

Audit-Report Gem Wallet Rust Wallet Core 09.2026

Cure53, Dr.-Ing. M. Heiderich, Dr. A. Pirker, Dr. N. Kobeissi, MSc. C. Mayr

Index

[Introduction](#)

[Scope](#)

[Identified Vulnerabilities](#)

[GEM-01-001 WP3: No authorization required on wallet deletion \(Medium\)](#)

[GEM-01-006 WP2: Swap permits signed with unrelated wallet \(Low\)](#)

[GEM-01-007 WP1: Wallet deletion removes secrets before DB commit \(Low\)](#)

[GEM-01-008 WP3: User authentication not configured by default \(Medium\)](#)

[GEM-01-010 WP3: Unreadable Android storage can overwrite password \(Low\)](#)

[GEM-01-012 WP3: Biometric authorization bypass via hooks \(Medium\)](#)

[GEM-01-013 WP3: Potential phishing via deep link hijacking \(Medium\)](#)

[GEM-01-014 WP3: DoS of Android app via malformed intents \(Low\)](#)

[GEM-01-016 WP3: Biometric authorization bypass via file manipulation \(Medium\)](#)

[GEM-01-017 WP1: Missing zeroization in mnemonic generation \(Low\)](#)

[GEM-01-020 WP2: Missing zeroization when deriving private keys \(Low\)](#)

[Miscellaneous Issues](#)

[GEM-01-002 WP2: No checks on entropy size for seed on deriving priv. keys \(Info\)](#)

[GEM-01-003 WP1: No minimum checks on Argon2id parameters \(Info\)](#)

[GEM-01-004 WP1: Missing expected type check on decrypting secrets \(Info\)](#)

[GEM-01-005 WP2: Signing keys lack binding to approved accounts \(Info\)](#)

[GEM-01-009 WP3: Concurrent iOS password creation can lock-out wallet \(Low\)](#)

[GEM-01-011 WP3: Android Keystore failure exposes encryption keys \(Medium\)](#)

[GEM-01-015 WP3: Android application supports outdated SDK versions \(Low\)](#)

[GEM-01-018 WP3: Missing screenshot protection for seedphrase export \(Low\)](#)

[GEM-01-019 WP2: Existing wallet check blocks secrets recovery \(Low\)](#)

[GEM-01-021 WP2: Secp256k1 malleability in recover_ethereum_address \(Info\)](#)

[Conclusions](#)

Introduction

"Gem Wallet is an open-source, self-custody crypto wallet that puts financial freedom into everyone's hands."

From <https://gemwallet.com/about/>

This report describes the results of a penetration test and source code audit conducted against the Gem Wallet (Rust core).

To give some context regarding the assignment's origination and composition, Gem Wallet LLC contacted Cure53 in July 2026. The test execution was scheduled for September 2026, namely in CW39. A total of thirteen days were invested to reach the coverage expected for this project, and a team of four senior testers was assigned to its preparation, execution, and finalization.

The methodology conformed to a white-box strategy, whereby assistive materials such as sources, as well as all further means of access required to complete the tests were provided to facilitate the undertakings.

The work was split into three separate work packages (WPs), defined as:

- **WP1:** White-box audits & reviews against the v4 keystore & UniFFI secret boundary
- **WP2:** White-box audits & reviews against the derivation, import & signing primitives
- **WP3:** White-box audits & reviews against the iOS/Android password custody

All preparations were completed in September 2026, specifically during CW38, to ensure a smooth start for Cure53. Communication throughout the test was conducted through a dedicated and shared Telegram group, established to combine the teams of Gem Wallet and Cure53. All personnel involved from both parties were invited to participate in this group. Communications were smooth, with few questions requiring clarification, and the scope was well-prepared and clear. No significant roadblocks were encountered during the test. Cure53 provided frequent status updates, shared its findings, and offered live reporting through the aforementioned Telegram group in the form of shared Markdown files.

The Cure53 team achieved good coverage over the scope items, and identified a total of twenty-one findings. Of the twenty-one security-related findings, eleven were classified as security vulnerabilities, and ten were categorized as general weaknesses with lower exploitation potential.

Overall, the results of this audit left a mixed impression on the Cure53 team. Positively, no issues of *High* or *Critical* severity were identified, which indicates strong awareness where protection against more severe threats is concerned.

Moreover, cryptographic primitives, parameter selection, and secure randomness across the Rust crates and mobile clients were all found to be solidly implemented.

However, it is advised that the testing team still observed several areas where further attention is required. Most notably, systemic authorization flaws were discovered - including missing access checks, authorization bypasses, and insecure default configurations. It is advised that these findings should be resolved as soon as possible, and that the development team should invest further time in properly strengthening the Gem Wallet core. While the findings made in this report are not of critical severity, it is still recommended that they should be fully addressed in a timely manner.

The report will now shed more light on the scope and testing setup, and will provide a comprehensive breakdown of the available materials. Following this, the report will list all findings identified in chronological order, starting with the *Identified Vulnerabilities* and followed by the *Miscellaneous Issues* unearthed. Each finding will be accompanied by a technical description, Proof-of-Concepts (PoCs) where applicable, and any fix or preventative advice to action.

In summation, the report will finalize with a *Conclusions* chapter in which the Cure53 team will elaborate on the impressions gained toward the general security posture of the Gem Wallet (Rust core).

Scope

- **Audits & reviews against the Gem Wallet (Rust core)**
 - **WP1:** White-box audits & reviews against the v4 keystore & UniFFI secret boundary
 - **Focus areas:**
 - Integrity of the v4 encrypted keystore format & authenticated metadata
 - Validation of key-derivation & encryption parameters
 - Enforcement of secret kinds & size limits
 - Determinism of key store identifiers & binding to wallets
 - On-disk secrets storage
 - Atomic writes
 - Permissions
 - Concurrency
 - Deletion
 - Recovery
 - Correct use of randomness
 - Key derivation
 - Hashing
 - Decrypted key material boundary
 - **Source code:**
 - URL:
 - <https://github.com/gemwalletcom/wallet/releases/tag/2.114.40>
 - Tag:
 - 2.114.40
 - **WP2:** White-box audits & reviews against the derivation, import & signing primitives
 - **Focus areas:**
 - Mnemonic normalization & account derivation
 - Population of accounts, addresses and public keys
 - Determinism of wallet-identifier derivation
 - Binding between wallet, chain, path, address & public key
 - Private key import & export codec, encoding & decoding, checksum & length validation
 - Zeroization
 - Correct use of EC libraries including
 - Key construction and validation
 - Signature & recovery identifier encoding
 - Secret lifetime
 - **Source code:**
 - URL:
 - <https://github.com/gemwalletcom/wallet/releases/tag/2.114.40>
 - Tag:
 - 2.114.40

- **WP3:** White-box audits & reviews against the iOS/Android password custody
 - **Focus areas:**
 - Storage, generation, retrieval and passes of keystore password including Keychain and encrypted-storage configuration
 - Access and authentication policy
 - Legacy password paths protecting current password
 - Password to byte conversion
 - Lifetime and zeroization of transient secrets
 - Selection of wallet and keystore identifiers
 - Prevention of unintended key exposure
 - Signing paths reviewed only for password handling and secret-boundary preservation
 - **Source code:**
 - URL:
 - <https://github.com/gemwalletcom/wallet/releases/tag/2.114.40>
 - Tag:
 - 2.114.40
 - Android:
 - *gem_wallet_universal_2.114.40.apk*
 - iOS:
 - *gem_wallet_2.114.40.ipa*
 - <https://testflight.apple.com/join/GUrGydJz>
- **Test-supporting material was shared with Cure53**
- **All relevant sources were shared with Cure53**

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., GEM-01-001) to facilitate any future follow-up correspondence.

GEM-01-001 WP3: No authorization required on wallet deletion (*Medium*)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While dynamically testing the Gem Wallet Android app, it was noted that the Gem Wallet app allows its users to enforce passcode checks on critical operations. This guards sensitive actions - e.g. displaying the seed phrase for a wallet - by requiring explicit authorization from the user. However, it was discovered that the deletion of a wallet fails to enforce such an authorization check.

It should be noted that deleting a wallet represents a critical business logic operation - effectively removing the wallet entirely from the Gem Wallet mobile app. An attacker who acquired access to the unlocked device of a victim with the Gem Wallet mobile app open could delete all of the victim's wallets from the device.

