KUDELSKI SECURITY

Audit of Threshold ECDSA

Multisig Labs

31 October 2022

Version: 1.1

Presented by:
Kudelski Security Research Team
Kudelski Security - Nagravision Sàrl

Corporate Headquarters
Route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland

TABLE OF CONTENTS

1 EXECUTIVE SUMMARY

Kudelski Security ("Kudelski", "we"), the cybersecurity division of the Kudelski Group, was engaged by Multisig Labs ("the Client") to conduct an external security assessment in the form of a code audit of the cryptographic library ECDSA-CGGMP ("the Product"). The assessment was conducted remotely by the Kudelski Security Team and coordinated by Sylvain Pelissier, Cryptography Expert, Antonio De La Piedra, Senior Cybersecurity Engineer and Nathan Hamiel, Senior Director of Research. The audit took place from August 25, 2022 to September 9, 2022 and invloved 10 person-days of work. The audit focused on the following objectives:

- To provide a professional opinion on the maturity, adequacy, and efficiency of the software solution in exam.
- To check compliance with existing standards.
- To identify potential security or interoperability issues and include improvement recommendations based on the result of our analysis.

This report summarizes the analysis performed and findings. It also contains detailed descriptions of the discovered vulnerabilities and recommendations for remediation.

1.1 Engagement Scope

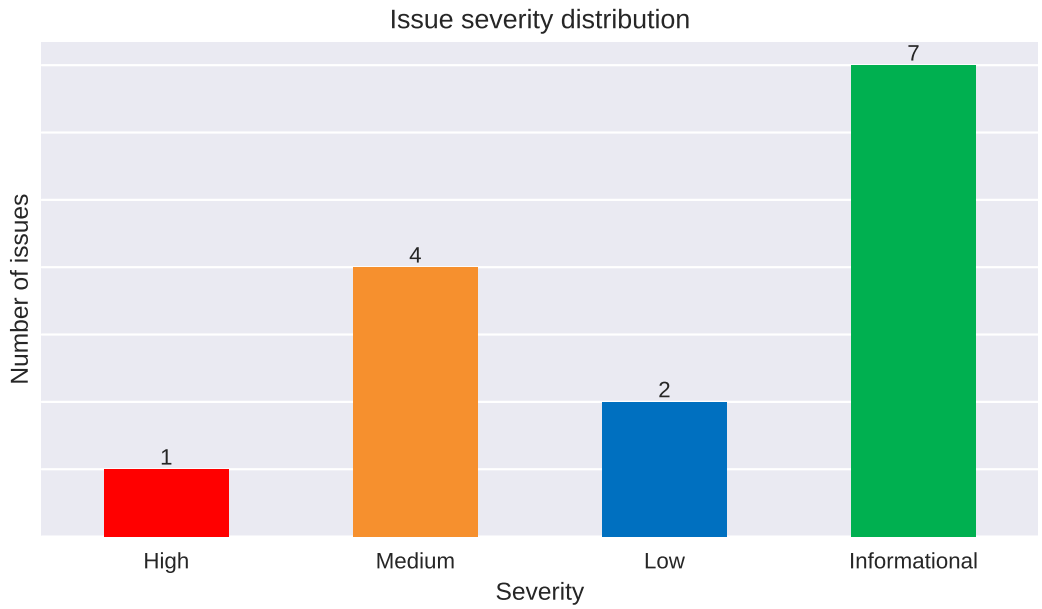
The scope of the audit was a code audit of the Product written in Go, with a particular attention to safe implementation protocols and potential for misuse and leakage of secrets. The target of the audit was the cryptographic code located in the sub-branches `protocols/cmp/sign` and `protocols/cmp/keygen` at https://github.com/multisig-labs/multi-party-sig. We audited the commit number: `1c20cbbca313a5bbbdbeae497144294658257100`. Particular attention was given to side-channel attacks, in particular constant timeness and secure erasure of secret data from memory. The Cryptography primitives implementation and identifiable abort protocols were not in the scope. Denial of Service attack vectors were not considered by the client.

1.2 Engagement Analysis

The engagement consisted of a ramp-up phase where the necessary documentation about the technological standards and design of the solution in exam was acquired,

followed by a manual inspection of the code provided by the Client and the drafting of this report.

As a result of our work, we have identified **1 High**, **4 Medium**, **2 Low** and **7 Informational** findings.



