Monero Bulletproofs Security Audit

Final Report, 2018-07-23

FOR PUBLIC RELEASE

Sponsored by

https://ostif.org

Contents

1 Summary

Monero is a decentralized, open-source cryptocurrency that provides strong privacy protections thanks to state-of-the-art cryptographic components.

Monero is deploying a new version of its protocol, using the "bulletproofs" NIZK proofs by Bünz et al.[1].

The Monero community hired Kudelski Security to perform a security assessment of the new protocol's implementation, based on the C++ implementation at https://github.com/moneromooo-monero/bitmonero/ in the branch "bp-multi-aggregation-pippenger", under `src/ringct`. This implementation is expected to match the Java prototype at https://github.com/b-g-goodell/research-lab/blob/master/source-code/StringCT-java/src/how/monero/hodl/bulletproof/MultiBulletproof.java.

This document reports the security issues identified and our mitigation recommendations, as well as our general assessment of the bulletproofs implementation. A "status" section reports the feedback from Monero developers, and includes a reference to the patches related to the issues reported. One issue present in the initial report was removed from the final report, after developers noticed that we had misunderstood the expected functionality.

We report:

- 3 potential security issues of allegedly low severity
- 8 observations related to general code safety, which we noticed in bulletproofs

[1] https://eprint.iacr.org/2017/1066.pdf

implementations or in other component of the Monero code base.

We did not identify substantial discrepancies between the Java version and C version of the integration of bulletproofs in RingCT. As suggested in this report, it would be safer to unit test lower-level mathematical operations, both in terms of correctness and matching with established test values.

Having spent approximately 75 hours on the project, we would like to stress that further analysis could well uncover dangerous code execution paths which we were unable to follow due to limited time. The faults which we observed indicate that further QA should be spent on the implementation. For example, the code base would likely benefit from additional fuzz testing.

The audit was lead by Dr. Jean-Philippe Aumasson, VP Technology, jointly with Yolan Romailler, Cryptography Engineer.

This audit was organized with the support of the OSTIF, thanks to their primary sponsor, Private Internet Access, and the fundraising done by the Monero Community.

2 Findings

This section reports security issues found during the audit.

The “status” section includes feedback from the developers received after delivering our draft report.

2.1 BP-F-001: Unsafe use of environment variables

Severity: Low

Description

The relevant code is in bulletproofs.cc:

```
static inline rct::key multiexp(const std::vector<MultiexpData> &data, bool HiGi)
{
  static const size_t STEP = getenv("STRAUS_STEP") ? atoi(getenv("STRAUS_STEP")) : 0;
  if (HiGi || data.size() < 1000)
    return straus(data, HiGi ? HiGi_cache: NULL, STEP);
  else
    return
```

Here `getenv()` gets the value of an environment variables, which is more likely to be an attack vector than a config file (with appropriate permissions). Furthermore, an invalid value for `STRAUS_STEP` will not be detected:

- If a string that is not a number is given, then `atoi()` will just return 0 (and 192 will be used as a `STEP` value in `straus()`)
- If a negative value is given, then it will be cast to an unsigned `size_t`, potentially changing a low-absolute-value negative number to a large integer.

This potentially allows an attacker to influence the efficiency of the implementation of Straus' exponentiation algorithm.

Recommendation

Use a configuration file (with proper permissions) rather than environment variables. Check that the value of `STRAUS_STEP` is a positive integer.

Status

This function no longer uses environment variables to set this value, as patched in commit `68f7606`.

2.2 BP-F-002: Lack of input validation in prover

Severity:

Low

Description

`bulletproof_PROVE()` takes two arguments v and γ (using the paper's notation) that are $\mathbb{Z}_p$ where $p = 2^{252} + 27742317777372353535851937790883648493$.

However that function will not check that its arguments are less than p, and will return a success error code upon invalid input. Also, degenerate values 0 and 1 are also accepted as values for v and γ. These values are not disallowed from the mathematical specification of the function, but due to their specific behavior they may lead to insecure operations.

Likewise, the elements of a proof in `bulletproof_VERIFY()` aren't explicitly checked. Only the representation of `V` happens to be checked within `ge_frombytes_vartime()`.

Recommendation

Check that the arguments belong to $\mathbb{Z}_p$ (or to their respective domain).

Status

Input scalars are now checked to ensure they are within the proper range in the prove and verify routines, as patched in `68f7606`. However, it should be noted that in the prove routine, an attacker can always supply invalid scalars regardless of code checks.

2.3 BP-F-003: Integer overflow in bulletproof L size computation

Severity:

Low

Description

`size_t n_bulletproof_amounts()` will overflow if `proof.L.size()` is larger than 69:

```
size_t n_bulletproof_amounts(const Bulletproof &proof)
{
        CHECK_AND_ASSERT_MES(proof.L.size() >= 6, 0, "Invalid bulletproof L size");
        return 1 << (proof.L.size() - 6);
}

size_t n_bulletproof_amounts(const std::vector<Bulletproof> &proofs)
```

```
{
        size_t n = 0;
        for (const Bulletproof &proof: proofs)
        {
                size_t n2 = n_bulletproof_amounts(proof);
                if (n2 == 0)
                        return 0;
                n += n2;
        }
        return n;
}
```

Consequently, `size_t n_bulletproof_amounts()` will return 0 if any of the proofs in the vector received is longer than 69. (Note that this can depend on the compiler, and might also result in undefined behavior.)

