Zcash Sapling Security Audit

Final Report, 2019-01-30

FOR PUBLIC RELEASE

Contents

1 Summary

Zcash is a cryptocurrency that provides strong privacy protections thanks to state-of-the-art cryptographic components, notably non-interactive zero-knowledge (NIZK) protocols. At the core Zcash is a type of NIZK zero-knowledge succinct non-interactive argument of knowledge (zk-SNARK) that relies on elliptic-curve cryptography, namely pairings operations. In 2018, Zcash will deploy an upgrade of its protocol, called *Sapling*, which will include more secure and faster zk-SNARKs.

Zcash hired Kudelski Security to perform a security assessment of the Sapling upgrade, with a focus on its cryptographic components:

- "pairing", the library implementing the pairing arithmetic over the elliptic curve BLS12-381, available at `github.com/ebfull/pairing`.
- "bellman", the implementation of the zk-SNARK on top of pairing, available at `github.com/ebfull/bellman`.

Based on these we reviewed the security and soundness of the new protocol in Sapling as specified by Zcash, as well as its implementation, available at `github.com/zcash-hackworks/sapling-crypto`.

This document reports the potential security issues found, improvement suggestions, as well as our assessment of Sapling's security. Our general conclusions are the following:

- The Sapling cryptography is based on solid, well-researched building blocks and protocols. It has evidently been designed and reviewed by leading experts, intimately familiar with the Zcash internals. The reference (and production)

implementation is coded in Rust, which eliminates certain classes of bugs and overall contributes to a higher assurance. These factors contribute to reducing the risk, and demonstrate engineering practices well above those of a typical cryptocurrency.

- The high complexity of the cryptography mechanisms in Sapling, the novelty of their application, as well as the relative complexity of the code, contribute to an increased security risk caused by unforeseen properties of its protocols. The Zcash protocol nonetheless exposes only a limited attack surface, and the implementation appears to be conform to the specification, which contribute to mitigate that risk.

The audit was lead by Dr. Jean-Philippe Aumasson, VP Technology at Kudelski Security, and was organized in several phases between February 2018 and June 2018, and took approximately 70 hours of work in total.

2 Pairing

The source code reviewed is in the repository `github.com/ebfull/pairing` in the branch master (commit `7b6e13b`, Feb 14 2018). This repository includes 6893 lines of Rust code, including all tests, 5760 lines excluding integration tests (in `test/`), and 3145 lines excluding integration and unit tests. We didn't review tests for security, but for coverage only, and as a basis for security tests.

As agreed with Zcash, side channel leaks should be ignored and not reported as security issues (e.g., timing leaks in exponentiation algorithms, or sensitive data held in memory).

We report the following findings: 1 low-severity issue, and 1 informational observation.

After approximately 16 hours of audit, we are confident that this implementation of cryptographic pairing is free of major bugs or other security shortcomings. Indeed, the attack surface is very limited, and the implementation performs all the necessary validations in order to ensure that potentially untrusted data can't be leveraged to abuse the program. Furthermore, the fact that the library is in Rust, a safety-oriented language, contributes to the high level of assurance. In particular, the Rust code does not include "unsafe" blocks that could potentially trigger memory corruption bugs, nor does it rely on libraries that do so in their functions used by pairing.

The issues reported below are more improvements in terms of best practice and defensive coding, rather than actual security issues.

2.1 PAIRING-001: Unbounded exponent in exponentiation

Severity: Low

Description

The field exponentiation `pow()` takes as exponent an unbounded array of `u64`'s, then performing a double-and-add algorithm. Very large exponents (unlikely to happen if the library is used properly) could slow down a program, but the risk of DoS is extremely low.

Recommendation

It may be faster to first reduce the exponent modulo the order of the field and then perform the actual exponentiation, and to set a hard limit on the size of the exponent.

Status

The library as used in Zcash's protocol does not allow users or attackers to control the exponent length, which makes this issue irrelevant for Zcash.

2.2 PAIRING-002: RNG choice delegated to callers

Severity: Informational

Description

The library does not define the random number generator (RNG) to be used, but instead an object implementing the `Rng` trait is passed as an argument to the `rand()`

methods. These methods in turn generate an (extension) field element by picking uniformly random field elements.

A risk is that callers could use an unsafe RNG when a safe one is needed. Some users may for example lazily copy from the test routines and use a `XorShiftRng`.

Note: While reviewing the randomness generation, we noticed the use of `rand::distributions::range::Range` in some tests, and observed that the `SampleRange::sample_range()` method does not properly check the range bounds.

Recommendation

In general, it would be safer to directly use Rust's `OsRng` safe RNG. However, there was probably good reasons not to use it in the first place.

Status

This is a design choice to allow callers to use the PRNG of their choice, nothing will be changed.

3 Bellman

The source code reviewed is in the repository `github.com/ebfull/bellman` in the branch master (commit `33feb37`, Mar 5 2018). This repository includes 2914 lines of Rust code, including all tests, 2752 lines excluding integration tests (in `test/`), and 1926 lines excluding integration and unit tests. We didn't review tests for security, but for coverage only, and as a basis for security tests.

As agreed with Zcash, side channel leaks should be ignored and not reported as security issues (e.g., timing leaks in exponentiation algorithms, or sensitive data held in memory).

We report the following findings: 1 low-severity issue, and 2 informational observations.

After approximately 15 hours of audit, we believe that this library is likely free of major bugs or other security shortcomings. The audit work consisted in code review as well as dynamic analysis using modified version of the test routines.

The attack surface is very limited, especially as used in the context of Zcash's Sapling protocol. As with the pairing library (used in bellman), Rust contributes to the high level of assurance. Although certain dependencies include "unsafe" blocks of code, this code does not seem to be used in bellman.

The issues reported below are more improvements in terms of best practices and defensive coding, rather than actual security issues.

3.1 BELLMAN-001: Outdated and unmaintained dependencies

Severity: Informational

Description

Among the third-party dependencies,

- `crossbeam` is defined as "outdated"[1] and as an "early work in progress"[2]. A risk is that this library may contain unstable code that could make bellman less reliable.
- `bit_vec` is "in maintenance mode (..) and will generally no longer be improved"[3]. A risk is that bugs found in this library would not be fixed. We did a quick review of this library, and only noticed a potential division by zero when a `BitBlock` is empty, but this does not seem to affect bellman.

Recommendation

If possible, use more actively maintained libraries.

Status

N.A.

3.2 BELLMAN-002: Lossy cast to usize

Severity: Informational

[1] `https://crates.io/crates/crossbeam`
[2] `https://docs.rs/crossbeam/0.3.2/crossbeam/`
[3] `https://github.com/contain-rs/bit-vec/blob/master/README.md`

Description

`Parameters::read()` includes multiple occurrences of

```
let len = reader.read_u64::<BigEndian>()? as usize;
```

This reads eight bytes but will cast to a 4-byte `usize` on 32-bit platforms. Here the data read is potentially controlled by a user of the library, and sending a too large `len` could overflow the `usize` and thereby allow many different encodings of a same set of parameters on 32-bit platforms. (On 64-bit platforms, see BELLMAN-003.)

Recommendation

If this is deemed to be a non-negligible risk (possibility of 32-bit platforms, or user-controlled parameters), then a simple check for the `u64`'s value would eliminate the risk.

Status

N.A.

3.3 BELLMAN-003: Potential DoS through user-controlled length argument

Severity: Low

Description

This is related to BELLMAN-002, and caused by lines such as the following, in `Parameters::read()` and `VerifyingKey.read()`:

```
let len = reader.read_u64::<BigEndian>()? as usize;
for _ in 0..len {
    h.push(read_g1(&mut reader)?);
}
```

Here the data read is potentially controlled by a user of the library, and sending a too large `len` will cause `Parameters::read()` to fail when attempting to read more bytes than the buffer holds. If `unwrap()` is used in production as in the unit tests, then this would trigger a panic.

Recommendation

Ensure that production code will not panic in case of a read error.

Status

The library as used in Zcash does not allow users to trigger such a behavior, therefore nothing needs to be fixed.

4 Sapling

We partially audited the Sapling protocol update, as specified in `https://github.com/zcash/zips/blob/master/protocol/sapling.pdf`. The source code reviewed is in the repository `github.com/zcash-hackworks/sapling-crypto` in the branch master (commit `e639750`, Mar 22 2018), which includes the critical cryptography components. This repository includes 6880 lines of Rust code, including all tests, and 3845 lines excluding tests. We didn't review tests for security, but only used them and as a basis for security tests. We have not reviewed the general protocol implementation in `https://github.com/zcash/zcash`, except when it helped clarify our understanding of the documentation.

We first reviewed the documentation, in order to gain a good understanding of the protocol before reviewing the implementation. The Sapling specification is of great depth and complexity, so we focused on the most critical components, reviewing in particular (in no specific order):

- The general logic of shielded transactions with Sapling
- The cryptographic primitives
- The RedDSA signature and RedJubJub curve logic
- The relevance and correctness of security assumptions (IND-CCA security, collision resistance, and so on)
- The new and critical components, in particular the Pedersen commitment constructions
- Key derivation mechanisms, and potential risks of entropy loss

- The sampling methods distribution uniformity
- The randomness beacon

We did not find any shortcoming in these components. However, we did not perform a rigorous security analysis of the protocol and did not assess its provable security properties. For example, we reviewed the consistency of security levels across primitives and cryptographic constructions, but did not verify theoretical secure composition results.

We then reviewed the source code in the sapling-crypto repository. Generally the code is well structured, minimally commented but generally clear enough, although it implements complex operations. We looked for issues of the following type (in no specific order):

- Language-specific security issues or unexpected behavior, such as unsafe blocks of code or panics
- The processing and parsing of potentially untrusted inputs
- The security of critical dependencies, such as the rand library
- Discrepancies between the specification and the implementation
- Issues reported by static analyzers or code linters
- Coding errors or bugs in critical components, such as the Pedersen hash circuit

Again we didn't identify any meaningful security issue, although we noticed things that we initially thought to be potential problems.

For example, the logic implemented in the code does not always exactly match the specification (for example the order of operations), but implements it in a functionally equivalent way (for example regarding elements' order validation).

Specifically, we carefully examined the critical cryptography components and their interactions.

We noticed that the version of the rand library is not the latest one (0.5 vs 0.4), however this is not a security problem since 0.5 did not fix security issues in 0.4, and

includes many breaking changes that prevent a straightforward upgrade of the package in sapling.crypto.

We reviewed tests' coverage and correctness, however we could not review the correctness of the implementation against other implementations' test vectors, since several of the components implemented are new and had no reference implementations.

Overall, we believe that the highest risk to Zcash's security is a flaw in its protocol logic and theoretical properties, rather than in its implementation. The most critical components (such as the Pedersen hash and circuit creation) would nonetheless benefit from a more careful review by a specialist, to ensure that the version implemented totally matches the specified logic.

5 About

Kudelski Security is an innovative, independent Swiss provider of tailored cyber and media security solutions to enterprises and public sector institutions. Our team of security experts delivers end-to-end consulting, technology, managed services, and threat intelligence to help organizations build and run successful security programs. Our global reach and cyber solutions focus is reinforced by key international partnerships.

Kudelski Security is a division of Kudelski Group. For more information, please visit `https://www.kudelskisecurity.com`.

Kudelski Security
route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland