Code Review of the Go TUF Implementation for OSTIF

Final Report and Management Summary

2023-05-31

PUBLIC

X41 D-SEC GmbH
Krefelderstr. 123
D-52070 Aachen
Amtsgericht Aachen: HRB19989

`https://x41-dsec.de/`
`info@x41-dsec.de`

Organized by the Open Source Technology Improvement Fund

Revision	*Date*	*Change*	*Author(s)*
1	2023-03-17	Final Report and Management Summary	L. Gommans, M. Vervier, N. Abel
2	2023-05-31	Public Release	L. Gommans

Contents

Dashboard

Target

Customer	OSTIF
Name	go-tuf
Type	Command Line Application
Version	Commit fab805a8e00b5520

Engagement

Type	Code Review
Consultants	3: Luc Gommans, Markus Vervier and Niklas Abel
Engagement Effort	12 person-days, 2023-02-27 to 2023-03-10

Total issues found	2

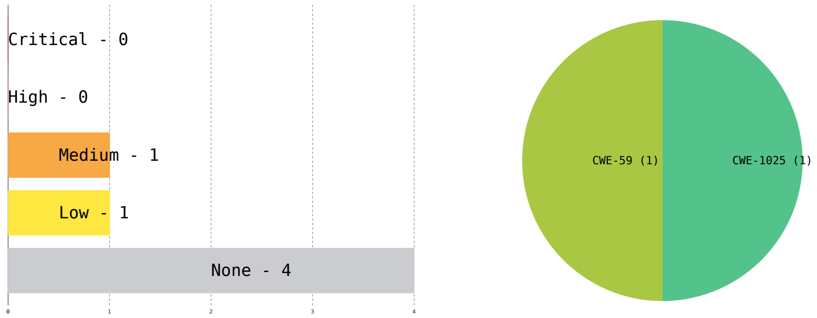

Figure 1: Issue Overview (l: Severity, r: CWE Distribution)

1 Executive Summary

In March 2023, X41 D-Sec GmbH performed a source code audit against go-tuf to identify vulnerabilities and weaknesses in the application. go-tuf is a Go implementation of The Update Framework which allows developers to securely manage updates of software and to minimize the impact on users in the event that update distribution servers or mirrors are compromised.

Two vulnerability were discovered during the test by X41, rated as a medium and a low severity. A further four issues without a direct security impact were identified. An analysis of fuzz testing for the project was additionally performed.

The test was performed by three experienced security experts between 2023-02-27 and 2023-03-10. In a source code audit, all information about the system is made available, although possible environmental and setup specifics might still lead to weaknesses that are not visible in the source code.

One discovered issue allows an attacker to provide erroneous updates under certain circumstances. The update files are not malicious themselves, but due to not being the expected files, this likely still leads to the system not functioning anymore after an update or, again depending on circumstances, could lead to a compromise of the target system.

Specifically, files can be taken from one repository and provided as updates in another. The main prerequisite is using the same signing keys, which could happen if both TUF repositories are managed by the same individual. The attack can be technically mitigated by tying versions to each other (a consecutive update would not be valid because it would link to a different preceding file) or by making root metadata specific to a project such as by including a random value that would not be present in the other project. Stating in the documentation that unique keys are required would be an organizational solution, but might be missed by a maintainer.

Another issue might allow an attacker to overwrite files on the local system of a committer by placing symbolic links into a compromised repository. This might happen due to a compromise in transit of files on which go-tuf then operates. Whether the files are already signed by other functionaries does not matter because such signatures are currently not validated.

Among informational notes, it was noticed that `keys/` directory permissions are set to be world-readable. This does not allow a local attacker to read the key files themselves, but it is best practice to assign only the minimum permissions necessary.

Overall, the project shows a high maturity in terms of security. X41 recommends to resolve the identified issues and continue to apply best practices and hardening to the application.

2 Introduction

X41 reviewed the Go implementation of The Update Framework. The system allows developers to securely manage updates of software and to minimize the impact on users in the event that update distribution servers or mirrors are compromised.

Attackers could try to attack the various components of the system which each have to work correctly, including the parsing of downloaded data, the download mechanism itself, and the signatures of the metadata and target files.

2.1 Methodology

The review consisted of a source code audit.

A manual approach for penetration testing and for code review is used by X41. This process is supported by tools such as static code analyzers and industry standard web application security tools[1].

X41 adheres to established standards for source code reviewing and penetration testing. These are in particular the *CERT Secure Coding*[2] standards and the *Study - A Penetration Testing Model*[3] of the German Federal Office for Information Security.

---

[1] `https://portswigger.net/burp`

[2] `https://wiki.sei.cmu.edu/confluence/display/seccode/SEI+CERT+Coding+Standards`

[3] `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/Studies/Penetration/penetration_pdf.pdf?__blob=publicationFile&v=1`

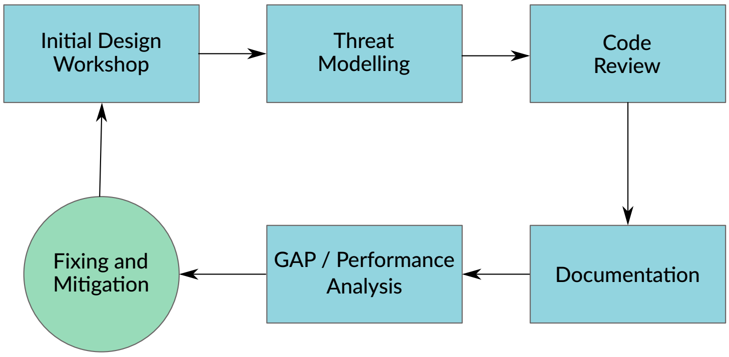

Figure 2.1: Code Review Methodology

2.2 Findings Overview

DESCRIPTION	SEVERITY	ID	REF
Key Reuse Allows Cut-And-Paste Attack	LOW	GOTUF-CR-23-01	4.1.1
Symbolic Link Attacks Possible During Commit Operation	MEDIUM	GOTUF-CR-23-02	4.1.2
Key Folder Permissions	NONE	GOTUF-CR-23-100	4.3.1
Hint About Insecure Passwords	NONE	GOTUF-CR-23-101	4.3.2
Unhandled Errors	NONE	GOTUF-CR-23-102	4.3.3
Suboptimal Scrypt Parameters	NONE	GOTUF-CR-23-103	4.3.4

Table 2.1: Security-Relevant Findings

2.3 Scope

The scope consists of the source code in the following repository. As target version, the untagged commit ID[4] *fab805a8e00b5520c09855385b32761f41b67a6f* is used, which was the latest commit at the start of the audit.

`https://github.com/theupdateframework/go-tuf`

---

[4] Identifier

A repository server implementation is also contained, but the focus of the audit should be on the client code. The specification to which the code will be compared can be found here:

`https://theupdateframework.github.io/specification/latest/`

The specification itself is not part of the scope as it was already audited with the reference Python implementation[5]. Dependencies of go-tuf are also not part of the scope.

2.4 Coverage

A security assessment attempts to find the most important or sometimes as many of the existing problems as possible, though it is practically never possible to rule out the possibility of additional weaknesses being found in the future.

The time allocated to X41 for this assessment yielded a decent coverage of the given scope. Direct dependencies could not be investigated in the time given.

The nature of the project (no interface to present user data using, e.g., HTML[6]) and the use of Go excludes various bug classes such as most memory corruption vulnerabilities. The use of a key-value database which does not use SQL[7] excludes the bug class of query language injections. The key generation mechanism was reviewed for weaknesses. Sensible cryptographic primitives are used throughout the code, ruling out various cryptographic attacks. Client commands such as `init` are robust, removing old state (i.e.: key material) if it was accidentally issued twice with different repositories.

In addition to manual review, static code analysis was performed using gosec, which yielded only false positives.

As requested, good targets for fuzzing were investigated but few could be found. Parsing happens in third-party components such as LevelDB and the Go core JSON[8] parser. It is deemed more promising to use fuzzing for identifying differences between implementations that might lead to security-relevant issues.

---

[5] `https://theupdateframework.github.io/python-tuf/2022/10/21/python-tuf-security-assessment.html`
[6] HyperText Markup Language
[7] Structured Query Language
[8] JavaScript Object Notation

2.5 Recommended Further Tests

X41 recommends to mitigate the issues described in this report. Afterwards, CVE[9] IDs should be requested and users be informed (e.g., via a changelog) to ensure that they can make an informed decision about upgrading or other possible mitigations. Dependencies, if not previously audited, should also be subjected to a security source code review. Testing any deployments is recommended to uncover environmental issues in a specific setup.

