# Zeal Wallet Penetration Test

## Table of Contents

# Secure Technical Solutions LLC

## Summary

**Zeal Wallet**

Zeal's wallet solution enables users to interact with the blockchain from their web browser. The wallet is compatible with multiple EVM-compatible chains. The wallet also offers a feature to watch other accounts by providing only the account address. One unique aspect of Zeal is that the wallet is embedded into the dApp page when connected, and is not in a popup like some wallets. This offers a more seamless user experience by removing the need to expand and hide a pop-up during dApp interactions. The full system includes a frontend component and a backend component. The frontend consists of the wallet extension in the browser, which is where the private keys are stored and how the user interacts with blockchain applications. The backend is what the frontend communicates with to perform all external interactions, whether that is acquiring price data, performing RPC calls, or checking the assets held by an account.

## Testing Assumptions

The entire Zeal monorepo was in scope for this review, but the focus was on the frontend TypeScript components because this is where the most sensitive data (private keys) are stored. Time was also spent reviewing the backend application, but the AWS infrastructure and non-custom applications (Kafka, Redis, etc.) in the backend were specifically out of scope for this penetration test. The repository was reviewed at commit hash 688a5cd813d8f5943251a1b0763556b42b546d0f which corresponded to extension version 0.2.17. The Chrome web browser versions that were used during this test were versions 109.0.5414.119 and 110.0.5481.77 (beta). The penetration test was performed by one security engineer for two weeks.

After the primary test was completed, additional testing of the new EOA wallet creation feature was performed. Testing of this feature happened on commit hash 9de613a1987c750409e0733b02977012c06a0cc3 which corresponded to extension version 0.3.2.

## Testing Methodology

Given the architecture of the Zeal wallet solution, the following list of key security areas were considered:

- Security best practices: verify dependencies are updated and contain no vulnerabilities, perform static analysis of source code using common scanning tools, assess Zeal wallet implementation protection to past security incident attack vectors on similar technology stacks
- Known crypto wallet weaknesses: Compare with some past works (for example, audit reports of other crypto wallets, and articles like this and this) to check if the common weaknesses have been addressed
- Supply chain attack vectors: assess dependency confusion risks using tools like confused or confuser in addition to a manual review of the dependency chain and pipeline
- Cryptographic library integration errors: assess whether the cryptographic libraries used are the best choice for the job, validate that the best practices for each library is followed, confirm all critical cryptographic functions are implemented securely
- Browser extension weaknesses: assess exposure to attack vectors in recent extension-related security research, validate browser extension best practices are followed, confirm private extension data is saved securely in storage, determine overall privacy strength and any extension fingerprint identifiers
- Backend security: assess security of custom API, check Java application implementation for common Java security weaknesses, verify connections to external integrations are implemented securely

Some specific attack vectors that were considered during this review included:

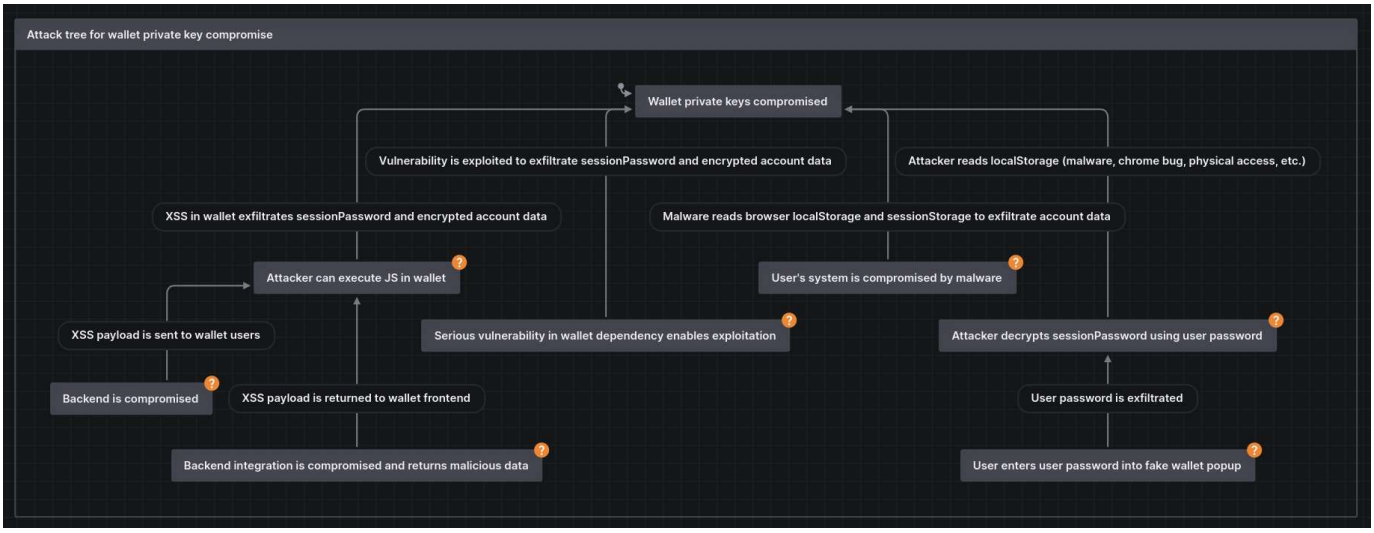
1. Can the password prompt to login to the wallet be bypassed?
2. Is the password material in the wallet stored securely before and after login?
3. Are the wallet private keys stored securely?
4. Could another extension installed in the same browser read sensitive data from the wallet?
5. When signing a transaction, are the keys handled in a secure manner?
6. When an account is removed from the wallet, is all data for this account properly removed?
7. What safety measures does the wallet provide against malicious transactions and how easy are they to bypass?

8 What data is passed from the backend to the wallet and is this data sanitized?

9 How does the backend integrate with and receive data from external data sources?

10 Are there any common API vulnerabilities in the system from the OWASP Top 10?

Because the most critical risk of a crypto wallet is the compromise of private keys, especially at scale (as opposed to the compromise of a single user's keys), an attack tree was sketched out to understand various entry points that a malicious party may pursue if they were targeting weaknesses in the wallet. Note that other attack trees for different end goals, such as identifying the public blockchain addresses associated with Zeal wallet users, would involve a completely different attack tree with different entry points and vulnerabilities.




Secure Technical Solutions LLC makes no warranties regarding the security of the code and provides no guarantee that the code is free from defects. The authors of this report do not represent nor imply to third parties that the code has been fully audited nor that the code is free from defects. By using this code, Zeal wallet users agree to use the code at their own risk.

## High Findings

None.

## Medium Frontend Findings

### 1. Medium - Vulnerable frontend dependencies

**Summary**

When including dependencies of dependencies, the wallet has over 3000 dependencies, making it hard to keep them all updated to be free of vulnerabilities. Several vulnerable packages were found among the dependencies.

**Vulnerability Detail**

Several vulnerable dependencies were found in the project, including:

- glob-parent
- got
- trim
- nth-check
- postcss
- trim-newlines

Some dependencies are deprecated, including:

- fsevents
- resolve-url
- urix
- source-map-url
- multicodec and other packages superceded by multiformats
- eth-sig-util

Even if a dependency is not vulnerable, it may still be outdated. The latest version of a dependency should generally be used or at least considered for use, and the first step to making that decision is awareness that a new version exists. Consider regularly using `yarn outdated` (or `npm outdated` if `yarn outdated` gives an error, as it did at the time of this review) to see which packages are outdated. At the time of this review, the following upgrades could be made in package.json:

- @metamask/browser-passworder 3.0.0 -> 4.0.2
- @sentry/react 7.29.0 -> 7.36.0
- @types/eslint 7.29.0 -> 8.21.0
- @types/uuid 8.3.4 -> 9.0.0
- web-vitals 2.1.4 -> 3.1.1

Any project relies on its dependencies, and the dependencies should be regularly updated to newer versions to include patches for any security issues that are published. There are many tools for this, including yarn audit, OWASP Dependency Check, and Snyk's browser-based health check tool.

#### Recommendation

Resolve the issues highlighted by dependency check tool. Run `yarn audit` in CI testing to catch vulnerable dependencies and consider running OWASP Dependency Check in CI as well. Additionally consider signing up to an alert feed that goes beyond Github Dependabot's alerts to be notified of any security-related dependency issues.

### 2. Medium - Improve extension's manifest.json

#### Summary

The extension's manifest.json file contains minimal information. Adding additional fields can strengthen the security of the extension.

