



CRITICAL SECTION

Abbreviated technical report

Project name: Sui Wallet source code review

Client: Mysten Labs, Inc.
379 University Ave., Suite 200
Palo Alto, CA 94301
United States

Authored by: Critical Section Security Oy
Fabianinkatu 4 B 10
00130 HELSINKI
Finland

Email: hello@criticalsection.fi
<https://www.criticalsection.fi/>
Business ID: 270661-4
VAT ID: FI32706614

Version: 0.9

Table Of Contents

1. Background	3
2. Architecture	5
2.1 High level components and message passing	5
2.2 Key creation and storage	7
3. Findings	8
3.1 Finding 1: Error messages propagated across trust boundary	8
3.2 Finding 2: Types not enforced across trust boundary	8
3.3 Finding 3: Content Script can connect to UI port and access raw entropy	10
3.4 Finding 4: Message event handlers don't reject synthetic events	11
3.5 Finding 5: Unlock password persists in memory after locking the wallet	12
3.6 Finding 6: Background tabs can keep the wallet unlocked	14
3.7 Finding 7: Privacy: Some account information stored unencrypted	14
4. Other risks and controls audited	16

1. Background

1.1 Target application

The application/code base under review is the Sui Wallet ([Chrome Extension Store](#), [Github](#)), a browser plug-in wallet for the Sui blockchain. The client informed us that they've performed both an internal and external audit of the wallet, but contracted us for a further code review as this is a security-critical component.

In terms of lines-of-code estimates, the client estimated ~8k LoC of sensitive code in a larger app of about ~19k LoC. All in-scope code was in TypeScript. The audit was done mostly based on GitHub revision 3c6aa516961b02bdac5b5c803561564ae43fc6f.

1.2 Focus areas and targeted scenarios

The client highlighted the following code paths as focus areas:

- ./src/background/*
- ./src/dapp-interface/*
- ./src/content-script/*
- ./src/manifest/*
- ./src/shared/*
- ./src/ui/app/background-client/*
- ./src/ui/app/wallet/*
- ./src/ui/app/pages/initialize/*
- ./src/ui/app/pages/approval-request/*
- ./src/ui/app/pages/site-connect/*
- ./src/ui/app/components/menu/content/*
- ./src/ui/app/redux/*

Further, in terms of top risks scenarios, the client highlighted the following risks:

- Remote exfiltration of keys
- Signing/sending transactions without approval
- Tricking the user into signing a malicious transaction (e.g., showing one recipient address in the signing prompt and swapping for an attacker address)

Based on these scenarios, we prioritized the perspective of an attacker who operates a malicious web site. Mapping the attack surface from that vantage point requires some understanding of the architecture and message passing so we will describe this part of the architecture in section 2.1. We'll further discuss key storage and generation in section 2.2

1.3 Exclusions

The audit was performed as a 5-working day audit, so this audit – like no audit – can capture all risks. We highlight the following risks as examples of topics that we did not review in detail during this timeframe:

- Making sure the UI accurately represents the semantics of complex transactions – this would require more time for us to study the details of Sui transactions and blockchain
- The code quality and trustworthiness of dependencies
- Analytics and configuration providers (Growthbook, Posthog, Sentry) and whether their use could cause sensitive information to leak to third parties. This is suspected to be the root cause behind the [Slope wallet case](#) in the Solana ecosystem.
- The security of the development, build, and test environments and pipelines
- Interaction with Ledger devices
- State management in frameworks such as React itself, Redux, Formik, or @tanstack/react-query, and whether sensitive parts of the state (for instance, Redux action parameters containing the password) might be persisted or logged somewhere
- Privacy risks – for instance it seems possible to track users by setting a custom favicon URL for each user (and monitoring the IP address/referer field values of requests hitting it), but we didn't explore this further.

