CRITICAL SECTION

Abbreviated technical report

Project name:	Targeted review of certain Sui Wallet updates (zklogin focus)
Client:	Mysten Labs, Inc. https://mystenlabs.com/
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
Date:	October 11, 2023

Table Of Contents

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. This is a targeted source code review of a few recent changes, primarily those related to a new ZkLogin feature (code paths identified in section 1.2).

In terms of lines-of-code estimates, the client estimated ~2k LoC of sensitive code . All in-scope code was written in TypeScript. The audit was performed mainly based on GitHub revision `da0d7881e366f78abb43b7b27661b50c8083c079`, though some checks were with more recent versions of the source code. Due to frequent changes to the code base in the course of the review, a few sections in this report can appear inconsistent if they refer to the state of code at different revisions.

1.2 Focus areas and targeted scenarios

The client highlighted the following code paths as focus areas:

- ./src/background/account-sources/*
- ./src/background/accounts/*
  - Emphasis on files under under zk/
- ./src/background/session-ephemeral-values.ts

Further, in terms of top risks scenarios, client personnel highlighted the following risks/research questions:

- How [the application] stores accounts in storage and memory
- A SourceHash mechanism whereby the wallet stores the hashed entropy of a mnemonic account source for use in comparisons (risks with the approach and alternatives)
- How the wallet does the OAuth-> zkproof flow
- If, by presenting a malicious OAuth app, there was [a way] to trick the wallet into weird behaviors.

1.3 Exclusions and limitations

The was a small source code review done in the equivalent of approximately two and a half days, so this audit – like no audit – can capture all risks.

It is important to understand that this is a review of the code in the wallet, so – despite the ZkLogin focus – **most parts of ZkLogin mechanism itself were not in scope** for this review. This includes how the proof is constructed (including algorithm choice and validity, arithmetic circuit(s) involved, proving service implementation), validation of ZkLogin transaction signatures on the server side, and further non-wallet concerns.

Further, we highlight the following risks as examples of topics that we did not review in detail during this timeframe:

- The ZkLogin SDK (`sdk/zklogin` directory) was mentioned as possibly not yet ready for review, so we didn't explicitly target it for review. However, we still looked at several functions inside this SDK as necessary to understand the bigger picture of the ZkLogin flow.
- The configuration of individual OIDC providers and whether e.g. the redirects URLs or any provider-specific permissiones had been configured correctly
- The code quality and trustworthiness of dependencies
- The security of the development, build, and test environments and pipelines as well as production infrastructure
- Interaction with Ledger devices beyond reviewing the in-scope code for obvious issues
- Deeper practical tests to investigate storage of temporary values incl. 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

2 ZkLogin integration

2.1 Architecture

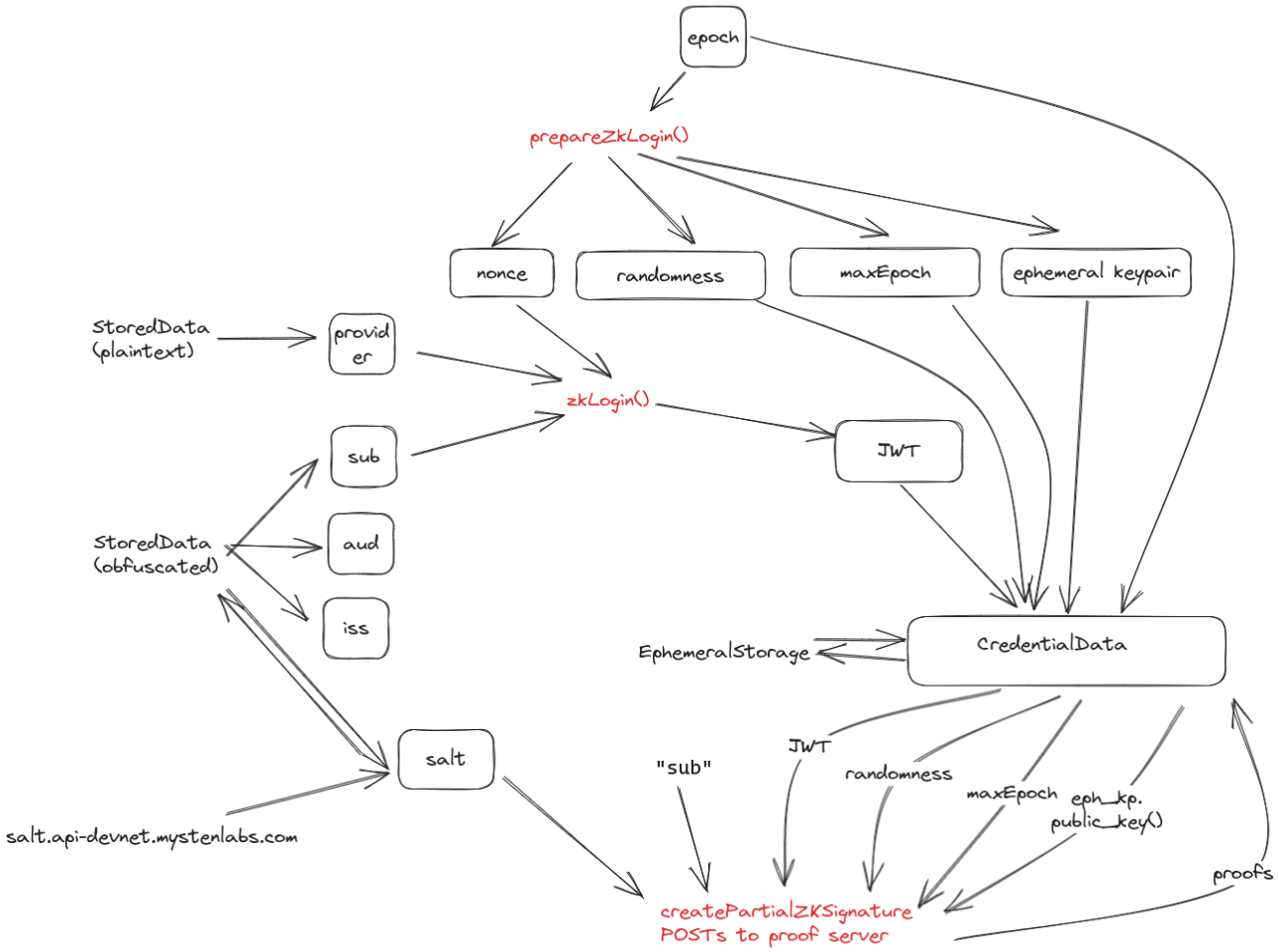
The ZkLogin feature essentially takes OpenID Connect (OIDC) -conforming OAuth 2.0 identity providers such as Google, Facebook, and Twitch, and at a high level allows the user to:

- Create a Sui address that is on one hand bound to their identity in the provider's system but on the other hand unlinkable to it by outsiders
- Authorize an ephemeral key to sign transactions as the corresponding account by creating a zero-knowledge proof that the user possessed a valid JWT issued by the OIDC provider

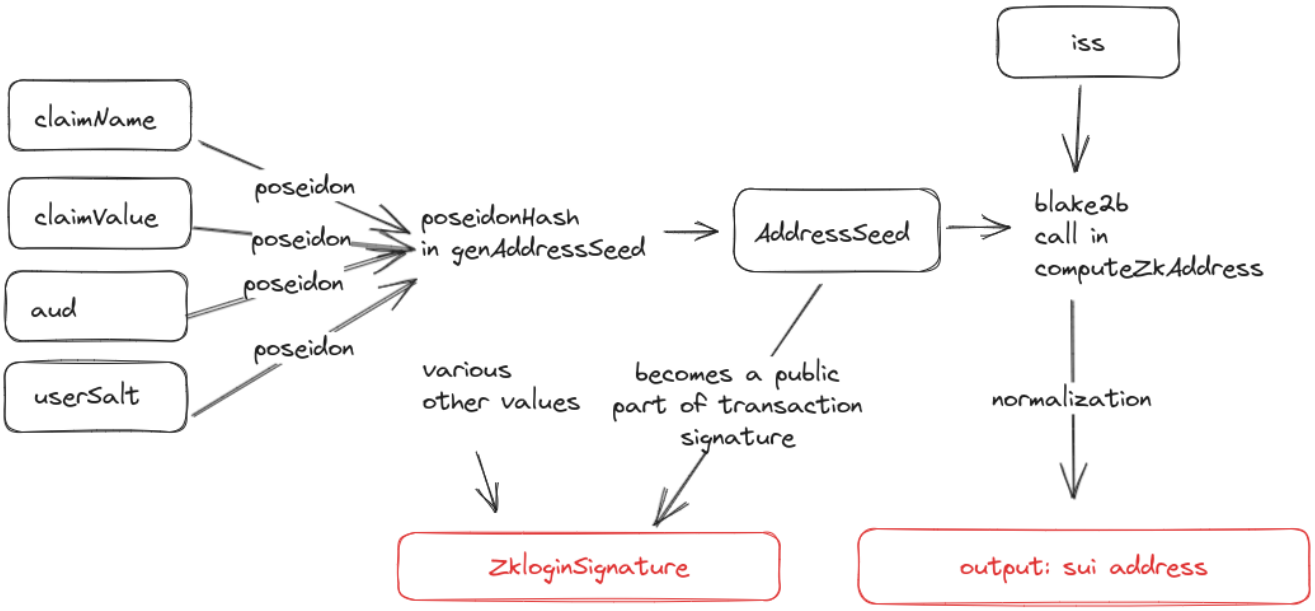
More technically, the contours of the login process are as follows:

1. The wallet creates an **ephemeral keypair** on client side
2. A nonce is created based on hashing the ephemeral public key, last validity epoch, and a random value ("JWT randomness")
3. An authentication flow is run by pointing the browser to a URL to the selected OIDC provider with the above nonce embedded in it. The resulting JWT (signed by the OIDC and provided to the relying party i.e. the wallet) is saved to the wallet's ephemeral storage. An key detail is that the signed JWT contains the nonce from step #2, cryptographically binding the values from step #2 together with the JWT
4. The JWT from step #3 is used to fetch a user-specific salt from **a salt server** (at time of original code review at `salt.api-devnet.mystenlabs.com`, but since then moved to `https://salt.api.mystenlabs.com`)
5. The JWT along with the salt and few other values is passed onto **a proof server** to create a zero-knowledge proof asserting the validity of the token and its binding to various intermediary values. The resulting proof is saved into the wallet's ephemeral storage
6. Transactions are then signed with the ephemeral key. The final signature embedded in transactions includes 1) the signature done with the ephemeral key combined with 2) the proof and a few other values (referred to `ZkLoginInputs`). This contains an address seed value (essentially a hash of the user identity and the salt), that can be used to derive the resulting sui address. The idea is that observers can check the proof that it maps to the given ephemeral public key, and then verify the signature done by that key.

The key in-scope files that implement the log-in are the `ZkAccount.ts` and `utils.ts` files in the zk/ subdirectory (later on moved to `zklogin/` and further renamed). The flow of values during the log-in process is depicted on the next page at the abstraction level of utils.ts calls.



The non-linkability of the the resulting address and the user id at the OIDC provider is achieved by the userSalt value at step #4, effectively making the salt service a privacy service. The following diagram illustrates the data flows related to calculation of the resulting ZkLogin sui address. An intermediary "address seed" value is published as part of BCS (Binary Canonical Serialization) encoded transaction signatures, allowing outsiders to construct the same sui address without revealing the identity of the user. We drew the diagram below during the audit to clarify the information that goes into the derivation of the sui address.