#### Vulnerability Detail

Fields that are missing from the wallet's manifest.json but could improve the security of the extension (and are found in other popular wallet extensions) include:

- minimum_chrome_version: Specify a minimum version of chrome to ensure the browser where the wallet is installed does not have significant security vulnerabilities. Given the adoption cadence of Chrome browser versions and the currently supported versions, a minimum version of 107 should allow support of most users who regularly update their computers. And if the Zeal wallet is expected to be primarily a hot wallet for users, then the systems installing it should be online and regularly updated. Another approach is to use a value like defaults from browserslist
- update_url: This should be set to https://clients2.google.com/service/update2/crx to make sure updates for this extension only come from Google and not from a 3rd party
- key: Setting a key allows the extension to continue using the same unique ID value. Follow the instructions in the developer docs to set this value.

- content_security_policy: Setting a strict CSP can reduce the risk of a XSS attack vector in the wallet. While the default policy is fairly strict, consider adding restrictions beyond the default for added security. This Google talk about secure extension development may assist with choosing a good CSP.
- use_dynamic_url in web_accessible_resources: enabling the dynamic url feature will make it much harder for websites such as this fingerprinting demo site to fingerprint the wallet extension, as explained here.

Other fields that are not security-specific but are recommended by the Chrome developer docs include:

- description: provide the user proper information when they view their install extensions at chrome://extensions/
- default_locale: specify language

#### Recommendation

Make the changes described above to the extension's manifest.json.

### 3. Medium - Plaintext localStorage data is easy to manipulate

#### Summary

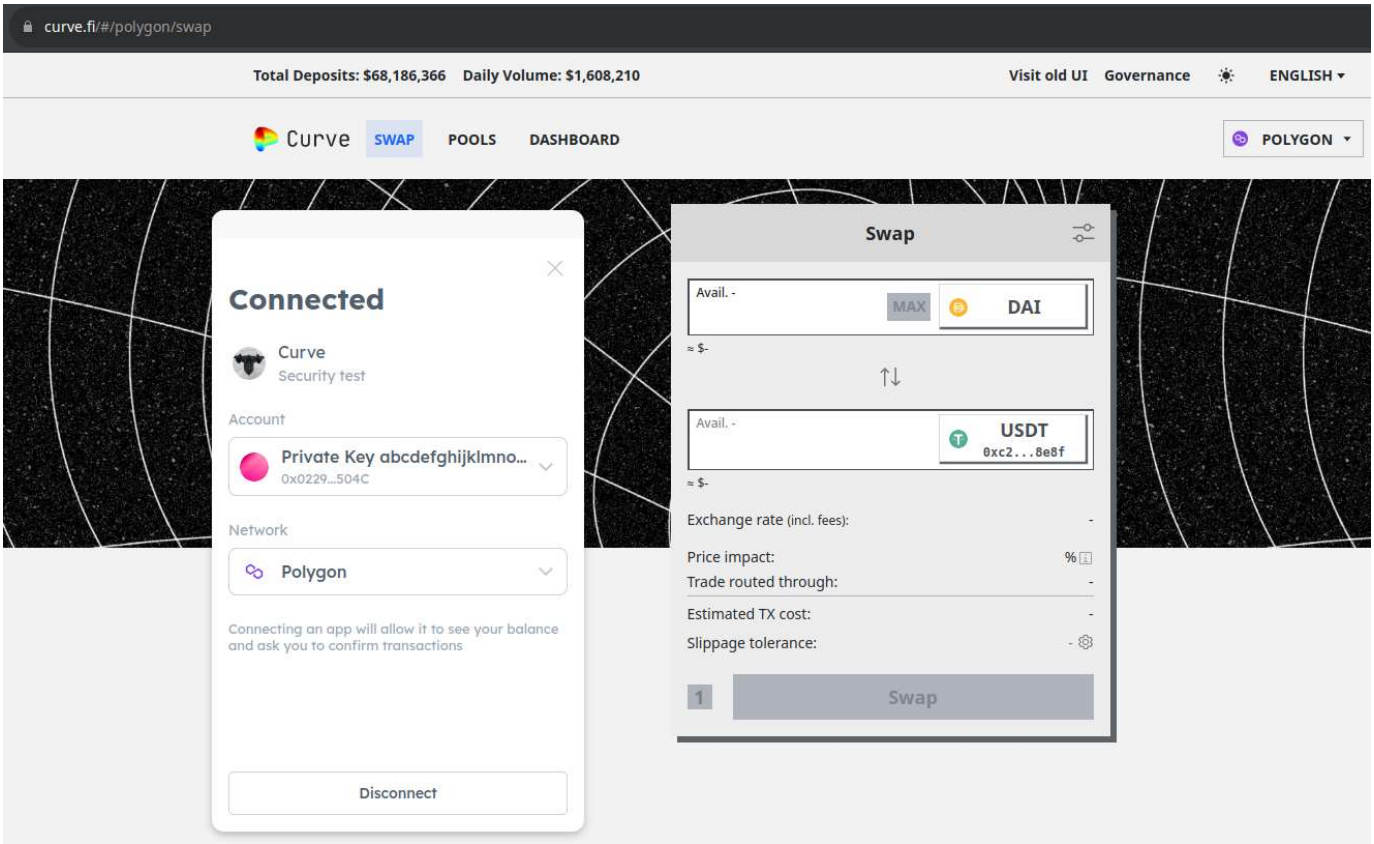
There is a lot of account information stored in localStorage. Data stored in localStorage is persistent and is not removed between browser sessions. Much of the data stored in localStorage is stored in plaintext. Plaintext data is an easy target for modifications. Some of these modifications can break assumptions about how the extension operates.

#### Vulnerability Detail

localStorage is where data about the wallet accounts is stored. Cached data like dApp icons is also stored in localStorage. This cached data is stored in plaintext. Plaintext data is vulnerable to manipulation. Although browser localStorage is isolated by origin and should be safe from malicious websites and browser-based attacks, other malicious software on the user's system could impact the integrity of the data. For example, it's possible that a data field stored in localStorage is vulnerable to a XSS payload when rendered in the wallet UI (stored XSS vulnerability), which may allow a malicious program to inject the vulnerability into the wallet. Another way that modifying localStorage could be used is for targeted spearphishing attacks. If the cached data indicates that the phishing site is recognized as the legitimate dApp (Curve, for example), the user is more likely to become a victim. Other assumptions can be broken by manually modifying the localStorage, including:

- Editing the cached icons URLs to render different images when certain dApp URLs are visited
- The dApp hostname can be modified
- Modifying the account name so it exceeds 35 character limit (some XSS payloads that can bypass various checks are very long, so length limits can be a useful security measure)
- Adding or removing dApps from the cache history. Added dApps can contain custom data
- Modify the balance of a token in the wallet or the value of that balance (the modification will only persist until the backend returns an a value that updates the wallet)
- Delete the localStorage contents, requiring the zeal wallet account to be recreated

The image below shows the first three points in action (custom icon image, custom dApp hostname, extra long account name)




Encrypted data, on the other hand, is very hard to manipulate without knowing the key to decrypt the data. The data stored in localStorage should all be encrypted, similar to how the keystore map data is stored in localStorage.

#### Recommendation

Encrypt all data stored in localStorage to maintain data integrity. Append a checksum to this data before encrypting it, such as a hash of the data or part of the data, to help verify that the decrypted data is properly decrypted. Encrypting localStorage data would additionally resolve the pre-login localStorage privacy finding.

## Medium Backend Findings

### 4. Medium - Vulnerable backend dependencies

#### Summary

Several Java dependencies in the backend are outdated.

#### Vulnerability Detail

The gradle.properties file contains the version numbers of several outdated dependencies, including:

- com.fasterxml.jackson.core
- javalin
- web3j
- log4j
- newRelicAgentVersion

#### Recommendation

Update all dependencies. Integrate OWASP Dependency Check into the build CI pipeline.

### 5. Medium - Plaintext secrets in repository

#### Summary

Secrets should not be hardcoded into source code, but stored in a more secure location that the program queries as needed.

#### Vulnerability Detail

The file data/src/main/scala/Bronze.scala contains plaintext secrets, including AWS access keys and kafka login data. Even if the repository is private, weakly protected secrets are a common supply chain attack target. CI, AWS servers, and other parts of the build pipeline have access to these plaintext secrets.

#### Recommendation

Remove hardcoded secrets from the repository. Furthermore, run a tool like git-secrets or gitleaks in the project's CI to detect secrets uploaded to the repository. Some use cases in Java applications may want to consider using a Java keystore.

