Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

Pentest-Report TUF/Notary 05.-06.2018

Cure53, Dr.-Ing. M. Heiderich, M. Wege, MSc. N. Krein, MSc. N. Kobeissi,
Dipl.-Ing. A. Inführ, BSc. J. Hector

Index

Introduction

> *"The Notary project comprises a server and a client for running and interacting with trusted collections. See the service architecture documentation for more information. Notary aims to make the internet more secure by making it easy for people to publish and verify content.*
>
> *We often rely on TLS to secure our communications with a web server which is inherently flawed, as any compromise of the server enables malicious content to be substituted for the legitimate content."*

From https://github.com/theupdateframework/notary

This report documents the findings of a security assessment targeting the TUF/Notary software compound. The project, which comprised a source code audit and a classic penetration test, was carried out by Cure53 in 2018 and yielded only four security-relevant findings. It must be underlined that the assessment of the TUF/Notary scope was requested by The Linux Foundation (TLF) / CNCF. In terms of resources, a budget of eighteen days was allocated to the project, which was promptly completed by seven testers from the Cure53 team. Proceeding on schedule, the two components of the dedicated audit and the more classic series of penetration tests took place in late May and early-to-mid June of 2018.

As for the approach, this assessment relied on a so-called white-box methodology. Under this premise, Cure53 could take advantage of all TUF/Notary sources. This was the sole sensible path as the tested components of the TUF/Notary project are openly available, so impersonating an attacker must account for this context. It is also quite a standard operating procedure for this kind of projects for the Cure53 team to employ a two-fold mixed-methods approach. This means that both a code audit and penetration testing were incorporated to the assessment tasks in order to ensure reaching a comprehensive and maximum coverage. In addition, Cure53 also set up a shared cloud VM to work with and ran the software locally on various instances. Further, the available documentation was studied in great detail, especially since the scope was rather complex and often difficult to grasp. More details on the matters related to the steps and methods can be found in the *Test Methodology* section.

While the Cure53 team was briefed on the scope as well as on the threat- and risk-models applicable to the TUF/Notary project, the white-box premise also allowed for open communications and a continued dialogue between the in-house team maintaining the TUF/Notary project and the Cure53 testers. Throughout the project the exchanges were done via email and a shared document was created and updated to guarantee that mutual understanding of the project's goals is maintained and any issues around coverage are addressed "on the go". As already noted, four security-relevant findings were unveiled as a result of the Cure53 team's efforts. The issues were documented under separate categories of actual vulnerabilities and general weaknesses, with two results assigned to each class. One of the issues may be seen as a duplicate since it was already reported by NCC in 2015[1]. The reason for its inclusion in this report is that it is yet to be fully tackled on all affected entities. Nevertheless, to foreshadow the conclusions, it should be said that the overall results mirror a rather positive impression gained by Cure53 during previous assessments of various TLF/CNCF projects built on top of the Go-codebase. In the following sections, the report will first elaborate on the scope by supplying links for the audited Github repository contents (Notary, TUF and TAPs). In the ensuing sections, first the test coverage notes are provided and then all four spotted issues are discussed.

Finally, the *Report* closes with a conclusion, furnishing a broader verdict on the general security posture of the TUF/Notary project in light of the Cure53's findings.

---

[1] https://github.com/theupdateframework/notary/blob/mas...r_notary_audit_2015_07_31.pdf

Scope

- **TUF / Notary**
  - https://github.com/theupdateframework/notary
  - https://github.com/theupdateframework/tuf
  - https://github.com/theupdateframework/taps

Test Methodology

This section describes the methodology used during the source code audit and penetration testing of the TUF/Notary project. The test was divided into two phases with specific two-fold goals and focal points in scope. The first phase concentrated mostly on manual source code reviews. These reviews aimed at spotting insecure code constructs with a capacity to lead to memory corruption, information leakage or other similar flaws. The second phase of the test was dedicated to classic penetration tests which examined whether the security promises made by TUF/Notary in fact hold in the real-life attack scenarios.