Steps to reproduce:

1. Create a new wallet in the Gem Wallet Android app.
2. Enforce passcode authorization in the settings.
3. Delete the wallet created in Step 1.

Similar steps also apply for the iOS app.

To mitigate this issue, Cure53 recommends enforcing explicit authorization on deleting a wallet, if the user has enabled passcode authorization in the app settings.

GEM-01-006 WP2: Swap permits signed with unrelated wallet (*Low*)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Source review identified that swap quoting and transfer preparation resolve the current wallet independently. A quote records Wallet A's address, but `get_transfer` reads the current wallet again. If this preference changes to Wallet B, `permit2_data` signs the quote's `Permit2` approval with B's key.

It neither requires an account on the source chain, nor compares the signing address with `quote.request.wallet_address`. Signing occurs when preparing the transfer, before the confirmation screen.

This can produce an approval from a wallet the user did not select. If A subsequently signs the swap transaction, then the router rejects the mismatched permit and the transaction reverts, costing gas. The signature is embedded locally in router calldata; direct theft was not demonstrated.

On iOS, changing wallets normally dismisses the swap sheet, restricting the race to the interval before the interface updates. Notification delivery cannot reliably target this interval.

The excerpt below shows where the wallet for the transfer is selected. It is taken from the current wallet of the session:

Affected file:

`wallet-2.114.40/core/gemstone/src/services/swap/quote.rs`

Affected code:

```
pub async fn get_transfer(&self, quote: Quote) -> Result<GemSwapTransfer, SwapperError> {  
    let wallet = self.session.current_wallet().await.map_err(|error|  
SwapperError::TransactionError(error.to_string()))?;  
    self.swap.get_transfer(wallet, quote).await  
}
```

The signing call in `permit2_data` uses this newly-resolved wallet from the parameter:

Affected file:

`wallet-2.114.40/core/gemstone/src/services/swap/mod.rs`

Affected code:

```
fn permit2_data(&self, wallet: &Wallet, quote: &Quote, approval:  
&swapper::Permit2ApprovalData) -> Result<Permit2Data, SwapperError> {  
    [...]  
    let signature = signer  
        .sign_with_keystore(self.keystore.clone(),  
keystore_id_for_wallet(wallet.id.id()), password)  
        .map_err(|error|  
SwapperError::TransactionError(error.to_string()))?;  
    [...]  
}
```

Steps to reproduce:

1. Configure a service test with two Ethereum wallets, A and B, and select A as the current wallet.
2. Obtain a Uniswap quote that requires a *Permit2* approval and retain the returned quote for A.
3. Set the current-wallet preference to B before calling `GemSwapQuoteService::get_transfer` with that quote.
4. Recover the signer from the generated EIP-712 permit and compare it with the quote's address. The permit identifies B, although the quote identifies A.

To mitigate this issue, it is recommended to resolve and retain the wallet identity for the entire swap. Before retrieving a password or signing, it is recommended to require an account on the source chain and compare its address with the quote's address using the chain's address-comparison rules. This expected address should be passed to the signing-boundary check described in [GEM-01-005](#), and transfer validation should be performed before creating the permit.

GEM-01-007 WP1: Wallet deletion removes secrets before DB commit (Low)

Source review identified that `GemWalletService::delete_wallet` destroys the wallet's secret before deleting its database row. Keystore deletion synchronizes the directory before returning, whereas the subsequent store callback can fail. An I/O error, cancellation, or process termination between these operations therefore leaves a visible wallet where the secret has already been durably removed.

The affected wallet can no longer sign or reveal its recovery phrase. Without an external copy of the secret, spend authority cannot be recovered. A user with the phrase must delete the remaining wallet entry before importing it again, because ordinary re-import is blocked by [GEM-01-019](#).

Affected file:

`wallet-2.114.40/core/gemstone/src/services/wallet/mod.rs`

Affected code:

```
pub async fn delete_wallet(&self, wallet_id: WalletId) ->
Result<GemWalletDeletion, GemServiceError> {
    [...]
    if wallet.wallet_type != WalletType::View {
        self.keystore.delete_wallet_secrets(wallet.id.id(),
rules::legacy_keystore_id(&wallet));
    }
    self.store.delete_wallet(wallet.id.clone()).await?;
    [...]
}
```

Steps to reproduce:

1. Import a test wallet and verify that its database row and v4 keystore file both exist.
2. Configure the `GemWalletStore::delete_wallet` callback to return a storage error without deleting the row.
3. Invoke `GemWalletService::delete_wallet` and allow the preceding keystore deletion to complete.
4. Verify that the operation returns an error, the row remains, and the keystore file is absent. Attempt signing or phrase export to observe the missing-secret error.

The current ordering is deliberate: `docs/KEYSTORE_V4.md` prioritizes removing every secret copy before metadata so that secrets are not orphaned. Changing the order therefore requires a corresponding recovery mechanism.

To mitigate this issue, it is recommended to persist deletion intent and commit metadata removal before destroying the secret, with idempotent cleanup that resumes after failure. If orphan reconciliation is used, then it is advised to ship it with the ordering change and to refuse destructive cleanup when the database is empty, unreadable, or newly recreated. It is also advised to update the documented deletion contract in the same change, test failures at each boundary, and retain the shared password while other wallets depend on it.

GEM-01-008 WP3: User authentication not configured by default (Medium)

Fix note: *This issue has been mitigated by the Gem Wallet team and verified by Cure53 to be working as expected.*

During dynamic testing, it was found that user authentication via biometrics / passcode is not enabled by default on either the Android or iOS platform. This means that anyone who obtains the device while it is unlocked is able to open the app, view and sign transactions, and export the wallet seed phrase without any further authentication via the user interface.

This is because the operating system's keychain / keystore item used to decrypt the wallet keystore is accessible without additional access control flags (`SecAccessControl-CreateFlags`), which allows the application to export the key material without further user interaction.

An attacker would still have to obtain access to the device in an unlocked state. However, this could occur in everyday situations - e.g., when lending the phone to someone to make a phone call or to show them photographs - as well as in exceptional situations - such as a thief taking hold of the device in an unlocked state.

In each of the situations above, the current holder of the device would be able to open the Gem Wallet app in its default configuration and export the seed phrase without additional authentication. Therefore, the confidentiality requirements of the seed phrase outweigh the usability advantage of not having to authenticate with biometrics or enter the PIN when opening the wallet.

Steps to reproduce:

1. Start from a fresh install of the Gem Wallet application on either an Android or iOS build.
2. Create a new wallet.
3. Observe that throughout the onboarding process, no biometric or passcode authentication is configured.
4. Close the app and observe that from an unlocked phone, a user can open the app, sign transactions, and export the seed phrase, without having to re-authenticate.

Cure53 recommends that the default during the onboarding process should be for the user to provide biometric authentication and / or passcode or pin authentication. This would secure the confidentiality of the key material from other users accessing the unlocked device.

GEM-01-010 WP3: Unreadable Android storage can overwrite password (Low)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Source review identified that Android treats a missing preference value as permission to generate a shared keystore password. The Android 14 *SharedPreferences* implementation also returns an empty map for certain read, permission, and XML parsing failures. *TinkEncryptedKeyValueStore.getString* returns *null* in this state before attempting authenticated decryption, and *getOrCreatePassword* generates a replacement.

The startup migration requests password creation whenever at least one legacy wallet password remains. Wallets already using the shared password are excluded from this migration. If the value store is temporarily unreadable and a replacement is successfully committed, then those wallets remain encrypted under the previous password and become inaccessible. This requires a storage failure and a mixed migration state; an ordinary process termination after a successful commit does not establish the precondition. Recovery may require an external copy of each affected secret.

