# Multi-party ECDSA Security Audit

Final Report, 2019-10-22

FOR PUBLIC RELEASE

# Contents

# 1 Summary

KZen hired Kudelski Security to perform a security assessment of their multi-party ECDSA library written in Rust, and provided us access to their source code and documentation.

The repository concerned is:
https://github.com/KZen-networks/multi-party-ecdsa
we specifically audited commit `c2964f9` in the branch "audit1".

This document reports the security issues identified and our mitigation recommendations, as well as some observations regarding the code base and general code safety. A "Status" section reports the feedback from KZen's developers, and includes a reference to the patches related to the reported issues. All changes have been reviewed by our team according to our usual audit methodology.

We report:

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

The audit was performed jointly by Dr. Tommaso Gagliardoni, Cryptography Expert, and Yolan Romailler, Senior Cryptography Engineer.

# 2 Methodology

In this code audit, we performed three main tasks:

1. informal security analysis of the original protocol;
2. actual code review with code safety issues in mind;
3. compliance of the code with the papers.

This was done in a static way and no dynamic analysis has been performed on the codebase. We discuss more in detail our methodology in the following sections.

## 2.1 Protocol Security

We analyzed the protocol in view of the claimed goals and use cases, and we inspected the original protocol description, looking for possible attack scenarios. We focused on the following aspects:

- possible threat scenarios;
- necessary trust assumptions between involved parties;
- edge cases and resistance to protocol misuse.

## 2.2 Code Safety

We analyzed the provided code, looking for issues related to safety, including:

- general code safety and susceptibility to known vulnerabilities;
- bad coding practices or unsafe behavior;
- leakage of secrets or other sensitive data through memory mismanagement;

- susceptibility to misuse and system errors;
- error management and logging;
- safety against malformed or malicious input from other participants.

## 2.3 Cryptography

We analyzed the cryptographic primitives and subprotocols used, with particular emphasis on randomness and hash generation, signatures, key management, and encryption. We checked in particular:

- matching of the proper cryptographic primitives to the desired cryptographic functionality needed;
- security level of cryptographic primitives and of their respective parameters (key lengths, etc.);
- safety of the randomness generation in the general case and in case of failure;
- assessment of proper security definitions and compliance to the use cases;
- known vulnerabilities in the primitives used.

## 2.4 Protocol Specification Matching

We analyzed the original papers, and checked that the code matches the specification. We checked for things such as:

- proper implementation of the different protocol phases;
- proper error handling;
- adherence to the protocol logical description.

# 3 Findings

This section reports security issues found during the audit.

The "Status" section includes feedback from the developers received after delivering our draft report.

## KZM-F-01: Safe primes are not used

Severity: Medium

### Description

It appears that in the multi-party ECDSA paper [GG18], safe primes are required because the ZK proofs used are holding under the Strong RSA assumption.

The current codebase is not generating safe primes, however it should be noted that the ZK proofs used are not necessarily the same as the ones used in the paper. As such it is not clear whether other components of the papers are also requiring the Strong RSA assumption, in which case safe primes should still be used, or not.

### Recommendation

Using safe primes comes at a slight performance cost during key generation, but can only increase the security of the protocol, as such we recommend implementing a safe prime key generation algorithm instead of the current key generation.

### Status

According to KZen's feedback and research on the issue, it turns out that finding safe primes takes a considerable amount of time more than finding regular primes (see e.g. `https://crypto.stackexchange.com/questions/66076/how-to-efficiently-generate-a-random-safe-prime-of-given-length`). This is because of the low density of safe primes (see e.g. `https://www.researchgate.net/publication/3822432_Safe` _primes_density_and_cryptographic_applications). This is so slow that it makes GG18 key generation take minutes to generate two 1024-bit safe primes per party.

Because the effect of using safe vs. normal primes is unclear in this case, KZen opted to allow regular primes as default, and added a comment that switching to safe primes in production is recommended. To this scope, in the new version the `rust-pailler` master branch was changed to support finding safe primes, and an option was added in `multi-party-ecdsa` to create a key with safe primes.

