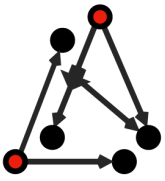
Sapling Implementation / RPC Interface Review

Security Audit Report

Zcash

Report Version: 30 January 2019



Least Authority

Freedom Matters

Table of Contents

Overview

Zcash has requested that Least Authority perform a security audit of changes and updates made to the second implementation of Sapling and to the RPC interface. This second Sapling implementation analysis and code review is the last in our 2018 series and incorporates both past and new concerns. This review follows the most recent Sapling release on October 29, 2018.

The audit was performed from November 12 - December 14 by Emery Rose Hall, Dominic Tarr, Chris Wood, Ramakrishnan Muthukrishnan, and Jean-Paul Calderone. The initial report was issued on December 20, 2018. An updated report has been issued on January 30, 2019 reflecting that the security issues called out in the initial audit report have been opened on Github and are in now the public domain. A final report will be issued following the discussion and verification phase of any and all outstanding open issues.

Coverage

Target Code and Revision

For this audit, we performed research, investigation, and review of the Sapling implementation and the changes made to the RPC interface followed by issue reporting, along with mitigation and remediation instructions outlined in this report. The following code repositories are in scope:

Specifically, we examined the Git revisions:

`78c228fc4dc6dfd289909b9c0d3e8e214b312d14`

All file references in this document use Unix-style paths relative to the project's root directory.

Areas of Concern

Our investigation focused on the following areas:

- Common and case-specific implementation errors
- Concerns raised in the Overwinter and Sapling reviews
- Any newly identified areas of concerns since the previous review
- Anything else as identified during the initial analysis phase

Methodology

We like to work with a transparent process and make our reviews a collaborative effort. The goals of our security audits are to improve the quality of systems we review and aim for sufficient remediation to help protects users. The following is the methodology we use in our security audit process.

Manual Code Review

In manually reviewing all of the code, we look for any potential issues with code logic, error handling, protocol and header parsing, cryptographic errors, and random number generators. We also watch for areas where more defensive programming could reduce the risk of future mistakes and speed up future audits. Although our primary focus is on the in-scope code, we examine dependency code and behavior when it is relevant to a particular line of investigation.

Vulnerability Analysis

Our audit techniques included manual code analysis, user interface interaction, and whitebox penetration testing. We look at the project's web site to get a high level understanding of what functionality the software under review provides. We then meet with with the developers to gain an appreciation of their vision of the software. We install and use the relevant software, exploring the user interactions and roles. While we do this, we brainstorm threat models and attack surfaces. We read design documentation, review other audit results, search for similar projects, examine source code dependencies, skim open issue tickets, and generally investigate details other than the implementation. We hypothesize what vulnerabilities may be present, creating Issue entries, and for each we follow the following Issue Investigation and Remediation process.

Documenting Results

We follow a conservative, transparent process for analyzing potential security vulnerabilities and seeing them through successful remediation. Whenever a potential issue is discovered, we immediately create an Issue entry for it in this document, even though we have not yet verified the feasibility and impact of the issue. This process is conservative because we document our suspicions early even if they are later shown to not represent exploitable vulnerabilities. We generally follow a process of first documenting the suspicion with unresolved questions, then confirming the issue through code analysis, live experimentation, or automated tests. Code analysis is the most tentative, and we strive to provide test code, log captures, or screenshots demonstrating our confirmation. After this we analyze the feasibility of an attack in a live system.

Suggested Solutions

We search for immediate mitigations that live deployments can take, and finally we suggest the requirements for remediation engineering for future releases. The mitigation and remediation recommendations should be scrutinized by the developers and deployment engineers, and successful mitigation and remediation is an ongoing collaborative process after we deliver our report, and before the details are made public.

Responsible Disclosure

Before our report or any details about our findings and suggested solutions are made public, we like to work with your team to find reasonable outcomes that can be addressed as soon as possible without an overly negative impact on pre-existing plans. Although the handling of issues must be done on a case-by-case basis, we always like to agree on a timeline for resolution that balances the impact on the users and the needs of your project team. We take this agreed timeline into account before publishing any reports to avoid the necessity for full disclosure.

Findings

Code Quality

The code is very well organized and follows best practices while avoiding known bugs and vulnerabilities. As an example, care is taken to make comparisons constant-time, which avoids side channel leakages. In addition, error handling is very well executed and thoroughly implemented. This effort is commendable and certainly decreases the risk of any vulnerabilities.

Organizational practices, such as grouping the implementation of similar functionality into a single directory, help provide discoverability of the code for a certain feature makes for an easily navigated code base, which results in a more efficient code review. However, this is slightly hampered by the subtlety of some of the groupings. For example, the RPC implementation comprises several such groupings; these groupings help but the existence of more than one seems to reflect a deeper principle which is difficult to infer from the code alone. Similar comments can be made for the naming convention used. There is

clearly a method to it but this is not immediately discoverable. Practices and conventions, such as the use of “flat case” for RPC-exposed methods, should be made explicit.

