// Private Golang Security Assessment 05.20.2024 - 05.22.2024

Fast Updates
Flare Network

HALBORN

Fast Updates - Flare Network

Prepared by: HALBORN

Last Updated 06/12/2024

Date of Engagement by: May 20th, 2024 - May 22nd, 2024

Summary

100% OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
1	0	0	0	1	0

1. Introduction

`Flare Network` engaged Halborn to conduct a security assessment of the Fast Updates Go client beginning on *2024-05-20* and ending on *2024-05-22.* The security assessment was scoped to the source code provided to the Halborn team.

2. Assessment Summary

The team at Halborn was provided three days for the engagement and assigned a full-time security engineer to verify the security of the Fast Updates Go client. The security engineer is a penetration testing expert with advanced knowledge in penetration testing, blockchain protocols, and source code auditing.

The goals for this assessment are:

- Ensure that functions operate as intended.

- Identify potential security issues within the source code that may compromise or alter the functionality.

In summary, Halborn identified one low security issue that was successfully addressed by the `Flare Network team`.

TABLE OF CONTENTS

7.1 Key file not encrypted

3. Test Approach And Methodology

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of this assessment. While manual testing is recommended to uncover flaws in logic, process and implementation; automated testing techniques help enhance coverage of the code and can quickly identify items that do not follow security best practices.

Several tests were carried out during the assessment; including, but not limited to:

- Manual code review and walkthrough

- Test for logical issues and misimplementations

- Manual testing by custom scripts

- Automated code auditing with SAST tool

The security assessment for the code review included automated tests, including static code analysis, and vulnerability scanning with tools like Semgrep. Manual testing involved detailed code walkthroughs, identifying security anti-patterns, and verifying the application of best practices and coding standards. Execution tests were performed on the sections to evaluate runtime behavior, including input validation checks, and secure data handling practices. The process also included evaluating the implementation of security controls, and reviewing the code for common vulnerabilities.

4. RISK METHODOLOGY

Vulnerabilities or issues observed by Halborn are ranked

based on the risk assessment methodology by measuring the LIKELIHOOD of a security incident and the IMPACT should an incident occur. This framework works for communicating the characteristics and impacts of technology vulnerabilities. The quantitative model ensures repeatable and accurate measurement while enabling users to see the underlying vulnerability characteristics that were used to generate the Risk scores. For every vulnerability, a risk level will be calculated on a scale of 5 to 1 with 5 being the highest likelihood or impact.

RISK SCALE - LIKELIHOOD

5 - Almost certain an incident will occur.
4 - High probability of an incident occurring.
3 - Potential of a security incident in the long term.
2 - Low probability of an incident occurring.
1 - Very unlikely issue will cause an incident.

RISK SCALE - IMPACT

5 - May cause devastating and unrecoverable impact or loss.
4 - May cause a significant level of impact or loss.
3 - May cause a partial impact or loss to many.
2 - May cause temporary impact or loss.
1 - May cause minimal or un-noticeable impact.

The risk level is then calculated using a sum of these two values, creating a value of 10 to 1 with 10 being the highest level of security risk.

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONA

- **10** - CRITICAL
- **9 - 8** - HIGH
- **7 - 6** - MEDIUM
- **5 - 4** - LOW
- **3 - 1** - VERY LOW AND INFORMATIONAL

5. SCOPE

FILES AND REPOSITORY

(a) Repository: go-client

(b) Assessed Commit ID: e9e2fb4

(c) Items in scope:

- sortition
- updates
- keygen

Out-of-Scope:

REMEDIATION COMMIT ID:

- 825c958825c958

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM
0	0	0

LOW

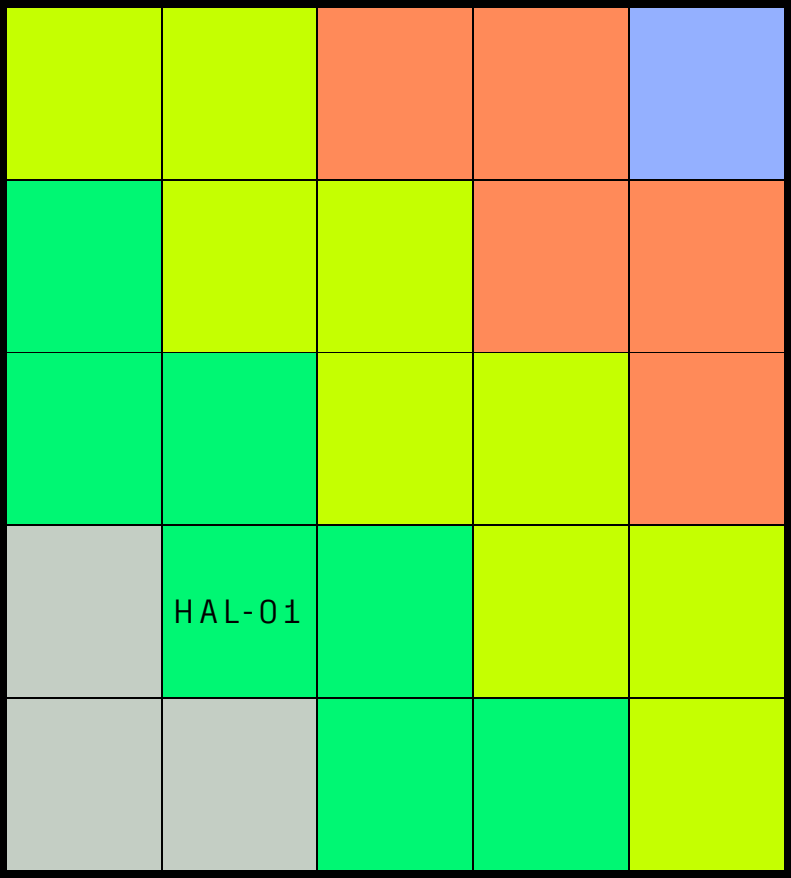
1

INFORMATIONAL

0

LIKELIHOOD

IMPACT



SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
KEY FILE NOT ENCRYPTED	LOW	SOLVED

7. FINDINGS & TECH DETAILS

7.1 KEY FILE NOT ENCRYPTED

// LOW

Description

Having sensitive files unencrypted on the disk poses significant security risks. Without encryption, sensitive data such as personal information, financial records, and proprietary business documents can be easily accessed by anyone who gains physical or remote access to the host. This includes malicious actors, unauthorized employees, or anyone who finds or steals the device.

Proof of Concept

The source code file `fast-updates/go-client/keygen/keygen.go` provides an example Go implementation for client to generate its own private / public key pair. In the following source code lines, the key JSON file is being generated and stored in plain text in disk:

```
47  if *InFileFlag == "" && *InFlag ==
48  "" {
49          logger.Info("No input
50  specified, generating a new key
51  pair.")
52          keys, err =
55              log.Fatal(err)
56          }
57          keyStrings :=
58  keyStrings{PrivateKey: "0x" +
59  keys.Sk.Text(16), PublicKeyX: "0x" +
```

```
60  keys.Pk.X.Text(16), PublicKeyY: "0x"
61  + keys.Pk.Y.Text(16)}
62          keyBytes, err :=
    json.Marshal(keyStrings)
63          if err != nil {
64              log.Fatal(err)
65          }
66
68          if *KeyOutFlag == "" {
69              logger.Info("Key
    generated: " + string(keyBytes))
70          } else {
71              f, err :=
    os.Create(*KeyOutFlag)
72              if err != nil {
73                  log.Fatal(err)
74              }

                _, err =
    f.Write(keyBytes)
                if err != nil {
                    log.Fatal(err)
                }
                err = f.Close()
                if err != nil {
                    log.Fatal(err)
                }
                logger.Info("Saved key
    in file " + *KeyOutFlag)
            }
```

key_out.json

{"PublicKeyX":"0x1fdbc87a4fc17cae32baed01c5b6e52d43e3a6d77d9c645a6fb177df0d633c7a","PublicKeyY":
"0xeb16400dbe337fa15fa461133262f2c50b0e8e262ba27dc4ae2cb3afb1b9a86","PrivateKey":
"0x1975bcec73eed426e21c1d9e498a950e5af6efe7c4843b09fc70fdceb3abc10e"}

Having the key file being stored unencrypted in the disk increases the possibility that a malicious actor obtains its information by directly accessing the file, if previously access to the host containing the file has been gained by other means.

other means.

Score

Impact: 2

Likelihood: 2

Recommendation

It is recommended to enforce the possibility of the output key file being encrypted using a password or passphrase provided by the user prior to its writing on the disk. This way, before using the key file, the file itself must be decrypted using this password, which guarantees that the public - private key information in the file is not stored in plain text on the disk.

Remediation Plan

SOLVED: The **Flare team** fixed this finding by encrypting the private key with AES GCM and a password provided by the user in the commit 825c95826db80ef2627a335754ad4e35326f0687.

Remediation Hash

825c95826db80ef2627a335754ad4e35326f0687

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.