# DKLS18 2-of-2 Threshold ECDSA

## Coinbase

03 November 2021

Version: 1.0

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

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

# TABLE OF CONTENTS

**1 EXECUTIVE SUMMARY** **4**

1.1 Engagement Scope . . . . . 4

1.2 Engagement Analysis . . . . . 4

1.3 Issue Summary List . . . . . 5

**2 TECHNICAL DETAILS OF SECURITY FINDINGS** **7**

2.1 KS-SBCF-F-01: Add extra parameter in challenge c to avoid replay attacks 7

2.2 KS-SBCF-F-02: Base point (generator g of the group G over the curve E) is missing in the hash computation of the Schnorr proof . . . . . 7

2.3 KS-SBCF-F-03: Validate input parameters in Schnorr proofs . . . . . 8

2.4 KS-SBCF-F-04: Check that the witness is not 0 in the Prove implementation 9

2.5 KS-SBCF-F-05: Missing modular reductions in the Schnorr proof implementation, in the signing operation and in the transfer function . . . . . 9

2.6 KS-SBCF-F-06: Use rejection sampling when generating random scalars \in Z_q to avoid mod bias . . . . . 10

2.7 KS-SBCF-F-07: Missing EC point validation . . . . . 11

2.8 KS-SBCF-F-08: Enforce input validation to avoid pointer nil dereferences . 11

**3 OTHER OBSERVATIONS** **13**

3.1 KS-SBCF-O-01: Absence of robust 2-party channel . . . . . 13

3.2 KS-SBCF-O-02: There is no test suite for the implementation of zero-knowledge proofs and commitments . . . . . 13

3.3 KS-SBCF-O-03: Test suite of dkg does not test for edge cases and aborts . 14

3.4 KS-SBCF-O-04: Enable subgroup check if curves with composite group and prime order subgroup are used . . . . . 14

3.5 KS-SBCF-O-05: Speed-up: Perform simultaneous multiplication in the verification of Schnorr proofs and in padTransfer (OT) . . . . . 14

3.6 KS-SBCF-O-06: Use of parameter V in ECDSA signature . . . . . 15

3.7 KS-SBCF-O-07: Provide Alice the capabilities to verify the signature . . . . 15
3.8 KS-SBCF-O-08: There is no domain separation in proofs . . . . . 15
3.9 KS-SBCF-O-09: Initialize the nonce with a random value in padTransfer . . 16
3.10 KS-SBCF-O-10: Comment in sign.go concerning Alice responses . . . . . 17
3.11 KS-SBCF-O-11: Do no concatenate, but separately hash to avoid length-extension attacks in padTransfer . . . . . 17

**4 APPENDIX A: ABOUT KUDELSKI SECURITY 18**

**5 APPENDIX B: METHODOLOGY 19**

5.1 Kickoff . . . . . 19
5.2 Ramp-up . . . . . 19
5.3 Review . . . . . 20
5.4 Reporting . . . . . 21
5.5 Verify . . . . . 22
5.6 Additional Note . . . . . 22

**6 APPENDIX C: DOCUMENT HISTORY 23**

**7 APPENDIX D: SEVERITY RATING DEFINITIONS 24**

**REFERENCES 25**

# 1 EXECUTIVE SUMMARY

Kudelski Security ("Kudelski", "we"), the cybersecurity division of the Kudelski Group, was engaged by Coinbase ("the Client") to conduct an external security assessment in the form of a code audit of the cryptographic library DKLS ("the Product"). The audit was conducted remotely by the Kudelski Security Team. The audit took place from October 18th, 2021 to October 29th, 2021. 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.

