Zellic

Prepared for
Josh Stevens
Avara

Prepared by
Jasraj Bedi
Filippo Cremonese
Zellic

February 28, 2024

Family Wallet

iOS Application Security Assessment

Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Avara from January 29th to February 23rd, 2024. During this engagement, Zellic reviewed Family Wallet's code for security vulnerabilities, design issues, and general weaknesses in security posture.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to determine the following:

- The state of security of on-device at-rest key material storage
- The state of security of key material iCloud backups
- The state of security of key material generation and derivation
- The state of security of on-device in-use key material (limited)
- The state of security of the in-app browser RPC (confirmation screen bypasses)

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Dependencies — security concerns related to GRDB, WalletConnect, Amplitude, or any other dependencies unrelated to signing or handling key material.
- Transaction life cycle — we will not be addressing all aspects of transaction handling. To elaborate, we will not actively investigate bugs that might result in, for example, displaying a transaction as confirmed when it is not, or vice versa.
- Reviews of low-level cryptography implementations.

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

During this assessment, issues registering an Apple developer account prevented us from testing on a physical device. This impacted our ability to perform live assessments with device-authentication features enabled.

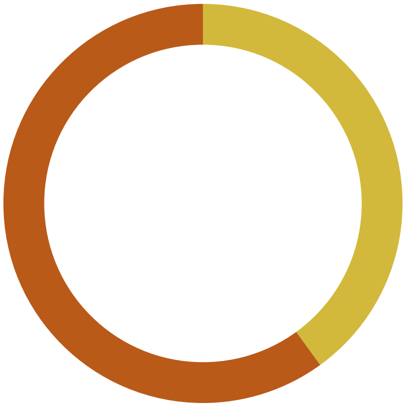
1.4. Results

During our assessment on the scoped Family Wallet codebase, we discovered five findings. No critical issues were found. Three findings were of high impact and two were of medium impact.

Additionally, Zellic recorded its notes and observations from the assessment for Avara's benefit in the Discussion section (4. ↗) at the end of the document.

Breakdown of Finding Impacts

Impact Level	Count
🟥 Critical	0
🟧 High	3
🟨 Medium	2
🟩 Low	0
◼ Informational	0



2. Introduction

2.1. About Family Wallet

Family Wallet is a self-custody Ethereum wallet designed for everyday use.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

> **Basic coding mistakes.** Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the codebase.
>
> **Business logic errors.** Business logic is the heart of any application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the logic to ensure that the code implements the expected functionality as specified in the platform's design documents.
>
> **Integration risks.** Several well-known exploits have not been the result of any bug within the application itself; rather, they are an unintended consequence of the application's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks.
>
> **Code maturity.** We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped codebase itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Family Wallet Codebase

Repositories	https://github.com/losfelizengineering/wallet-ios ↗ https://github.com/losfelizengineering/wallet-ios-core ↗ https://github.com/losfelizengineering/PINRemoteImage ↗
Versions	wallet-ios: `abe1d78615984662c3a9f1f80a443ec51b889f2a` wallet-ios-core: `0e1f258cb0435ff498ecb4e5904fdadae7a88188` PINRemoteImage: `a606ac705a4caaee81c67396d3d4c06266f94c59`
Program	*.swift
Type	swift
Platform	iOS

2.4. Project Overview

Zellic was contracted to perform a security assessment with two consultants for a total of six person-weeks. The assessment was conducted over the course of four calendar weeks.

Contact Information

The following project manager was associated with the engagement:

Chad McDonald
Engagement Manager
chad@zellic.io ↗

The following consultants were engaged to conduct the assessment:

Jasraj Bedi
Co-founder
jazzy@zellic.io ↗

Filippo Cremonese
Engineer
fcremo@zellic.io ↗

2.5. Project Timeline

The key dates of the engagement are detailed below.

January 29, 2024	Start of primary review period
January 30, 2024	Kick-off call
February 5, 2024	Review period paused
February 9, 2024	Review period resumed
February 23, 2024	End of primary review period

3. Detailed Findings

3.1. Origin spoofing

Target	BrowserScriptMessageHandler::BrowserWebClientID		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

When an RPC request from the embedded browser is received, the wallet needs to identify the origin of the request.

