Security Auditing Report

Frame for Desktop (Hot Signer Testing)

Prepared for: Jordan Muir
Prepared by: Lorenzo Stella, Luca Carettoni
Date: 08/28/2019

Table of Contents

Revision History

Version	Date	Description	Author
1	08/27	First release of the final report	Lorenzo Stella
2	08/28	Document Peer Review	Luca Carettoni

Contacts

Company	Name	Email
Frame.sh	Jordan Muir	jordan@frame.sh
Doyensec, LLC.	Luca Carettoni	luca@doyensec.com
Doyensec, LLC.	Lorenzo Stella	lorenzo@doyensec.com

Executive Summary

Overview

Frame.sh engaged Doyensec to perform a feature-focused security assessment of the Frame.sh Electron application. The project commenced on 08/26/2019 and ended on 08/27/2019 requiring 1 security researcher. The project resulted in 9 findings of which 2 were rated as medium or high.

The project consisted of a manual Electron application security assessment.

Testing was conducted remotely from Doyensec EMEA and US offices.

Scope

Through meetings with Frame the scope of the project was clearly defined. We list the agreed upon assets below:

- Frame Desktop App
  - Hot Signer feature only

The testing took place in a development environment using the latest version of the software at the time of testing. In detail, this activity was performed on the following releases:

- Frame 0.2
  - 20cf02e2f24f3b86e519d8293ca05553a1fb0824 https://github.com/floating/frame/tree/0.2

Scoping Restrictions

During the engagement, Doyensec did not encountered any difficulty testing the desktop platform.

Frame was very responsive throughout the entire engagement.

Please note that this project focused specifically on the Hot Signer feature (see https://github.com/floating/frame/tree/0.2/main/signers/hot). With such a limited time frame, Doyensec did not perform any extensive testing on other functionalities or features in the application.

Findings Summary

Doyensec researchers discovered and reported 9 vulnerabilities in Frame's desktop application. While most of the issues are departure from best practices and low-severity flaws, Doyensec identified at least 1 issue rated as high.

It is important to reiterate that this report represents a snapshot of the security posture of the application at a point in time.

The findings included a number of departure from cryptographic best practices that could weaken the overall security posture, the lack of several Electron security features, and a source of potential information leaks.

Overall, the security posture of the Frame Electron Application and of its Hot Signer feature was found to be in line with industry best practices.

At the design level, Doyensec has found the wallet platform to be well architected with the exclusion of the following aspects:

- While the Electron-based application is designed in a modular fashion with a clear separation from UI and business logic components, the application does not currently leverage sandboxing and other mechanisms which help reduce the impact of client-side attacks (e.g. Cross Site Scripting)

Recommendations

The following recommendations are proposed based on studying Frame's security posture and

the vulnerabilities discovered during this engagement.

Short-term improvements

- Work on mitigating the discovered vulnerabilities. You can use **Appendix B** - Remediation Checklist to make sure you covered all of the areas

- Improve the platform's resilience against attack by implementing all defense in depth mechanisms specified within our *Low* and *Informational* findings

Long-term improvements

- Increase your confidence in all third-party NPMs by:

    A. Locking all dependencies on specific versions within private forks
    B. Creating and maintaining a priority list based on project maturity, public scrutiny, functional vs security relevance
    C. Perform extensive graybox testing on all NPMs (in order)
    D. Update dependencies whenever needed for new features or bug fixes

Methodology

Overview

Doyensec treats each engagement as a fluid entity. We use a standard base of tools and techniques from which we built our own unique methodology. Our 30 years of information security experience has taught us that mixing offensive and defensive philosophies is the key for standing against threats, thus we recommend a *graybox* approach combining dynamic fault injection with an in-depth study of source code to maximize the ROI on bug hunting.

During this assessment, we have employed standard testing methodologies (e.g. OWASP Testing guide recommendations) as well as custom checklists to ensure full coverage of both code and vulnerabilities classes.

Setup Phase

Frame provided access to the online environment, source code repository and binaries for the wallet Electron-based application.

Doyensec had to instrument the desktop application to allow debugging and to facilitate testing.

Tooling

When performing assessments, we combine manual security testing with state-of-the-art tools in order to improve the efficiency and efficacy of our effort.

During this engagement, we used the following tools:

- Burp Suite
- Sublime Text
- Electron Devtron
- Chromium Developers Tool
- Electronegativity
- Curl, netcat and other Linux utilities

Electron Apps Testing

Doyensec has been the first security company company to publish a comprehensive security overview of the Electron framework. Thanks to our research efforts, we have extensive experience in analyzing desktop runtime environments based on web technologies. Throughout the engagement, we refined our understanding of the framework threat models and identified vulnerabilities that could subvert security assumptions.

We reviewed security mechanisms to ensure isolation between sites, improving web security protections, and to prevent untrusted remote content from compromising the security of the host.

Examples of issues discovered during Electron app security reviews include, but are not limited to:

- Outdated components and dependencies with known vulnerabilities
- NodeIntegration bypasses
- Sandboxing bypasses
- Flaws in preload scripts
- Weaknesses in custom protocol handlers
- Insecure APIs
- Privacy and secure UX flaws
- Deviations from browser security standards (e.g. SOP)

Web Application and API Testing

Web assessments are centered around the data sent between clients and servers. In this realm, the principle audit tool is the Burp Suite, however we also use a large set of custom scripts and extensions to perform specific audit tasks. We focus on authorization, authentication, integrity and trust. We study how data is interpreted, parsed, stored, and relayed between producers and consumers.

We subvert the client with malicious data through reflected and DOM based Cross Site Scripting and by breaking assumptions in trust. We test the server endpoints for injection style flaws including, but not limited to, SQL, template, XML, and command injection flaws. We look at each request and response pair for potential Cross Site Request Forgery and race conditions. We study the application for subtle logic issues, whether they are authorization bypasses or insecure object references. Session storage and retrieval is scrutinized and user separation is thoroughly tested.

Web security is not limited to popular bug titles. Doyensec researchers understand the goals and needs of the application to find ways of breaking the assumed control flow.

Project Findings

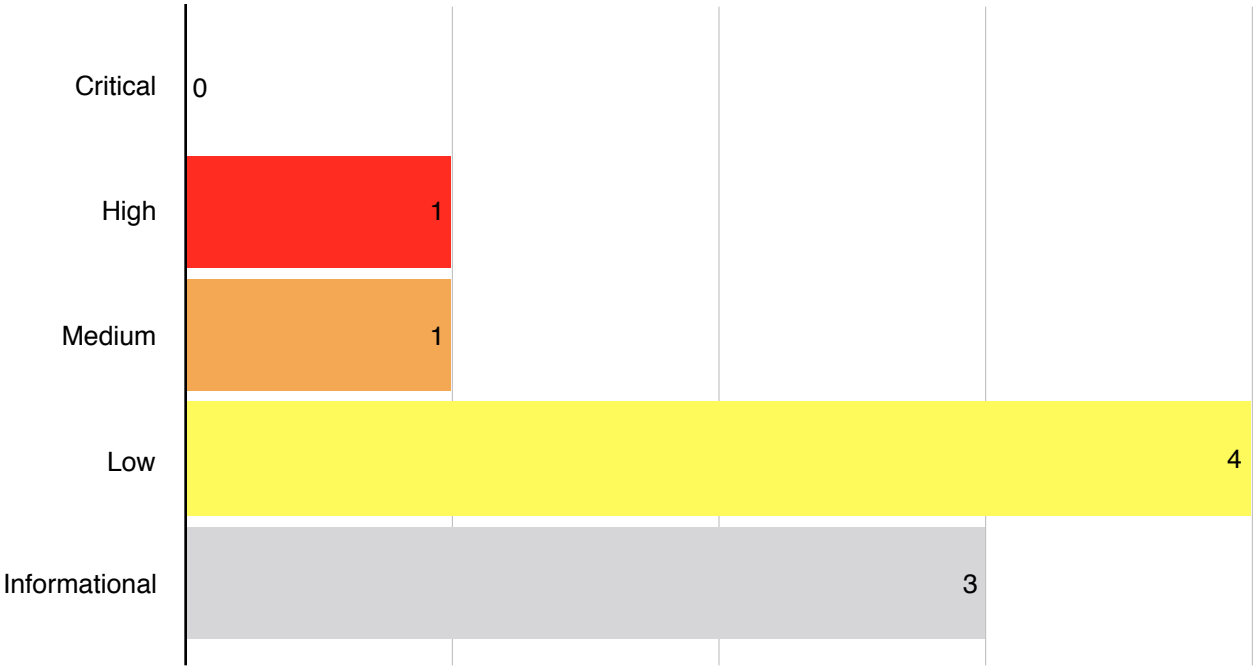
The table below lists the findings with their associated ID and severity. The severity ranking and vulnerability classes are defined in **Appendix A** at the end of this document. The vulnerability class column groups the entry into a common category, while the status column refers to whether the finding has been fixed at the time of writing.

Findings Recap Table

ID	Title	Vulnerability Class	Severity	Status
1	Missing Certificate Pinning	Cryptography – Missing	Medium	Open
2	Electron-Builder Update Signature Bypass (Windows)	Cryptography – Incorrect	High	Open
3	Insecure Manual Updates Mechanism (Linux)	Cryptography – Missing	Informational	Open
4	CSP Bypass In object-src Directive	Security Misconfiguration	Low	Open
5	Missing Permission Request Handler for Untrusted Origins	Security Misconfiguration	Low	Open
6	Secure Memory Not Used for Secrets	Insecure Design	Informational	Open
7	Missing `sandbox` on Main BrowserWindow webPreferences	Security Misconfiguration	Low	Open
8	Insufficient Hot Signer File Deletion	Information Exposure	Low	Open
9	HotSigner Workers Constant Time Token Comparison	Cryptography – Incorrect	Informational	Open

