Security Analysis of Electric Coin Company's Example Mobile Wallets

Taylor Hornby
taylor@electriccoin.co
Electric Coin Company

June 17, 2020

Contents

Executive Summary

Our review found no high-severity issues that were not already described as weaknesses in the threat model.

It is important to note that the threat model describes weaknesses that some users may consider to be high-severity vulnerabilities such as the fact that traffic analysis attacks can be used to learn *when* the user is sending and receiving transactions, and perhaps even learn *who* they are sending or receiving funds to or from. Also, if the `lightwalletd` server is compromised, then the attacker can make it look like the user received a payment when they in fact did not.

We strongly encourage the prioritization of efforts to fix the known weaknesses, especially ones that may not be obvious to users.

1 Introduction

This report documents the results of an approximately six-day security review of the Electric Coin Company's example mobile wallet apps.

The example apps are written natively for iOS and Android and are based on ECC's previously-released and improved wallet SDKs. These apps are distinguished from other wallet apps in the marketplace by their support for Zcash shielded transactions.

1.1 Architecture

The ECC example wallet apps are light wallets, meaning they do not download and validate the entire block chain. Unlike Bitcoin (SPV) light wallets, they do not connect to standard Zcash nodes, either. To operate, they must rely on `lightwalletd` software [6] instances to inform them of facts about the Zcash block chain, while retaining spend authority on the mobile device.

Each example wallet app is comprised of three components:

1. At the lowest level, the `librustzcash` library [5], written in Rust, provides the cryptography necessary for creating, validating, and processing shielded Zcash transactions. The relevant code has not yet been merged into `librustzcash`'s master branch, it presently exists in the `note-spending-v7` branch [7].
2. At the next higher level, the `librustzcash` library is wrapped by the SDKs, which are written independently for iOS [4] and Android [2]. The SDK han-

dles most of the functionality of the wallet, including communicating with the `lightwalletd` server.

3. The example app itself, which is a native app for either iOS [3] or Android [1], makes use of the corresponding SDK to implement a functional wallet.

This review only covers the third, highest-level, app component.

1.2 Scope

This review was limited to the two codebases for the iOS and Android apps [3, 1]. Notably, the review did *not* cover:

- The apps' dependencies, except for certain critical ones like the BIP39 libraries used to generate seed mnemonics.
- The SDKs. An older version of the SDKs have been reviewed, but we plan to review them again soon since new features (like memo support) have been added and that code has not yet been reviewed.
- The cryptography in `librustzcash`.

At the time of review, the latest commit hashes of the repositories were:

- zcash-android-wallet: `37a361ef1daf1149100b6d2f465e18c5c2855be`
- zcash-ios-wallet: `80d5f80b8723c147dafb27857fd72df2ad51c609`
- zcash-android-wallet-sdk: `812e51b0f8b8549f3cfe7f24fa0ae2d0f6849159`
- ZcashLightClientKit: `15e9a95ccb94d39f7e8d4e38636363377d6adf83`

1.3 Threat Model

The ECC example wallet apps have a detailed threat model [8]. The threat model explains which security properties users are able to rely on given varying assumptions about the adversary trying to attack them.

There are several known weaknesses described in the threat model. The ones we expect users to find the most counter-intuitive are:

- As part of the protocol for sending and receiving transactions, the `lightwalletd` server learns information about the user's use of their wallet such as when

they are sending or receiving shielded transactions, and in some cases, even *who* they are sending or receiving shielded transactions to or from.

- An adversary observing the user's or `lightwalletd`'s network traffic could learn the same information by exploiting side-channels in the amount of bandwidth the wallets use. The acts of sending and receiving transactions have recognizable traffic patterns, even though the connection is encrypted.
- If the `lightwalletd` server is compromised or malicious, it can violate the integrity of the transaction and balance information that the wallet displays. The server should not be able to directly steal funds from the wallet, but it could make it appear to the user as though they received a payment when they in fact did not.