The readability and maintainability of the code could be further aided by providing a very high-level developer-oriented document explicitly stating the organizational principles and conventional uses applied to the code and/or by simplifying those principles within the code. As the Zcash codebase grows, this approach will make the code easier to understand, maintain and audit for further efficiency in the execution of Zcash's goals.

Issues

We list the issues we found in the code in the order we reported them. In most cases, remediation of an issue is preferable, but mitigation is suggested as another option for cases where a trade-off could be required.

ISSUE / SUGGESTION	STATUS
Issue A: RPC HTTP Server is Vulnerable to DNS-Rebinding Attacks	Reported / Open Issue
Issue B: Instance of Undefined Behavior	Reported / Open Issue
Issue C: Private Address Disclosure Vector	Reported / Open Issue
Issue D: Address Interpretation Surprise	Reported / Open Issue
Issue E: Unclear Transaction Interpretation	Reported / Open Issue
Suggestion 1: Strengthen the Wallet Password System	Reported / Open Issue

Issue A: RPC HTTP Server is Vulnerable to DNS-Rebinding Attacks

Synopsis

DNS-rebinding allows a website that is open locally to connect to the local Zcash node.

Impact

A website could attempt to brute force a password and then spend funds or extract private keys.

Preconditions

The attacker must create an attack site and a malicious DNS server. The victim must browse to this website on the same computer they run their Zcash node, while it is running. They must have a guessable password.

Feasibility

Unlikely to succeed at a targeted attack, but possible as a non-targeted attack. It is within reason that Zcash users are well informed about security practices and are likely to choose a strong password, but it’s also likely that they do not expect attacks from their local machine, and as a result, might choose a convenient password instead of a strong one. Additionally, given that they are probably used to logging into websites, their idea of a “strong password” might not quite reflect the situation appropriately.

Technical Details

To execute a DNS rebinding attack, the attacker creates a website (zcash-attack.com) and serves a malicious website. The attacker also controls a special DNS server that is the name server for their site, which uses a very low TTL so that DNS requests are not cached for a long period. As a result of the same-origin policy including the port in the origin, the website is required to open the attack site on the same port as Zcash RPC. Opening a website on a non-standard port looks suspicious, but the attack site could be hosted on the ordinary port before loading an attack script in an iframe on the attack port. This allows potential victims to be tricked to open their website - which then makes another request back to zcash-attack.com, causing the browser to call the DNS server again (because the TTL is low). In this instance, it returns 127.0.0.1 resulting in the same origin policy thinks it's requesting to zcash-attack.com, while actually connecting to localhost. The attack site can then guess weak passwords and, because the site is local, they can easily make tens of thousands of requests.

Mitigation

Zcash users should always set a strong password.

Remediation

RPC HTTP server should check that the host header is localhost:8232 (or the configured port) and not anything else such as `zcash-attack.com`. This method will prevent a website attacking the Zcash server. In addition, the RPC server should cause exponential delay between failed authentication attempts, which will prevent brute force password attempts. Exponential delays between password attempts also prevents an attacker who might have alternative methods where they could set the host header to make local HTTP requests.

Status

Reported. Issued A captured on GitHub (Issue 3791).

Verification

Pending issue completion and verification review.

Issue B: Instance of Undefined Behavior

Synopsis

Shifting a signed 32-bit integer representing the number 1, to the left by 31 bits is undefined. This is a bug. However It does not appear to impact security.

Impact

Low.

Preconditions

Not applicable.

Feasibility

Hard.

Technical Details

src/arith_uint256.cpp: base_uint<BITS>::bits(): Line 177

```
for (int bits = 31; bits > 0; bits--) {
```

```
    if (pn[pos] & 1 << bits)
        return 32 * pos + bits + 1;
}
```

pn[] is uint32_t. “bits” is an integer (loop variable). 1 << bits is a signed number (i.e. 1) shifted by 31 bits, which is undefined behaviour, because 1 << 31 = $2^{31,}$ However, a signed 32-bit number can only represent a number in the range -2^{31} to $2^{31} - 1$.

Mitigation

Suggest the following code (changes highlighted in red):

```
for (size_t bits = 31; bits > 0; bits--) {
    if (pn[pos] & 1U << bits)
        return 32 * pos + bits + 1;
}
```

Remediation

Implement the above suggestion.

Status

Reported. Issue B captured on GitHub (Issue 3792).

Verification

Pending issue completion and verification review.

Issue C: Transaction Details Disclosure Vector

Synopsis

The shell-based zcash-cli workflow encourages users to create a record of the private sending and payment addresses involved in their transactions.

Impact

Low.

Preconditions

Not applicable.