Affected file:

*wallet-2.114.40/android/app/src/main/kotlin/com/gemwallet/android/data/password/
TinkEncryptedKeyValueStore.kt*

Affected code:

```
override fun getString(key: String): String? {  
    val encryptedValue = sharedPreferences.getString(storageKey(key), null)  
    ?: return null  
    val decryptedValue =  
    aeadProvider.get().decrypt(Base64.getDecoder().decode(encryptedValue),  
    associatedData(key))  
    return String(decryptedValue, UTF_8)  
}
```

Password creation then accepts the null result, as shown below:

Affected file:

*wallet-2.114.40/android/app/src/main/kotlin/com/gemwallet/android/data/password/
TinkPasswordStore.kt*

Affected code:

```
override fun getOrCreatePassword(key: String): String {
    encryptedStore.getOrCreateMigrate(legacyStore, key)?.let { return it }
    val password = ByteArray(32)
    return try {
        random.nextBytes(password)
        val value = password.hex.append0x()
        encryptedStore.putString(key, value)
        legacyStore.removeString(key)
        value
    } finally {
        password.fill(0)
    }
}
```

Steps to reproduce:

1. Prepare a test installation with Wallet A encrypted under the shared password and Wallet B retaining a legacy per-wallet password.
2. Stop the process and inject a transient read failure for *shared_prefs/gem_wallet_passwords.xml* during its next load, while leaving the keyset, legacy store, and subsequent writes available.
3. Launch the application, and allow *migrateToSharedPassword* to call *get_password(true)*.
4. Verify that a new shared password is persisted and no longer decrypts A's unchanged keystore file.

In order to mitigate this issue, Cure53 recommends:

- Distinguishing verified absence from unreadable storage before allowing password creation.
- Not relying solely on a marker inside the same preferences file, since a failed load also hides that marker.
- Requiring independent evidence of first initialization or a validated migration in which all existing secrets remain recoverable.
- Surfacing password store failures at startup.
- Deploying this recovery handling before adding device-unlock requirements to the password key, so that newly-introduced access failures cannot trigger replacement.

GEM-01-012 WP3: Biometric authorization bypass via hooks (*Medium*)

Fix note: *This issue has been mitigated by the Gem Wallet team and verified by Cure53 to be working as expected.*

The Gem Wallet app contains a setting to enforce passcode authorization for sensitive and cryptographically-critical operations - such as transaction signing or displaying the seed phrase of a wallet. To this end, the Android mobile app uses Android's built-in passcode authorization scheme. It was discovered that this passcode authorization is optional, and hence can be bypassed on a rooted phone using Frida¹.

This would enable an attacker who gained access to a victim's rooted phone, to disable all passcode checks throughout the entire app. For this purpose, the attacker would attach to the running Gem Wallet process using Frida, and hook the authorization call implemented in the *UserConfig* class's *authRequired* function. When opening the app, the biometric authorization would be bypassed, and the same would be true when displaying the seed phrases of wallets.

Since the mobile apps implement root detection, it was assumed that rooted devices were within the threat model of this engagement. As a consequence, it was decided to report this issue as a vulnerability, instead of a miscellaneous issue.

The excerpt below demonstrates the Frida hook used to bypass the authorization dialog. It hooks the *b* function of the *sha* class. It should be noted that this corresponds to the *authRequired* function of the *UserConfig* class prior to obfuscation. Every time the app invokes this function, the hook returns *false*, effectively disabling all of the app's biometric authorization checks.

Frida hook (*gem_wallet_hook.js*):

```
Java.perform(function () {  
    var U = Java.use("sha");  
    U.b.implementation = function () {  
        var real = this.b();  
        console.log("[gem] sha.b() (authRequired?) real=" + real + " ->  
forcing false");  
        return false;  
    };  
    console.log("[gem] sha.b hooked");  
});
```

¹ <https://frida.re/>

Steps to reproduce:

1. Connect the mobile device to a computer that has Frida installed.
2. On a rooted Android phone, start the Frida server.
3. Start the Gem Wallet mobile app.
4. Enable passcode in the mobile app's settings.
5. Attach to the running Gem process using the command shown below (inside the `venv` with Frida installed):

Command:

```
$ frida -U -n Gem -l gem_wallet_hook.js
```

6. In the Gem Wallet mobile app, navigate to one of the wallets created in the app, and display the seed phrase. Observe that no passcode authorization dialog is shown, and that the seed phrase can be displayed. The bypass is also evident from the output shown below:

Output:

```
$ frida -U -n Gem -l gem_wallet_hook.js
[...]
. . . . . Connected to Mi Note 10 Lite (id=20a2fd4)
Attaching...
[gem] sha.b hooked
[Mi Note 10 Lite::Gem ]-> [gem] sha.b() (authRequired?) real=true ->
forcing false
[gem] sha.b() (authRequired?) real=true -> forcing false
```

The excerpt below shows the `authRequired` function of the `UserConfig` class. This function is used by nearly all authorization checks, and hence, can be instrumented and hooked using Frida:

Affected file:

`wallet-2.114.40/android/data/services/gemstone/src/main/kotlin/com/gemwallet/android/data/services/gemstone/config/UserConfig.kt`

Affected code:

```
fun authRequired(): Boolean =
    secureStore.get(SecureKey.Auth.string)?.toBooleanStrictOrNull() ?:
    configStore.getBoolean(ConfigKey.Auth.string)
```

To mitigate this vulnerability, Cure53 recommends protecting all sensitive, cryptographically-relevant items using the `setUserAuthenticationRequired`² function, together with the `setUserAuthenticationParameters`³ of the `KeyGenParameterSpec.Builder` class, set to `STRONG`.

² [https://developer.android.com/reference/androidx/security/crypto/KeyGenParameterSpec.Builder#setUserAuthenticationRequired\(boolean\)](https://developer.android.com/reference/androidx/security/crypto/KeyGenParameterSpec.Builder#setUserAuthenticationRequired(boolean))

³ [https://developer.android.com/reference/androidx/security/crypto/KeyGenParameterSpec.Builder#setUserAuthenticationParameters\(int,%20int\)](https://developer.android.com/reference/androidx/security/crypto/KeyGenParameterSpec.Builder#setUserAuthenticationParameters(int,%20int))

This will enforce biometric authorization every time the app accesses key material persisted in the TEE of the Android device, and the device will throw an exception if authorization cannot be granted.

Furthermore, it is advised that passcode authorization should be mandatory, and not optional. To also mitigate simple authorization checks not involving cryptographic keys, Cure53 recommends utilizing hardware-backed security, by adopting Android's crypto object when unlocking or authorizing the app⁴.

GEM-01-013 WP3: Potential phishing via deep link hijacking (Medium)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

The Gem Wallet app exports a single activity - namely the *MainActivity*. This activity can be launched via several different intents. Two of these intents correspond to deep links - namely the *wc://* and the *gem://* scheme. When a user opens such a deep link, Android fetches all suitable intent handlers for the registered apps on the mobile device, and prompts the user to choose an app to handle the intent. This introduces a phishing vector.

An attacker could craft an app that handles the *wc://* or the *gem://* deep links. In the first step, the attacker needs to convince the victim to install the malicious app on the mobile device. If the victim consequently opens a deep link with either the *wc://* or the *gem://* scheme, then Android will prompt the victim to choose an app for handling the link. If the victim selects the attacker's app, then a phishing attack could trick the victim into providing secret information such as a seed phrase.

Steps to reproduce:

1. Deploy an app claiming to handle deep links for the *gem://* scheme on a mobile device where the Gem Wallet app is installed.
2. Send the intent shown below to the mobile device:

Command to trigger phishing:

```
$ adb shell am start -a android.intent.action.VIEW -d  
"gem://localhost"  
Starting: Intent { act=android.intent.action.VIEW dat=gem://localhost  
}
```

3. This results in an app-picker dialog showing both the attacker's app and the Gem Wallet app, as demonstrated in the screenshot below:

⁴ <https://developer.android.com/training/sign-in/biometric-auth>

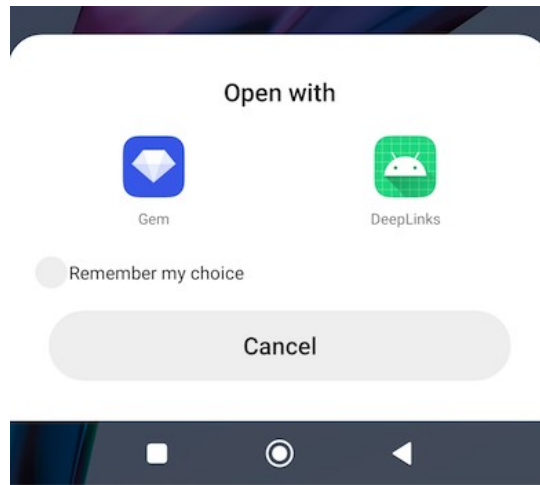


Fig.: This screen demonstrates the successful phishing attack.

