Secure Code Review of `BLS` and `Gennaro-DKG` Libraries

Technical Report

Lit Protocol

February 15, 2024
Version: 1.1

Corporate Headquarters
Kudelski Security – Nagravision Sàrl
Route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland

Public

DOCUMENT PROPERTIES

VERSION	1.1
FILE NAME	2024-01-03_LitProtocol_Crypto_Libraries_v1.1.pdf
PUBLICATION DATE	February 15, 2024
CONFIDENTIALITY LEVEL	Public
DOCUMENT RECIPIENT	Lit Protocol
DOCUMENT STATUS	Final
CLIENT COMPANY NAME	Lit Protocol

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. EXECUTIVE SUMMARY

Lit Protocol engaged Kudelski Security to perform a secure code assessment of two cryptographic libraries implemented in Rust: `blsful` and `gennaro-dkg`. The assessment was conducted remotely by the Kudelski Security Team. Testing took place between 05.10.2023 and 24.11.2023, and focused on the following objectives:

- Provide the customer with an assessment of their overall security posture and any risks that were discovered with both cryptographic libraries.
- To provide a professional opinion on the maturity, adequacy, and efficiency of the security measures that are in place.
- To identify potential issues and include improvement recommendations based on the result of our tests.

1.1. Key Findings

The following are the major themes and issues identified during the testing period. These, along with other items, within the findings section, should be prioritized for remediation to reduce the risk they pose.

- Lack of input control.
- Secret keys and commitments can be created from unverified array of bytes.
- Risk of division by zero.

Fuzzing provided the following results:

- Encryption of empty message makes `blsful` crash. (fuzzing result)
- Uncontrolled inputs on public functions `from_be_bytes` and `from_le_bytes` result in crashes. (fuzzing result)

It is important to highlight that both libraries `blsful` and `gennaro-dkg` are only perform mathematical computations of the different schemes implemented. The communication between parties or the secure storage of the keys was not provided.

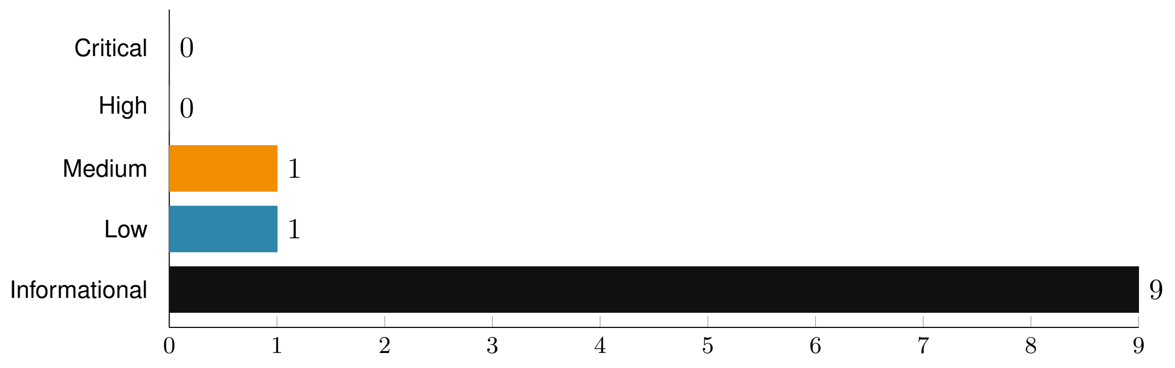

Figure 1. Findings Ranked by Severity.

2. PROJECT SUMMARY

This report summarizes the engagement, tests performed, and findings. It also contains detailed descriptions of the discovered vulnerabilities, steps the Kudelski Security team took to identify and validate each issue, as well as any applicable recommendations for remediation.

2.1. Context

`blsful` implements three instances of the digital signature BLS:

- the traditional digital signature,
- the aggregated signature
- and the multisignature.

This library also proposes implementations of:

- the public key encryption scheme ElGamal,
- a timelock encryption from threshold BLS scheme,
- a signcryption scheme and
- a proof of knowledge scheme.

The second library implements the Distributed Key Generation algorithm named Gennaro.

2.2. Scope

The scope consisted of all Rust files in the following folders:

- `blsful/scr` [1]
  - Commit hash : `7555c49aa1844446b58c90876cb387868d8a7d86`
  - BLS Signature Scheme
- `gennaro-dkg/src` [2]
  - Commit hash : `55b928d049e4d9fc8750790c0d53c4654bf4fd09`
  - gennaro-dkg

A re-review was performed on 03.01.2024 on the following Commit hashes:

- `blsful/scr` [1]
  - Commit hash : `d2bd89c3183915abc36c2910e09cb770831f4a7d`
- `gennaro-dkg/src` [2]
  - Commit hash : `1c15242b3f833c0fe42f36c3f623f2c6d8bc64b1`

Follow-up

After the initial report, Lit Protocol addressed the vulnerabilities and weaknesses in the following codebase revision:

- `blsful/scr` [1]
  - Commit hash : `d2bd89c3183915abc36c2910e09cb770831f4a7d`
- `gennaro-dkg/src` [2]
  - Commit hash : `1c15242b3f833c0fe42f36c3f623f2c6d8bc64b1`

2.3. Remarks

During the code review, the following positive observations were noted regarding the scope of the engagement:

- Code was well written by developers that had good notions of secure implementation such as randomness implementation.
- Tests were also provided as part of the project, which is convenient for better understanding how the library works and useful for elaborating scenarios and validating findings.
- Finally, we had regular and very enriching technical exchanges on various topics.

2.4. Fuzzing

During the time of the audit, Kudelski Security performed fuzzing on some parts of the code that plays a crucial role in the libraries to uncover bugs, vulnerabilities, or weaknesses that might not be found through traditional testing methods. The fuzzing has been executed on the following schemes:

- Signcryption
- BLS digital signature
- BLS multisignature
- BLS aggregated signature
- ElGamal encryption scheme
- Timelock encryption scheme

Fuzzing executed by Kudelski Security provided the following results:

- Encryption of empty message makes `blsful` crash.
- Uncontrolled inputs on public functions `from_be_bytes` and `from_le_bytes` result in crashes.

Fuzzing is not by default part of a Secure Code Review, but it can be performed when time and quality of documentation allow for it.

2.5. Test Coverage

Both libraries were provided tests and Kudelski Security checked the test coverage of both libraries.

- `blsful` reaches a test coverage of 52.07%.
- `gennaro-dkg` reaches a test coverage of 66.37%.

2.6. Additional Note

It is important to notice that, although we did our best in our analysis, no code audit assessment is per se guarantee of absence of vulnerabilities. Our effort was constrained by resource and time limits, along with the scope of the agreement.

In assessing the severity of some of the findings we identified, we kept in mind both the ease of exploitability and the potential damage caused by an exploit. In the specific case of this cryptographic libraries, we focused on logic, randomness, secret information leaks.

While assessing the severity of the findings, we considered the impact, ease of exploitability, and the probability of attack. This is a solid baseline for severity determination. Information about the severity ratings can be found in Section 5 of this document.

3. FINDINGS

The Findings section provides detailed information on each of the findings, including methods of discovery, explanation of severity determination, recommendations, and applicable references.

The following table provides an overview of the findings.

#	SEVERITY	TITLE	STATUS
KS-LC-1	Medium	Missing Input Sanitization	Resolved
KS-LC-2	Low	Division by Zero	Resolved
KS-LC-3	Informational	Use of Libraries that were not Reviewed	Acknowledged
KS-LC-4	Informational	Unclear Calculation of Blinder Generator	Acknowledged
KS-LC-5	Informational	Warnings in Dependencies	Acknowledged
KS-LC-6	Informational	Incorrect Error Handling	Resolved
KS-LC-7	Informational	Confusing Names	Resolved
KS-LC-8	Informational	No Secure Policy	Resolved
KS-LC-9	Informational	Consider Throwing Error Instead of Skip Validation	Acknowledged
KS-LC-10	Informational	Duplicate Code	Acknowledged
KS-LC-11	Informational	Examples in `README.md` are Outdated	Acknowledged

3.1. KS-LC–1 Missing Input Sanitization

Overall Severity:	MEDIUM
Status:	Resolved

Impact	Likelihood
MEDIUM	MEDIUM

Description

In the `blsful` cryptographic library, the public functions `from_le_bytes` and `from_be_bytes` allow users to convert an array of bytes into a secret key or commitment for the proving procedure. However, no input sanitization is performed on the input array before conversion. Thus, the array of bytes can be zero or not a valid encoding of a secret key or commitment at all. Zero keys are not allowed for BLS or El-Gamal.

Additionally, the Gennaro DKG Documentation's main page presents contains asserts to check if the sampled shares are not zero. This seems to further suggests that such values should be discarded when sampled.

Impact

For secret keys generation, secret keys or shares generated as 0 are insecure, because they can easily be tested for and thus identified and used by a malicious entity. Additionally, other keys with insufficient entropy can be created.

For the commitment creation, the security impact of this finding is decreased by the fact that, later in the proving procedure, it verifies that the commitment is not equal to 0. However, the only requirement is that the size of the array `bytes` is 32. This is the reason why fuzzing were used to create commitment using random array of bytes, and it demonstrated that lack of verification for the input makes the proving procedure crash. In the worst case scenario, this could result in a Denial of Service.

Evidence

```rust
/// Convert a big-endian representation of the secret key.
pub fn from_be_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
    Self> {
  scalar_from_be_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
}

/// Convert a little-endian representation of the secret key.
pub fn from_le_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
    Self> {
  scalar_from_le_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
}
```

Conversion with no input sanitization in `blsful/src/secret_key.rs`.

```rust
/// Convert a big-endian representation of the secret key.
pub fn from_be_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
    Self> {
  scalar_from_be_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
}

/// Convert a little-endian representation of the secret key.
pub fn from_le_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
    Self> {
  scalar_from_le_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
}
```

Conversion with no input sanitization in `blsful/src/proof_commitment.rs`.

```rust
/// Convert a big-endian representation of the secret key.
pub fn from_be_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
    Self> {
  scalar_from_be_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
}

/// Convert a little-endian representation of the secret key.
pub fn from_le_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
    Self> {
  scalar_from_le_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
}
```

Conversion with no input sanitization in `blsful/src/proof_commitment.rs`.

Affected Resources

- `blsful/src/secret_key.rs` lines 67-75
- `blsful/src/proof_commitment.rs` lines 165-172 and 217-225

Recommendation

Even if the primary role of these functions is to perform type conversion, it is still recommended to discard unsuitable values here. This can prevent further security issues occurring in case of an unintended use of the library functions. While this occurs with negligible probability, rejecting zero as the secret key/ the identity as the public key is recommended, which is done in the code. Verify that the array `bytes` is not 0 or representing a scalar too large. An additional recommendation could be to check the minimal and maximal length of the input array for both functions. This would ensure that the key is generated from a long enough array of bytes.

References

- CWE-20: Improper Input Validation

3.2. KS-LC–2 Division by Zero

Overall Severity:	LOW
Status:	Resolved

Impact	Likelihood
LOW	LOW

Description

One of the steps in the [2] protocol requires the participants to perform a Lagrange interpolation. In order to reconstruct a secret by Lagrange interpolation, all the shares of the parties must be distinct from each other. This check is not performed.

Impact

When performing division by $x_i - x_j$, this will effectively be a division by zero and the application will crash.

Evidence

```
fn lagrange_interpolation(
share: G::Scalar,
shares_ids: &[G::Scalar],
index: usize,
) -> G::Scalar {
  let mut basis = G::Scalar::ONE;
  for (j, x_j) in shares_ids.iter().enumerate() {
    if j == index {
      continue;
    }
    let denominator = *x_j - shares_ids[index];
    basis *= *x_j * denominator.invert().unwrap();
  }
  basis * share
}
```

gennaro-dkg/src/participant.rs. The denominator is not checked that it is different from the zero field element before inversion.

Affected Resources

- gennaro-dkg/src/participant.rs lines 211-225
- gennaro-dkg/src/participant.rs line 90

Recommendation

Check that the participants shares are distinct or that the denominator is not zero, and throw a suitable error message.

References

- CWE-369: Divide By Zero

3.3. KS-LC–3 Use of Libraries that were not Reviewed

Overall Severity:	**INFORMATIONAL**
Status:	**Acknowledged**

Description

The source code uses some dependencies implementing sensitive cryptographical operations that have not been reviewed or audited. In particular, the `gennaro-dkg` and `blsful` implementations rely on the Rust library `vsss-rs`.

Impact

This library is used for important computations. Therefore, if vulnerabilities are discovered in `vsss-rs`, these could impact security.

Evidence

The following libraries come with the the following disclaimers in their documentation.

`vsss-rs`

The documentation of `vsss-rs` (see References) mentions in bold characters: “This implementation has not been reviewed or audited. Use at your own risk.”

`ChaCha20`

The documentation for `rand:ChaChaRng` (see References) specifies the following:

> A random number generator that uses the ChaCha20 algorithm [1].
>
> The ChaCha algorithm is widely accepted as suitable for cryptographic purposes, but this implementation has not been verified as such. Prefer a generator like OsRng that defers to the operating system for cases that need high security.
>
> [1]: D. J. Bernstein, ChaCha, a variant of Salsa20

Affected Resources

- `blsful`
- `gennaro-dkg`

Recommendation

When possible, replace such modules with audited and reviewed dependencies. However, such alternatives might not exist.

References

- `rand::ChaChaRng` documentation
- `vsss-rs`

3.4. KS-LC–4 Unclear Calculation of Blinder Generator

Overall Severity:	**INFORMATIONAL**
Status:	**Acknowledged**

Description

The generator point `blinder` is created in function `Parameters::new` using the bytes of `message_generator` as seed. This deterministic approach will effectively sample the bytes `0xBA2C15C19DEDDC61E5BA61CC6A3AC2207D6043063B96C178DC93B0D1DDCD1A1A` to be interpreted as a point. From the protocol description, all parties must have the same generator for the blinding point, however it is not clear if this must be sampled deterministically in this manner. Moreover, the comment above the function describes a ”*random blinder_generator*”, which makes it unclear if it should be sampled randomly every time or not.

Impact

While there is no direct evidence of security issues sampling the blinder generator this way, the implications of using the same generator point for different sessions are unknown.

Evidence

```rust
impl<G: Group + GroupEncoding + Default> Parameters<G> {
    /// Create regular parameters with the message_generator as the default generator
    /// and a random blinder_generator
    pub fn new(threshold: NonZeroUsize, limit: NonZeroUsize) -> Self {
        let message_generator = G::generator();
        let mut seed = [0u8; 32];
        seed.copy_from_slice(&message_generator.to_bytes().as_ref()[0..32]);
        let rng = rand_chacha::ChaChaRng::from_seed(seed);
        Self {
            threshold: threshold.get(),
            limit: limit.get(),
            message_generator: G::generator(),
            blinder_generator: G::random(rng),
        }
    }
```

Generation of `blinder_generator` in `Parameters::new`.

Affected Resources

- `gennaro-dkg/src/parameters.rs` lines 30-41

Recommendation

Seed `rand_chacha::ChaChaRng` with true entropy provided by the hardware, or any other suitably cryptographic source. If, instead, `blinder_generator` is meant to be a hard-coded constant, then this might be a deviation from the protocol.

References

- CWE-337: Predictable Seed in Pseudo-Random Number Generator (PRNG)
- [2]

3.5. KS-LC–5 Warnings in Dependencies

Overall Severity:	**INFORMATIONAL**
Status:	**Acknowledged**

Description

cargo_audit detected the following 2 allowed warnings in the dependencies.

Impact

Generally, vulnerabilities in dependencies could be exploited to compromise the security of the system. In this case, however, they seem to have little to no impact on security.

Evidence

```
Crate:     serde_cbor
Version:   0.11.2
Warning:   unmaintained
Title:     serde_cbor is unmaintained
Date:      2021-08-15
ID:        RUSTSEC-2021-0127
URL:       https://rustsec.org/advisories/RUSTSEC-2021-0127
```

```
Crate:     xsalsa20poly1305
Version:   0.7.1
Warning:   unmaintained
Title:     crate has been renamed to `crypto_secretbox`
Date:      2023-05-16
ID:        RUSTSEC-2023-0037
URL:       https://rustsec.org/advisories/RUSTSEC-2023-0037
```

Affected Resources

- gennaro-dkg

Recommendation

Follow the recommendations of cargo audit. If suitable, replace serde_cbor with either ciborium or minicbor as suggested by the author here. Additionally, consider renaming xsalsa20poly1305 to crypto_secretbox.

References

N/A

3.6. KS-LC–6 Incorrect Error Handling

Overall Severity:	**INFORMATIONAL**
Status:	**Resolved**

Description

Different functions in the code panic either by calling it explicitly `panic!` or by performing an operation that can panic, such as accessing an array out of bounds. Explicit error handling (such as using `Result`) should be preferred instead of `panic!`, as outlined in the Programming Rules to Develop Secure Applications with Rust.

For example, the function `try_from`, which is called when performing the signing procedure of BLS multisignature, takes as input an array of bytes representing the signature of different parties. This function performs a loop on the inputs starting with the index 1 (e.g. `for s in &sigs[1..]`) without checking that the length of the array `sigs`.

Impact

Instructions that can cause the code to panic at runtime may cause the application to crash unexpectedly. In the case of the index out of bound example, users could execute the signing procedure for the BLS multisignature for only one party. This will make the execution unexpectedly panic and in the worst case scenarios create a Denial of Service.

Evidence

Possible Array out of Bounds

```
fn try_from(sigs: &[Signature<C>]) -> Result<Self, Self::Error> {
  let mut g = <C as Pairing>::Signature::identity();
  for s in &sigs[1..] {
    if !s.same_scheme(&sigs[0]) {
      return Err(BlsError::InvalidSignatureScheme);
    }
    let ss = match s {
      Signature::Basic(sig) => sig,
      Signature::MessageAugmentation(_) => {
        return Err(BlsError::InvalidSignatureScheme);
      }
      Signature::ProofOfPossession(sig) => sig,
    };
    g += ss;
  }
  match sigs[0] {
    Signature::Basic(s) => Ok(Self::Basic(g + s)),
    Signature::MessageAugmentation(s) => Ok(Self::
 MessageAugmentation(g + s)),
    Signature::ProofOfPossession(s) => Ok(Self::ProofOfPossession
 (g + s)),
  }
```

```
    }
```

Index out of Bounds `blsful/src/multi_signature.rs`

Explicit `panic!`

```
                _ => panic!("Signature::conditional_select:␣mismatched␣
    variants"),
```

In `blsful/src/aggregate_signature.rs` the function `conditional_select` panics.

Affected Resources

- `blsful/src/aggregate_signature.rs` line 79
- `blsful/src/multi_signature.rs` line 79
- `blsful/src/proof_commitment.rs` line 85
- `blsful/src/proof_of_knowledge.rs` line 112
- `blsful/src/signature_share.rs` line 66
- `blsful/src/signature.rs` line 79
- `blsful/src/multi_signature.rs` line 85

Recommendation

Remove instructions that can cause the code to panic like `unwrap` or `panic!`. Always test values of type `Result` or `Option` before using them or unwrapping them. Refer to the Rust documentation for the idiomatic way of handling errors. Validation of inputs plays also a pivotal role, for example the `try_from` function in file `blsful/src/multi_signature.rs` needs to perform a verification that at least two signatures are given as inputs to the function. In other words, ensure that the array of bytes `sigs` has length at least equal to two. By checking this, the risk of index out of bounds is avoided. Additionally, a multisignature should involve at least two parties (otherwise, a traditional digital signature would have sufficed.

References

- Option & unwrap, Rust by Example
- To `panic!` or Not to `panic!`, The Rust Programming language
- CWE-703: Improper Check or Handling of Exceptional Conditions
- ANSSI Guidelines - Programming Rules to Develop Secure Applications with Rust, Section 4.4

3.7. KS-LC–7 Confusing Names

Overall Severity:	**INFORMATIONAL**
Status:	**Resolved**

Description

Various files in `gennaro-main-dkg` use single-letter variable names such as “u” or “p”.

Impact

Using descriptive name for variables and function can help with understanding the source code and debugging. This reduces the risk of accidentally introducing vulnerabilities during development.

Evidence

```rust
pub fn serialize<S: Serializer>(input: &Arc<Mutex<Protected>>, s: S
    ) -> Result<S::Ok, S::Error> {
  let mut p = input
  .lock()
  .map_err(|_e| ser::Error::custom("unable␣to␣acquire␣lock".
   to_string()))?;
  let u = p
  .unprotect()
  .ok_or_else(|| ser::Error::custom("invalid␣secret"))?;
  u.as_ref().serialize(s)
}
```

Single-letter variable names in `gennaro-dkg/src/secret_share.rs`.

Affected Resources

- `gennaro-dkg/src/secret_share.rs`
- `gennaro-dkg/src/protected.rs`
- `gennaro-dkg/src/lib.rs`
- `gennaro-dkg/src/participant/round4.rs`

Recommendation

Rename the concerned variables and functions such that their role is defined clearer. Refer to Fowler, M. (2019) for general guidelines on refactoring.

References

- Fowler, M. (2019) *Refactoring: Improving the Design of Existing Code*. 2nd edn. Addison-Wesley

3.8. KS-LC–8 No Secure Policy

Overall Severity:	**INFORMATIONAL**
Status:	**Resolved**

Description

The source code repository contains no instructions for how to report a security vulnerability regarding the repositories and the website, nor any security contacts.

Impact

If anyone discovers a vulnerability, they might not know who to contact in order to get it addressed before the vulnerability is exploited.

Recommendation

Create a SECURITY.md file in the root directory with all the necessary information. See the references below on how to proceed.

References

- Github - Adding a security policy
- Security Policy Generator

3.9. KS-LC–9 Consider Throwing Error Instead of Skip Validation

Overall Severity:	**INFORMATIONAL**
Status:	**Acknowledged**

Description

In multiple places across the rounds in `gennaro-dkg`, the relevant data is stored in a buffer and parsed using a loop. In some cases, the loop is interrupted using a `continue`. This is justified in some cases. For example, a party might not need to validate the input corresponding to its own id. However, in other cases, aborting the protocol or throwing an error might be a more suitable response.

Impact

By continuing the protocol where complaining or throwing an error might be more suitable, the state of the participants might end up in an invalid. Concealing any irregularities in received messages may impede the ability to identify and investigate potential malicious activities, hindering effective recovery efforts.

Evidence

```
// If not using the same generator then its a problem
if bdata.blinder_generator != self.components.pedersen_verifier_set
    .blinder_generator()
    || bdata.message_generator
        != self.components.pedersen_verifier_set.secret_generator()
    || bdata.pedersen_commitments.len() != self.threshold
{
    continue;
}
```

`gennaro-dkg/src/participant/round2.rs`. If something went wrong with the protocol, the user should be notified.

```
if !self.round1_p2p_data.contains_key(id) {
    // How would this happen?
    // Round 2 removed all invalid participants
    // Round 3 sent echo broadcast to double check valid
   participants
    self.valid_participant_ids.remove(id);
    continue;
}

```

`gennaro-dkg/src/participant/round4.rs`. If this edge case is unreachable, as the comment suggests, perhaps throwing an error might be a more suitable response.

Affected Resources

- `gennaro-dkg/src/participant/round4.rs` Line 100
- `gennaro-dkg/src/participant/round4.rs` Lines 50, 57
- `gennaro-dkg/src/participant/round5.rs` Lines 41, 48

Recommendation

Consider either throwing a suitable error OR let the end users know of irregularities in the messages received in the protocol.

3.10. KS-LC–10 Duplicate Code

Overall Severity:	**INFORMATIONAL**
Status:	**Acknowledged**

Description

Both libraries, `gennaro-dkg` and `blsful`, contain duplicate code. For example, in the case of `gennaro-dkg`, the sanity checks for values such as `broadcast_data` and `p2p_data` are duplicated all throughout the `round1.rs` · `round5.rs` files.

Impact

Generally, duplicate functionality could diverge as development progressses, causing unexpected behavior and introducing vulnerabilities.

As `gennaro-dkg` is a library implementing an academic paper, the code logic is not be expected to evolve. However, assume another developer wants to extends this library in the future (with network functionality and complaint mechanisms). In this case, having duplicated code could accidentally introduce divergent code.

Evidence

Duplicate Code in `gennaro_dkg`

As an example, `broadcast_data` is validated in `round2.rs`, `round4.rs` and `round5.rs` in (nearly) the same way. The functionality could be regrouped under an auxiliary function `validate_broadcast_data`, which takes into account the current round and the threshold to validate against.

```rust
if broadcast_data.is_empty() {
  return Err(Error::RoundError(
    Round::Two.into(),
    "Missing broadcast data from other participants".to_string(),
  ));
}
if p2p_data.is_empty() {
  return Err(Error::RoundError(
    Round::Two.into(),
    "Missing peer-to-peer data from other participants".to_string(),
  ));
}
// Allow -1 since including this participant
// This round doesn't expect this participant data included in the broadcast_data map
if broadcast_data.len() < self.threshold - 1 {
  return Err(Error::RoundError(
    Round::Two.into(),
    format!(
        "Not␣enough␣secret␣participant␣data.␣Expected␣{},␣received␣{}
    ",
        self.threshold,
        broadcast_data.len()
      ),
  ));
}
```

gennaro-dkg/src/participant/round2.rs.

Duplicate Code in `blsful`

In `blsful`, the functions that cast the type `SecretKey` to an array of `bytes` or the other way around are duplicated in different parts of the code.

For example, the following code:

```
    /// The commitment secret raw value
  #[serde(serialize_with = "traits::scalar::serialize::<C, _>")]
  #[serde(deserialize_with = "traits::scalar::deserialize::<C, _>")
  ]
  pub <<C as Pairing>::PublicKey as Group>::Scalar,
  );

  impl<C: BlsSignatureImpl> ProofCommitmentSecret<C> {
    /// Get the big-endian byte representation of this key
    pub fn to_be_bytes(&self) -> [u8; SECRET_KEY_BYTES] {
      scalar_to_be_bytes::<C, SECRET_KEY_BYTES>(self.0)
    }

    /// Get the little-endian byte representation of this key
    pub fn to_le_bytes(&self) -> [u8; SECRET_KEY_BYTES] {
      scalar_to_le_bytes::<C, SECRET_KEY_BYTES>(self.0)
    }

    /// Convert a big-endian representation of the secret key.
    pub fn from_be_bytes(bytes: &[u8; SECRET_KEY_BYTES]) ->
  CtOption<Self> {
      scalar_from_be_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
    }

    /// Convert a little-endian representation of the secret key.
    pub fn from_le_bytes(bytes: &[u8; SECRET_KEY_BYTES]) ->
  CtOption<Self> {
      scalar_from_le_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
    }
```

blsful/src/proof_commitment.rs.

is duplicated in the file `blsful/src/secret_key.rs` :

```
/// The commitment secret raw value
```

```rust
#[serde(serialize_with = "traits::scalar::serialize::<C, _>")]
#[serde(deserialize_with = "traits::scalar::deserialize::<C, _>")]
pub <<C as Pairing>::PublicKey as Group>::Scalar,
);

impl<C: BlsSignatureImpl> ProofCommitmentSecret<C> {
  /// Get the big-endian byte representation of this key
  pub fn to_be_bytes(&self) -> [u8; SECRET_KEY_BYTES] {
    scalar_to_be_bytes::<C, SECRET_KEY_BYTES>(self.0)
  }

  /// Get the little-endian byte representation of this key
  pub fn to_le_bytes(&self) -> [u8; SECRET_KEY_BYTES] {
    scalar_to_le_bytes::<C, SECRET_KEY_BYTES>(self.0)
  }

  /// Convert a big-endian representation of the secret key.
  pub fn from_be_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
   Self> {
    scalar_from_be_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
  }

  /// Convert a little-endian representation of the secret key.
  pub fn from_le_bytes(bytes: &[u8; SECRET_KEY_BYTES]) -> CtOption<
   Self> {
    scalar_from_le_bytes::<C, SECRET_KEY_BYTES>(bytes).map(Self)
  }
```

blsful/src/secret_key.rs.

Affected Resources

- gennaro-dkg/src/participant/round1.rs
- gennaro-dkg/src/participant/round2.rs
- gennaro-dkg/src/participant/round3.rs
- gennaro-dkg/src/participant/round4.rs
- gennaro-dkg/src/participant/round5.rs
- blsful/src/secret_key.rs
- blsful/src/proof_commitment.rs

Recommendation

Refactor the code base such that the duplicated sanity checks are found in a single place. Refer to Fowler, M. (2019) for general guidelines on refactoring.

References

- MITRE (2018) *CWE-1041: Use of Redundant Code*. Available at: here (Accessed: 16 November 2023).
- Fowler, M. (2019) *Refactoring: Improving the Design of Existing Code*. 2nd edn. Addison-Wesley

3.11. KS-LC–11 Examples in README.md are Outdated

Overall Severity:	**INFORMATIONAL**
Status:	**Open**

Description

The examples provided in the repository for `blsful` library do not reflect functionalities for the latest version of the library. In particular the examples for creating new secret key seems to use methods not present in the library to create new objects, in addition to missing type annotations for `SecretKey`.

Impact

There is no direct security impact, but wrong information of how to use the library could confuse the users.

Evidence

```
error[E0282]: type annotations needed for `blsful::SecretKey<C>`
    --> src/main.rs:62:9
     |
     62 |      let sk = SecretKey::random(rand_core::OsRng);
     |            ^^
     |
    help: consider giving `sk` an explicit type, where the type for
    type parameter `C` is specified
     |
     62 |      let sk: blsful::SecretKey<C> = SecretKey::random(
   rand_core::OsRng);
     |            ++++++++++++++++++++++
error[E0599]: no function or associated item named `new` found for
   struct `blsful::ProofOfPossession` in the current scope
    --> src/main.rs:64:34
      |
   64 |      let pop = ProofOfPossession::new(&sk).expect("a␣proof␣of
   ␣possession");
      |                                 ^^^ function or associated
   item not found in `ProofOfPossession<_>`
error[E0599]: no function or associated item named `hash` found for
    struct `blsful::SecretKey` in the current scope
     --> src/main.rs:66:25
       |
   66  |      let sk = SecretKey::hash(b"seed␣phrase");
       |                            ^^^^ function or associated item
   not found in `SecretKey<_>`
error[E0599]: no variant or associated item named `new` found for
   enum `Signature` in the current scope
      --> src/main.rs:69:26
```

```
        |
 69 |     let sig = Signature::new(&sk, b"00000000-0000-0000-0000-000000000000").expect("a valid signature");
        |                              ^^^ variant or associated item not found in `Signature<_>`
```

Errors compiling examples

Affected Resources

- `blsful/README.md`

Recommendation

`README.md` needs to be update with the latest guideline for the good use of the library

4. METHODOLOGY

For this engagement, Kudelski Security used a methodology that is described at high-level in this section. This is broken up into the following phases.

Kickoff
Ramp-up
Review
Report
Verify

4.1. Kickoff

The project was kicked off when all of the sales activities had been concluded. We set up a kickoff meeting where project stakeholders were gathered to discuss the project as well as the responsibilities of participants. During this meeting we verified the scope of the engagement and discussed the project activities. It was an opportunity for both sides to ask questions and get to know each other. By the end of the kickoff there was an understanding of the following:

- Designated points of contact
- Communication methods and frequency
- Shared documentation
- Code and/or any other artifacts necessary for project success
- Follow-up meeting schedule, such as a technical walkthrough
- Understanding of timeline and duration

4.2. Ramp-up

Ramp-up consisted of the activities necessary to gain proficiency on the particular project. This included the steps needed for gaining familiarity with the codebase and technological innovations utilized, such as:

- Reviewing previous work in the area including academic papers
- Reviewing programming language constructs for the languages used in the code
- Researching common flaws and recent technological advancements

4.3. Review

The review phase is where a majority of the work on the engagement was performed. In this phase we analyzed the project for flaws and issues that could impact the security posture. This included an analysis of the architecture, a review of the code, and a specification matching to match the architecture to the implemented code.

In this code audit, we performed the following tasks:

1. Security analysis and architecture review of the original protocol;
2. Review of the code written for the project;
3. Assessment of the cryptographic primitives used;
4. Compliance of the code with the provided technical documentation.

The review for this project was performed using manual methods and utilizing the experience of the reviewer. No dynamic testing was performed, only the use of custom-built scripts and tools were used to assist the reviewer during the testing. We discuss our methodology in more detail in the following subsections.

4.3.1. Code Review

We analyzed the provided code, checking for issues related to the following categories:

1. general code safety and susceptibility to known issues;
2. poor coding practices and unsafe behavior;
3. leakage of secrets or other sensitive data through memory mismanagement;
4. susceptibility to misuse and system errors;
5. error management and logging.

This is a general and not comprehensive list, meant only to give an understanding of the issues we have been looking for.

4.3.2. Cryptography

We analyzed the cryptographic primitives and components as well as their implementation. We checked in particular:

1. matching of the proper cryptographic primitives to the desired cryptographic functionality needed;
2. security level of cryptographic primitives and their respective parameters (key lengths, etc.);
3. safety of the randomness generation in general as well as in the case of failure;
4. safety of key management;
5. assessment of proper security definitions and compliance to use cases;
6. checking for known vulnerabilities in the primitives used.

4.3.3. Technical Specification Matching

We analyzed the provided documentation and checked that the code matches the specification. We checked for things such as:

1. proper implementation of the documented protocol phases.
2. proper error handling.
3. adherence to the protocol logical description.

4.4. Reporting

Kudelski Security delivered to Lit Protocol a preliminary report in PDF format that contained an executive summary, technical details, and observations about the project, which is also the general structure of the current final report.

The executive summary contains an overview of the engagement, including the number of findings as well as a statement about our general risk assessment of the project as a whole.

In the report we not only point out security issues identified but also informational findings for improvement categorized into several buckets:

1. **Critical**;
2. **High**;
3. **Medium**;
4. **Low**;
5. **Informational**.

The technical details are aimed more at developers, describing the issues, the severity ranking and recommendations for mitigation.

As we performed the audit, we also identified issues that are not security related, but are general best practices and steps, that can be taken to lower the attack surface of the project.

As an optional step, we can agree on the creation of a public report that can be shared and distributed with a larger audience.

4.5. Verify

After the preliminary findings have been delivered, we verified the fixes applied by Lit Protocol. After these fixes were verified, we updated the status of the finding in the report. The output of this phase was the current, final report with any mitigated findings noted.

5. VULNERABILITY SCORING SYSTEM

Kudelski Security utilizes a custom approach when computing the vulnerability score, based primarily on the **Impact** of the vulnerability and **Likelihood** of an attack.

Each metric is assigned a ranking of either low, medium or high, based on the criteria defined in in Section 5.2 and Section 5.3. The overall severity score is then computed as described in the next section.

5.1. Severity

Severity is the overall score of the finding, weakness or vulnerability as computed from Impact and Likelihood. Other factors, such as availability of tools and exploits, number of instances of the vulnerability and ease of exploitation might also be taken into account when computing the final severity score.

LIKELIHOOD / IMPACT	LOW	MEDIUM	HIGH
HIGH	MEDIUM	HIGH	HIGH
MEDIUM	LOW	MEDIUM	HIGH
LOW	LOW	LOW	MEDIUM

Table 1. How to compute overall Severity from Impact and Likelihood. The final severity factor might vary depending on a project's specific context and risk factors.

- **Critical** The identified issue may be immediately exploitable, causing a strong and major negative impact system-wide. They should be urgently remediated or mitigated.
- **High** The identified issue may be directly exploitable causing an immediate negative impact on the users, data, and availability of the system for multiple users.
- **Medium** The identified issue is not directly exploitable but combined with other vulnerabilities may allow for exploitation of the system or exploitation may affect singular users. These findings may also increase in severity in the future as techniques evolve.
- **Low** The identified issue is not directly exploitable but raises the attack surface of the system. This may be through leaking information that an attacker can use to increase the accuracy of their attacks.
- **Informational** Informational findings are best practice steps that can be used to harden the application and improve processes. Informational findings are not assigned a severity score and are classified as "Informational" instead.

5.2. Impact

The overall effect of the vulnerability against the system or organization based on the areas of concern or affected components discussed with the client during the scoping of the engagement.

- **High** The vulnerability has a severe effect on the company and systems or has an affect within one of the primary areas of concern noted by the client.
- **Medium** It is reasonable to assume that the vulnerability would have a measurable affect on the company and systems that may cause minor financial or reputational damage.
- **Low** There is little to no affect from the vulnerability being compromised. These vulnerabilities could lead to complex attacks or create footholds used in more severe attacks.

5.3. Likelihood

The likelihood of an attacker discovering a vulnerability, exploiting it, and obtaining a foothold varies based on a variety of factors including compensating controls, location of the application, availability of commonly used exploits, difficulty of exploitation and institutional knowledge.

- **High** It is extremely likely that this vulnerability will be discovered and abused.
- **Medium** It is likely that this vulnerability will be discovered and abused by a skilled attacker.
- **Low** It is unlikely that this vulnerability will be discovered or abused when discovered.

6. CONCLUSION

The objective of this secure code review was to evaluate whether there were any vulnerabilities that would put the users of both libraries at risk.

The Kudelski Security Team identified 2 security issues: 1 medium risks and 1 lower risk. On average, the effort needed to mitigate these risks is estimated as low.

In order to mitigate the risks posed by this engagement's findings, the Kudelski Security Team recommends applying the following best practices:

1. Improve function inputs verification
2. Explain the limitation of both libraries in terms of application

Kudelski Security remains at your disposal should you have any questions or need further assistance.

Kudelski Security would like to thank Lit Protocol for their trust, help and support over the course of this engagement and is looking forward to cooperating in the future.

RECIPIENT CONTACTS

NAME	POSITION	CONTACT INFORMATION
Mike Lodder	Senior Software Engineer	mike@litprotocol.com

DOCUMENT HISTORY

VERSION	DATE	AUTHOR	COMMENT
1.0	24.11.2023	Reviewer 1	First version with findings
1.1	03.01.2024	Reviewer 2	Updated report with re-review

BIBLIOGRAPHY

[1] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[2] Rosario Gennaro, Stanislaw Jarecki, Hugo Krawczyk, and Tal Rabin. Secure distributed key generation for discrete-log based cryptosystems. *Journal of Cryptology*, 20(1):51–83, 2007.