1.3 Issue Summary List

The following security issues were found:

ID	Severity	Finding	Status
KS-SBCF-F-01	**High**	ECDSA signature can be forged for every messages.	**Remediated**
KS-SBCF-F-02	**Medium**	Missing proof in round 3 of key generation	**Remediated**
KS-SBCF-F-03	**Medium**	Zero-knowledge proofs are replayable	**Acknowledged**
KS-SBCF-F-04	**Medium**	Possible nil dereference in key generation	**Remediated**
KS-SBCF-F-05	**Medium**	Collisions in hash function used for commitments	**Remediated**
KS-SBCF-F-06	**Low**	Dependency with vulnerability in codebase	**Acknowledged**

ID	Severity	Finding	Status
KS-SBCF-F-07	**Low**	SID is constant by default	**Acknowledged**

The following are observations related to general design and improvements:

ID	Severity	Finding
KS-SBCF-O-01	**Informational**	Missing security policy
KS-SBCF-O-02	**Informational**	Wrong test error message
KS-SBCF-O-03	**Informational**	Unnecessary large commitments broadcasted
KS-SBCF-O-04	**Informational**	Key generation and signing test suites fail with data race detector enabled
KS-SBCF-O-05	**Informational**	Shares are not protected
KS-SBCF-O-06	**Informational**	Taurus implementation does not provide an authenticated communication channel
KS-SBCF-O-07	**Informational**	Taurus specification document generates lambda and r parameters in keygen Round 1 from incorrect groups

2 TECHNICAL DETAILS OF SECURITY FINDINGS

This section contains the technical details of our findings as well as recommendations for mitigation.

2.1 KS-SBCF-F-01: ECDSA signature can be forged for every messages.

Severity: High

Status: Remediated

Location: protocols/cmp/sign/round5.go:154

Description

The signature $(\mathcal{O}, 0)$ is a valid signature for all the message and all the public keys. The problem comes from insufficient check on input values and also that the `Invert` function from `Scalar` interface return the value 0 when the input value is non invertible. In addition, the point at infinity $\mathcal{O}$ of the curve is represented as $(0, 0)$ in affine coordinates. Here is a proof of concept:

```
func TestSignature_Verify_Zero(t *testing.T) {
    group := curve.Secp256k1{}

    m := []byte("any message is valid")
    x := sample.Scalar(rand.Reader, group)
    X := x.ActOnBase()

    // s = 0
    s := group.NewScalar()
    assert.Equal(t, true, s.IsZero())
    R := s.ActOnBase()
    sig := &Signature{
        R: R,
        S: s,
    }
    if !sig.Verify(X, m) {
```

```
        t.Error("verify failed")
    }
}
```

This means anyone is able to forge ECDSA signature for this protocol if the function `Verify` is used to verify the signature. This attack was previously described for Java language as Psychic signature [4].

Recommendation

For an ECDSA signature (R, s), verify that s is an integer in range $[1, q-1]$ and then that R is a point on the curve and not the point at infinity. From a general point of view a value should be checked to be invertible before calling the the `Invert` function.

Status

In commit `871e651eb436ab1b6c59e4415f793374d4504eb3` the Client added the following modifications to the `Verify` function:

```
    r := sig.R.XScalar()

    if r.IsZero() || sig.S.IsZero() {
        return false
    }

    // TODO Do we also need to check for R or S > the group
     ↪  modulus?
```

The function `XScalar` call returns a value which is normalized (https://pkg.go.dev/github.com/decred/dcrd/dcrec/secp256k1/v4#FieldVal) thus, it should be less than the modulus. Then the TODO comment could be removed but a test should be added to ensure this behavior.

2.2 KS-SBCF-F-02: Missing proof in round 3 of key generation

Severity: Medium

Status: Remediated

Location: protocols/cmp/keygen/round3.go, Taurus specification.

Description

According to Canetti *et al.* p.24, Figure 6, Round 3, Step 2 [2] the proof `fac` is performed. However, in the Taurus specification this proof is not created [1].

The `fac` proof or "no small factor proof" allows a party to prove that the Paillier modulus $N = p \cdot q$ contains $p, q > 2^l$. According to the paper section 6.4.1, the 'fac proof prevents then small values close to zero to have noticeably more weight than other values, modulo $\phi(\hat{N})$

Also, if the other parties could recover her Paillier private key and all the shares of the affected party are sent to the broadcast channel, they could be decrypted by other parties, thus having access to one share of the secret. This would be equivalent to compromise the affected party and steal his share of the secret key.

Recommendation

We recommend to not deviate from protocol specifications.

Status

The Client added the missing proof in the commit 7cb2349656789af9a77f5152b7a74dcf9210add0 which include deviations in comparison to the paper of Canetti *et al*.

2.3 KS-SBCF-F-03: Zero-knowledge proofs are replayable

Severity: Medium

Status: Acknowledged

Location: pkg/zk/

Description

The zero-knowledge proofs utilized in the MPC protocol are replayable: the challenge only contains the parameters that are part of the Fiat-Shamir transformation and protocol parameter like party identifier, threshold value, ... For instance, the Schnorr proof (`sch`) implemented at `zk/sch.go` computes the challenge as:

```
func challenge(hash *hash.Hash, group curve.Curve, commitment
 ↪  *Commitment, public, gen curve.Point) (e curve.Scalar, err
 ↪  error) {
    err = hash.WriteAny(commitment.C, public, gen)
    e = sample.Scalar(hash.Digest(), group)
    return
}
```

Another example is the `prm` proof, where the challenge is computed as:

```
func challenge(hash *hash.Hash, public Public, A
 ↪  [params.StatParam]*big.Int) (es []bool, err error) {
    err = hash.WriteAny(public.N, public.S, public.T)
    for _, a := range A {
        _ = hash.WriteAny(a)
    }

    tmpBytes := make([]byte, params.StatParam)
    _, _ = io.ReadFull(hash.Digest(), tmpBytes)

    es = make([]bool, params.StatParam)
    for i := range es {
        b := (tmpBytes[i] & 1) == 1
        es[i] = b
    }

    return
}
```

According to RFC 8235 [3]:

> Finally, when a security protocol relies on the Schnorr NIZK proof for proving the knowledge of a discrete logarithm in a non-interactive way, the threat of replay attacks shall be considered. For example, the Schnorr NIZK proof might be replayed back to the prover itself (to introduce some undesirable correlation between items in a cryptographic protocol). This particular attack is

> prevented by the inclusion of the unique UserID in the hash. The verifier shall check the prover's UserID is a valid identity and is different from its own. Depending on the context of specific protocols, other forms of replay attacks should be considered, and appropriate contextual information included in OtherInfo whenever necessary.

Recommendation

We recommend the client to include an unique identifier in each proof. Moreover, to ensure that replays are not possible during several executions of the protocol a counter or a nonce must be included in the challenge computation.

Status

According to the client, the issue is mitigated by the application that generates a unique session ID for each execution of the protocol (Issue #10).

2.4 KS-SBCF-F-04: Possible nil dereference in key generation

Severity: Medium

Status: Remediated

Location: protocols/cmp/keygen/round3.go:60

Description

The method `StoreBroadcastMessage` checks that every parameter in the `broadcast3` structure is not `nil`:

```
    // check nil
    if body.N == nil || body.S == nil || body.T == nil ||
     ↪  body.VSSPolynomial == nil {
        return round.ErrNilFields
    }
    // check RID length
    if err := body.RID.Validate(); err != nil {
        return fmt.Errorf("rid: %w", err)
```

```
    }
    if err := body.C.Validate(); err != nil {
        return fmt.Errorf("chainkey: %w", err)
    }
    // check decommitment
    if err := body.Decommitment.Validate(); err != nil {
        return err
    }
```

However, the `body.SchnorrCommitments` parameter is not validated. If this parameter is set to `nil`, the following panic happens:

```
panic: value method github.com/taurusgroup/multi-party-
 ↪  sig/pkg/zk/sch.Commitment.Domain called using nil *Commitment
 ↪  pointer

goroutine 12 [running]:
github.com/taurusgroup/multi-party-sig/pkg/zk/sch.(*Commitment)
 ↪  .Domain(0xc000362008?)
    <autogenerated>:1 +0x34
[...]
FAIL
 ↪  github.com/taurusgroup/multi-party-sig/protocols/cmp/keygen
 ↪  1.694s
FAIL
```

Recommendation

We recommend the client to validate that every received parameter is not `nil`.

Status

This issue was corrected by commit `189f06564c48f257c4efa6f7fed6dca9e5672afb`.

2.5 KS-SBCF-F-05: Collisions in hash function used for commitments

Severity: Medium

Status: Remediated

Location: protocols/cmp/keygen/round3.go:154

Description

The hash function `WriteAny` does not build domain separation properly and collisions can be easily built. It can allow to make a honest participant looks culprit during the key generation. The following test is a proof of concept code:

```
func TestHash_WriteAny_Collision(t *testing.T) {
    var err error

    testFunc := func(vs ...interface{}) ([]byte, error) {
        h := New()
        for _, v := range vs {
            err = h.WriteAny(v)
            if err != nil {
                return nil, err
            }
        }
        return h.Sum(), nil
    }
    b1 := []byte("1)(big.Int\x02*data_added*")
    b2 := []byte("3")
    n2 := new(big.Int)
    n2.SetString(hex.EncodeToString(b2), 16)
    h1, err := testFunc(b1, n2)
    assert.NoError(t, err)

    b1 = []byte("1")
    b2 = []byte("*data_added*)(big.Int\x023")
    n2 = new(big.Int)
    n2.SetString(hex.EncodeToString(b2), 16)
    h2, err := testFunc(b1, n2)
    assert.NoError(t, err)
```

```
    assert.Equal(t, h1, h2)
}
```

Recommendation

Use a hash function with proper domain separation and add the previous test in the test set.

Status

The commit `a428759601b38a4199bb1920794a20eecad3c89c` introduced the domain separation: `<length domain><domain><length data><data>` with a length encoded on 8 bytes to prevent collisions.

2.6 KS-SBCF-F-06: Dependency with vulnerability in codebase

Severity: Low

Status: Acknowledged

Location: General

Description

The `golang-x-text` dependency (`pkg:golang/golang.org/x/text@v0.3.3`) is affected by an Out-of-bounds problem as described by `CVE-2021-38561` in https://ossindex.sonatype.org/vulnerability/CVE-2021-38561?component-type=golang&component-name=golang.org.

Recommendation

We recommend the client to updated the dependency.

Status details

The client acknowledge the finding and will fix the issued (https://github.com/multisig-labs/multi-party-sig/issues/14).

2.7 KS-SBCF-F-07: SID is constant by default

Severity: Low

Status: Acknowledged

Location: code/protocols/cmp/keygen/keygen.go:19

Description

By default, the session identifier depends on the curve group name, the party identifiers and the threshold. Thus, for two different sessions with the same threshold, the session identifier stays constants even though the code comments of the `NewSession` function says:

```
// NewSession creates a new *Helper which can be embedded in the
 ↪  first Round,
// so that the full struct implements Session.
// `sessionID` is an optional byte slice that can be provided by
 ↪  the user.
// When used, it should be unique for each execution of the
 ↪  protocol.
// It could be a simple counter which is incremented after
 ↪  execution,  or a common random string.
// `auxInfo` is a variable list of objects which should be
 ↪  included in the session's hash state.
func NewSession(info Info, sessionID []byte, pl *pool.Pool,
 ↪  auxInfo ...hash.WriterToWithDomain) (*Helper, error) {
```

This would allow to replay some message in different sessions of the protocol. For example, at round 1 of the key generation, the variable V_i may be replayed after being recorded in a previous session.

Recommendation

We recommend to ensure the `sessionID` value is unique per protocol execution.

Status details

According to the client, the issue is mitigated by the application that uses this library which generates a unique session ID for each execution of the protocol.

3 OTHER OBSERVATIONS

This section contains additional observations that are not directly related to the security of the code, and as such have no severity rating or remediation status summary. These observations are either minor remarks regarding good practice or design choices or related to implementation and performance. These items do not need to be remediated for what concerns security, but where applicable we include recommendations.

3.1 KS-SBCF-O-01: Missing security policy

Location: .

Description

Currently there is no instructions for how to report a security vulnerability regarding the repository nor security contacts.

Recommendation

Create a `SECURITY.md` file in the root directory with all the necessary information. See for example: https://docs.github.com/en/code-security/getting-started/adding-a-security-policy-to-your-repository

Notes

The client will fix the problem.

3.2 KS-SBCF-O-02: Wrong test error message

Location: protocols/cmp/sign/sign_test.go:55

Description

Taproot is mentioned in CMP tests:

```
for _, r := range rounds {
        require.IsType(t, &round.Output{}, r, "expected result
↪   round")
```

```
        resultRound := r.(*round.Output)
        require.IsType(t, &ecdsa.Signature{}, resultRound.Result,
↪   "expected taproot signature result")
        signature := resultRound.Result.(*ecdsa.Signature)
        assert.True(t, signature.Verify(publicPoint,
↪   messageHash), "expected valid signature")
    }
```

Recommendation

The messages should indicate ECDSA signature.