### 6. Medium - No API rate limiting

#### Summary

If someone wanted to cause a denial of service for certain wallet functions across all wallet users, they may choose to perform a large number of backend API calls to 3rd party integrations to use up the full API rate limit. Once the maximum number of API calls is reached, the API calls will fail for all wallet users.

#### Vulnerability Detail

Some of the backend integrations are with 3rd party APIs. Some of these APIs have rate limits. For example, Tenderly has a limit for simulations allowed per month depending on the subscription plan, Coingecko has a limit of 1000 requests/min, and QuickNode has a limit of 100 requests/sec. Although there is no publicly listed requests/min limit for Zapper.fi, there is a cost for requests. Without rate limiting IP addresses, it would be possible for anyone who knows about how the API works to spam requests and reach the rate limit, impacting availability to other users and potentially increasing the cost for the Zeal API subscription. This could then cause problems for all wallet users. Lack of rate limiting is in the 4th spot on the OWASP API Security Top 10 list.

#### Recommendation

Use a rate limiting tool such as fail2ban or the more modern crowdsec for requests to the

backend API. Add alerting to notify the backend devs when the number of requests has exceeded a certain level, for example 75% of the rate limit, so that appropriate action can be taken if necessary. Using a service like Cloudflare can provide some protection against a DDoS attack.

## Low Frontend Findings

### 1. Low - Privacy concern from pre-login localStorage

#### Summary

localStorage data has no expiration time, unlike sessionStorage. The localStorage of the wallet extension before a user logs in contains substantial plaintext data about the accounts that are in the wallet. Because this data is visible even without knowing the wallet password, it may leak information to unauthorized parties who gain access to this information.

#### Vulnerability Detail

If a wallet has been used in the browser in the past, the Zeal wallet stores some information about the wallet, such as the addresses that exist in the wallet. The localStorage of the extension contains substantial amounts of information about the wallet even before the correct password is entered to unlock the wallet. This data includes:

- Addresses in the wallet
- How many different private keys and secret phrases are stored in the wallet (this prevent plausible deniability)
- dApps history (because icon URLs are cached)
- keystore KDF parameters for addresses restored as a v3_keystore
- encryptedPhrase, IV, and salt (this item is the least concerning because this information is necessary to decrypt the key information)

#### Recommendation

Consider an approach where all data in localStorage is encrypted. The sensitive data related to user accounts can be stored in one localStorage variable. The other cached data can be encrypted (to prevent unauthorized read/write operations) and the decrypted cached data can be stored in a sessionStorage variable in the same way that the decrypted sessionPassword is, so that this data is cleared at the end of the browsing session or when the user logs out of the wallet.

To hide how many private keys are stored in the wallet, consider a redesign where the sessionPassword unlocks a master keystore which holds all the other keystores, instead of the same sessionPassword unlocking all individual keystores. This would mean only one keystore is visible regardless of how many accounts are in the wallet, preserving some amount of plausible deniability. This would also postpone the need to standardize the keystore types (which should be done at some point anyway) because all keystores would be stored in a single keystore.

## 2. Low - Dependency improvements

#### Summary

Every dependency increases the possible attack surface of the system. Some dependencies do not appear to be used and should be removed. Some dependencies in package.json are likely only used for development but are not listed under “devDependencies” and are instead under “dependencies”.

#### Vulnerability Detail

A brief high-level comparison of the Zeal wallet with a few other wallet repositories showed that the Zeal wallet has more than double the number of dependencies of some other wallets. A large number of dependencies can make the code more difficult to audit, increases the chances of a dependency becoming a victim of a supply chain attack, and increases the odds of unwanted negative interactions between different dependencies. On top of all that, every additional dependency is a possible entry point for a malicious party if there is a vulnerability in that dependency.

One tool to help reduce the number of dependencies is depcheck, which checks whether there are unused dependencies in a project. It is not perfect, but can use a custom configuration in the event of false positives. The depcheck output for the wallet shows a number of potentially unused dependencies.

```
Unused dependencies
* @ledgerhq/hw-transport-webusb
* @ledgerhq/logs
* @types/jest
* @types/keccak
* buffer
* chrome-types
* crypto
* crypto-browserify
* ethers
* jest-puppeteer
* keccak
* stream
* web-vitals
Unused devDependencies
* @sentry/cli
* @storybook/addon-actions
* @storybook/addon-essentials
* @storybook/addon-interactions
* @storybook/addon-knobs
* @storybook/addon-links
* @storybook/builder-webpack5
* @storybook/manager-webpack5
* @storybook/node-logger
* @storybook/preset-create-react-app
* @storybook/testing-library
* babel-plugin-named-exports-order
* node-sass
* prop-types
* storybook
* typed-scss-modules
```

Another way to reduce the dependencies in the project is by verifying that no development dependencies are accidentally categorized as production dependencies. There are some dependencies in the wallet's package.json that look miscategorized. The

devs should review package.json dependencies very carefully to reduce the size of the list as much as possible. Some dependencies that are likely only needed for development and should be removed from the “dependencies” section of package.json include:

- jest-canvas-mock
- jest-image-snapshot
- jest-puppeteer

There is a TODO comment indicating that the ethereumjs-wallet dependency will be replaced with the web3 library. This is a good change and should ideally be implemented before production release. Additionally, support should be considered for the new 4.x.x release of the web3 library. On the other hand, be aware that the web3 functions used by the wallet have not been audited according to the warning at the top of the documentation.

The dependencies should also be reviewed manually to determined whether any are unimportant and could be removed in future refactoring, or replaced with a component from an already imported library. For example, slate is described as a “framework for building rich text editors” but it is only used in one file for restoring accounts from a secret phrase. It is likely that a simpler and smaller dependency could replace slate for this one purpose.

#### Recommendation

Follow the recommendations provided by depcheck and remove the suggested dependencies. If depcheck makes a mistake and a dependency is removed when it is actually needed, good test coverage should quickly identify this. Once the depcheck output shows zero unnecessary dependencies, by resolving all outputs or by adding a custom depcheck config file if there is a false positive, adding depcheck to the CI pipeline would help to avoid unnecessary dependencies in the future. For specific examples, remove the ethereumjs-wallet and slate dependencies when convenient.

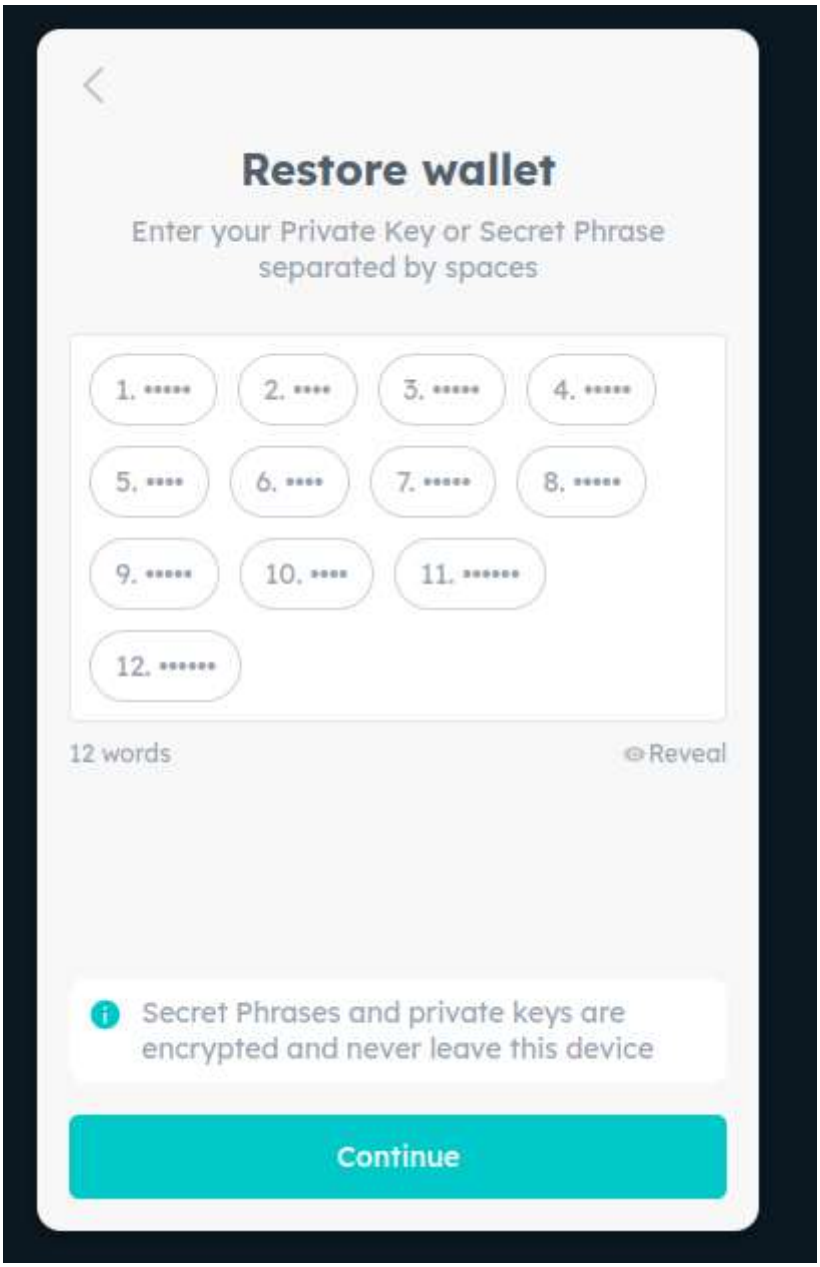
## 3. Low - Minor entropy leak

#### Summary

The wallet has a nice feature of automatically hiding the secret words when they are pasted into the wallet. If the goal is to minimize an information leak in the event that another party is monitoring the screen, the length of each secret word should be the same, but this is not currently the case.

#### Vulnerability Detail

The wallet allows user's to restore an account using the BIP39 secret words. These values can be pasted or typed manually. In both cases, the hidden version of these words uses the same number of characters as the original word it is hiding. This is a minor entropy leak that reduces the search space if a malicious party is trying to brute force the secret words. Because the length of BIP-39 secret words can range from 3 characters to 8 or more characters, the correct words become significantly easier to brute force with the length information.



#### Recommendation

It would be better to avoid any entropy leak when the values are hidden by keeping all the secret words the same length. Maintain a constant length, say of four characters, so every value looks consistent like ••••.

### 4. Low - Unencrypted HTTP links in extension

#### Summary

There are two links in the wallet with HTTP URLs, not HTTPS. The links redirect to HTTPS websites when visited, but this should still be fixed before production.

#### Vulnerability Detail

HTTP is an unencrypted protocol and can lead to man-in-the-middle (MITM) attacks as well as other security issues. Only HTTPS should be used if security is remotely of concern. There are two HTTP links hardcoded into the app. Although they redirect to HTTPS sites, this should still be fixed in the wallet. The links are:

- http://zeal.app/privacy-policy in Layout.tsx
- http://zeal.app/terms-and-conditions in Layout.tsx

#### Recommendation

Only use HTTPS links in the wallet.

### 5. Low - package.json improvements

#### Summary

The package.json file could be modified in ways that may provide small security benefits.

#### Vulnerability Detail

The wallet's package.json could be improved in a few ways:

- engines: Make sure the proper version of node is used. Even though the devs may all have the same version of node installed, this could be useful to confirm that the CI pipeline is using the right version of node.
- The browserlist values should be revised because the extension only supports Chrome but other browsers are named. The version numbers specified for browserslist are quite outdated, with some version several years old. It may be better to support only newer browsers due to known vulnerabilities in older versions.
- The proxy field only indicates one URL, but request.ts indicates two URLs. The proxy field might need modification before production.

- Specify version numbers for dependencies in package.json that currently redirect to other packages. For example, this line with `"stream": "npm:stream-browserify",` does not indicate a version number for this dependency.

#### Recommendation

Consider revisions to the wallet's package.json as described above.

## 6. Low - Dependency confusion supply chain risk

#### Summary

The dependency `eslint-plugin-zeal-domains` is a private package that the wallet depends on. Depending on the build process, a public package of the same name may be deployed on npm and get used instead of this private package, injecting malicious code into the extension.

#### Vulnerability Detail

The npm URL corresponding to the name `eslint-plugin-zeal-domains` is not reserved yet. This would allow a malicious party to register that package name and host a package with malicious code. The novel "dependency confusion" attack vector describes how a public npm package can be published with the same name and a higher version number to take priority over an internal package with the same name.

#### Recommendation

Use one of the several mitigations to protect against this attack vector. Consider referencing the Microsoft whitepaper for more details. Another option is to publish the `eslint-plugin-zeal-domains` package to npm.

The LlavaMoat sandboxing library, built by Metamask, is specifically designed for supply chain attack prevention and could provide useful security benefits if integrated properly.

If other internal packages may be added in the future, consider adding a CI scan from a dependency confusion detection tool such as confused or confuser.

## 7. Low - Wildcard postmessage may allow data interception

### Summary

window.postMessage() is designed to safely enable cross-origin communication between Window objects. If a wildcard is used for the targetOrigin value, then any window origin can dispatch and receive the message.

### Vulnerability Detail

A wildcard postmessage call was found. This issue was caught with this semgrep rule. Mozilla documents this issue and it is explained well in this blog post.

### Recommendation

Add a semgrep to the project CI to run a scan of frontend files on every commit. Consider avoiding wildcard postmessage calls.

## 8. Low - Privacy concern from inability to set RPC endpoint

### Summary

Privacy concerns of browser wallets have recently been under discussion. One of the features that privacy-conscious users want is the ability to set a custom RPC endpoint to avoid using Infura or other endpoints that log user data. The Zeal wallet does not yet have this feature.

### Vulnerability Detail

All RPC requests from the Zeal wallet are sent to the Zeal backend. This means users of the Zeal wallet must trust the Zeal backend for any RPC requests, and it is unclear to the average user if Zeal's backend logs IPs, wallet addresses, user actions, and other information. Even if Zeal's code is trusted, some users may not trust Amazon, where Zeal's backend is hosted. The inability to set a custom RPC endpoint prevents users from using a privacy-conscious RPC provider, like routing RPC requests through the Nym privacy network or using their own trusted Ethereum node. Metamask has recently introduced features like this.

### Recommendation

Implement a feature to allow users to opt out of trusting the Zeal backend for RPC calls.

## 9. Low - Icon requests can leak data

### Summary

When the wallet is used with a dApp, the wallet requests an icon for the dApp from icons.llama.fi to display in the wallet. This request effectively leaks what dApps are used to

a 3rd party entity.

#### Vulnerability Detail

When the wallet is inserted into the dApp webpage, it shows an icon of the dApp. This icon does not come from a request to the dApp, but from a request to icons.llama.fi. The URL is hardcoded in several locations in the backend. This causes the wallet user to trust a 3rd party with the information of which dApps a certain IP address has visited, because icons.llama.fi (or servers on the path to this endpoint) may log this information. DuckDuckGo previously got into a public debate with a similar approach, though the icons were served by DuckDuckGo directly and not a 3rd party.

#### Recommendation

Consider removing the icon, icon caching, and corresponding 3rd party requests from the wallet UI. Another option is to serve the icon from the Zeal backend instead of from icons.llama.fi, because this would not reveal the user's IP address to a 3rd party. This would limit the number of trusted parties because the current wallet design, where all RPC requests are routed to Zeal's backend, means Zeal is already aware of the user's actions and IP address from other API requests.

### 10. Low - User has no easy way to clear icon caching

#### Summary

When the wallet interacts with dApps, the wallet displays an icon for the dApp. The external URLs for these icons are cached in the localStorage. There is no way to clear this information from the extension. localStorage data has no expiration time, unlike sessionStorage. This history of visited dApps cannot be easily cleared and could be a privacy concern for some users.

#### Vulnerability Detail

If a wallet has been used in the past and the user has visited the webpages for different dApps, the URLs for the dApp icons are stored in the wallet's localStorage. This include the dApps that the user visited but never interacted with, unlike dApps with user interactions (which would be part of the public record on the blockchain). This list of dApps that users have visited is similar to a user's browsing history, but unlike browsing history, there is no easy way for a user to clear this dApp history. Even using an incognito window will not remove this information.

The caching of dApp icons could be a privacy concern for some users because:

1. The list of cached dApp icons is not possible for a user to clear, unless they manually modify the data in the localStorage

2. The list of cached dApp icons is visible in localStorage before the wallet is unlocked with the proper password

This is a problem because:

1. This information should not be accessible before the wallet is unlocked
2. A privacy-conscious user should be able to choose whether this history is recorded in the first place
3. A privacy-conscious user should be notified this data is saved and be able to easily clear this history when they want

The icon cache is not even cleared when an account is removed from the wallet. Even if the wallet has only had one account performing all transactions and that address is removed, the wallet still stores the dApp data but changes the type to disconnected. Instead the wallet should remove the dApp from the wallet completely.

#### Recommendation

Remove the caching of app icons if this privacy vector is a concern. Don't store this data in plaintext in localStorage. If the data is stored in localStorage, it:

1. should be encrypted, not stored in plaintext
2. should provide the user with an option to manually clear the wallet's cache when they choose to do so
3. should properly remove dApp data when an account is deleted
4. could be removed regularly, such as on browser shutdown and/or startup, which is similar to how sessionStorage works

## 11. Low - Cryptographic protection of account data

#### Summary

There are different ways that accounts are stored with keystores. The security of these different approaches may vary because they rely on slightly different dependencies and encryption algorithms. Accounts should be stored consistently with the highest level of security available. A minor improvement in generating the `sessionPassword` for new accounts may also be possible.

#### Vulnerability Detail

At a high level, the key material stored in the wallet is generated in the following sequence:

1. User creates a password for their wallet. The password is known only to the user. The password is used to encrypt the sessionPassword

2. The sessionPassword is generated using UUID v4 from RFC 4122. This sessionPassword is used to encrypt the keystores.
3. The keystore holds the privacy key material for the accounts. There is more than one type of keystore.

The getPrivateKey implementation in keystore includes several types of keystores. These keystores store their data in different ways with different libraries. This means there are different security guarantees depending on the type of keystore that the account uses. This increases complexity and attack surface due to a non-uniform approach to accounts. These two keystore implementations are compared below:

| | v3_keystore | secret_phrase_key |
|---|---|---|
| Library | web3 library | metamask/browser-passworder |
| Encryption algorithm | AES-128-CTR | AES-GCM |
| Encryption library | `cipher.update` | `crypto.webcrypto.subtle.encrypt` |
| IV generation function | `cryp.randomBytes` | `global.crypto.getRandomValues` |
| Salt generation function | `cryp.randomBytes` | `global.crypto.getRandomValues` |
| KDF | scrypt | PBKDF2 |
| KDF hash function | SHA-256 (per scrypt specifications) | SHA-256 |
| Audited | No | Maybe indirectly |

The wallet password is encrypted using metamask/browser-passworder and therefore has the same properties as the secret_phrase_key column in the previous table. Put another way, sessionPassword is encrypted by the user password in the same way that the secret_phrase_key keystore data is encrypted by the sessionPassword. The v3_keystore keystore is the odd one out.

When a new account is added to the wallet, a new `sessionPassword` value is generated. The `sessionPassword` value is from the uuid npm package and follows version 4 of RFC 4122. The uuid library does advertise “Cryptographically-strong random values”. Under the hood, the uuid library generates version 4 uuid values using randomness from

`crypto.randomFillSync`. This is a reasonable approach, but there is an open issue in the uuid library with a discussion about the security of the uuid generation in the scenario where a malicious party can read memory. More specifically, a related issue discusses switching to `crypto.randomUUID`, which added a function to generate version 4 UUID values. This switch is even suggested elsewhere. There, the uuid library dependency can be removed entirely and be replaced with `crypto.randomUUID`. The `crypto.randomUUID` function is actually the function that the uuid library uses under certain criteria. Like the uuid library, `crypto.randomUUID` receives randomness from `crypto.randomFillSync`, but it only consistently does this in the case where the entropy cache is disabled. Consider whether the `crypto.randomUUID` function would be a reasonable replacement to the uuid library. It could remove a single-purpose dependency with a library that the wallet already uses, and the crypto library may be better maintained.

**Recommendation**

Several changes should be made:

1. Refactor the code to use a consistent library for all account types. If this is not possible, make sure a consistent encryption algorithm and KDF algorithm is used between accounts.
2. If the web3 library is going to be the main library for handling account data in the future, specify AES-GCM-128 for `options.cipher` as a more secure encryption algorithm in the web3.eth.accounts.encrypt function instead of the default AES-CTR-128.
3. Assess the wallet's threat model to determine whether an attacker with access to a user device's memory is a scenario that requires protecting against. If so, remove the uuid and types/uuid dependencies and replace with `crypto.randomUUID`. Consider enabling the disableEntropyCache option for `crypto.randomUUID` if it is worth the tradeoffs.

## 12. Low - Constant sessionPassword checksum might not be ideal

**Summary**

A constant UUID value is encrypted with `sessionPassword` to ensure that `encryptedPassword` is correctly decrypted. Using a known constant value is slightly less secure if the encryption algorithm has any biases.

**Vulnerability Detail**

In theory, a perfect encryption algorithm should leak no data about its contents and should protect the encrypted data from modifications from parties without the encryption key. In reality, few algorithms are perfect. The value that is used to check if the decryption of sessionPassword happened properly is a constant UUID value of fab0dafe-7257-4dd8-b2b8-627ca07f5234. This value is used during initial encryption and every subsequent decryption. An advanced adversary that wanted to modify the encrypted sessionPassword data in a way that the decryption would successfully return the constant UUID but not the correct sessionPassword may be able to use biases in the encryption algorithm to do so. This process would be made easier by the fact that the checksum value is a constant value across all encrypted sessionPasswords. Another attack vector would be using the encryption of a constant value to derive the unknown encrypted data, similar to a known plaintext attack. If the checksum value was dynamic and dependent on the sessionPassword value, this process would be harder because the checksum value and the sessionPassword value are both unknown. Using a dynamic checksum would not only make it more difficult to modify the encrypted value, but more costly if a brute forcing mechanism was used, because the dynamic values would have to be recalculated to check the decryption process after each attempt.

**Recommendation**

Replace the constant UUID checksum value stored in `id` of `encryptedPassword` with a dynamic value. For example, make `id` equal to the SHA256 hash of the `sessionPassword`.

## 13. Low - No opt out from sentry logging privacy concern

**Summary**

Sentry logs are recorded when errors occur. But there does not appear to be any way for users to opt out of this logging. Other DeFi applications, even Remix, offer a way to opt out of such features.

**Vulnerability Detail**

There exist many TODO comments indicating more sentry logging will be added to the wallet. But there is no way for a user to opt out of this tracking to avoid additional backend

calls. Providing the user with an opt out feature is important for privacy-conscious users.

#### Recommendation

Implement all sentry calls with a flag that allows sentry calls to be skipped if the flag is enabled. Allow users to toggle this flag in the wallet settings to minimize information sent to the backend.

### 14. Low - Migrate from bip39 to scure-bip39

#### Summary

The scure-bip39 library has fewer dependencies than bip39 and is audited, which is a better security posture than the bip39 library.

#### Vulnerability Detail

The @scure/bip39 library is a more secure alternative to bip39. The most important operation that the bip39 library performs is generating a new EOA wallet. Consider switching to the more secure dependency. For reference, Metamask uses the generateMnemonic function from @scure/bip39 for wallet creation.

#### Recommendation

Replace the bip39 dependency with @scure/bip39.

## Low Backend Findings

### 15. Low - Git history stores removed secrets

#### Summary

The git history contains certain secrets that have since been removed. If these secrets are still valid, they should be revoked.

#### Vulnerability Detail

The commonly used gitleaks tool identified several files that used to contain secrets that have since been modified. These include:

- Config files with secrets (prod config, dev config, local config
- a github action

These are a few examples but there are other secrets in the git history not listed here. These secrets are exposed to all github accounts that have read access to this repository. The git history may also be exposed to certain CI tools, and CI tools have a history of getting compromised in supply chain attacks in the past.

#### Recommendation

Secrets that may have been exposed should be revoked. If there is a plan to open source this wallet repository in the future, it would be better to create a new repository without the current commit history to prevent the exposure of these old artifacts.

### 16. Low - Unescaped HTML in mustache template

#### Summary

Triple braces in a mustache template return unescaped HTML per the docs. This may enable XSS if the user can control this data.

#### Vulnerability Detail

Files including modelEnum.mustache and pojoBody.mustache use triple braces, which returns unescaped HTML. This issue was caught with this semgrep rule.

#### Recommendation

Avoid using unescaped HTML in mustache templates. Using the escaped HTML that mustache normally returns with double braces reduces the risk of a XSS vector. Add semgrep to the project's CI to run a scan of backend files on every commit.