---

[9] Common Vulnerabilities and Exposures

3 Rating Methodology for Security Vulnerabilities

Security vulnerabilities are given a purely technical rating by the testers as they are discovered during the test. Business factors and financial risks for OSTIF are beyond the scope of a penetration test which focuses entirely on technical factors. Yet technical results from a penetration test may be an integral part of a general risk assessment. A penetration test is based on a limited time frame and only covers vulnerabilities and security issues which have been found in the given time, there is no claim for full coverage.

In total, five different ratings exist, which are as follows:

Severity Rating
None
Low
Medium
High
Critical

A low rating indicates that the vulnerability is either very hard for an attacker to exploit due to special circumstances, or that the impact of exploitation is limited, whereas findings with a medium rating are more likely to be exploited or have a higher impact. High and critical ratings are assigned when the testers deem the prerequisites realistic or trivial and the impact significant or very significant.

Findings with the rating ‘none’ are called informational findings and are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

3.1 Common Weakness Enumeration

The CWE[1] is a set of software weaknesses that allows the categorization of vulnerabilities and weaknesses in software. If applicable, X41 provides the CWE-ID for each vulnerability that is discovered during a test.

CWE is a very powerful method to categorize a vulnerability and to give general descriptions and solution advice on recurring vulnerability types. CWE is developed by *MITRE*[2]. More information can be found on the CWE website at `https://cwe.mitre.org/`.

---

[1] Common Weakness Enumeration

[2] `https://www.mitre.org`

4 Results

This chapter describes the results of this test. The security-relevant findings are documented in Section 4.1. The fuzz testing analysis is described in Section 4.2. Additionally, findings without a direct security impact are documented in Section 4.3.

4.1 Findings

The following subsections describe findings with a direct security impact that were discovered during the test.

4.1.1 GOTUF-CR-23-01: Key Reuse Allows Cut-And-Paste Attack

Severity:	LOW
CWE:	1025 – Comparison Using Wrong Factors
Affected Component:	Architectural

4.1.1.1 Description

Updates do not tie to previous versions and metadata does not tie to a specific *root* metadata. In a scenario where the same developer maintains multiple TUF repositories with the same cryptographic keys, an attacker can take files from one repository and provide them as an update in the other. No key compromise needs to occur for the client to accept the other repository's files as a legitimate update, but the filename (e.g.: `stable/updater.exe.bz2`) needs to match for it to be accepted and the metadata version number needs to be greater to not trigger the rollback attack prevention.

One could imagine that different TUF repositories are provided for programs such as `gzip` and `bash`. The TUF specification does not require using unique keys per repository. The repository maintainer could reason that it is safer to have only one set of keys: that way, users need to do only one correct key exchange for all their software rather than creating the opportunity for MITM[1] with every new piece of software they want to use. (Another option is to use a single repository, but one might not go that route.)

By using the same keys in `root.json`, a client can be sent an update which contains a `bash` binary when they were trying to obtain a new `gzip` binary. `go-tuf` will see update as legitimate, providing it to the updater which would install it on the system. The most likely result is that the program behaves unexpectedly (`bash` will not produce compressed data) and causes a simple denial of service. However, because many programs have command execution as a feature[2], there is also the potential for gaining arbitrary command execution. User-controlled data being sent to `gzip` is no problem, but (partially) user-controlled data finding its way into the standard input of `bash` allows arbitrary command execution (also because it proceeds after unknown commands, eventually executing the attacker's input so long as no syntax errors are found up until that point).

4.1.1.2 Solution Advice

There are different solutions possible:

- Including a notice in the documentation that TUF repositories do not provide all security guarantees unless fresh keys are used for every repository. This is the most risky option because that notice might go unnoticed.
- Including a NONCE[3] field in the `root.json` which has to be matched by other metadata files, making it impossible to cut and paste files from another repository. This effectively works in the same manner as the previous option, but keeps key management separate from repository management. Here, too, users may not realize that the `root.json` is supposed to be unique and must not be reused, though the presence of a *`nonce`* field with accompanying documentation is deemed less error-prone than a documentation notice alone.
- Linking updates by some mechanism that will not occur across repositories. The current version numbers have a ≈50% chance of being 'compatible' to perform this attack. Replacing the version with a hash of the previous version would mitigate this attack: only identical repositories could still be swapped out, because target file hashes are included in `targets.json`, which is of no use to an attacker.

