Beam-mw Security Audit

Final Report, 2018-12-21

FOR PUBLIC RELEASE

Contents

1 Summary

Beam-mw is a confidential cryptocurrency based on the MimbleWimble protocol, which is MimbleWimble is a cryptographic construction based on the Pedersen commitments and elliptic-curve cryptography. Beam-mw provides for confidential transactions, transaction cut-through, and notably uses range proofs based on the bulletproofs construction.

Beam hired Kudelski Security to perform a security assessment of their blockchain, providing access to source code and documentation. The repository concerned is: https://github.com/BeamMW/beam we specifically audited commits and changed to commit `ec633b71` during the work, because of the significant changes that occurred on the codebase during the audit.

This document reports the security issues identified and our mitigation recommendations, as well as our general assessment of the implementation and architecture. A "Status" section reports the feedback from Beam's developers, and includes a reference to the patches related to the reported issues.

We report:

- 2 security issues of medium severity
- 7 security issues of low severity
- 10 observations related to general code safety

The audit was performed by Dr. Jean-Philippe Aumasson, VP Technology, jointly with Yolan Romailler, Cryptography Engineer, and involved 20 person-days of work.

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 BEAM-F-001: Invalid SHA-256 hashes on 32-bit platforms

Severity: Low

Description

On 32-bit platforms, repeated calls to `Hash::Processor::Write(const void* p, uint32_t n)` with large buffer size may overflow `hash->bytes` (`size_t`), the byte count in SHA-256:

```
static void secp256k1_sha256_write(secp256k1_sha256_t *hash, const unsigned char
 ↪  *data, size_t len) {
    size_t bufsize = hash->bytes & 0x3F;
    hash->bytes += len;
    while (bufsize + len >= 64) {
        /* Fill the buffer, and process it. */
        memcpy(((unsigned char*)hash->buf) + bufsize, data, 64 - bufsize);
        data += 64 - bufsize;
        len -= 64 - bufsize;
        secp256k1_sha256_transform(hash->s, hash->buf);
        bufsize = 0;
    }
    if (len) {
        /* Fill the buffer with what remains. */
        memcpy(((unsigned char*)hash->buf) + bufsize, data, len);
    }
}
```

As a result, the padding will be invalid and the hash value invalid as well.

`Hash::Processor` is used for nonce generation and within oracle operations, but it seems that Beam will not call it with sufficiently large values to trigger this misbehavior.

Recommendation

Check that no overflow occurs, or forcing the use of a 64-bit byte counter.

Status

Beam decided not to fix this, because there are no data of indefinitely large size that gets hashed and also because this is located in a part of the code provided by a third party.

2.2 BEAM-F-002: Table-based AES implementation potentially vulnerable to cache-timing attacks

Severity: Low

Description

The AES implementation in `aes.cpp` is table-based, by default generating tables on the fly. Such an implementation is generally vulnerable to cache-timing attacks, potentially allowing an attacker to recover plaintexts and/or keys when having partial access to the host machine.

Recommendation

Most platforms where the application will run will support native AES instructions (AES-NI), hence we recommended defaulting to these, and falling back on the table-based version otherwise.

Status

Beam decided not to fix this.

2.3 BEAM-F-003: ECB mode AES encryption (and lack of authentication)

Severity: Medium

Description

The keystore file is encrypted with AES in ECB mode, which would allow to identify duplicate blocks and to trivially forge ciphertexts by swapping blocks:

```
void aes_encrypt(void* buffer, size_t bufferLen, const void* password, size_t
↪  passwordLen) {
    AES::Encoder enc;
    init_aes_enc(enc, password, passwordLen);
    uint8_t* p = (uint8_t*)buffer;
    uint8_t* end = p + bufferLen;
    for (; p<end; p+=AES::s_BlockSize) {
        enc.Proceed(p, p);
    }
}
```

Furthermore, the ciphertext is not authenticated, allowing an attacker to manipulate the encrypted message in a way that will not be detected when decrypting it.

Recommendation

We recommend to use an authenticated cipher, such as the standard AES-GCM.

Status

Beam entirely removed the keystore part of the code, and is now relying on AES-CTR when it needs a stream cipher.

