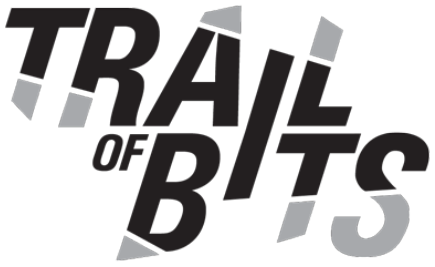

Zcash

Security Assessment

April 29, 2020

Prepared For:
Benjamin Winston | *Electric Coin Co.*
bambam@electriccoin.co

Prepared By:
Ben Perez | *Trail of Bits*
benjamin.perez@trailofbits.com

Will Song | *Trail of Bits*
will.song@trailofbits.com

Executive Summary

From April 20 through April 29, 2020, Electric Coin Co. engaged Trail of Bits to review the security of ZIP 213 and ZIP 221. Trail of Bits conducted this assessment over the course of three person-weeks with two engineers reviewing the diffs and code at three different commits from the Zcash repository. These include 084303d, the first MMR commit; 659ac40, the second MMR commit; and 35bff6a, the shielded coinbase commit.

The first week was spent examining the Merkle mountain range (MMR) implementation, as specified in ZIP 221. We focused most of our effort on `coins.cpp` and `history.cpp`, which contain the core MMR data structure logic. After closely examining the code for both logic bugs and common C++ errors, we found it to be generally high-quality and faithful to the specification. Additional coverage was generated with a set of automated tools in an effort to more easily catch common C++ errors and improve the Zcash test suite. We ran the included test suites with `AddressSanitizer` and `UndefinedBehaviorSanitizer` as well as a simple pass of `cppcheck`, as detailed in Appendix B.

Subsequently, we assessed the addition of shielded coinbase transactions (ZIP 213) to the Zcash protocol. This commit enabled shielded reward addresses in coinbase transactions, and we concluded that the proposed consensus changes are correctly implemented as per the ZIP. Because the sanitizer runs from the previous week were performed on the latest commit hash available in master at the time, the sanitizer runs cover the testing code touched by this commit, namely `test_checktrasaction.cpp`.

Overall, we did not discover any serious cryptographic issues during the timeframe allotted for this engagement. However, we were able to improve test quality by building the source code with sanitizers. We recommend looking further into any memory errors produced by ASAN and UBSAN, and integrating these builds and test runs into CI. This can help ensure that normal operation of `zcashd` and `zcash-cli` will not produce any issues.

Project Dashboard

Application Summary

Name	ZIP 213, ZIP 221
Version	Commits 084303d, 659ac40, 35bff6a
Type	C++
Platforms	Zcash blockchain

Engagement Summary

Dates	April 20–April 29, 2020
Method	Whitebox
Consultants Engaged	2
Level of Effort	3 person-weeks

Vulnerability Summary

Total High-Severity Issues	0	
Total Medium-Severity Issues	0	
Total Low-Severity Issues	0	
Total Informational-Severity Issues	3	■ ■ ■
Total Undetermined-Severity Issues	0	
Total	3	

Category Breakdown

Configuration	1	■
Undefined Behavior	2	■ ■
Total	3	

Engagement Goals

The engagement was scoped to provide a security assessment of the ZIP 213 and ZIP 221 implementations. In a prior engagement, Trail of Bits reviewed the security implications of these ZIPs on the Zcash protocol itself. That review did not reveal any issues in the specifications, but did warn of a potential loss of privacy for shielded addresses using FlyClient.

In this review, we sought to answer the following questions:

- Are Merkle mountain ranges properly constructed?
- Is there a possibility for hash collisions?
- Has each ZIP been properly integrated with the Zcash system?
- Is the code free of common C++ bugs?

Coverage

Shielded coinbase (ZIP 213). We examined `miner.cpp` and `transaction_builder.cpp` for adherence to the ZIP 213 specification, along with sections of the code where small modifications were required to incorporate shielded coinbase transactions. Mainly, we ensured that shielded addresses were accepted by the consensus mechanism as valid recipients of coinbase transactions, and verified that no undefined behavior or logic bugs were present.

Merkle mountain range (ZIP 221). We manually inspected `coins.cpp` and `history.cpp` for adherence to the ZIP 221 specification. This involved verifying that:

- The Merkle mountain range data structure was accurately implemented
- This data structure was properly integrated into the Zcash consensus mechanism
- These additions were free of common C++ bugs

We also examined the tests for ZIP 221 to ensure they had sufficient coverage.

C++ code quality. We manually modified the build system to use `clang`, compiled the source with `-fsanitize=address` and `-fsanitize=undefined`, and reported some results in Appendix B. `Cppcheck` was also run on the code.

Recommendations Summary

This section aggregates all the recommendations made during the engagement. Short-term recommendations address the immediate causes of issues. Long-term recommendations pertain to the development process and long-term design goals.

Short Term

❑ **Build and review sanitizer output.** The test runs with both ASAN- and UBSAN-produced errors. Investigate and resolve these issues.

❑ **Develop a fuzzer with `libFuzzer`.** `LibFuzzer` is one of the easiest to use and best coverage-guided C++ fuzzers available for free. Fuzzing Zcash will allow early detection of bugs before they can cause any serious damage.

Long Term

❑ **Integrate sanitizers into CI.** Manually building and running sanitizers can become difficult as your build process evolves. When you integrate sanitizers into your CI environment, these checks become automatic and help catch bugs in the development process.

❑ **Develop and document a more flexible build system.** `Scan-build` is a very nice static analyzer, but it requires you to replace your C++ compiler on the fly during the build process. The current build system does not make that easy to do.

❑ **Run fuzzers continuously.** Ensure that fuzz tests are constantly being run either with OSS-Fuzz and their CI integration or via a custom fuzzing farm.

Findings Summary

#	Title	Type	Severity
1	Inflexible build system	Configuration	Informational
2	ASAN/UBSAN errors and cppcheck errors	Undefined behavior	Informational
3	Lack of fuzzing	Undefined behavior	Informational

1. Inflexible build system

Severity: Informational
Type: Configuration
Target: `zcutil/build.sh`

Difficulty: N/A
Finding ID: TOB-ZCASH-001

Description
The current documentation around the Zcash build system directs users to run the included `zcutil/build.sh` build script. While this does call into the included automake scripts, they are not well documented. Furthermore, the system compiler is statically selected via an included script per host operating system. This disallows on-the-fly reconfiguration of the C and C++ compilers via environment variables. Because some static analyzers rely on using their own fake compiler to gain frontend information about include paths and defines, this inflexibility makes it difficult to use these tools. One of our preferred static analyzers, the `clang` analyzer `scan-build`, operates in this fashion. For large C++ projects where unintended undefined behavior can slowly run rampant, the design choice to use hard-coded compilers should be considered a regression.

Recommendation
Long term, redesign the build system to be more in line with standard automake builds that allow for the configuration of the C and C++ compilers by setting the `CC` and `CXX` environment variables. It is okay to still keep an all-in-one build script for less experienced developers, but others would greatly appreciate a more configurable system. While doing so, also document some of the more important configure options.

2. ASAN/UBSAN errors and cppcheck errors

Severity: Informational
Type: Undefined behavior
Target: src/test_bitcoin, src/zcash-gtest

Difficulty: N/A
Finding ID: TOB-ZCASH-002

Description
While running the sanitizer tests described in Appendix B, we encountered an ASAN error with test_bitcoin, and several UBSAN errors with both test_bitcoin and zcash-gtest. We believe that the ASAN error stems from a regression in the testing frontend and not an error in the core Zcash code itself. UBSAN, however, reports errors a bit more generically. Here is a sample of the errors it reported with test_bitcoin and zcash-gtest.

```
test/script_P2PKH_tests.cpp:50:5: runtime error: index 25 out of bounds for type 'const unsigned char
[23]'
test/script_P2PKH_tests.cpp:55:5: runtime error: index 25 out of bounds for type 'const unsigned char
[7]'
../dist/./../src/db/db_dup.c:139:15: runtime error: member access within null pointer of type 'BKEYDATA'
(aka 'struct _bkeydata')
../dist/./../src/btree/bt_delete.c:153:13: runtime error: member access within null pointer of type
'BKEYDATA' (aka 'struct _bkeydata')
../dist/./../src/btree/bt_delete.c:86:13: runtime error: member access within null pointer of type
'BINTERNAL' (aka 'struct _binternal')
/usr/include/string.h:43:28: note: nonnull attribute specified here
wallet/test/rpc_wallet_tests.cpp:1962:9: runtime error: downcast of address 0x55a291e7c160 which does
not point to an object of type 'AsyncRPCOperation_sendmany'
0x55a291e7c160: note: object is of type 'AsyncRPCOperation_mergetoaddress'
00 00 00 00  38 0a 27 8d a2 55 00 00  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  00 00 00 00
             ^~~~~~~~~~~~~~~~~~~~~~
             vptr for 'AsyncRPCOperation_mergetoaddress'
```

Figure 2.1: UBSAN output.