X41 recommends to implement one of the proposed solutions.

---

[1] Machine-in-the-middle Attack

[2] `http://0x90909090.blogspot.com/2015/07/no-one-expect-command-execution.html`

[3] Number only used once

4.1.2 GOTUF-CR-23-02: Symbolic Link Attacks Possible During Commit Operation

Severity:	MEDIUM
CWE:	59 – Improper Link Resolution Before File Access ('Link Following')
Affected Component:	local_store.go

4.1.2.1 Description

It was discovered that the commit operation does not check if the repository contains POSIX[4]-compliant symbolic links. Should the committer use a repository located at an untrustworthy storage location, such as a remotely mounted NFS[5] or SSHFS[6], an attacker could have placed symbolic links into it.

Two attack scenarios were discovered and verified:

1. An attacker replaces the directory `targets/` in the repository with a symbolic link to another directory. The symbolic link will be resolved locally on the committer's system and the committed files will be written into this directory. The name of these files is not controlled by the attacker.
2. An attacker replaces a file in directory `targets/` with a symbolic link to another file. Since the name of the committed files inside the repository contains a SHA-512[7] hash and the original name of the file, an attacker needs to know the name of the added and committed file and their exact contents in advance. Then they can place a malicious symbolic link into the repository to a local file, which will be overwritten when the commit is performed.

As an example, consider the scenario where an attacker knows that an empty file with the name `myfile` is going to be committed to the repository by a developer and the attacker can modify the compromised repository:

```
user@cr-go-tuf:~/testdevel/repository/targets$ ln -s /home/user/sekret
↪  'cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d36ce9ce47d0d13c5d85f2b0ff8318d2877e⌋
↪  ec2f63b931bd47417a81a538327af927da3e.myfile'
user@cr-go-tuf:~/testdevel/repository/targets$ ls -l
total 4
```

---

[4] Portable Operating System Interface

[5] Network File System

[6] SSH File System

[7] Secure Hashing Algorithm 2, 512-bit

```
lrwxrwxrwx 1 user user 17 Mar 17 15:17 'cf83e1357eefb8bdf1542850d66d8007d620e4050b5715dc83f4a921d⌋
↪  36ce9ce47d0d13c5d85f2b0ff8318d2877eec2f63b931bd47417a81a538327af927da3e.myfile' ->
↪  /home/user/sekret
-rw-r--r-- 1 user user 14 Mar 17 15:07  fe7f13440694fc8ba856ff92b658277ba639da395db730d3bc98323d5⌋
↪  c7a1017ca6fb620324318677bda87a3156d5adc6ce8e7944e23802dea0243cb16a2f43a.foo.txt
```

Listing 4.1: Attacker Backdoors The Repository

Now, when a developer is committing the file, the local file (`/home/user/sekret`) will be overwritten by the committed file's contents:

```
user@cr-go-tuf:~/testdevel/repository/targets$ cat /home/user/sekret
valid
user@cr-go-tuf:~/testdevel/repository/targets$ # the sekret file contains the data as expected
user@cr-go-tuf:~/testdevel$ touch staged/targets/myfile
user@cr-go-tuf:~/testdevel$ tuf add myfile
user@cr-go-tuf:~/testdevel$ tuf snapshot
Staged snapshot.json metadata with expiration date: 2023-03-24 14:17:31 +0000 UTC
user@cr-go-tuf:~/testdevel$ tuf timestamp
Staged timestamp.json metadata with expiration date: 2023-03-18 14:17:34 +0000 UTC
user@cr-go-tuf:~/testdevel$ tuf sign snapshot.json
Signed snapshot.json with 1 key(s)
user@cr-go-tuf:~/testdevel$ tuf sign timestamp.json
Signed timestamp.json with 2 key(s)
user@cr-go-tuf:~/testdevel$ sudo sshfs repo@server:repository repository
user@cr-go-tuf:~/testdevel$ tuf commit
Committed successfully
user@cr-go-tuf:~/testdevel/repository/targets$ cat /home/user/sekret
user@cr-go-tuf:~/testdevel/repository/targets$ # the sekret file was overwritten
```

Listing 4.2: Developer Commit and File Overwrite

This would result in a potential compromise of the developer's machine, depending on the preconditions described above.