2.4 BEAM-F-004: Weak password key derivation

Severity: Medium

Description

The keystore encryption key is directly taken as the SHA-256 of the password, allowing efficient bruteforce search of the password and possibly offline attacks if one of the blocks is predictable:

```
void init_aes_enc(AES::Encoder& enc, const void* password, size_t passwordLen) {
    ECC::NoLeak<ECC::Hash::Processor> hp;
    ECC::NoLeak<ECC::Hash::Value> key;
    hp.V.Write(password, passwordLen);
    hp.V >> key.V;
    enc.Init(key.V.m_pData);
}
```

Recommendation

We recommend to use a password hashing function that mitigates bruteforce attacks by being slow, such as PBKDF2 (with at least 50000 iterations) or Argon2.

Status

Beam fixed this by removing the weak password derivation.

2.5 BEAM-F-005: Possibly unsafe file size computation

Severity: Low

Description

In `peer_storage.cpp` and `keystore.cpp`, a file's size is computed using the `fseek()` and `ftell()` combination, for example as follows:

```
static bool seek_end(FILE* f, long& offset) {
    if (fseek(f, 0, SEEK_END) != 0) return false;
    offset = ftell(f);
    return true;
}
```

However, on non-POSIX systems this is undefined behavior, as described in §7.21.9.2 and §7.21.9.4 of the C standard (ISO/IEC 9899:2011), as summarized in `https://wiki.sei.cmu.edu/confluence/x/o9YxBQ`.

Recommendation

As commented in the above linked page, use `fseeko()`/`ftello()` or simply `fstat()`.

Status

Beam fixed this by removing completely this part of the code and is now handling this through its `FStream` class.

2.6 BEAM-F-006: Potential integer overflow is size counter

Severity: Low

Description

In the `Counter` update mechanism in `serialize.h`, an overflow of `m_Value` could occur:

```
struct SerializerSizeCounter
{
        struct Counter
        {
                size_t m_Value;

                size_t write(const void * /*ptr*/, const size_t size)
                {
                        m_Value += size;
                        return size;
                }

        } m_Counter;
```

This seems hard to exploit on 64-bit platforms.

Recommendation

We suggest to check that `(size_t)(m_Value + size)` is greater than `m_Value`, in order to ensure that no overflow occurs.

Status

Beam added debug assertions to check that no overflow occur, but this class is not used for indefinitely large data and so no checks are performed in release build.

2.7 BEAM-F-007: Non-constant-time elliptic curve equality operator

Severity: Low

Description

The following function in `ecc.cpp` checks if a memory region is all-zero, returning false as soon as it finds a non-zero byte:

```
bool memis0(const void* p, size_t n)
{
        for (size_t i = 0; i < n; i++)
                if (((const uint8_t*)p)[i])
                        return false;
        return true;
}
```

For an arbitrary memory region with random bytes the function will return on average after one iteration, whereas it will perform `n` iterations for an all-zero region. This timing difference may be measurable for high values of `n`, and this timing difference then leaks the return value of the function.

A similar leak occurs in the general equality operator of a `Scalar`:

```
bool Scalar::Native::operator == (const Native& v) const
{
    for (size_t i = 0; i < _countof(d); i++)
        if (d[i] != v.d[i])
            return false;
    return true;
}
```

Notice that we could not find an easy way to exploit these leaks.

Recommendation

The function can easily be rewritten to be constant-time, for example as follows:

```
bool memis0_ct(const void* p, size_t n)
{
        uint8_t b=0;
        for (size_t i = 0; i < n; i++)
                b |= ((const uint8_t*)p)[i];
        return (0 == b);
}
```

A same trick can be used for the general == operator.

Status

Beam fixed this by avoiding using `memis0()` and by using `Scalar::Native::operator == (Zero_)` instead, which is constant-time.

2.8 BEAM-F-008: Potentially unsafe use of /dev/urandom

Severity: Low

Description

ECC's `GetRandom()` method calls the `/dev/urandom` device file on non-Windows platforms to get some random bytes:

```
    int hFile = open("/dev/urandom", O_RDONLY);
    if (hFile >= 0)
    {
        if (read(hFile, p, nSize) == nSize)
            bRet = true;

        close(hFile);
    }
```

This checks that the file is properly open and that the number of bytes read is equal to the number of bytes requested. However, it's missing a number of checks (see for example LibreSSL's getentropy_urandom()) and in particular the guarantee that the PRNG has accumulated enough entropy.