This function is called in `is_canonical_bulletproof_layout()` where a 0 return value will cause the function to return `true` without entering the `while` loop, and therefore without doing the checks it is supposed to do.

When `is_canonical_bulletproof_layout()` is called in `core::handle_incoming_tx_accumulated_batch()`, a `true` return value will lead the proofs vector to be added to the delayed batch verification vector, despite the invalide `L` value contained in one of the proofs.

This behavior, if unintended, may be exploited in order to force the verification of proofs with non-canonical layout.

Recommendation

Have correct boundary checks to avoid the overflow.

Status

This has been patched in `68f7606`.

3 Observations

This section reports various observations that are not security issues to be fixed.

3.1 BP-O-001: Possible DoS in the Java version

Unlike the C version that checks the maximum length M of a proof before setting `maxMN` to 2^M, the Java version accepts any value of `maxLength` in the verification function:

```java
public static boolean VERIFY(ProofTuple[] proofs)
{
    // Figure out which proof is longest
    int maxLength = 0;
    for (int p = 0; p < proofs.length; p++)
    {
        if (proofs[p].L.length > maxLength)
            maxLength = proofs[p].L.length;
    }
    int maxMN = (int) Math.pow(2,maxLength);
```

Since `maxMN` is a 32-bit signed integer, and the `double` result is cast to an `int`, this code will only behave correctly for proofs of length up to 30.

Status

The prototyping Java code was not written for use in production, and for this reason does not implement this DoS protection. Therefore nothing will be changed.

3.2 BP-O-002: Unit tests have no test vectors

The unit tests in `tests/unit_tests/bulletproofs.cpp` cover the code used in production checking that the functionality works as expected, a few edge cases. However it contains no reproducible test values to check this implementation against another one. In particular these tests don't check that the implementation matches the Java version.

Test vectors can't be directly generated since the generation of a proof is not deterministic, but a typical trick is to use a DRBG with a fixed seed. Having reproducible test values would also help debugging third-party implementations.

Status

The Java and C++ implementations produce proof elements that are not compatible with each other. Therefore, their test vectors would not match.

3.3 BP-O-004: Functions and variables naming improvements

The naming of functions and variables could be improved to make the code easier to understand and to match against the Bulletproofs paper. In particular:

- The `skGen()` and `skvGen()` functions are often used not to generate secret keys but just to get random scalars (which they do). It would be cleaner to have a function to generate random values, and call it from a key generation function as needed.
- The g and h generators from the paper (e.g. as in equation 52) are respectively `H` and `G` in the code, via `rct::scalamultH()` and `rct::scalarmultBase`. The same change of notation is made in the Java version.
- The type `rct::key` is used to hold different objects that should not necessarily be interoperables, such as scalars, or curve points, typically storing private keys and public keys.

Status

The listed functions and variables are named according to their use elsewhere in production code. Therefore nothing will be changed.

3.4 BP-O-005: Conversion functions leak amount range

The conversion functions `d2h()`, `d2b()`, `h2b()`, `b2h()` don't run in constant-time with respect to the XMR value converted and therefore may leak the range of the amount.

This seems unlikely to be a security issue, but this is easily avoided.

For example:

```
void d2h(key & amounth, const xmr_amount in) {
    sc_0(amounth.bytes);
    xmr_amount val = in;
    int i = 0;
    while (val != 0) {
        amounth[i] = (unsigned char)(val & 0xFF);
        i++;
        val /= (xmr_amount)256;
    }
}
```

Status

Local transaction code is not required to run in constant time in the considered threat model. Therefore nothing will be changed.

3.5 BP-O-006: Statistical bias in randomness used for testing

The modular reduction introduces a bias in `randXmrAmount()`:

```
//generates a random uint long long (for testing)
xmr_amount randXmrAmount(xmr_amount upperlimit) {
    return h2d(skGen()) % (upperlimit);
}
```

This function is only used in testing routines though, therefore this is not a security risk.

Using rejection sampling would eliminate the bias.

Status

Nothing will be changed.

3.6 BP-O-007: Condition based on modulo 1

Certain checks are relying on condition checked against some value % `NUM_BLOCKS_PER_CHUNK` For example:

```
if (m_cur_height % NUM_BLOCKS_PER_CHUNK == 0) {
        num_blocks_written += NUM_BLOCKS_PER_CHUNK;
}
```

but the `NUM_BLOCKS_PER_CHUNK` is defined as being 1, and operation modulo 1 are always equal to 0.

This seems to be "in case multi-block chunks are later supported", but certain code path are never taken as a consequence:

```
if (m_cur_height % NUM_BLOCKS_PER_CHUNK != 0)
{
        flush_chunk();
}
```

There are other such occurrences where a modulo 1 reduction take place.

Status

This code was originally written with future expansion in mind.

3.7 BP-O-008: Undefined behavior shifting signed value

In `crypto/crypto-ops.c`, the following function shifts a signed 64-bit value (left of right):

```
static int64_t signum(int64_t a) {
  return (a >> 63) - ((-a) >> 63);
}
```

As reported by cppcheck, the shift of a signed type is undefined or implementation-defined (ISO C99, 6.5.7, 1193 & 1196).

Status

The function has been rewritten using a ternary operator in `68f7606`.

4 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