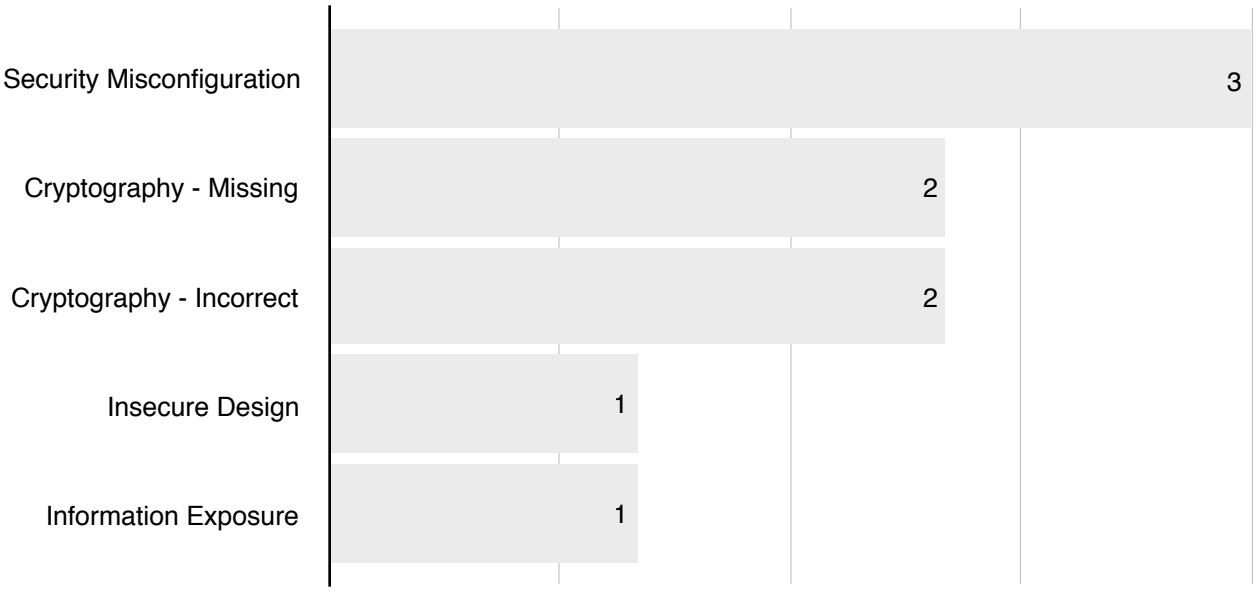
Findings per Severity

The table below provides a summary of the findings per severity.



Findings per Type

The table below provides a summary of the findings per vulnerability class.



1. Missing Certificate Pinning	
Severity	**Medium**
Vulnerability Class	Cryptography – Missing
Component	/app/store/index.js:32-36
Status	Open

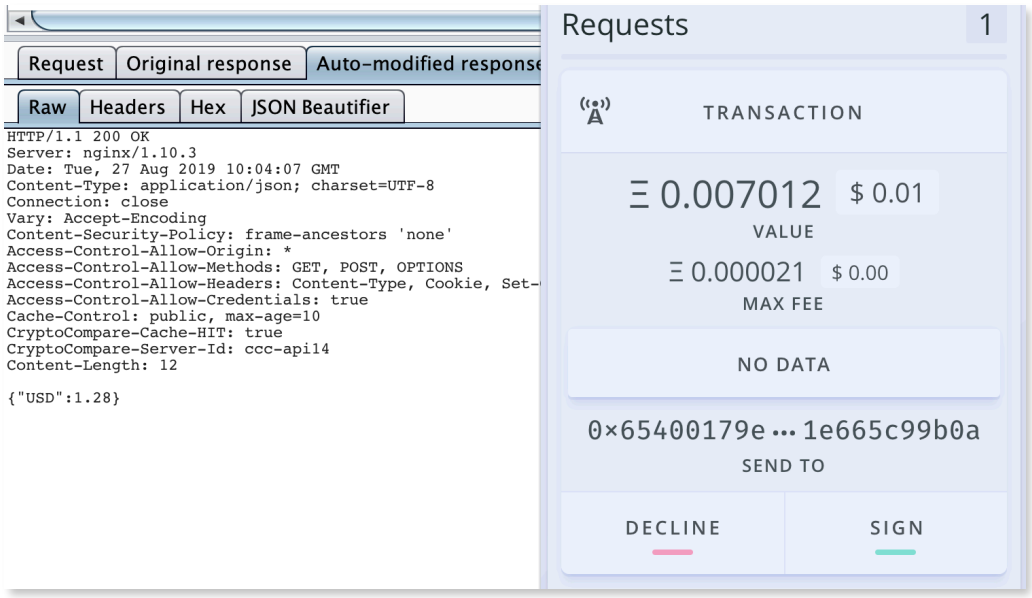
Description

Certificate Pinning is the process of associating a host with its expected X509 certificate, public key or chain of trust. The goal of certificate pinning is to prevent applications from accepting an imposter's TLS certificate that is used to pose as a legitimate service provider during Man-In-The-Middle attacks. The lack of this security measure opens the door to several possible abuses:

1) Exchange Rate Service

Frame periodically polls the CryptoCompare API to access the cryptocurrency market data and rates. The returned values are used to calculate the dollars equivalent of the currency when accepting a transaction.

When Frame attempts to establish a TLS connection with the min-api.cryptocompare.com server, it does not validate the specific certificate chain thus facilitating network interception and tampering. Please note that Node.js does ensure that the certificate is issued by a trusted CA, however, no pinning on specific certificates is performed.