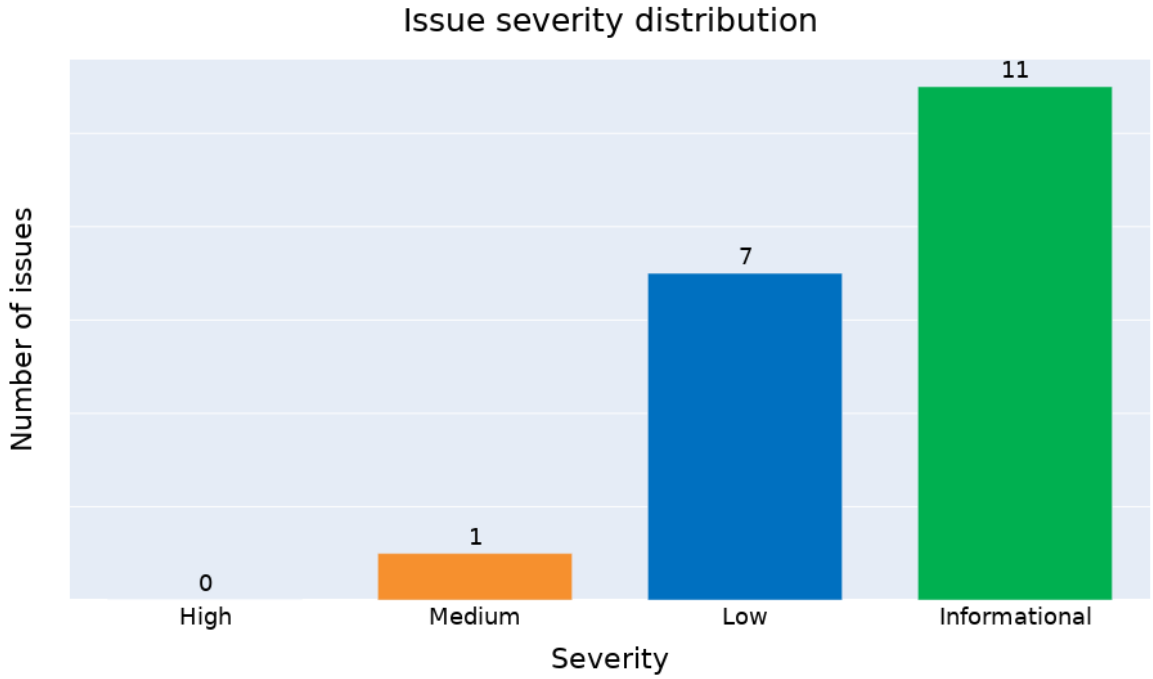
## 1.1 Engagement Scope

The scope of the audit was a code audit of the Product written in Go, with a particular attention to safe implementation of hashing, randomness generation, protocol verification, and potential for misuse and leakage of secrets. The target of the audit was the cryptographic module `kryptology`, particularly the `dkls` implementation located at `src/kryptology/pkg/tecdsa/dkls`. Particular attention was given to side-channel attacks, in particular constant timeness.

## 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 Medium**, **7 Low** and **11 Informational** findings.




## 1.3 Issue Summary List

The following security issues were found:

| ID | Severity | Finding | Status |
|---|---|---|---|
| KS-SBCF-F-01 | **Medium** | Add extra parameter in challenge c to avoid replay attacks | **Open** |
| KS-SBCF-F-02 | **Low** | Base point (generator g of the group G over the curve E) is missing in the hash computation of the Schnorr proof | **Acknowledged** |
| KS-SBCF-F-03 | **Low** | Validate input parameters in Schnorr proofs | **Open** |
| KS-SBCF-F-04 | **Low** | Check that the witness is not 0 in the Prove implementation | **Acknowledged** |
| KS-SBCF-F-05 | **Low** | Missing modular reductions in the Schnorr proof implementation, in the signing operation and in the transfer function | **Acknowledged** |

| ID | Severity | Finding | Status |
|---|---|---|---|
| KS-SBCF-F-06 | **Low** | Use rejection sampling when generating random scalars \in Z_q to avoid mod bias | **Acknowledged** |
| KS-SBCF-F-07 | **Low** | Missing EC point validation | **Acknowledged** |
| KS-SBCF-F-08 | **Low** | Enforce input validation to avoid pointer nil dereferences | **Acknowledged** |

The following are observations related to general design and improvements:

| ID | Severity | Finding |
|---|---|---|
| KS-SBCF-O-01 | **Informational** | Absence of robust 2-party channel |
| KS-SBCF-O-02 | **Informational** | There is no test suite for the implementation of zero-knowledge proofs and commitments |
| KS-SBCF-O-03 | **Informational** | Test suite of dkg does not test for edge cases and aborts |
| KS-SBCF-O-04 | **Informational** | Enable subgroup check if curves with composite group and prime order subgroup are used |
| KS-SBCF-O-05 | **Informational** | Speed-up: Perform simultaneous multiplication in the verification of Schnorr proofs and in padTransfer (OT) |
| KS-SBCF-O-06 | **Informational** | Use of parameter V in ECDSA signature |
| KS-SBCF-O-07 | **Informational** | Provide Alice the capabilities to verify the signature |
| KS-SBCF-O-08 | **Informational** | There is no domain separation in proofs |
| KS-SBCF-O-09 | **Informational** | Initialize the nonce with a random value in padTransfer |
| KS-SBCF-O-10 | **Informational** | Comment in sign.go concerning Alice responses |
| KS-SBCF-O-11 | **Informational** | Do no concatenate, but separately hash to avoid length-extension attacks in padTransfer |

# 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: Add extra parameter in challenge c to avoid replay attacks

**Severity:** Medium

**Status:** Open

**Location:** schnorr.go:52, schnorr.go:27

### Description

This issue is mainly relevant if the implementation is deployed in a multi-user environment and the communication channel allows message replays by an eavesdropper. If one proof is captured by an attacker, it can be replayed to prove that they have the secret key related to a public key, even if this party doesn't. This is possible because the challenge only contains (see schnorr.go:52, schnorr.go:27) the commitment and the statement.

### Recommendation

Follow https://datatracker.ietf.org/doc/html/rfc8235 to construct the challenge $c$ using the following structure: c = H(g || V || A || UserID || OtherInfo) and use UserID or OtherInfo to make the challenge not replayable, for instance using a random nonce provided by one of the parties.

## 2.2 KS-SBCF-F-02: Base point (generator g of the group G over the curve E) is missing in the hash computation of the Schnorr proof

**Severity:** Low

**Status:** Acknowledged

**Location:** schnorr.go:52, schnorr.go:27

**Description**

The implementation of the NIZK Schnorr proof deviates from the RFC 8235 and does not include the base point in the challenge.

According to the implementation, it can be possible to forge a valid response by an attacker if the Prover convinces the Verifier to use a particular generator. For instance, We can choose $G$ = point-at-infinity. In this case, pk = sk. However, in the implementation that we have audited, the Verifier maintains its own copy of the generator.

Moreover, another reason for including the base point is to provide separation in deployments where different elliptic curves are utilized and there is a certain probability of obtaining the same challenge if the base point is not included (even if the probability in this is very small).

**Recommendation**

Our recommendation is to follow RFC 8235 and insert $g$, the group generator, in the hash computation (that is, the challenge $c$) in the Prover and Verifier function as described in https://datatracker.ietf.org/doc/html/rfc8235.

**Notes**

Coinbase ackwnoledged that the verifier is not using the generator sent by the prover.

## 2.3 KS-SBCF-F-03: Validate input parameters in Schnorr proofs

**Severity:** Low

**Status:** Open

**Location:** schnorr.go:52, schnorr.go:27

**Description**

There is no input validation in the Schnorr proof implementation. For instance, the public key is not validated and the parameters $C$ and $S$ are not checked.

**Recommendation**

To avoid interoperability problems check if either $C$ or $S$ are `nil`. Moreover, if the public key Pub is sent to the verifier, check if it belongs to the curve

and that is not the point-at-infinity. This check is described in Section 3.2 of https://datatracker.ietf.org/doc/html/rfc8235#page-6.

## 2.4 KS-SBCF-F-04: Check that the witness is not 0 in the Prove implementation

**Severity:** Low

**Status:** Acknowledged

**Location:** schnorr.go:52, schnorr.go:27

### Description

Otherwise, $h$ becomes 1, $s$ becomes $k$ and $k$ is leaked. Check also that $k$ is not zero, even if the probability is very small. If any of these cases happens, abort.

### Recommendation

Even if it the probability of obtaining $0$ is negligible, perform input validation on the witness in case of misuse.

## 2.5 KS-SBCF-F-05: Missing modular reductions in the Schnorr proof implementation, in the signing operation and in the transfer function

**Severity:** Low

**Status:** Acknowledged

**Location:** schnorr.go:52, schnorr.go:27, sign:129, ot.go:397

### Description

The challenge $c$ is not reduced mod $q$ in the Prove and Verify implementation of the Schnorr proof. To avoid interoperability problems, obtain $S$ as $s = a \cdot c + k$ mod $q$. The same applies to the digest computed in the signing operation. Moreover, in the first round of the signing protocol, $k_A$ is computed as $k_A = H(R') + kA'$ but is not module $q$ reduced. Further, the parameters $t_A, t_B$ are not reduced mod $q$ in the transfer function either.

**Recommendation**

In order to avoid interoperability problems we recommend Coinbase to reduce it mod $q$ after the computation of the hash operation. Rejection sampling could be used instead of the mod operation in this case to avoid bias.

Perform also the modular reduction in the other cases.