Recommendation

We recommend to use Linux' `getrandom()` syscall instead of direct access to `dev/urandom`.

Status

Beam will not fix this, since their nonces are not solely relying on the machine provided randomness, by leveraging a deterministic method with some entropy. This means the quality of the entropy does not affect the scheme too much.

2.9 BEAM-F-009: Cut-trough is never applied in the transaction merging/creation process

Severity: Low

Description

The MimbleWimble protocol provides with a so-called "cut-through" feature, which is applied in two different cases: when transactions are constructed or merged, and when blocks are merged. However this was never applied in the first case, since in the function `TxVectors::Perishable::NormalizeP` performing the cut-through, a `for` loop was ignored because of incorrect boundaries:

```
size_t i1 = m_vOutputs.size();
for (size_t i0 = 0; i0 < m_vInputs.size(); i0++)
{
        Input::Ptr& pInp = m_vInputs[i0];

        for (; i1 < m_vOutputs.size(); i1++)
        {
                Output::Ptr& pOut = m_vOutputs[i1];

                int n = TxBase::CmpInOut(*pInp, *pOut);
                if (n <= 0)
                {
                        if (!n)
                        {
                                pInp.reset();
                                pOut.reset();
                                nDel++;
                        }
                        break;
                }
        }
}
```

As you can see, the inner `for` loop is ignored since the `i1` variable is already set to `m_vOutputs.size()` outside of the outer `for` loop. This prevents any cut-through in the transaction merging/creation process and causes the `nDel` value to always be 0.

Since this is verified in the `TxBase::Context::ValidateAndSummarize` function, the consequence of this bug is that under certain circumstances a node could produce a block or broadcast a merged transaction which would be banned by all other parties, since it would fail the verification that all transactions elements are in a well-defined order and that all the spent outputs are already eliminated, which would not have been the case here.

Recommendation

We recommend reviewing this cut-through process and fixing it. Furthermore, adding more unit tests would greatly help to catch such bugs.

Status

This has been fixed in the current codebase.

3 Observations

This section reports various observations that are not security issues to be fixed, such as improvement or defense-in-depth suggestions.

3.1 BEAM-O-001: 2038 bug

The `Timestamp` type is `uint64_t` but the value computer will actually be a `time_t`, which will be 32-bit on 32-bit Linux platforms. If Beam happens to run on 32-bit devices in 2038, this could be an issue.

3.2 BEAM-O-002: `AmountBig` arithmetic does not handle wraparounds

Addition and subtraction operators defined for `AmountBig` will wraparound if an over/underflow happens. For example, for additions:

```
void AmountBig::operator += (Amount x)
{
    Lo += x;
    if (Lo < x)
        Hi++;
}

void AmountBig::operator += (const AmountBig& x)
{
    operator += (x.Lo);
    Hi += x.Hi;
}
```

This is normal behavior for a C integer type, however in the context of Beam an over/underflow of `AmountBig` should not occur unless something is wrong in the code or if the program is being abused. We suggest to add assert statements to check that no overflow happens.

Status

`AmountBig` is defined as a 128 bits integer, but the amount of a single UTXO is a 64 bits integer, which means that in practice this should never be a problem.

3.3 BEAM-O-003: Invalid `HeightRange` instances supported

Here the initializer will not complain if `h0 > h1`:

```
HeightRange(Height h0, Height h1) {
    m_Min = h0;
    m_Max = h1;
}
```

Status

It appears that this is actually a valid case of `HeightRange` instances, since the only way to define an empty `HeightRange` is to set it so that `m_Max<m_Min`. All the code handling `HeightRange` instances should be taking this into account already.

3.4 BEAM-O-004: Non POSIX compliance

The code includes structs and types with suffix `_t`, such as `secp256k1_hmac_sha256_t`. However this suffix is reserved for POSIX types.

Status

Since this is located in third party code, the Beam team prefers to avoid changing it. This is better for maintainability, as it would otherwise require them to maintain the modified third party code in the future.