Part 1. Manual Code Audit

A list of items below details the key steps undertaken during the first part of the test, specifically the part which entailed the manual code audit against the sources of the TUF/Notary software in scope. This is to underline that in spite of the relatively low number of findings, substantial thoroughness characterized the completion of the assessment and considerable efforts have gone into this test.

- The source code of the Notary client and server applications was checked for any cryptographic implementation flaws. It was also examined in relation to the general soundness of the design. The code was found to be of an exceptionally high quality, boasting an impressively clear and complete documentation and a comprehensive test suite.
- The threat model, security model and protocol documents for the Notary project were reviewed. An issue was found in which the effective security of the Notary deployments did not exceed that of the traditional TLS 1.2/TLS 1.3 deployments in most use-cases. The problem stems from strong reliance on the Notary server never being compromised, even just temporarily. This is a minor design note which is already recognized by the developers in the existing threat model.
- The Notary application uses the *gorm* library to safely store the application in the user-defined database. The library was installed locally and tested to discover potential issues but no flaws were found.

- The exposed HTTP *Go* routes of the Notary server were analyzed for potential security issues regarding the handling of user-controlled variables. One potential local file inclusion sink has been proven to be properly protected through a verification of the passed variable's validity.
- One of the root causes for TUF-01-001 seems to be the inability to retrieve the *root.json*. This has been further investigated to determine whether the underlying cause of the questionable behavior could have broader security implications. This was driven by the fact that the *root.json* is a key component. However, the root cause could not be identified and -given the limited time available- the investigation into this matter stopped at that point.
- The exposed *gRPC* endpoints have been assessed for security risks currently not covered by the threat model. This assumes that the Notary server was compromised and the *signer* can solely be reached internally. No threats were discovered in this realm.
- The SQL queries where checked to ensure that no injection was possible. All instances used prepared statements properly escaping the values. This has been further tested during a classical black-box style audit yet no vulnerability was found.
- The usage of sensitive functions from the OS package was investigated to check for potential vulnerabilities. However, due to either lack of or limited input-control, it was determined that no vulnerability existed.

Part 2. Code-Assisted Penetration Testing

The items listed below shed light on the steps completed during the second stage of the test, namely the code-assisted penetration testing against the TUF/Notary software in scope. Given that the manual source code audit did not yield a large number of findings, the second approach was chosen as an additional measure for maximizing the test coverage. As means of clarification, the steps executed to enrich this phase are discussed next.

- The Notary project supports *JWT* token authentication. To properly test this feature, the https://notary.docker.io endpoint mentioned in the *Getting started* documentation was used. The authentication scheme was not vulnerable against typical *JWT* attacks like modifying the algorithm.
- It was verified that the potential local file inclusion discovered during Part 1 could not be exploited. The protection in place was particularly strong and a bypass was deemed impossible.
- The *JWT* token is also used for storing keys in the *signer* keystore database. The development team hinted at the decoding of the JWT token potentially being vulnerable to the no algorithm attack. This was envisioned in connection of it

being supported by *jose2go*, that is the library used for encoding/decoding. However, during an insertion of a key into the database a fixed algorithm is used and no scenario was found that could lead to problems during decoding.

- A discrepancy was discovered that leads to different behaviors within the process of decoding JSON through *json.Unmarshal()* and *json.Decode()*. The inconsistency discrepancy was investigated further to see if the difference in behaviors could lead to an exploitable issue. For instance, Cure53 tried to provide two payloads with only one properly signed. However, no exploitable scenario was found.
- The JSON decoding process has been further investigated to check for any unintended behaviors, specifically for supplying two identical JSON key value pairs. Furthermore, attacks similar to the XML signature wrapping were tested. None of these attempts yielded exploitable scenarios.
- During input manipulation tests pertaining to the generated requests between the Notary client and server it was noticed that tampered JSON requests also leave Notary in a state of the already published *collections* becoming broken. Although this only appears to happen with attackers that are in possession of the required *signer* keys, it was still considered a vulnerability. The reason behind this decision was that the *collection* has to be completely removed from the database and re-installed from the beginning in this scenario.
- The docker images were analyzed for sensitive configuration data and stored secrets. The only possible flaws related to *GPG_KEY* and *MYSQL_ALLOW_EMPTY_PASSWORDS="true"* on the MySQL database. However, these are only used locally and require *root* access to the machine. If implemented in a production pipeline, this should be protected by docker secrets.
- The base *OS-image* does not show any signs of hardening and binaries like *Curl, Wget, NC* are present. If an attacker was to gain access to the container, these tools could be used to pivot and establish persistence.
- Security, hardening and separation throughout the docker images that were tested is solely based on the end-users' capability to configure and administer a secure orchestration pipeline.
- The conclusion was made that the Notary team depends on the end-users to establish their own threat model and implement the Notary infrastructure on the basis of that assumption.

Identified Vulnerabilities

The following sections list both vulnerabilities and implementation issues spotted during the testing period. Note that findings are listed in a chronological order rather than by their degree of severity and impact. The aforementioned severity rank is simply given in brackets following the title heading for each vulnerability. Each vulnerability is additionally given a unique identifier (e.g. *TUF-01-001*) for the purpose of facilitating any future follow-up correspondence.

TUF-01-001 General: Input manipulation permanently breaks *collections* (*Medium*)

While testing out-of-order execution for editing and publishing *collections,* it was noticed that one small input manipulation attack could lead to a trusted *collection* becoming broken and unrecoverable. The exact cause of this issue was not discovered during the test, neither it is entirely clear whether the *collection* is actually easily recoverable from remote after all. Nevertheless, this issue is documented as an input validation flaw that should be treated as soon as possible. To reproduce this issue, the only needed action is to intercept the resulting HTTP request when a *collection* gets published. This is seen in the following.

Console output:

```
$ ./notary publish trustedcollection1
Pushing changes to trustedcollection1
Enter passphrase for snapshot key with ID 9cdacd4:
[...]
```

The above generates the following HTTP request which gets intercepted and changed to the one presented next. Note that here a simple *key:value* pair is added to the JSON body.

Intercepting and changing a request:

```
POST /v2/trustedcollection1/_trust/tuf/ HTTP/1.1
Host: notary-server:4443
User-Agent: Go-http-client/1.1
Content-Length: 929
Content-Type: multipart/form-data;
boundary=877b9aca50de4986fe69804888ddb1b906442083fb8fb66a01e8fbdbb8c4
Connection: close

--877b9aca50de4986fe69804888ddb1b906442083fb8fb66a01e8fbdbb8c4
Content-Disposition: form-data; name="files"; filename="snapshot"
Content-Type: application/octet-stream

{"signed":{"_type":"Snapshot","expires":"2021-05-
31T17:25:16.770584584+02:00","meta":{"root":{"hashes":
```

Dr.-Ing. Mario Heiderich, Cure53
Bielefelder Str. 14
D 10709 Berlin
cure53.de · mario@cure53.de

```
{"sha256":"iVf6Fu+jqC2t0aQiNrXg35rrXGSsgT3vv1wmIaobDlk=","sha512":"moiFW5I9ICkSX
K09HMayApjmY5hyvFE+Xl2WGLaRE4uagfhfj5Rixzn5/pYQlBEOU8RyHb4OWda08tK9RWuDFg=="},"l
ength":2373},"targets":{"hashes":
{"sha256":"JNRcd06iRp5TGqIJGrv1b2pmPYVrqrst2Go7+T1htY0=","sha512":"FKy/9kMR4/sj0
wP6pwRXGa+5pC+u+DCe6EYcDCJyvnK7W/JIDlLro0wdXWBpHjvcbxqeIlleJFyoQlNZ64enfQ=="},"l
ength":345}},"version":3},"signatures":
[{"keyid":"9cdacd4c9ad878e4dd209e6626db0a64342c28fe76767dcd050ec989008aa470","me
thod":"ecdsa","sig":"ewzfCrFWbI3NDnfFCZ8gLVV4io9uk5cg+swkAw6LVv4Y1vuVVFUNFJsndVK
gf1tcHe+x4w9FKiy8O9YqSMAY9g=="}],"abc":"xyz"}
--877b9aca50de4986fe69804888ddb1b906442083fb8fb66a01e8fbdbb8c4--
```

