`zecwallet-lite-cli` Security and Privacy Analysis

Taylor Hornby
zecsec@defuse.ca

November 29, 2022
Report Version 2
Audit performed in November 2022

Contents

1 Introduction

This report documents a 9-day security review of `zecwallet-lite-cli` [7] that was performed for the Zcash Ecosystem Security grant [6]. `zecwallet-lite-cli` is both a command-line light client for Zcash as well as the library for block processing, scanning, and transaction-building underlying the Zecwallet Lite wallet [8]. Zecwallet Lite itself will be the subject of a future audit.

`zecwallet-lite-cli` connects to a modified version of the `lightwalletd` server. We also reviewed those modifications as part of this audit.

1.1 Scope

This audit covered the following four repositories and git commits:

- `adityapk00/zecwallet-lite-cli` - fd5d11f0f28e0f1628ea8fdad0cce16d50e6bb98
- `aditapk00/librustzcash` - adf49f8b848a5ac85e1476354614eeae9880206a
- `aditapk00/orchard` - 0a960a380f4e9c3472c9260f3df61cd5e50d51b0
- `aditapk00/lightwalletd` - a67fcecc7d0e91aa3ad4d8b154d532652ce1381f

The code in `zecwallet-lite-cli` was manually reviewed in its entirety. For the latter three repositories, we only reviewed the differences between the versions used for `zecwallet-lite-cli` and Electric Coin Co's upstream versions.

Regarding `zecwallet-lite-cli`, `librustzcash`, and `orchard`, we were most interested in finding bugs that put users' funds at risk or impacted users' privacy. For `lightwalletd`, we were most interested in finding bugs that would allow a remote attacker to compromise the light client server, deny service to wallet users, or that would leak information about users' activities.

Beyond the modifications to `librustzcash` and `orchard`, we did not spend a significant amount of time analyzing dependencies. The greatest risk from dependencies comes from the cryptography in `librustzcash`, which has already passed Electric Coin Co's strict internal review process and/or has been reviewed by third-party audits, so we feel this is safe to defer.

The following areas were explicitly not in scope of this audit:

- Review of dependencies.
- Multithreading issues (race conditions) in `zecwallet-lite-cli`; see future work below.
- The upstream contents of `librustzcash`, `orchard`, and `lightwalletd`.

1.2 Threat Model

A detailed threat model for Zcash shielded wallet apps is available online [4]. The security and privacy issues documented in that threat model also apply to `zecwallet-lite-cli`; we do not repeat them in this report. Any security and privacy issues not already documented there are considered bugs and are reported in this document.

2 Security & Privacy Findings

This section describes the security and privacy issues that were found.

The severity of each issue is graded to aid with prioritization. A rating of "*Critical*" means a critical vulnerability that can definitely be exploited to impact many users. "*High*" means a vulnerability that may have a severe impact for many users. "*Medium*" means a vulnerability of lesser impact or one that may only be exploitable in special circumstances. "*Low*" means a vulnerability whose exploitation would have very little impact on any user or which is unlikely to ever be exploited in practice.

2.1 Auto-shielding makes de-anonymization attacks possible

Severity: High

When generating a transaction, `zecwallet-lite-cli` preferentially spends users' transparent funds:

```
async fn select_notes_and_utxos(
    &self,
    target_amount: Amount,
    transparent_only: bool,
    prefer_orchard: bool,
) -> (Vec<SpendableOrchardNote>, Vec<SpendableSaplingNote>, Vec<Utxo>, Amount) {

    ...

    // First, we pick all the transparent values, which allows the auto shielding
    let utxos = self
        .get_utxos()
        .await
        .iter()
        .filter(|utxo| utxo.unconfirmed_spent.is_none() && utxo.spent.is_none())
        .map(|utxo| utxo.clone())
        .collect::<Vec<_>>();
    ...

    let mut remaining_amount = target_amount - transparent_value_selected;

    ...
    o_notes = self.select_orchard_notes(remaining_amount.unwrap()).await;
```

This is done in order to shield transparent funds as quickly as possible. Unfortunately, it creates a severe privacy leak that can be exploited in two ways.

Firstly, as long as a user has transparent funds in their wallet, all of their outgoing transactions will be "tagged" as originating from them, since they will all spend funds from the same transparent address. If the user intends to send an anonymous shielded transaction to a z-addr, their privacy will be lost since their recipient and all other blockchain observers will see their t-addr.