The origin is tracked by an object of type `BrowserWebClientID`.

```swift
struct BrowserWebClientID : Hashable, CustomStringConvertible {
    private let value: String

    private init(_ value: String) {
        self.value = value
    }

    init?(from url: URL) {
        guard let host = url.host else { return nil }

        self.init(host)
    }

    init(from origin: WKSecurityOrigin) {
        self.init(origin.host)
    }
    // ...
}
```

Regardless of whether the object is initialized from a URL or from a WKSecurityOrigin, only the host part of the origin is considered, while the protocol and port are ignored.

Impact

This issue causes different origins (for example, https://somedapp.com:443 and http://somedapp.com:8080) to be considered identical.

There are multiple ramifications due to this, but fundamentally this opens up multiple paths that could allow an attacker to spoof the origin of an RPC request.

For example, an attacker on the same network of a victim user could perform a Man in the Middle ("MITM") attack and redirect any page the user visits to a fake plaintext clone of a dApp. Note that the attacker can also do this by altering any HTTP page loaded to inject an iframe with a spoofed origin. The code in the iframe could wait until the page is in background and the user starts using the legitimate dApp and then request a signing operation that would very convincingly be presented as coming from the legitimate dApp.

The RPC message handler determines which webview has originated an RPC request by matching the `BrowserWebClientID` constructed from the URL of each open webview with the `BrowserWebClientID` constructed from the security origin of the frame (accessing `message.frameInfo.securityOrigin` in `BrowserWebScriptCoordinator::response`).

This allows a request coming from a background tab or an iframe to be treated as if coming from a different webview and potentially be presented as originating from a legitimate dApp.

Recommendations

Include protocol and port in the elements that are used to represent the origin or an RPC request.

Remediation

This issue was remediated in the following commits:

- `b779c7954d5b020e2dfe5e2763376ff5bfc721ad`
- `2b5aaec16e69466a112de48e3f27ec45b51dd3ce`

3.2. Improper URL handling with dangerous scheme

Target	Family Wallet		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The in-app embedded browser does allow navigating to atypical URLs, such as `ftp://` and `javascript:`. This can be abused (also in conjunction with Finding 3.1. ↗) to spoof the origin of an RPC request.

Impact

Special `javascript:` URLs could be abused to misrepresent the origin of a signing request — specifically, if the in-app browser navigates to the following URL (even in a nested iframe).

```
javascript://somedapp.com/%0amalicious_code()
```

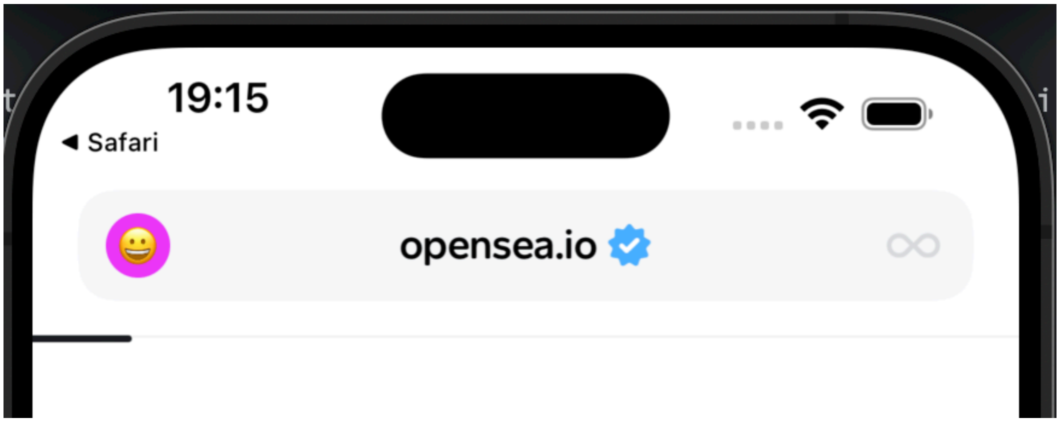
The `malicious_code()` would be executed, and if that code requested a wallet operation via the RPC, the origin would be indistinguishable from https://somedapp.com:443. This is because when parsing the URL according to RFC 3986, the host is somedapp.com, the protocol (schema) is `javascript`, and the path is `\nmalicious_code()`. However, `javascript:` URLs have special treatment in browsers, and the following JavaScript code would be executed:

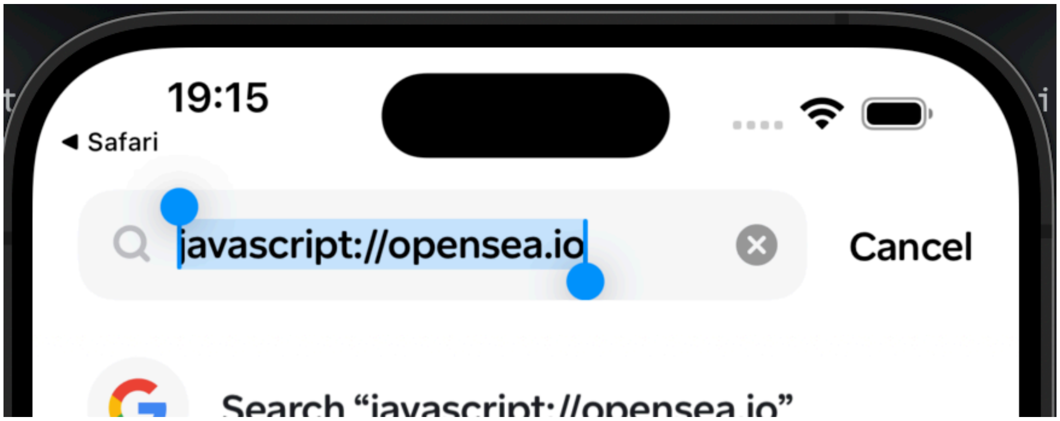
```
//somedapp.com/
malicious_code()
```

This exploit can also be initiated from Safari by abusing the built-in deep link functionality to open the Family Wallet in-app browser on a given URL. The attacker could trick the user into clicking on a link that points to the following URL:

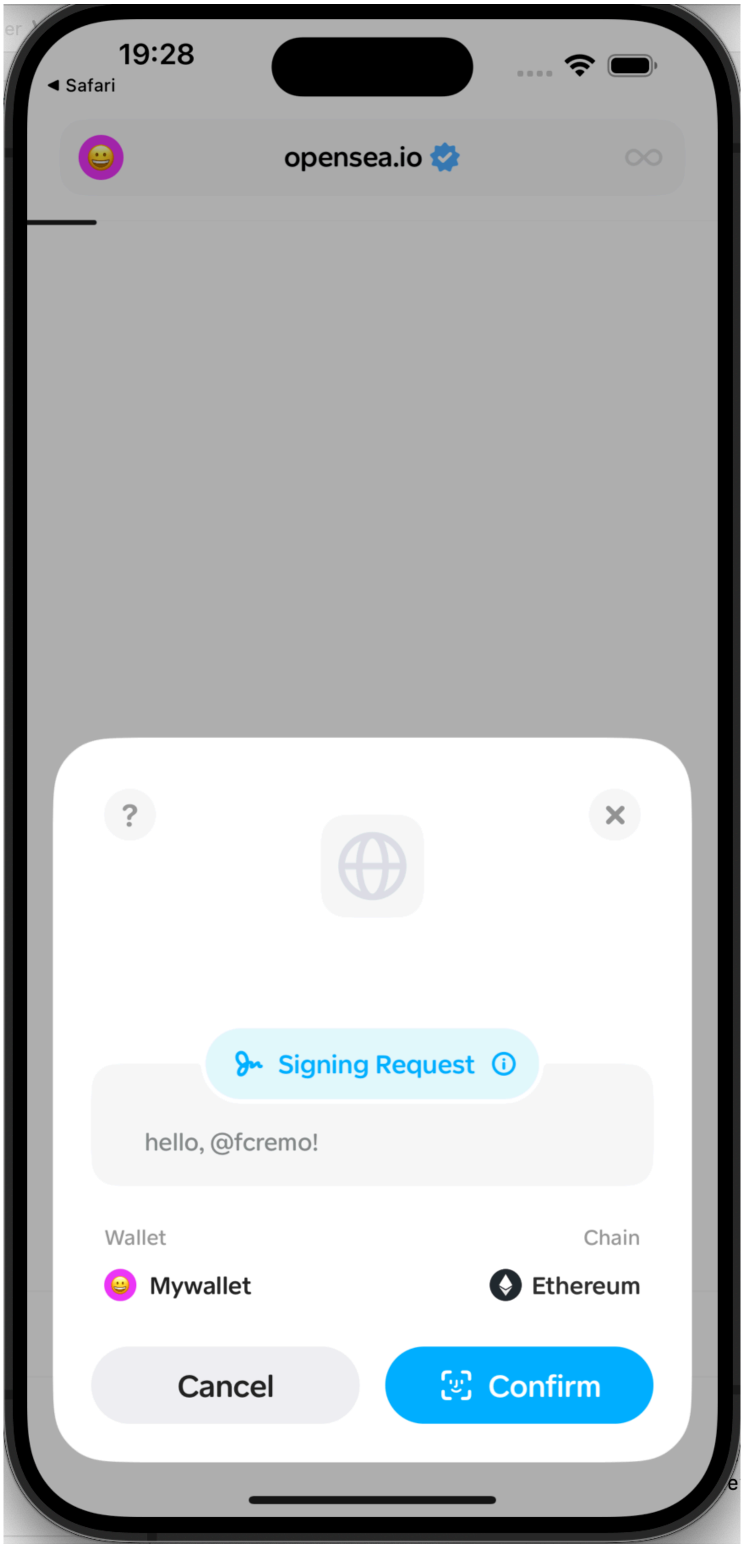
```
familywallet://browse?url=javascript:%2f%2fsomedapp.com%2f%250amalicious_code()
```

The following two screenshots show how a `javascript:` URL is misleadingly displayed as coming from an incorrect origin.





The JavaScript code can present a signature request that does not explicitly present the origin of the request; however, the URL bar will display a spoofed origin, and even though the code used to create the following screenshot does not attempt to recreate a legitimate dApp, it is possible to perfectly recreate the appearance of a legitimate dApp instead of a white page.



Signing Request

hello, @fcremo!

Wallet: Mywallet

Chain: Ethereum

Note that the blue checkmark is a consequence of Finding 3.3. ↗ and that abusing `javascript:` URLs is also applicable in the context of Finding 3.1. ↗.

Recommendations

Reject navigation to URLs that are not HTTPS or HTTP. Consider rejecting or asking user confirmation when navigating to HTTP URLs.

Remediation

This issue was remediated in the following commits:

- `238149578fe19dda3c904d30a651e8e9c1a9cbb1`

3.3. Improper URL handling with host suffix

Target	WCAppMappingsManager		
Category	Coding Mistakes	**Severity**	High
Likelihood	High	**Impact**	High

Description

The in-app embedded browser recognizes trusted dApps by downloading a list of well-known hosts. The `WCAppMappingsManager` class is responsible for matching the URL of the origin of an RPC request to a known dApp.

The match is done by first attempting to match the precise URL against a list of known dApp URLs. If this first check fails to find a match, all the existing known dApp hostnames are matched against the URL that originated the RPC request. We speculate this is done to recognize subdomains.

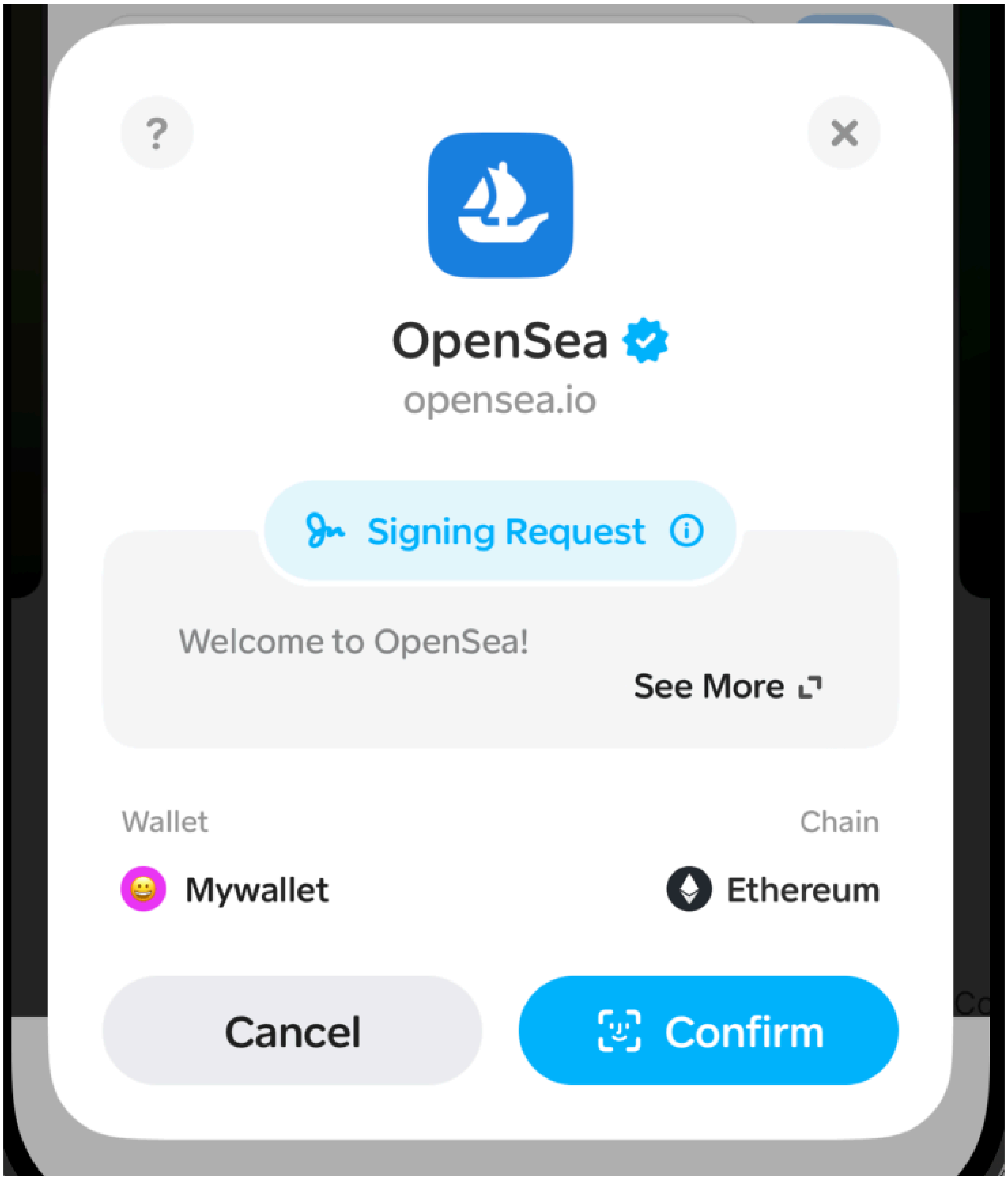
```
private func getMappedInfo(from url: BaseURI) -> (baseURI: BaseURI, mapping:
    WalletConnectAppMapping)? {
    return self.mappings.withLock { mappings in
        if let mappedInfo: WalletConnectAppMapping = mappings[url] {
            return (url, mappedInfo)
        } else {
            for key in mappings.keys {
                if url.value.hasSuffix(key.value), let mapping = mappings[key]
    {
                    return (key, mapping)
                }
            }

            return nil
        }
    }
}
```

This check can be bypassed by using a domain that *ends* like a known dApp URL. For instance, an attacker could spoof knowndapp.com by registering the domain attacker-knowndapp.com and tricking the user to visit the malicious domain.

Impact

This issue allows an attacker to spoof the origin of an RPC request. When the wallet recognizes a known dApp, it adds additional UI elements (e.g., a blue checkmark) that reassure the user about the origin of the request and automatically populates the dApp name, icon, and domain into confirmation screens.

The following screenshot shows what a spoofed RPC signing request would look like:



OpenSea

opensea.io

Signing Request

Welcome to OpenSea!

Recommendations

Ideally, match known dApps against a strict list of hostnames. Alternatively, prepend a period character . to the list of known hostnames. Note that this does not fully resolve the issue, as an attacker could leverage a subdomain takeover to spoof the origin of an RPC request. An attacker could also perform an MITM attack on a user and redirect them to a spoofed HTTP version of a legitimate dApp. Therefore, we also recommend to restrict known dApp matching to secure (HTTPS) contexts.

Remediation

This issue was remediated in the following commits:

- `238149578fe19dda3c904d30a651e8e9c1a9cbb1`

3.4. Suboptimal symmetric key derivation

Target	CryptographicEngine		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Family Wallet supports backup of private key material to iCloud. The app encrypts the private key material using CryptoKit ChaChaPoly, with a 256-bit symmetric key derived from a user password. The user password is required to pass some security requirements. [1]

The app supports three key derivation schemas:

- `v1` — the symmetric key is derived from the SHA-256 hash of the user password, converted to ASCII hex and truncated to the first 32 characters.
- `v2` — the symmetric key is derived from the raw binary SHA-256 hash of the user password.
- `v3` — the symmetric key is derived with PBKDF2-SHA256, using 750k rounds and a fixed salt.

When creating a new item, the app automatically uses the newest available schema; older schemas are supported for backwards compatibility purposes.

All three schemas have weaknesses and do not adhere to common best practices. Furthermore, there is no mechanism for reencrypting old backups that were created using an older and weaker scheme.

Impact

The `v1` and `v2` schemas are weak towards brute-force attacks, as they only consist of a single round of SHA-256. They are also vulnerable to rainbow tables, as there is no salt. The `v1` schema unnecessarily truncates the real entropy of the key to 128 bits, since the output of SHA-256 is encoded as ASCII hex and truncated from 64 to 32 bytes.

While the `v3` schema is more resistant to brute-force attacks, it is still vulnerable to rainbow-table attacks, because the salt is fixed.

[1] The user password must be at least 10 characters, include a number and a special character, or be at least 15 characters and include at least three different characters.

Recommendations

Implement a random salt for each symmetric key. Consider using a stronger key-derivation function such as Argon2id.

Remediation

This issue concerning a lack of randomized salt was remediated in the following commit:

- `9a31e844ed8c6fdecfe4bec00f4819e82f9496c9`

3.5. Broken LAContext `canEvaluatePolicy` authorization check

Target	LocalProtectedStore		
Category	Coding Mistakes	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

Family Wallet uses a check to `LAContext.canEvaluatePolicy(..)` to determine if a user presence check is needed for a given action. The check relies on `canEvaluatePolicy(..)` returning `true` if a device-authentication mode is configured. This check is used in multiple places in the codebase, namely the `LocalProtectedStore` and `RemoveWalletFlowViewController`.

A problem with relying on the return value of `canEvaluatePolicy(..)` to determine if a device-protection method has been configured on the device is that if that very same device auth method has been timed out via excessive attempts, then `canEvaluatePolicy(..)` will begin to return `false`. This introduces a bypass to this logic, allowing an attacker to get around this initial prompt to confirm user presence.

Depending on the circumstance, this can have minimal impact. For example, in the `LocalProtectedStore`, assuming that a secret was generated and stored in the keychain when the device had an auth method configured, accessing that entry should still require the user to validate the authorization requirements configured on the entry. However, this does not necessarily configure to entries stored in the keychain earlier if they were stored *before* a device authorization method was configured.

Notably, while `LocalProtectedStore` is partially protected via the keychain access control requirements configured on the keychain entries, `RemoveWalletFlowViewController` is not so lucky. Using this attack, an attacker is able to get around the control put in place to prevent inadvertant, or malicious wallet removal.

Impact

This flaw in trusting the output of `LAContext.canEvaluatePolicy(..)` to determine if authentication is configured on the device can allow access to *some* secrets, if generated and saved when an authentication method was previously not present. Additionally, it allows an attacker to bypass the user presence requirements when removing a wallet.

Recommendations

Consider checking the returned error value for lockout conditions to determine if an operation should or should not proceed.

Remediation

This issue was remediated in the following commits:

- `825723d0166e3ba02dd46900e582cbf82019d17b`
- `8e781c122d462efd068aba2d5ff10eca8a6f6c90`

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Private key material storage

Family Wallet stores private key material using the keychain APIs provided by iOS. The wallet supports storing private key material as a raw private key as well as using the mnemonic representation.

Access to the keychain entries is only allowed to Family Wallet, and other apps cannot access private key material. If a PIN code or password are configured on the device, the wallet app requires user presence to be established. The access control policy is instantiated as such in `LocalProtectedStore::getUserPresentAccessControl`.

```
do {
    // if this fails, we simply do not ask for access control
    // this will fail when a passcode is not set.
    try self.checkCanAuthenticateUserIsDeviceOwner(using: context)
} catch {
    return .success(nil)
}

var error: Unmanaged<CFError>?
var access: SecAccessControl?

access = SecAccessControlCreateWithFlags(nil, kSecAttrAccessibleWhenUnlocked,
    .userPresence, &error)
```

We note that `kSecAttrAccessibleWhenUnlocked` allows keychain entries to be backed up and transferred to other devices. This means backups of the keychain made (e.g., via iTunes) will contain the (encrypted) private key data. Family Wallet developers could consider using the `kSecAttrAccessibleWhenUnlockedThisDeviceOnly` access-control flag to prevent keychain entries from being able to be backed up and transferred to other devices.

This was addressed in the following commit:

- `20abd9a02bd9da1711f2fc87ca97be3f82790336`

4.2. Potential key leaks in screenshots

At certain times, such as when importing an existing wallet or when performing a manual backup, the wallet displays private keys or seed phrases. The app does not prevent screenshots from recording private key details. A user could, voluntarily or involuntarily, take a screenshot of the application. The screenshot would be saved to the gallery and be much more exposed.

Additionally, a screenshot of the application is automatically captured by iOS when the app is put in background (and used to display the app in the app switcher). This screenshot is stored on the device file system, and while not directly accessible on a nonjailbroken device, it does not have the same security guarantees offered to an item stored in the iOS keychain. The screenshot is also displayed by the app-switcher menu and can be seen with possession of the unlocked device.

It is possible to prevent screenshots of sensitive information by implementing information hiding in the `sceneDidEnterBackground` or `applicationDidEnterBackground` life cycle events.

This was addressed in the following commits:

- `e1b4c43ce0e8db013305c863f2127e264755a772`

4.3. Address and transactions privacy

We note that Family Wallet uses RPC servers managed by the developers to push transactions and obtain information about the network. Additionally, in order to perform some operations, the wallet needs to authenticate to the backend using information that includes the hash of the user wallet address. Devices perform a remote attestation using iOS DeviceCheck APIs in order to prove to the backend that a real iOS device is performing the requests.

This implies that the Family backend gets some information that could potentially be correlated to associate wallet addresses and transactions to the device and IP address that originated them. While this does not constitute a vulnerability in and of itself, it is important that users are aware of the potential privacy implications.

4.4. Sandboxed iframes

Preventing malicious JavaScript execution is critical to ensuring the security of the dApp browser context. For NFTs, which are untrusted SVGs rendered in a WKWebView, we recommend placing the untrusted content inside a sandboxed iframe. By leveraging a sandboxed iframe, browser-level controls are leveraged to prevent JavaScript execution. If JavaScript execution *is* required for a given NFT, the sandbox attribute allows configurable control over what is and is not allowed in the context. Additionally, by using a sandboxed iframe, the JavaScript executes in an isolated origin.

This was addressed with an alternative approach in the following commit:

- `1b9a002f39853a5f415cdead06f56aa1136dfd39`

4.5. Unsafe handling of text input

When handling sensitive information like seeds or private keys, additional care should be taken to obscure and handle the sensitive information. We recommend evaluating if `isSecureTextEntry` should be set on given `UITextInput`s. For example, it may make sense within the scope of the application to enable this setting on the `UITextView` within the `IWInputPrivateKeyViewController`.