See the full threat model for a more nuanced description of the wallets' security properties.

2 Findings

Our review found several weaknesses in the apps that were not anticipated by the threat model. We describe those findings in this section.

For each one, we rate its severity as either High, Medium, Low, or Informational. We also rate each issue's exploitability as either Easy, Medium, or Hard. The exploitability rating captures how likely the issue is to be exploited by an attacker, and the severity rating describes how harmful the consequences of exploitation would be to the user.

Please see the GitHub tickets corresponding to each issue for updates on how the issues have been resolved.

2.1 Official-looking text can be controlled through the memo field

Severity: Medium

Exploitability: Easy

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/127`

The Android app contains the following code to parse a "sent from" address out of the Zcash memo field.

```
const val INCLUDE_MEMO_PREFIX = "sent from"
...
memo.contains(INCLUDE_MEMO_PREFIX) -> {
    val address = memo.split(INCLUDE_MEMO_PREFIX)[1].trim()
    "${address.toAbbreviatedAddress()} paid you"
}
```

There is no code to validate that the "address" that gets parsed out is actually a valid Zcash address. Any user of the app could therefore include a memo with text such as "sent from Zooko", and the recipient's app will state "Zooko paid you..." with no indication that the "Zooko" text was user-created or untrustworthy. This could be exploited in social engineering attacks, including being used for phishing by placing fake warnings or URLs in the official-looking text. An example of what this looks like is given in Figure 1.

This problem can be fixed by checking if the string that was extracted is a valid Zcash address.

2.2 Vanity addresses can be used to control official-looking text

Severity: Low

Exploitability: Medium

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/132`

Once the previous finding is resolved by validating the address in the memo field, there is still a problem that remains. The interface elides the middle part of the address, leaving only a small number of characters from the address on either side of the ellipses. It may be possible for an attacker to brute-force an address that passes the validity checks, but still contains text that might confuse the user. This could be done using the same tools that exist to produce vanity addresses.

This problem is partially mitigated by the fact that Zcash addresses must begin with either "z", "zc", "zs", or "t". It could be mitigated even further by displaying more characters from the address, since it would increase the computational effort required to create addresses that look like text.

2.3 The UI expresses too much confidence in "sent from" addresses

Severity: Low

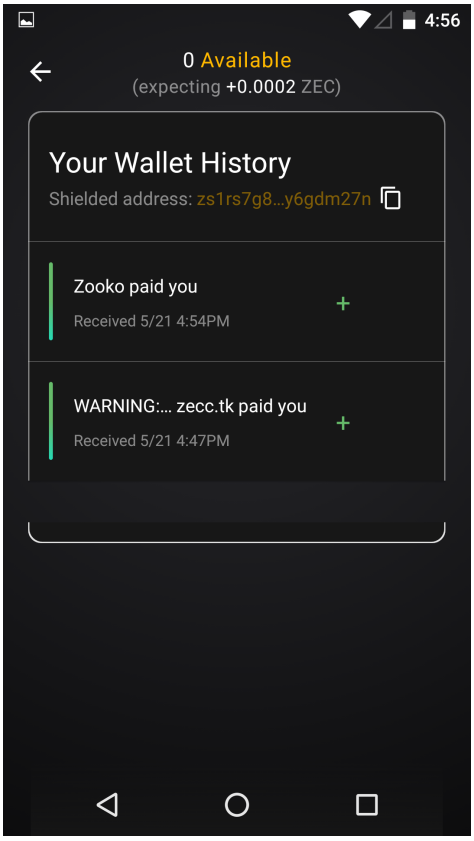

Figure 1: The victim's wallet UI while being targeted by Finding 2.1.

Exploitability: Easy

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/126`

The wallet's UI displays the "sent from" address that was extracted from a memo as though it were a fact. Because the Zcash protocol provides anonymity to the transaction sender, there is no way to verify that the address included in the memo is the real address of the sender, at least not without also having the sender include a cryptographic signature or viewing key too.

The sender can lie by including a different address, which might confuse users, and must be considered if an address book functionality (automatic mapping from addresses to human names) is ever added in the future.

Fix this by informing the user through the UI that the address may not be the

sender's real address.

2.4 Adding reply-to addresses to memos might lose information

Severity: Informational

Exploitability: N/A

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/98`

Zcash memos have a maximum length. The iOS wallet includes the following code to truncate the memo field when it adds a reply-to address.

```
static func includeReplyTo(address: String, in memo: String, \
    charLimit: Int = SendFlowEnvironment.maxMemoLength) -> String {

    let replyTo = "\nfrom \(address)"

    if (memo.count + replyTo.count) >= charLimit {
        let truncatedMemo = String(memo[memo.startIndex ..< memo.index(\
            memo.startIndex, offsetBy: (memo.count - replyTo.count))])

        return truncatedMemo + replyTo
    }
    return memo + replyTo

}
```

The user may have put important information at the end of their memo, and this code would silently truncate it, leading to information loss. Fix this by making sure the user is informed when their memo was truncated, or by disabling the option to add a reply-to address if the memo would be too long.

2.5 Seed phrase verification is not implemented or enforced

Severity: Medium

Exploitability: N/A

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/125`

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/107`

Both the iOS and Android wallets will let the user send funds to their wallet before proving that they have made a backup of their seed phrase. Anecdotally, accidental loss of funds from funding a wallet without backing up the seed is quite common, so this should be enforced.

It should *not* happen on the first launch of the wallet, since it would encourage users who are in a rush to do something insecure like take a screenshot or save it in a file on their computer. Instead, users should be prompted to back up their wallet either before they first add funds, or once their balance crosses a threshold.

2.6 Seed phrase checksum is not validated in the Android app

Severity: High

Exploitability: N/A

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/120`

The wallets use BIP39 seed phrases as a convenient backup solution for the wallet. Some of the words in a BIP39 seed phrase are used to make a checksum, guarding against typos or accidental transposition of words during the restoration process.

The Android app does not validate the checksum in the seed phrase. A user might make an error when restoring their wallet, such as transposing two words. They might add funds to the erroneously-restored wallet without realizing they made the error. Then, if they ever restore their wallet again they will find that the funds are missing, and will not be able to recover them unless they can remember exactly which error they made.

This has already been fixed by re-implementing BIP39 with unit tests that make sure the checksum is validated.

2.7 Users are not informed that Crashlytics collects information

Severity: Medium

Exploitability: Hard

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/122`

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/106`

Both apps contain code for submitting bug reports and analytics to the Crashlytics service. Although credentials are required to access the reports, the information included in them could compromise users' privacy. No keys or other memory contents are included in these reports.

The user must be informed whenever crash reporting or analytics is enabled, so that they can consent to the collection of that information.

2.8 Unspendable balance from extreme amounts of dust

Severity: Medium

Exploitability: Hard

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/130`

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/108`

Ticket: `https://github.com/zcash/zcash/issues/4538`

The following code from the Android app assumes that it only costs one miners' fee to spend the remaining balance:

```
private fun onBalanceUpdated(balance: WalletBalance) {
    binding.textLayoutAmount.helperText =
        "You have ${balance.availableZatoshi.coerceAtLeast(0L)\
            .convertZatoshiToZecString(8)} available"
    maxZatoshi = (balance.availableZatoshi - ZcashSdk.MINERS_FEE_ZATOSHI)\
        .coerceAtLeast(0L)
}
```

It is possible for an attacker to create a situation where the wallet displays "You have 100 ZEC available" but much less than that is actually spendable.

The attacker can do this by actually sending the wallet 100 ZEC for real, but broken into extremely small-valued Sapling outputs. In the transaction that spends the outputs, there will have to be a Sapling input corresponding to each output. There can only be so many inputs before they fill up an entire block, so at minimum the recipient would need to pay a miners' fee for every block worth of inputs. The attacker could make the fees take up nearly all the value so that practically none of it is spendable, but they would have to fill up many blocks themselves to do so.

The attacker does not need to pay fees, because they don't care about their transactions looking the same as all others, at least as long as miners are willing to accept zero-fee transactions.

A generalized restatement of this attack is: By filling around N blocks with sapling inputs destined to a wallet, an attacker can send the wallet an arbitrary balance that requires around N blocks to completely spend. This might matter in situations where users expect their funds to be immediately available (e.g. for trading).

This is also an issue in `zcashd`. One mitigation is to enforce a minimum dust threshold for every output in the Zcash protocol itself [9], but that would not prevent the general attack. Another mitigation is note merging [10], where wallets could merge notes in the background so that all of the user's funds are spendable immediately.

2.9 Seed generation timing may leak information about the seed

Severity: High

Exploitability: Likely Impossible

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/117`

In the Android app, when the wallet is first created, a feedback system is used to measure the time certain steps take:

```
feedback.measure(wallet_created) { mnemonics.run {
    feedback.measure(entropy_created) { nextentropy() }.let { entropy ->
        feedback.measure(seed_phrase_created) { nextmnemonic(entropy) }
            .let { seedphrase ->
                feedback.measure(seed_created) {
                    toseed(seedphrase)
                }.let {
                bip39seed ->

                    lockbox.setcharsutf8(lockboxkey.seed_phrase, seedphrase)
                    lockbox.setboolean(lockboxkey.has_seed_phrase, true)

                    lockbox.setbytes(lockboxkey.seed, bip39seed)
                    lockbox.setboolean(lockboxkey.has_seed, true)
```

```
                    bip39seed
                }
            }
    }
} }
```

The conversion from the entropy into the seed phrase involves indexing into the wordlist array, where the index is a secret value (part of the entropy). The precise amount of time this takes depends on the secret index and other things like the state of the processor's cache, so it is possible for information about the seed to be leaked out through these timing measurements. For example, if the seed phrase happens to contain the same word twice, it will be slightly faster, because processing the first occurrence will load the word into the CPU's cache and then processing the second occurrence will be fast because the word is already in the cache.

It is extremely unlikely that this would ever be exploitable in practice; it would take millions of runs of this code starting with varying CPU cache states to actually extract seed words from the timing information. This problem is completely mitigated by the fact that the code only runs once for a given seed.

2.10 Null characters truncate strings between Swift and `librustzcash`

Severity: Low

Exploitability: Easy

Ticket: `https://github.com/zcash/ZcashLightClientKit/issues/130`

The following code in `ZcashRustBackend.swift` in the iOS app exposes a `librustzcash` function for validating a Zcash address:

```
static func isValidShieldedAddress(_ address: String) throws -> Bool {
    guard zcashlc_is_valid_shielded_address([CChar](address.utf8CString)) \
    else {
        if let error = lastError() {
            throw error
        }
        return false
    }
    return true
```

```
}
```

It is possible for the address string passed into the function to contain null characters, which are included in the output of `address.utf8CString`. The Rust layer will attempt to build a `CStr` out of a pointer to the first character, which is done by scanning until the first occurrence of a null byte. If the string coming from Swift contains a null character, then the string that `librustzcash` receives will be truncated.

Invalid addresses that are created by taking a valid address and appending a null byte plus more data will erroneously appear to be valid.

Several of the other wrapper functions that pass strings on to `librusthzcash` suffer from the same problem. Fix this by explicitly handling the case where the incoming string contains a null byte, or by avoiding the use of C-style strings in the translation between Swift and `librustzcash`.

