KZen Curv Security Audit

Final Report, 2019-03-01

FOR PUBLIC RELEASE

Contents

1 Summary

KZen Curv is a library written in Rust providing low-level elliptic curve cryptography functionalities (ECC), as well as higher-level protocols such as key-exchange, secret sharing, zero-knowledge, and multi-party computation.

KZen hired Kudelski Security to perform a security assessment of their solution, providing access to source code and documentation. The repository concerned is: https://github.com/KZen-networks/curv we specifically audited commits 42ea2b7.

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 developers, and includes a reference to the patches related to the reported issues.

We report:

- 3 security issues of medium severity
- 5 security issues of low severity
- 17 observations related to general code safety

After the audit, KZen patched their codebase accordingly in the new release of Curv (commit 5f2da7f): https://github.com/KZen-networks/curv/commit/5f2da7f5c435fad697782d1dd8adbdc605417fc2.

The audit was performed jointly by Dr. Tommaso Gagliardoni, Cryptography Expert, and Yolan Romailler, Senior Cryptography Engineer, with support of Dr. Jean-Philippe Aumasson, VP of Technology, and involved 13 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 KZENC-F-001: Biased Ed25519Scalar Random Generation

Severity: Medium

Description

In `ed25519.rs`, the `new_random()` function is implemented as follows:

```
fn new_random() -> Ed25519Scalar {
            let mut scalar_bytes = [0u8; 32];
            let rng = &mut thread_rng();
            rng.fill(&mut scalar_bytes);
            let rnd_bn = BigInt::from(&scalar_bytes[..]);
            let rnd_bn_mod_q = BigInt::mod_mul(&rnd_bn, &BigInt::from(8), &FE::q());
            ECScalar::from(&rnd_bn_mod_q)
      }
```

This introduces a modulo bias on the value of the `ECScalar`, since it is not generated using rejection sampling, but using a modulo reduction to the value q.

Recommendation

Use rejection sampling as it is currently done in `Samplable` for `Mpz`.

Status

Corrected in new release.

2.2 KZENC-F-002: Secret Data Variables Not Zeroized After Use

Severity: Medium

Description

Variables containing sensitive data (e.g., secret keys, curve points used in intermediate computation, etc) are not overwritten after use. This might potentially leave sensitive data in some areas of memory.

As an example, in `sigma_ec_ddh.rs`, the `NISigmaProof::prove()` function is implemented as follows:

```
    fn prove(w: &ECDDHWitness, delta: &ECDDHStatement) -> ECDDHProof {
        let s: FE = ECScalar::new_random();
        let a1 = &delta.g1 * &s;
        let a2 = &delta.g2 * &s;

        let e =
            HSha256::create_hash_from_ge
                (&[&delta.g1, &delta.h1, &delta.g2, &delta.h2, &a1, &a2]);
        let z = s + e.clone() * &w.x;
        ECDDHProof { a1, a2, z }
    }
```

This leaves unzeroized the secret scalar `s`, which might allow to recover the witness from the proof.

Recommendation

Always overwrite with zeroes any variable containing potentially sensitive data after use. In Rust, this can be easily done with the `clear_on_drop` crate.

Status

Corrected in new release.

2.3 KZENC-F-003: Potential Side-Channel Timing Attack Due to Use of GMP

Severity: Low

Description

Operations on `BigInt` data type are performed using the GMP library. However, in GMP the modulo operation `mpz_mod()` is not constant time, so it will leak data that could potentially be used to recover secret keys.

Recommendation

The potential timing leak is implementation-specific here, and difficult to quantify - but likely very low. GMP is used in a plethora of cryptographic tools, probably the best that could be done is try to measure said timing attack channel and assess its risk of exploitability.

Status

KZen acknowledged the issue.

2.4 KZENC-F-004: Potential Length Extension Attack with SHA-256

Severity: Low

Description

In `create_commitment_with_user_defined_randomness`, SHA-256 is used to hash $(\text{mess}|\text{rand})$, and rand can be controlled by the user, which means that this is vulnerable to a hash length extension attack.

Recommendation

Although it is unclear whether in the particular case the above issue could lead to a practical attack, it would be better to avoid length-extension altogether, for example by using SHA-3 instead of SHA-256.

Status

Switched to SHA-3 in new release.

2.5 KZENC-F-005: Loss of Security Bits by Unnecessary Multiplication

Severity: Low

Description

In the function `base_point2` in `ed25519.rs` the scalar is multiplied by 8 at the end (to avoid being in the small cofactor subgroup). However, this is already done in the function `from_bytes`, thus actually multiplying by 64, and losing 6 bits of security instead of 3.

Recommendation

Avoid the unnecessary multiplication by 8.

Status

Fixed in new release.

2.6 KZENC-F-006: Lack of Control on Feldman VSS Parameters

Severity: Medium

Description

In `feldman_vss.rs`, a verifiable secret sharing is implemented, where t-out-of-n shares are necessary and sufficient to reconstruct the secret. However, there is no control over the consistency of t and n parameters. This is troublesome, because the above parameters are user-controlled. More in detail:

1. There is no upper bound enforced on n and t. This might allow a malicious user input to exceed memory limitations.
2. There is no check that t is not greater than n. This might lead to unexpected behavior.
3. In `share_at_indices`, there is no check that the t indices are actually different. This might lead to unexpected behavior, or to failure in recovering the secret.

Recommendation

Enforce all of the above checks, abort on error.

Status

Regarding the above points:

1. KZen considers the natural bound given by the type `uint` sufficient for n and t. If necessary, more restrictive types can be used in future releases.
2. Check added in new release.
3. `share_at_indices` removed in new release, so no further check necessary.

2.7 KZENC-F-007: Possible Timing Leak in Mpz::Modulo::mod_sub

Severity: Low

Description

In `big_gmp.rs`, the `Mpz::Modulo::mod_sub()` function is implemented as follows:

```
    fn mod_sub(a: &Self, b: &Self, modulus: &Self) -> Self {
        let a_m = a.mod_floor(modulus);
        let b_m = b.mod_floor(modulus);
        if a_m >= b_m {
            (a_m - b_m).mod_floor(modulus)
        } else {
            (a + (-b + modulus)).mod_floor(modulus)
        }
    }
```

The conditional statement introduces a possible timing leak: by measuring whether the inversion of the sign happens or not, one can infere whether b is greater than a.

Recommendation

Rewrite arithmetic flow by keeping it constant time. For example, always perform the sign inversion using an auxiliary variable, and conditionally on the >= evaluation do the subtraction with one or the other variable.

Status

KZen removed the branching altogether in the new release.

2.8 KZENC-F-008: Possible Timing Attack in ECScalar::from()

Severity: Low

Description

In ed25519.rs, the ECScalar::from() function is implemented as follows:

```
        fn from(n: &BigInt) -> Ed25519Scalar {
                let mut v = BigInt::to_vec(&n);
                let mut bytes_array_32: [u8; 32];
                if v.len() < SECRET_KEY_SIZE {
                        let mut template = vec![0; SECRET_KEY_SIZE - v.len()];
                        template.extend_from_slice(&v);
                        v = template;
                }
                bytes_array_32 = [0; SECRET_KEY_SIZE];
                let bytes = &v[..SECRET_KEY_SIZE];
                bytes_array_32.copy_from_slice(&bytes);
                bytes_array_32.reverse();
                Ed25519Scalar {
                        purpose: "from_big_int",
                fe: SK::from_bytes(&bytes_array_32),
                }
            }
```

The conditional if statement before padding introduces a possible timing leak in case the secret key has a lot of leading zeroes.

The same issue appears in sec256p_k1.rs.

Recommendation

Drop the conditional if statement as it is not necessary: in most of the cases v.len() will actually be the same as SECRET_KEY_SIZE, so extend_from_slice will simply extend the representation by an empty vector. The behavior in Rust in this case must be checked. However, for error control, and depending on the source of the input n, it would be advisable to add a conditional if statement to check whether v.len() *exceeds* SECRET_KEY_SIZE instead (i.e. checks whether n is by any chance too large), and raise an exception if that is the case.

Status

KZen acknowledged the issue and will check if it is possible to remove the conditional statement.

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 KZENC-O-001: Add SHA Testing

In `hash_sha256.rs` and `hmac_sha512.rs` the testing functionality is very elementary, while in `hash_sha512.rs` is totally absent.

Status

Some more basic tests for functionality correctness were added in new release. In general, statistical tests for hash functions are 1) hard to write 2) hard to understand.

3.2 KZENC-O-002: Non-Standard Diffie-Hellman Protocol Implementation

The key exchange protocol implemented in `dh_key_exchange.rs` is not a standard Diffie-Hellman, but a variant where the first public message is first transmitted only in committed form, and finally revealed at a last step.

```
//This is an implementation of a Diffie Hellman Key Exchange.
// Party1 private key is "x",
// Party2 private key is "y",
//protocol:
// party1 sends a commitmemt to P1 = xG a commitment to a proof of knowledge of x
// party2 sends P2 = yG and a proof of knowledge of y
// party1 verifies party2 proof decommit to P1 and  to the PoK
// party2 verifies party1 proof
// the shared secret is Q = xyG
// reference can be found in protocol 3.1 step 1 - 3(b)
//         in the paper https://eprint.iacr.org/2017/552.pdf
```

This adds a significant overhead to the key exchange, and seems to be overkill for a simple key exchange scheme.

Status

The non-standard DH is due to specific use case encountered. However, KZen added a new file with standard DH in the new release and renamed accordingly.

3.3 KZENC-O-003: Deleting Source Code File Causes Issues

Currently the discrete log proofs are used in the modified DH key exchange protocol for the commitment phase (see KZENC-O-002), so it is not clear why the file `dlog_zk_protocol.rs` can safely be deleted despite the comment in the source code:

```
// TODO: delete this file
/// THIS IS A COPY OF sigma_protocol_dlog. IT IS NOT DELETED FOR BACKWARD COMPATIBILITY.
```

In fact, deleting this file makes compiling impossible.

Status

In the new release KZen switched from using `dlog_zk_protocol` to `sigma_dlog` and removed the `dlog_zk_protocol` file from the source.

3.4 KZENC-O-004: Out-of-Date Comments

In `sigma_valid_pedersen_blind.rs` the following comment seems to be outdated, as the abstraction layer has been done already

```
// TODO: abstract for use with elliptic curves other than secp256k1
```

The same happens in `secp256_k1.rs` for `impl Secp256k1Point`:

```
//TODO: implement for other curves
```

Status

Comments removed in the new release.

3.5 KZENC-O-005: Document Use of Magic Numbers

Constants should be defined and documented more clearly. For instance, in `base_point2() -> Secp256k1Point`:

- Hashing 3 times the generator point to obtain a second base point looks odd.

```
    pub fn base_point2() -> Secp256k1Point {
        let g: Secp256k1Point = ECPoint::generator();
        let hash = HSha256::create_hash(&[&g.bytes_compressed_to_big_int()]);
        let hash = HSha256::create_hash(&[&hash]);
        let hash = HSha256::create_hash(&[&hash]);
        let mut hash_vec = BigInt::to_vec(&hash);
```

  This has been explained by KZen as a pragmatical "nothing up my sleeves" approach, as hashing three times the generator is the minimum required to hit a representation of another suitable base point for this particular curve.

- The number 2 is the parity of the curve, a constant required for the `secp256k1_eckey_pubkey_parse` function defined in the relevant C binding library to parse it correctly, but it seems to be just a magic number in the Rust code:

```
let mut template: Vec<u8> = vec![2];
        template.append(&mut hash_vec);
```

Status

Comment added in the new release.

3.6 KZENC-O-006: Outdated Ring Library

The version of the `ring` library in use is 0.13.5, but the latest version is 0.14.5 (cf. `https://crates.io/crates/ring`).

Status

The merkle library is using `ring 0.13`. Since it is only used for hash and hmac functions, KZen considers it ok to keep version 0.13.5 for now.

3.7 KZENC-O-007: Mismatched `Purpose` Attribute

In `curve_ristretto.rs`, the `ECPoint::y_coor()` function sets the purpose of `RistrettoScalar` to `base_fe`. This seems to suggest that the corresponding field element is somewhat associated to the generator, which is not necessarily the case.

Status

Purpose field changed in new release.

3.8 KZENC-O-008: Missing `x_coor` Code

In `curve_ristretto.rs`, the `ECPoint::x_coor` function is currently empty.

Status

Added a placeholder "unimplemented" in new release.

3.9 KZENC-O-009: Missing Recovery Option for User-Defined Feldman Share Indices

The `reconstruct()` function assumes that the secret shares have been generated through `share()`, i.e. there is currently no reconstruct option implemented for `share_at_indices()`.

Status

KZen removed `share_at_indices()` altogether, as it is never used.

3.10 KZENC-O-010: Representation of Feldman Share Indices Through `uint32` Values

Indices for the secret shares of the VSS are treated as uint32 values from the user's perspective, but they are actually FEs. This choice should probably be documented.

Status

Comment added in the new release.

3.11 KZENC-O-011: Typo in File Name

The file name sigma_correct_homomrphic_elgamal_enc.rs contains a typo ("homomrphic" vs. "homomorphic").

Status

Fixed in new release.

3.12 KZENC-O-012: Mismatched Purpose Attribute

In secp256_k1.rs, the ECScalar<SK>::sub function sets the purpose of Secp256k1Scalar to mul instead of, e.g., sub.

Status

Fixed in new release.

3.13 KZENC-O-013: Empty Source Code File

The source file coin_flip.rs is empty.

Status

File deleted in new release.

3.14 KZENC-O-014: Test Always Succeeding

The following test will always succeed:

```rust
#[test]
fn test_to_hex() {
    let a = Mpz::from(11);
    assert_eq!(a.to_str_radix(16), a.to_hex());
}
```

Indeed, to_hex() is defined as follows:

```rust
fn to_hex(&self) -> String {
    self.to_str_radix(super::HEX_RADIX)
}
```

Status

Test was updated to test that `to_hex` on decimal 11 is equal to hard coded "b" in new release.

3.15 KZENC-O-015: Inconsistent Struct Fields Types

In `secp256_k1.rs`, `purpose` of scalar and point structures is a `String` structure, whereas their ristretto counterparts define `purpose` as a string literal.

A comment in `secp256_k1.rs` says that "it has to be a non constant string for serialization", yet the ristretto versions support serialization in a similar fashion. We thus failed to understand the need for different string types here.

Status

Changed `secp256k1` curve's purpose to string literal in new release.

3.16 KZENC-O-016: Seemingly Unnecessary Check For Zero Element

In `sigma_correct_homomrphic_elgamal_enc.rs`, the function `fn HomoELGamalProof::prove()` branches to explicitly check that the value of the witness is not zero. We recommend documenting this choice.

Status

According to KZen, the multiplication routine they are using does not allow to multiply by zero, this is why the need for the check.

3.17 KZENC-O-017: Unnecessary Randomness Generation

In `secp256_k1.rs` the function `fn Secp256k1Point::random_point()` seems to generate additional useless randomness.

Status

The unnecessary code was left by mistake from a previous implementation of the function and will be removed in future releases.

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