As this is not a cryptographic review and we are not cryptographers, we will not attempt to do a deeper review of the ZkLogin system's cryptographic guarantees. We also reiterate here that the review did not include major parts of the ZkLogin system such as the server-side validation of the signatures - see Exclusions section earlier in this report. For completeness, we'll mention however that the scheme appears to be heavily reliant Poseidon's (a relatively new cryptographic algorithm) preimage resistance.

A further topic to account for when developing ZkLogin and its integration to the wallet could be to look at threat models where Mysten Labs (or its infrastructure) would be considered an adversary. In the current implementation the wallet has two main dependencies to Mysten Labs infrastructure: the salt server and the proof server. Thus on a technical level they appear to have the ability to deny access to a specific individual's funds by e.g. refusing to hand out the salt of an individual user. However, we note that this salt value appears to be cached on first log-in and the ZkLogin documentation mentions use cases where the user would select the salt themselves. Alternatively, the proof service could refuse to construct the necessary proof for a specific user. We did not explore this angle in depth as it was not identified as a prioritized scenario or threat model, but mention it here for completeness.

2.2 Findings

2.2.1 Finding #1: [Medium] Weaknesses in dev-only client-id implementation

In related documentation (`doc/src/build/zk_login.md`), integrators of the ZkLogin SDK are advised of the option to use a so-called "dev-only client id". The documentation says this is for dev/testnets only, but there was no obvious control that would prevent someone from accidentally using this in production or developing a similar solution (as there was no warning about the security consequences).

The mechanism used for redirect URLs is dangerous since it permits any redirect URI. As a result any website the user visits could complete the OpenID flow (potentially silently) for a nonce of their choice and then follow the process described in section 2.1, ultimately obtaining the ability to sign ZkLogin transactions as the victim user. As this issue is clearly labelled not for production and is not exploitable in the wallet, we have downgraded this to a Medium issue (if it would affect the wallet, we would assess it as a Critical issue).

We also note that "dev redirect" backend developed for this feature can be used as a cross-site scripting (XSS) vulnerability, see:
https://zklogin-dev-redirect.vercel.app/api/auth#state=redirect_uri%3Djavascript%3Aalert%28document.domain%29%3B//

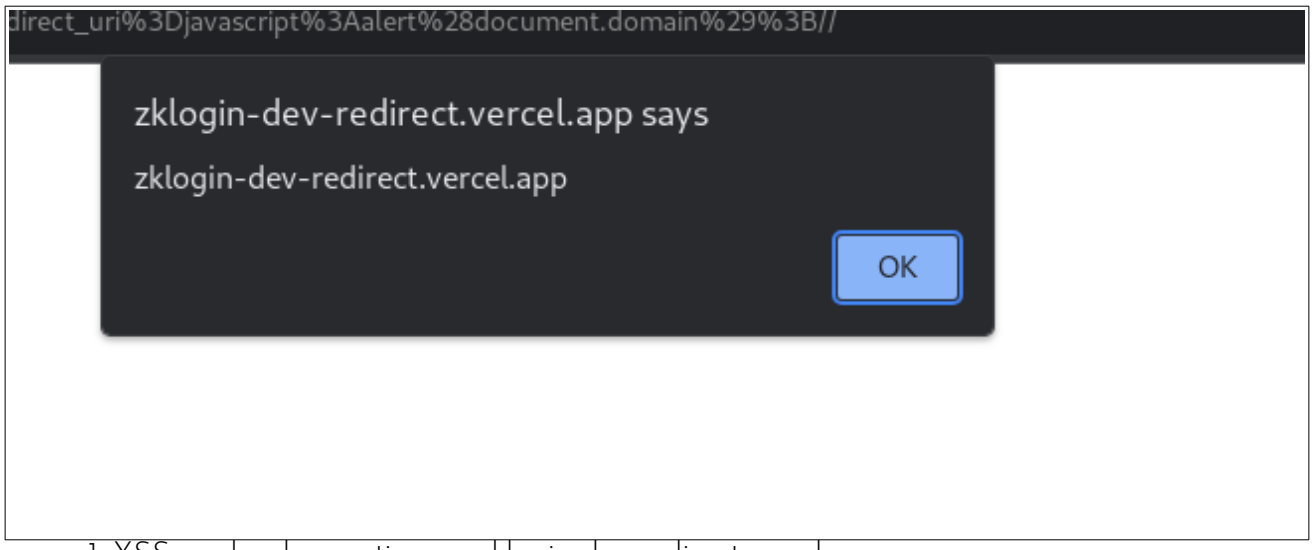


Image 1: XSS payload executing on zklogin-dev-redirect.vercel.app

If you wish to keep this mechanism available, one approach to limit risk at least to some level could be to maintain an allowlist of (user id → [allowed redirect URLs]) mapping and have users (=developers) authorize particular redirect URLs.

Status: Mysten Labs communicated to us on September 29th that they have shut down these client ids.

Finding #2: [Informational] Google infra implicitly trusted even when using other OIDC providers

When the `zkLoginAuthenticate` function (in utils.ts at revision `7f880af`) kicks off the OAuth process using the Browser.identity.launchWebAuthFlow API, it creates a redirect URL using the Browser.identity.getRedirectURL() function.

```
  88   |     params.append('redirect_uri', Browser.identity.getRedirectURL());
...
  98   |     if (!responseURL) {
  99   |         responseURL = new URL(
 100   |             await Browser.identity.launchWebAuthFlow({
 101   |                 url: authUrl,
 102   |                 interactive: true,
 103   |             }),
 104   |         );
 105   |     }
```

This is a convenience mechanism that makes the browser intercept the request (link to Chromium source code) to this "dummy" return URL and return the value as the result of the `launchWebAuthFlow` call. Per documentation the return URL is of the form `https://<app-id>.chromiumapp.org/*`. This means that the OpenID providers used by the wallet have to be configured to allow this URL as their redirect URL.

In normal operation (when launched via launchWebAuthFlow) the redirect never actually makes it to the chromiumapp.org server, and furthermore there is no server currently hosted at that address. However, as Google appears to control that domain, they or someone controlling their infrastructure could theoretically host a server at that address and then create a website that launches a normal OAuth flow (without using launchWebAuthFlow), with the redirect going to this malicious chromiumapp.org server. This would enable them to capture the results similar to the attack described above for Finding #1.

We're reporting this as an informational issue mostly for completeness and don't necessarily recommend a fix as Chrome extensions already place a level of trust Google by virtue of Google hosting Chrome's update servers and the extension store. In case you want to further minimize this risk, per Chromium source code it appears to be possible to construct redirect URLs using the custom `chrome-extension://` scheme where no server can be hosted. However, not all OIDC providers support such non-HTTP URLs.

3 Account storage and persistence

3.1 Architecture

The extension uses two main types of storage:

- **Ephemeral storage**, which is used to keep credentials for accounts/account sources whilst the wallet is in an unlocked state. Under the hood, the values are stored in session storage in an obfuscated form. In other words, they are. Encrypted but the necessary information to derive the keys is also in session storage and/or the code of the extension itself.
- **Stored Data**, which is used for longer-term persistence and is used to store the accounts the user has in their wallet. Key material here (if present) is encrypted with the user password. In some cases, such as some metadata for ZkLogin accounts, the stored data is obfuscated by encrypting it with a hardcoded password. The underlying storage here is IndexedDB (wrapped with Dexie) with a backup stored in local storage.

The encryption method used here is same as in our previous review i.e. based on the `@metamask/browser-passworder` libarary which in turn 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.

Overall, based on the source code review no major recommendations were found with this approach. When unlocked, the wallet by design necessarily needs to keep the keys decryptable, and the wallet takes some precautions to make them harder to extract from memory. As agreed with the client, no memory scanning or similar tests were done here to assess the difficulty of extracting the keys from memory in practice. We include a few minor findings in the section below.

3.2 Findings

Finding: #3 [Low] Privacy issue in SourceHash mehcanism

We were specifically asked to look into a so-called sourceHash mechanism, which is used to make sure the same mnemonic-based (base) account is not added multiple times to the wallet by the user.

This is implemented in `background/account-sources/MnemonicAccountSource.ts` by calculating a hash of the to-be-added key and comparing it with such hashes of existing accounts. On a technical level, the sourceHash is calculated by a straightforward SHA256 hash:

```
73  |         const dataSerialized: MnemonicAccountSourceSerialized = {
74  |             id: makeUniqueKey(),
75  |             type: 'mnemonic',
76  |             encryptedData: await encrypt(password, decryptedData),
77  |             sourceHash: bytesToHex(sha256(entropy)),
78  |             createdAt: Date.now(),
79  |         };
```

In terms of unauthorized acccess to the key material we don't see a risk in this. Assuming no cryptographic vulnerability in SHA256 (a very well-known cryptographic hash algorithm), brute-forcing the entropy to recover the sourceHash preimage is of similar difficulty as bruteforcing the key directly, which should be not be feasible.

If we had to criticise this design, one potential privacy issue remains: without the password, it is possible to compare two wallets (say, on two people's computers) and conclude which accounts overlap. Furthermore, given the straightforward implementation it is possible that other software calculates a similar "SHA256 fingerprint" of the key, allowing an observer to conclude that the same key is stored into the Sui wallet. For the `encryptedData` field this concern does not apply as the underlying encryption function inside `@metamask/browser-passworder` generates a random salt that even for the same data gets encrypted to a different value each time.

If you choose to address this issue, one alternative is to generate a per-wallet salt value and save it into the storage. Using this per-wallet key, the sourceHash value could be calculated using a salted hash algorithm such as PBKDF2. As the sourceHash of the same key would be completely different between two wallets, an observer would not be able to deduce the overlap between the mnemonic accounts stored in two different wallets.

Finding #4: [Informational] Ledger account authentication can likely be bypassed

This issue is by design but for completeness we note that the wallet attempts to do its own password authentication for the Ledger accounts. This can lead to a false sense of security as it is done entirely at code level (without a cryptographic guarantee) and can thus likely be bypassed e.g. by using Chrome Developer Tools by for example changing the encrypted value to one encrypted with another password. However, our understanding is that Ledger devices implement their own access control, which is supposed to be the real security control preventing unauthorized access.

```
12  | export interface LedgerAccountSerialized extends SerializedAccount {
13  |     type: 'ledger';
14  |     derivationPath: string;
15  |     // just used for authentication nothing is stored here at the moment
16  |     encrypted: string;
17  | }

81  |     async passwordUnlock(password: string): Promise<void> {
82  |         const { encrypted } = await this.getStoredData();
83  |         await decrypt<string>(password, encrypted);
84  |         await this.setEphemeralValue({ unlocked: true });
85  |         await this.onUnlocked();

92  |     async verifyPassword(password: string): Promise<void> {
93  |         const { encrypted } = await this.getStoredData();
94  |         await decrypt<string>(password, encrypted);
95  |     }
```

3. Summary of Findings

Earlier in this report, we've identified the following issues (in priority order based on our assessment of risk):

- [Medium] Weaknesses in dev-only client-id implementation
- [Low] Privacy issue in SourceHash mehcanism
- [Informational] Ledger account authentication can likely be bypassed
- [Informational] Google infrastructure implicitly trusted even when using other OIDC providers

None of these issues identified would result in loss of funds in the current implementation, but are more precautionary warnings and/or minor privacy issues. We understand the first issue (dev-only client-id) has already been remediated by disabling the mechanism and a ticket has been logged for the second one. The informational issues are likely to be accepted/acceptable risks but are flagged for awareness.