Secondly, an attacker (a blockchain analytics company, for example) can use an active attack to track which Zecwallet Lite wallets are generating which transactions. This can be done by the following process:

1. Identify all transactions on the blockchain that "look like" Zecwallet Lite autoshielding: transactions with shielded outputs that also spend transparent funds.
2. Periodically send a small amount of ZEC to each t-addr implicated in the above transactions to ensure that they always have nonzero balance.
3. Now, each transaction generated by the vulnerable wallets will be "tagged" with their t-addr because there will always be funds to auto-shield. The attacker learns which wallets are creating which transactions.

Transaction sources are already disclosed to the `lightwalletd` server, and this is mentioned in the current threat model. This attack exposes the transaction source to anyone who can download a copy of the blockchain. The attack is noisy; many users would likely notice the unexpected transparent funds arriving at their wallet and raise the alarm that someone is exploiting this weakness.

To fix this, modify the auto-shielding behavior so that no transparent funds are ever spent in transactions to shielded addresses. Instead, periodically auto-shield transparent funds using a dedicated, separate transaction. This is the approach that Electric Coin Co's wallets use. Using this approach, some information is still leaked through timing correlations (e.g. when a user generates a shielded transaction shortly after their wallet auto-shielded), but it leaks less information with less certainty than the current approach.

2.2 UI encourages passing seed phrase as a command-line argument

Severity: Medium

To restore a wallet from a seed phrase, `zecwallet-lite-cli` must be invoked with the `--seed` switch, providing the seed as a command-line argument. This exposes the users' seed in two ways. Firstly, the seed phrase will be entered into their shell's history, making it available to malware and negating the purpose of the wallet encryption. Secondly, the seed phrase will be visible to other (unprivileged) users on the same system, by looking in `/proc/*/cmdline`. On a shared system, i.e. one that can be SSHed into by other people as in a university lab, the user's funds can be stolen.

The same problem exists with providing the the wallet encryption password through the `--password` switch; the password gets saved to the user's shell history and is disclosed to other users of the system.

To fix this, remove the `--seed` and `--password` switches and replace them with a `--restore-from-seed` switch that takes zero arguments and interactively prompts the user to enter the seed and (optional) wallet password over standard input (stdin).

2.3 Locking the wallet does not clear the Orchard keys

Severity: Medium

When an encrypted wallet is locked, the plaintext keys must be deleted from the wallet's data structure or else they will remain accessible in memory and in the saved wallet file without any need to know the wallet's password.

In `src/lightwallet/keys.rs`, the `lock()` function is responsible for doing this. It correctly clears the seed, transparent address keys, and Sapling keys, but it does not clear the Orchard keys:

```
// Empty the seed and the secret keys
self.seed.copy_from_slice(&[0u8; 32]);

// Remove all the private key from the zkeys and tkeys
self.tkeys
    .iter_mut()
    .map(|tk| tk.lock())
    .collect::<io::Result<Vec<_>>>()?;

self.zkeys
    .iter_mut()
    .map(|zk| zk.lock())
    .collect::<io::Result<Vec<_>>>()?;
```

The orchard keys are stored in the `okeys` variable, which is not cleared:

```
// Unified address (Orchard) keys actually in this wallet.
// If wallet is locked, only viewing keys are present.
pub(crate) okeys: Vec<WalletOKey>,
```

As a result, even when the wallet is locked, the plaintext Orchard keys will be written to the wallet on disk:

```
pub fn write<W: Write>(&self, mut writer: W) -> io::Result<()> {
    ...
    // Write all orchard keys
    Vector::write(&mut writer, &self.okeys, |w, ok| ok.write(w))?;
```

If an attacker gains access to a user's encrypted wallet, their Orchard funds can be stolen without needing to guess their password.

To prevent this kind of bug in the future, we recommend storing all of the plaintext keys under a single `plaintext_keys` object, which can be cleared all in one operation, and would not be written in case the wallet is locked (`unlocked == false`).

A feature should be added to detect encrypted wallets that have had their Orchard keys leaked, warning the user to move any vulnerable funds to new Orchard addresses.

2.4 Wallet encryption uses an improper password hashing function

Severity: Low

To derive an encryption key from a user-provided password, best practice is to use a memory-hard key stretching function like Argon2 or Scrypt. `zecwallet-lite-cli` uses a double application of SHA256 to the password to derive the encryption key for locked wallets. As a result, a brute-force attack against the wallet encryption is much faster than it would be if a proper password hashing function were used.