2 Architecture

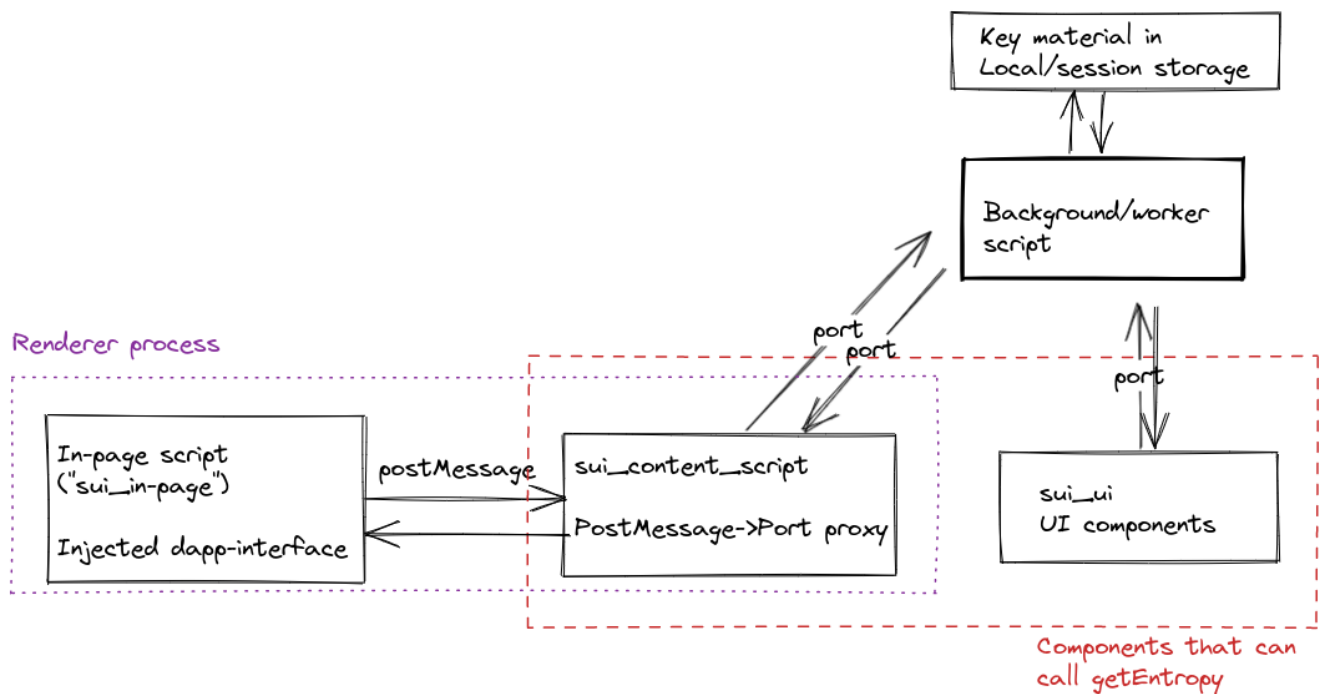
2.1 High level components and message passing

The architecture involved in the wallet's interaction with dapps can be broken down to four components that communicate with each other

1. In-page script - apps/wallet/src/dapp-interface
2. Content script - apps/wallet/src/content-script
3. Background/worker script - apps/wallet/src/background
4. "The UI" running in chrome-extension://opc... - apps/wallet/src/ui

When navigating to a dapp, the content script (#2) injects the web3 API by means of temporarily insertion of a script tag, which loads the dapp-interface script (#1), which in turn allows the web pages APIs interact with the WalletStandardInterface class.

The content script (#2) shares the DOM but not the JS *isolated world* with the dapp, whereas the injected script (#1) shares the DOM and JS *isolated world* with the dapp. Due to the need to have access to the DOM, both #1 and #2 execute in the same Chrome renderer process, which in turn means they have less sandboxing between them.



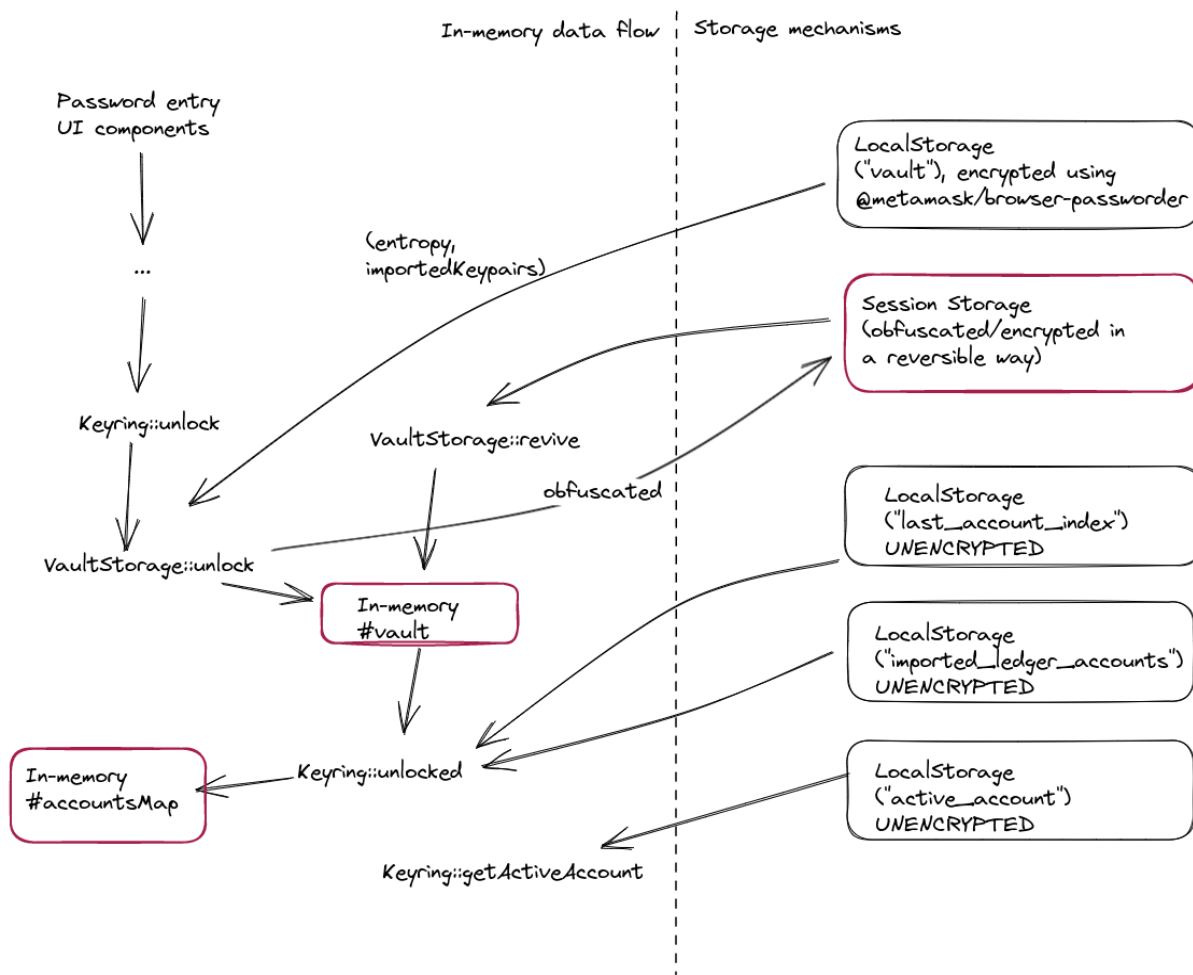
In understanding the trust boundary, it is important to clarify a few things

- A website can skip `WalletStandardInterface` to directly interact with the content script (which forwards the messages to the worker script) using **`window.postMessage`**. Thus no assumptions should be made in about the well-formedness of messages processed by `ContentScriptConnection.handleMessage`. While no major vulnerabilities were found here, we will highlight a few opportunities to harden this trust boundary in the findings section.
- Despite the UI, background/worker script, and content script (#2, #3, #4 above) being separate components, they are relatively similar in trust level in the current design. This is mainly because the **code executing in the content script could connect to the UI port of background script** (confirmed w/ an experiment done in Chrome DevTools). The UI port of the background supports sensitive operations such as `getEntropy`, which returns the raw entropy. We created finding #3 to denote this hardening opportunity.

2.2 Key creation and storage

Keys are added to the wallet either by creating them, importing them, deriving them or using a key stored on an external Ledger device. When necessary, the new entropy needed ultimately comes from the `randomBytes` function of the `@noble/hashes/utils` dependency. The data flow is `randomBytes() → getRandomEntropy() → VaultStorage::create → Vault` constructor. A quick look into the code of this dependency [suggests](#) the underlying entropy comes from the built-in [WebCrypto API function `getRandomValues\(\)`](#) which according to documentation should generate values of sufficient cryptographic quality.

To illustrate the storage of existing keys, we drew the diagram below which illustrates the key material handling as performed by the wallet during unlocking/reviving the wallet. The components with plaintext/reversibly encrypted key material have been highlighted in red. The encryption used is `@metamask/browser-passworder` which seems to be based on AES-GCM and PBKDF2 ([with 10 000 iterations](#)). In the scope of this review (and as we're not cryptography specialists), we will not be discussing the merits of PBKDF2 vs other algorithms such as `scrypt` / `Argon2`.



3. Findings

Summary

During the course of this review, no major exploitable vulnerabilities were identified. Still, we identified several concrete measures that we recommend implementing in order to improve security and decrease the likelihood of exploitation. These recommendations will be described below in rough order of our estimation of priority, but prioritization is always to some degree subjective and depends on the client's business priorities.

Finding 1: Error messages propagated across trust boundary

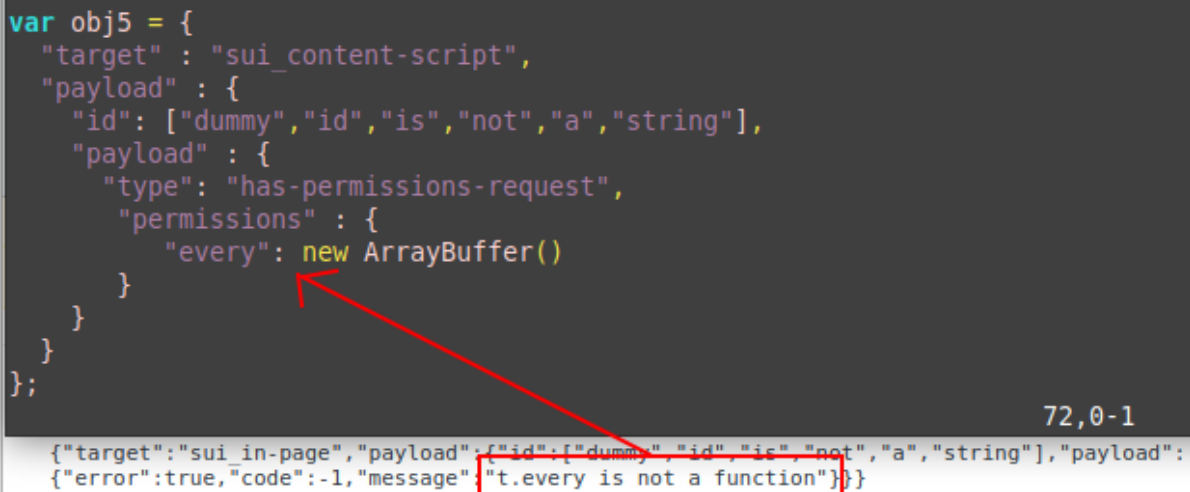
ContentScriptConnection.ts has the following code:

```
172         } catch (e) {  
173             this.sendError(  
174                 {  
175                     error: true,  
176                     code: -1,  
177                     message: (e as Error).message,  
178                 },  
179                 msg.id  
180             );  
181         }
```

This propagates raw error messages across the trust boundary (see Architecture section for discussion of the trust boundaries). At times error messages can include sensitive information. We recommend propagating only error messages specifically designed to be safe to be sent to untrusted parts of the code. See the following finding for more detail and a concrete example.

Finding 2: Types not enforced across trust boundary

While the code has extensive "schema" specifications of expected payload formats in **src/shared/messaging/messages/payloads**, there is no runtime enforcement of this schema. Combined with the error message propagation described in finding #1, this provides unnecessary avenues for an attacker to manipulate the program into unexpected states. The following screenshot shows our experiment where we got the trusted code to invoke a provided ArrayBuffer as a function, which naturally fails:



```
var obj5 = {  
  "target" : "sui_content-script",  
  "payload" : {  
    "id": ["dummy","id","is","not","a","string"],  
    "payload" : {  
      "type": "has-permissions-request",  
      "permissions" : {  
        "every": new ArrayBuffer()  
      }  
    }  
  }  
};
```

72,0-1

```
{"target":"sui_in-page","payload":{"id":["dummy","id","is","not","a","string"],"payload":{"error":true,"code":-1,"message":"t.every is not a function"}}}
```

For completeness, we note here that we confirmed that it was not possible to pass a function (instead of an `ArrayBuffer`) as the `"every"` element and achieve code execution this way. We attempted this in practice as well as confirmed that the underlying [structured clone](#) algorithm used by Chrome to transport the messages does not support functions:

"Function objects cannot be duplicated by the structured clone algorithm; attempting to throws a `DataCloneError` exception." ([Source: Mozilla web docs](#))

If there ever was a future change to JavaScript that would support structured cloning of functions or other callable gadgets that would allow user-provided logic to execute, this could lead to a catastrophic compromise of the security guarantees provided by the wallet (any website could read the entropy). For this reason, we strongly recommend enforcing the schema. As a quick stopgap a roundtrip to JSON and back would limit the types of objects that can be transported across the trust boundary, but full schema validation would be preferable.

Finding 3: Content Script can connect to UI port and access raw entropy

The content script is more exposed to attackers than the background/UI scripts due to at least two factors:

- 1) it shares the DOM with the untrusted page and
- 2) it runs in the Chrome render process (and e.g. could be compromised with a V8 exploit without any further sandbox escape vulnerabilities).

Therefore, as defense in depth measure, we should try to limit the privileges an attacker could obtain as a result of a content script compromise. As described in the Architecture section, there is no control preventing the content script from connecting to the port meant for UI components:

```
27         let connection: Connection | KeepAliveConnection;  
28         switch (port.name) {  
29             case ContentScriptConnection.CHANNEL:  
30                 connection = new ContentScriptConnection(port);  
31                 break;  
32             case UiConnection.CHANNEL:  
33                 connection = new UiConnection(port); // [1]  
34                 break;
```

Note the lack of checks at point [1] in the code. We've also confirmed this in practice with an experiment done using Chrome DevTools (steps for doing this were described to Mysten developers on Slack).

As the UI port provides access to sensitive operations such as `getEntropy`, which returns the raw entropy, we recommend adding an origin check to make sure that the connections to the UI port indeed come from the UI.

Finding 4: Message event handlers don't reject synthetic events

In addition to using `window.postMessage`, message event handlers can be communicated with by creating and dispatching so-called *synthetic events* which can violate normal assumptions. For instance, fields such as the *source* and *origin* can be *spoofed*. For this reason, security-conscious event handlers should confirm that inbound events are not synthetic by checking the `isTrusted` property of inbound events. The property is documented as follows

"The `isTrusted` read-only property of the `Event` interface is a boolean value that is true when the event was generated by a user action, and false when the event was created or modified by a script or dispatched via `EventTarget.dispatchEvent()`."

Source: <https://developer.mozilla.org/en-US/docs/Web/API/Event/isTrusted>

The code in `WindowMessageStream.ts` does not have such a check:

```
29         this.messages = fromEvent<MessageEvent<WindowMessage>>(
30             window,
31             'message'
32         ).pipe(
33             filter(
34                 (message) =>
35                     message.source === window &&
36                     message.data.target === this._name
37             ),
38             map((message) => message.data.payload),
39             share()
40         );
41     }
```

As a result, the message handlers process synthetic events such as the one demonstrated by the code below:

```
var data = {  
  "target" : "sui_content-script",  
  "payload" : {  
    "id": "dummy-id",  
    "payload" : {  
      "type": "stake-request",  
      "validatorAddress": "1",  
    }  
  }  
};  
  
var fakeEvent = new MessageEvent(  
  "message", {  
    "data" : data,  
    "origin": document.location.href,  
    "source": window});  
  
window.dispatchEvent(fakeEvent)
```

However, we were not able to identify any security consequences of this despite the spoofing possibilities: unlike `window.postMessage`, synthetic events cannot be dispatched to other origins due to the normal browser cross-origin checks. Thus a website seems to be able to attack only its own content script (which correctly reports the origin down to the background script). Regardless, we recommend adding the `isTrusted` check here as a hardening measure.

Finding 5: Unlock password persists in memory after locking the wallet

The wallet contains a "lock" functionality. Based on the naming and presentation of the feature, a user might reasonably expect that after locking their wallet, it is inaccessible without the password even if their laptop got stolen. We performed brief testing and verified that the unlock password persists in RAM even after the wallet is locked.

```

0000 0000 0000 0000 0000 0000 0000 0000 .....
6165 6e67 3853 6861 6967 0000 0000 0000 aeng8Shaig. ....
da32 7730 e939 d215 aa96 5f4e 9e7b bdc4 .2w0.9...._N.{..
4269 7463 6f69 6e20 7365 6564 0000 0000 Bitcoin seed....
aldb 17da 5e55 26bc 43a7 1b8b e68a 2ee1 ....^U&.C.....
6564 3235 3531 3920 7365 6564 e68a 2ee1 ed25519 seed....
d826 2412 a2e7 5ac5 1646 4116 0cde 81ad .&$...Z..FA.....
6d6e 656d 6f6e 6963 0000 0000 0000 0000 mnemonic.....
0000 0001 0001 4080 0000 0000 0000 0000 .....@.....
6564 3235 3531 3920 7365 6564 0000 0000 ed25519 seed....
0000 0001 0001 4080 0000 0000 0000 0000 .....@.....
7375 6920 7661 6c69 6461 7469 6f6e 0000 sui validation..
6d6e 656d 6f6e 6963 0000 0000 0000 0000 mnemonic.....

```

```

0308dac0: 1000 0000 0000 0000 0800 0000 0000 0000 .....
0308dad0: 2400 0000 0e00 0000 4500 6e00 7400 6500 $......E.n.t.e.
0308dae0: 7200 2000 5000 6100 7300 7300 7700 6f00 r. .P.a.s.s.w.o.
0308daf0: 7200 6400 0000 0000 1000 0000 0000 0000 r.d.....
0308db00: 0800 0000 0000 0000 1800 0000 0800 0000 .....
0308db10: 7000 6100 7300 7300 7700 6f00 7200 6400 p.a.s.s.w.o.r.d.
0308db20: 1000 0000 0000 0000 0800 0000 0000 0000 .....
0308db30: 0800 0000 0000 0000 1000 0000 0000 0000 .....
0308db40: 0800 0000 0000 0000 1800 0000 0800 0000 .....
0308db50: 7000 6100 7300 7300 7700 6f00 7200 6400 p.a.s.s.w.o.r.d.
0308db60: 1000 0000 0000 0000 0800 0000 0000 0000 .....
0308db70: 1c00 0000 0a00 0000 6100 6500 6e00 6700 .....a.e.n.g.
0308db80: 3800 5300 6800 6100 6900 6700 0000 0000 8.S.h.a.i.g....
0308db90: 1000 0000 0800 0000 7061 7373 776f 7264 .....password
0308dba0: 0800 0000 0000 0000 1000 0000 0000 0000 .....

```

To test this locally, one first takes a memory dump of the Chrome process(es). On Linux, we used a simple Python script similar to [this](#). We note that this didn't require root permission in the distro tested (Mint Linux) - just running it from the same UID as Chrome was enough. Chrome consists of several processes so you may want to script this. For reference, this is what we used on Linux: `for i in `pgrep chrome`; do echo $i; python dump.py $i; done;` On Mac the process would likely involve running `lldb --attach-pid <pid>` and then `process save-core "filename"`.

After dumping of the Chrome processes, one needs to search the dump files for the password (both straight ASCII and the Unicode variant that has the zero byte between characters). For instance, we used `rg 'S.?E.?C.?R.?E.?T' dumpfile` As this is a binary file, we will only get a yes/no response, to debug more you one pipe it via something like `xxd -c30` before grepping.

Unfortunately, we don't have a great recommendation for how to avoid this as long as the password prompting is done in JavaScript - JS does not allow for manual memory management.

Finding 6: Background tabs can keep the wallet unlocked

The wallet's auto-lock functionality locks the wallet after a user-selected period of inactivity. However, we found out that a background tab is able to pop up for instance a stake request window every $N-1$ minutes (where N =timeout, which we assume the attacker knows). The mere opening of this window is considered "user activity" even if the user does not even move their mouse.

This way a malicious site in a background tab could prevent the wallet from locking up. Popping up stake request window does not require any permissions and the user has no way of knowing which of their dozens+ of tabs might be responsible for it. In discussions with Mysten Labs developers, we learned that they are planning to change the definition of user activity to correspond to transactions, which should address the issue

The screenshot below shows the JavaScript code we used to demonstrate the issue:

```
var obj4 = {
  "target" : "sui_content-script",
  "payload" : {
    "id": {"dummy2" : "object"},
    "payload" : {
      "type": "stake-request",
      "validatorAddress": "1",
    }
  }
};

function schedule() {
  setTimeout(function x() {console.log("posting");window.postMessage(obj4);schedule();}, 10000);
}
schedule();
```

Finding 7: Privacy: Some account information stored unencrypted

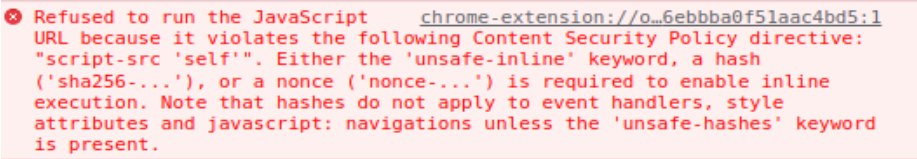
During our assessment of the key store, we observed that the code stores the active account address and the addresses of imported ledger accounts unencrypted in the local storage. While this does not allow access to the funds in those accounts, users might have the expectation that without the password it should not be possible to link their accounts to them even if someone steals their laptop. Combined with public account balances (at least in most blockchains, we haven't confirmed with Sui), this can be viewed as a privacy issue.

Given that the application already implements encryption of the entropy and importedKeypairs values, it looks relatively straightforward to encrypt these values too.

Relevant section of code in `apps/wallet/src/background/keyring/index.ts` is shown below:

```
495     private async getSavedLedgerAccounts() {  
496         const ledgerAccounts = await getFromLocalStorage<  
497             SerializedLedgerAccount[]  
498         >(STORAGE_IMPORTED_LEDGER_ACCOUNTS, []);  
499         return ledgerAccounts || [];  
500     }
```

4. Other risks and controls audited

Risk	Controls observed that address this risk
Risk of a cross-site script vulnerability in the content script	<p>The content script interaction with the web page is limited to:</p> <ul style="list-style-type: none"> • approx. 5 lines of code in <code>injectDappInterface()</code> • approx 20 lines of relevant code in <code>WindowMessageStream.ts</code> and <code>messages-proxy.ts</code> <p>The small amount of code limits the attack surface to a minimum. We found no found cross-site scripting vulnerabilities here. (Finding #4 is not a cross-site scripting vulnerability). Further, implementing our recommendation for finding #3 should decrease the consequences even if such a vulnerability was to be exploited.</p>
Using clickjacking against the UI components to make the user inadvertently approve a transaction	<p>Chrome does not allow for resources to iframed or linked to from websites unless they are listed as <code>web_accessible_resources</code> in the extension's manifest. Only the <code>dapp-interface.js</code> script file marked as <code>web_accessible_resource</code> in the manifest of the Sui Wallet.</p>
Cross-site scripting vulnerabilities in UI pages	<p>The code uses the React framework which mitigates most XSS risks. No calls to the <code>dangerouslySetInnerHTML</code> escape hatch were identified. One remaining risk with React are links to <code>javascript: URLs</code>, however we confirmed that the existing Content Security Policy blocks such JavaScript execution. Below is a screenshot of where we simulated a link to a <code>javascript: url</code></p> 
The risk that a malicious page could iframe a victim dapp and use <code>window.postMessage</code> to spoof transactions in its name	<p>The origin checks in <code>WindowMessageStream.ts</code> should thwart this attack. Furthermore, the iframed victim dapp would not even get its own content script as the content script is only injected to top-level frames. This is due to the fact that <code>all_frames</code> has not been set in the manifest of this extension.</p>

Deanonymizing users in Incognito windows by using state of the wallet	The wallet does not seem to function in incognito mode.
---	---