## 2.6 KS-SBCF-F-06: Use rejection sampling when generating random scalars \in Z_q to avoid mod bias

**Severity:** Low

**Status:** Acknowledged

**Location:** ec_scalar.go:81

**Description**

In ec_scalar.go, random scalars (for instance, in the Bitcoin curve) are generated as:

```
81 func (k K256Scalar) Random() (*big.Int, error) {
82     b := make([]byte, 48)
83     n, err := crand.Read(b)
84     if err != nil {
85         return nil, err
86     }
87     if n != 48 {
88         return nil, fmt.Errorf("insufficient bytes read")
89     }
90     v := new(big.Int).SetBytes(b)
91     v.Mod(v, btcec.S256().N)
92     return v, nil
93 }
```

The modular operation with certain orders could create a mod bias. Particularly with those curves which rely on groups of composite order where $q$ is not a power of 2. For instance, this affects the generation of $k$ the in Schnorr Prove implementation.

**Recommendation**

We recommend to use rejection sampling as described in https://research.kudelskisecurity.com/2020/07/28/the-definitive-guide-to-modulo-bias-and-how-to-avoid-it/.

## 2.7 KS-SBCF-F-07: Missing EC point validation

**Severity:** Low

**Status:** Acknowledged

**Location:** schnorr.go:52, schnorr.go:27, sign.go:129, ot.go:78

**Description**

In the DKLS implementation the following points (generally, acting as public keys) are not validated and could create interoperability problems and/or make the protocol abort by one of the parties:

- Validate $D_B$ in the multiplication and instance key exchange phase.
- Validate $R'$ sent by Alice to Bob in step 4 of signing.
- Validate public key in seedOTReceiver.

**Recommendation**

Validate these points:

- Is this point part of the curve ? Is the point-at-infinity ?
- Can be used by one of the parties to abort the protocol ?

## 2.8 KS-SBCF-F-08: Enforce input validation to avoid pointer nil dereferences

**Severity:** Low

**Status:** Acknowledged

**Location:** conn.go:17, conn.go:23, conn.go:29, multiply.go:36, multiply:46, ot.go:293, ot.go:312, proto.go:101, schnorr.go:24, schnorr.go:80, sign.go:65

### Description

In certain functions of the DKLS implementation, pointers are not validated:

- conn.go: check if pipeWrapper pointer is `nil`.
- multiply.go: check if seedOTReceiver, seedOTsender are `nil`.
- ot.go: check if newCOTReceiver sender, receiver are `nil`.
- proto.go check if NewAlice and params pointer are `nil`.
- schnorr.go: In the Prover and proveCommit implementation, check if `x` is `nil`.
- sign.go: check if the params pointer is `nil`.

### Recommendation

To avoid a `nil` pointer dereference, we recommend to enforce input validation in the described functions. Then, abort if the pointers are `nil`.

# 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: Absence of robust 2-party channel

**Location:** conn.go

**Description**

Alice and Bob must communicate using a robust channel, where not only both can write and read in a sequential manner but where the following properties are provided:

- Messages should be at least, authenticated, in order to avoid replay attacks (this is particularly important to avoid the replaying of Schnorr proofs).
- Aborts due to input validation reported in the Findings section of this report should be notified to the parties involved in the protocol.

## 3.2 KS-SBCF-O-02: There is no test suite for the implementation of zero-knowledge proofs and commitments

**Location:** src/kryptology/test/

**Description**

There are no test methods for the functions defined in schnorr.go, particularly:

- The discrete log proofs.
- The commit-and-prove scheme.

## 3.3 KS-SBCF-O-03: Test suite of dkg does not test for edge cases and aborts

**Location:** src/kryptology/test/

**Description**

For instance:

- During key generation if the commit-and-prove release fails.
- During key generation if the Schnorr proofs does not verify.
- If the input parameters of each message are incorrect.

## 3.4 KS-SBCF-O-04: Enable subgroup check if curves with composite group and prime order subgroup are used

**Location:** For curves defined pkg/core/curves/ that can be used in the DKLS implementation

**Description**

For instance if the bls12-381 and curve25519 curves will be used.

## 3.5 KS-SBCF-O-05: Speed-up: Perform simultaneous multiplication in the verification of Schnorr proofs and in padTransfer (OT)

**Location:** schnorr.go, ot.go:104

**Description**

For point arithmetic operations of type $a \cdot A + b \cdot B$ consider implementing a simultaneous point multiplication method. Moreover, since multiple Schnorr proofs are performed during key generation and signing, consider enabling also precomputation with bases $G$ and public key. There exist multiple algorithms with different trade-offs. You can check for instance those implemented in the research-oriented RELIC library: https://github.com/relic-toolkit/relic/blob/main/src/ep/relic_ep_mul_sim.c.

