Zilliqa's Schnorr Signatures Security Audit

Final Report, 2018-12-19

FOR PUBLIC RELEASE

Contents

1 Summary

Zilliqa is a new blockchain platform that is designed to scale in transaction rates. It features EC-Schnorr based multi-signatures to reduce the signature size when multiple signatures are required on a message.

Zilliqa hired Kudelski Security to perform a security assessment of their Schnorr implementation, providing access to source code, documentation, and review guidelines including references to the most critical components.

The repositories concerned are:

- https://github.com/Zilliqa/Zilliqa/tree/master/src/libCrypto
  (using branch "master", commit `292cd0d0` on October 8th)
- https://github.com/Zilliqa/Zilliqa-JavaScript-Library
  (using branch "master", commit `56b3a5f` on October 19th).

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

We report:

- 1 security issue of high severity
- 1 security issue of medium severity
- 2 security issue of low severity
- 6 observations related to the codebase.

The audit was led by Yolan Romailler, Cryptography Engineer, and involved 5 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 ZILLIQ-F-001: the k values are not generated using state of the art methods

Severity: Low

Description

The security of the Schnorr signature system crucially relies on the randomness of the value k. Reusing k values, using predictable ones or biased ones can lead to a private key recovery.

Currently these values are generated using the following code:

```
606 // 1. Generate a random k from [1, ..., order-1]
607 do {
608         // -1 means no constraint on the MSB of k
609         // 0 means no constraint on the LSB of k
610         err =
611 (BN_rand(k.get(), BN_num_bits(m_curve.m_order.get()), -1, 0) == 0);
612         if (err) {
613                 LOG_GENERAL(WARNING, "Random generation failed");
614                 return false;
615         }
616 } while ((BN_is_zero(k.get())) ||
617         (BN_cmp(k.get(), m_curve.m_order.get()) != -1));
```

Recommendation

The same method as used by OpenSSL 1.1 in `BN_generate_dsa_nonce` should be preferred. This would consist of:

1. In the signature method, instantiate a PRNG.
2. Using `BN_ran_range`, draw a random value of the bit size of k.
3. Use this value along with the hash of the message and the private key to seed the PRNG.
4. Generate the value k using this seeded PRNG.

This could be also solved by simply using OpenSSL 1.1 and its `BN_generate_dsa_nonce` method.

Status

The Zilliqa team has addressed this problem using OpenSSL 1.1 and its `BN_generate_dsa_nonce` method, as recommended.

2.2 ZILLIQ-F-002: the `CommitSecret` constructor introduces a modulo bias

Severity: Medium

Description

When generating a value `commit->secret` that should be in $[2, ..., n-1]$, for n the order of the base point, the current code base is drawing a random value and reducing it modulo the curve order, which introduces a so-called "modulo bias" in `MultiSig.cpp`.

```
39 do {
40         const Curve& curve = Schnorr::GetInstance().GetCurve();
41
42         err = (BN_rand(m_s.get(), BN_num_bits(curve.m_order.get()), -1, 0) == 0);
43         if (err) {
44                 LOG_GENERAL(WARNING, "Value to commit rand failed");
45                 break;
```

```
46          }
47
48          err = (BN_nnmod(m_s.get(), m_s.get(), curve.m_order.get(), NULL) == 0);
            ↪  // This introduces a bias
49          if (err) {
50                  LOG_GENERAL(WARNING, "Value to commit gen failed");
51                  break;
52          }
53  } while (BN_is_zero(m_s.get()) || BN_is_one(m_s.get()));
```

Notice that this problem is not present in `PrivKey`, where another method is used, only in `CommitSecret`.

Since the CommitSecret values are equivalent to the k values in the multi-signature case, this could lead to lattice attacks of the same sort that we have already seen applied to ECDSA k value biases.

Notice that we did not quantify the bias introduced given the short engagement time.

Recommendation

Use OpenSSL's method `BN_rand_range` instead to securely generate a random integer in the desired range. Or use the same rejection sampling method as in `PrivKey`.

Status

The Zilliqa team has addressed this problem using OpenSSL's method `BN_rand_range` as recommended.

2.3 ZILLIQ-F-003: the last signer can forge multi-signatures

Severity: High

Description

The multi-signature scheme discussed in Zilliqa Whitepaper, as it is implemented in the audited library is vulnerable to forgeries by one of the co-signers and to so-called "key cancellation".

Let P_1 and d_1 be the public and private keys of a given signer, "the last signer", let m be the message to sign and let $P_2, \cdots, P_n$ be the public keys of the other signers.

When generating a multi-signature, if our signer can set his public key to be $P_1 - \sum_{i=2}^{n} P_i$ then, since the aggregated public key P is computed as being $P = \prod_{i=1}^{n} P_i$, the resulting aggregated public key would be

$$P = P_1 - \sum_{i=2}^{n} P_i + \sum_{i=2}^{n} P_i = P_1$$

for which that last signer knows the private key d_1.