### 17. Low - API errors leak system information

#### Summary

Some unexpected API calls leak information about the backend system. This can be useful for an attacker learning about the backend stack to create a more targeted payload.

#### Vulnerability Detail

Several API calls return errors that leak information about the system.

- URL: https://iw8i6d52oi.execute-api.eu-west-2.amazonaws.com/transaction/history/
- Response: https://javalin.io/documentation#notfoundresponse
- Problem: Indicates Java or Kotlin backend
- URL: https://iw8i6d52oi.execute-api.eu-west-2.amazonaws.com/wallet/rate/default/Ethereum/0x0/
- Response: Can't fetch coingecko data for 'ETHEREUM0x0' crypto currency id.
- Problem: Indicates data source
- URL: https://iw8i6d52oi.execute-api.eu-west-2.amazonaws.com/swagger

- Response: Swagger UI
- Problem: There is no reason this path should be exposed publicly, even if it does not load much data

Other error message in the backend that may be displayed to users but were not specifically triggered by a URL during security testing include:

- “Error occurred during trying to fetch Rarible collection...”
- “Unable to fetch Polygon GasStation fee estimations.”
- “Error fetching DappRadar list of dApps.”
- “Error fetching DefiLlama list hash protocols.”

#### Recommendation

Use generic error messages that do not expose information about the backend components. Consider replacing the names of integrated resources with error code numbers. For example, use “Account data is unavailable” instead of “Zapper is unavailable”. Add mitigations to the backend to avoid returning default errors that leak information about the backend technology stack.

### 18. Low - Arbitrary RPC calls allowed

#### Summary

The RPC call methods passed through the /wallet/rpc/ API are not whitelisted. This means that the user could use this API to call RPC methods for other purposes outside the scope of normal wallet actions.

#### Vulnerability Detail

QuickNode is the main RPC used for the backend RPC calls at the time of testing (although some Alchemy support is also found in the source). Each QuickNode API call has a number of RPC credits, or cost, assigned to it. RPC call data is taken from user input and forwarded to QuickNode without any modification. Because the RPC methods are not limited, and the user can call some RPC methods that most likely are never needed by the wallet, a user who chooses to manually call RPC methods with a high API credit value may exhaust the available credits and create a denial of service (DoS) if the RPC is not reachable. Even if the Zeal backend can handle a large number of unnecessary requests, QuickNode may start rate limiting or other measures on the Zeal access to QuickNode as a side effect of this. There is no reason for certain RPC methods to be passed from the API to QuickNode, so these should be blocked by the backend. Some of unnecessary RPC calls that work but should be blocked include:

- txpool_content (Note: costs 500 API credits)
- txpool_inspect
- web3_clientVersion
- net_version
- net_peerCount

**Recommendation**

Consider only allowing whitelisted RPC methods received by the API to be passed on to the RPC nodes. Any RPC methods that are received but not whitelisted should return an error.

## 19. Low - Avoid concatenating strings to build URLs

**Summary**

Concatenating strings to form URLs can be dangerous, especially in the event that some of the values concatenated are dynamic or contain user input. There are safer options available than using the "+" operator to concatenate strings.

**Vulnerability Detail**

Several backend Java files, like SanitizedUri and RaribleClient, concatenate strings to form a URL without sufficiently thorough safety checks. These are two examples but there are likely other instances of this because an exhaustive search was not performed This could lead to the crafting of a URL that accessing unintended backend resources (a SSRF attack). Even in the event that none of these strings are directly controlled by end users, if the strings are dynamic in any way, it may still be preferable to use security best practices to concatenate strings to form a URL. A small but potential not fully sufficient improvement in this particular instance may be to use this alternative of `URI(String scheme, String authority, String host, String path, String fragment)` instead of using `URI.create(String str)` because URI does some validation on these input arguments.

**Recommendation**

If the URL is derived only from static hardcoded values, then no changes are needed. There are multiple approaches to mitigating potential SSRF vectors and insecure URL concatenation, but the proper solution depends on how much of the URL is controlled by user input (directly or indirectly). Refer to OWASP's SSRF prevention cheat sheet as a starting point.

## 20. Low - No Redis auth performed by backend

**Summary**

Redis docs show how to connect to Redis using jedis, but the Redis docs show an authentication step that doesn't appear in the backend source. Consider requiring authentication for connections to Redis.

#### Vulnerability Detail

The Redis docs show the steps to connect to Redis with jedis. The line of code `jedis.auth("password");` doesn't appear in the backend, and when `new JedisPool()` is used in PersistenceModule it does not have a password value. Instead, the backend can set and get Redis value without authentication. Even if the backend has a hardened exterior, it is a best practice to enable secure authentication for all components of the system, even if they are hidden away in a backend.

#### Recommendation

Add secure authentication to the Redis instance and modify the backend Java code accordingly. More Redis security best practices are explained in the Redis docs.

### 21. Low - Input from integrations should be considered untrusted

#### Summary

User input should always be considered untrusted inputs that can lead to unexpected edge cases, but malformed or malicious input from external integrations can have a similar impact. Because certain data from these integrations get displayed to the user in the wallet, this risk vector could lead to security impact of the private keys held in the wallet.

#### Vulnerability Detail

There is a general lack of validation of data received from integrations. Every piece of data returned from an integration should be checked for potential malicious input. If the return value is an integer, boolean, or similar type that rarely contains potentially malicious values, then a type check may be sufficient. If the return value is a string, then the string should be treated as user input and have thorough validation performed. One example of this lack of validation is in PolygonGasStationClient, where received data is deserialized with `objectMapper.readValue()` from com.fasterxml.jackson.databind. Deserialization weaknesses in Java have been common in recent years, and the jackson-databind library specifically has had many CVEs published. A better example is in RateProvider, where `fetchDefaultFiatRate()` treats the return value as an integer as performs mathematical operations on the data returned from Coingecko. If the return value is not an integer as expected, then an error will be thrown rather than the input being trusted. All return values received from `httpClient.get()` should be treated more like this.

#### Recommendation

Perform input validation on any data returned from integrations. For string values, reference OWASP's Input Validation Cheat Sheet for a possible starting point. For data deserialization, see OWASP's deserialization cheat sheet.

## Info Frontend Findings

### 1. Info - Consider removing clipboardRead permission

#### Summary

Other wallet extensions with similar features don't use the clipboardRead permission, meaning a different solution for this functionality can allow it to be removed. This may be a tradeoff between a design feature and a security feature.

#### Vulnerability Detail

The manifest.json permissions determine what special privileges an extension has. Similar extensions do not have the `clipboardRead` permission, so an alternative solution to implementing this feature should allow the wallet to reduce the privileges it asks for. This permission is only needed for `document.execCommand('paste')`, which appears in only the one file getClipboardText.ts. Consider modifying the implementation to allow this permission to be removed.

#### Recommendation

Consider removing the `paste_button` to allow this permission to be removed. This would make a user manually paste their data instead of clicking a special button.

### 2. Info - Remaining TODO comments in frontend

#### Summary

The wallet has several TODO comments indicating unfinished features. These comments may point to less polished parts of the project where there is a higher chance of security issues.

#### Vulnerability Detail

There are many places with TODO comments. In particular, the icon value for tokens other than ETH that are cached in localStorage also appears as the four letters "TODO". This are not necessarily a security issue, but incomplete code or code that will soon be changed can provide information to malicious parties watching the project for changes that introduce weaknesses.

#### Recommendation

Before a production release, remove all comments from the code to avoid disclosing information about unfinished features. Make sure localStorage doesn't store TODO values.

## 3. Info - Wallet values round down

**Summary**

The wallet values are rounded down to the nearest whole number. This can underestimate the funds held in the account.

**Vulnerability Detail**

If an account hold 5 USDC but the price of USDC is 0.9999, the wallet will show a value of $4 of USDC is held in the account. This can also lead to a case where the individual value of the assets, each one rounded down, does not sum to the total value of the wallet.

**Recommendation**

Consider a more precise representation of value for individual tokens by default, or add a "Show value decimals" option to show more precise token values. It would help to document that values are rounded down to prevent users from asking why their listed value of tokens is less than reality.

## 4. Info - Some tokens are not visible

**Summary**

If a wallet holds some dust, say under $0.1 worth of a token, the wallet does not display these tokens. Testnet tokens are not shown at all.

**Vulnerability Detail**

The wallet does not currently support testnets, which is understandable. But it might help to allow users to show or hide tokens below a certain amount of value.

**Recommendation**

Similar to the previous issue, consider modifying the values so they do not round down, because values under $1 would round down to zero and become hidden. Consider adding support for testnets at a future point in time.

## 5. Info - Lack of secret rotation

**Summary**

The wallet does not require regular rotation of the user password (in fact, the user is unable to change their wallet password) and does not regularly rotate the session password (AKA encryptedPassword).

#### Vulnerability Detail

Some sources recommend rotating cryptographic material and other sources suggest passwords without secure 2FA should be deprecated. This offers some benefit, but unlike a normal breach of a password where the account may be accessed quietly for months, a breach of a wallet's private key is often catastrophic and could cause all value to be drained almost immediately, so secret rotation may hold less value in this scenario.

One approach that may make sense is to regularly change the sessionPassword. Currently the wallet uses one sessionPassword for the entire lifespan of the wallet. What could be done instead is that every time the user logs in, the sessionPassword is decrypted, the keystores are decrypted, and then the sessionPassword is replaced with a new uuid value with which the keystores are reencrypted. This approach is likely overly complex at the moment, but the benefits of this approach might be worth weighing in the future depending on the attack models that the wallet is designed for.

#### Recommendation

The decision around rotating secrets is a design decision, and it may make more sense to follow the design decisions of existing wallets or follow market demand rather than guarantee the highest level of security but with an onerous, unfriendly login process.

### 6. Info - Low password entropy requirement

#### Summary

The current password requirements allows the user to choose a low entropy password like Password1 or Aaaaaaa1.

#### Vulnerability Detail

Because the wallet password serves almost the same purpose as the password to a bank account, consider raising the entropy requirement for the wallet password. Consider adding a package that measures password entropy to encourage or enforce higher entropy choices for the wallet password. Popular packages that provide this feature include check-password-stength and owasp-password-strength-test. A related but older package is zxcvbn which determines how guessable or crackable a password may be.

#### Recommendation

Encourage or enforce higher entropy password choices when the user is creating their wallet account. Although NIST security guidelines on passwords don't require this, it may still be useful for this particular case. Multifactor authentication support may also be a useful feature to add in the future.

### 7. Info - Potential license compatibility issue

#### Summary

If the wallet source code is made open source in the future, it may need to use a copyleft license because at least one of its dependencies uses MPL-2.0.

#### Vulnerability Detail

A copyleft license is less flexible than other more permissive licenses. Using a dependency with a copyleft license can limit the ways in which the source code of a project can be distributed. The ethereumjs/tx dependency uses a MPL-2.0 license, which is considered a copyleft license. Use `yarn licenses list` to view dependency licenses.

#### Recommendation

License choice is a very important decision for projects that is often not given the thought it deserves. The report author is aware of the pain that a license change process can cause.

### 8. Info - General access control security best practices

#### Summary

Supply chain attacks are a concern for high-security products because the attack surface can be large. Maintaining secure access controls is a key step in protecting against such attacks.

#### Vulnerability Detail

The entire build and release system should be assessed for weak points, because this was out of scope of this review. One starting point is to verify that every github account that has write access to the wallet repository has 2FA enabled in their github account. There are past examples of breached git accounts committing vulnerable code to allow attacks access to systems that deploy the vulnerable version. Even with 2FA enabled, the github account passwords of the users with write access should regularly change their password. Another security measure that can be taken in the github account is using a branch protection rule to require a certain number of approvals before merging. The CI, testing, and build process is another key step that should have proper security controls added.

The infrastructure involved in updating the Zeal wallet extension in the Chrome extension store is also crucial. There have been past examples of compromised developer accounts leading to malicious extension updates. The account(s) that can publish new extension versions to the Chrome extension store should have 2FA enabled (which should be already required by Google).

#### Recommendation

Consider the security of all accounts and endpoints related to the build and deployment process of a new version of the wallet. The security of the supply chain is only as good as its weakest link.

## 9. Info - Add security static analysis into CI pipeline

#### Summary

Integrating security into the development process is important when building a system that requires high levels of security. Security should not be left to the end of the development process. Adding continuous integration (CI) tests for security issues, is an easy step to make to improve the system's security.

#### Vulnerability Detail

Several additions to the wallet's CI process could reduce the risk that a commit will introduce a vulnerability that goes unnoticed.

Adding a security scanner into the CI pipeline would help catch any obvious issues that could be caught automatically. A multilanguage scanner like semgrep of SonarQube can be used to test the frontend and backend files together. More specialized tools can be used for specific languages, like this JS security scanner or this Java static analyzer.

There are several additional ESLint plugins that may be useful to add for catching issues in this project:

- eslint-plugin-no-unsanitized
- eslint-plugin-security
- eslint-config-react-security
- eslint-plugin-react
- eslint-plugin-react-hooks
- eslint-plugin-storybook
- eslint-plugin-jest
- eslint-plugin-import
- eslint-plugin-node

No Regex DoS expressions were found, but consider adding a tool for regex DoS scanning into the CI. One tool that may work is regexploit-js.

The confused or confuser tools help to detect dependency confusion vulnerabilities. This may also be useful to test for in the CI.

#### Recommendation

Consider adding static analysis security tools and addition eslint plugins to the CI pipeline. Also consider running static analysis tools and linters on the node_modules directory, to make sure the dependencies follow the best practices.

## 10. Info - Yarn wallet scripts mismatch

#### Summary

The wallet has two scripts, upload-sourcemaps.ts and upload-to-google.ts. The scripts make slightly different assumptions about where the build directories are.

#### Vulnerability Detail

upload-sourcemaps.ts and upload-to-google.ts in frontend/wallet/scripts rely on a build directory above the wallet directory. upload-to-google.ts assumes the build directory is located at "../.." and that value is hardcoded into the script. upload-sourcemaps.ts assumes the path will be passed as an input argument. The scripts should use the same assumption about where the build directory is. Note that accessing directories above the current directory can lead to security issues in some cases, but the existing usage with the current directory structure is fine.

#### Recommendation

Modify upload-to-google.ts to accept an input argument to set the expected location of the build directory.

## 11. Info - Logged in wallet may never log out

#### Summary

The purpose of the login page of a wallet is to only allow the authorizes user access to the wallet. But if the wallet remains logged in all the time, the security of the login feature is effectively removed.

#### Vulnerability Detail

If a user logs into a wallet and never logs out or closes the browser, the wallet will remain logged in because there is no session timeout. In web applications, a session timeout is a normal security feature to forcibly log out users who have been logged in for too long.

#### Recommendation

Consider adding a timeout to log out a user from the wallet if the wallet remains logged in for a certain timeframe, such as 7 days.

## 12. Info - Versions not pinned

### Summary

The package.json of the wallet uses caret ranges for semver versioning, which means the versions of dependencies are not fixed. This could lead to a compromised dependency with malicious code finding its way into the wallet if the maliciously modified package receives a new package number.

### Vulnerability Detail

When the extension is in the Chrome store, the dependencies are fixed until a new version of the extension is released to the Chrome store. But when the new extension is pushed to the store, are the dependencies for the wallet built immediately before the new extension is packaged for the store? If so, it's possible a dependency could have a malicious update that adds unexpected code to the wallet right before a new wallet version is pushed to the store.

The first issue in this old audit of Metamask suggests this issue is critical, but I disagree with this severity.

#### Recommendation

First work on reducing the number of dependencies in the wallet. Then consider pinning dependencies to an exact version when the wallet is in production.

### 13. Low - Implementation variation

#### Summary

A minor implementation difference exists when the lock button is pressed to logout of the wallet.

#### Vulnerability Detail

The `on_lock_zeal_click` case calls `chrome.storage.session.remove('password')`. Other similar operations instead call `chrome.storage.session.clear()`. While only password may be stored in the sessionStorage now, clearing all chrome storage would ensure security if future implementation changes modify what is stored in chrome storage.

#### Recommendation

Use `chrome.storage.session.clear()` instead of `chrome.storage.session.remove('password')`.

### 14. Info - In-memory storage may be more secure

#### Summary

While sessionStorage is a convenient location to store sensitive data, in-memory data storage can offer some security benefits.

#### Vulnerability Detail

A couple of security sources (1, 2) suggest that in-memory storage can be a more secure option than sessionStorage for sensitive data like the decrypted sessionPassword value. The main vector that this provides protection against is XSS. Switching the wallet to use in-memory storage comes with its own risks and extra complexity. Such a switch should not be considered a priority, but may offer benefits depending on the attack model and attack vectors that are prioritized.

#### Recommendation

Consider using in-memory storage of decrypted secrets in the future, like the sessionPassword value.