## KZM-F-02: Missing check in KeygenSecondMsg

Severity: Medium

### Description

In `two_party_ecdsa/party_two.rs` the function `verify_commitments_and_dlog_proof()` implements step 5. of [Lin17], Protocol 3.1, and correctly performs checks (a) (commitment of public key and range proof) and (b) (encryption of correct discrete log from language $L_{PDL}$), but does not verify check (c) (length of Paillier key):

```
        let mut flag = true;
        if party_one_pk_commitment
                == &HashCommitment::create_commitment_with_user_defined_randomness(
                        &party_one_public_share.bytes_compressed_to_big_int(),
                        &party_one_pk_commitment_blind_factor,
        ) {
                flag = flag
        } else {
                flag = false
        }
        if party_one_zk_pok_commitment
                == &HashCommitment::create_commitment_with_user_defined_randomness(
                        &party_one_d_log_proof
                                .pk_t_rand_commitment
                                .bytes_compressed_to_big_int(),
                        &party_one_zk_pok_blind_factor,
                ) {
                        flag = flag
                } else {
                        flag = false
                };
        assert!(flag);
        DLogProof::verify(&party_one_d_log_proof)?;
        Ok(KeyGenSecondMsg {})
}
```

Notice the following:

1. we have found a typo in [Lin17] Protocol 3.1, looking at the explanation in the introduction of that paper it seems that it should be $\max(3\log|q|+1, n)$ not $\min$. We have reported this to the author of the paper, who acknowledged the typo and corrected it on ePrint.
2. the reason for this check is that $P_2$ wants non-repudiation from $P_1$: imagine $P_1$ and $P_2$ generate a signature, e.g. a Bitcoin transaction; after that $P_1$ might say "hey, I never authorized that signature! I think $P_2$ bruteforced my $c_{key}$ and recovered my secret share $x_1$ so he could output a signature without my approval!"

### Recommendation

It should be checked that the modulus $N$ of the Paillier key generated by the server is of length 2048 bits.

### Status

This has been corrected in the new release. However, the check was added in `verify_ni_proof_correct_key`, which is a more natural place for this scope.

## KZM-F-03: Message hashing left to the signer in both scheme

Severity: Medium

### Description

We want to stress that the library is assuming it receives a message digest as input, not a message, and as such is not performing the hashing, nor the mapping onto $\mathbb{Z}_q$.

In `two_party_ecdsa/party_two.rs`, it appears that `PartialSig()` takes as input a message which should be a `BigInt` (line 424) without hashing it beforehand as per [Lin17], Protocol 3.2.

The same is done also for [GG18], where the message is said in comment to be assumed signed by the signer. However, notice that it is then reduced modulo $2^{256}$, before being processed by the `ECScalar::from()` function, which reduces it modulo the `curve_order` $q$ in the secp256k case (notice that for other curves in the library, the `ECScalar::from()` function is not necessarily behaving in the same way.)

### Recommendation

Since this is an explicit choice, we recommend delegating entirely the task of hashing and mapping onto $\mathbb{Z}_q$ to the signer, and to reject message digests$\mathbb{Z}_q$ by erroring out.

As such, there should be no need to reduce the message digest modulo $2^{256}$. (Especially so, since this is done again in the `ECScalar::from()` function.)

### Status

KZen acknowledges this observation and removed the unnecessary modulo $2^{256}$ reduction. However, it was decided to apply the full change in hashing of the input in a future release.

The reason is that all KZen libraries are working under the design principle of letting the user do the hashing, the rational being that if a user would want to sign a message which is not a digest, that would be fine as long as the message can be mapped to an element in the field (which is the job of the library to do). KZen accepts the suggestion in the report, but since in this case they not see an immediate security risk and *all* KZen libraries would be affected, they decided to push the change in a coordinated way in the future, as otherwise it would be prone to errors and issues of interoperability.

## KZM-F-04: Error in computing shared secret key from ECDH

Severity: Medium

### Description

Currently the AES encryption keys used to establish a secure channel between all pairs of peers in the multi-party key generation client are setup using a wrong shared key algorithm. In `gg18_keygen_client.rs`, we can see that the shared key is created by adding the public points together:

```
enc_keys.push(
        (party_keys.y_i + decom_j.y_i)
                .x_coor()
                .unwrap(),
);
```

But this is not the standard elliptic curve Diffie-Hellman key agreement, since it means that the shared keys are generated by adding the public keys of both parties, which can be recomputed by anyone, as it only involves public information.

### Recommendation

Establish a secure channel, typically by using the regular Diffie-Hellman key agreement, in which the shared key is establish by scalar multiplication of the other party's public point with the local party's secret scalar:

```
enc_keys.push(
        (party_keys.u_i * decom_j.y_i)
                .x_coor()
                .unwrap(),
);
```

### Status

Corrected as per recommendation in the new release.

## KZM-F-05: Signing phase 4 is not explicitly doing ZKP verification

Severity: Low

### Description

In the multi-party ECDSA code, the function `verify_dlog_proofs()` is used to verify the proofs of knowledge of $x_i$ in the key generation, but in the signing phase 4, it should also be verified that the decommitted values proves the knowledge of $\gamma_i$.

The proofs in `b_proof_vec` are currently not directly verified, neither in `fn phase4()` in `party_i.rs`, neither in the `gg18_sign_client.rs` code.

```
let test_b_vec_and_com = (0..b_proof_vec.len())
.map(|i| {
        b_proof_vec[i].pk.get_element() ==
         ↪  phase1_decommit_vec[i].g_gamma_i.get_element()
                && HashCommitment::create_commitment_with_user_defined_randomness(
                        &phase1_decommit_vec[i]
                                .g_gamma_i
                                .bytes_compressed_to_big_int(),
                        &phase1_decommit_vec[i].blind_factor,
                ) == bc1_vec[i].com
})
.all(|x| x == true);
```

### Recommendation

Implement the additional verification step.

### Status

This is actually mitigated by the fact that the `b_proof_vec` is populated using the results from the MtAwc, which is handling the proof verification in the `verify_proofs_get_alpha()` function. However, it should be documented, as this check is important and shall the MtAwc be replaced by the simpler MtA version, that check would not be performed anymore. Therefore, a comment has been added in the new release.

## KZM-F-06: Secret temporary value not zeroized

Severity: Low

### Description

In `two_party_ecdsa/party_one.rs` one can find the following code:

```
let mut secret_share: FE = ECScalar::new_random();
let public_share = base * secret_share;
let h: GE = GE::base_point2();
let c = h * secret_share;
let mut x = secret_share;
let w = ECDDHWitness { x };
let delta = ECDDHStatement {
        g1: base,
        h1: public_share,
        g2: h,
        h2: c,
};
let d_log_proof = ECDDHProof::prove(&w, &delta);
let ec_key_pair = EphEcKeyPair {
        public_share,
        secret_share,
};
secret_share.zeroize();
x.zeroize();
```

However here, the value $w$ should also be zeroized, as it contains the same sensitive data as the value $x$. The same happens in `party_two.rs`, line 383.

### Recommendation

Zeroize these values as well.

### Status

Corrected as per recommendation in the new release.

## KZM-F-07: Deviation: range proofs are not performed in MtA

Severity: Informational

### Description

In the multi-party ECDSA case, in `mta.rs`, in `impl MessageA` the range proofs are not included for Alice, and in `impl MessageB`, instead it is only computing a proof of knowledge of the discrete logarithm of $g^b$, instead of proving that $b$ is in the appropriate range as per page 8 of [GG18], where it says Bob should be proving in ZK that $b < K$ (for $K q^3$).

The rational why we might want to have the range proof are in [GG18] on page 9, basically it would allow a malicious party to make the threshold signature fail verification and let them go unblamed.

### Status

KZen has been informed that this is an explicit deviation from the paper [GG18] that is notably motivated by the arguments discussed in Section 5 (page 19) of the same paper. The lack of "blame phase" is a fundamental problem with these protocols that KZen believes to be solvable. For now, an issue was opened specifically on this topic: https://github.com/KZen-networks/multi-party-ecdsa/issues/80.

KZen will allow for optional range proofs in future releases. An issue was opened on this topic: https://github.com/KZen-networks/multi-party-ecdsa/issues/79.

## KZM-F-08: Deviation: extra values are included in the commitment in multi-party ECDSA

Severity: Informational

### Description

In `phase5a_broadcast_5b_zkproof`, the value `let B_i = g * l_i_rho_i;` is computed and committed, which is not included in the original paper [GG18].Notice this value is then part of the proof, and is correctly checked. This is an undocumented deviation from the paper and is not a security issue.

### Status

KZen does not consider harmful to include this extra information.

## KZM-F-09: Deviation: elements hashed in reverse order

Severity: Informational

### Description

We noticed a very slight, undocumented deviation from the original Lindell paper [Lin17]: in the two party ECDSA, during the PDL routine for the key generation (see protocol 6.1 in [Lin17]) at one intermediate step, it is required to compute the commitment of secret values $a$ and $b$.

For example, in `party_one.rs`, this is done as:

```
let ab_concat = a.clone() + b.clone().shl(a.bit_length());
let c_tag_tag_test =
HashCommitment::create_commitment_with_user_defined_randomness(&ab_concat, &blindness);
```

This is equivalent to computing the commitment of `b||a`. However, in [Lin17] it is the value `a||b` that gets committed instead.

This difference is arguably irrelevant for the security of the protocol, however might cause interoperability issues if not documented.

### Status

A comment was added in the new release.

## KZM-F-10: Deviation: MtAwc is missing a ZK proof

Severity: Informational

### Description

In the multi-party ECDSA case, in `mta.rs`, it appears the ZK proof that $c_B = b \times_E E_A(\beta')$ is replaced by a different proof that recomputes the ciphertext on Alice's side and compare it with the actual one.

```
let ba_btag = &self.b_proof.pk * a + &self.beta_tag_proof.pk;
match DLogProof::verify(&self.b_proof).is_ok()
        && DLogProof::verify(&self.beta_tag_proof).is_ok()
        && ba_btag.get_element() == g_alpha.get_element()
{
        true => Ok(alpha),
        false => Err(InvalidKey),
}
```

It appears the outcome is the same as performing the ZK proof done in [GG18].

### Status

A comment was added in the new release. A more formal specification document will be provided in the future.

## KZM-F-11: Deviation: missing proof in the two-party-rotation protocol

Severity: Informational

### Description

In `two_party_ecdsa/party_one.rs`, the function `refresh_private_key()` for the server side correctly generates a range proof for the new key and a proof of valid Paillier key formation, but it appears one proof is missing: according to step 4 of the two-party-rotation protocol in [KZ19], Section 4.4, there should also be a $L_{PDL}$ proof, which is currently missing in the library.

### Recommendation

Add the proof, or document why this is not done in the library.

### Status

KZen's design goal is to not allow clients to have access or to manipulate private data such as the secret share. This is why if there is a need from outside of the library to do computations over the secret share it should be implemented in the multi-party ECDSA library, but only the sensitive parts are currently implemented. While the $L_{PDL}$ proof is also part of the key rotation, it is something a client could do.

KZen already implemented the rotation scheme in another library: the relevant part of the test that includes the $L_{PDL}$ proof can be found in the kms-secp256k1 repository in `src/ecdsa/two_party/test.rs`.

# 4 Observations

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

## KZM-O-01: Implicit secret inputs zeroized during two-party KeyGen

In `two_party_ecdsa/party_one.rs` (the same happens in `party_two.rs`) there are two different functions which can be called to produce the first message of the key generation protocol: one is `pub fn create_commitments()` and the other is `pub fn create_commitments_with_fixed_secret_share()`.

The former generates a secret share, makes sure that it falls in the correct range, and then correctly zeroizes the temporary value. The latter takes as input the secret share, which is assumed to be generated and falling *a priori* in the right range. Given this, it could be a good thing to `zeroize()` its input as done on line 212.

**Status**

Corrected in the new release.

## KZM-O-02: Hard-coded delays

Since the protocols are assuming a reliable broadcast channel, whereas the internet is not a synchronous network, delays are used in the communication in order to try and ensure delivery of the messages before proceeding further. However the delay value is currently hard-coded as a magic number (25ms) in the source.

We recommend having a way to set larger delays to avoid problems in practice, that could arise depending on the use-cases in which the protocols are used.

### Status

KZen argues that this intentional delay is mainly to run the demo. Other users of the library changed the delay to fit their use case. In practice KZen has another project for the communication layer: https://github.com/KZen-networks/white-city.

## KZM-O-03: Party labeling reversed in the two-party ECDSA

In the signing protocol, during the ephemeral key generation step, the roles of $P_1$ and $P_2$ as described in [Lin17], Protocol 3.2, are implemented in `party_two.rs` and `party_one.rs` respectively, i.e., in the reverse order, and then the roles reverse again during the signature process, just after the ephemeral key generation step.

Although this is unlikely to introduce vulnerabilities, we note that this represents a deviation from the formal protocol, and therefore we cannot guarantee the safety of this approach. Moreover, it hurts readability and might be documented more prominently.

### Status

KZen noticed that if the ephemeral DH key exchange is done with first message from party 2 (and not party 1), then instead of 5 messages the signing protocol can be done in 4 messages: the second message of party 2 is a combination of the second message of the DH key generation and the first part of the signature computation: https://github.com/KZen-networks/kms-secp256k1/blob/master/src/ecdsa/two_party/party2.rs#L245. KZen did not see necessary to document this modification.

## KZM-O-04: The range proofs are using a static salt

In `range_proof_ni.rs`, a static salt is used, but not explicitly documented:

```
const SALT_STRING: &[u8] = &[75, 90, 101, 110];
```

Notice this is simply the bytes of the string “KZen”. The usage of a static salt is not a security issue *per se* in the given setting, but in a library meant to be reused like this we recommend having (and documenting) a method to set the salt value.

### Status

Tracked at: https://github.com/KZen-networks/zk-paillier/issues/12.

## KZM-O-05: Clippy warnings

We notice some Clippy warnings that could slightly impact performance. However all of them are really low impact such as the ones about values that are passed by reference, whereas they could be passed by value, which is more efficient.

### Status

KZen acknowledges this observation. The issue is tracked at: `https://github.com/KZen-networks/multi-party-ecdsa/pull/71`.

# Bibliography

[GG18] Rosario Gennaro and Steven Goldfeder. “Fast Multiparty Threshold ECDSA with Fast Trustless Setup”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. Toronto, Canada: ACM, 2018, pp. 1179–1194. ISBN: 978-1-4503-5693-0.

[KZ19] Team KZen. *Bitcoin Wallet Powered by Two-Party ECDSA – Extended Abstract*. Tech. rep. KZen Research, 2019. URL: https://github.com/KZen-networks/gotham-city/blob/master/white-paper/white-paper.pdf.

[Lin17] Yehuda Lindell. “Fast Secure Two-Party ECDSA Signing”. In: *Advances in Cryptology – CRYPTO 2017*. Ed. by Jonathan Katz and Hovav Shacham. Cham: Springer International Publishing, 2017, pp. 613–644.

# 5 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 or https://kudelski-blockchain.com/.

Kudelski Security
Route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland