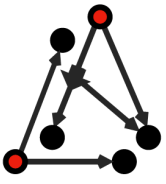
Security Audit Report: Implementation Analysis (1.0.15) and Overwinter Specification Review

Zcash

Report Version: 29 May 2018



Least Authority
Freedom Matters

Table of Contents

This audit makes no statements or warranties and is for discussion purposes only.

Overview

Least Authority performed a security audit of the currently implemented version of Zcash, *Magic Bean* v1.0.15, and the *Overwinter* specification for future implementation, at the request of the Zcash Company.

The audit was performed in February and March 2018 by Jack Lloyd, Ramakrishnan Muthukrishnan, James Prestwich, Emery Rose Hall and Dominic Tarr. The initial report was issued on March 29, 2018. The updated report was issued on May 29, 2018 after the verification phase.

Review Scope

For the *Magic Bean* code audit, we reviewed the following repositories of v1.0.15:

- https://github.com/zcash/zcash
- https://github.com/zcash/libsnark
- https://github.com/zcash/librustzcash
- https://github.com/zcash/zcash-seeder
- https://github.com/zcash/zcash-gitian

For the *Overwinter* specification, we reviewed the following Zcash Improvement Proposals (ZIPs):

- ZIP 143
- ZIP 200
- ZIP 201
- ZIP 202
- ZIP 203

In reviewing both the current codebase (v1.0.15) and the proposed *Overwinter* changes to the Zcash cryptography and consensus protocols, the goal was to uncover any issues that could expose users to loss, allow coin forgery, lead to network failure, break privacy guarantees, cause performance problems, lead to consensus failures and cause game-theoretic challenges.

In manually reviewing the v1.0.15 code, we looked for general formatting errors and common C++ programming language mistakes like buffer overflows, use-after-free etc, that can lead to leakage of sensitive information or denial of service attacks. We also did a cursory review for adherence to the current protocol specification, but did not expect to find much because of the previous audits that were completed. We also considered areas where more defensive programming could reduce the risk of future mistakes and speed up future audits.

In reviewing the *Overwinter* specification, we looked for any potential issues with logic, interaction dependencies and changes to assumptions that could lead to attacks on nodes or the possibility of network splits while in the protocol upgrade phase.

This audit makes no statements or warranties and is for discussion purposes only.

Findings

Although Zcash has only been around for a short time, it is clear that the contributing team has put significant effort into good programming practices and detail-oriented specifications. Overall, we found both the *Magic Bean* code and *Overwinter* specification to be of high quality and well-thought out.

Issues

We list the issues we found in the code in the order we reported them.

ISSUE / SUGGESTION	STATUS
Issue A: Pow leaks in `windowed_exp`	Verified: *Mitigated with Disclaimer*
Issue B: Exponent leaks via `power` function	Verified: *Mitigated with Disclaimer*
Issue C: Undefined behavior in `crypto/common.h`	Verified: *Version 1.1.1 -rc1*
Issue D: Undefined behavior in `CBaseDataStream::read`	Verified: *Version 1.1.1 -rc1*
Issue E: `CTxMemPool::check()` does nothing when turned on	Verified: *Version 1.1.1 -rc1*
Issue F: Transaction expiry reduces safety in reorgs	Verified: *Mitigated with Documentation*

Issue A: Pow leaks in `windowed_exp`

Synopsis

The `windowed_exp` function in `src/snark/libsnark/algebra/scalar_multiplication/multiexp.tcc` leaks the bits of pow through a cache-based side channel, as the lookup of `powers_of_g[outer][inner]` depends on the bits of e.

Impact

Leakage of private key material.

Preconditions

An attacker capable of running code on the same machine as zcashd (for example using JavaScript executing in a browser, or a cross-VM attack in a shared hosting provider).

Feasibility

Unlikely.

Remediation

Consider using blinding (eg, using `pow+rand()*order_of_group`) or a masked table lookup.

This audit makes no statements or warranties and is for discussion purposes only.

Verification

Not changed. Mitigation in the form of a disclaimer about resistance to side channel attacks in proving will be used in the short term.

Issue B: Exponent leaks via `power` function

Synopsis

The function `power` in `src/snark/libsnark/algebra/exponentiation/exponentiation.tcc` leaks bits of the exponent via use of a square-and-multiply algorithm.

Impact

Leakage of private key material.

Preconditions

An attacker capable of running code on the same machine as zcashd (for example using JavaScript executing in a browser, or a cross-VM attack in a shared hosting provider).

Feasibility

Unlikely.

Mitigation

Avoid running Zcash in environments where an attacker can execute arbitrary code on the same CPU.

Remediation

Switch to a different algorithm which does not leak information via timing or cache-based side channels.

Verification

Not changed. Mitigation in the form of a disclaimer about resistance to side channel attacks in proving will be used in the short term.

Issue C: Undefined behavior in `crypto/common.h`

Synopsis

Invalid casts in `crypto/common.h` invoke undefined behavior. In C/C++ casting a pointer from a smaller type (such as unsigned char) to a larger type can have undefined behavior. In particular, depending on the CPU platform this can cause crashes or incorrect computations if the data is misaligned. The Read* and Write* functions in `crypto/common.h` make use of these unsafe casts.

Impact

Invalid computations or crash depending on platform and compiler. It is unlikely that x86 platforms are affected.

Preconditions

Depends on architecture and compiler used.

This audit makes no statements or warranties and is for discussion purposes only.

Feasibility

This cannot be triggered by an external input, instead it is based on what optimizations the compiler performs.

Mitigation

Avoid using Zcash on systems where misaligned loads can cause incorrect computation (such as ARM or SPARC).

Remediation

Using `memcpy` is the C/C++ standard approved way of converting from a smaller type to a larger type. For example, `memcpy(&int32, char_buf, 4)` will work regardless of the alignment of `char_buf`.

Verification

Code change verified in version 1.1.1 -rc1.

Issue D: Undefined behavior in `CBaseDataStream::read`

Synopsis

`CBaseDataStream::read` can potentially call memcpy with a `NULL` pointer, if its `pch` argument is null. This is undefined behavior in C/C++. This was detected by compiling with UbSan and running the test suite.

Impact

Hypothetically a crash or remote code execution.

Preconditions

An attacker must contrive a `NULL` to be passed to this function.

Feasibility

This bug is unlikely to have a practical effect.

Mitigation

Avoid compiling Zcash with compilers which optimize heavily on the basis of undefined behavior.

Remediation

Check for a `NULL` argument and return immediately.

Verification

Code change verified in version 1.1.1 -rc1.

This audit makes no statements or warranties and is for discussion purposes only.

Issue E: `CTxMemPool::check()` does nothing when turned on

Synopsis

`CTxMemPool::setSanityCheck` converts a float to a 32-bit unsigned integer. When doing so, the exponent part is discarded. However, if the integer part overflows the target size of the integer, then the behaviour is undefined.

```
void setSanityCheck(double dFrequency = 1.0) { nCheckFrequency = dFrequency *
4294967296.0; }
```

Impact

The value of `nCheckFrequency` is undefined. On x86-64 platform running Debian GNU/Linux with g++ 7.3.0, the value was set to `0`. So, the sanity check (the `check` function in `txmempool.cpp` that makes sure that the pool does not contain two transactions that spend the same input, etc) returns without doing any checks.

Preconditions

Invoke `zcashd` with `-checkmempool` flag.

Feasibility

The bug is unlikely to have any practical effects as it seems to be only used for regression tests.

Mitigation

If `-checkmempool` is turned on and then the log prints related to the `mempool` checks should be ignored.

Remediation

Change the definition of `setSanityCheck` to:

```
void setSanityCheck(double dFrequency = 1.0) { nCheckFrequency =
static_cast<int>(dFrequency * 4294967295.0); }
```

Verification

Code change verified in version 1.1.1 -rc1.

Issue F: Transaction expiry reduces safety in reorgs

Synopsis

This is not a vulnerability. Rather it details the trade-offs made by a specific design decision, and the non-obvious impacts on users.

Overwinter adds an `expiryheight` field to transactions, which signals transaction expiry as detailed in ZIP 203 . This field specifies a block height at which the transaction becomes invalid. This may result in confirmed transactions becoming unconfirmed and invalid in a reorg.

In the harmless case, a transaction becomes permanently invalid in a short reorg, and must be resubmitted. This will be relatively common if very short expiry deltas are used. Subjectively users will see a transaction receive one or more confirmations, then those confirmations disappear as the transaction becomes invalid.

This audit makes no statements or warranties and is for discussion purposes only.

If the change in blockheight after such a reorg is longer than the recipient's chosen acceptance period (measured in confirmations), then the funds are functionally double spent. The recipient will accept the transaction for value. However, after the reorg, the transaction will be invalid, and the sender will be in control of the funds.

This may be exploited to directly monetize attacks that create long reorgs. Eclipse attacks and 51% attacks may be effectively monetized this way. If some users are known to accept 0-confirmation transactions, selfish miners may exploit this to become more profitable

As a side effect, this attack increases the ecosystem damage that caused by 51% attacks and consensus failures. A 51% attacker may mine a private chain and selectively include transactions with `expiryheight` set in order to maximize profit. This will be strictly more profitable than an equivalent attack in Bitcoin.

A consensus failure resulting in a long reorg -- as occurred with 24 blocks (approx. 4 hours) in Bitcoin in March 2013 and 165 blocks (approx. 1 hour) in Ethereum in November 2016 -- will cause users to double-spend accidentally. This may result in long chains of confirmed transactions becoming invalid. This means that consensus failures will cause strictly more damage to users than a Bitcoin reorg of equivalent length. If the resulting split chains proceed at different rates, consensus failures may be directly monetized via double-spends.

Consensus failures are uncommon, but possible in many types of upgrades. Notably, the March 2013 Bitcoin consensus failure was caused by a change in the operation of Bitcoin's BerkeleyDB, not by a consensus change.

Impact

Accidental double-spends may occur. Occasionally, miners may intentionally double-spend. Transactions spending the outputs of these transactions become permanently invalid. The effectiveness of certain attacks is increased, and certain attacks that were not previously directly monetizable become so.

Preconditions

A transaction with `expiryheight` set must be confirmed in a block which is orphaned in a re-org. The post-reorg chain must not include the transaction before it expires.

Example:

A transaction with an `expiryheight` of 5005 is included in block 5000 ("5000-a") and verified by a client. The client then learns of a new chain tip at height 5010, which builds on an alternate block 5000 ("5000-b"), which has greater total difficulty and does not include the transaction. Despite previously having 1 confirmation in block 5000-a, the transaction is permanently invalid in the post-reorg chain building off block 5000-b.

Feasibility

High feasibility of permanent disconfirmation in short reorgs (the low-harm case).
Medium feasibility of permanent invalidation of a transaction chain.
Low feasibility of a deliberately engineered double-spend via miner or network manipulation.

In a consensus failure:
High probability of ecosystem damage.
High feasibility of deliberate exploitation.

This audit makes no statements or warranties and is for discussion purposes only.

Mitigation

- Users should not accept transactions without at least 25 confirmations (approximately 1 hour), to prevent accidental double spends.
- When accepting transactions, users should check their chain tip against outside sources to prevent reorgs caused by eclipse attacks.
- UI should be modified to inform users that confirmed transactions with `expiryheight` set may become permanently invalid in certain circumstances.
- The ZCash team should take additional precautions in soft forks and other network upgrades to prevent consensus failures and network partitions.

Remediation

Consider a consensus rule treating children of transactions with `expiryheight` set as if they had a relative locktime of at least 50 blocks (approximately 2 hours). This greatly increases the difficulty of practical exploitation.

Verification

The `expiryheight` field is now documented in 2018.0-beta-19 specification.

Recommendations

Zcash is the first significant deployment of zk-SNARKSs and there is still a lot of uncertainty in the potential attacks that could be utilized against this distinguishing feature of Zcash. Because of the lack of zk-SNARKs deployments, even experienced cryptographic engineers will find the implementation of the zero-knowledge proofs challenging to understand and interact with.

Although we found the Zcash code to be of high quality and well-thought out, it is potentially more complex than necessary. While this is expected in these novel implementations, the complexity results in obscurity.

Also, the Zcash team has made a commendable effort to simplify these concepts with thorough explanations. Many topics relating to Zcash's design and implementation have been covered in documentation. While this documentation is helpful in explaining the general functioning of zero-knowledge proofs and design decisions, we found it lacking in references to how these concepts are implemented in the actual code.

We recommend that the Zcash team look for opportunities to simplify key components of the code and expand documentation code references to allow for better maintenance, including for future audits and encouraging community contributions.

This audit makes no statements or warranties and is for discussion purposes only.