2) Trezor Connect API

Trezor Connect is a JavaScript platform for easy integration of Trezor devices with third-party services (websites or applications). Frame uses Trezor to support all Trezor devices, embedding it in an iframe within the main *frame.html* context. An attacker posing as a Man-In-The-Middle between the Trezor Connect and the victim's client may gain access to the renderer context and leak information or perform more complex attacks (e.g. nodeIntegration bypasses exploits).

3) Updates Server (via Github API)

Updates in Frame are checked by periodically fetching the api.github.com/repos/floating/frame/releases endpoint. The routine responsible to check for new versions of Frame does not perform a certificate pinning check on the Github's API responses. This may allow an attacker to keep the victim's client from receiving important security updates or to serve binaries with a spoofed signature.

4) DApps Permissions

As an additional layer of defense, you could implement a certificate pinning mechanism on the enabled DApps. When a new DApp is added to the permissions list, the Frame application would need to retrieve the SSL certificate of the DApp and to save it, checking it at every new connection. Note that by using this additional security measure you would have to also implement a mechanism to support a future certificate change in case the current one expires or gets rotated, warning the user of the change.

Reproduction Steps

This issue can be reproduced by simply proxying the app traffic through a local https proxy (e.g. Burp Suite).

The Electron framework uses by-default the system proxy settings. You can even use the following command line argument if you run the Electron application directly. Please note that this does not work when using the bundled app.

```
—proxy-server=address:port
```

Or, programmatically with these lines in the main app:

```
const {app} = require('electron')
app.commandLine.appendSwitch('proxy-server', '127.0.0.1:8080')
```

Impact

An attacker may trick a victim in a transaction, spoofing the real rates for the currency. The absence of certificate pinning increases the risk of successful Man-In-The-Middle attacks when the application is used when connected to untrusted networks (e.g. hotels, coffee shops, etc.).

This finding is rated as "Medium" since an attacker would also need to forge a valid certificate for the host, which has been wildly demonstrated to be feasible for a highly-motivated attacker.

Complexity

While the exploitation of this issue can be performed using off-the-shelf tools like https://www.bettercap.org/, it does require a valid min-api.cryptocompare.com certificate (e.g. issued by a compromise CA). Users and attackers should also share the same network segment. Moreover, the attacker has to redirect the user traffic to a malicious host (e.g. using DNS poisoning, fake captive portal, etc.).

Remediation

Implement TLS Certificate Pinning, at least for the connections established to known services.

Please note that this issue affects *libchroumiumcontent* connections as well. However, considering that Frame does not control the receiving servers (i.e. *cryptocompare.com*), it is practically complex to manage pins and enforce certificate pinning.

Resources

- http://hassansin.github.io/certificate-pinning-in-nodejs
- https://gist.github.com/ryankurte/3a40c35f95d130ed3d9a8f3a63cd3e72 (not vetted)

2. Electron-Builder Update Signature Bypass (Windows)	
Severity	**High**
Vulnerability Class	Cryptography – Incorrect
Component	Electron Builder (Auto Update) electron-updater
Status	Open

Description

Frame Desktop's automatic update mechanism relies on Electron-Builder - https://github.com/electron-userland/electron-builder

While testing the update validation mechanism on Windows we found that the signature verification check performed by Electron-Builder is simply based on a string comparison between the binary's *publisherName* and the hardcoded value defined in the *latest.yml* file.

To retrieve the binary's publisher, it uses the following command:

```
powershell.exe -NoProfile -NonInteractive -InputFormat None -Command "Get-AuthenticodeSignature 'C:\Users\<USER>\AppData\Roaming\Frame\__update__\<update name>.exe' | ConvertTo-Json -Compress"
```

However, we discovered that an attacker can bypass the entire signature verification by triggering a parse error in the script responsible for parsing the filename. This can be achieved by using a filename containing a single quote.

Reproduction Steps

In order to reproduce this issue, it is necessary to simulate a compromise of the update server by:

- Polluting the DNS record (or leveraging the machine's hosts file) to redirect traffic to an attacker-controlled machine
- Setup a webserver with the correct directory structure and the following latest.yml file. *Note the quote in the filename*

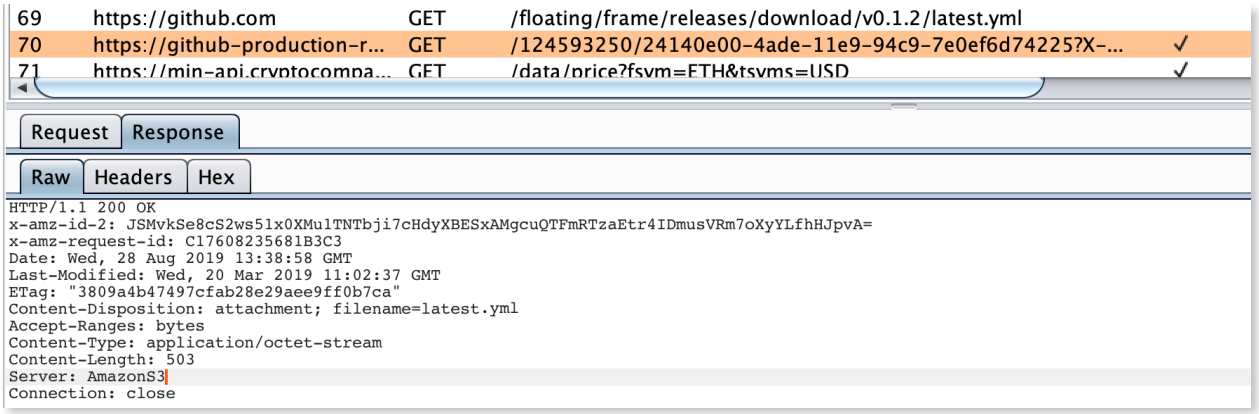
```
version: 0.1.2
files:
  - url: f'rame-setup-0.1.2.exe
sha512:
Elh9UhKyCaPQ7cvX0MDL10UxPmAZr1Xcx22VkYAx9ZrR5X1us4lIQtkYPEvMxDWgNkb8tPCNZLTbKWcDEOJzeA==
    size: 44653917
path: f'rame-setup-0.1.2.exe
```

```
sha512:
EIhgUhKyCaPQ7cvX0MDL10UxPmAZr1Xcx22VkYAx9ZrR5X1us4llQtkYPEvMxDWgNkb8tPCNZLTbKWcDE
OJzeA==
releaseDate: '2019-03-20T11:17:02.627Z'
```

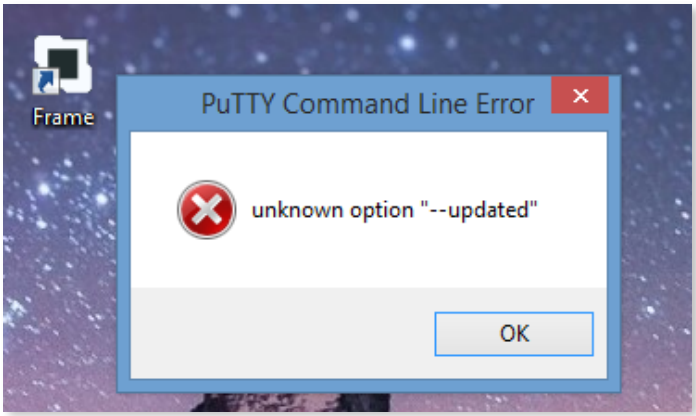
- Instead of the original update, replace the binary with e.g. putty.exe (or any binary signed by a 3rd party) and adjust the binary hashes:

```
shasum -a 512 putty.exe | cut -d" " -f1 | xxd -r -p | base64
```

- Finally, rename the binary to f'rame-setup-0.1.2.exe
- On the victim machine, install an older version of Frame and observe the update download:



- Finally, verify the successful execution of the Putty binary when a warning is triggered



It appears that the filename containing a quote breaks the parsing. As a result, the Powershell script is executed with the following arguments:

```
powershell.exe -NoProfile -NonInteractive -Command "ConvertTo-Json test"
```

Impact

A server compromise of api.github.com, or an advanced MiTM attack, can be leveraged to force a malicious update on Windows clients.

Complexity

An attacker would either need to compromise api.github.com or perform MiTM leveraging the lack of certificate pinning on the Github's APIs.

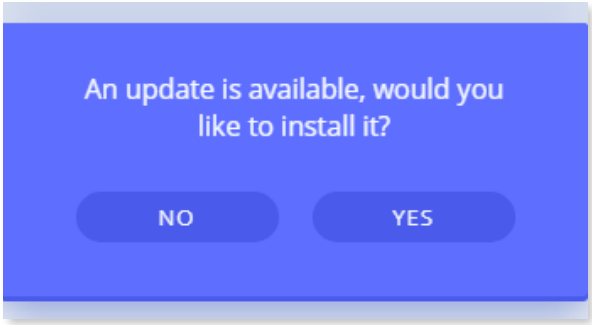
Remediation

Leveraging the Windows App Store seems like the only solution to enforce strong signature verification on Windows platforms without having to re-implement the entire updater logic. Enforcing certificate pinning will mitigate the MiTM attack scenario. Mac updates are not affected by this issue.

3. Insecure Manual Updates Mechanism (Linux)	
Severity	**Informational**
Vulnerability Class	Insecure Design
Component	/main/updater/index.js:64-65
Status	Open

Description

Frame currently uses electron-updater module to provide the automatic update functionality. This feature is available for both Windows and Mac OS, while on Linux it will only trigger a notification for available updates, but users would need to install it manually.



The code responsible to check for updates is located at /main/updater/index.js and periodically parses the api.github.com/repos/floating/frame/releases responses to verify whether a new version of Frame is available or not. If present, the function sets the updater.availableVersion and updater.availableUpdate properties to the retrieved tag name and download url of the update.