A basic analysis with cppcheck was also performed. We enabled checking all unique #define configurations but set no extra flags for extra preprocessing, which caused some macro expansion to fail. See the error log here.

Recommendations
Short term, build with sanitizers enabled, and investigate and fix each error. We recommend ASAN, MSAN, and UBSAN for Zcash. Running tests with no memory errors ensures greater operational safety of the production client and daemon.

Long term, integrate these types of checks into a CI environment. Automatic testing is only good if it is actually automatic. Luckily, there is already a configure.ac option, so CI scripts only need very minor adjustments.

3. Lack of fuzzing

Severity: Informational
Type: Undefined behavior
Target: /

Difficulty: N/A
Finding ID: TOB-ZCASH-003

Description
Fuzzing is an emerging technology that aims to solve complex problems with sheer brute force. If one throws enough random inputs, perhaps guided by a few constraints, eventually we can discover some amount of interesting behavior that a function or a program will exhibit.

`Clang`'s `libFuzzer` is the best available fuzzer for integration with this repository. `LibFuzzer` is a coverage-guided fuzzer that tries to explore as many code paths of the assembly of the fuzz target as possible, with built-in support inside `clang`. There is no need to use the custom compiler bundled with `afl`. Just build your fuzz target with `-fsanitize=fuzzer` and run the resulting fuzzer program. You can even combine it with any other sanitizer of your choice, e.g., `ASAN` or `MSAN`, to fuzz for other types of bugs.

Recommendation
The Zcash team has already spent a lot of effort trying to fuzz its product with `afl`. Short term, we recommend directing future fuzzing efforts towards building a `libFuzzer` fuzz suite to find bugs before they are detected in a live release.

Long term, integrate the `libFuzzer` suite with OSS-Fuzz and their CI-Fuzz environment if Zcash is an allowable project. Otherwise, ensure that fuzz tests are constantly being run when new commits are added, especially before an upcoming release.

A. Vulnerability Classifications

Vulnerability Classes	
Class	**Description**
Access Controls	Related to authorization of users and assessment of rights
Auditing and Logging	Related to auditing of actions or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to protecting the privacy or integrity of data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to race conditions, locking, or order of operations
Undefined Behavior	Related to undefined behavior triggered by the program

Severity Categories	
Severity	**Description**
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for

	client's reputation, moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation, or serious legal or financial implications

Difficulty Levels	
Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in-depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details, or must discover other weaknesses in order to exploit this issue

B. Improved C++ Testing

The Zcash codebase has an admirable number of unit tests. However, static analyzers and other forms of generic C++ analysis are not currently used in the build toolchain. `Clang` offers fantastic sanitizer support and it is easy to build Zcash with them enabled. `Gcc` also supports the `-fsanitize` option but we have not tested Zcash using `gcc`. To start using `clang` and sanitizers, simply edit `depends/hosts/linux.mk`, assuming you are building on Linux, and add `-fsanitize=<whatever-you-want>` to `CFLAGS`. Then change the compiler from `gcc -m32/64` to `clang -m32/64`, and do the same for `g++`. After that, all you need is a `./zcutil/clean.sh && ./zcutil/build.sh` to rebuild everything under the new settings (apart from OpenSSL).

The sanitizers we find most useful are AddressSanitizer, MemorySanitizer, and UndefinedBehaviorSanitizer. `ASAN` can help find obscure C++ memory errors, while `UBSAN` is able to detect some undefined behavior at runtime, all while being significantly faster than the alternative, `valgrind`. When `test_bitcoin` ran under `ASAN`, it discovered a heap use after free by different threads. We think this may be related to how the tests are executed, but more scrutiny is definitely required. `UBSAN` offers a different view of the picture: `test_bitcoin` and `zcash-gtest` produce a similar set of errors (see Figure 2.1 for more details). This may also be used in conjunction with the Python test suite to gain even more coverage over the codebase.

One final tool we would like to suggest is `cppcheck`, a static analyzer that can test for undefined behavior statically under all build configurations. An error log while running it with minimal arguments is shown here. We also recommend updating the build system to support the `clang` static analyzer. This requires setting the C and C++ compilers at build time, which we were unable to do within the timespan of this audit.

It was later pointed out by the Zcash team that the included `configure.ac` script already has an option to build with sanitizers enabled. However, the C and C++ compilers are still not easily changed. We recommend documenting this in some way.

Finally, `gcc-10` is introducing its `-fanalyzer` compiler analysis tool. Although they claim it is still experimental and not likely to work for non-C code, it may still be worth looking into.