### 15. Info - Typos

#### Summary

There are some minor typos that could be fixed, though they are unrelated to security.

**Vulnerability Detail**

Revise the following:

- cancelSubmitedToSubmited -> cancelSubmittedToSubmitted
- parsePBKDFParamsOut -> parseSCryptParamsOut, because the KDF used for this key material is Scrypt not PBKDF
- secretPharaseMap -> secretPhraseMap
- on_encrypted_secret_phrase_submited -> on_encrypted_secret_phrase_submitted
- word_mispelled_or_invalid -> word_misspelled_or_invalid

A minor consistency nitpick is that the Keystore index.js uses `web3.eth.accounts` in one place but `new Web3().eth.accounts` in a different place. Consider using a consistent approach and use `new Web3().eth.accounts` everywhere, including in other files.

This line is redundant and should be deleted because chrome.storage is cleared as part of logout on the previous line.

A simplification may be possible by removing PasswordCheckPopup or LockScreen because the logic of the files are identical.

**Recommendation**

Fix typos.

## 16. Info - Consider using an alternative to JSON.stringify

**Summary**

JSON.stringify is used in several places. Depending on the data source, the possibility of circular references or other issue may point to an external library being a better choice than JSON.stringify.

**Vulnerability Detail**

JSON.stringify can introduce some problems. XSS vulnerabilities are one possible problem with JSON.stringify, but the return value of JSON.stringify most likely is not used in a script context in the wallet. Another problem that can occur are circular references, for which a 3rd party library might improve support for.

**Recommendation**

If there is no need to change the JSON.stringify usage after assessing the comments about, leave the code as it is. If a change may help, consider using json-stringify-safe.

## 17. Info - Consider replacing mnemonic validation with bip39.validateMnemonic

**Summary**

The bip39 library contains a built-in validateMnemonic function that performs a similar task to the validation file does. Because the dependency is already imported, there many be benefits to using its validateMnemonic function.

**Vulnerability Detail**

Consider comparing the implementation details of the custom mnemonic validation with the bip39 library implementation of validateMnemonic. Even if the custom implementation is preferred, the bip39 implementation may have some details that could be included in the custom implementation, like improved support for multiple language wordlists.

**Recommendation**

Consider replacing or augmenting the mnemonic validation code with what the implementation in bip39.validateMnemonic.

## 18. Info - Wallet EOA generation only works with 1 recovery phrase

**Summary**

It is only possible to create an EOA with one recovery phrase from the wallet. After this

EOA is created, further EOAs can only be derived using the same recovery phrase.

#### Vulnerability Detail

After a new wallet EOA is created, the ability to create a new wallet EOA with a new recovery phrase disappears. The only option remaining to users is to “Quick add an account”, which derives a new address from the existing recovery phrase. This logic is hardcoded into the app with a fixed BIP44 offset and account label. Some users many want the added security offered by using different recovery phrases to generate accounts, so that some accounts remain safe even if another is compromised.

#### Recommendation

Consider allowing more than one EOA recovery phrase to be generated per installation for increased security and storage flexibility of user funds.

### 19. Info - URL for prod and dev environment exposed

#### Summary

The production and development API URLs are exposed in the extension. Ideally the production extension should not contain the development URL, and the development extension should not include the production URL.

#### Vulnerability Detail

The two different backend environment URLs are hardcoded into the extension. Even if the extension repository is not open sourced, the extension source code can be examined in Chrome developer tools to reveal the two different URLs. The final extension published to the chrome store should not include the development environment URL.

#### Recommendation

Avoid hard coding the URLs into the source. Instead consider pulling the necessary URL from an environment variable or config file that is not packaged with the extension.

## Info Backend Findings

## 20. Info - Backend could include caching headers

**Summary**

The user's web browser can cache a local copy of data returned from the web server. This cache could be accessed by other users on the system, even those without access to the wallet password. In order to fully protect the user's privacy, backend data should not be cached.

**Vulnerability Detail**

Applications should return caching directives instructing browsers not to store local copies of any sensitive data. The web server should include the following HTTP headers in all responses that should not be cached: Cache-control: no-store Pragma: no-cache

**Recommendation**

Consider modifying the backend server configuration to avoid caching user data.

## 21. Info - Undocumented API endpoint

**Summary**

The OpenAPI documentation does not include the /healthcheck endpoint.

**Vulnerability Detail**

All entry points to the backend system should be clearly documented, even if the entrypoint does not appear to be a security risk. The /healthcheck endpoint is not documented in the wallet.yml OpenAPI documentation.

**Recommendation**

Document the /healthcheck endpoint properly.

## 22. Info - Backend blacklists are very limited

**Summary**

The blacklists stored in backend variables, used to protect users against scams, only one or two entries. To provide a reasonable amount of safety to users, a more extensive list of scams should be in the blacklist.

**Vulnerability Detail**

The `BLACKLISTED_HOSTS` variable only contains the value "marketplace.geodb.com" which is hardcoded. Similarly the `blacklistedAddresses` variable has only two addresses. Protecting wallet users from only one or two scam projects is not very useful and this like should be expanded to improve user safety.

#### Recommendation

Expand the list of BLACKLISTED_HOSTS to include other crypto scam projects to increase the protection the wallet provides. Identify useful data sources to learn about scams, possibly by examining how other wallets integrate similar safety features.

### 23. Info - Poor effectiveness of checkSuspiciousCharacter()

#### Summary

The backend checks if a dApp URL has any “suspicious characters”, which currently means any non-alphanumeric characters. This check has some merit but can be considered somewhat weak.

#### Vulnerability Detail

The checkSuspiciousCharacter() logic assumes a URL that has a non-alphanumeric character is suspicious, but this logic is not very precise. The W3 made it possible for URLs can contain non-latin characters. While most dApps currently focus on English language users, this is not true of all dApps. A malicious dApps that has non-latin characters is likely using non-latin characters to appear similar to a legitimate well-known dApp as part of a phishing attack. Such look-alike scams might not even use non-alphanumeric characters, but instead may replace the letter “o” with a zero. One way to detect this specific type of attack is to check how similar the URL is compared to well known dApps. Common ways of comparing string similarity are Levenshtein distance and Hamming distance. There are Java libraries available that can do this comparison, but a database of known apps will be needed to perform this comparison, which may add latency. This is one simple improvement idea, but there are undoubtedly many others factors that can also be considered.

#### Recommendation

Consider improving the checkSuspiciousCharacter() logic.

### 24. Info - Weak origin test

#### Summary

A wildcard test of the Origin header is used in JavalinFactory. A stricter value could be used.

#### Vulnerability Detail

While JavalinFactory uses a wildcard origin test, because the origin should always start with “chrome-extension://”, a stricter test may be possible to prevent spamming of the backend.

#### Recommendation

Consider a stricter origin test.

### 25. Info - Remaining TODO comments in backend

#### Summary

The backend has several TODO comments indicating unfinished features. These comments may point to less polished parts of the project where there is a higher chance of security issues.

#### Vulnerability Detail

There are many places with TODO comments. These are not necessarily a security concern, but incomplete code or code that will soon be changed may provide information to attackers.

#### Recommendation

Resolve and remove all TODO comments in the code. Before a production release, remove all comments from the code to avoid disclosing information about unfinished features.

## Final Remarks

The wallet's security is roughly as expected at this point in the development process. Some design changes are needed to enhance the wallet's privacy and security stance. Adding security testing in continuous integration will improve the wallet's security during future development. Ideally documentation should be written as the wallet is developed to make sure no nuanced details are forgotten and to make sure documentation is available for new developers joining the team or external security audits like this one. One worthwhile exercise to do is competitive benchmarking. Several days could be spent assessing the design choices of other browser wallets, especially in regards to how data is stored and which data is stored in certain storage locations. Even wallets that are not open source can be installed in the browser and observed with chrome developer tools.

Note that one attack vector that was not examined in extreme depth but could be a high risk vector is XSS attacks, because a XSS attack can read locally stored data (local storage, sessionStorage, etc.). Because the overall security of the user's device is a key part of wallet security, it would be worth doing an exercise to assess the attack model and primary attack vectors that the wallet should be designed around. This exercise can create a better framework for assessing which mitigations take priority and which ones might not be necessary under certain assumptions. A diagram showing one attack vector is in the beginning of this repo. If the team improves their understanding of historic chrome

extension security weaknesses and supply chain attacks, everyone will have a better understanding of how to mitigate against these types of security issues when implementing new features.