When the user will choose to install the updates, if the download URL protocol will be "https:", Frame will use the openExternal handler to open a new window on the system's default browser, forcing a manual download of the executable.

```
installAvailableUpdate: (install, dontRemind) => {
  if (dontRemind) store.dontRemind(this.availableVersion)
  if (install) {
    if (this.availableUpdate === 'auto') {
      autoUpdater.downloadUpdate()
    } else if (this.availableUpdate.startsWith('https')) {
      shell.openExternal(this.availableUpdate)
    }
  }
  windows.broadcast('main:action', 'updateBadge', '')
  this.availableUpdate = ''
}
```

Manually downloading the update will not allow the electron-updater module to check the code signature of the update before the installation. An attacker may abuse this behavior to force a user to execute code and gain persistence on the victim's machine.

Reproduction Steps

Perform a Man-In-The-Middle attack on the Frame app, forging Github's API responses to serve an unsigned package.

Impact

High, an attacker may achieve remote execution on the user device and gain persistence.

Complexity

Medium to High, an attacker would need to install a certificate on the victim's device or forge a valid one for Github. User and attacker would also need to share the same network segment. Moreover, the attacker has to redirect the user traffic to a malicious host (e.g. using DNS poisoning, fake captive portal, etc.).

Remediation

Always validate updates signatures, even for non-Windows or non-Mac-OS releases. It is suggested to use a package manager (i.e. apt) to check the code signature of the update before the installation.

4. CSP Bypass In object-src Directive	
Severity	**Low**
Vulnerability Class	Security Misconfiguration
Component	/app/tray.html:4
Status	Open

Description

Content Security Policy (CSP) is an added layer of security that helps to detect and mitigate certain types of attacks, including Cross-Site Scripting (XSS) and data injection attacks. One of the current CSP rules set by the Frame web application is too loose and should be tightened.

The HTTP Content-Security-Policy object-src directive specifies valid sources for the <object>, <embed>, and <applet> elements. These elements controlled by object-src are to be considered legacy HTML elements and aren't receiving new standardized features (such as the security attributes sandbox or allow for <iframe>), therefore it is recommended to restrict this fetch-directive (e.g. explicitly set object-src 'none' if possible). The fact that an object-src directive is missing in Frame means that the injection of plugins that can execute JavaScript is a possibility.

Reproduction Steps

The current CSP is set trough a meta tag in the html head of the tray.html file:

```
<meta http-equiv='Content-Security-Policy' content="default-src 'self'; connect-src *; style-src 'self' 'unsafe-inline'; frame-src https://connect.trezor.io;"/>
```

Formatted:

```
default-src 'self';
connect-src *;
style-src 'self' 'unsafe-inline';
frame-src https://connect.trezor.io;
```

Impact

High, an attacker may evade the additional layer of protection against cross-site-scripting attacks and data injection attacks provided by CSP.

Complexity

Low, basic web application skills are required. An attacker may be able to easily bypass the CSP mitigation when exploiting a XSS. Many XSS vectors to bypass similar policies are publicly available.

Remediation

Add an object-src directive to the CSP and set it to None.

Resources

- https://research.google.com/pubs/archive/45542.pdf
- https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP
- https://github.com/cure53/XSSChallengeWiki/wiki/H5SC-Minichallenge-3:-%22Sh*t,-it's-CSP!%22

5. Missing Permission Request Handler for Untrusted Origins	
Severity	**Low**
Vulnerability Class	Security Misconfiguration
Component	N/A
Status	Open

Description

When loading remote untrusted content, it is recommended to enable Session's permissions handler, which can be used to respond to permission requests. The current version of Electron allows control of the following permissions:

- *media*
- *geolocation*
- *notifications*
- *midiSysex*
- *pointerLock*
- *fullscreen*
- *openExternal*

By way of example, the *media* type permissions include video or audio elements (i.e. camera & microphone permissions). While browsers have implemented notification to inform the user that a remote site is capturing the webcam stream, Electron does not display any notification.

One means to access the Session of existing pages is by using the session property of WebContents, or from the session module. Using setPermissionRequestHandler, it is possible to write custom code to limit specific permissions (e.g. openExternal) in response to events from particular origins.

This setting can be used to limit the exploitability of certain issues. Not enforcing custom checks for permission requests (e.g. media) leaves the Electron application under full control of the remote origin.

Reproduction Steps

No setPermissionRequestHandler has been found in the codebase. Consequently, the application does not limit session permissions at all, leaving the configuration open to abuses.

Impact

A Cross-Site Scripting (XSS) vulnerability can be used to access the browser media system and silently record audio/video.

Complexity

An attacker would need to find and exploit a Cross-Site Scripting (XSS) vulnerability first.

Remediation

It is recommended to enable Session's permissions handler to filter out external origins or fully block any permission request.
Please note that Electron's Session object is a powerful mechanism with access to many properties of the browser sessions, cookies, cache, proxy settings, etc., therefore any custom modification should be carefully evaluated.

Resources

- https://electronjs.org/docs/all#sessetpermissionrequesthandlerhandler
- https://electronjs.org/docs/all#4-handle-session-permission-requests-from-remote-content

6. Secure Memory Not Used For Secret / Private keys	
Severity	**Informational**
Vulnerability Class	Cryptography – Missing
Component	/main/signers/*
Status	Open

Description

Sensitive cryptographic material, such as the seed and private keys, are stored in Buffer objects, which can be swapped out do disk. Using the Buffer component for cryptographic operations may expose the content of buffers to core dumps, swapping, and forking.