## 3.6 KS-SBCF-O-06: Use of parameter V in ECDSA signature

**Location:** sign.go:246

In the signFinal function of sign.go the parameter $V$ of the bitcoin ECDSA signature is always updated according to the parity of the coordinate $Y$ of the component $r$.

```
246 bob.Sig.V = int(R.Y.Bit(0))
```

This value is either 1 or 0 in this case, converted to an integer type. However, bitcoin describes this parameter as a byte type, computes as $v = 27 + (y \bmod 2)$.

Further, in case this implementation targets Ethereum and EIP-155, and extra parameter should be considered (see https://github.com/ethereum/EIPs/blob/master/EIPS/eip-155.md.

## 3.7 KS-SBCF-O-07: Provide Alice the capabilities to verify the signature

**Location:** sign.go

In the original description of 2-of-2 [1] Bob has the responsibility of verifying the signature after creation. However, the case where Bob decides to accept an invalid signature or output and invalid one is not contemplated.

In the case the protocol implementation is going to be use with other implementations, you could provide Alice the ability to also verify that the signature is correct. If this is not case, Alice can abort the protocol.

## 3.8 KS-SBCF-O-08: There is no domain separation in proofs

**Location:** schnorr.go:52, schnorr.go:27

**Description**

Domain separation is typically mandatory in schemes proved in the UC model, which is not the case of [1].

However you could include a public salt in the generation of the challenge, known and public to both parties to provide domain separation.

## 3.9 KS-SBCF-O-09: Initialize the nonce with a random value in padTransfer

**Location:** ot.go:104

In padTransfer, as described in [1]:

```
However, if (public)
nonces are used in each of the hash invocations, then the
Public Key phase can be run once and the resulting (single)
public key B can be reused in as many Transfer and Verification
phases as required without sacrificing security.
```

In ot.go, Coinbase does:

```
104 func (sender *seedOTSender) padTransfer(rw io.ReadWriter) error {
105     enc := gob.NewEncoder(rw)
106     dec := gob.NewDecoder(rw)
107     // returns the challenges xi, concated into a block.
108     input := &seedOtTransfer{}
109     if err := dec.Decode(input); err != nil {
110         return err
111     }
112     result := &seedOtVerification{}
113
114     for i := 0; i < kappa; i++ {
115         d, err := input[i].ScalarMult(sender.b)
116         if err != nil {
117             return err
118         }
119         sender.Rho[i][0] = sha256.Sum256(append(d.Bytes(), byte(i)))
```

Nonces are typically random values. Our recommendation is to initialize the nonce as random at the beginning of the protocol in a way that both parties know the initialization value. Then, increase accordingly.

## 3.10 KS-SBCF-O-10: Comment in sign.go concerning Alice responses

**Location:** sign.go:162

The following comment is written in sign.go:162:

```
162     // Alice's response here is _two_ (i.e., collated) responses to the
          ↪ multiplication protocol.
163     // followed by Alice's R', followed by a schnorr proof for R (!).
          ↪ folllowed by \eta^{\phi} and \eta^{sig}.
```

However, to best of our knowledge, we cannot find the Schnorr proof for R in either the paper or in the code.

## 3.11 KS-SBCF-O-11: Do no concatenate, but separately hash to avoid length-extension attacks in padTransfer

**Location:** ot.go

### Description

SHA-2 is sensitive to length-extension attacks. In padTransfer functions there is a concatenation between a random value (the nonce), user generated and the sensitive value $d$.

### Recommendation

Even if the extend of a length-extension attack in this case is very small, it is a good practice to avoid concatenation and update the hash computation with each parameter separately.

# 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.


## 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 TypeScript 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 TypeScript.

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 D** of this document.

# 6 APPENDIX C: DOCUMENT HISTORY

| Version | Status | Date | Document Owner | Comments |
|---|---|---|---|---|
| 0.1 | Draft | 18th October 2021 | Kudelski Security Research Team | |

| Reviewer | Position | Date | Version | Comments |
|---|---|---|---|---|
| Nathan Hamiel | Head of Security Research | 3rd November 2021 | Final | |

| Approver | Position | Date | Version | Comments |
|---|---|---|---|---|
| Nathan Hamiel | Head of Security Research | 3rd November 2021 | Final | |

# 7 APPENDIX D: 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]
Jack Doerner, Yashvanth Kondi, Eysa Lee, and Abhi Shelat. 2018. Secure two-party threshold ECDSA from ECDSA assumptions. *IACR Cryptol. ePrint Arch.* (2018), 499. Retrieved from https://eprint.iacr.org/2018/499