After the request passes through, the *notary* executable continues normally. However, all subsequent requests such as removing or adding further *GUN*s or publishing will always result in the server rejecting the operation.

Continued console output:

```
[...]
Pushing changes to trustedcollection1
Enter passphrase for snapshot key with ID 9cdacd4:
Successfully published changes for repository trustedcollection1
$ ./notary publish trustedcollection1
Pushing changes to trustedcollection1
Enter passphrase for targets key with ID be25971:
Enter passphrase for snapshot key with ID 9cdacd4:

* fatal: trust server rejected operation.
```

In this scenario of the rejection, the *notary* server responds consistently with the output below. It does not matter what type of action is taken, so the issue renders the *collection* unusable.

Server response:

```
{"errors":[{"code":"VERSION","message":"A newer version of metadata is already
available.","detail":{}}]}
```

The only method to sufficiently restore the *collection* was to remotely delete the tampered *GUN* entry from the database, which means that this flaw can be considered a Denial of Service vulnerability. Although it is hard to give a general recommendation when the exact issue is not known, one can conclude that unsigned parts of the JSON request should be removed when they are processed and inserted into the database. Better yet, the request should be completely discarded as soon as unsigned parts appear in the JSON request.

TUF-01-002 General: Manipulated version permanently breaks *collections* (Low)

Note: *This vulnerability has been already reported by the NCC group during a pentest back in 2015. It is therefore treated as a duplicate. To tackle this issue the Notary team submitted a proposal to the TUF team. The approach would permit requiring two different roles to sign the same metadata of a file. In case one of the two roles is compromised, this ensured that submitting malicious metadata and breaking the* collection *ceases to be possible.*

Note from TUF: *"We plan to revise the TUF specification to clarify the steps clients should take after failing to validate requested metadata. The detailed work flow section should clarify that (1) requested metadata that is invalid/unsigned should be discarded (2) cached metadata should be deleted/untrusted if it is no longer valid (3) clients should be able to recover following a failed validation check on requested metadata."*

The TUF/Notary server utilizes a *version* property for any new updates. This is deployed for snapshots, targets or root information, among others, and is used to ensure that the version stored by the server is always the newest one. It was discovered that an authorized client can abuse this property to permanently break a *collection*.

A benign Notary client will fetch the latest version number, increase it by one, sign the corresponding role's payload and send it to the server. The server trusts the client in that it is sending a correct version number. An attacker can manipulate the client to set the version number to *2147483647*, the maximum for a 32-bit-signed integer. After a request with the modified version number is submitted, no other user is able to push new updates. The reason is that the benign client can send the version of *2147483648*, which will overflow into a negative number. Therefore, the server will always reject the request as being outdated because of being lower than the latest update.

This behavior was verified by manipulating the version number of a *target* update, as well as manipulating a *snapshot* update. The following Proof-of-Concept (PoC) for the *snapshot* update demonstrates that.

Request to manipulate version number:

```
POST /v2/flow4/_trust/tuf/ HTTP/1.1
[...]

--83cc4e6ccc7de3b48ec6546a1157824fc7bf89194b71877057c62ac73ebe
Content-Disposition: form-data; name="files"; filename="snapshot"
Content-Type: application/octet-stream

{"signed":{"_type":"Snapshot",
[...]
{"hashes":
{"sha256":"Gq0GPxxRgkfCdulLajV3S1iGUJx9Z7RKea0vGeu2yAk=","sha512":"ENmPqXqc8rrzg
TFhMI31ILP9xZgB+6RIZLKXTpYa20XSt1u2TFcpNeqaAASWH3F9kW3jZ1GxYiZOUFdW43mD4w=="},"l
ength":723}},"version":2147483647},"signatures":
[{"keyid":"65ab7f5f2cfe5ed52729ca9ad82b155657a6872d146f3823159348ba776cb526","me
thod":"ecdsa","sig":"FgL1E9eb/OGKwyqPtONkWxJEAC3AEfhpREFgceMoMwPib5oCU52QF27iOR1
3fozZL8/2aYZ81Wef0BEHe1pi4w=="}]}
--83cc4e6ccc7de3b48ec6546a1157824fc7bf89194b71877057c62ac73ebe--

HTTP/1.1 200 OK
Date: Fri, 08 Jun 2018 10:16:30 GMT
Content-Length: 0
Connection: close
```

Request sent by a benign client afterwards:

```
POST /v2/flow4/_trust/tuf/ HTTP/1.1
[...]

--117016a25c9b9c2581b676aba1e9bd12a9b65f66453c62a3dacdcd8a1823
Content-Disposition: form-data; name="files"; filename="snapshot"
Content-Type: application/octet-stream

{"signed":{"_type":"Snapshot",
[..]
,"targets":{"hashes":
{"sha256":"Gq0GPxxRgkfCdulLajV3S1iGUJx9Z7RKea0vGeu2yAk=","sha512":"ENmPqXqc8rrzg
TFhMI31ILP9xZgB+6RIZLKXTpYa20XSt1u2TFcpNeqaAASWH3F9kW3jZ1GxYiZOUFdW43mD4w=="},"l
ength":723}},"version":2147483648},"signatures":
[{"keyid":"65ab7f5f2cfe5ed52729ca9ad82b155657a6872d146f3823159348ba776cb526","me
thod":"ecdsa","sig":"OqnPO1T1UXwgKeLBARNqdO/K4+Zsz7kgAvSKRfXXBk8sekN0515XnxrtZDb
A+0eF0MTxgTj7n6icUywMG9cvsQ=="}]}
--117016a25c9b9c2581b676aba1e9bd12a9b65f66453c62a3dacdcd8a1823--

HTTP/1.1 400 Bad Request
Content-Type: application/json; charset=utf-8
Date: Fri, 08 Jun 2018 10:17:07 GMT
Content-Length: 106
Connection: close

{"errors":[{"code":"VERSION","message":"A newer version of metadata is already
available.","detail":{}}]}
```

Miscellaneous Issues

This section covers those noteworthy findings that did not lead to an exploit but might aid an attacker in achieving their malicious goals in the future. Most of these results are vulnerable code snippets that did not provide an easy way to be called. Conclusively, while a vulnerability is present, an exploit might not always be possible.

TUF-01-003 Protocol: Effective security slightly reduced in some situations *(Info)*

It was observed that the Notary protocol design puts a strong emphasis on the resistance of the Notary server against a compromise. However, in the event of a server compromise, or even simply if a malicious or a temporarily compromised server takes hold, the server could issue signatures for time-stamping or snapshotting.

The above is well-understood by the Notary design team and specifically outlined in the Notary threat model. Earlier drafts of our findings indicated that the capabilities of a principal that compromises a Notary server were more widespread, but discussions with the Notary team limited this scope.

The Notary design could be potentially further improved with regards to snapshotting by requiring a pre-shared key to be entered (or loaded via an HSM module) prior to snapshot generation. However, given our current understanding of the scope of this issue, this does not seem to be necessary.

TUF-01-004 Client: Path traversal in *collection* initialization *(Info)*

When a new *collection* is initialized all its corresponding data is stored at *~/.notary/tuf/<collection_name>*. It was discovered that the client fully trusts the defined *collection name* and therefore allows traversing down the file structure.

The following example illustrate the behavior at stake by traversing down and creating the *collection folder* in the *tmp* directory.

Command:

```
./notary init ../../../../tmp/reverse
```

Created folder:

```
ls /tmp/reverse
> changelist  metadata
```

File:

Notary-master\client\client.go

Code:

```
func NewFileCachedRepository(baseDir string, gun data.GUN, baseURL string, rt
http.RoundTripper,
retriever notary.PassRetriever, trustPinning trustpinning.TrustPinConfig)
(Repository, error) {
        // filepath.join will create normalize the path to /tmp/reverse
        cache, err := store.NewFileStore(
        filepath.Join(baseDir, tufDir, filepath.FromSlash(gun.String()),
        "metadata"),
        "json",
        )
```

There is no immediate threat resulting from this behavior as the Notary binary is not a *root* binary and therefore has the same permissions as the current user. Nevertheless it should be taken into consideration to normalize the specified *collection name* to prevent all kinds of path traversal.

Conclusions

The results of this Cure53 assessment of the TUF/Notary project are exceptionally positive. Funded by The Linux Foundation (TLF) / CNCF and carried out by seven members of the Cure53 team, this security-centered assignment only managed to uncover four issues with limited severities. Despite sophisticated approaches and substantial scrutiny applied by the testers investigating the scope over the course of eighteen work-days, the TUF/Notary components held strong against the attack attempts and thorough audits. The findings of this assignment, performed in late-May and early-to-mid June of 2018, attest to a praiseworthy robustness of the tested project.

To add some important details, it should be reiterated that the threat model and system architecture were communicated to Cure53 in a comprehensive manner. The internal development teams at the TUF/Notary entities were appropriately involved in the project, both in answering Cure53's questions before the start of the project, and in addressing doubts arising later. The communications were facilitated by the assessment status and progress report document, as well as through email messages. The open channels made the communications fluent and productive.

It should be emphasized that a positive absence of security risks in the TUF/Notary entities can be linked to the typically high-standards characterizing the CNCF's *Go*-based projects. Even a two-tiered approach of the code audit paired with testing neither led to a compromise, nor signalled a penetration of the well-set up defenses. The fact that Cure53 audited Uptane implementations and other related tools in the past also did not alter the outcome of this project.

Once again, the choice of *Go* as the underpinning programming language, together with the decisions made around libraries and tools, was found to be clearly advantageous for the software compound. The TUF/Notary entities can be distinguished by traits of clarity, impenetrability and general robustness. Similarly, the cryptographical aspects included in the coverage were fully verified and found to be comparable to TLS in most aspects, but noticeably better in terms of a potential recovery after a compromise.

As the tests progressed, many high-sophistication attack vectors were investigated for increased risks. These items ranged from examining the chosen OS functionality, to file system usage, and to communication endpoints relating to the database access. Few were found to be exposed or cause any danger. Authentication token handling and data exchange encoding was also analyzed in an in-depth manner but nothing security-noteworthy has emerged in this realm either.

Though the provided deployment containers were checked and found to be unhardened by various measures and best practices, this was a choice that a development team has made consciously. In other words, the individual hardening of containers is left up to the user on purpose. Finally, the overall system documentation proved to be accurate, clearly written and complete. The provided examples were concise and useful for the test setup and execution, giving Cure53 high confidence when it comes to reliability and validity of the outcomes.

The testing team can fully recommend the TUF/Notary application compound as secure and verify that the distribution of digital components meets the set out security standards and thresholds. In light of this assessment, Cure53 can only ascertain that the project is mature and ready for the next stages of deployment.

Cure53 would like to thank Justin Cappos, David Lawrence, Vladimir Diaz and Justin Cormack of the TUF/Notary teams as well as Chris Aniszczyk of The Linux Foundation for their excellent project coordination, support and assistance, both before and during this assignment. Special gratitude also needs to be extended to The Linux Foundation for sponsoring this project.