3 Non-Bugfix Recommendations

3.1 Test TLS certificate validation

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/131`

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/109`

Because the wallets trust the `lightwalletd` server to provide accurate transaction information, and they disclose to the `lightwalletd` server which transactions are theirs, it is essential for the connection between the wallet and `lightwalletd` to be encrypted and authenticated with TLS.

Testing that the wallets reject invalid TLS certificates must either be part of the wallets' automated test suite or must be done manually prior to each release.

3.2 Create a system for detecting outdated dependencies

Ticket: `https://github.com/zcash/zcash-android-wallet/issues/116`

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/103`

Vulnerabilities in the dependencies the apps use could lead to vulnerabilities in the apps themselves. It is important to recognize when dependencies become out of date and upgrade quickly, especially if there are known security vulnerabilities

in the old versions. We recommend implementing a system, possibly integrated into the CI, for detecting and alerting when dependencies go out of date.

3.3 Allow the app to crash on unexpected conditions

Ticket: `https://github.com/zcash/zcash-ios-wallet/issues/129`

We found at least one instance in the Android code where a critical error is ignored and the app is allowed to continue running in a potentially-invalid state. In `HomeViewModel.kt`,

```
suspend fun refreshBalance() {
    try {
        (synchronizer as SdkSynchronizer).refreshBalance()
    } catch (e: RustLayerException.BalanceException) {
        twig("Balance refresh failed. This is probably caused by a critical \
         error but we'll give the app a chance to try to recover.")
    }
}
```

As a general policy, it is preferable for the app to stop running whenever a critical error is detected. This will be inconvenient for users, but it makes it more likely for the errors to be reported back to the developers so that they can be fully investigated, and it prevents the app from ever continuing to run in a potentially dangerous state.

3.4 Code improvements to ease security analysis

While reviewing the code we made several minor recommendations for how the code could be improved to save security auditors' time. These have been filed as issues on GitHub:

`https://github.com/zcash/zcash-ios-wallet/issues/100`—SeedManager should handle a missing seed itself instead of relying on librustzcash.

`https://github.com/zcash/zcash-ios-wallet/issues/104`—Add test vector tests for BIP39 randomness to seed phrase.

`https://github.com/zcash/zcash-ios-wallet/issues/105`—Make toSeed validate the seed phrase first.

`https://github.com/zcash/zcash-android-wallet/issues/118`—Make LockBox more robust to accidental mistakes.

`https://github.com/zcash/zcash-android-wallet/issues/128`—Remove apparent logging of characters typed.

`https://github.com/zcash/zcash-android-wallet/issues/121`—Possible shell injection in `writeLogcat()` (not exploitable).

4 Conclusions

Our review found no high severity vulnerabilities, and in particular no problems that could lead to direct theft of funds. However, we caution that some users may assume the apps provide more privacy guarantees than are established in the threat model, and could unknowingly use the apps in ways that put themselves at risk. We feel it is important to either prioritize fixing the privacy leaks in the light wallet protocol or run user studies to confirm that users understand and consent to the limitations.

References

[1] zcash-android-wallet.
https://github.com/zcash/zcash-android-wallet, .

[2] zcash-android-wallet-sdk.
https://github.com/zcash/zcash-android-wallet-sdk, .

[3] zcash-ios-wallet.
https://github.com/zcash/zcash-ios-wallet, .

[4] ZcashLightClientKit.
https://github.com/zcash/ZcashLightClientKit, .

[5] librustzcash.
https://github.com/zcash/librustzcash.

[6] lightwalletd.
https://github.com/zcash/lightwalletd.

[7] librustzcash note-spending-v7.
https://github.com/str4d/librustzcash/tree/note-spending-v7.

[8] Wallet App Threat Model.
https://zcash.readthedocs.io/en/latest/rtd_pages/wallet_threat_model.html.

[9] Zcash Issue #4112.
https://github.com/zcash/zcash/issues/4112, .

[10] Zcash Issue #4332.
https://github.com/zcash/zcash/issues/4332, .