



Monero Bulletproofs Security Audit

Draft Report, 2018-07-14

FOR PUBLIC RELEASE



Contents

1	Summary	3
2	Findings	5
2.1	BP-F-001: Unsafe use of environment variables	5
2.2	BP-F-002: Incorrect exponentiation algorithm	6
2.3	BP-F-003: Lack of input validation in prover	7
2.4	BP-F-004: Integer overflow in bulletproof L size computation	8
3	Observations	10
3.1	BP-O-001: Possible DoS in the Java version	10
3.2	BP-O-002: Unit tests have no test vectors	11
3.3	BP-O-004: Functions and variables naming improvements	11
3.4	BP-O-005: Conversion functions leak amount range	11
3.5	BP-O-006: Statistical bias in randomness used for testing	12
3.6	BP-O-007: Condition based on modulo 1	12
3.7	BP-O-008: Undefined behavior shifting signed value	13

4 About

14

1 Summary

Monero is a 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 Foundation 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.

We report:

- 4 potential security issues of low or undetermined severity
- 8 observations related to general code safety, which we noticed in bulletproofs 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. Although we didn’t find major security issues, the relative complexity of the new protocol combined with the limited set of tests (in

¹<https://eprint.iacr.org/2017/1066.pdf>

particular, without test values) is a potential risk. As suggested in our report, it would be safer to unit test lower-level mathematical operations, both in terms of correctness and matching of established test values.

More generally, the code base would likely benefit from fuzz testing and automated static analysis.

The audit was lead by Dr. Jean-Philippe Aumasson, VP Technology, jointly with Yolan Romainier, Cryptography Engineer, and took approximately 65 hours, reporting included.

2 Findings

This section reports security issues found during the audit.

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

Severity: Low

Description

The relevant code is in bulletproofs.cc:

```
1 static inline rct::key multiexp(const std::vector<MultiexpData> &data, bool HiGi)
2 {
3     static const size_t STEP = getenv("STRAUS_STEP") ? atoi(getenv("STRAUS_STEP")) : 0;
4     if (HiGi || data.size() < 1000)
5         return straus(data, HiGi ? HiGi_cache: NULL, STEP);
6     else
7         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 `_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.

2.2 BP-F-002: Incorrect exponentiation algorithm

Severity:

?

Description

The function `vector_power_sum()` does not what is supposed to do:

```
1  /* Given a scalar, return the sum of its powers from 0 to n-1 */
2  static rct::key vector_power_sum(const rct::key &x, size_t n)
3  {
4      if (n == 0)
5          return rct::zero();
6      rct::key res = rct::identity();
7      if (n == 1)
8          return res;
9      rct::key prev = x;
10     for (size_t i = 1; i < n; ++i)
11     {
12         if (i > 1)
13             sc_mul(prev.bytes, prev.bytes, x.bytes);
14         sc_add(res.bytes, res.bytes, prev.bytes);
```

```
15     }  
16     return res;  
17 }
```

When the exponent argument n is zero this algorithm returns 0 whereas $x^0 = 1$ for any x ; when the exponent is 1 this algorithm returns 1 whereas $x^1 = x$.

This incorrect behavior may compromise the verification of a bulletproof, where it is used.

Likewise, `vector_power()` is incorrect.

This is not obviously exploitable, but not obviously unexploitable either.

Recommendation

Compute exponentiations correctly, unless there is a legitimate reason for that behavior.

2.3 BP-F-003: 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 valid values for v and γ .

As a consequence, invalid values passed accidentally or maliciously will not be detected.

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).

2.4 BP-F-004: 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:

```
1  size_t n_bulletproof_amounts(const Bulletproof &proof)
2  {
3      CHECK_AND_ASSERT_MES(proof.L.size() >= 6, 0, "Invalid bulletproof L size");
4      return 1 << (proof.L.size() - 6);
5  }
6
7  size_t n_bulletproof_amounts(const std::vector<Bulletproof> &proofs)
8  {
9      size_t n = 0;
10     for (const Bulletproof &proof: proofs)
11     {
12         size_t n2 = n_bulletproof_amounts(proof);
13         if (n2 == 0)
14             return 0;
15         n += n2;
16     }
```

```
17     return n;  
18 }
```

Consequently, `size_t n_bulletproof_amounts()` will return 0 if any of the proofs in the vector received is longer than 69.

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 true is returned,

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 invalid 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.

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:

```
1  public static boolean VERIFY(ProofTuple[] proofs)
2  {
3      // Figure out which proof is longest
4      int maxLength = 0;
5      for (int p = 0; p < proofs.length; p++)
6      {
7          if (proofs[p].L.length > maxLength)
8              maxLength = proofs[p].L.length;
9      }
10     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.

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.

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.

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:

```
1 void d2h(key & amounth, const xmr_amount in) {
2     sc_0(amounth.bytes);
3     xmr_amount val = in;
4     int i = 0;
5     while (val != 0) {
6         amounth[i] = (unsigned char)(val & 0xFF);
7         i++;
8         val /= (xmr_amount)256;
9     }
10 }
```

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

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

```
1 //generates a random uint long long (for testing)
2 xmr_amount randXmrAmount(xmr_amount upperlimit) {
3     return h2d(skGen()) % (upperlimit);
4 }
```

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

Using rejection sampling would eliminate the bias.

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:

```
1  if (m_cur_height % NUM_BLOCKS_PER_CHUNK == 0) {
2      num_blocks_written += NUM_BLOCKS_PER_CHUNK;
3  }
```

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

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

```
1  if (m_cur_height % NUM_BLOCKS_PER_CHUNK != 0)
2  {
3      flush_chunk();
4  }
```

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):

```
1  static int64_t signum(int64_t a) {
2      return (a >> 63) - ((-a) >> 63);
3  }
```

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

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

This report and all its content is copyright (c) Nagravision SA 2018, all rights reserved.