3.5 BEAM-O-005: Hex character conversion misses bound checks

The following utility function in `ecc.cpp` converts a value between 0 and 15 to an hex character in [0-9a-f], however it will not check that the input value is less than 16:

```
char ChFromHex(uint8_t v)
        {
                return v + ((v < 10) ? '0' : ('a' - 10));
        }
```

Status

This should never be a problem since this function is used only locally on already-verified input.

3.6 BEAM-O-005: Deprecated Windows crypto API usage

ECC's `GetRandom()` method calls the deprecated CryptoAPI rather than the "next generation" CNG API:

```
#ifdef WIN32

HCRYPTPROV hProv;
if (CryptAcquireContext(&hProv, NULL, NULL, PROV_RSA_SCHANNEL, CRYPT_VERIFYCONTEXT))
{
    if (CryptGenRandom(hProv, nSize, (uint8_t*)p))
        bRet = true;
    verify(CryptReleaseContext(hProv, 0));
}
```

Instead, BCryptGenRandom() should be used.

Status

This has been fixed in the current codebase.

3.7 BEAM-O-006: Insecure HMAC key lengths supported

The HMAC API supports any key length, including 0-byte keys, which exposes the API to accidental misuse:

```
static void secp256k1_hmac_sha256_initialize(secp256k1_hmac_sha256_t *hash, const
 ↪  unsigned char *key, size_t keylen) {
int n;
unsigned char rkey[64];
if (keylen <= 64) {
    memcpy(rkey, key, keylen);
    memset(rkey + keylen, 0, 64 - keylen);
} else {
```

This doesn't seem exploitable since the application uses adequate key lengths, but can probably be fixed to add a layer of defense against misuse.

Status

Since this is located in third party code, the Beam team prefers to avoid changing it. This is better for maintainability, as it would otherwise require them to maintain the modified third party code in the future.

This should not be a problem as long as the Beam codebase is using secure key lengths.

3.8 BEAM-O-007: Linter usage is not enforced

In order to improve readability, reduce the risks of bugs and ensure code quality, we recommend enforcing the usage of linters such as Clang-tidy on the codebase. For example, using such linter would warn that in C++ it is better for type safety to use `nullptr` instead of `0` when a pointer is expected to be null. It might also detect problematic casts of a give type to another. Or allow to prune the code of unused code snippets (for example, the function `need_to_request_known_servers` does not appear to be used, and furthermore has a strange return value that is randomized if some previous conditions are not met.)

Overall the code quality could be improved by using such tools, which is always a good thing, and part of the so-called defense-in-depth approach to securing software.

3.9 BEAM-O-008: Non-constant time comparisons

In some places, including `uintBig_t::cmp`, which is used by `Scalar::cmp`, the function `memcmp` is used to compare buffers, which is non-constant and might slightly leak information.

A constant-time memcmp could be used to guarantee no leak occurs, by typically performing an OR operation on the XOR of the bytes of both buffers `ret |= *a++ ^ *b++`, while iterating over all bytes.

3.10 BEAM-O-009: Scheme might be sensitive to fault due to deterministic nonces

While it is theoretically fine to use deterministic nonce generation, it renders signature scheme such as ECDSA or Schnorr schemes more sensitive to faults[1].

[1]As discussed in https://link.springer.com/chapter/10.1007/978-3-319-44524-3_11.

We recommend, in order to avoid fault attacks when using deterministic schemes, to use a bit of entropy in the deterministic nonce generation, as done for instance in XEdDSA. This is possible using RFC6979 "additional data", and the quality of the added entropy needs not be up to usual cryptographic standards, typically using an available, non-blocking random generator from the operating system suffices.

Status

In order to mitigate this, the Beam team has added some entropy through the "additional data" field of the RFC6979 nonce generation method, so that now all nonces are generated using some random data along with the deterministic derivation.

3.11 BEAM-O-010: OpenSSL could be used in its latest version

Currently, it appears the codebase is relying on OpenSSL 1.0.2n, which is not the latest OpenSSL release, and is not future-proof, especially with the introduction of TLS 1.3. Furthermore the current SSL settings allow for a downgrade to TLS 1.1 or 1.0, the latest being now considered deprecated. We recommend migrating to OpenSSL 1.1.1, which is supported until 2023 and is including support for TLS 1.3.

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