The affected code lacking a check for symbolic links was identified to be present in file `local_store.go` shown in listing 4.3 on the next page.

```go
func (f *fileSystemStore) createRepoFile(path string) (*os.File, error) {
    dst := filepath.Join(f.repoDir(), path)
    if err := os.MkdirAll(filepath.Dir(dst), 0755); err != nil {
        return nil, err
    }
    return os.Create(dst) // MARK 1: Default os.Create will follow symbolic links
}

func (f *fileSystemStore) Commit(consistentSnapshot bool, versions map[string]int64, hashes
↪   map[string]data.Hashes) error {
    isTarget := func(path string) bool {
        return strings.HasPrefix(path, "targets/")
    }
    copyToRepo := func(fpath string, info os.FileInfo, err error) error {
        if err != nil {
            return err
        }
        if info.IsDir() || !info.Mode().IsRegular() {
            return nil
        }
        rel, err := filepath.Rel(f.stagedDir(), fpath)
        if err != nil {
            return err
        }
        relpath := filepath.ToSlash(rel)

        var paths []string
        if isTarget(relpath) {
            paths = computeTargetPaths(consistentSnapshot, relpath, hashes)
        } else {
            paths = computeMetadataPaths(consistentSnapshot, relpath, versions)
        }
        var files []io.Writer
        for _, path := range paths {
            file, err := f.createRepoFile(path)
```

Listing 4.3: Code Writing Files To The Repository

As one example scenario, when signature thresholds are used, the two key holders need to exchange the partially signed file as well as the material to verify and sign. There does not seem to be documentation on the requirements for this procedure or exchange. The attack is possible also when one functionary already signed the data because there is no procedure to check the signature.

Since this attack is not feasible in all situations, the severity has been reduced to *MEDIUM*.

4.1.2.2 Solution Advice

X41 recommends to not follow symbolic links for directories and files in the repository. To prevent race conditions during verification, a workaround could be to copy the repository contents to a secure storage location, verify them, and then perform the update of the repository.

Alternatively, using a different file access API[8] than ***os.Create*** can prevent following symbolic links. For example, the following call to ***os.OpenFile*** specifies to never follow symbolic links:

```
file, err := os.OpenFile("my_logs", os.O_RDWR|syscall.O_NOFOLLOW, 0644)
```

Listing 4.4: Opening With No Symlink Follow

---

[8] Application Programming Interface

4.2 Fuzz Testing

During the project, X41 investigated suitable attack surface for fuzz testing the go-tuf project.

4.2.1 Fuzzing Overview

Fuzz testing[9] is a method for automated software testing. It is used to test for security vulnerabilities and other implementation errors. Classically a *fuzzer* or *fuzzing harness* is a program that generates input for another software with the hopes of triggering bugs. While classically used as a black box analysis method, fuzzing is nowadays often used with instrumentation and coverage analysis techniques, either by compiling instrumented binaries or by using dynamic binary instrumentation techniques.

With all fuzzing approaches, the goals for a successful fuzzing operation are:

- Speed: A high number of iterations per second
- Accuracy: A high number of relevant inputs that provide a high coverage of the target code
- High Signal-to-Noise Ratio: Fuzzing results should be valid bugs

4.2.2 Fuzzing go-tuf

X41 investigated methods used to fuzz test go-tuf in a time boxed fashion. Due to the main focus of the work being the review of the source code for vulnerabilities, the fuzzing investigation should be seen as preliminary exploration only.

The scope of this work excludes dependencies such as the JSON parser, therefore the fuzzing attack surface is limited to the following vectors:

- Differential fuzzing (investigating differences between different implementations of TUF, e.g.: python-tuf vs go-tuf)
- Fuzzing targeted at internal processing of go-tuf: Fuzzing data processed after the file and format parsing happened

Additionally, a full scope fuzzing approach can always be applied (end-to-end fuzzing) to find vulnerabilities that are within the scope, but also ones outside.

---

[9] `https://owasp.org/www-community/Fuzzing`

4.2.3 Differential Fuzzing

Differential fuzzing works by executing two different implementations of the specification on the same input. The aim here is to identify differences in parsing and processing of that same input. If such a difference is found, it will either represent a bug in one of the implementations or something not well-defined in the specification.

Differential fuzzing can be described using the following steps:

1. Generate input
2. Run implementation A and B individually on the input generated in step 1
3. Compare the results
4. Crash or abort if the results differ

Due to the time available, a differential fuzzing approach could not be implemented. Such an implementation would need to involve a program that is executing the two implementations individually on the same TUF repository and staging data and perform a comparison afterwards. Such a comparison could be applied to the return value or by comparing the resulting repository and staging directory contents. After each iteration the data has to be restored to a clean state.

4.2.4 Fuzzing Targeted At Internal Processing of go-tuf

It was investigated how fuzzing is currently done in go-tuf. A pull request[10] exists which fuzzes the cryptographic key handling code using the built-in *Go Fuzzing*[11].

The fuzzers try to generate keys from the input fuzz data in the hope of triggering bugs in key generation and handling. Since cryptographic primitives are not in scope for this project, because they are handled by a third-party dependency, this approach was not further investigated. In general, it is feasible to use Go's built-in fuzzing harness for specialized fuzzers.

Due to the time required to craft specialized fuzzers for certain parts of the internal processing, it was decided to follow an end-to-end approach for fuzzing where a general-purpose fuzzer is used to fuzz all parts of the processing.

---

[10] `https://github.com/theupdateframework/go-tuf/pull/365/files`

[11] `https://go.dev/security/fuzz/`

4.2.5 End-to-End Fuzzing of go-tuf

X41 selected *AFLplusplus*[12] (also called AFL++) as fuzzing framework since it is a proven tool which provides flexibility and is considered to be one of the most performant fuzzing tools.

4.2.5.1 Fuzzing Setup

The following setup was chosen according to the documented best practice of AFL++ running on a Debian 11 system with Go v1.20:

```sh
#!/bin/sh
sudo apt-get update
sudo apt-get install -y build-essential python3-dev automake cmake git flex bison libglib2.0-dev
↪  libpixman-1-dev python3-setuptools cargo libgtk-3-dev
# try to install llvm 12 and install the distro default if that fails
sudo apt-get install -y lld-12 llvm-12 llvm-12-dev clang-12 || sudo apt-get install -y lld llvm
↪  llvm-dev clang
sudo apt-get install -y gcc-$(gcc --version|head -n1|sed 's/\..*//'|sed 's/.* //')-plugin-dev
↪  libstdc++-$(gcc --version|head -n1|sed 's/\..*//'|sed 's/.* //')-dev
sudo apt-get install -y ninja-build # for QEMU mode

export LLVM_CONFIG=/usr/bin/llvm-config-12
```

Listing 4.5: AFL++ Setup / Dependencies

4.2.5.2 Fuzzing go-tuf With AFL++

It was found that compatible coverage instrumentation was not available in Go v1.20, therefore the efforts resorted to the use of no instrumentation (dumb fuzzing) and QEMU[13] mode fuzzing.

A repository was created according to the examples provided in the go-tuf documentation[14].

Then the following command was used to fuzz potentially untrustworthy files from the repository such as the file `repository/1.root.json`:

---

[12] `https://github.com/AFLplusplus/AFLplusplus`

[13] Quick EMUlator

[14] `https://github.com/theupdateframework/go-tuf/tree/v0.5.2#examples`

```
afl-fuzz -n -i corp-snapshot/ -o output-snapshot/ -f
↪ /home/user/AFLplusplus/t/repository/1.root.json -- /home/user/go/bin/tuf commit
```

Listing 4.6: AFL++ Dumb Fuzzing

As initial corpus, the original `1.root.json` was used that resulted from the example workflow. The non-instrumented fuzzing was running with a speed of ∼300 iterations per second on a modern Intel i7-12700H CPU[15], inside a Xen guest virtual machine.

Since the non-instrumented fuzzing approach was used, combined with the fact that the JSON parsing was part of the fuzz attack surface, no results were expected and found during 12 hours of fuzzing. To improve this approach, the black box binary fuzzing technique QEMU mode was used:

```
afl-fuzz -Q -i corp-snapshot/ -o output-snapshot/ -f
↪ /home/user/AFLplusplus/t/repository/1.root.json -- /home/user/go/bin/tuf commit
```

Listing 4.7: AFL++ QEMU Mode Fuzzing

This resulted in a tenfold slowdown, making the approach impractical.

4.2.5.3 Fuzzing go-tuf Conclusion and Recommendation

As a conclusion it shows that fuzzing go-tuf effectively would require dedicated efforts to either apply advanced techniques for general purpose fuzzing or to developed dedicated fuzzers that aim at certain parts of the internal go-tuf processing.

To effectively fuzz go-tuf with AFL++, LLVM coverage instrumentation needs to be available at compile time for Go binaries. This should be achievable, but will require further investigation and potentially custom patches to the Go compiler and linker. Other approaches such as hypervisor-based fuzzing are also promising, but require custom efforts as well, which could not be achieved time given.

In the short term, X41 recommends to create more custom fuzzers using the Go Fuzzing built-in framework.

---

[15] Central Processing Unit

4.3 Informational Notes

The following observations do not have a direct security impact, but are related to security hardening, affect functionality, or other topics that are not directly related to security. X41 recommends to mitigate these issues as well, because they often become exploitable in the future. Doing so will strengthen the security of the system and is recommended for defense in depth.

4.3.1 GOTUF-CR-23-100: Key Folder Permissions

Affected Component:	local_store.go

4.3.1.1 Description

The `keys/` folder which is generated during initialization in `go-tuf/local_store.go` lines 317–324 uses mode *0755*, allowing anyone on the system to access and list the contents of the `keys/` folder.

```
func (f *fileSystemStore) createDirs() error {
    for _, dir := range []string{"keys", "repository", "staged/targets"} {
        if err := os.MkdirAll(filepath.Join(f.dir, dir), 0755); err != nil {
            return err
        }
    }
    return nil
}
```

Listing 4.8: Folder Permissions Can be Improved

An attacker can only see which keys exist and their metadata, such as modification times, because the permissions of the key files are set such that only the owner can read and write them.

4.3.1.2 Solution Advice

X41 recommends to assign the minimum permissions necessary and set the mode of the `keys/` folder to a more restrictive value such as *0750*.

4.3.2 GOTUF-CR-23-101: Hint About Insecure Passwords

Affected Component:	gen-key

4.3.2.1 Description

When setting key passwords with the ***tuf gen-key*** command it is allowed to use an insecure passphrase including empty passwords. While this is the responsibility of the user, they may not realize what the involved risks are.

4.3.2.2 Solution Advice

X41 recommends to show a warning to users when an obviously insecure password is entered for the signing keys, such as any string below twelve characters, or a string below twenty characters when no uppercase and digits are used (likely a passphrase which should consist of a sufficient number of words).

4.3.3 GOTUF-CR-23-102: Unhandled Errors

Affected Component:	client/client.go local_store.go client/testdata/go-tuf/generator/generator.go

4.3.3.1 Description

When closing files as well as io streams, the Go `defer` method is used to ensure that files get closed in multiple parts of the program.

When the ***Close()*** method produces an error, the error would not be handled by the program.

```
file, err := os.Open(fpath)
if err != nil {
    return err
}
defer file.Close()
```

Listing 4.9: Unhandled Error When Closing Files in local_store.go:343

This happens in the following lines:

- `local_store.go:343`
- `local_store.go:415`
- `local_store.go:422`
- `local_store.go:447`
- `local_store.go:660`
- `client/client.go:914`
- `client/client.go:687`
- `client/client.go:621`
- `client/testdata/go-tuf/generator/generator.go:57`

4.3.3.2 Solution Advice

X41 recommends to improve the error handling and raise errors or warnings as applicable for the operation that failed.

4.3.4 GOTUF-CR-23-103: Suboptimal Scrypt Parameters

Affected Component:	encrypted/encrypted.go

4.3.4.1 Description

Scrypt is used for strengthening user passwords. While it is good to consider that not all users may have the fastest hardware, the parameters appear to be calibrated for being fast on decade-old hardware[16]. An attacker could perform a large amount of guesses using commodity hardware.

Automatic parameter detection might not be accurate if the system is busy with other tasks.

4.3.4.2 Solution Advice

X41 recommends to choose parameters for modern KDFs[17] in the following order of priorities:

1. increase parallelism
2. increase memory usage
3. increase repetitions

For parallelism, a common choice is the number of CPUs.