The team accepted the risk here as it would have negative impacts on the user experience.

5. Threat Model

5.1. RPC handlers

This section briefly documents the RPC endpoints exposed to dApps. The RPC endpoints provide a subset of the standard Ethereum Provider interface also implemented by other wallets, making it easily interoperable.

`eth_chainId` and `net_version`

These two RPC methods simply return the currently selected chain ID. As they do not constitute sensitive information, they do not require any user permission.

`eth_accounts`

This only returns the currently selected account address (not the list of available accounts).

`eth_requestAccounts`

This returns the currently selected account and asks for authorization if it is not already granted.

`personal_sign`, `eth_sign`, `eth_signMessage`, `eth_signTypedData`, `eth_signTypedData_v1`, `eth_signTypedData_v3`, and `eth_signTypedData_v4`

These methods are handled by the same three classes.

- `BrowserSignMessageRPCMethodHandler` class; they are forwarded directly.
- `WCSignMessageFlowHelper` is a utility class; one of its responsibilities is parsing the message being signed to return a user-readable representation.
- `WCSignMessageRequestViewController` is the controller responsible for showing the user a confirmation screen with information about the data being signed. If the user confirms, it invokes a closure that in turn uses `WCSignMessageFlowHelper::runSigning` to compute the required signature.

`personal_ecRecover`

This handler implements `ecRecover` for EIP-712 signatures. The actual cryptographic operations are delegated to the third-party library function `HDWalletFactory.recoverPublicAddressRawValue`.

`wallet_requestPermissions`

This requests permission to interact with a wallet. The call chain is this:

- `handleRequestPermissions`

- `context.connection.getUserPermissionToInteractWithWalletIfNeeded()` -> `BrowserWebContext::Connection::getUserPermissionToInteractWithWalletIfNeeded`
- `self.root.getUserPermissionToInteractWithWalletIfNeeded(...)` -> `BrowserWebContext::_RootConnection::getUserPermissionToInteractWithWalletIfNeeded`

The call to `_RootConnection::getUserPermissionToInteractWithWalletIfNeeded` determines if explicit permission is required. We find the name slightly confusing, as what is happening is that the user is asked to select a wallet they want to interact with if they have previously selected a wallet that was not theirs (read-only). Technically, the user is prompted if the currently selected wallet does **not** have the `isPersonal` attribute.

`wallet_getPermissions`

This returns the permissions granted for the selected wallet. This endpoint is read-only and can only return `eth_accounts` if a wallet is selected or no permission at all.

`eth_sendTransaction`

This sends a transaction over the Ethereum network. Will open a new `WCTransactionRequestViewController` with the transaction details as requested.

The `from` address must match `context.connection.walletAddressToInteractWith`, which validates the network as well as the address.

`wallet_switchEthereumChain`

This allows to switch the working chain — only supports a hardcoded set of chain IDs (Ethereum, Polygon, Arbitrum, Optimism, Base). Notifies user via toast when switch occurs.

Relayed as-is RPC methods

The following method names are relayed as-is to an ETH RPC hosted on api.family.co:

- `eth_gasPrice`
- `eth_getBalance`
- `eth_getBlockByHash`
- `eth_getBlockByNumber`
- `eth_estimateGas`
- `eth_getBalance`
- `eth_call`
- `eth_blockNumber`
- `eth_getLogs`
- `web3_clientVersion`

- `eth_getTransactionCount`
- `eth_getTransactionByHash`

None of these APIs work with or expose sensitive data. The app is safe from injections that could allow a malicious dApp to cause the wallet to perform a request to an arbitrary host or path.

6. Assessment Results

At the time of our assessment, the reviewed code was not deployed to the Apple App Store.

During our assessment on the scoped Family Wallet codebase, we discovered five findings. No critical issues were found. Three findings were of high impact and two were of medium impact. Avara acknowledged all findings and implemented fixes.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.