Many cryptographic libraries such as libsodium[1] take care of this automatically, but when performing custom operations it is important to always adopt correct memory management to limit the exposure of sensitive information.

Reproduction Steps

Review the code for *Buffer* objects used for storing sensitive cryptographic data.

Impact

The incomplete removal of the sensitive values in memory may extend the time frame available for an information leak attack. Assuming that an attacker gains access to the heap, the failure of this precautionary measure will help her obtain a variable's content before it is deleted. Additionally, Buffer objects are stored in memory regions that can be swapped out to disk. An attacker with access to the swap file may recover the seed and private keys.

Complexity

Access to the victim's swap file and forensic skills are required for successful exploitation.

Remediation

This issue is caused by the current implementation of Node.js memory management. Because of this, there are no strong guarantees on when a value of the memory will definitely be cleared.
For now, no real remediation is available but only best-effort mitigations: **ensuring that all sensitive variables are correctly overwritten and garbage-collected** can be a first step. We have raised the issue to make the team aware of this limitation and potentially evaluate alternative options to use secure memory**.** See the discussion linked in Resources for additional details.

[1] https://download.libsodium.org/doc/memory_management

Resources

- https://github.com/nodejs/node/issues/18896
- https://www.valentinog.com/blog/memory-usage-node-js/

7. Missing Sandbox Setting for Trezor Connect Renderer	
Severity	**Informational**
Vulnerability Class	Insecure Design
Component	/src/app/index.js:53
Status	Open

Description

While reviewing the overall architecture of the wallet, we also inspected how the main BrowserWindow was instantiated. Despite already having several security flags enabled, Doyensec found that the sandbox setting is not in use.

The lack of this flag would mitigate potential harmful behaviors caused by the Trezor Connect service being compromised since its user interface is presented in an iframe served from https://connect.trezor.io/7/popup.html. In this case, the interaction with all other components would be necessarily implemented using IPC only.

A possible solution would be to move the Trezor Connect service, currently residing in an iframe embedded in the main tray.html window, to a dedicated and sandboxed renderer.

Enabling this security setting would force Chromium to execute in a sandbox all the blink rendering/JavaScript code, using OS-specific features to ensure that exploits in the renderer process cannot harm the system.

Sandboxed renderers expose by-default only the JavaScript APIs. Additionally, a sandboxed renderer does not have a Node.js environment running (except for preload scripts) and the renderers can only make changes to the system by delegating tasks to the main process via IPC. This option should be enabled whenever there is a need for loading untrusted content in a browser window.

Reproduction Steps

For BrowserWindow, sandboxing needs to be explicitly enabled in the declared webPreferences or the application has to be launched with the —enable-sandbox command line flag. The absence of both these solutions indicates that the sandboxing is not enabled. Here's more information about the sandbox.

Impact

Even with nodeIntegration disabled, the current implementation of Electron does not completely mitigate all risks introduced by loading untrusted resources such as the one served from the Trezor Connect service. As such, it is recommended to enable sandbox.

Complexity

Considering the use of trusted local resources and limited navigation options for the user, it is considered complex to leverage a XSS vulnerability or to impersonate the Trezor Connect service via a targeted Man-In-The-Middle attack.

Remediation

Work towards sandboxing the Trezor Connect service along with every future external untrusted origin.

For BrowserWindow, you can enable sandboxing using:

```
mainWindow = new BrowserWindow({
  "webPreferences": {
    "sandbox": true
  }
});
```

If you wish to enable sandboxing for all BrowserWindow instances, a command line argument is necessary:

```
$ electron --enable-sandbox app.js
```

Please note that programmatically adding the command line switch “enable-sandbox" is not sufficient, as the code responsible for appending arguments runs after it is possible to make changes to Chromium's sandbox settings. Electron needs to be executed from the beginning with the "enable-sandbox" argument.

Resources

- https://electronjs.org/docs/all#sandbox-option
- https://chromium.googlesource.com/chromium/src/+/master/docs/design/sandbox.md
- https://doyensec.com/resources/us-17-Carettoni-Electronegativity-A-Study-Of-Electron-Security-wp.pdf

8. Insufficient Hot Signer File Deletion	
Severity	**Low**
Vulnerability Class	Information Exposure
Component	/main/signers/index.js:42 /main/signers/hot/HotSigner/index.js:47
Status	Open

Description

Every time a signer is removed, the abstract delete function is called for each signer. While analyzing the implementations of these removal functions, Doyensec found that:

- The removeAllSigners method only unlinks the files present in the signers directory, without performing a secure deletion:

```
fs.readdir(directory, (err, files) => {
  if (err) throw err
  for (const file of files) {
    fs.unlink(path.join(directory, file), err => {
      if (err) throw err
    })
  }
  })
```

- The delete method for files belonging to the HotSigner type also does not perform a secure deletion of the hot signer file:

```
delete () {
    // Remove file
    removeSync(path.resolve(SIGNERS_PATH, `${this.id}.json`))
    // Log
    log.info('Signer erased from disk')
  }
```

When a file is deleted using fs.unlink()[2], only the reference to the file is removed from the file system table. The file still exists on disk until other data overwrites it, leaving it vulnerable to recovery. For increased security, make sure you do not use fs.unlink() to delete sensitive files without overwriting them before.