It is also best practice to provide a random "salt" to the password hashing function to prevent amortized brute-force attacks (i.e. rainbow tables). Even though it does not use a salt, `zecwallet-lite-cli`'s wallet encryption is thankfully not vulnerable to these kinds of attacks because the derived key is immediately used with a fresh, random Salsa20 nonce.

We recommend instead using the `argon2` crate to derive the key using Argon2, with a random salt that gets saved with the ciphertext.

2.5 Wallet encryption is malleable

Severity: Low

The wallet encryption in `zecwallet-lite-cli` encrypts each key individually using the XSalsa20Poly1305 algorithm. The ciphertexts that result are non-malleable in the sense that an attacker wishing to modify their contents needs to know the key to do so. However, since the encrypted wallet is composed of many individual ciphertexts, an attacker with access to modify the encrypted wallet is free to reorder the ciphertexts, potentially making the wallet use incorrect addresses/keys when it is later decrypted.

This is only a risk if users are storing their encrypted wallet somewhere where an attacker could modify it, such as in a cloud backup. Even then, there is probably not much an attacker can do to affect the user by reordering ciphertexts, so it is probably okay to leave the current implementation as-is.

To definitively remove this risk, we recommend encrypting the wallet by serializing all of the data into a single string and then encrypting that string.

2.6 `cargo audit` reports vulnerabilities in dependencies

Severity: Low

Running `cargo audit` in `zecwallet-lite-cli` finds the use of several unmaintained crates and that the version of `time` used contains a known vulnerability.

The vulnerability in `time` cannot be used to exploit `zecwallet-lite-cli` users, but it should be updated anyway.

It is extremely difficult to avoid using unmaintained Rust crates, so most of these can be ignored. The only slightly concerning one is that `sodiumoxide` (used for wallet encryption) is unmaintained and deprecated. Another compatible library should be used, such as RustCrypto's XSalsa20Poly1305:

`https://github.com/RustCrypto/AEADs/tree/master/xsalsa20poly1305`

2.7 `lightwalletd` logs all client IPs in calls to GetBlockRange

Severity: Medium

In `frontend/service.go` of `lightwalletd`, there is code in the `GetBlockRange` function that logs all clients' IPs as they call `GetBlockRange`. In a previous version of the code, these IPs were only logged every 1152 blocks, corresponding to 24 hours, allowing the number of active users at a point in time to be estimated.

If the logs generated by the current version of the code are saved indefinitely, this makes the server an attractive target for attackers wishing to de-anonymize wallets. The logs include the client IP and range of blocks requested. The timing of these logged requests can be correlated with other activity, such as transactions appearing in the mempool, to de-anonymize the origin of transactions. This increases the likelihood that someone will want to break into the `lightwalletd` server.

We recommend not logging client IPs at all, or if measuring daily active users is necessary, then limiting the logging as in the old version of the code, and allowing the user to opt-in or opt-out of the logging through a setting in their wallet.

2.8 `lightwalletd`'s Prometheus endpoint may be publicly exposed

Severity: Low

The modified `lightwalletd` exposes a Prometheus endpoint which will make various metrics such as number of blocks served, error counts, and counts of calls to various `lightwalletd` APIs accessible to monitoring software.

In the `lightwalletd` code, it appears as though the endpoint will be exposed, accessible to the public Internet. We tried to access this endpoint on the live servers used by `zecwallet-lite-cli`, and we could not, so we believe there are additional defenses in place protecting access to this endpoint.

It seems likely that other users of this version of `lightwalletd` would forget to add these defenses, so we recommend adding an authentication check to the `lightwalletd` code itself. A simple IP filter or HTTP Basic Auth with a random password loaded from a configuration file would be sufficient.

3 Recommendations

3.1 Rebase forked repositories on upstream main branches for easier review

It was difficult to review the modifications made in `adityapk00/librustzcash`, `adityapk00/orchard`, and `adityapk00/lightwalletd` because the changes are not cleanly applied on top of the upstream versions and there are merge commits from the upstream repos back into the forks. It would significantly speed up future audits of these forks to rebase the changes on top of the current upstream `main` branches.

Doing this will also pull in any bug fixes that have been made in the upstream repos since the last merge.

3.2 Retain ECC's privacy warnings in the `lightwalletd` README

`adityapk00/lightwalletd`'s README is missing the privacy warnings that exist in the upstream `lightwalletd`'s README. These are worth including to make the security audit status of the repository clear to potential users.

3.3 More integration testing for reorg edge-cases

One of the trickiest parts of writing a Zcash wallet is state management in the face of edge cases like reorgs. If old state is leftover after a reorg, for example, a user may think they have a spendable note when in fact they do not.

Test cases for such scenarios have been written for Electric Coin Co's wallets, using the "darksidewalletd" feature of lightwalletd. Darksidewalletd allows the test-writer to produce arbitrary blocks and trigger reorgs arbitrarily in order to test scenarios that are possible but only occur rarely on mainnet and testnet.

We recommend implementing the same style of tests in `zecwallet-lite-cli` as a defense against state-management bugs. This should be a high priority for future funding and development.

Note that `zecwallet-lite-cli` already has unit tests for some of these edge cases. It would still be valuable to have integration tests of the entire wallet's state management stack.

3.4 Miscellaneous test improvements

In `zecwallet-lite-cli`, it would be worthwhile to augment the `lightclient/tests.rs` tests to test address derivation with more than one index. This will ensure that the key derivation paths are implemented correctly.

4 Good Things

4.1 Clear privacy warnings in the README

`zecwallet-lite-cli` has good privacy warnings in its README, for example listing the kinds of information that the `lightwalletd` server learns. This is an excellent practice as it informs users of the wallet and developers who want to build on the library.

4.2 Using certificate authority pinning

`zecwallet-lite-cli` hard-codes the Let's Encrypt root certificate and uses it exclusively to authenticate connections to `lightwalletd`. This significantly reduces the risk of the connection becoming compromised, since no other (potentially compromised) certificate authority can be used to authenticate the connection.

4.3 Thorough unit testing

Almost all of the files in `zecwallet-lite-cli` include unit tests, and many of them are quite extensive. While we recommended more integration testing of the state management in the face of edge cases like reorgs, these unit tests are likely to catch many kinds of bugs at the individual component level. The author(s) of `zecwallet-lite-cli` have clearly invested significantly in testing, which is encouraging to see.

5 Future work

5.1 Analysis against a malicious light wallet server

Currently, Zcash wallets operate in a model where the light wallet server is assumed to be honest. In the future, wallets should remove this assumption by validating block headers and hashes of the information they receive (such as notes and nullifiers). If and when Zcash wallets move to a model where the light wallet server is assumed to be malicious, more security analysis should be done on `zecwallet-lite-cli` to make sure it is secure in the face of a malicious server, for example making sure that the adversary cannot overcome the integrity checks, brick the wallet, or otherwise attack the user.

5.2 More review of potential multithreading race condition issues

Rather than using a transactional database to serialize modifications to the wallet's state, `zecwallet-lite-cli` uses in-memory structures that are periodically written to disk. `zecwallet-lite-cli` implements its own protections against race conditions using mutexes. While the Rust language ensures that there cannot be conflicting access to data at an individual object level, there is still a potential for race conditions that would leave the wallet in an inconsistent state. A weakness of this audit is that we did not spend a lot of time looking for such issues, so we recommend that future audits focus more on these kinds of bugs.

6 Conclusion

In conclusion, we found no bugs in the modifications made to `orchard` or `librustzcash` and we found minor privacy concerns with the modifications to `lightwalletd`. In `zecwallet-lite-cli`, we found a high-severity privacy leak in the autoshielding behavior, along with various cryptographic weaknesses in the wallet encryption. Our main recommendation, as for all Zcash wallets, is to strengthen the integration testing of the state management in the face of edge cases like reorgs. `zecwallet-lite-cli` already has a sufficient amount of unit testing.

7 Acknowledgements

This security analysis was performed as part of the Zcash Ecosystem Security Grant [6] funded by Zcash Community Grants [5]. Thanks to the Zcash Grants Committee and the broader Zcash community for supporting the security of privacy-enhancing open-source software.

References

[1] adityapk00/librustzcash.
https://github.com/adityapk00/librustzcash.

[2] adityapk00/lightwalletd.
https://github.com/adityapk00/lightwalletd.

[3] adityapk00/orchard.
https://github.com/adityapk00/orchard.

[4] Zcash wallet app threat model.
https://zcash.readthedocs.io/en/latest/rtd_pages/wallet_threat_model.html.

[5] Zcash community grants.
https://zcashcommunitygrants.org.

[6] Zcash ecosystem security grant.
https://forum.zcashcommunity.com/t/zcash-ecosystem-security-lead/42090
https://zecsec.com, .

[7] adityapk00/zecwallet-lite-cli.
https://github.com/adityapk00/zecwallet-light-cli, .

[8] Zecwallet-lite.
https://zecwallet.co/, .