Notes

The client will fix the problem.

3.3 KS-SBCF-O-03: Unnecessary large commitments broadcasted

Location: code/protocols/cmp/keygen/round1.go:92

Description

Commitment V_i should be 32-byte long according to Canetti *et al.* [2] but in the implementation they are 64-byte long. This creates unnecessary bandwidth consumption.

Recommendation

Commitment length should be reduced.

Notes

For the application, the additional bandwidth of 32 bytes does not matter for the client.

3.4 KS-SBCF-O-04: Key generation and signing test suites fail with data race detector enabled

Location: src/internal/test/round.go lines 40 and 55, src/protocols/cmp/keygen/keygen_test.go, src/protocols/cmp/sign/sign_test.go

Description

When running `go test -race -run` for the tests: `TestKeygen`, `TestRefresh` in keygen_test.go and `TestRound` in sign_test.go, we obtain:

```
WARNING: DATA RACE
Write at 0x00c000113110 by goroutine 25:
  github.com/taurusgroup/multi-party-
    ↪ sig/internal/test.Rounds.func1()
      internal/test/round.go:55 +0x1eb
  golang.org/x/sync/errgroup.(*Group).Go.func1()
      go/pkg/mod/golang.org/x/sync@v0.0.0-20210220032951-
        ↪ 036812b2e83c/errgroup/errgroup.go:57
        ↪ +0x91

Previous write at 0x00c000113110 by goroutine 24:
  github.com/taurusgroup/multi-party-
    ↪ sig/internal/test.Rounds.func1()
      internal/test/round.go:55 +0x1eb
  golang.org/x/sync/errgroup.(*Group).Go.func1()
      go/pkg/mod/golang.org/x/sync@v0.0.0-20210220032951-
        ↪ 036812b2e83c/errgroup/errgroup.go:57
        ↪ +0x91

Goroutine 25 (running) created at:
  golang.org/x/sync/errgroup.(*Group).Go()
      pkg/mod/golang.org/x/sync@v0.0.0-20210220032951-
        ↪ 036812b2e83c/errgroup/errgroup.go:54
        ↪ +0xee
  github.com/taurusgroup/multi-party-sig/internal/test.Rounds()
      internal/test/round.go:40 +0x24b
```

```
  github.com/taurusgroup/multi-party-
    ↪ sig/protocols/cmp/keygen.TestKeygen()
      protocols/cmp/keygen/keygen_test.go:80 +0x144
  testing.tRunner()
      /usr/lib/go-1.18/src/testing/testing.go:1439 +0x213
  testing.(*T).Run.func1()
      /usr/lib/go-1.18/src/testing/testing.go:1486 +0x47
--- FAIL: TestKeygen (20.49s)
    testing.go:1312: race detected during execution of test
```

This dead race happens due to the testing subsystem of Taurus, defined in `internal/test`. More precisely, in the line 55 of the Rounds function:

```
    // get the second set of messages
    for id := range rounds {
        idx := id
        r := rounds[idx]
        errGroup.Go(func() error {
            var rNew, rNewReal round.Session
            if rule != nil {
                rReal := getRound(r)
                rule.ModifyBefore(rReal)
                outFake := make(chan *round.Message, N+1)
                rNew, err = r.Finalize(outFake)
                close(outFake)
                rNewReal = getRound(rNew)
                rule.ModifyAfter(rNewReal)
                for msg := range outFake {
                    rule.ModifyContent(rNewReal, msg.To,
  ↪ getContent(msg.Content))
                    out <- msg
                }
            } else {
                rNew, err = r.Finalize(out)
            }
```

where `rNew` is updated:

```
            } else {
rNew, err = r.Finalize(out)
}
```

Notes

The finding will be investigate in the future by the client.

3.5 KS-SBCF-O-05: Shares are not protected

Location: src/protocols/cmp/sign and src/protocols/cmp/keygen

Description

Generated shares after key generation are not protected. We recommend the client to encrypt and authenticated those after key generation and validate their authenticity prior to signing.

Notes

The client considered this observation not in scope for the library and it must be addressed at application level.

3.6 KS-SBCF-O-06: Taurus implementation does not provide an authenticated communication channel

Location:: General

Description

The work of Canetti *et al.* [2] requires a an authenticated and synchronous broadcast mechanism for communication (see p. 6, Communication model).

We recommend the client to consider the warnings described in the paper when not using a synchronous broadcast mechanism:

We note that the use of authenticated communication is in fact essential for obtaining proactive security. Indeed, without already-established authenticated communication, an adversary that formally "left" a previously corrupted party and controls all the communication between the party and the rest of the network can continue impersonating that party indefinitely [15]. Furthermore, the use of a synchronous broadcast mechanism is essential for accountability: Accountability by definition requires consensus, and without bounded communication delay it is impossible to hold a signatory accountable for not responding.

Finally, we stress the importance of providing authentication and authenticity to the communication channel that the client relies on.

Notes

The client considered this observation not in scope for the library and it must be addressed at application level.

3.7 KS-SBCF-O-07: Taurus specification document generates lambda and r parameters in keygen Round 1 from incorrect groups

Location: https://github.com/taurusgroup/multi-party-sig/blob/main/docs/Threshold.pdf

Description

Taurus provides performs the key generation and refresh/auxiliary parameter generation at the same time in their implementation, which is specified [1].

In [1] p.2, Round 1, third step, λ is generated from $Z^*_{N^i}$ and r from $Z^*_{\phi(N^i)}$. However, according to [2], p.24, Figure 6, Round 1, r is generated from $Z^*_{N^i}$ and λ from $Z^*_{\phi(N^i)}$.

We encourage the client to analyze if the implementation follows the document of Taurus (which is not part of the scope of this audit), hence contradicting the paper of Canetti et al [2].

4 APPENDIX A: ABOUT KUDELSKI SECURITY

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

Kudelski Security
5090 North 40th Street
Suite 450
Phoenix, Arizona 85018

5 APPENDIX B: METHODOLOGY

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

Figure 1: Methodology flow

5.1 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

5.2 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

5.3 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.

Code Safety

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

- General code safety and susceptibility to known issues
- Poor coding practices and unsafe behavior
- Leakage of secrets or other sensitive data through memory mismanagement
- Susceptibility to misuse and system errors
- 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.

Cryptography

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

- Matching of the proper cryptographic primitives to the desired cryptographic functionality needed
- Security level of cryptographic primitives and their respective parameters (key lengths, etc.)

- Safety of the randomness generation in general as well as in the case of failure
- Safety of key management
- Assessment of proper security definitions and compliance to use cases
- Checking for known vulnerabilities in the primitives used

Technical Specification Matching

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

- Proper implementation of the documented protocol phases
- Proper error handling
- Adherence to the protocol logical description

5.4 Reporting

Kudelski delivered to the Client 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:

- High
- Medium
- Low
- 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.

5.5 Verify

After the preliminary findings have been delivered, we verified the fixes applied by the Client. 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.

5.6 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. Since this is a library, we ranked some of these vulnerabilities potentially higher than usual, as we expect the code to be reused across different applications with different input sanitization and parameters.

Correct memory management is left to Go and was therefore not in scope. Zeroization of secret values from memory is also not enforceable at a low level in a language such as Go.

While assessment 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 **Appendix C** of this document.

6 APPENDIX C: SEVERITY RATING DEFINITIONS

Kudelski Security uses a custom approach when determining criticality of identified issues. This is meant to be simple and fast, providing customers with a quick at a glance view of the risk an issue poses to the system. As with anything risk related, these findings are situational. We consider multiple factors when assigning a severity level to an identified vulnerability. A few of these include:

- Impact of exploitation
- Ease of exploitation
- Likelihood of attack
- Exposure of attack surface
- Number of instances of identified vulnerability
- Availability of tools and exploits

Severity	Definition
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.

REFERENCES

[1] J. P. Aumasson, A Hamelink, and L Meier. 2021. Adaptations from CGGMP21. Retrieved from https://raw.githubusercontent.com/multisig-labs/multi-party-sig/main/docs/Threshold.pdf

[2] Ran Canetti, Rosario Gennaro, Steven Goldfeder, Nikolaos Makriyannis, and Udi Peled. 2020. UC non-interactive, proactive, threshold ECDSA with identifiable aborts. In *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security* (CCS '20), Association for Computing Machinery, New York, NY, USA, 1769–1787. Retrieved from https://doi.org/10.1145/3372297.3423367

[3] Feng Hao. 2017. Schnorr Non-interactive Zero-Knowledge Proof. DOI:https://doi.org/10.17487/RFC8235

[4] Neil Madden. 2022. CVE-2022-21449: Psychic signatures in java. Retrieved from https://neilmadden.blog/2022/04/19/psychic-signatures-in-java/