The removeSync method is also an alias provided by the fs-extra module for rimraf.sync[3], and does not perform a secure deletion when called.

[2] https://nodejs.org/api/fs.html#fs_fs_unlink_path_callback

[3] https://github.com/jprichardson/node-fs-extra/blob/master/lib/remove/index.js

Impact

Medium, an attacker may still recover the supposedly removed HotSigner data, along with its addresses and encrypted keys.

Complexity

High, an attacker would need physical access to the victim's device.

Remediation

Use a secure deletion method for sensitive files to ensure permanent deletion.
If you want to guarantee that the file content can never be recovered, you have to overwrite the content first. A sample snippet to do this following in shown below:

```
const deletionPath = path.resolve(SIGNERS_PATH, `${this.id}.json`);
// Remove file
s.truncate(deletionPath, 0, function() {
    fs.writeFile(deletionPath, RandomString, function (err) {
        if (err) {
            return console.log("Error writing file: " + err);
        }
    });
});
```

Resources

- https://www.npmjs.com/package/secure-remove

9. HotSigner Workers Constant Time Token Comparison	
Severity	**Informational**
Vulnerability Class	Cryptography - Incorrect
Component	/main/signers/hot/HotSigner/worker.js
Status	Open

Description

While reviewing the source code of the application, Doyensec discovered that the function handleMessage, used to process the IPC messages sent through the channel established between the parent (*HotSigner's main*) and the child (*HotSigner's worker*), is performing an insecure comparison.

By-design every message sent by the HotSigner's *main* requires a token to be accepted by the worker, which performs most of the crypto operations. This token consists of a 32-bytes hexadecimal string and is created when instantiating a signer's worker, but could be leaked by a carefully crafted timing attack leveraging byte-by-byte comparison.

Byte-by-byte comparison between strings in Javascript is usually done by using the standard equality operators (=== or !==), which are designed to return as soon as they encounter two bytes that do not match. Timing oracle leaks information to an attacker, enabling byte-by-byte brute forcing of the data.

There is research on measuring nanosecond long timing differences over the internet in timing attack scenarios[4]. In the case of local execution (like in the Frame's), the time difference can be measured with much better precision and lower timing differences, helping the attacker in the brute-forcing of the token.

Reproduction Steps

The function executes the following code to do the actual comparison:

```
handleMessage ({ id, method, params, token }) {
  …
  // Verify token
  if (token !== this.token) return pseudoCallback('Invalid token')
  …
}
```

Impact

High, cryptographically insecure string comparisons are oracles for malicious actors. This opens a vector to brute force the token value.

4 https://codahale.com/a-lesson-in-timing-attacks/

Complexity

High, this attack is very noisy and requires a lot of IPC requests and responses to measure both system latency and response time.

Remediation

Do a constant time comparison of the passed token. For example, it is possible to use Node's *crypto.timingSafeEqual(a, b)* or bcrypt.js *safeStringCompare()*.

Appendix A - Vulnerability Classification

Vulnerability Severity	**Critical**
	High
	Medium
	Low
	Informational
Vulnerability Type	Authentication and Session Management – Incorrect
	Authentication and Session Management – Missing
	Authorization – Incorrect
	Authorization – Missing
	Components with known vulnerabilities
	Covert Channel (Timing Attacks, etc.)
	Cross Site Request Forgery (CSRF)
	Cross Site Scripting (XSS)
	Server-Side Request Forgery (SSRF)
	Unrestricted File Uploads
	Unvalidated Redirects and Forwards
	Cryptography – Incorrect
	Cryptography – Missing
	Denial of Service (DoS)
	Information Exposure
	Injection Flaws (SQL, XML, Command, Path, etc)
	Insecure Design
	Insecure Direct Object References
	Memory Corruption (Buffer and Integer Overflows, Format String, etc)
	Race Conditions
	Security Misconfiguration
	User Privacy

Appendix B - Remediation Checklist

The table below can be used to keep track of your remediation efforts inside this report. Mark the boxes when a fix has been implemented for the vulnerability.

☐	**Implement TLS Certificate Pinning.**
☐	**Leverage the Windows App Store to provide signed updates.**
☐	**Always validate updates signatures, even for non-Windows or non-Mac-OS releases.**
☑	**Add an object-src directive to the CSP and set it to None.**
☑	**Enable Session's permissions handler to filter out external origins or fully block any permission request.**
☐	**Ensure that all sensitive variables are correctly overwritten and garbage-collected.**
☑	**Work towards sandboxing the Trezor Connect service along with every future external untrusted origin.**
☑	**Use a secure deletion method for sensitive files to ensure permanent deletion.**
☑	**Do a constant time comparison of the passed token.**

When done patching the listed vulnerabilities, many clients find it worthwhile to perform a retest. During a retest Doyensec researchers will attempt to bypass and subvert all implemented fixes. Retests usually take one or two days. Please reach out if you'd like more information on our retesting process.