The excerpt below highlights the issue. The *MainActivity* of the Gem Wallet app is exported, and can be launched via intents that have either the *wc* or *gem* scheme:

Affected file:

wallet-2.114.40/android/app/src/main/AndroidManifest.xml

Affected code:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">
    [...]
    <application
        [...]
    >
        <activity
            android:name="com.gemwallet.android.MainActivity"
            android:exported="true"
            [...]
        >
            [...]
            <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="wc" />
            </intent-filter>
```

```
<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="gem" />
  </intent-filter>
  [...]
</activity>
[...]
</application>
</manifest>
```

To mitigate this issue, Cure53 recommends phasing-out deep links, and instead to use app links⁵ as an alternative. This will ensure that Android utilizes only legitimate apps for handling such deep links.

GEM-01-014 WP3: DoS of Android app via malformed intents (*Low*)

Fix note: This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.

The Gem Wallet app exposes the *MainActivity* activity through several intents. Fuzzing the intent handlers uncovered a Denial-of-Service (DoS) vulnerability via malformed intents, where malformed intents can be used to crash the app.

An attacker could craft an app that sends malformed intents to the Gem Wallet app. The attacker must then convince the victim to install the malicious app on the mobile device. If the victim then starts both the Gem Wallet app and the attacker's app, and foregrounds the attacker's app, then the attacker's app can crash the Gem Wallet app via a malformed intent.

It should be noted that until Android SDK version 28, apps were able to send intents even when backgrounded. Due to the fact that the Android app supports as minimum SDK version 28, this finding was rated with a severity of *Low*, reflecting the more limited attack vector.

Steps to reproduce:

1. Start the Gem Wallet mobile app.
2. Attach the Android mobile device to the computer, and observe Logcat using Android Studio.
3. Install the IntentFuzzer⁶ app, navigate to "NonSystemApps", and select "Gem".

⁵ <https://developer.android.com/training/app-links>

4. Click "Serialize Fuzz All" in the IntentFuzzer app.
5. Observe the following output in Logcat, demonstrating the crash:

Logcat:

```
2026-09-23 18:53:22.931 20653-20653 AndroidRuntime
com.gemwallet.android D Shutting down VM
2026-09-23 18:53:22.932 20653-20653 AndroidRuntime
com.gemwallet.android E FATAL EXCEPTION: main
```

Process: com.gemwallet.android, PID: 20653

```
java.lang.RuntimeException: Parcelable encountered
ClassNotFoundException reading a Serializable object (name =
com.android.intentfuzzer.util.SerializableTest)
```

```
at android.os.Parcel.readSerializable(Parcel.java:3514)
[...]
```

To mitigate this issue, Cure53 recommends implementing proper, generic, input data handling for all exposed activities. Furthermore, it is advised that the handling of such data should be safeguarded with corresponding exception handling.

GEM-01-016 WP3: Biometric authorization bypass via file manipulation (*Medium*)

Fix note: *This issue has been mitigated by the Gem Wallet team and verified by Cure53 to be working as expected.*

Following the discovery of [GEM-01-012](#), Cure53 scrutinized the codebase for further bypasses of the biometric authorization feature where secure sensitive operations were concerned.

Reviewing the source code of the *wallet* repository revealed that the Android app's *UserConfig* class could not only be instrumented for a bypass, but also that it contains a fallback logic. In particular, the Boolean setting for whether biometric authorization is required (referred to as the "Enable passcode" setting in the app) is stored in two places - namely a secure storage and a config storage. The secure store encrypts its entries, whereas the config storage stores them in plaintext. It was discovered that the *authRequired* function of the *UserConfig* class reads the "Enable passcode" setting from the secure storage, falling back to the plain config storage if the item is absent in the secure storage. This leads to a bypass vulnerability for the biometric authorization feature on a rooted Android device.

⁶ <https://github.com/MindMac/IntentFuzzer>

An attacker who gained access to the rooted Android device of a victim could remove the biometric authorization feature from the app via two steps. In the first step, the attacker would open the *gem_secure_preferences.xml* file (backing up the secure storage) in the app data folder and delete the corresponding entry (as detailed in the reproduction steps shown below) from the file. To complete the bypass, the attacker would then set the *auth*-value of the *config.xml* file (backing up the config storage) in the app data folder to *false*. Restarting the app removes biometric authorization entirely, which enables the attacker not only to read the seed phrases of wallets, but also to sign transactions.

Steps to reproduce:

1. Connect a rooted Android device to a computer.
2. Open the Gem Wallet app, and enable passcode authorization.
3. Open Android Studio, and navigate to the */data/data/com.gemwallet.android* folder.
4. Download the files *config.xml* and *gem_secure_preferences.xml* file locally.
5. In the *config.xml* file, set the *auth*-Boolean to *false*.
6. In the *gem_secure_preferences.xml* file, delete the entry for the key *gateway_secure_preferences_8296[...]*.
7. Upload both modified files to the Android device.
8. Close the Gem Wallet app and restart it. Observe that the app opens without prompting the user for authorization via passcode.

The excerpt below demonstrates the issue. It is clear that the *authRequired* function falls back to the *configStore* object if the *secureStore* does not contain the *SecureKey.Auth* item:

Affected file:

wallet-2.114.40/android/data/services/gemstone/src/main/kotlin/com/gemwallet/android/data/services/gemstone/config/UserConfig.kt

Affected code:

```
fun authRequired(): Boolean =  
    secureStore.get(SecureKey.Auth.string)?.toBooleanStrictOrNull() ?:  
    configStore.getBoolean(ConfigKey.Auth.string)
```

To mitigate this vulnerability, Cure53 recommends removing the fallback from the app entirely. This is due to the fact that if the item *SecureKey.Auth* is not found in the secure storage, then someone must have removed it from the *gem_secure_preferences.xml* file manually, as the app does not contain a function to remove it. This indicates a tamper attempt, and it is advised that the app should no longer start, so as to protect the victim from potential fund theft.

GEM-01-017 WP1: Missing zeroization in mnemonic generation (*Low*)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While auditing the Rust Keystore boundary, it was found that the mnemonic generation function does not properly zeroize the generated mnemonic before it is returned. Notably, the entropy used to generate the mnemonic properly implements zeroization. However, when the mnemonic is finally returned, the iterator is collected into a non-zeroizing vector of plain strings, which do not implement zeroization.

Additionally, the *bip39* crate used to derive the mnemonic from the generated entropy does not implement zeroization without explicitly enabling the feature in the projects *Cargo.toml*. Hence, the *Mnemonic* object does not properly zeroize at the end of the lifecycle.

Affected file:

wallet-2.114.40/core/crates/gem_keystore/src/mnemonic.rs

Affected code:

```
pub fn generate(word_count: usize) -> Result<Vec<String>, KeystoreError> {
    let entropy_len = entropy_len_for_word_count(word_count)?;
    let entropy = Zeroizing::new(gem_crypto::random::bytes::<32>()?);
    let mnemonic = Bip39Mnemonic::from_entropy_in(Language::English,
&entropy[..entropy_len]).map_err(|_|
KeystoreError::invalid_input("mnemonic"))?;
    Ok(mnemonic.words().map(|word| word.to_string()).collect())
}
```

Cure53 recommends enabling the *zeroize* feature⁷ for the *bip39* dependency, and properly zeroizing the returned values by wrapping the vector in the *Zeroizing* wrapper type, which clears the underlying memory containing the seed phrase words when the object's lifetime ends and it is dropped. Zeroizing the return value has already been implemented for the other relevant implementations of the *Mnemonic* struct.

⁷ <https://docs.rs/crate/bip39/2.2.2/features>

GEM-01-020 WP2: Missing zeroization when deriving private keys (Low)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

While scrutinizing the codebase for missing zeroization, it was discovered that the `gem_derivation` crate does not use zeroization for the BIP-32 and Bitcoin implementations when deriving keys. Particularly, the `derive_secp256k1_private_key` function of the `bip32.rs` file, and the `derive_extended_public_key` function of the `bitcoin.rs` file both create variables that do not have zeroization support. This implies that secret data remains in memory even when the current scope of the function ends.

Affected files:

- `wallet-2.114.40/core/crates/gem_derivation/src/mnemonic/bip32.rs`
- `wallet-2.114.40/core/crates/gem_derivation/src/mnemonic/bitcoin.rs`

The excerpt below demonstrates the issue for the `bip32.rs` file. The function shown creates a variable named `master` from the `seed` parameter, corresponding to the root of key derivation. It should be noted that this variable does not support zeroization. Similarly, the `derived` variable also lacks zeroization support. Similar observations apply to the `bitcoin.rs` file for the `master` and `account_private_key` variables.

Affected code (`bip32.rs`):

