Least Authority
PRIVACY MATTERS

Stacks Wallet Extension
Security Audit Report

Hiro

Final Report Version: 29 April 2021

Table of Contents

Overview

Background

Hiro requested that Least Authority perform a security audit and penetration testing of the Stacks Wallet Extension, a new browser extension for Chrome and Firefox that enables users to perform the following:

- Authenticate web applications with 12 or 24-word mnemonic keys;
- Set passwords for the encryption and storage of the keys client-side;
- Manage usernames registered with the Blockchain Naming System (BNS);
- View fungible and non-fungible token holdings on the Stacks blockchain;
- Send and receive tokens;
- Sign transactions with Clarity smart contracts as published to the Stacks blockchain;
- View recent transactions associated with holdings; and
- Configure node for relevant Stacks network.

The Stacks Wallet Extension is an upgraded version of Blockstack Connect 1.0.

Project Dates

- **February 10 - March 5**: Code review *(Completed)*
- **March 11**: Delivery of Initial Audit Report *(Completed)*
- **April 26 - 28:** Verification *(Completed)*
- **April 29:** Delivery of Final Audit Report *(Completed)*

Review Team

- JR, Cryptography Researcher and Engineer
- Jan Winkelmann, Cryptography Researcher and Engineer
- Jehad Baeth, Security Researcher and Engineer

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Stacks Wallet Extension followed by issue reporting, along with mitigation and remediation instructions outlined in this report.

The following code repositories are considered in-scope for the review:
- blockstack/ux
- blockstack/stacks.js
    - stacks/wallet-sdk
    - stacks/auth
    - stacks/encryption

Specifically, we examined the Git revisions for our initial review:

blockstack/ux: `8d76df98a84518bdf9a95aae910336c1d4e9da01`

blockstack/stacks.js: `20fc4fe4ff5874934a3533801a4e076b6527433f`

For the verification, we examined the Git revision:

blockstack/ux: 3a387b8e0343276e88ba21da9aa79b71d7e3b7e3

blockstack/stacks.js: 3808875ddcb22272f4884f8824c727765adcc8fe

For the review, these repositories were cloned for use during the audit and for reference in this report:

https://github.com/LeastAuthority/ux

https://github.com/LeastAuthority/stacks.js

All file references in this document use Unix-style paths relative to the project's root directory.

In addition, any dependency and third party code, unless specifically mentioned as in-scope, were considered out of scope for this review.

Supporting Documentation

The following documentation was available to the review team:

- Project Documentation: https://docs.blockstack.org/build-apps/overview
- Protocol Documentation: https://docs.blockstack.org/authentication/overview
- Audit Scope: https://paper.dropbox.com/doc/Security-audit-for-Stacks-Wallet-extension-stacksconnect-lDKO8rHBbykDyAy1KZPIU
- Visual Representation Design: https://www.figma.com/file/8CaxP6TRWskoTO9cmlvX9M/%F0%9F%93%90Connect?node-id=319%3A2

In addition, this audit report references the following document:

- J.P. Degabrield, A. Lehmann, K.G. Paterson, N.P. Smart, M. Strefler, 2011, "On the Joint Security of Encryption and Signature in EMV." *IACR ePrint Archive*, 2011, [DLP+11]

Areas of Concern

Our investigation focused on the following areas:

- Correctness of the implementation and adherence to best practices;
- Exposure of any critical information during user interactions with the blockchain and external libraries, including authentication mechanisms;
- Adversarial actions and other attacks that impact funds, such as the draining or the manipulation of funds;
- Mismanagement of funds via transactions;
- Vulnerabilities in the code, as well as secure interaction between the related and network components;
- Proper management of encryption and storage of private keys, including the key derivation process;
- Inappropriate permissions and excess authority;
- Data privacy, data leaking, and information integrity; and
- Anything else as identified during the initial analysis phase.

Findings

General Comments

System Design

The Stacks Wallet Extension system design is constrained by the security challenges inherent to all browser-based wallets. The presence of Chrome browser APIs allow malicious extensions to access critical secrets, which poses a threat to both the security of the application and the end user's assets. While browsers aim at creating secure sandboxes for untrusted code, they are also frequently vulnerable to 0-day exploits that allow malicious websites to escape the code sandbox and achieve Remote Code Execution (RCE) on the user's computer. We acknowledge that these fundamental security challenges are beyond the control of the Hiro team and that security should be considered a shared responsibility with the broader community of application developers, in addition to the users. There are currently no known solutions to the inherent vulnerabilities and security challenges resulting from the use of these applications. Nonetheless, development teams are advised to utilize all means possible to provide users with the safest framework for entrusting their assets. In addition, individual teams should educate users on potential, real scenarios in which their secrets might be compromised while using browser-based wallets.

Furthermore, the utilization of browser extensions introduces additional attack vectors, thus increasing the risk of successful attacks on cryptocurrency wallets. For example, browser extensions have suffered from multiple extension hijacking attacks where Google or Mozilla Developer Hub accounts were compromised. Existing extensions uploaded to the Chrome store were then replaced with malicious versions that enabled attackers to gain access to users accounts. These common attack vectors that browser extensions are susceptible to should be incorporated into the threat model, helping the design and building of a robust system with security in mind.

An important approach to the system design of browser-based wallets is having a well defined security architecture and protocol. The Stacks Wallet Extension's existing protocol design for authenticating and requesting the signing of transactions, found mostly in the `stacks.js/encryption` and `ux/connect` packages, appears to be lacking a methodological approach to design and implementation (Suggestion 11). For example, there is currently no trust in the key used to sign an authentication request, thus providing the verifier with no additional trust in the signed data and the extension will accept authentication requests without knowing who is in the possession of the key. Within the current protocol design, a remediation for this issue is not possible and can only be mitigated by less desirable means (Issue E).

Another example pointing to the benefit of a redesigned security architecture is that, in the current protocol, any extension installed on the user's browser has access to privileged APIs that allow for executing arbitrary JavaScript on a Hiro application page that bypasses the Same Origin Policy (SOP) and gives access to sensitive application-specific data stored in `localStorage`. A successful attack of this type may result in de-anonymizing a user, decrypting data stored on chain, and signing JSON Web Tokens (JWT) for use in other attacks (Issue E, Issue F). Given the absence of a secure enclave in Chrome, attacks of this type are difficult to mitigate against and depend on the end user's own security practices. We recommend advising users to utilize dedicated browsers without other extensions. If it is found to be possible to move the data stored in `localStorage` into the wallet sandbox, this could solve the exposure problem, but would create new problems of how to build trust relationships between application and wallet code (Issue B). Given that this mitigation comes with important and risky trade-offs and relies considerably on the end user, we strongly suggest revisiting the protocol design with keen focus on the security architecture. This exercise will ultimately determine a design approach that can eliminate the presence of such and similar vulnerabilities (Suggestion 11).

Furthermore, a methodological approach to the protocol design can then be used as the basis from which the protocol is implemented. In order to clearly understand the required functional and security properties of the building blocks used, the protocol should be engineered such that it fulfills both the desired functionality while adhering to the required security properties. This begins with stating the system's overall desired functional and security properties, as well as listing the security properties and assumptions provided by the protocol's individual components. While this is a significant undertaking, the effort can be aided by knowledgeable security teams through secure design consulting, followed by a security audit of the protocol once it has been redesigned and reimplemented (Suggestion 11).

System Components

Browser Security

We identified several specific areas within the system design that are particularly susceptible to browser security issues, as detailed below.

The Stacks Wallet Extension when loaded by a malicious page monitoring the `navigator.clipboard` object can lead to the exposure of a user's secret key phrase when they are logged into the wallet. Prior to being quickly patched by the Hiro team, this could have been triggered by a clickjacking attack where an attacker loads the wallet inside a hidden iframe. Since knowledge of the secret key phrase can be used to clone a wallet and seize its assets, we recommend preventing the key phrase from ever being accessible to the clipboard available to the browser by disabling the selection of text and force users to download a file. In addition, we suggest investigating if the clipboard object can be disabled for that page entirely (Issue A). We reported this issue to the Hiro team immediately during the audit and they promptly remediated the vulnerability by notifying their community and providing an update to address the vulnerability.

At present, the secret key phrase is stored in memory of the browser in cleartext. As a result, attackers that are able to dump the memory of the extension's process will be able to steal the secret key phrase and take control of a wallet and its assets. We recommended that the key phrase be encrypted when not in use, and that neither the encrypted key phrase nor encryption key be stored in a text format that will appear in the output of the `strings` command. We suggest using a binary object, which would not prevent the key from appearing in a binary search, but would create significant obstacles for an attacker to overcome (Issue D).

A malicious extension can listen for the `stacksTransactionRequest` Document Object Model (DOM) event and immediately trigger another event with a different JWT that will override the pop-up window with an attacker controlled transaction. As a result, an unsuspecting user may be tricked into confirming a transaction they did not intend, sending their assets to an attacker controlled address. We recommend appending a collision resistant hash to the window name used in the `window.open()` function, preventing the same pop-up window from being recycled by the attacker and notifying the user that a problem exists (Issue F).

Use of Cryptography

We identified several specific areas within the system design that are particularly susceptible to security issues resulting from the application and use of cryptography, as detailed below.

There are several instances in the `stacks.js/encryption` package where common cryptographic algorithms have been implemented by the Hiro team. Implementation of cryptographic systems can be prone to subtle, yet catastrophic errors. In order to avoid the potential for implementation errors leading to serious vulnerabilities, we recommend using established and trusted cryptography libraries like `libsodium.js`, which provides the same interface and security guarantees as the currently implemented algorithms. In addition, use of a cryptography library will help to safeguard secret keys and other confidential information, reduce the chances of introducing serious bugs, and improve the overall security of the system (Issue J).

The same key pair is currently used to both sign and encrypt data, which constitutes the reuse of key material between different algorithms and deviates from key management best practices. Key reuse has led to very serious security vulnerabilities in other protocols and adherence to best practices and standards is an important step towards building and maintaining a robust system. As a result, we recommend the use of distinct keys for signing and encryption, in addition to including the public key in the signed authentication request so the relation can be verified (Issue C).

The first segment of the BIP-32 derivation path used by the Stacks Wallet Extension indicates that a BIP-44 derivation path is in use. However, the rest of the path does not comply with the BIP-44 specification. This method of derivation leads to incompatibilities with other wallets. Furthermore, if a single account key is compromised, it becomes much more feasible to also compromise other account keys than with the derivation specified in the original BIP-44 path. As a result, we recommend adhering closely to the BIP-44 specification (Issue L).

In addition, the Stacks Wallet Extension uses the BIP-44 derivation tree for non-wallet keys (i.e. the key used to encrypt the configuration before uploading to Gaia Hub). This deviates from the intended use of keys in the BIP-44 derivation tree and may result in future attacks. As a result, we recommend using the `m/888'/` tree for non-wallet derivations in order to avoid any ambiguities about the purpose of the key and key reuse across primitives, which may lead to failing security guarantees (Issue M).

The Stacks Wallet Extension currently makes use of PBKDF2, which is a purely CPU-bound key derivation function. This class of algorithms has been advised against for several years due to negative security implications. In this instance, the Stacks Wallet Extension may accept messages with a correct signature, created using a key under the control of the attacker. We recommend making use of key derivation functions based on memory-hard functions such as `Argon2`, which is a more favorable and secure alternative (Issue N).

Data Validation and Code Security Issues

We identified several areas within the system design that are particularly susceptible to data validation and code-specific security issues, as detailed below.

The JWT signature is not validated when processing authentication and transaction requests, such that a malicious browser extension can take advantage of the wallet messaging system to invoke a transaction without knowledge of the `appPrivateKey`. If the user confirms the transaction, an incorrect JWT signature does not prevent the transaction from succeeding. We also identified that the security guarantees of the signature are tenuous and suggest re-evaluating the protocol (Suggestion 11; Issue E).

In addition, the `stxAddress` in the JWT for a transaction request is not required to match the sending address and an attacker would not need to have knowledge of the `stxAddress` of the victim prior to forging a JWT for use in another attack. We recommend ensuring that the `stxAddress` matches the address of the wallet when validating the JWT (Issue G).

An instance of improper handling of promises was identified. Currently, an error thrown by `provider.authenticationRequest` might not be caught by its encapsulating `try-catch` block, which is not the recommended approach to handle exceptions in JavaScript and TypeScript. The `catch` block cannot use `try-catch` statements thrown by `promises` because it might reach the `catch` block before the promise handling returns asynchronisly. An improper handling of an asynchronous code execution may lead to unintended exception handling and the wrong execution flow could result in the code not behaving as intended. We recommend utilizing the chain catch method of `provider.authenticationRequest` promise to handle errors (Issue H).

During the processing of an authentication request, the application domain is inferred from the redirect URL instead of using the `appDomain` field of the request, allowing inconsistent requests to be accepted. A request that uses inconsistent URIs should be considered invalid and rejected, in order to guarantee that no part of the code can be subject to application domain confusion attacks. A more cautious and precise approach would be to validate the consistency of the request and then use the application domain explicitly specified in the authentication request. As a result, we recommend verifying whether the URLs in the request are consistent and use the `appDomain` value as the application domain (Issue I).

Dependencies

A considerable number of security advisories were published for several of the dependencies used by the Stacks Wallet Extension while running npm audit. In addition, the `valid-url` dependency has not been maintained for eight years and has several open issues on GitHub. Use of unmaintained dependencies and dependencies with known issues could lead to the introduction of vulnerabilities and bugs into the Stacks Wallet Extension code base. For example, the dependency could allow invalid URLs to pass validation resulting in cross site scripting attacks (XSS).

The use of automated tools discovered a large number of vulnerabilities. For example, in the `blockstack/ux` repository, 4,692 vulnerabilities were identified. While most vulnerabilities found may be minor bugs, we were not able to review each due to the volume and time required to investigate each individual vulnerability. We recommend using regularly audited and maintained dependencies in addition to regularly maintaining dependencies to ensure they are current and that known bugs and issues have been addressed. We advise the Hiro team to take note of published advisories and update dependencies when fixes are released. Furthermore, as a future-proof measure, we recommend including automated dependency auditing into the Continuous Integration (CI) workflow or enabling Dependabot on GitHub, which automatically notifies developers about published security advisories relevant to the code base (Issue K).

We also recommend pinning dependencies to exact versions to retain more control over when and where upgrades take place. This will prevent unwanted versions from being inadvertently installed, thus increasing the attack surface. In addition, this will help both developers and reviewers to identify new vulnerabilities discovered in dependencies, which is paramount to the security of the codebase (Suggestion 1).

Code Quality

The code is organized into modules, allowing easy readability of the code in both the `blockstack/ux` and `blockstack/stacks.js` repositories. Both repositories make use of code formatters and ESLint, a linter for conducting static code analysis to help check for common programming errors and provide suggestions of areas of code that should be reviewed for security and stability. The use of linters further aids in early detections of common code smells, bugs, and suspicious patterns.

In particular, the `stacks.js` packages in-scope reflected good overall code quality and maturity. The code base is segmented to well-defined and purpose specific modules making it maintainable and extensible. The code contains comments (see code comments), is correctly formatted, and is clear with low cyclomatic complexity (i.e. functions are kept simple and self-contained). Unit tests and linters included in the are included in the Continuous Integration (CI) workflow.

The `blockstack/ux` repository utilizes safe API functions and input sanitization to help protect against and prevent XSS attacks. Input sanitization is done by using libraries such as `DOMPurify`, to sanitize danegerous HTML, and `dangerouslySetInnerHtml`, a React function used as a replacement for the XSS attacks prone `innerHTML` function. The use of these libraries demonstrates considerations for security.

Tests

`blocksack/ux` contains sufficient coverage of integration tests but contains very few unit tests. Inclusion of unit tests in the CI and continuous deployment workflows is considered best practice and aids in automating the process of verification. Higher test coverage increases trust in the system and enables early detection of bugs and errors. As a result, we recommend the Hiro team increase unit test coverage (Suggestion 10).

`blockstack/stack.js` reflects a considerable amount of unit test coverage across the `auth`, `encryption`, and `wallet-sdk` packages. We suggest maintaining both high branch and line coverage ratios for security critical packages. High branch coverage allows more possible execution flows to be covered in the system while sufficient line coverage provides a better indication of how much code is being tested. We recommend improving both line and branch coverage ratios and continuously maintaining a high unit test coverage, as it contributes to better trust of future development (Suggestion 10).

Documentation

The existing documentation for the Stacks Wallet Extension, including the project documentation and the audit specific documentation provided to our team, was helpful in that it successfully explains how to run and use the system at a high level. However, the existing documentation would significantly benefit from a specification and description of the protocol. In the absence of this documentation, the analysis was largely performed based on understanding derived from the code. We recommend further expanding the documentation to provide detailed specification and description of the protocol (Suggestion 3).

In order to understand and evaluate the Hierarchical Deterministic (HD) wallet derivation tree, it is important to first understand the mechanisms and assumptions of the system. Additional documentation would aid considerably in understanding which derivation is used in what cases, providing specific details on why it is used in some instances and why it is not used in others. We recommend clearly and publicly documenting the HD wallet derivation tree in order to allow both users and reviews to understand the system and reason about and identify potential vulnerabilities (Suggestion 9).

Code Comments

The `auth`, `encryption` and `wallet-sdk` packages reviewed in `blockstack/stacks.js` include sufficient code comment coverage. However, the packages contained within `blockstack/ux` have a significantly lower amount of code comments. While the packages contained within `blockstacks/stacks.js` are considered to be more security critical, the `blockstack/ux` would benefit from additional code comments. We recommend that code comments coverage is comprehensive and consistent throughout all packages of the code base (Suggestion 2).

Scope

The scope of the Stacks Wallet Extension was sufficient in that it contained all security-critical components of the system design and implementation. This included the `blockstack/ux` monorepo, which contains the core functionality and packages for running the wallet application and browser extension. In addition, our scope also included a review of the `wallet-sdk`, `auth`, and `encryption` internal dependencies in `blockstack/stacks.js`. Review of these dependencies was particularly important since they handled keychain and seed phrase management and encompassed the authentication protocol handling encryption, decryption, and session management, in addition to other cryptographic operations that are used in the authentication flow.

Penetration Testing

During penetration testing, two key scenarios were investigated within the scope of this audit: a malicious Stacks application or web page and a malicious browser extension. In both scenarios, we investigated the messaging systems to discover the attack surface and proceeded in attempting to extract secret information from the wallet or initiate malicious transactions.

From a penetration testing perspective, the project posed challenges due to the absence of tooling normally available for testing web technologies. Since the messaging between applications and the wallet does not leave the browser, usage of network based proxies was not possible to intercept requests. These challenges were overcome by examination of the source code and documentation, as well as the helpfulness of the Hiro team in answering questions. Attack vectors were then devised and implemented using custom code built on top of demo applications provided by Hiro for developers.

Specific Issues & Suggestions

We list the issues and suggestions found during the review, in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: Secret Key Exposure	Resolved
Issue B: Malicious Browser Extensions Can Steal Sensitive Data	Unresolved
Issue C: Same Key Used for Signing and Encryption	Unresolved
Issue D: Secret Keyphrase Stored in Cleartext in Memory	Partially Resolved
Issue E: Incorrect Use of JWTs	Partially Resolved
Issue F: Transactions are Over-Writeable by Malicious Extensions	Unresolved
Issue G: stxAddress in JWTs is Not Verified	Partially Resolved
Issue H: Use of Improper Promise Handling	Resolved
Issue I: Authenticator Ignores Parts of Request	Resolved
Issue J: Custom Implementations of Common Cryptographic Algorithms	Unresolved
Issue K: Management and Maintenance of Dependencies	Resolved
Issue L: Non-Compliant use of HD Derivation BIP-44 Paths	Unresolved
Issue M: HD Derivation Uses BIP-44 Paths for Non-Wallet Keys	Unresolved
Issue N: Use a More Secure Password-Based Key Derivation Mechanism	Resolved

Suggestion 1: Pin Dependencies to Specific Versions	Resolved
Suggestion 2: Improve Code Comments	Resolved
Suggestion 3: Expand Documentation	Resolved
Suggestion 4: Improve Password Strength Parameters	Unresolved
Suggestion 5: VBookmarkerify Strings Before Rendering	Unresolved
Suggestion 6: Use Conforming DIDs	Unresolved
Suggestion 7: Use FNV-1a Instead of hashCode	Unresolved
Suggestion 8: Error Message in getBufferFromBN Looks Incorrect	Unresolved
Suggestion 9: Document the HD Wallet Derivation Tree	Unresolved
Suggestion 10: Increase Test Coverage	Partially Resolved
Suggestion 11: Revisit Security Architecture Design	Unresolved

Issue A: Secret Key Exposure

Location

`chrome-extension://ffmccdpbokklglpamkcddkcaghgbpgni/index.html#/settings/secret-key`

Synopsis

The Stacks Wallet Extension is vulnerable to a clickjacking attack. A malicious web page can load the extension in a hidden iframe while monitoring the `navigator.clipboard` object, leading to the exposure of a user's secret key phrase when they are logged into the wallet.

Impact

An attacker can use the secret key phrase to initialize a different extension and take control of all the assets in the wallet.

Preconditions

The victim must be logged into the wallet while visiting a malicious page. If the victim clicks on a malicious item in the page, the *Copy to Clipboard* button will be triggered, and the secret key phrase will be available to the attacker in the `navigator.clipboard` object.

Feasibility

The attack is easy to create, although the user would need to visit the malicious site and perform a single action. Given that an attacker is highly incentivized, they would be motivated to reach as many potential victims as possible.

Technical Details

The *Copy to Clipboard* button will store the secret key phrase in plaintext in the `navigator.clipboard` object. A malicious site can run a function that checks the contents of this object periodically, and if it recognizes a valid key phrase, it can then exfiltrate the key phrase to a remote location without the victim being aware.

The clickjacking attack facilitates getting the key phrase into the clipboard. As a result, while it can be considered a stage in the overall attack, mitigating clickjacking attacks will not solve the problem of the key phrase being stored in the `navigator.clipboard` object.

Mitigation

Protection against clickjacking attacks is traditionally implemented by utilizing the X-Frame-Options HTTP Header. Since the extension pages are not loaded from an HTTP server, research by the Hiro team can be conducted to assess how this could be implemented in the context of a browser extension. Utilizing Content Security Policies (CSP) that prevent framing resources would be a possible solution.

To protect against the clipboard attack, it would be best to prevent the key phrase from ever being accessible to the clipboard available to the browser. One solution is to disable selection of the text and force users to download a file. Another option would be to investigate if the clipboard object can be disabled for that page entirely. This latter option might prove unfeasible, as users will want to be able to copy and paste addresses.

Status

This issue was reported to and discussed with the Hiro team during the audit. The Hiro team released a patch adding the frame-ancestors CSP direction, thus resolving this issue. Our team verified the changes and the clickjacking portion of the exploit is no longer possible.

Verification

Resolved.

Issue B: Malicious Browser Extensions Can Steal Sensitive Data

Synopsis

Any extension installed on the user's browser has access to privileged APIs that allow for the execution of arbitrary Javascript on a Blockstack application page that bypasses the SOP and gives access to sensitive application-specific data stored in local storage. This includes the `transitKey`, `appPrivateKey`, `decentralizedID`, and wallet address. The `appPrivateKey` is used for signing transactions as well as encrypting user application data on chain.

Impact

A successful attack of this type can result in de-anonymizing a user, decrypting data stored on chain, and forging JWT for use in other attacks (Issue E; Issue F).

Preconditions

Another browser extension is installed in the same browser where the Stack Wallet Extension is used. This can occur because either a user has installed a malicious extension or in the event that the browser was previously compromised and an extension was installed without their knowledge.

Feasibility

The attack is trivial to perform because any browser extension can be compromised in this way. Moreover, the potential for significant reward motivates potential attackers to allocate resources to an attack of this type, or create browser extensions specifically targeting typical users of the Stack Wallet Extension.

Technical Details

Chrome extensions have access to several privileged APIs that are not restricted by the SOP that otherwise sandboxes the domain. The `chrome.tabs.executeScript()` API can be leveraged from an extension to read local storage. With knowledge of a tab's ID gathered using the `chrome.tabs.query()` API, an extension can execute the following to steal the `appPrivateKey` for a particular TAB_ID:

```
chrome.tabs.executeScript(TAB_ID, {code: "(function() {return
JSON.parse(localStorage.getItem('blockstack-session')).userData.appPrivateKey
})()"},

        (resp) => {

          console.log(resp)})
```

Mitigation

Given the absence of a secure enclave in Chrome, attacks of this type are difficult to mitigate against. Solving this problem may prove exceedingly difficult in the current environment. However, exploitation requires a previous compromise of the browser thus largely depends on the end-users' own security practices.

A mechanism to prevent this attack in the short-term would be to advise users to use dedicated browsers without extensions. If it is found to be possible to move the data stored in `localstorage` into the wallet sandbox, this could solve the exposure problem, but would then create new problems of how to build trust relationships between application and wallet code. We recommend this short-term mitigation and suggest that a longer term remediation be pursued in conjunction.

Remediation

As there is no clear remediation path, we recommend the Hiro team revisit their security architecture design (Suggestion 11) so it is a secure framework for constructing the application.

Status

The Hiro team has responded and acknowledged that they are aware of this issue. Given that remediating this issue would require considerable changes to the protocol design, they will continue to consider a long term strategy by revisiting their security architecture design and have stated their intent to notify users of best practices (i.e. using a dedicated browser) when using Stacks applications. At the time of this verification, the suggested mitigation and remediation remain unresolved.

Verification

Unresolved.

Issue C: Same Key Used for Signing and Encryption

Location

`auth/src/messages.ts#L204`

`auth/src/messages.ts#L108`

Synopsis

The transit key pair is used to sign and encrypt data. A more secure practice is to use a separate encryption key pair and to sign the corresponding public key using the signing key. Similarly, the `appPrivateKey` is used to both sign transaction request JWTs and encrypt application specific data on-chain.

Impact

In the event that encryption using the transit key is broken, the `appPrivateKey` would be leaked, severely undermining the security of the protocol.

In the event that signing using the appPrivateKey is broken, an attacker would be able to issue transaction requests.

In the event that encryption using the appPrivateKey is broken, an attacker would be able to decrypt encrypted on-chain application data.

Preconditions

The attacker would have to break the joint security of ECDSA and ECIES.

Feasibility

While [DLP+11] shows that the ECDSA and ECIES are jointly secure, the assumptions the security analysis is based upon are very strong. Therefore, the results of that paper do not provide the confidence that is expected for this class of algorithm.

Technical Details

The discrete logarithm problem is the difficult problem underlying several cryptographic schemes, including most elliptic curve cryptography. This presents the possibility to reuse key material between different algorithms. However, it is not always clear whether this is secure practice. Additionally, different key uses often require different key lifecycles, which is not possible if the same key material is used. The fact that this need has not been identified as of yet does not mean that this is not an issue. Often, such requirements only come up as applications mature.

While there is a security proof available for the joint security of ECIES and ECDSA [DLP+11], the proof makes use of the random oracle model and the generic group model, both very strong (and therefore undesirable) assumptions.

Remediation

We recommend using distinct keys for signing and encryption, in addition to including the public key in the signed authentication request so the relation can be verified.

Status

The Hiro team has acknowledged this issue and are considering changes to the protocol in order to adhere to key management best practices. At the time of this verification, the suggested remediation remains unresolved.

Verification

Unresolved.

Issue D: Secret Keyphrase Stored in Cleartext in Memory

Location

Browser memory

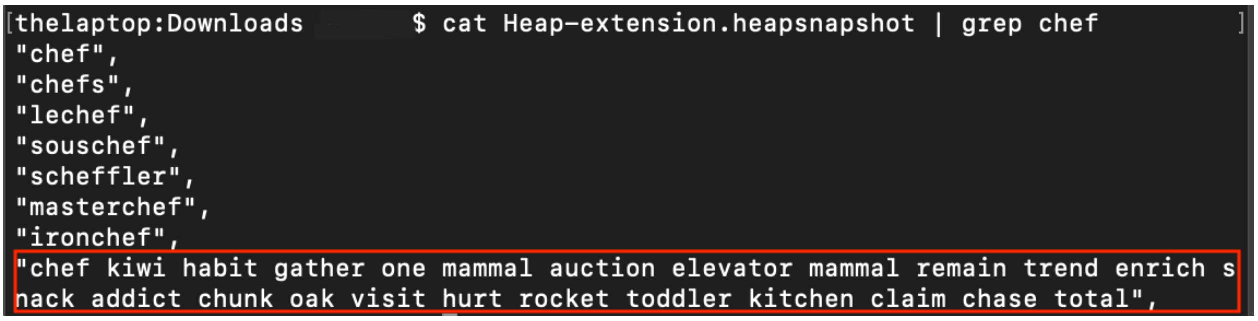
Synopsis
The secret key phrase is stored in the memory of the browser in cleartext. Attackers that are able to dump the memory of the extension's process will be able to steal the secret key phrase and take control of a wallet.

Impact
With knowledge of the secret key phrase, an attacker can instantiate a clone of the wallet and gain control of all of its assets.

Preconditions
An attacker will need to be able to dump the memory from the extension's process. This would require either physical access to the browser or a post-exploitation condition on the victim's machine.

Feasibility
The attack is trivial to perform if the preconditions listed above are satisfied. Given the incentive for attackers, common malwares could incorporate a check to dump memory from browsers and search for strings in the format of a secret key phrase for exfiltration.

Technical Details
With the extension open in the browser, open *Chrome Developer Tools*, select *Memory*, click *Take Snapshot*, and then *Save*. With the downloaded file, run the `strings` command, and for quick results grep for a word in the secret key phrase to view in clear text.



```
[thelaptop:Downloads        $ cat Heap-extension.heapsnapshot | grep chef        ]
 "chef",
 "chefs",
 "lechef",
 "souschef",
 "scheffler",
 "masterchef",
 "ironchef",
 "chef kiwi habit gather one mammal auction elevator mammal remain trend enrich s
nack addict chunk oak visit hurt rocket toddler kitchen claim chase total",
```

Mitigation
While removing the key phrase from memory entirely is not possible, it is recommended that the key phrase be encrypted when not immediately in use, and that neither the encrypted key phrase nor encryption key be stored in a text format that will appear in the output of the strings command. We recommend using a binary object instead. While this would not prevent the key from appearing in a binary search, it would create significant obstacles for an attacker to overcome.

Status
The Hiro team's current solution takes steps towards preventing the mnemonic from being easily identified in memory, but does not address the issue of the mnemonic being retrievable by running the strings command on a Heap Snapshot. By encoding as ASCII Hex, the mnemonic will now be identifiable as being a hex string beginning with 0x, and containing 12 or 24 space characters, identified as the byte 20 in Hex. Since ASCII hex is easily converted into plaintext, this solution transfers the problem into a different encoding.

In order to implement a more robust mitigation, we recommend the following steps: In the `stringToHex` function, a binary buffer is created and then turned into a hex string. If the `toString('hex')` method is omitted, the wallet would store the mnemonic in a binary buffer, which would have a different representation on the Heap than a string, and therefore unlikely to appear in the output of the `strings` command.

Verification
Partially Resolved.

Issue E: Incorrect Use of JWTs

Location
`chrome-extension://ffmccdpbokklglpamkcddkcaghgbpgni/index.html#/transaction?request=`

`common/hooks/use-wallet.ts#L133`

`encryption/src/wallet.ts#L39-L46`

`components/transactions/stx-transfer-details.tsx#L7`

Synopsis
The keys backing the tokens do not have any trust associated with them. In the instance of `transitKey`, this is because it was only just generated (and could have potentially been generated by anyone). In the case of `applicationSecretKey`, this is because other extensions can read it from `localStorage` (see Issue B).

Additionally, the JWT signature is not validated when processing authentication transaction requests.

Impact
A malicious browser extension can take advantage of the wallet messaging system to invoke a transaction without knowledge of the `appPrivateKey`. If the user confirms the transaction, an incorrect JWT signature does not prevent the transaction from succeeding. This would allow an attacker to perform an attack against Issue F even in the absence of the vulnerability documented in Issue B.

Additionally, a malicious browser extension can inject itself into a Blockstack application, hijack the authentication UI, receive the `appPrivateKey` from the wallet extension, decrypt it, authenticate it to the wallet and exfiltrate the key.

Feasibility
Medium. An attacker does not need to provide a valid signature for the JWT, and can use the signature of an arbitrary request generated either in code or online. Malicious browser extensions will have access to the signing key, and can create a valid signature by using the `jsontokens` npm package.

Breaking sign-in using malicious authentication requests is not immediately possible due to messaging restrictions implemented using browser security mechanisms. These browser security mechanisms only protect against attacks coming from web pages. If the browser is in a post-compromise state, a malicious extension can easily bypass these restrictions.

Technical Details
The `index.html#/transaction?request=<JWT>` route in the extension takes a JWT in the `request` parameter. The message of the JWT is loaded into the extension popup as trusted data. While the presence of the JWT signature is required, it was found that the message body could be changed with arbitrary values without altering the signature with no impact on the success of the transaction.

Similarly, the code handling authentication requests does not verify the validity of the tokens.

Mitigation

Given the absence of a secure enclave in Chrome, attacks of this type are difficult to mitigate against. Solving this problem may prove exceedingly difficult in the current environment. However, exploitation requires a previous compromise of the browser thus largely depends on the end-users' own security practices.

A mechanism to prevent this attack in the short-term would be to advise users to use dedicated browsers without extensions. If it is found to be possible to move the data stored in `localstorage` into the wallet sandbox, this could solve the exposure problem, but would then create new problems of how to build trust relationships between application and wallet code. We recommend this short-term mitigation and suggest that a longer term remediation be pursued in conjunction.

Remediation

As there is no clear remediation path, we recommend the Hiro team revisit their security architecture design (Suggestion 11) so it is a secure framework for constructing the application.

Status

JWTs are now signed and internal components are checked for accuracy, with one exception related to the `stxAddress`. Validating signatures for transaction requests means that only parties with access to the `appPrivateKey` can create valid transaction requests. This issue is considered partially resolved as the exposure of the `appPrivateKey` to other extensions means that the parties with access to the key are still potentially unknown.

Verification

Partially Resolved.

Issue F: Transactions are Over-Writeable by Malicious Extensions

Location

`chrome-extension://ffmccdpbokklglpamkcddkcaghgbpgni/index.html#/transaction?request=`

Synopsis

A malicious extension can listen for the `stacksTransactionRequest` DOM event and immediately trigger another event with a different JWT that will override the pop-up window with an attacker controlled transaction. This occurs because the `window.open()` method will return a previously opened window if the window name is the same on subsequent calls. As the name of the extension's pop-up is hardcoded, the same popup window will always be overwritten with the latest transaction request.

Impact

An unsuspecting user may be tricked into confirming a transaction they did not intend, sending their coins to an attacker controlled address. The use of JWTs offer no protection against this attack as an attacker can create a valid JWT with the `appPrivateKey` (Issue B; Issue E).

Preconditions

The user's browser would need to have been compromised with a malicious browser extension capable of injecting Javascript code into the Stacks Wallet Extension application's tab.

Feasibility

Medium. A user's browser must be in a post-compromise state with a malicious extension listening for DOM events and a user must not double check the transaction before confirming. Despite these mitigating circumstances, the incentive of a successful attack will motivate attackers to compromise browsers en masse.

Technical Details

A malicious extension can execute the following code to overwrite the popup window for an arbitrary JWT on the tab with identifier `TAB_ID`.

```
     let req = SOME_JWT;

            let payload = 'function newReq() {'+

                'evt = new CustomEvent("stacksTransactionRequest",
{detail:{transactionRequest: "'+req+'" }});'+

                'document.dispatchEvent(evt);'+

            '}'+

            'document.addEventListener("stacksTransactionRequest", (evt) =>
newReq());'

            chrome.tabs.executeScript(TAB_ID, {code: payload});
```

Remediation

We recommend appending a collision resistant hash to the window name used in the `window.open()` function. This will prevent the same pop-up window from being recycled by the attacker and will notify the user that a problem exists.

Status

The Hiro team has responded that they have further investigated this issue and found that a malicious extension does not need to overwrite any existing transaction request, but can also create one. The Hiro team has also stated that remediating this issue requires core protocol design changes, which they are currently evaluating. At the time of this verification, the suggested remediation remains unresolved.

Verification

Unresolved.

Issue G: `stxAddress` in JWTs is Not Verified

Location

`chrome-extension://ffmccdpbokklglpamkcddkcaghgbpgni/index.html#/transaction?request=`

Synopsis

The `stxAddress` in a transaction request is not checked to match the sending address.

Impact

An attacker would need no knowledge of the `stxAddress` of the victim prior to creating a request for use in another attack. It should be noted that the address of the user is available in `localStorage` per Issue B.

Remediation

When processing the transaction request, we recommend ensuring that the `stxAddress` matches the address of the wallet.

Status

The `stxAddress` is now checked in the `decodeToken` function. However, if the `stxAddress` is omitted, no check is performed. It is unclear whether a transaction would possibly not have a sending address and what the use case for this situation is. Bypassing the check by omitting the `stxAddress` would make an attacker's job easier. We recommend a more secure solution of implementing a strict check that requires the `stxAddress` to be present and correct.

Verification

Partially Resolved.

Issue H: Use of Improper Promise Handling

Location

`app/src/auth.ts#L77`

Synopsis

An error thrown by `provider.authenticationRequest` might not be caught by its encapsulating `try-catch` block. This is not the recommended approach to handle exceptions in JavaScript and TypeScript, as detailed in MDN Web Docs.

Impact

An improper handling of an asynchronous code execution may lead to unintended exception handling, as the wrong execution flow could result in the code not behaving as intended. For example, returning wrong errors could prevent users from quickly identifying bugs.

Remediation

We recommend using the `promise.then` and `promise.catch` methods, which is the asynchronous equivalent of a `try-catch` block when calling `provider.authenticationRequest` to handle errors.

Status

`provider.authenticationRequest` now properly handles exceptions that may be thrown as per the recommendation of MDN Web Docs.

Verification

Resolved.

Issue I: Authenticator Ignores Parts of Request

Location

`common/hooks/use-wallet.ts#L120`

Synopsis

The application domain is inferred from the redirect URL instead of using the `appDomain` field of the request, allowing inconsistent requests to be accepted. A more cautious and precise approach would be to validate the consistency of the request, and then use the application domain explicitly specified in the authentication request.

Impact

While we did not identify a way to exploit this issue, a request that uses inconsistent URLs should be considered invalid and rejected, in order to guarantee that no part of the code can be subject to application domain confusion attacks.

Technical Details

The significance of the application domain in the Stacks Wallet Extension is comparable to that of the origin in web applications. Confusing the access control systems of the Stacks Wallet Extension about this value may result in attacks where a malicious application poses as one application to parts of the extension. As a result, it is important to verify for consistency with the redirect URL and the manifest URL.

Remediation

We recommend verifying whether the URLs in the request are consistent and use the `appDomain` value as the application domain.

Status

The Hiro team has removed the additional `appDomain` function parameter and all metadata is now derived from the authentication request. This is an effective measure to prevent inconsistencies and the possibility of confusion attacks.

Verification

Resolved.

Issue J: Custom Implementations of Common Cryptographic Algorithms

Location

`encryption/src/ec.ts#L271`

`encryption/src/ec.ts#L94`

`encryption/src/ec.ts#L336`

`encryption/src/wallet.ts#L52`

Synopsis

Elliptic Curve Integrated Encryption Scheme (ECIES) and Authenticated Encryption (AE) are common cryptographic primitives. As a result, the custom implementation of common cryptographic algorithms is necessary and may result in serious errors.

Impact

Secret keys as well as other confidential information could be leaked due to errors in the implementation, severely undermining the security of the Stacks Wallet Extension.

Preconditions

A critical flaw in the implementation needs to exist and be identified by a malicious actor.

Feasibility

The existing code is relatively straightforward and no mistakes were identified in this audit. However, future changes may introduce new, unknown vulnerabilities.

Technical Details

Implementing cryptographic systems is prone to subtle, yet catastrophic mistakes and can be avoided by making use of well known and more secure alternatives. In this instance, use of an existing cryptography library would limit the attack surface resulting from custom implementations of common cryptographic algorithms.

Since encryption and signing keys should be separated (see Issue C), the algorithm used to choose the encryption key can be chosen more freely.

Remediation

We recommend using the wasm-compiled `libsodium.js`. For ECIES, we recommend using libsodium's `crypto_box_seal` and `crypto_box_seal_open` functions. These provide the security of ECIES as well as a stable and well-maintained code base. For AE, we recommend using sodium's `secretbox` functions.

Status

The Hiro team has responded they are evaluating how to safely update their libraries to use `libsodium.js`. At the time of this verification, the suggested remediation remains unresolved.

Verification

Unresolved.

Issue K: Management and Maintenance of Dependencies

Location

`packages/app/package.json`

`packages/app/package.json#L79`

Synopsis

Running npm audit revealed that security advisories were published for several dependencies and the use of automated tools found numerous vulnerabilities in the existing dependencies. In addition, the `valid-url` dependency has not been maintained for eight years and has several open issues on GitHub.

Impact

The large number of reported vulnerabilities were an obstacle in verifying the impact as our team was unable to review each vulnerability due to time constraints. Using unmaintained dependencies may lead to critical vulnerabilities in the code base. For example, the unmaintained `valid-url` dependency may allow invalid URLS to pass validation resulting in XSS attacks.

Remediation

For proper management and maintenance of dependencies, we recommend the following:

- Manually audit and upgrade dependencies, in order to avoid unmaintained dependencies known issues. This would require extensive testing to ensure there are no backward compatibility issues introduced by upgrading dependencies.
- Include the automated dependency auditing into the CI workflow or enable Dependabot on GitHub, which automatically notifies developers about published security advisories relevant to the code base.
- Act on published advisories and update dependencies when fixes are released.

Status

All dependencies used by the Stacks Wallet Extension have been pinned to versions with no known vulnerabilities in security advisories. Furthermore, the Hiro team has added a Github action which runs `Yarn Audit` to automatically notify developers of issues discovered in existing or newly introduced dependencies.

Verification

Resolved.

Issue L: Non-Compliant use of HD Derivation BIP-44 Paths

Location

`wallet-sdk/src/derive.ts#L81`

Synopsis

The first segment of the BIP-32 derivation path used by the Stacks Wallet Extension (`44'`) indicates that this is a BIP-44 derivation. However, the rest of the derivation path does not match the BIP-44 specification. In addition to not complying to the specification, this version is less secure since account keys are not derived using hardened derivation.

Impact

This method of derivation leads to incompatibilities with other wallets. In the event that one account key is compromised, it becomes significantly more feasible to also compromise other account keys than with the derivation in the original BIP-44 specification.

Preconditions

An attacker needs a single account secret key and the public key and chain code of the key with derivation path `m/44'/5757'/0'/0` in order to learn the secrets of all accounts.

Feasibility

Medium.

Technical Details

The Stacks Wallet Extension uses the BIP-32 derivation path `m/44'/5757'/0'/1/$i` for the account with index $i. The first segment of the path (`44'`) indicates that this is a BIP-44 derivation. However, in BIP-44, the third segment is used to specify which account is used. In addition, hardened derivation is used on this level and, as a result, the derivation path should be `m/44'/5757'/$i'/0` (or `m/44'/5757'/$i'/1` for change addresses).

Mitigation

We recommend making the account key derivation hardened. While this results in being less compliant, since this level is not hardened in BIP-44, it prevents the issue of leakage of parent keys from unhardened subkeys.

Additionally, we recommend the Hiro team stop using 44' in the first segment and specify their own standard.

Remediation

We recommend adhering to the BIP-44 specification and use the third segment for accounts, using hardened derivation.

Status

The Hiro team has responded they are investigating changes to the key derivation logic while maintaining backward compatibility. At the time of this verification, the suggested mitigation and remediation remain unresolved.

Verification

Unresolved.

Issue M: HD Derivation Uses BIP-44 Paths for Non-Wallet Keys

Location

`wallet-sdk/src/derive.ts#L9`

Synopsis

The Stacks Wallet Extension uses the BIP-44 derivation tree for non-wallet keys, specifically for the key used to encrypt the configuration before uploading to Gaia Hub. This does not comply with the intended use of keys in the BIP-44 derivation tree, and may lead to future attacks.

Impact

Possible interaction between the encryption of the configuration and the derivation of address keys by wallets adhering to BIP-44, constituting key reuse and possibly leading to the loss of security guarantees.

Preconditions

A BIP-44 compliant wallet for STX must be used alongside configuration encryption for Gaia and an attack on the joint security of ECDSA and ECIES would need to be found (similar to Issue C). The former largely depends on the behavior and security practices of the user. The latter requires significant cryptographic knowledge and resources by the attacker.

Feasibility

Using a separate BIP-44 compliant wallet next to the Stacks Wallet Extension is likely not very common. See Issue C in regards to the joint security of ECDSA and ECIES.

Technical Details

The BIP-44 derivation tree is intended only for wallet keys. However, the Stacks Wallet Extension also derives keys used to encrypt the configuration before uploading to Gaia Hub. As noted, this does not comply with the intended use of keys in the BIP-44 derivation tree. Disagreement and ambiguity about the purpose of the key leads to key reuse across primitives, and key reuse across primitives may lead to failing security guarantees.

Remediation

We recommend using the `m/888'/` tree for non-wallet derivations. This tree appears to be only used by the Stacks Wallet Extension and, as a result, this is a secure approach given that internal conflicts and inconsistencies are ruled out.

Status

The Hiro team has responded they are investigating improved methods to generate non-wallet keys while maintaining backward compatibility. At the time of this verification, the suggested remediation remains unresolved.

Verification

Unresolved.

Issue N: Use a More Secure Password-Based Key Derivation Mechanism

Location

`common/hooks/use-wallet.ts#L133`

Synopsis

The Stacks Wallet Extension currently makes use of PBKDF2, which is a purely CPU-bound key derivation function. This class of algorithms has been advised against for several years due to negative security implications.

Impact

In this instance, the Stacks Wallet Extension may accept messages with a correct signature, created using a key under the control of the attacker.

Preconditions

The attacker needs to be able to circumvent other security restrictions on cross-tab messaging by the browser.

Feasibility

Attacks on browsers are not uncommon, but can usually be quickly patched. The feasibility mostly depends on the users update regimen and information from the development team.

Technical Details

It is feasible to significantly speed up dictionary attacks on CPU bound hashes (e.g., using FPGAs) because the task is easily parallelizable. As a result, using CPU-bound key derivation for passwords has been advised against in favor of memory-hard functions like `Argon2`. These are more difficult to parallelize, because RAM and fast access to it is expensive to build in hardware.

Remediation

`Argon2` comes in multiple variants optimized for different attack models, but the balanced Argon2id variant would be well suited in this instance. Section 4 of the `Argon2` RFC provides a procedure for choosing good parameters for specific use cases. When deciding on the maximum allowed time the derivation is allowed to take place, keeping in mind that it is slower in the browser, there is still a speedup for an attacker who tries to brute-force it using native code.

We recommend making use of key derivation functions based on memory-hard functions such as `Argon2`, which is a more favorable and secure alternative.

Status

The Hiro team has updated the extension to use the `Argon2id` memory hard hash function in place of PBKDF2. The parameters chosen are reasonable and the migration of existing wallets to use keys derived from `Argon2` takes place automatically during the first unlocking of the wallet following the update.

Verification

Resolved.

Suggestions

Suggestion 1: Pin Dependencies to Specific Versions

Location

`packages/app/package.json`

Synopsis

Many dependencies are not pinned to a specific version, instead they are pinned to a closed range of releases (e.g. `package.json` is set to accept major and minor changes). Pinning dependencies to exact versions is a sound approach to retaining more control over when and where upgrades take place. This will prevent unwanted versions from being inadvertently installed, thus minimizing the attack surface. In addition, this will help both developers and reviewers to identify new vulnerabilities discovered in dependencies, which is paramount to the security of the codebase.

Upgrading pinned versions of dependencies in accordance with an internal review to assess whether an upgraded version introduces compatibility issues adheres to best practices. In doing so, the necessary changes can be made to the Stacks Wallet Extension code base to mitigate against potential and existing issues. In addition, it's critical to check if the upgraded version of the dependency has reported vulnerabilities. This will allow for informed decisions to use a more secure alternative.

Finally, `recoil` version is set to a nightly build, which could easily result in bugs or breaking changes when a user downloads the Stacks Wallet Extension. Instead, the version of `recoil` can be pinned to a stable release.

Mitigation

We recommend pinning dependencies to specific versions, including pinning build-level dependencies in the `package.json` file to a specific version and only upgrading dependencies upon careful internal review for potential backward compatibility issues and vulnerabilities.

Status

All dependencies used by the Stacks Wallet Extension have been pinned to versions with no known vulnerabilities or security advisories. Furthermore, the Hiro team has added a Github action that automatically checks and prevents usages of ranges in `package.json`.

Verification

Resolved.

Suggestion 2: Improve Code Comments

Location

[`blockstack/ux`]

Synopsis

While the packages contained within `blockstack/stacks.js` have sufficient code comments, the packages contained within `blockstack/ux` have a significantly lower ratio of code comments. In particular, the `blockstack/ux` packages have many UI and React components code which would benefit from additional code comments.

Mitigation

We recommend that code comments coverage is comprehensive and consistent throughout all packages of the code base, as they highlight key information and contribute to easier readability and understanding of the code for both users and reviewers.

Status

The Hiro team has added code comments better explaining many of the complicated functions in the code base. In addition, the Hiro team has stated that they intend to adopt a better approach to add comments to newly introduced code.

Verification

Resolved.

Suggestion 3: Expand Documentation

Location

`https://docs.blockstack.org/`

Synopsis

The existing documentation for the Stacks Wallet Extension, including the project documentation and the audit specific documentation provided to our team, was helpful in that it successfully explains how to run and use the system at a high level. However, the existing documentation would significantly benefit from a specification and description of the protocol. In the absence of this documentation, the analysis was largely performed based on understanding derived from the code.

Mitigation

We recommend further expanding the documentation to provide detailed specification and description of the protocol. In doing so, we suggest that the documentation encompass the intricacies of system behavior and the reasoning behind the design for individual components.

Status

The Hiro team has improved the existing documentation and stated that they intend to continue to improve the project documentation.

Verification

Resolved.

Suggestion 4: Improve Password Strength Parameters

Location

`dropbox/zxcvbn`

Synopsis

The Stacks Wallet Extension utilizes `zxcvbn` as a password strength estimator, which uses parameters based on limited and outdated data. While this covers a wide range of password composition, it does not account for a worldwide audience and common compromised passwords. We recommend using a password strength estimator that accounts for a larger range of inputs or using a manually compiled version of `zxcvbn` that utilizes more comprehensive data dictionaries.

Mitigation

We recommend the use of a more expansive password and names data dictionaries for `zxcvbn` estimations, thus enforcing the use of better passwords.

Status

The Hiro team has responded they are considering solutions that will not impact the user experience while maintaining security. As a result, at the time of this verification the suggested mitigation remains unresolved.

Verification

Unresolved.

Suggestion 5: Verify Strings Before Rendering

Location

`components/transactions/stx-transfer-details.tsx#L7`

Synopsis

Elements of the JWT are expected to be strings but not verified before being rendered in their respective React components. This results in `[object Object]` rendered to the screen as React calls the `toString()` method. Since the data is first sanitized using `JSON.parse()`, no security issues were uncovered. However, if in the future a vulnerability is discovered in `JSON.parse()`, this could possibly result in remote code execution.

Mitigation

Currently, protection against this attack comes from `JSON.parse()`, which is a widely regarded sanitization method that safely deserializes JSON payloads. However, we recommend a defense in depth strategy that could include verifying that elements are strings before rendering them in React components or using a secure serializer such as `serialize-javascript`.

Status

The Hiro team has responded they have chosen not to implement the suggestion mitigation as an exploit would require a vulnerability in the `JSON.parse` library. At the time of this verification, the suggested mitigation remains unresolved.

Verification

Unresolved.

Suggestion 6: Use Conforming DIDs

Location

`auth/src/dids.ts#L6-L15`

Synopsis

Decentralized Identifiers (DIDs) are a relatively new standard that are used in JWT, and in turn are used by the Stacks Wallet Extension. The DIDs used in this case refer to Bitcoin addresses and do not conform to the format as described in the specification and corresponding registry.

Mitigation

We recommend ensuring that the DIDs used make use of the format mandated in the specification.

Status

The Hiro team has stated they intend to take a broader look at changing their DID format as changing the existing DID format would require updates to other aspects of the system. At the time of this verification, the suggested mitigation remains unresolved.

Verification

Unresolved.

Suggestion 7: Use FNV-1a Instead of `hashCode`

Location

`encryption/src/utils.ts#L27-L36`

Synopsis

The `hashCode` function is used to compute the HD derivation index for applications, which appears to be ad-hoc reengineering of the known and proven FNV function family.

Mitigation

We recommend using the 32 bit FNV-1a hash and truncate it to 31 bit.

Status

The Hiro team has responded that implementing the suggested mitigation would break backward compatibility for their existing applications. They further state that, given their limited knowledge of whether a user has logged into the application before, changing the logic risks breaking compatibility for many users. At the time of this verification, the suggested mitigation remains unresolved.

Verification

Unresolved.

Suggestion 8: Error Message in `getBufferFromBN` Looks Incorrect

Location

`encryption/src/ec.ts#L130`

Synopsis

The error message “`Generated a 32-byte BN for encryption. Failing.`” appears to be a forgotten copy and paste, as the function is concerned with getting the binary encoding of a given big number, instead of generating a new big number.

Mitigation

We recommend correcting this error and replacing it with a correct and more accurate error message.

Status

The Hiro team has responded that they are tracking this issue and intend to improve this error message. At the time of this verification, the suggested mitigation remains unresolved.

Verification

Unresolved.

Suggestion 9: Document the HD Wallet Derivation Tree

Location

`wallet-sdk/src/derive.ts#L81`

Synopsis

In order to understand and evaluate the HD wallet derivation tree, it is important to first understand the mechanisms and assumptions of the system. Additional documentation would aid considerably in understanding which derivation is used in what cases, providing specific details on why it is used in some instances and why it is not used in others.

Mitigation

We recommend clearly and publicly documenting the HD wallet derivation tree in order to allow both users and reviews to understand the system and reason about and identify potential vulnerabilities.

Status

The Hiro team has responded that they are considering the existing derivation path logic and are exploring alternative backwards-compatible approaches. At the time of this verification, the suggested mitigation remains unresolved.

Verification

Unresolved.

Suggestion 10: Increase Test Coverage

Location

`blockstack/ux`

`blockstack/stacks.js`

Synopsis

`blocksack/ux` contains very few unit tests. Inclusion of unit tests in the CI and continuous deployment workflows is considered best practice and aids in automating the process of verification. Higher test coverage increases trust in the system and enables early detection of bugs and errors.

`blockstack/stack.js` reflects a considerable amount of line unit test coverage across the `auth`, `encryption`, and `wallet-sdk` packages. However, we suggest maintaining both high branch and line coverage ratios for security critical packages. High branch coverage allows more possible execution flows to be covered in the system while sufficient line coverage provides a better indication of how much code is being tested.

Mitigation

We recommend improving both line and branch coverage ratios (80% or higher) and continuously maintaining a high unit test coverage for both `blocksack/ux` and `blockstack/stack.js`.

Status

While the Hiro team has increased test coverage, the ratio is still below our recommendation. As such, this suggestion is partially resolved and we recommend increasing test coverage to 80% or higher.

Verification

Partially Resolved.

Suggestion 11: Revisit Security Architecture Design

Synopsis

The security architecture of the Stacks Wallet Extension is susceptible to vulnerabilities due to the nature of the browser ecosystem. The security guarantees of cryptographic secrets are tenuous given any extension in the browser will have access to an application's `localStorage`. This makes secure message signing impossible, as signing keys cannot be intrinsically trusted. Furthermore, the use of JWT for authentication and transaction requests consequently is illogical as wallet extension has no guarantees to the identity of the signing party. As a result, the trust relationship between applications and wallet cannot be secured.

Remediation

We recommend redesigning the system architecture so it is a secure framework for constructing the application. This begins with a methodological approach to the protocol design to be used as the basis from which the protocol is implemented and stating the system's overall desired functional and security properties, as well as listing the security properties and assumptions provided by the protocol's individual components.

In order to clearly understand the required functional and security properties of the building blocks used, we recommend engineering the protocol such that it fulfills both the desired functionality while adhering to the required security properties. This includes cryptographic primitives as well as the security mechanisms provided in the browser.

Once the assumptions and functionality are specified, a methodological approach can be taken to designing the solution, accompanied by clear documentation that describes how it achieves the security claims. While this is a significant undertaking, the effort can be aided by knowledgeable security teams through secure design consulting, followed by a security audit of the protocol once it has been redesigned and reimplemented.

Status

The Hiro team has responded that they will continue to consider a long term strategy by revisiting their security architecture design and have stated their intent to notify users of best practices to help mitigate potential issues (i.e. using a dedicated browser) when using Stacks applications. At the time of this verification, the suggested remediation remains unresolved.

Verification

Unresolved.

About Least Authority

We believe that people have a fundamental right to privacy and that the use of secure solutions enables people to more freely use the Internet and other connected technologies. We provide security consulting services to help others make their solutions more resistant to unauthorized access to data and unintended manipulation of the system. We support teams from the design phase through the production launch and after.

The Least Authority team has skills for reviewing code in C, C++, Python, Haskell, Rust, Node.js, Solidity, Go, and JavaScript for common security vulnerabilities and specific attack vectors. The team has reviewed implementations of cryptographic protocols and distributed system architecture, including in cryptocurrency, blockchains, payments, and smart contracts. Additionally, the team can utilize various tools to scan code and networks and build custom tools as necessary.

Least Authority was formed in 2011 to create and further empower freedom-compatible technologies. We moved the company to Berlin in 2016 and continue to expand our efforts. Although we are a small team, we believe that we can have a significant impact on the world by being transparent and open about the work we do.

For more information about our security consulting, please visit https://leastauthority.com/security-consulting/.

Our Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protect users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.