Feasibility

Hard.

Technical Details

By using zcash-cli in a shell session, arguments passed to the command appear with the command in the user’s shell history as one would expect. These arguments include the addresses from which funds are taken for transactions as well as the addresses to which those funds are sent. These addresses may then be recovered by an attacker at a future point if the shell history is compromised. While this would require

access to the user's persistent storage, such storage is not often considered security or privacy critical and it may not be protected to the same degree as other information related to the user's Zcash wallet.

Similarly, but less likely, the addresses will appear in the system process table for the duration of the command and therefore be visible to an attacker running `ps` or similar on the system at the same time.

The result of such a compromise may be the undesired linking of transactions to a user or many of the other disclosures the advent of private payments is meant to prevent.

Mitigation

A user can avoid having this information recorded in their shell history entirely by taking appropriate steps to configure their local environment. Depending on their shell, they can likely disable shell history entirely They can disable shell history entirely (e.g., for bash, run `unset HISTFILE`) or for just for the zcash-cli commands (e.g., for bash, prefix the command with a single space).

Remediation

The CLI wallet could be modified to support a style of command where transaction information, including addresses, is not part of the process `argv`. For example, there could be a form of commands where transaction information could be read from stdin instead.

A GUI wallet would most likely not suffer from this exact form of the issue because GUI applications do not have the same convention of recording activities (though it would certainly be *possible* for a GUI application to log this information).

Status

Reported. Issue C captured on GitHub (Issue 3793).

Verification

Pending issue completion and verification review.

Issue D: Address Interpretation Surprise

Synopsis

Users may be surprised by zcash-cli's willingness to accept BIP 173-encoded addresses in transaction commands such as `z_listunspent` and `z_sendmany`.

Impact

Low.

Preconditions

Not applicable.

Feasibility

Medium.

Technical Details

Throughout, DecodePaymentAddress (`src/key_io.cpp`) is used to decode address strings supplied to wallet operations. This performs the expected Base 58 decoding but if this fails it attempts Bech32 decoding as well. If the string can be decoded according to the rules of Bech32, an address is returned to the caller.

This appears to mean that Bech32-encoded addresses can be used in zcash-cli commands. While a user is unlikely to intentionally supply such an address string (and even less likely to be surprised if it works), this encoding may surprise users who unintentionally (or obliviously) supply such an address. In particular, if inputs are scanned for desirability assuming only Base 58 encoding then certain undesirable addresses may be able to bypass detection by using Bech32 encoding instead. An example of this could be an attempt to reject use of all transparent addresses by inspecting addresses for a "t1" prefix. A Bech32-encoded transparent address would not be detected by such a scheme.

Though inspection of the code makes it seem like this is possible, we did not confirm this through experimentation.

Mitigation

Users can visually inspect any addresses received from second-parties before creating a transaction using them.

Remediation

If there is a good reason to continue to support Bech32-encoded addresses, make this support more obvious and discoverable for users. If there is not, remove it.

A complementary approach could be to allow wallets or sender addresses to be marked persistently as being allowed to participate only in shielded transactions. This would provide a measure of automatic protection against accidental transaction disclosures by failing any attempted transactions that involve a transparent address - regardless of address encoding.

Status

Reported. Issue D captured on GitHub (Issue 3794).

Verification

Pending issue completion and verification review.

Issue E: Unclear Transaction Interpretation

Synopsis

Transactions can be constructed using zcash-cli using JSON strings which do not have a single unambiguous interpretation. These may not have a stable meaning across different versions of Zcash or they may mean something different to Zcash than they mean to a user.

Impact

Low.

Preconditions

Not applicable.

Feasibility

Medium.

Technical Details

The `zcash-cli z_sendmany` command accepts a JSON encoded representation of a transaction to perform. A JSON encoded string containing duplicates of certain properties is accepted and a transaction is created. These JSON encoded strings do not have an unambiguous interpretation. For example, `{"address": A1, "address": A2}` can only represent an object with an address property

with a value of *either* A1 *or* A2. If such ambiguity is allowed in the course of creating a transaction, there is some scope for users to be surprised by the resulting behavior.

An example of exploiting this is that a user may be given an encoded JSON string representing a transaction to submit. The user may inspect the JSON string to verify certain properties (for example, that it has only shielded outputs). If the JSON string is of the form `[{"memo": …, "address": "t1…", "amount": …, "address": "zc…"}]` then the transparent address "hidden" in the middle may be overlooked. However, as of Zcash 2.0.1, that is the address which will actually be used. If other tools are used to evaluate the desirability of the transaction, similar issues may arise as other software may have a different interpretation of the JSON.

The same issue applies to other properties of these objects - `amount` and `memo` - which may be used to fool users in other ways (for example, getting them to send a different amount than they expected).

