

Tezori (T2)

Security Assessment

11.12.2020

Prepared For:

Marco Schurtenberger | *Tezos Foundation*
marco.schurtenberger@tezos.com

Prepared By:

Artur Cygan | *Trail of Bits*
artur.cygan@trailofbits.com

Will Song | *Trail of Bits*

will.song@trailofbits.com

Kristin Mayo | *Trail of Bits*

kristin.mayo@trailofbits.com

Paweł Płatek | *Trail of Bits*

pawel.platek@trailofbits.com

Changelog:

Dec 11, 2020: Draft delivered

Dec 15, 2020: Final report

[Executive Summary](#)

[Project Dashboard](#)

[Engagement Goals](#)

[Coverage](#)

[Recommendations Summary](#)

[Short term](#)

[Long term](#)

[Findings Summary](#)

- [1. Remote code execution via openExternal](#)
- [2. Unencrypted secretes in memory](#)
- [3. Insecure private key in-memory encryption](#)
- [4. Access to raw private key and signing without additional authentication](#)
- [5. Signing valid operation hashes via UI dialog](#)
- [6. Wallet file permissions](#)
- [7. Ignored exceptions](#)
- [8. Users can be tricked to blindly sign transactions](#)
- [9. Client-side request forgery through dApp authentication](#)
- [10. User interface bugs](#)
- [11. Wallet's password not cleared from a dialog box](#)
- [12. Operations can be injected by a Tezos node before signing](#)
- [13. Entrypoint not validated, possible injection of data to sign](#)
- [14. URL components not encoded](#)
- [15. Arguments to contract's parameters not encoded](#)
- [16. JSONPath argument is not escaped](#)
- [17. Discrepancies between master branch and latest release](#)

[A. Vulnerability Classifications](#)

[B. Code Quality Recommendations](#)

[C. Electron Application Hardening](#)

[D. Fix Log](#)

[Detailed Fix Log](#)

Executive Summary

From November 30 through December 11, 2020 Cryptonomic engaged Trail of Bits to review the security of Cryptonomic's Tezori (T2) wallet implementation along with ConseilJS, ConseilJS-softsigner, and ConseilJS-ledgersigner libraries. We conducted this assessment over the course of 4 person-weeks with 4 engineers.

During the first week, we reviewed available documentation and resources to familiarize ourselves with the product in addition to building the wallet and running existing tests. From there we ran some static analysis tools over the codebase and reviewed the results. We started an in-depth manual code review particularly focusing on:

- Untrusted input validation and possible injections
- Private key and seed phrase security
- Strength of key generation
- Correct use of cryptographic primitives
- Error handling
- Configuration of the Electron application

In the second week we continued with the manual code review and further investigated issues which we found in the first week.

We found seventeen issues ranging from high to informational severity. The seven high severity issues concern remote code execution on wallet user's machine ([TOB-TEZORI-001](#)), exposure and insecure private key manipulation ([TOB-TEZORI-002](#), [TOB-TEZORI-003](#), [TOB-TEZORI-004](#)), and a number of possibilities for third parties to perform unauthorized operations on user's behalf ([TOB-TEZORI-008](#), [TOB-TEZORI-012](#), [TOB-TEZORI-013](#)). The first medium severity finding is about a possibility to perform a limited request forgery on services listening on wallet user's private networks ([TOB-TEZORI-009](#)). The second allows an attacker to trick a user into signing an operation ([TOB-TEZORI-005](#)), similar to [TOB-TEZORI-008](#), however less automatic and requiring more sophisticated phishing. The third medium severity finding describes issues around maintenance of T2's master branch. The remaining findings are not an immediate threat, however fixing them will further improve the wallet's security. We also included [Appendix C](#) to help with Electron application hardening.

Currently, the wallet is vulnerable to a number of attacks which might lead to loss of funds. The weakest point is insufficient data validation which manifests itself in the majority of findings. The in-memory key storage and signing implemented by ConseilJS-softsigner remains insecure, and we advise to use a Ledger device instead until the issues are mitigated. Security critical areas, such as signing, secret storage, and code execution were not handled with appropriate care, which resulted in the most severe findings.

We recommend fixing the issues according to the recommendations without taking shortcuts, prioritizing the high severity ones. Tezori would benefit from a fix review, to ensure that all the high and medium severity issues are correctly mitigated. We also

recommend performing additional security assessments as the codebase evolves and more features are introduced.

Project Dashboard

Application Summary

| | |
|-----------|--|
| Name | Tezori (T2), ConseilJS |
| Version | T2 - 1.1.7b ConseilJS - e280b3e ConseilJS-softsigner - 92e54d1 ConseilJS-ledgersigner - b0dec4d |
| Type | TypeScript |
| Platforms | Windows, Linux, macOS (Electron) |

Engagement Summary

| | |
|---------------------|---------------------------------|
| Dates | November 30 - December 11, 2020 |
| Method | Whitebox |
| Consultants Engaged | 4 |
| Level of Effort | 4 person-weeks |

Vulnerability Summary

| | | |
|-------------------------------------|----|---------------|
| Total High-Severity Issues | 7 | ■ ■ ■ ■ ■ ■ ■ |
| Total Medium-Severity Issues | 3 | ■ ■ ■ |
| Total Low-Severity Issues | 2 | ■ ■ |
| Total Informational-Severity Issues | 5 | ■ ■ ■ ■ ■ |
| Total | 17 | |

Category Breakdown

| | | |
|-----------------|---|-------------------|
| Data Validation | 9 | ■ ■ ■ ■ ■ ■ ■ ■ ■ |
| Access Controls | 1 | ■ |
| Authentication | 1 | ■ |
| Cryptography | 1 | ■ |
| Data Exposure | 2 | ■ ■ |
| Error Reporting | 1 | ■ |
| Patching | 1 | ■ |

| | | |
|--------------------|----|---|
| Undefined Behavior | 1 | ■ |
| Total | 17 | |

Engagement Goals

The engagement was scoped to provide a security assessment of Tezori (T2) wallet in addition to the ConseilJS, ConseilJS-softsigner, and ConseilJS-ledgersigner libraries.

Specifically, we sought to answer the following questions:

- Is the untrusted input validated properly to prevent injections?
- Is it possible to perform unauthorized operations and steal funds in particular?
- Are secrets such as the private key, seed phrase, and wallet password stored and processed securely?
- What is the strength of the seed phrase generation?
- Are cryptographic primitives used correctly?
- Is the Electron application configured properly?
- Are the errors handled safely?

Coverage

T2. Manual code review, UI testing from a user's perspective, and static analysis tools.

ConseilJS. Manual code review and static analysis tools.

ConseilJS-softsigner. Manual code review and static analysis tools.

ConseilJS-ledgersigner. Manual code review and static analysis tools.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short term

❑ **Validate (or white-list if possible) URIs before using them in `shell.openExternal` function.** Only necessary schemes (like “https://”) should be allowed. Consider all the scenarios where data is passed to `shell.openExternal` including `openBlockExplorerForOperation` and `openBlockExplorerForAccount` (see Figure 1.1), which operates on values from configuration files. Also, add protections against request forgery (see [TOB-TEZORI-009](#)). [TOB-TEZORI-001](#)

❑ **Prompt a user for a password to the wallet before performing any action requiring access to the private key.** Store private key only as a ciphertext in memory by default and decrypt it with wallet’s password only when absolutely necessary, ensuring it will be swiftly removed from memory after it’s no longer used. [TOB-TEZORI-002](#), [TOB-TEZORI-003](#)

❑ **Prompt a user for a password to the wallet before performing any action requiring access to the private key.** For a better user experience, on systems which support biometrics, store the password in a secure system store and require biometric authentication to fetch the password. Store private key only as a ciphertext in memory by default and decrypt it with wallet’s password only when absolutely necessary ensuring it will be swiftly removed from memory after it’s no longer used. Additionally, implement a timeout for the “Account Keys” dialog after which the secret key is removed from the UI and memory. [TOB-TEZORI-004](#)

❑ **Limit the Sign & Verify dialog box to accept only data which is not a BLAKE hash.** For example, prefix the data with a constant string longer than BLAKE hash length. [TOB-TEZORI-005](#)

❑ **Explicitly set restrictive permissions for newly created wallet files, and check file permissions for already existing wallets upon opening.** [TOB-TEZORI-006](#)

❑ **Identify all the places where exceptions are ignored and add error handling code or remove the catch block to let the application crash.** If the latter happens, generate a crash report with instructions for volunteering users to send to developers. [TOB-TEZORI-007](#)

- ❑ **Limit the dApp Authentication Request dialog box to accept only data which won't lead to compromising the user's account.** For example, prefix the data with a constant string of length longer than BLAKE hash output. [TOB-TEZORI-008](#)
- ❑ **Display req.callback value somewhere in the application, inform the user that the signature will be sent to this address, and ask him to validate it before clicking on the "Authenticate" button.** [TOB-TEZORI-009](#)
- ❑ **Fix the mentioned UI issues.** [TOB-TEZORI-010](#)
- ❑ **Clear password input box when SendConfirmationModal is closed.** [TOB-TEZORI-011](#)
- ❑ **Validate blockHead.hash parameter after obtaining it from a remote node.** Right before signing, when operations will not be further manipulated, decode the data which is about to be signed and display it clearly to the user who will be able to catch a potential injection. [TOB-TEZORI-012](#)
- ❑ **Validate the endpoint length before encoding.** [TOB-TEZORI-013](#)
- ❑ **Properly encode, filter, or validate arguments before using them as part of an URL.** [TOB-TEZORI-014](#)
- ❑ **Properly sanitize data before using it in Michelson context.** [TOB-TEZORI-015](#)
- ❑ **Sanitize the operationGroupId value before passing it to the JSONPath function.** Enable the preventEval option. [TOB-TEZORI-016](#)
- ❑ **Warn users that they should not build the wallet from the master branch.** Provide clear information about which branch/commit/tag is the newest/recommended one. Moreover, inform users about which version Galleon is currently using. [TOB-TEZORI-017](#)

Long term

- ❑ **Always strictly validate data which can be manipulated by third parties.** Pay special attention to dangerous functions which have high exploitation potential. [TOB-TEZORI-001](#)
- ❑ **Always limit secret exposure to the absolute minimum.** Long exposure of secrets increases the window of opportunity for an attacker to steal them. [TOB-TEZORI-002](#), [TOB-TEZORI-004](#)
- ❑ **Don't store encryption keys alongside the encrypted data.** It doesn't provide additional security benefits. [TOB-TEZORI-003](#)
- ❑ **Make sure that it is not possible to sign hashes of operations with `signText` function, unless it's an intended functionality.** Add warnings for potentially dangerous functionalities. Also, make sure that data provided to `signOperation` is not a valid, encoded operation coming from a potentially untrusted source. We recommend prefixing such data with a fixed string, which must not be a valid transaction prefix. Less preferably, detect if the data starts with suspicious bytes and reject it. [TOB-TEZORI-005](#), [TOB-TEZORI-008](#)
- ❑ **Always explicitly set file permissions for sensitive data to be as restrictive as possible.** [TOB-TEZORI-006](#)
- ❑ **Add a note to wallet development guidelines to either handle the exceptions correctly or let them propagate through the call stack.** [TOB-TEZORI-007](#)
- ❑ **Ensure that the user is able to verify addresses that come from untrusted sources before performing requests to them.** If possible, white-list available destinations. Otherwise black-list them, disallowing addresses that are part of local (private) network ranges. [TOB-TEZORI-009](#)
- ❑ **Perform extensive UI and UX tests before each release.** [TOB-TEZORI-010](#)
- ❑ **Ensure that sensitive data is not cached and does not persist during UI navigation if not necessary.** [TOB-TEZORI-011](#)
- ❑ **Validate all data that comes from untrusted sources such as Tezos node or Conseil node, before using it.** [TOB-TEZORI-012](#)
- ❑ **Apply validation to all arguments that may come from untrusted input.** Enforce data formatting that complies with documentation. Take special care of data length and

allowed bytes - for example Michelson strings are restricted to a printable subset of 7-bit ASCII. [TOB-TEZORI-013](#)

❑ **Avoid creating “stringly typed” APIs when data is not an arbitrary string.** Introduce new data structures which will validate the data upon construction. [TOB-TEZORI-014](#), [TOB-TEZORI-015](#), [TOB-TEZORI-016](#)

❑ **Design, implement, and describe project release policy, versioning, and purpose of GitHub branches and tags.** Do it for all repositories concerned. [TOB-TEZORI-017](#)

Findings Summary

| # | Title | Type | Severity |
|----|---|--------------------|---------------|
| 1 | Remote code execution via openExternal | Data Validation | High |
| 2 | Unencrypted secretes in memory | Data Exposure | High |
| 3 | Insecure private key in-memory encryption | Cryptography | High |
| 4 | Access to raw private key and signing without additional authentication | Authentication | High |
| 5 | Signing valid operation hashes via UI dialog | Data Validation | Medium |
| 6 | Wallet file permissions | Access Controls | Low |
| 7 | Ignored exceptions | Error Reporting | Informational |
| 8 | Users can be tricked to blindly sign transactions | Data Validation | High |
| 9 | Client-side request forgery through dApp authentication | Data Validation | Medium |
| 10 | User interface bugs | Undefined Behavior | Informational |
| 11 | Wallet's password not cleared from a dialog box | Data Exposure | Low |
| 12 | Operations can be injected by a Tezos node before signing | Data Validation | High |
| 13 | Entrypoint not validated, possible injection of data to sign | Data Validation | High |
| 14 | URL components not encoded | Data Validation | Informational |
| 15 | Arguments to contract's parameters not encoded | Data Validation | Informational |
| 16 | JSONPath argument is not escaped | Data Validation | Informational |
| 17 | Discrepancies between master branch and latest release | Patching | Medium |

1. Remote code execution via openExternal

Severity: High

Type: Data Validation

Target: T2/src/utils/general.ts

Difficulty: Medium

Finding ID: TOB-TEZORI-001

Description

The wallet opens URI links with the `shell.openExternal` function which accepts URIs with protocol schemes other than `https://`. The related code is presented in Figures 1.1, 1.2 and 1.3. There are a few particularly dangerous schemes like `file://` and `smb://` which perform code execution when provided to the `shell.openExternal` function. We found that the argument to `shell.openExternal` can be controlled by a malicious party who can use the `file://` scheme to execute a binary present on the system where the wallet is running or `smb://` scheme on Windows to download and execute an arbitrary binary.

```
export function openLink(link) {
  shell.openExternal(link);
}

export function openBlockExplorerForOperation(operation: string, network: string =
'mainnet') {
  shell.openExternal(`${blockExplorerHost}/${network}/operations/${operation}`);
}

export function openBlockExplorerForAccount(account: string, network: string = 'mainnet') {
  shell.openExternal(`${blockExplorerHost}/${network}/accounts/${account}`);
}
```

Figure 1.1: [T2/src/utils/general.ts lines 113-123](#)

```
const onClick = (link: string) => {
  openLink(link);
};
```

Figure 1.2: [T2/src/featureModals/Auth/Auth.tsx lines 89-91](#)

```
function MessageBar() {
  /* redacted */
  function openLinkHandler(link) {
    if (link) {
      openLink(link);
      onClose();
      return;
    }
    /* redacted */
  }
  /* redacted */
  return (
    <SnackbarWrapper
      anchorOrigin={{ vertical: 'bottom', horizontal: 'center' }}
      open={open}
      onClose={() => onClose()}
      message={
```

```

        <MessageContent
          content={text}
          hash={formatHash()}
          realHash={hash}
          openLink={({link?: string}) => openLinkHandler(link)}
          onClose={() => onClose()}
          isError={isError}
          localeParam={localeParam}
        />
      }
    />
  );
}

```

Figure 1.3: [T2/src/components/MessageBar/index.tsx lines 30-88](#)

Exploit Scenario

An attacker prepares a URI in the form of `galleon://...` that contains a malicious payload targeting a wallet owner with address `tz1cisJCRTZ81fYaB3jMdziDcAuv97RnHxN3` (Figure 1.4).

```

In [1]: from urllib.parse import quote
In [2]: import base64
In [3]: import json
In [4]: 'galleon://auth?r=' +
quote(base64.b64encode(json.dumps({'requrl': 'file:///System/Applications/Calculator.app',
'target': 'tz1cisJCRTZ81fYaB3jMdziDcAuv97RnHxN3'}).encode()))
Out[4]:
'galleon://auth?r=eyJyZXF1cmwiOiAiZm1sZTovLy9TeXN0ZW0vQXBwbGljYXRpb25zL0NhbnGN1bGF0b3IuYXBwIi
wgInRhcmlldlCI6ICJ0eJfjaXNKQ1JUWjgxZ1lhQjNqTW96aURjQXV2OTdSbk4TjMifQ%3D%3D'

```

Figure 1.4: Generating malicious payload in Python.

The attacker tricks the wallet owner into opening the link, for example by placing it inside a malicious website or in an email. After opening the link, the user is asked by the operating system to open it in Tezori. If the user agrees and the wallet is already opened, code from Figure 1.5 is executed and the “dApp Authentication Request” dialog is presented (Figure 1.6).

```

} else if (['sign', 'auth', 'beaconRegistration', 'beaconEvent'].includes(pathname) &&
searchParams.has('r')) {
  const req = searchParams.get('r') || '';

  dispatch(setModalValue(JSON.parse(Buffer.from(req, 'base64').toString('utf8')),
pathname));

  if (pathname === 'sign') {
    dispatch(setModalActiveTab(pathname));
    dispatch(setModalOpen(true, pathname));
  } else {
    dispatch(setModalOpen(true, pathname));
  }
}

```

```
}
```

Figure 1.5: [T2/src/containers/App/index.tsx lines 59-70](#)

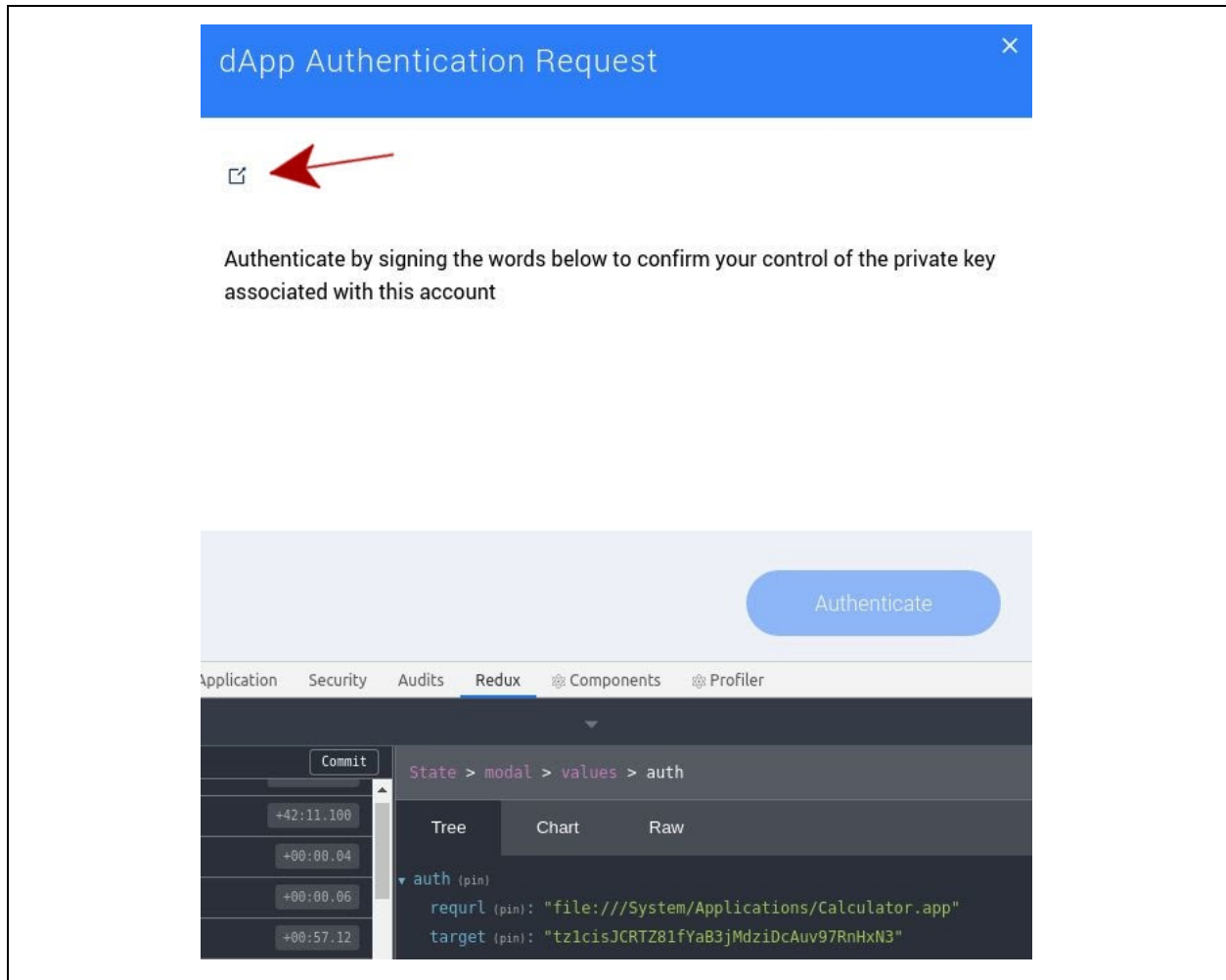


Figure 1.6: Malicious dApp Authentication Request dialog box.

When the victim clicks on the icon in the top left corner, the `shell.openExternal` function will be called and `Calculator.app` will execute. As stated in the description, more dangerous files could be executed from a remote NFS or SMB server.

Recommendation

Short term, validate (or white-list if possible) URIs before using them in `shell.openExternal` function. Only necessary schemes (like "https://") should be allowed. Consider all the scenarios where data is passed to `shell.openExternal` including `openBlockExplorerForOperation` and `openBlockExplorerForAccount` (see Figure 1.1), which operates on values from configuration files. Also, add protections against request forgery (see [TOB-TEZORI-009](#)).

Long term, always strictly validate data which can be manipulated by third parties. Pay special attention to dangerous functions which have high exploitation potential.

References

- [The dangers of Electron's shell.openExternal\(\)—many paths to remote code execution](#)

2. Unencrypted secretes in memory

Severity: High

Type: Data Exposure

Target: Wallet's memory

Difficulty: High

Finding ID: TOB-TEZORI-002

Description

The wallet stores the secret key and password as plaintext in memory. We verified that by reading the Redux store content in the debug console under `wallet.walletPassword` and `wallet.identities[0].secretKey` (Figure 2.1).

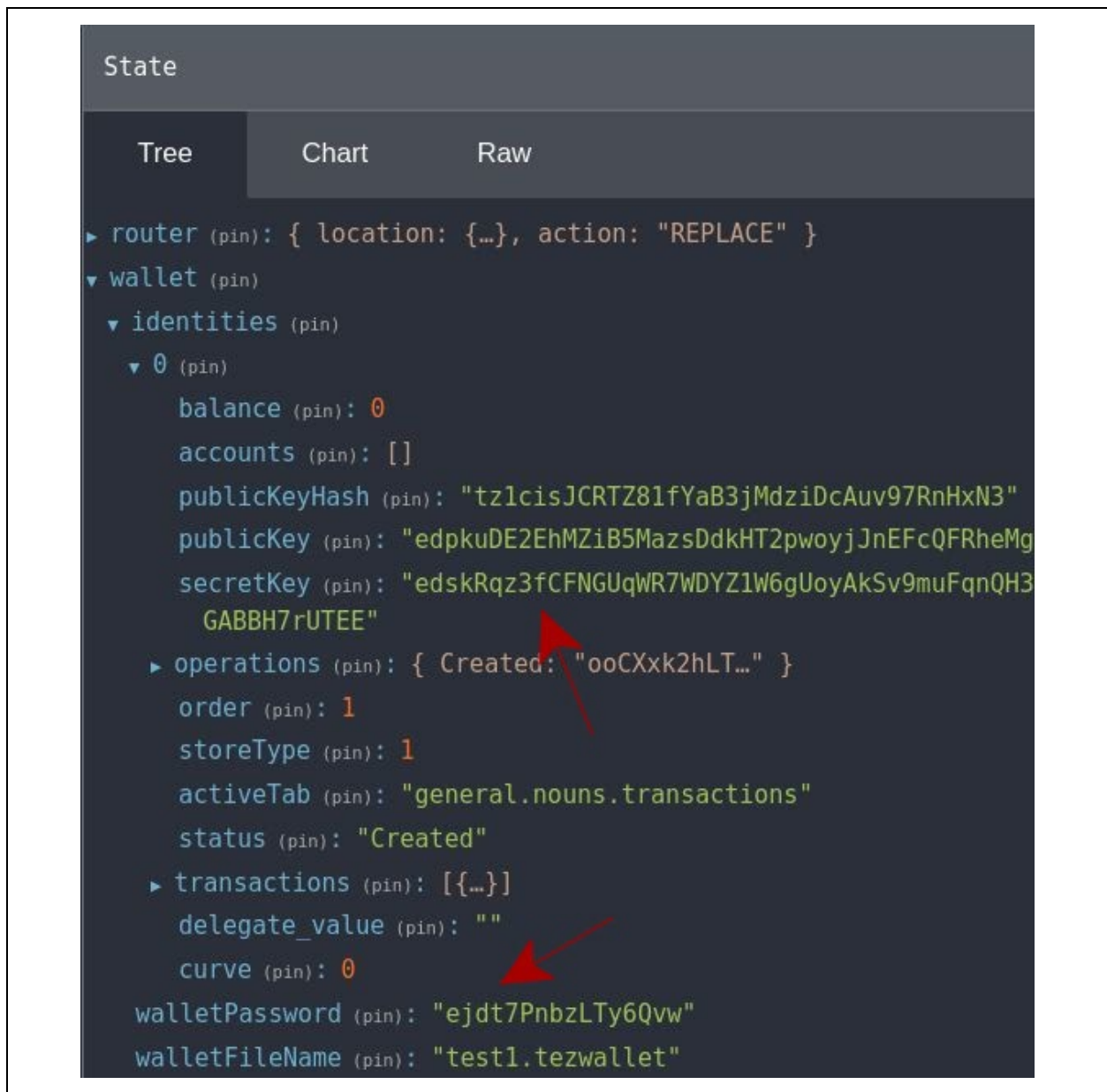


Figure 2.1: Redux store containing plaintext secrets.

There is also a `secretKey` field in the `KeyStore` interface, which may encourage storing the key as plaintext.

```
export interface KeyStore {  
  publicKey: string;  
  secretKey: string;  
  publicKeyHash: string;  
  curve: KeyStoreCurve;  
  storeType: KeyStoreType;  
  seed?: string;  
  derivationPath?: string;  
}
```

Figure 2.2: [ConseilJS/src/types/ExternalInterfaces.ts lines 14 - 22](#)

Exploit Scenario

An attacker finds a way to read the wallet's memory contents (for example by using Meltdown and Spectre attacks), extracts the private key and uses it to steal the owner's funds.

Recommendation

Short term, prompt a user for a password to the wallet before performing any action requiring access to the private key. Store the private key only as a ciphertext in memory by default and decrypt it with wallet's password only when absolutely necessary, ensuring it will be swiftly removed from memory after it's no longer used.

Long term, always limit secret exposure to the absolute minimum. Long exposure of secrets increases the window of opportunity for an attacker to steal them.

3. Insecure private key in-memory encryption

Severity: High

Type: Cryptography

Target: ConseilS-softsigner/src/SoftSigner.ts

Difficulty: High

Finding ID: TOB-TEZORI-003

Description

The private key (`redux.app.signer._secretKey`) is stored in memory as ciphertext along with a random passphrase (`redux.app.signer._passphrase`) and salt (`redux.app.signer._salt`) used to encrypt the key. The passphrase and salt are stored nearby the key as plaintext. When the application's memory is leaked, the mechanism provides no additional security apart from obfuscation.

Exploit Scenario

An attacker gains access to the wallet's process memory and reads the encrypted private key, password, and salt. The attacker can now decrypt the key and use it to steal the wallet owner's funds.

Recommendation

Short term, prompt a user for a password to the wallet before performing any action requiring access to the private key. Store the private key only as a ciphertext in memory by default and decrypt it with wallet's password only when absolutely necessary, ensuring it will be swiftly removed from memory after it's no longer used.

Long term, don't store encryption keys alongside the encrypted data. It doesn't provide additional security benefits.

4. Access to raw private key and signing without additional authentication

Severity: High
Type: Authentication
Target: Wallet UI

Difficulty: Medium
Finding ID: TOB-TEZORI-004

Description

The private key can be displayed (Figure 4.1) and used to sign arbitrary data (Figure 4.2) without additional authentication. Anyone with access to an unlocked wallet (once unlocked, it stays unlocked until the application is closed or user logs-out) can perform those two actions.

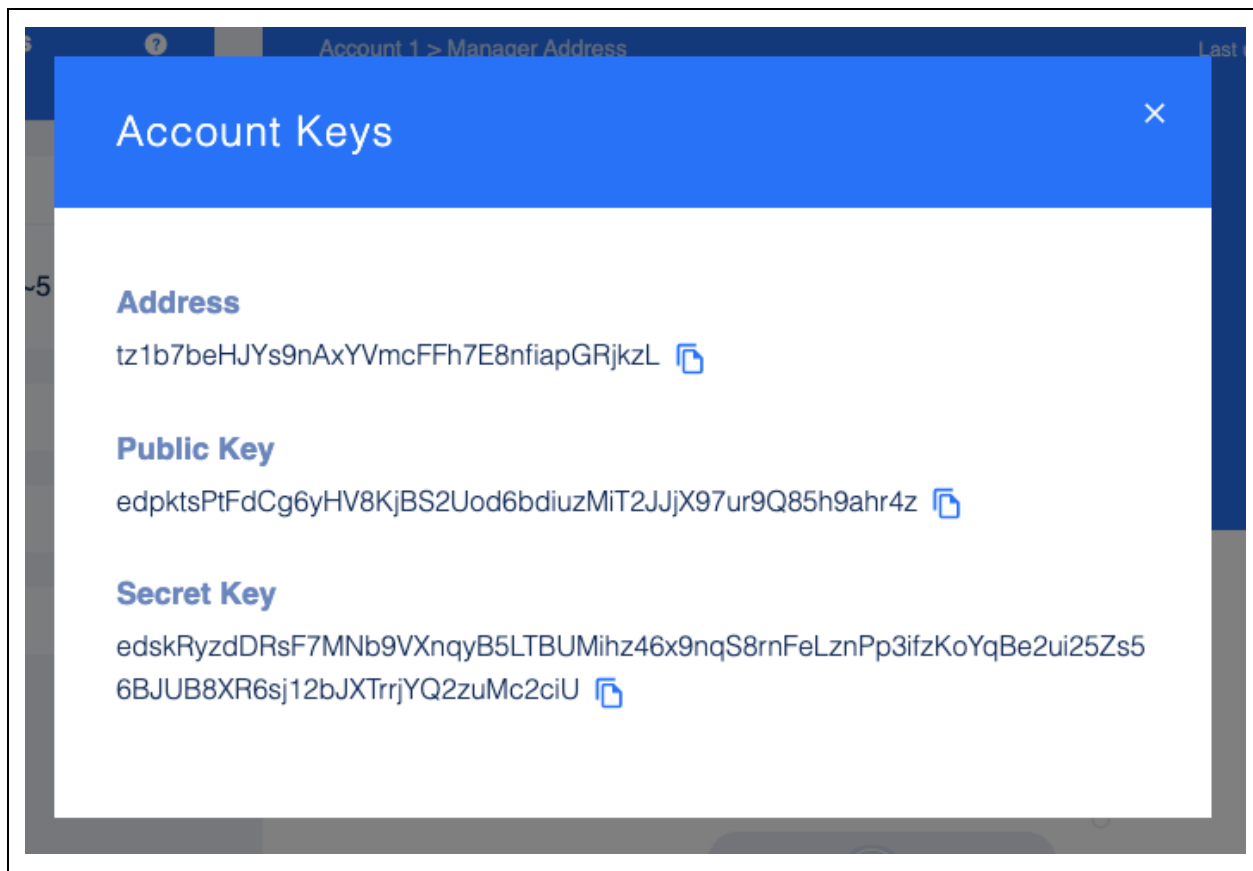


Figure 4.1: Secret key displayed in the UI without the owner's presence.

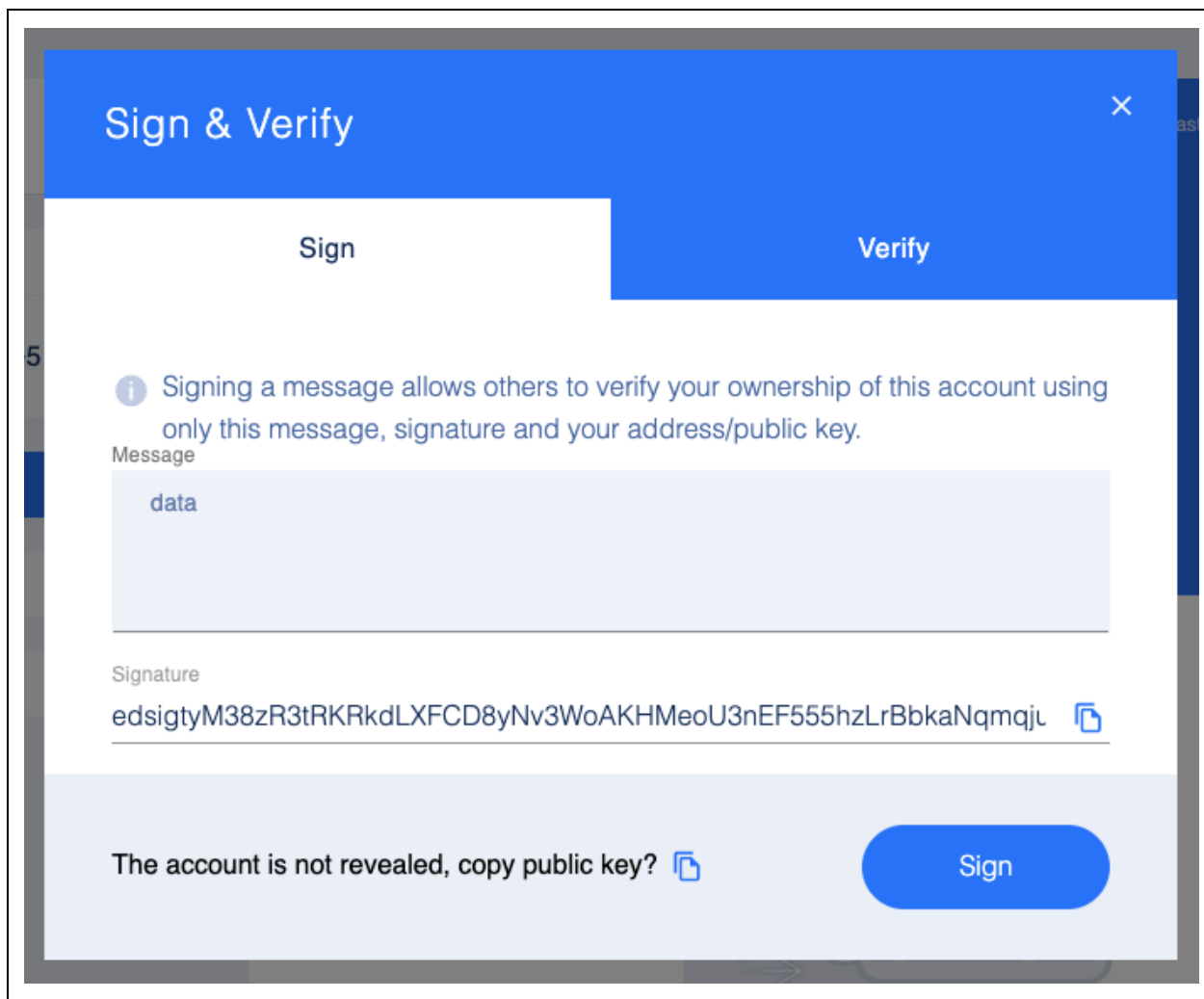


Figure 4.2: Signing arbitrary data without the owner's presence.

Exploit Scenario

A wallet user leaves the wallet unlocked and walks away from the computer. An attacker reveals the secret key and takes a photo of it. The wallet is compromised without the owner's knowledge.

In an alternative scenario, a wallet user gets infected with low-privileged malware which steals the private key, for example by capturing screenshots.

Recommendation

Short term, prompt a user for a password to the wallet before performing any action requiring access to the private key. For a better user experience, on systems which support biometrics, store the password in a secure system store and require biometric authentication to fetch the password. Store the private key only as a ciphertext in memory by default and decrypt it with wallet's password only when absolutely necessary, ensuring it will be swiftly removed from memory after it's no longer used. Additionally, implement a timeout for the "Account Keys" dialog after which the secret key is removed from the UI and memory.

Long term, always limit secret exposure to the absolute minimum. Long exposure of secrets increases the window of opportunity for an attacker to steal them.

References

- [Developing Secure Ledger Apps - Signing/disclosing keys without user approval](#)

5. Signing valid operation hashes via UI dialog

Severity: Medium

Type: Data Validation

Target: "Sign & Verify" UI dialog

Difficulty: High

Finding ID: TOB-TEZORI-005

Description

The message text box in the Sign tab in the Sign & Verify dialog box accepts arbitrary data, including binary, non-ascii bytes. Users can be tricked to sign data that is a valid hash of some operation. The input is encoded to utf8 prior to signing, but this does not prevent all possible attacks, it only makes them more difficult.

The messages are signed with the `signText` function. Figure 5.1 compares algorithms used to sign data in target functionalities and during operation processing.

| Function name in ConseilJS-softsigner | <code>signText</code> | <code>signOperation</code> |
|---------------------------------------|---|---|
| High-level algorithm | <code>sign(encode(data, 'utf8'))</code> | <code>sign(blake_hash(data, 32))</code> |
| Where used | Sign & Verify, dApp dialog boxes | ConseilJS library |

Figure 5.1: Comparison of signature algorithms.

As long as users can "manually" sign data that could be a hash, there is a risk of transaction fraud.

```
public async signText(message: string): Promise<string> {
  const messageSig = await CryptoUtils.signDetached(Buffer.from(message, 'utf8'), await
this.getKey());

  return TezosMessageUtils.readSignatureWithHint(messageSig, 'edsig');
}

/* redacted */

public async signTextHash(message: string): Promise<string> {
  const messageHash = TezosMessageUtils.simpleHash(Buffer.from(message, 'utf8'), 32);
  const messageSig = await CryptoUtils.signDetached(messageHash, await this.getKey());

  return TezosMessageUtils.readSignatureWithHint(messageSig, 'edsig');
}
```

Figure 5.2: [ConseilJS-softsigner/src/utis/CryptoUtils.ts lines 90 - 94 and 102 - 107](#)

Exploit Scenario

An attacker tricks the user to copy-paste and sign a specially crafted message which is a BLAKE hash of a transaction under the pretext of verification of ownership of the account. The victim sends the signature to the attacker who uses it to transfer away the funds.

Recommendation

Short term, limit the Sign & Verify dialog box to accept only data which is not a BLAKE hash. For example, prefix the data with a constant string longer than BLAKE hash length.

Long term, make sure that it is not possible to sign hashes of operations with `signText` function, unless it's an intended functionality. Add warnings for potentially dangerous functionalities. Also, make sure that data provided to `signOperation` is not coming from a potentially untrusted source, otherwise, if required, prefix such data with a fixed string which is not a valid transaction prefix. Less preferably, detect if the data starts with suspicious bytes and reject it.

References

- [Developing Secure Ledger Apps - Avoid blindly signing data](#)
- [Related GitLab issue](#)

6. Wallet file permissions

Severity: Low

Type: Access Controls

Target: T2/src/utls/wallet.ts

Difficulty: High

Finding ID: TOB-TEZORI-006

Description

Wallets are created with default file-access permissions (0666 & ~umask). On some systems this may result in creation of a world-readable file.

```
fs.writeFile(filename, JSON.stringify(encryptedWallet), (err) => {  
  if (err) {  
    reject(err);  
    return;  
  }  
  resolve();  
});
```

Figure 6.1: [T2/src/utls/wallet.ts lines 103-109](#)

Exploit Scenario

An attacker gets access to the user computer as a low-privileged identity and is able to steal the encrypted wallet. The attacker attempts to crack the wallet's password based on additional knowledge obtained about the victim.

Recommendation

Short term, explicitly set restrictive permissions for newly created wallet files, and check file permissions for already existing wallets upon opening.

Long term, always explicitly set file permissions for sensitive data to be as restrictive as possible.

7. Ignored exceptions

Severity: Informational
Type: Error Reporting
Target: T2

Difficulty: High
Finding ID: TOB-TEZORI-007

Description

The codebase has some places which catch all exception types without further action, effectively silencing them. An example is presented in Figure 7.1. This practice can be found in multiple files and appears to be a systemic issue.

```
newTransactions = newTransactions.map(transaction => {  
  const params = transaction.parameters.replace(/\\s/g, '');  
  if (transferPattern.test(params)) {  
    try {  
      /* ...redacted... */  
    } catch (e) {  
      /* */  
    }  
  } else if (mintPattern.test(params)) {  
    try {  
      /* ...redacted... */  
    } catch (e) {  
      /* */  
    }  
  } else if (burnPattern.test(params)) {  
    try {  
      /* ...redacted... */  
    } catch (e) {  
      /* */  
    }  
  } else {  
    // TODO  
  }  
});
```

Figure 7.1: [T2/src/contracts/TokenContract/util.ts lines 21-70](#)

Exploit Scenario

An exception is thrown because of an error which corrupted the application state. The exception handler ignores the error, and the program continues as if nothing happened. The code now operates in an invalid state which may lead to different security issues.

Recommendation

Short term, identify all the places where exceptions are ignored and add error handling code or remove the catch block to let the application crash. If the latter happens, generate a crash report with instructions for volunteering users to send to developers.

Long term, add a note to wallet development guidelines to either handle the exceptions correctly or let them propagate through the call stack.

8. Users can be tricked to blindly sign transactions

Severity: High

Type: Data Validation

Target: T2/src/featureModals/Auth/Auth.tsx

Difficulty: Medium

Finding ID: TOB-TEZORI-008

Description

The wallet implements a custom `galleon://...` URI handler which opens the “dApp Authentication Request” dialog (see [TOB-TEZORI-001](#)). This functionality could be exploited to send signatures of attacker-controlled messages to attacker-controlled URL addresses (Figure 8.1). In the dialog (Figure 8.2), after pressing the “Authenticate” button, the `req.prompt` parameter value is signed (similar to [TOB-TEZORI-005](#)) and sent to the `req.callback` address. Both values came from the URI handler controlled by a third party.

```
const response = await  
fetch(`${req.callback}&sig=${Buffer.from(signature).toString('base64')}`);
```

Figure 8.1: [T2/src/featureModals/Auth/Auth.tsx line 78](#)

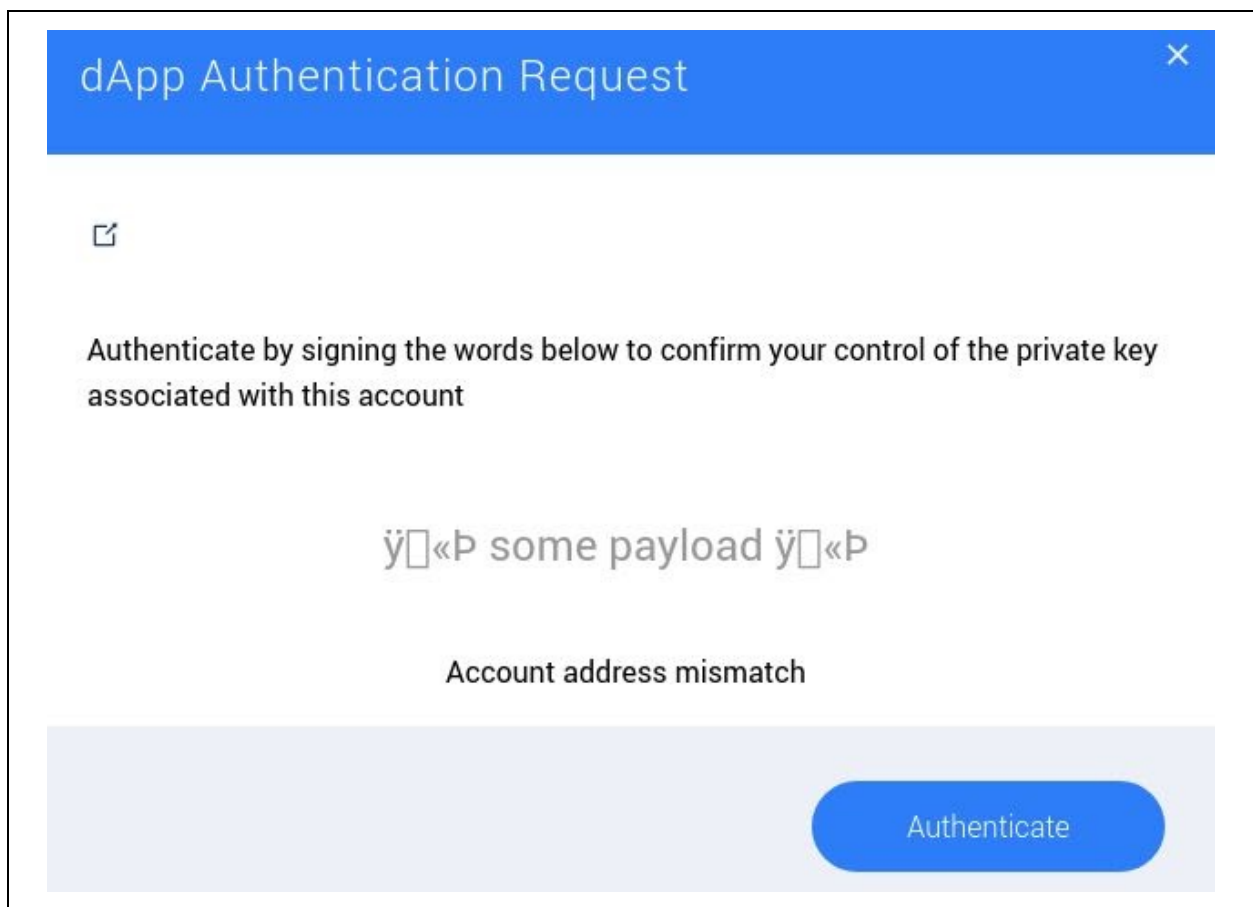


Figure 8.2: Dialog with an attacker-chosen payload to be signed.

Exploit Scenario

An attacker prepares a malicious dApp pretending to be a legitimate one. A wallet user is tricked into opening a malicious `galleon://...` URI provided by the attacker for authentication. The user is convinced that the link comes from a safe dApp and doesn't detect the phishing. Upon clicking the "Authenticate" button, the message provided by the attacker is signed and sent back to the attacker who uses it to transfer away the victim's funds.

Recommendation

Short term, limit the dApp Authentication Request dialog box to accept only data which won't lead to compromising the user's account. For example, prefix the data with a constant string of length longer than BLAKE hash output.

Long term, make sure that it is not possible to sign hashes of operations with `signText` function, unless it's an intended functionality. Add warnings for potentially dangerous functionalities. Also, make sure that data provided to `signOperation` is not coming from a potentially untrusted source, otherwise, if required, prefix such data with a fixed string which is not a valid transaction prefix. Less preferably, detect if the data starts with suspicious bytes and reject it.

9. Client-side request forgery through dApp authentication

Severity: Medium

Type: Data Validation

Target: T2/src/featureModals/Auth/Auth.tsx

Difficulty: High

Finding ID: TOB-TEZORI-009

Description

The same functionality as described in [TOB-TEZORI-008](#) can be used to perform HTTP request forgery attacks against local services which are running on wallet user's private networks. The called URI is almost fully controlled by a third party.

Exploit Scenario

A malicious dApp generates a `galleon://...` URI which forces the wallet to perform an HTTP GET request to a server which is bound to a local network. The wallet user is not informed what the destination address is and does not detect a phishing attack. The local server exposes a development HTTP API which allows an attacker to perform code execution.

Recommendation

Short term, display `req.callback` value somewhere in the application, inform the user that the signature will be sent to this address, and ask him to validate it before clicking on the "Authenticate" button.

Long term, ensure that the user is able to verify addresses that come from untrusted sources before performing requests to them. If possible, white-list available destinations. Otherwise black-list them, disallowing addresses that are part of local (private) network ranges.

References

- [OWASP Cheat Sheet Series - Server Side Request Forgery Prevention](#)

10. User interface bugs

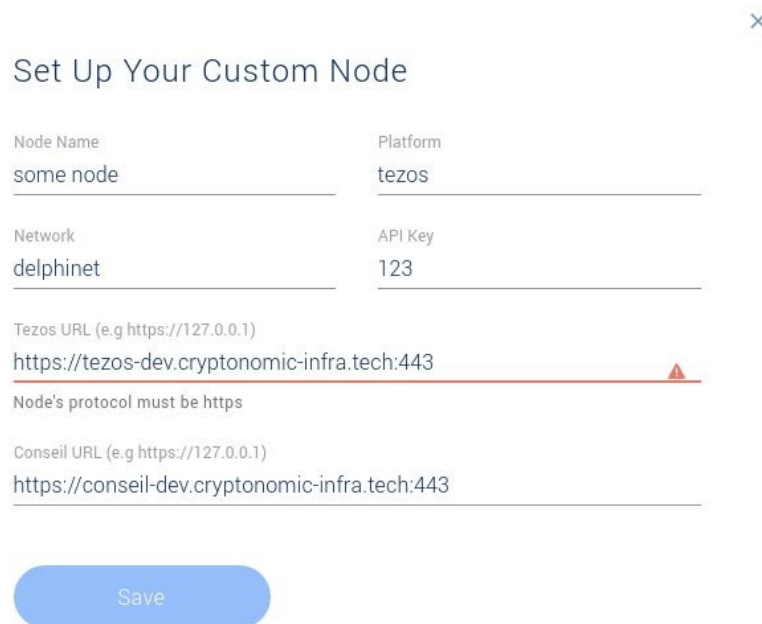
Severity: Informational
Type: Undefined Behavior
Target: T2

Difficulty: High
Finding ID: TOB-TEZORI-010

Description

There are a few issues that do not pose direct risk to users but may be used to leverage other vulnerabilities. They also impact usability of the wallet:

- There is no way to review existing custom node settings from the application. Users can't easily validate correctness of the settings.
- Two or more custom nodes with the same name can be created.
- When creating a new custom node, if an invalid URL is provided and a Save button is clicked, then the "Node's protocol must be https" error is shown and the button is disabled. Even after fixing the url, the button remains disabled until the application is reloaded.



The screenshot shows a dialog box titled "Set Up Your Custom Node" with a close button (X) in the top right corner. The dialog contains several input fields:

- Node Name:** some node
- Platform:** tezos
- Network:** delphinet
- API Key:** 123
- Tezos URL (e.g https://127.0.0.1):** https://tezos-dev.cryptonomic-infra.tech:443
- Conseil URL (e.g https://127.0.0.1):** https://conseil-dev.cryptonomic-infra.tech:443

A red error message is displayed below the Tezos URL field: "Node's protocol must be https". A red triangle icon is next to the error message. At the bottom of the dialog is a blue "Save" button.

Figure 10.1: Persistent protocol error.

- When creating a new wallet, the `.tezwallet` extension is not automatically appended. If a user creates the wallet without explicitly typing the extension, he won't be then able to open it, because it won't be shown in the open-file prompt. This issue appears only in Linux operating systems (Ubuntu 20.04.1).
- When creating a new wallet that overwrites an existing one, only the operating system warning is shown to the user. Overwriting the wallet could lead to data loss. it may be valuable to implement a warning dialog box in the application itself, rather than relying on the OS warning.

- In the Send tab, the Amount input box doesn't limit the number of characters that are displayed. Users can paste enormously long numbers, which overflow the confirmation dialog box.

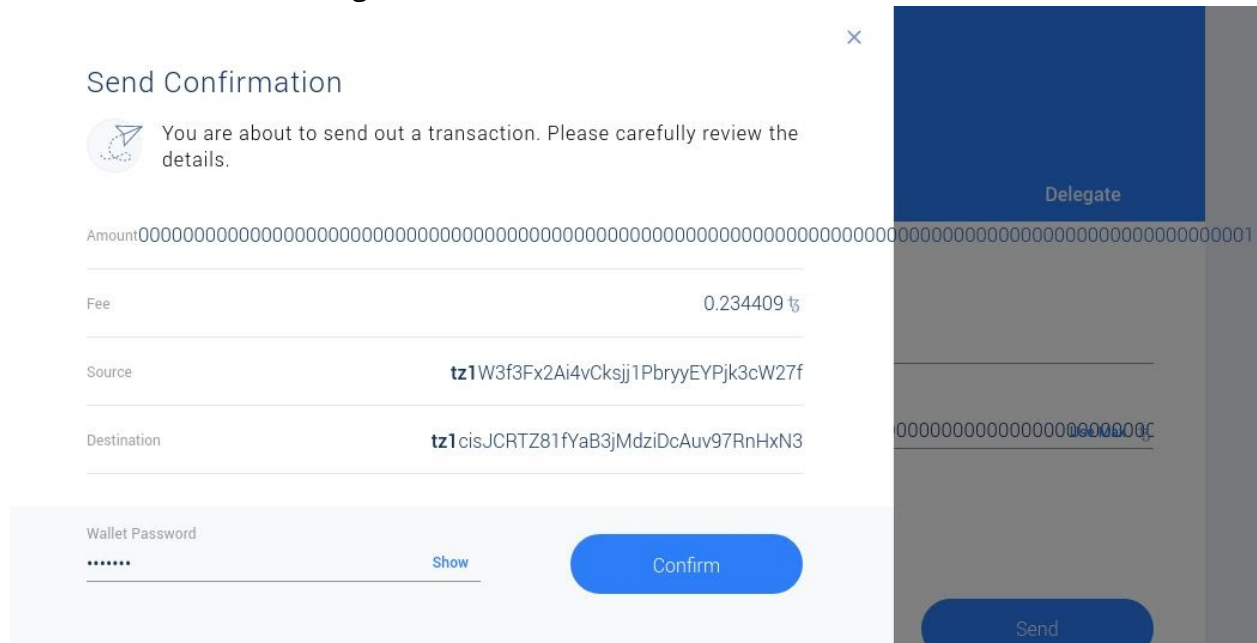


Figure 10.2: Overflow in Amount field.

- When typing a recipient address in the Send tab, a GET request is performed before local validation, causing redundant network traffic. That is, in `./src/contracts/ImplicitAccount/components/Send.tsx` file, `onChange` event triggers `handleToAddressChange` function which calls `getIsImplicitAndEmptyThunk`. Moreover, the address is not escaped (see [TOB-TEZORI-014](#)).



Figure 10.3: Sending an HTTP request before validation.

Recommendation

Short term, fix the mentioned UI issues.

Long term, perform extensive UI and UX tests before each release.

•

11. Wallet's password not cleared from a dialog box

Severity: Low
Type: Data Exposure
Target: T2

Difficulty: High
Finding ID: TOB-TEZORI-011

Description

The wallet password is not cleared from the Send Confirmation dialog box after the box is closed (with Confirm button or X button). It's likely cached in `SendConfirmationModal` local state. (Figure 17.1)

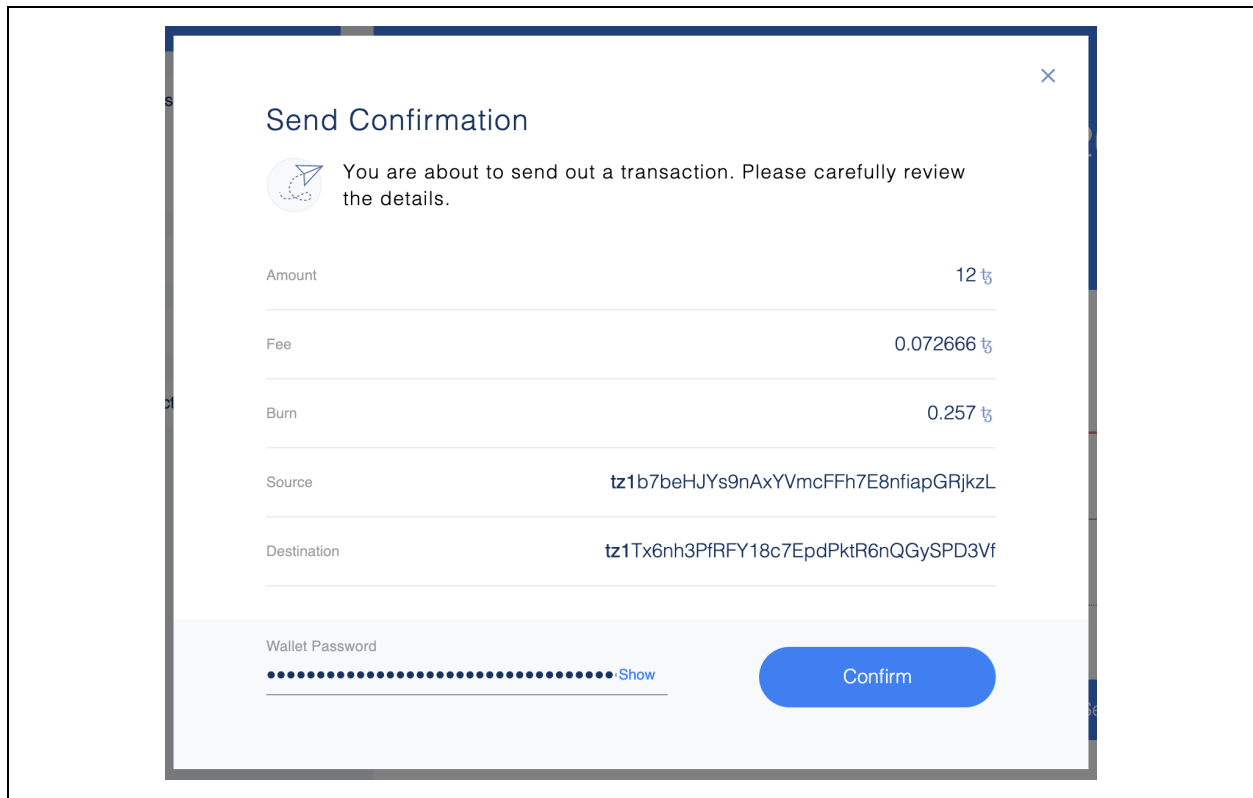


Figure 11.1: "Send Confirmation" dialog with cached password

Exploit Scenario

A wallet user performs a transaction and the wallet's password remains visible in the UI. The user is unaware of this and walks away from the computer. An attacker can now reveal the password and steal it.

Recommendation

Short term, clear password input box when `SendConfirmationModal` is closed.

Long term, ensure that sensitive data is not cached and does not persist during UI navigation if not necessary.

12. Operations can be injected by a Tezos node before signing

Severity: High

Difficulty: Medium

Type: Data Validation

Finding ID: TOB-TEZORI-012

Target: `ConseilJS/src/chain/tezos/TezosNodeWriter.ts`

Description

A compromised or malicious Tezos node can imperceptibly inject operations which will be signed along with other operations. The value of `blockHead.hash` received from the remote node is not validated and is used to construct operations which are about to be signed.

The data fetched by the `TezosNodeReader.getBlockAtOffset` function is assigned to the `blockHead` variable (Figure 12.1). The `blockHead.hash` value, fully controlled by the Tezos node, is passed to `forgeOperations` function (Figure 12.2). There, it's base58 decoded and prepended to the users' operations (Figure 12.3). The whole `forgedOperationGroup` string representing a sequence of operations is then signed.

```
export async function sendOperation(server: string, operations:
TezosP2PMessageTypes.Operation[], signer: Signer, offset: number = 54):
Promise<TezosTypes.OperationResult> {
  const blockHead = await TezosNodeReader.getBlockAtOffset(server, offset);
  const forgedOperationGroup = forgeOperations(blockHead.hash, operations);

  const opSignature = await
signer.signOperation(Buffer.from(TezosConstants.OperationGroupWatermark +
forgedOperationGroup, 'hex'));
```

Figure 12.1: [ConseilJS/src/chain/tezos/TezosNodeWriter.ts lines 172 - 174](#)

```
export function forgeOperations(branch: string, operations:
TezosP2PMessageTypes.Operation[]): string {
  log.debug('TezosNodeWriter.forgeOperations:');
  log.debug(JSON.stringify(operations));
  let encoded = TezosMessageUtils.writeBranch(branch);
  operations.forEach(m => encoded += TezosMessageCodec.encodeOperation(m));

  return encoded;
}
```

Figure 12.2: [ConseilJS/src/chain/tezos/TezosNodeWriter.ts lines 52 - 59](#)

```
export function writeBranch(branch: string): string {
  return base58check.decode(branch).slice(2).toString("hex");
}
```

Figure 12.3: [ConseilJS/src/chain/tezos/TezosMessageUtil.ts lines 234 - 236](#)

Because `blockHead.hash` is not limited to the expected length, the excess of it will be treated as encoded data, possibly an operation.

Exploit Scenario

A wallet user requests the wallet perform an operation. The wallet connects to a malicious Tezos node which injects an operation into the `blockHead.hash` value. The user is not aware of injection and signs the whole operation sequence, including the unwanted injected operation.

Recommendation

Short term, validate `blockHead.hash` parameter after obtaining it from a remote node. Right before signing, when operations will not be further manipulated, decode the data which is about to be signed and display it clearly to the user who will be able to catch a potential injection.

Long term, validate all data that comes from untrusted sources such as a Tezos node or Conseil node, before using it.

References

- [P2P message format - operation](#)

13. Entrypoint not validated, possible injection of data to sign

Severity: High

Difficulty: High

Type: Data Validation

Finding ID: TOB-TEZORI-013

Target: `ConseilJS/src/chain/tezos/TezosMessageCodec.ts`

Description

In function `encodeTransaction`, the length of `transaction.parameters.entrypoint` is not validated which may lead to injection of excessive data into encoded operations. An attacker controlling the vulnerable parameter, that is the smart contract's function name, may add new operations that the user will sign.

The issue arises from the fact that `entrypoint` length is encoded as 1 byte. If it's longer than 255 bytes, the length will be truncated in the encoding process. However, the whole `entrypoint` will be appended to the result. Code with the bug is presented in Figure 13.1.

```
hex += 'ff'  
+ ('0' + composite.entrypoint.length.toString(16)).slice(-2)  
+ composite.entrypoint.split('').map(c => c.charCodeAt(0).toString(16)).join('');
```

Figure 13.1: [ConseilJS/src/chain/tezos/TezosMessageCodec.ts lines 440 - 442](#)

Note that there is a comment in the function's docstring warning about the expected `entrypoint` format, but it should be explicitly validated in the code.

Exploit Scenario

An attacker tricks a wallet user to copy-paste specially crafted data into the `Entry Point` text box in the `Invoke` tab. The user thinks he will only call a smart contract's function, but he will also perform a transaction.

Recommendation

Short term, validate the `entrypoint` length before encoding.

Long term, apply validation to all arguments that may come from untrusted input. Enforce data formatting that complies with documentation. Take special care of data length and allowed bytes - for example Michelson strings are restricted to a printable subset of 7-bit ASCII.

References

- [Protocol 005 - entrypoint](#)
- [P2P message format - named entrypoint](#)
- [Michelson constants - strings](#)
- [How to find vulnerabilities? - 3. Documentation research](#)

14. URL components not encoded

Severity: Informational
Type: Data Validation
Target: ConseilJS

Difficulty: High
Finding ID: TOB-TEZORI-014

Description

In a few places, URL addresses are constructed from arguments that have not been properly sanitized. This may result in broken functionality or be leveraged as a part of an attack.

```
export async function executeEntityQuery(serverInfo: ConseilServerInfo, platform: string,
network: string, entity: string, query: ConseilQuery): Promise<any[]> {
  const url = `${serverInfo.url}/v2/data/${platform}/${network}/${entity}`;
  log.debug(`ConseilDataClient.executeEntityQuery request: ${url},
${JSON.stringify(query)}`);

  return fetch(url, {
    method: 'post',
    headers: { 'apiKey': serverInfo.apiKey, 'Content-Type': 'application/json' },
    body: JSON.stringify(query)
  })
}
```

Figure 14.1: [ConseilJS/src/reporting/ConseilDataClient.ts lines 22 - 30](#)

This issue can be also found in the following files:

- ConseilJS/src/reporting/ConseilDataClient.ts
- ConseilJS/src/reporting/ConseilMetadataClient.ts
- ConseilJS/src/chain/tezos/TezosNodeReader.ts
- ConseilJS/src/chain/tezos/TezosNodeWriter.ts

Recommendation

Short term, properly encode, filter, or validate arguments before using them as part of an URL.

Long term, avoid creating “stringly typed” APIs when data is not an arbitrary string. Introduce new data structures which will validate the data upon construction.

15. Arguments to contract's parameters not encoded

Severity: Informational
Type: Data Validation
Target: ConseilJS

Difficulty: High
Finding ID: TOB-TEZORI-015

Description

The parameters string is created based on unsanitized inputs to the transferBalance function. The source and destination values can contain arbitrary characters as round brackets and double quotes. This can be used to perform an injection attack which alters the semantics of the data structure described by parameters (Figure 15.1).

```
const parameters = `(Pair "${source}" (Pair "${destination}" ${amount}))`;
```

Figure 15.1: [ConseilJS/src/chain/tezos/contracts/TzbtTokenHelper.ts line 88](#)

Exploit Scenario

A developer uses the transferBalance function passing a source or destination value without any validation. The value can be controlled by an attacker influencing the function's behavior.

Recommendation

Short term, properly sanitize data before using it in Michelson context.

Long term, avoid creating "stringly typed" APIs when data is not an arbitrary string. Introduce new data structures which will validate the data upon construction.

16. JSONPath argument is not escaped

Severity: Informational
Type: Data Validation
Target: ConseilJS

Difficulty: High
Finding ID: TOB-TEZORI-016

Description

The operationGroupId argument is interpolated into the path argument passed to the JSONPath function without sanitization (Figure 16.1).

```
export async function getMempoolOperation(server: string, operationGroupId: string, chainid: string = 'main') {
  const mempoolContent: any = await performGetRequest(server,
    `chains/${chainid}/mempool/pending_operations`).catch(() => undefined);
  const jsonresult = JSONPath({ path: `$.applied[?(@.hash=='${operationGroupId}')]`, json: mempoolContent });

  return jsonresult[0];
}
```

Figure 16.1: [ConseilJS/src/chain/tezos/contracts/TezosNodeReader.ts lines 184 - 189](#)

operationGroupId may contain code that will be passed to the eval function by JSONPath, potentially leading to code execution. At the time of writing, no known exploit for JSONPath exists, but it may be found in the future.

Exploit Scenario

A developer uses the getMempoolOperation function passing an operationGroupId value without any validation. The value can be controlled by an attacker, thus leading to code execution.

Recommendation

Short term, sanitize the operationGroupId value before passing it to the JSONPath function. Enable the preventEval option.

Long term, avoid creating “stringly typed” APIs when data is not an arbitrary string. Introduce new data structures which will validate the data upon construction.

References

- [JSONPath properties - preventEval](#)

17. Discrepancies between master branch and latest release

Severity: Medium

Type: Patching

Target: T2

Difficulty: Medium

Finding ID: TOB-TEZORI-017

Description

The master branch of T2 is not regularly maintained for dependency issues and is behind the most recent release by several months. For an unwitting user wishing to develop the wallet, this could pose a threat. While this version of the product is not in scope, Trail of Bits considered it noteworthy due to the fact that dependency auditing revealed instances of outdated npm modules at varying severity levels.

Exploit Scenario

A user builds his own production version of T2 wallet from the master branch. Unknown to the user, his version is based on old code which contains vulnerabilities. An attacker may now use bugs learned from commit history or fixed issues to harm the user.

Recommendation

Short term, warn users that they should not build the wallet from the master branch. Provide clear information about which branch/commit/tag is the newest/recommended one. Moreover, inform users abouts which version Galleon is currently using.

Long term, design, implement, and describe project release policy, versioning, and purpose of GitHub branches and tags for all repositories concerned.

A. Vulnerability Classifications

| Vulnerability Classes | |
|-----------------------|--|
| Class | Description |
| Access Controls | Related to authorization of users and assessment of rights |
| Auditing and Logging | Related to auditing of actions or logging of problems |
| Authentication | Related to the identification of users |
| Configuration | Related to security configurations of servers, devices or software |
| Cryptography | Related to protecting the privacy or integrity of data |
| Data Exposure | Related to unintended exposure of sensitive information |
| Data Validation | Related to improper reliance on the structure or values of data |
| Denial of Service | Related to causing system failure |
| Error Reporting | Related to the reporting of error conditions in a secure fashion |
| Patching | Related to keeping software up to date |
| Session Management | Related to the identification of authenticated users |
| Timing | Related to race conditions, locking or order of operations |
| Undefined Behavior | Related to undefined behavior triggered by the program |

| Severity Categories | |
|---------------------|--|
| Severity | Description |
| Informational | The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth |
| Undetermined | The extent of the risk was not determined during this engagement |
| Low | The risk is relatively small or is not a risk the customer has indicated is important |
| Medium | Individual user's information is at risk, exploitation would be bad for client's reputation, moderate financial impact, possible legal |

| | |
|------|--|
| | implications for client |
| High | Large numbers of users, very bad for client's reputation, or serious legal or financial implications |

| Difficulty Levels | |
|-------------------|---|
| Difficulty | Description |
| Undetermined | The difficulty of exploit was not determined during this engagement |
| Low | Commonly exploited, public tools exist or can be scripted that exploit this flaw |
| Medium | Attackers must write an exploit, or need an in-depth knowledge of a complex system |
| High | The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue |

B. Code Quality Recommendations

The following recommendations are not associated with specific vulnerabilities. They enhance code readability and may prevent the introduction of vulnerabilities in the future.

ConseilJS-softsigner

- Magic value 32 is used in [ConseilJS-softsigner/src/KeyStoreUtils.ts:restoreIdentityFromMnemonic](#) function. It should be [crypto_sign_SEEDBYTES](#) constant.
- Magic value [33554432](#) is used as a RAM limit. Changing this value will break compatibility, so it should be moved to an explicit, named constant.
- [crypto_sign_seed_keypair](#) function is called with encoding set to an empty string, which may be misleading. Skip the argument or set it explicitly to uint8array.
- `CryptoUtils.checkSignature` and `KeyStoreUtils.checkTextSignature` return true if the signature is valid, same as [libsodium.js](#), but [sodium](#) returns 0 in such cases and -1 on failure. Document function return values explicitly in the documentation (docstring) to avoid mistakes.

ConseilJS-ledgersigner

- Wrong docstring documentation for [signOperation](#) function.

ConseilJS

- Link to the ConseilJS-core documentation in [README.md](#) not working.

C. Electron Application Hardening

There are a few configuration options available to improve the security of Electron applications that we recommend to consider for the Tezori wallet:

- Add Content Security Policy (CSP) feature to protect against potential XSS attacks.
- Disable creation of new windows implementing `setWindowOpenHandler` function.
- Disable middle-mouse button (Auxclick feature).
- Enable Context Isolation.
- Override default Session Permission Request Handler with one which disallows all permission requests.

References

- [Content Security Policy presence check and review](#)
- [Disable or limit creation of new windows](#)
- [AUX click - Limit navigation flows to untrusted origins](#)
- [Context Isolation](#)
- [Handle Session Permission Requests From Remote Content](#)
- [Electron Documentation - Security](#)
- [Electron Security Checklist](#)
- [Electronegativity Official Documentation](#)

D. Fix Log

Cryptonomic addressed the following issues in their codebase as a result of our assessment, and each of the fixes was verified by Trail of Bits.

| ID | Title | Severity | Status |
|----|---|---------------|-----------------|
| 01 | Remote code execution via openExternal | High | Fixed |
| 02 | Unencrypted secretes in memory | High | Not Fixed |
| 03 | Insecure private key in-memory encryption | High | Partially Fixed |
| 04 | Access to raw private key and signing without additional authentication | High | Partially Fixed |
| 05 | Signing valid operation hashes via UI dialog | High | Not Fixed |
| 06 | Wallet file permissions | Low | Partially Fixed |
| 07 | Ignored exceptions | Informational | Not Fixed |
| 08 | Users can be tricked to blindly sign transactions | High | Fixed |
| 09 | Client-side request forgery through dApp authentication | Medium | Not Fixed |
| 10 | User interface bugs | Informational | Not Fixed |
| 11 | Wallet's password not cleared from a dialog box | Medium | Not Fixed |
| 12 | Operations can be injected by a Tezos node before signing | High | Fixed |
| 13 | Entrypoint not validated, possible injection of data to sign | High | Not Fixed |
| 14 | URL components not encoded | Informational | Not Fixed |
| 15 | Arguments to contract's parameters not encoded | Informational | Not Fixed |
| 16 | JSONPath argument is not escaped | Informational | Not Fixed |
| 17 | Discrepancies between master branch and latest release | Medium | Fixed |

Detailed Fix Log

Finding 1: Remote code execution via openExternal

Fixed. Checks were added to ensure the URI starts with https://.

<https://github.com/Cryptonomic/T2/commit/c5fddc00a5799eb8094ff3b22765099a40>

<https://github.com/Cryptonomic/T2/commit/0de3a0b5666b172433e7c06da05f1cb72>

Finding 2: Unencrypted secretes in memory

Not fixed.

Finding 3: Insecure private key in-memory encryption

Partially fixed. The password is not stored along with the key anymore. However, the SoftSigner class is initialized with unencrypted key material in the createSigner function. The key could be loaded as a ciphertext from the wallet file reducing the key material exposure.

<https://github.com/Cryptonomic/ConseilJS-softsigner/commit/4a2fb55c6b855c2dec54fc>

Finding 4: Access to raw private key and signing without additional authentication

Partially fixed. The dialogs require typing a password to proceed however the dialog from Figure 4.1 stays opened indefinitely after the private key is revealed.

<https://github.com/Cryptonomic/T2/pull/299/files>

Finding 5: Signing valid operation hashes via UI dialog

Not fixed.

Finding 6: Wallet file permissions

Partially fixed. The file permissions are now set correctly on the wallet file creation however the wallet does not check existing wallet file permissions.

Finding 7: Ignored exceptions

Not fixed.

Finding 8: Users can be tricked to blindly sign transactions

Fixed. The signed payload is now restricted to words from the BIP39 set.

<https://github.com/Cryptonomic/T2/commit/96de1a9bd3dcfdd64cdb0cac185ae20bd>

Finding 9: Client-side request forgery through dApp authentication

Not fixed.

Finding 10: User interface bugs

Not fixed.

Finding 11: Wallet's password not cleared from a dialog box

Not fixed. The password is still accessible after a transfer is confirmed by opening another "Send Confirmation" dialog box.

Finding 12: Operations can be injected by a Tezos node before signing

Fixed. The `blockHead.hash` parameter's length is now validated.

<https://github.com/Cryptonomic/ConseilJS/commit/a02e0e5d2f414445af3892c0533>

Finding 13: Entrypoint not validated, possible injection of data to sign

Not fixed.

Finding 14: URL components not encoded

Not fixed.

Finding 15: Arguments to contract's parameters not encoded

Not fixed.

Finding 16: JSONPath argument is not escaped

Not fixed.

Finding 17: Discrepancies between master branch and latest release

Fixed. The T2 repository has now a branch called "trunk" which is the default and up to date.