Memory usage will depend on the target audience. Considering modern cracking hardware, 128 MiB would be a good lower bound.

Finally, the repetitions can simply be increased until the desired amount of time is reached. Because a maintainer is unlikely to run key decryption operations many times per day, it could be allowed to take a lot longer than the current 100 milliseconds.

In scrypt, there is unfortunately no way to set memory usage independently of repetitions[18].

Furthermore, most popular implementations of scrypt (including Go's) do not support parallelism. The documentation suggests to increase p as CPU parallelism increases, but the code runs sequentially[19]. For Argon2, Go does implement parallelism[20], as also confirmed by practical testing of both implementations.

---

[16] `https://github.com/theupdateframework/go-tuf/blob/master/encrypted/encrypted.go#L26-L33`
[17] Key Derivation Functions
[18] `https://words.filippo.io/the-scrypt-parameters/`
[19] `https://cs.opensource.google/go/x/crypto/+/refs/tags/v0.7.0:scrypt/scrypt.go;l=207-209`
[20] `https://cs.opensource.google/go/x/crypto/+/refs/tags/v0.7.0:argon2/argon2.go;l=182`

Therefore, X41 recommends to use Argon2id, although Scrypt is not a bad second choice.

Regarding backwards compatibility as mentioned in GitHub issue 467[21], the new KDF (settings) should be applied when the old KDF (settings) are detected, be that the old scrypt parameters or scrypt altogether. The 100 milliseconds that are cited as being the creator's recommendation should be reevaluated based on the purpose. Waiting one second with every screen unlock is a different user experience than waiting one second during a software release procedure.

---

[21] `https://github.com/theupdateframework/go-tuf/issues/467#issuecomment-1453757578`

5 About X41 D-Sec GmbH

X41 D-Sec GmbH is an expert provider for application security and penetration testing services. Having extensive industry experience and expertise in the area of information security, a strong core security team of world-class security experts enables X41 D-Sec GmbH to perform premium security services.

X41 has the following references that show their experience in the field:

- Source code audit of the Git source code version control system[1]
- Review of the Mozilla Firefox updater[2]
- X41 Browser Security White Paper[3]
- Review of Cryptographic Protocols (Wire)[4]
- Identification of flaws in Fax Machines[5,6]
- Smartcard Stack Fuzzing[7]

The testers at X41 have extensive experience with penetration testing and red teaming exercises in complex environments. This includes enterprise environments with thousands of users and vendor infrastructures such as the Mozilla Firefox Updater (Balrog).

Fields of expertise in the area of application security encompass security-centered code reviews, binary reverse-engineering and vulnerability-discovery. Custom research and IT security consulting, as well as support services, are the core competencies of X41. The team has a strong technical background and performs security reviews of complex and high-profile applications such as Google Chrome and Microsoft Edge web browsers.

X41 D-Sec GmbH can be reached via `https://x41-dsec.de` or `mailto:info@x41-dsec.de`.

---

[1] `https://x41-dsec.de/security/research/news/2023/01/17/git-security-audit-ostif/`
[2] `https://blog.mozilla.org/security/2018/10/09/trusting-the-delivery-of-firefox-updates/`
[3] `https://browser-security.x41-dsec.de/X41-Browser-Security-White-Paper.pdf`
[4] `https://www.x41-dsec.de/reports/Kudelski-X41-Wire-Report-phase1-20170208.pdf`
[5] `https://www.x41-dsec.de/lab/blog/fax/`
[6] `https://2018.zeronights.ru/en/reports/zero-fax-given/`
[7] `https://www.x41-dsec.de/lab/blog/smartcards/`

Acronyms

API	Application Programming Interface	17
CPU	Central Processing Unit	21
CVE	Common Vulnerabilities and Exposures	9
CWE	Common Weakness Enumeration	11
HTML	HyperText Markup Language	8
ID	Identifier	7
JSON	JavaScript Object Notation	8
KDF	Key Derivation Function	25
MITM	Machine-in-the-middle Attack	13
NFS	Network File System	14
NONCE	Number only used once	13
POSIX	Portable Operating System Interface	14
QEMU	Quick EMUlator	20
SHA-512	Secure Hashing Algorithm 2, 512-bit	14
SQL	Structured Query Language	8
SSHFS	SSH File System	14