-go/verifier/fuzz_verification.go_fuzz.go:261 +0x246
reflect.Value.call({0xa17780?, 0xa50248?, 0x13?}, {0x948b2c, 0x4}, {0x10c0001015f0, 0x2, 0x2?})
    reflect/value.go:584 +0x1bce
reflect.Value.Call({0xa17780?, 0xa50248?, 0x94960a?}, {0x10c0001015f0, 0x2, 0x2})
    reflect/value.go:368 +0x1fa
github.com/AdamKorcz/go-118-fuzz-build/testing.(*F).Fuzz(0x10c000058d98, {0xa17780?, 0xa50248})
    github.com/AdamKorcz/go-118-fuzz-build@v0.0.0-20230102134118-c5484365413e/testing/f.go:177 +0x9a6
github.com/notaryproject/notation-go/verifier.FuzzVerify(0x0?)
    github.com/notaryproject/notation-go/verifier/fuzz_verification.go_fuzz.go:251 +0x65
main.LibFuzzerFuzzVerify({0x612000000940, 0x130, 0x130})
    ./main.242133646.go:31 +0x133
main.LLVMFuzzerTestOneInput(0x612000000940, 0x130)
    ./main.242133646.go:24 +0xa5
```

Runtime stats

Continuity is an important element in fuzzing because fuzzers incrementally build up a corpus over time, therefore, the size of the corpus is a reflection of how much code the fuzzer has explored. OSS-Fuzz prioritises running fuzzers that continue to explore more code, and the CPU time presented by OSS-Fuzz runtime stats is thus a reflection of how much work the fuzzers have performed. The following tables lists for each fuzzer[3] the amounts of tests executed as well as the total CPU hours devoted:

Name	Total times executed	Total runtime (hours)
`FuzzArtifactReferenceParsing`	`764,180,569`	`260.2`
`FuzzAtomicUpdateHandlerMultipart`	`81,679,008`	`195.8`
`FuzzChangefeed`	`192,659,989`	`221.1`
`FuzzDeleteHandler`	`745,601,710`	`202.9`
`FuzzDocumentValidate`	`550,815,154`	`255`
`FuzzGetKeyHandler`	`2,108,328,866`	`181.6`
`FuzzImportKeysSimple`	`175,533,650`	`191.8`
`FuzzImportKeysStructured`	`156,116,530`	`220.3`
`FuzzKeyDBStore`	`177,700,699`	`206.2`
`FuzzParseDistinguishedName`	`211,556,578`	`320.7`
`FuzzParsePEMPrivateKey`	`4,784,683`	`189.9`
`FuzzRotateKeyHandler`	`839,503,323`	`223.6`
`FuzzServerStorage`	`7,796,947`	`205.3`
`FuzzServerStorageMemStorage`	`1,018,921,780`	`197.3`
`FuzzServerStorageTufStorage`	`22,209,556`	`166.7`
`FuzzSignatureCose`	`2,042,485,958`	`180.8`
`FuzzSignatureJws`	`2,321,707,044`	`184.1`
`FuzzVerify`	`84,533,667`	`258.7`
`fuzz`	`581,162,788`	`155.1`

[3] As per 14th March 2023.

Conclusions and future work

In this audit, Ada Logics initiated and matured the Notary Projects fuzzing suite. The efforts resulted in 19 fuzzers, 2 issues found and an OSS-Fuzz integration with support for continuous testing.

The fuzzing audit demonstrates that fuzzing provides value for Notary with immediate effect on its security posture. Fuzzing is a continuous discipline, and we recommend that Notary takes the following steps moving forward:

Maintaining the fuzzers

The fuzzers should keep running without breaking to keep testing the code for harder-to-find bugs and build up the corpus.
Over time, the Notary Projects fuzzing suite will improve passively from new features in fuzzing as the industry is working to implement new bug detectors into fuzzing. These will be added in a non-intrusive way, meaning that as long as the Notary Projects fuzzers run continuously, they will over time test for more classes of bugs.

Improve coverage

We recommend making it a continuous effort to identify missing test coverage of the fuzzers. This can be done using the code coverage visualisations provided by OSS-Fuzz.

Require fuzzers for new code

We recommend that new code contributions are required to be accompanied by fuzz tests. This will both ensure that new code gets tested from the moment it gets merged into Notary, and it will ensure that the fuzzer is written by the developer itself.

Acknowledgements

We thank the Linux Foundation for sponsoring this project as well as the team at the Notary Project for a fruitful and enjoyable collaboration.

We also thank the maintainers and team of OSS-Fuzz for their efforts in making open source fuzzing possible in this manner.