Mitigation

Users can very carefully manually review any JSON they are given to give to Zcash. Users can also build their own automatic, strict JSON validation tool which rejects such ambiguous inputs and apply this tool before passing any JSON along to zcash-cli.

Remediation

Zcash-cli could reject any ambiguous JSON inputs. This would alert users to a potential attack and force them to decide how to proceed (not submitting the transaction at all, editing the transaction JSON to make it unambiguous, etc).

Status

Reported. Issue E captured on GitHub (Issue 3795).

Verification

Pending issue completion and verification review.

Suggestions

Suggestion 1: Strengthen the Wallet Password System

Synopsis

A wallet encryption password has the option of being very weak and small. Though wallet encryption seems to be an experimental feature, we advise that it should be strengthened.

`src/rpcwallet.cpp:walletpassphrasechange()` allows a passphrase to have a length of one (1). Since this passphrase is encrypting the file at rest, we strongly suggest that implementing a stronger system making passwords more difficult to guess.

Mitigation

Creation of weaker passphrases should be disallowed at the UI level.

Status

Reported. Suggestion 1 captured on GitHub (Issue 3796).

Verification

Pending issue completion and verification review.

Recommendations

We recommend that the *Issues* and *Suggestions* stated above are addressed as soon as possible and followed up with verification by the auditing team.

Appendix 1: Activity Log

These are notes from the reviewers about their activities during the code audit. They detail the approach and investigative activities undertaken. All issues found are listed in the report. This is just for the purposes of transparency and could be helpful for another auditor to understand the evaluation activities.

Log 1:

2018-11-19/20

Ran Clang Analyzer (`scan-build`) over the v2.0.1 tag and investigated the results.

2018-11-20

Ran Valgrind over the v2.0.1 tag and investigated the results.

2018-11-21/22/23

Reading source code:

- tracing from the `accept()` in `main.cpp:ProcessMessage` up through high-level message dispatch.
- Reading implementation of different wallet operations.

Log 2:

2018-11-15

Built the code from v2.0.1 tag.

Started poking around the github issues fixed in v2.0.1:
https://github.com/zcash/zcash/pulls?page=2&q=is%3Apr+is%3Aclosed+rpc&utf8=%E2%9C%93

Started tracing the “life of an rpc call” - zcash-cli to the actual rpc invocation from the wallet.

Lots of use of foo() to mean, foo takes zero arguments. According to the standard, this means the opposite: foo() takes any number of arguments. At least in C. TODO: check if this is true for C++.

2018-11-20

src/univalue/include/univalue.h:

Univalue represents a value for the JSON type (Null, String, Number, Bool, Array, Object).

There are two public: declarations in the class. Makes reading the code a bit difficult. Can we combine them under one public section?

2018-11-21

Looking at src/wallet/asyncrpc*.cpp

2018-11-22/23

Looking at wallet rpcs. Many of the rpc calls take an amount (string representation of a floating point number like 0.12345). The internal representation of the Amount is the type CAmount which is nothing but int64_t. This is great. Had it been a floating point number, it would have been a extremely problematic.

Now, how is the user supplied string getting converted to an int64_t?
src/utilstrencoding.cpp:ParseFixedPoint() does that. Reading through that code.

2018-11-28/29

wallet.cpp: looks pretty nice so far.

2018-11-30

src/rpcwallet.cpp: CRPCCommand[] defines the rpc command table (category, name, function, safemode)

2018-12-3/4:

rpcwallet.cpp:walletpassphrasechange(): So, one can set an 1 character passphrase for the wallet. Should that be explicitly disallowed? Need a one line change in the code.

(19/Dec/2018: looks like wallet encryption is experimental, so ignoring this for now)

src/arith_uint256.cpp: base_uint<BITS>::bits(): Line 177

```
for (int bits = 31; bits > 0; bits--) {
   if (pn[pos] & 1 << bits)
      return 32 * pos + bits + 1;
}
```

pn[] is uint32_t. “bits” is an integer (loop variable). 1 << bits is a signed number (i.e. 1) shifted by 31 bits, which is undefined behaviour, because 1 << 31 = 2^{31}. However, a signed 32bit number can only represent a number in the range -2^{31} to $2^{31} - 1$. Suggest the following code (changes highlighted in red):

```
for (size_t bits = 31; bits > 0; bits--) {
   if (pn[pos] & 1U << bits)
      return 32 * pos + bits + 1;
}
```

Log 3:

2018-11-19/20

Reading rpc/protocol and rpc/server, rpc/httprpc, and looking at password practices recommended by the zcash documentation. Zcash recommends “strong passwords” but doesn’t explain what that means.

It seems nothing checks the host header, so I conclude zcash is vulnerable to dns-rebinding.

2018-11-28/29/30

Read through everything which exposes rpc methods. Take special note of which things are enabled in safe mode or not.

2018-12-7

Read through more rpc methods.