This means a co-signer is able to perform a key cancellation attack against another co-signer or even against all other co-signers.

Recommendation

Use either a secure variant such as the one presented in the paper "Simple Schnorr Multi-Signatures with Applications to Bitcoin"[1] by Maxwell *et al*.

Or explicitly verify the binding of the advertised public key with the respective private key, using the method proposed in the whitepaper.

Status

It appears the verification proposed in the whitepaper, that the public keys are effectively linked to a valid private key, is performed in a non-audited part of the codebase. (We recall that this audit was only performed on the libCrypto part of Zilliqa's codebase.) Thus, this should not be exploitable and should not be a concern for Zilliqa.

2.4 ZILLIQ-F-004: lack of boundary checking in the TS version

Severity: Low

Description

In the `verify` method, the signature values r and s are supposed to be in the range $[1, \cdots, n-1]$ for n the order of the base point. However the lower bound is not enforced in the current TypeScript implementation. This could possibly be exploited by using invalid values (such as 0) in the signature, depending on the elliptic curve implementations used, since the input can be controlled by an attacker.

[1] https://eprint.iacr.org/2018/068.pdf

Recommendation

Perform all the required boundary checks to ensure the input is strictly within the expected range.

Status

The Zilliqa team has addressed this problem by implementing the required checks.

3 Observations

This section reports various observations that are not security issues to be fixed, but recommendations for improvement or defense-in-depth.

3.1 ZILLIQ-O-001: `BN_rand_range` is not used

Whenever a random value in a given range is required, the current codebase is using `BN_rand` on the range bit size in a `while` comparing the generated value with the range value to generate a value in that range. For an example, see the code snippet in section 2.1.

OpenSSL provides a method doing this, but with more checks called `BN_rand_range`. While not a security issue, it might be better to use OpenSSL's method from a maintainability point of view.

3.2 ZILLIQ-O-002: The signature scheme differs from the quoted reference document

The scheme is currently deviating from the standard in two points:

- the challenge r value is computed using $H(Q, P_A, m)$ instead of $H(Q, m)$. This is a breaking change that prevents interoperability with other implementations and which requires custom test vectors to be generated.
- the s value in the signature is created by subtracting the $r \cdot d_A$ value from the k value, whereas the document BSI-TR-03111 referred to in the code is using an addition instead. This makes the current implementation incompatible with an implementation following the aforementioned document as reference. It is

however possible to use the inverted public key as a public key that would then verify the signatures correctly.

However, these changes do not weaken the scheme and do not constitute a problem per se.

3.3 ZILLIQ-O-003: The current implementation is not compatible with key recovery

Since the current implementation is following the specification from BSI-TR-03111 section 4.2.3, the signatures are directly exposing the challenge value r, whereas it could instead expose the commitment $Q = kG$, which would allow to perform public key recovery from the message and the signature, provided the public key is not hashed into the challenge as currently done and noted above in section 3.2.

This is true since $Q = [s]G + [r]P_A$, and if r can be computed by knowing only Q and the message m. One could then compute $P_A = [r^{-1}](Q - [s]G)$, allowing us to derive the value P_A from the values Q, m and s.

Notice this would require the current r value of a signature to be replaced by the commitment point Q, and the verification process would need to be changed as well to be done as in the multi-signature case.

This “feature” might not be desirable and we only report this in order to be as informative as possible.

3.4 ZILLIQ-O-004: Exceptions are handled but not thrown

In many locations, code exists to handle exceptions that can never occur because none are thrown. This causes unreachable code and is not good for maintainability.

3.5 ZILLIQ-O-005: Code quality could be improved

The codebase is using OpenSSL and does so in the best way possible, using the required clearing methods. Which is a good indicator of good code quality. However, the codebase contains portions of code that are never visited, such as this `if` condition in `Schnorr.cpp` that is always true, since otherwise the code would have already returned.

```
697 if (!err) {
698         res = (BN_is_zero(result.m_r.get())) || (BN_is_zero(result.m_s.get()));
699 }
```

Other examples include the clearing of certain buffers that are not always required and yet performed, or variables whose scope might be reduced.

This might be a problem for maintainability. We recommend enforcing code reviews, shall it not be already the case, in order to increase code quality as much as possible and to integrate so-called “Continuous Inspection” tools in the development process.

3.6 ZILLIQ-O-006: The multi-signature is using random commits

This means that as long as care has been taken to never re-use a commit value, even if a previous multi-signature attempt failed, then the signature scheme is not vulnerable to the derandomization attacks that tend to occur when a deterministic nonce is used instead of a random value in such schemes.

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