```
pub(super) fn derive_secp256k1_private_key(seed: &[u8], path: &str) ->
Result<Zeroizing<Vec<u8>>, AccountDerivationError> {
    let secp = ::bitcoin::secp256k1::Secp256k1::signing_only();
    let master = Xpriv::new_master(Network::Bitcoin,
seed).map_err(map_bip32_error)?;
    let derivation_path =
DerivationPath::from_str(path).map_err(map_bip32_error)?;
    let derived = master.derive_priv(&secp,
&derivation_path).map_err(map_bip32_error)?;
    Ok(Zeroizing::new(derived.private_key.secret_bytes().to_vec()))
}
```

To mitigate this issue, Cure53 recommends wrapping all variables containing sensitive information in zeroizeable variables, to ensure that those variables are cleared as soon as the current scope is left. Alternatively, it is advised that a dedicated struct could be created, implementing `Drop` and performing zeroization when an instance of the struct is dropped.

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.

GEM-01-002 WP2: No checks on entropy size for seed on deriving priv. keys ([Info](#))

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

A static source code review of the Cardano private key derivation from a mnemonic revealed that the underlying crate for deriving the private key fails to check the length of the entropy. Instead, the `derive_private_key` function accepts arbitrarily-sized `u8` arrays for the `entropy` parameter. This could pose a significant security risk if the `derive_private_key` function is used outside of the Gem Wallet app ecosystem, and means that the caller of this function must implement checks regarding the `entropy` array.

In particular, if an application uses the `derive_private_key` function without enforcing any constraints on the `entropy` array, then a user could supply short arrays. This could aid an attacker in attempting to enumerate potential private keys for a victim. However, due to the fact that the Gem Wallet uses the `derive_private_key` function only on checked and verified inputs, this issue was ranked with a severity level of *Info*.

Further review revealed that the same pattern applies to the BIP32, Bitcoin, and Slip10 private key derivation functions:

Affected files:

- `wallet-2.114.40/core/crates/gem_derivation/src/mnemonic/bip32.rs`
- `wallet-2.114.40/core/crates/gem_derivation/src/mnemonic/bitcoin.rs`
- `wallet-2.114.40/core/crates/gem_derivation/src/mnemonic/cardano.rs`
- `wallet-2.114.40/core/crates/gem_derivation/src/mnemonic/slip10.rs`

The excerpt below demonstrates the issue. It is evident that the `derive_private_key` function fails to implement any checks on the provided `entropy` array:

Affected code (cardano.rs):

```
pub(super) fn derive_private_key(entropy: &[u8]) ->
Result<Zeroizing<Vec<u8>>, AccountDerivationError> {
    let root = root_from_entropy(entropy)?;
    let payment = derive_path(root.clone(), PAYMENT_PATH)?;
    let stake = derive_path(root, STAKE_PATH)?;
```

```
let mut private_key = Vec::with_capacity(PRIVATE_KEY_LEN);
extend_node(&mut private_key, &payment);
extend_node(&mut private_key, &stake);
Ok(Zeroizing::new(private_key))
}
```

To mitigate this issue, Cure53 recommends enforcing a length check on the provided *entropy* array for all of the functions mentioned above, and to error-out if the supplied array does not have the expected length.

GEM-01-003 WP1: No minimum checks on Argon2id parameters ([Info](#))

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During a source code review of the *gem_keystore* crate, it was noted that the *crypto.rs* file performs all of the security-relevant operations. This includes a password-based key derivation function, and encryption using AES in Galois Counter mode. The former is facilitated by utilizing Argon2id, which is a state-of-the-art KDF. It was found that the default parameters for Argon2id comply with best-practice at the time of the assessment, however, it is advised that the implementation could also benefit from a guard enforcing these default values as a minimum security level, similar to the already existing checks for maximum values.

It should be noted that this issue is not currently exploitable, as there is no option to alter the existing values. However, in the future, and on upgrading the Gem Wallet app, this could turn into a missing check, and could therefore pose a security risk in future versions. Therefore, it was decided to report this finding as an informational issue.

The excerpt shown below demonstrates the issue. It is evident that the *validate* function fails to enforce minimum values on the parameters of Argon2id:

Affected file:

wallet-2.114.40/core/crates/gem_keystore/src/storage/crypto.rs

Affected code:

```
pub(super) fn validate(&self) -> Result<(), KeystoreError> {
    match self {
        Self::Argon2id {
            memory_kib,
            iterations,
            parallelism,
            output_len,
            ..
        } => {
            if *memory_kib == 0 || *memory_kib > MAX_ARGON2_MEMORY_KIB {
```

```

        return Err(KeystoreError::corrupt_file("invalid Argon2
            memory"));
    }
    if *iterations == 0 || *iterations > MAX_ARGON2_ITERATIONS {
        return Err(KeystoreError::corrupt_file("invalid Argon2
            iterations"));
    }
    if *parallelism == 0 || *parallelism > MAX_ARGON2_PARALLELISM {
        return Err(KeystoreError::corrupt_file("invalid Argon2
            parallelism"));
    }
    [...]
}
}
}

```

To mitigate this issue, Cure53 recommends implementing a guard against insecure parameters for Argon2id. This will equip the Gem Wallet app with a defense-in-depth guard for future iterations.

GEM-01-004 WP1: Missing expected type check on decrypting secrets ([Info](#))

When decrypting a secret, the *gem_keystore* crate first reads the entire file from disk and performs some checks on the secret's header. These checks include a verification of the *keystore_id* field of the header against an expected identifier, as well as implicit checks for the KDF parameters, among others. The header of a secret also contains a field encoding the *kind* of the ciphertext - either corresponding to a mnemonic or a private key. It was noted that the *decrypt_secret* function fails to perform a check of the *kind* field of a secret's header against an expected kind.

It should be noted that further code review confirmed that the calling sites of the *decrypt_secret* function implicitly perform such a check. However, the *gem_keystore* crate fails to implement this, and as a consequence, callers must remember to implement this check. Due to these circumstances, it was decided to report this missing defense-in-depth measure as an informational issue.

From the excerpt shown below, it is clear that the *decrypt_secret* function fails to enforce a check on the *type* of a secret:

Affected file:

wallet-2.114.40/core/crates/gem_keystore/src/storage/secret.rs

Affected code:

```

pub(super) fn decrypt_secret(parsed: ParsedFile, expected_id:
    Option<&KeystoreId>, password: &[u8]) -> Result<SecretPayload,
    KeystoreError> {

```

```
validate_v4_password(password)?;  
if let Some(expected_id) = expected_id  
    && parsed.header.keystore_id != expected_id.as_str()  
{  
    return Err(KeystoreError::corrupt_file("authenticated keystore id  
        does not match the requested id"));  
}  
let ParsedFile { header, ciphertext: mut body } = parsed;  
let aad = authenticated_bytes(&header)?;  
let key = derive_key(password, &header.kdf)?;  
// On tag failure aes-gcm leaves unauthenticated plaintext in the  
// buffer, so zeroize before bailing.  
if let Err(error) = decrypt_aes256_gcm(key.as_ref(),  
    header.cipher.nonce(), &aad, &mut body) {  
    body.zeroize();  
    return Err(error);  
}  
SecretPayload::from_bytes(header.kind, body)  
}
```

To mitigate this issue, Cure53 recommends implementing a guard check for the *type* of a secret from its header, by introducing a parameter to the *decrypt_secret* function, and comparing them.

GEM-01-005 WP2: Signing keys lack binding to approved accounts ([Info](#))

Source review identified that *GemKeystore::signing_key* loads a secret without verifying that its derived address matches the account selected for the operation. The confirmation check compares two addresses obtained from the same wallet record, rather than deriving an address from the loaded key. Authenticating the keystore identifier does not establish this account binding. Tron implements its own sender check, but the common signing boundary does not.

A wallet where the stored account and secret disagree can therefore produce signatures for a different address. The resulting failure may appear only at the network or relying application. The repository's legacy single Solana migration test explicitly accepts such a mismatch; the mismatch predates v4 and was not caused by the Rust rewrite. No general dApp-controlled route for substituting a signing key was demonstrated. This weakness compounds the concrete wallet-selection defect in [GEM-01-006](#), and is classified as a miscellaneous issue because its broader exploitation requires another binding failure.

Affected file:

wallet-2.114.40/core/gemstone/src/keystore/keystore.rs

Affected code:

```
pub(crate) fn signing_key(&self, keystore_id: &str, chain: Chain, password:
Vec<u8>) -> Result<Zeroizing<Vec<u8>>, GemstoneError> {
    let password = Zeroizing::new(password);
    self.load_private_key(keystore_id, chain, &password)
}
```

The transaction signing entry point likewise routes the key without a common address check, as demonstrated below:

Affected file:

wallet-2.114.40/core/gemstone/src/signer/chain.rs

Affected code:

```
pub fn sign_input(&self, input: GemSignerInput, private_key:
Zeroizing<Vec<u8>>) -> Result<Vec<GemSignedTransaction>, GemstoneError> {
    let signer_input: SignerInput = input.into();
    self.route(&signer_input, private_key.as_slice())
}
```

Steps to reproduce:

1. Use *migrate_v3_allows_legacy_single_solana_derivation_mismatch* in *keystore.rs* to prepare the accepted legacy mismatch.
2. Compare the address encoded in the test's *single_solana_2e6GNStEYy7M31duwabbAx3R4Tzz62LxryGrqcMhoaaV* wallet identifier with the address returned by *add_accounts*.
3. Retrieve the signing key in a crate-local test, and derive its Solana account with *derive_account_from_private_key*.
4. Verify that key retrieval succeeds even though the derived address differs from the account encoded in the wallet identifier.

To mitigate this issue, it is recommended to bind signing-key retrieval to the expected chain and account address, to derive the address from the loaded key, and to reject mismatches before producing any signature. It is also advised to apply the check to transaction, message, and authentication signing, as well as to use chain-appropriate address comparison - including case-insensitive comparison for Ethereum addresses. Existing validation found matching derived addresses across all 56 mnemonic-supported chains, providing a compatibility baseline for the check.

GEM-01-009 WP3: Concurrent iOS password creation can lock-out wallet (*Low*)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Source review identified that iOS creates the shared keystore password through an unsynchronized read, generate, and write sequence. Two first-use calls can both read an absent password. If A stores its password before B's later existence probe inside `KeychainDefault.set`, then B takes the update branch and overwrites A's value. A keystore encrypted with A's password then becomes unreadable using the password retained in Keychain.

The affected wallet cannot sign or export its secret. Recovery requires an external recovery phrase or private key, followed by deletion and re-import. The destructive ordering requires both calls to enter password creation before the first keystore file is written.

When both Keychain probes instead see absence, the second `add` fails with `errSecDuplicateItem`, and does not overwrite the password. Validation included a macOS Keychain probe; the race was not reproduced on an iOS device.

Affected file:

`wallet-2.114.40/ios/Packages/GemstoneServices/Sources/Keystore/LocalKeystore.swift`

Affected code:

```
public func keystorePassword(createIfMissing: Bool) throws -> String {
    let password = try keystorePassword.getPassword()
    if password.isEmpty {
        return password
    }
    guard createIfMissing else {
        throw KeystoreError.missingPassword
    }
    let newPassword = try SecureRandom.generateKey(length: 32).hex
    try keystorePassword.setPassword(newPassword, authentication: .none)
    return newPassword
}
```

Steps to reproduce:

1. Configure two concurrent first-wallet import calls in a test installation with no password or keystore file.
2. Pause both calls after their initial password reads return empty, before either writes a password.
3. Resume A until it stores password A and creates its keystore file.

4. Resume B so its existence probe in `KeychainDefault.set` finds A's password and updates it to password B.
5. Verify that the retained password cannot decrypt A's file, and that A's wallet cannot export its phrase.

The same empty-value handling also permits a watch-only user to enable authentication before a password exists. `changeAuthentication` stores an empty password, which a later real-wallet import replaces using `.none`, clearing the selected authentication policy.

To mitigate this issue, it is recommended to serialize password initialization across every custodian of the shared Keychain item with a synchronous shared lock. It is also advised to use add-only creation and re-read the existing value on `errSecDuplicateItem`, to distinguish absent and empty values, to reject empty password writes, and to preserve an authentication policy selected before the first secret-bearing wallet is created. It should be noted that an instance-local queue does not protect the other password custodians.

GEM-01-011 WP3: Android Keystore failure exposes encryption keys (*Medium*)

Fix note: *This issue has been mitigated by the Gem Wallet team and verified by Cure53 to be working as expected.*

Source review identified that the Android password store accepts Tink's cleartext-keyset fallback. The application configures `AndroidKeysetManager` with a master-key URI, then retrieves its keyset without checking whether Android Keystore protection was established. In the pinned Tink version (1.22.0), certain first-use master-key generation or access failures instead cause the keyset to be stored without encryption.

An attacker who obtains a read-only copy of the affected application's private storage can recover the password-encryption key from `gem_wallet_password_keyset_prefs.xml`. This key decrypts the shared password in `gem_wallet_passwords.xml`, which in turn opens the wallet keystores. The attacker does not need the device's non-exportable master key. This exposure is conditional on the fallback having occurred; ordinary applications cannot read the private directory, and application backups are disabled.

Affected file:

`wallet-2.114.40/android/app/src/main/kotlin/com/gemwallet/android/data/password/
TinkEncryptedKeyValueStore.kt`

Affected code:

```
private fun buildAead(): Aead {  
    AeadConfig.register()  
    val keysetHandle = AndroidKeysetManager.Builder()  
        .withSharedPref(context, config.keysetName,  
            config.keysetPreferencesFileName)  
        .withKeyTemplate(AesGcmKeyManager.aes256GcmTemplate())
```

```
        .withMasterKeyUri("android-keystore://${config.masterKeyAlias}")  
        .build()  
        .keysetHandle  
return keysetHandle.getPrimitive(RegistryConfiguration.get(),  
    Aead::class.java)  
}
```

Steps to reproduce:

1. Use a fresh test installation and inject a *ProviderException* during first-use master-key generation in Tink's *readOrGenerateNewMasterKey*.
2. Create a test wallet and inspect the serialized keyset in *shared_prefs/gem_wallet_password_keyset_prefs.xml*.
3. Parse this as an unencrypted Tink keyset and obtain its authenticated-encryption primitive in a separate test process without the Android Keystore alias.
4. Decrypt the shared-password ciphertext with associated data *wallet_password:password*, then use the recovered password to open the test wallet's v4 file.

Tink also accepts previously-stored cleartext keysets when reading existing storage, and does not automatically rewrite them under a recovered master key. The weakness can therefore persist after the original failure disappears.

To mitigate this issue, it is recommended to explicitly establish and verify the master key before permitting password storage, generating it only when its alias is genuinely absent. It is further advised to treat generation and access failures as errors, to reject unprotected keysets for new wallets, and to provide an explicit, verified migration for existing cleartext keysets. Finally, it is recommended to preserve password recovery during failure handling, as described in [GEM-01-010](#); introducing an unlock requirement without this prerequisite risks making existing wallets inaccessible.

GEM-01-015 WP3: Android application supports outdated SDK versions (*Low*)

A static analysis of the provided APK revealed that the Android app supports a minimum SDK of 28. Having support for such low SDK versions introduces a security risk, as such older versions no longer receive active security updates and patches. This unnecessarily increases the attack surface.

The excerpt below was extracted from the deployed APK on a mobile device using the *Drozer*⁸ tool. It is evident that the app supports a minimum SDK version of 28:

Android manifest (exported from deployed APK):

```
<manifest versionCode="853"  
    versionName="2.114.40"  
    compileSdkVersion="37"
```

⁸ <https://github.com/ReverseSecLabs/drozer/tree/develop>

```
        compileSdkVersionCodename="17"  
        package="com.gemwallet.android"  
        platformBuildVersionCode="37"  
        platformBuildVersionName="17">  
    <uses-sdk minSdkVersion="28"  
        targetSdkVersion="37">  
    </uses-sdk>  
    [...]  
</manifest>
```

To mitigate this issue, Cure53 recommends increasing the minimum-supported SDK version to at least 34, as this version still receives active security updates and patches.

GEM-01-018 WP3: Missing screenshot protection for seedphrase export (*Low*)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Testing and code review showed that the iOS build does not adequately protect against screenshots and screen recordings of the seed phrase export view, both when the app is active and when it is shown in an inactive state in the app switcher. Due to these missing mitigations, users could possibly leak their seed-phrase when accidentally performing a screenshot, or when they open the app switcher during a screen recording while the seed phrase view is still shown in the application.

Notably, both the app switcher inactive state and the screenshot / recording protection are implemented in the Android app.

Steps to reproduce:

1. Open the seed phrase export view in the Gem Wallet app.
2. Perform a screenshot and a screen recording of this view. Observe that the seed phrase is not obfuscated in either.
3. Background the app and start a screen recording. Open the app switcher and observe that the seed phrase view is visible and not obfuscated in the recording.

Cure53 recommends that the iOS Swift application should implement strong obfuscation for screenshots / screen recordings in the *ShowSecretPhraseViewModel*. Additionally, it is advised that the seed phrase should be covered while the application is inactive and shown in the app switcher.

GEM-01-019 WP2: Existing wallet check blocks secrets recovery (*Low*)

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

Source review identified that `import_wallet` treats an existing database row as sufficient evidence that a wallet is already imported. It returns *Existing* before checking the keystore or storing the supplied recovery phrase. Consequently, importing the correct phrase cannot repair a wallet where the row survived the loss of its keystore file - including the partial-deletion state described in [GEM-01-007](#).

Both applications present this outcome as an existing wallet, rather than a recovery failure. Signing and phrase export continue to fail. The secret remains recoverable from the supplied phrase, but the user must first delete the stale entry or use another application. The finding does not establish loss of funds for users who retain their recovery material.

Affected file:

`wallet-2.114.40/core/gemstone/src/services/wallet/mod.rs`

Affected code:

```
pub async fn import_wallet(&self, name: String, import:
GemWalletImportType, source: WalletSource) -> Result<GemWalletImportResult,
GemServiceError> {
    [...]
    let wallets = self.store.get_wallets().await?;
    if let Some(wallet) = rules::existing_wallet(&wallets, &wallet_id,
        preview.wallet_type) {
        return Ok(GemWalletImportResult::Existing { wallet });
    }
    [...]
}
```

The existing keystore probe also collapses read errors into absence.

Affected file:

`wallet-2.114.40/core/gemstone/src/keystore/keystore.rs`

Affected code:

```
pub fn exists(&self, keystore_id: String) -> bool {
    matches!(self.inner.get_meta(&keystore_id), Ok(Some(_)))
}
```

Steps to reproduce:

1. Import a test recovery phrase and retain the resulting wallet row.
2. Remove only that wallet's v4 keystore file from the test directory, reproducing a failed deletion.
3. Call `import_wallet` with the same phrase and wallet type.
4. Verify that the result is *Existing*, no keystore file is created, and signing still fails.

To mitigate this issue, it is recommended to check secret availability before returning *Existing* for a wallet that requires a secret, and to allow a verified recovery phrase to rebuild a genuinely missing file. In the same change, it is advised to make *exists* distinguish absence from read failure, and to authorize rollback only when the current import actually created the file. Otherwise, a transient read failure followed by a store error could cause rollback to delete a pre-existing healthy secret. Finally, it is recommended to preserve watch-only behavior and test recovery against a surviving database row.

GEM-01-021 WP2: Secp256k1 malleability in `recover_ethereum_address` ([Info](#))

Fix note: *This issue has been fixed by the Gem Wallet team and verified by Cure53 to be working as expected. The described issue no longer exists.*

During a source code review of the `signer` crate, it was noted that the crate contains wrappers for the Ed25519 and Secp256k1 elliptic curves. The wrapper for the latter curve contains a function to recover a public key from a signature. It was discovered that the signature's *s* value is not normalized, resulting in a potential malleability issue.

In particular, the utilized third-party function `VerifyingKey::recover_from_prehash` does not normalize the signature either, and instead computes the recovered key directly from the signature. Depending on its usage, this results in a malleability issue if not handled in calling sites.

For the Gem Wallet app, the calling site corresponds to the `gem_auth::verify_auth_signature` function, which verifies the signature by enforcing a check on the expected public key, thereby suffering from malleability.

The malleability issue is demonstrated in the unit test below. This first creates a signature for a fixed digest, resulting in the `signature` variable. The test then splits the signature into (r,s) and negates *s* to $-s$. From this, it creates the `forged_signature` instance, which is ultimately used as a parameter to the `recover_ethereum_address` function. The test concludes by demonstrating that both `signature` and `forged_signature` yield the same address:

PoC – Unit test:

```
#[test]
fn malleability_of_signature() {
    let private_key = hex::decode(TEST_PRIVATE_KEY).unwrap();
    let (rs, recovery_id) = sign_digest(&DIGEST, &private_key).unwrap();
```

```
let signature = [rs, vec![recovery_id]].concat();

let original =
Signature::try_from(&signature[..RECOVERY_ID_INDEX]).unwrap();
let (r, s) = original.split_scalars();
let forged_s = -*s.as_ref(); // n - s (high-S form)
let forged = Signature::from_scalars(r.to_bytes(),
forged_s.to_bytes()).unwrap();
let forged_signature = [forged.to_bytes().to_vec(), vec![recovery_id ^
1]].concat();

assert_ne!(forged_signature[..RECOVERY_ID_INDEX],
signature[..RECOVERY_ID_INDEX], "must differ byte-wise");
assert!(forged.normalize_s() != forged, "forged signature is high-S");

//confirm that both signatures verify, and give correct address
assert_eq!(recover_ethereum_address(&DIGEST, &signature).unwrap(),
TEST_PRIVATE_KEY_ETHEREUM_ADDRESS);
assert_eq!(recover_ethereum_address(&DIGEST,
&forged_signature).unwrap(), TEST_PRIVATE_KEY_ETHEREUM_ADDRESS);

let mut forged_offset = forged_signature.clone();
forged_offset[RECOVERY_ID_INDEX] += ETHEREUM_RECOVERY_ID_OFFSET;
assert_eq!(recover_ethereum_address(&DIGEST, &forged_offset).unwrap(),
TEST_PRIVATE_KEY_ETHEREUM_ADDRESS);
}
```

Output:

```
running 1 test
test secp256k1::tests::malleability_of_signature ... ok
```

```
test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 20 filtered
out; finished in 0.00s
```

The excerpt shown below demonstrates the issue. It is clear that the `recover_ethereum_address` function fails to normalize the signature:

Affected file:

wallet-2.114.40/core/crates/signer/src/secp256k1.rs

Affected code:

```
pub(crate) fn recover_ethereum_address(digest: &[u8; 32], signature: &[u8])
-> Result<String, SignerError> {
    let signature =
EthereumSignature::try_from(signature).map_err(SignerError::from_display)?.
normalized_s();
    let recovery_id = RecoveryId::new(signature.v(), false);
```

```
let signature =  
  SecpSignature::from_scalars(signature.r().to_be_bytes(),  
    signature.s().to_be_bytes()).map_err(|_|  
    SignerError::invalid_input("Invalid Ethereum signature"))?;  
  let verifying_key = VerifyingKey::recover_from_prehash(digest,  
    &signature, recovery_id).map_err(|_| SignerError::invalid_input("Invalid  
    Ethereum signature"))?;  
  let public_key = verifying_key.to_sec1_point(false);  
  Ok(Address::from_raw_public_key(&public_key.as_bytes()  
    [1..]).to_checksum(None))  
}
```

To mitigate this issue, Cure53 recommends normalizing signatures prior to any call to a third-party library, in order to ensure that malleability issues cannot occur. Furthermore, if malleability should be entirely rejected, then it is advised that the corresponding *recover* function should also enforce a check that the value of *s* did not change after normalization, and should error-out if the values differ.

Conclusions

As noted in the *Introduction*, this September 2026 penetration test and source code audit was conducted by Cure53 against the Gem Wallet (Rust core).

From a contextual perspective, thirteen working days were allocated to reach the coverage expected for this project. The methodology used conformed to a white-box strategy, pairing static code review with dynamic testing of the deployed mobile apps, and a team of four senior testers was assigned to the project's preparation, execution, and finalization.

The work was split into three separate work packages (WPs), defined as:

- **WP1:** White-box audits & reviews against the v4 keystore & UniFFI secret boundary
- **WP2:** White-box audits & reviews against the derivation, import & signing primitives
- **WP3:** White-box audits & reviews against the iOS/Android password custody

For communication purposes, the Gem Wallet and Cure53 teams shared a dedicated Telegram group. Communications were excellent, and few questions had to be asked. Cure53 provided frequent status updates, and live-reported all findings in Markdown format through the aforementioned Telegram group.

The Gem Wallet app is a crypto wallet app available for both iOS and Android. The iOS version of the app is written in Swift, and the Android version is written in Kotlin. Both rely on an ecosystem of Rust crates, and this forms the core of both applications. The source code of all three parts - namely both apps and the Rust core - was found to be well-written and clearly organized. As a consequence of this, it was straightforward for the testing team to familiarize itself with the codebase.

Prior to the engagement, the customer shared a link to the repository in scope for auditing. To freeze the code, a release tag for the audit was announced, which not only contained the source code, but also the APK for Android and an IPA file for iOS. To narrow-down the scope of the individual WPs, and to clarify the goals of the audit, the customer shared a link to a Gist page containing all required scope information. It should be noted that Cure53's testing efforts focused solely on the mobile apps, and did not include any backend / API testing.

The cryptography audit traced wallet secrets through import, encrypted storage, account derivation, signing, export, and deletion in the Rust core and its iOS and Android integrations. Source review and targeted tests covered the mobile password adapters, keystore parsing, account matching, and error handling.

It was positively noted that the iOS and Android apps utilize either the Keychain or the Android Keystore to persist some of the security critical information - such as passwords - for the mobile platforms. This demonstrates a good understanding of security principles when it comes to the secure storage of sensitive information.

Routine signing keeps decrypted private keys inside Rust; the mobile apps receive signed results, with secret export handled through separate calls. This reduces routine exposure of plaintext keys to the mobile layers. Derivation tests use fixed expected addresses, including published test vectors. Comparisons across all 56 mnemonic-supported chains matched. These tests help to detect accidental address changes, and it is advised that they should be kept when adding the missing account checks discussed later in this section.

It was found that a temporary Android Keystore failure could leave stored encryption keys unprotected ([GEM-01-011](#)). It is advised that handling this failure safely requires both refusing unprotected storage and migrating affected installations. Fixing initialization alone would leave existing unprotected keysets in place. The application sandbox and disabled backups limit exposure, since exploitation requires access to private application files.

The cryptographic primitives of the core crates include state-of-the-art algorithms, such as Argon2id and AES256-GCM. The parameters for Argon2id comply with current recommendations, and the usage of AES256-GCM is also sound. No reuses of nonce / encryption key pairs were found. The authentication of additional data, such as a secret header, was found to be complete. In this direction, the repository left a solid impression on the auditors. On a minor note, it was found that no minimum parameters are enforced for Argon2id ([GEM-01-003](#)). Furthermore, when decrypting a secret, it is advised that a *kind* check could enhance the security posture ([GEM-01-004](#)).

Randomness is essential for cryptographic applications, and hence Cure53 scrutinized the codebase for usages of insecure randomness in seed / mnemonic generation. It was concluded that the Gem Wallet project utilizes secure randomness for cryptographic operations - including mnemonic, nonce, and salt generation. For the creation of private keys out of an array of entropy, however, it was noted that the corresponding crate does not properly enforce array length checks ([GEM-01-002](#)).

Zeroization is a very important practice when dealing with cryptographic secrets in Rust applications. As the main core of the Gem Wallet app corresponds to Rust crates, special attention was given to the zeroization of cryptographically-relevant information. In this area, Cure53 confirmed two issues, both of which highlight a lack of zeroization of cryptographic values ([GEM-01-017](#), [GEM-01-020](#)).

The signing issues discovered in this engagement show that the wallet selected for an operation is not consistently tied to the key used ([GEM-01-005](#), [GEM-01-006](#)). It is advised that the core should verify that the loaded key derives the expected address for the requested chain, and swaps should retain their original wallet selection.

Validation found matching addresses across all 56 mnemonic-supported chains. It is advised that these checks should be retained as regression tests. Direct theft through the swap issue or a general dApp-controlled way to substitute signing keys was not demonstrated.

Another signing issue surfaced for the Secp256k1 elliptic curve, as it was discovered that the `recover_from_prehash` function allows for malleable signatures ([GEM-01-021](#)). However, due to the fact that higher layers in the apps perform a de-duplication based on the nonce, it was decided to report this finding in the *Miscellaneous Issues* section. Apart from this issue, the signer crate - which is responsible for wrapping the supported signature schemes - left a good impression, as it utilizes the underlying EC crates in a safe and correct manner.

It was found that the application does not reliably keep wallet records, secret files, and the shared password consistent after interrupted or failed operations ([GEM-01-007](#), [GEM-01-009](#), [GEM-01-010](#), [GEM-01-019](#)). This threatens continued access even when no attacker is involved. A surviving database entry should not prevent repair with a valid recovery phrase. Users with recovery material can restore access; those without it may lose the ability to spend.

The encoding and decoding of cryptographic keys on import and export functions were sound and left a good impression. No code execution flaws from insecure deserialization or DoS attack vectors were identified in this area.

Authentication and authorization are very important principles when developing a wallet app, and the auditors therefore scrutinized the apps and their workflows for any kind of bypasses, insecure defaults or missing checks. Multiple issues were found in this area. As described in [GEM-01-001](#), neither the iOS or Android app enforces authorization when deleting a wallet. In addition, neither the Android or iOS build enables user authentication for accessing the wallet by default, leaving it to be activated manually by the user. This significantly increases the attack surface for a malicious actor who gains physical possession of a device while it is in an unlocked state ([GEM-01-008](#)).

For users with rooted devices, the testing team also discovered ways to bypass the app's biometric authorization checks, either via hooks ([GEM-01-012](#)) or file manipulation ([GEM-01-016](#)). While both issues require a rooted device, the consequences are severe, as an attacker could not only gain access to seed phrases, but could also trigger any other action usually requiring authorization within the Android app.

During dynamic testing, and also when reviewing the Android app's manifest, Cure53 discovered several generic mobile app vulnerabilities in the Android app. In particular, the Android app was found to suffer from potential phishing attacks via deep link hijacking ([GEM-01-013](#)), as well as DoS situations triggered by external apps ([GEM-01-014](#)), and was also found to support outdated SDK versions ([GEM-01-015](#)). It is advised that this highlights a need to harden the app against common generic attack vectors.

The iOS Gem wallet is written in Swift 6 language mode, and integrates the same Rust core library as the Android wallet, using UniFFI and TypeShare bindings for the Swift / Rust boundary. The main key material lifecycle is handled within the Rust core library, which utilizes Zeroizing wrappers. The handling of key material outside of this boundary was audited as part of WP3. It was noted that to safely handle key material outside of the Rust core boundary, function calls querying the keystore using the password are wrapped in a deferred *withV4Password* function call, which zeroizes the decrypted password bytes after use.

Unlike the Android wallet, it was found that the iOS Gem wallet does not implement screenshot protection for sensitive views, such as the seed phrase export. This could expose the seed key material during everyday use ([GEM-01-018](#)).

Moving forward, it is advised that further testing should concentrate on failures and overlapping operations across the core and mobile stores. It should be checked that interrupted deletion can resume, that concurrent initialization cannot replace a password already in use, and that a recovery attempt cannot delete a healthy secret. Further, it is advised that storage read failures should remain distinct from missing values before password access is tightened. Finally, it is advised that existing installations affected by the Android fallback should also be migrated.

In summary, the Rust crates and the mobile apps for iOS and Android left a mixed impression on the Cure53 team. While some parts were in excellent shape from a security perspective - such as the cryptographic primitives, their parameter choices, and the usage of secure randomness - other areas were seen to suffer from systemic flaws. It is advised that the noted bypasses, lack of authorization, and insecure default settings that do not enforce authorization should be addressed as a priority. It is also advised that the reliability and order of wallet, secret, and account deletion should be given immediate attention. Finally, Cure53 recommends that the applications should be hardened against the common attack vectors identified as part of this engagement.

Cure53 would like to thank Herman and Tim from the Gem Wallet LLC team for their excellent project coordination, support and assistance, both before and during this assignment.