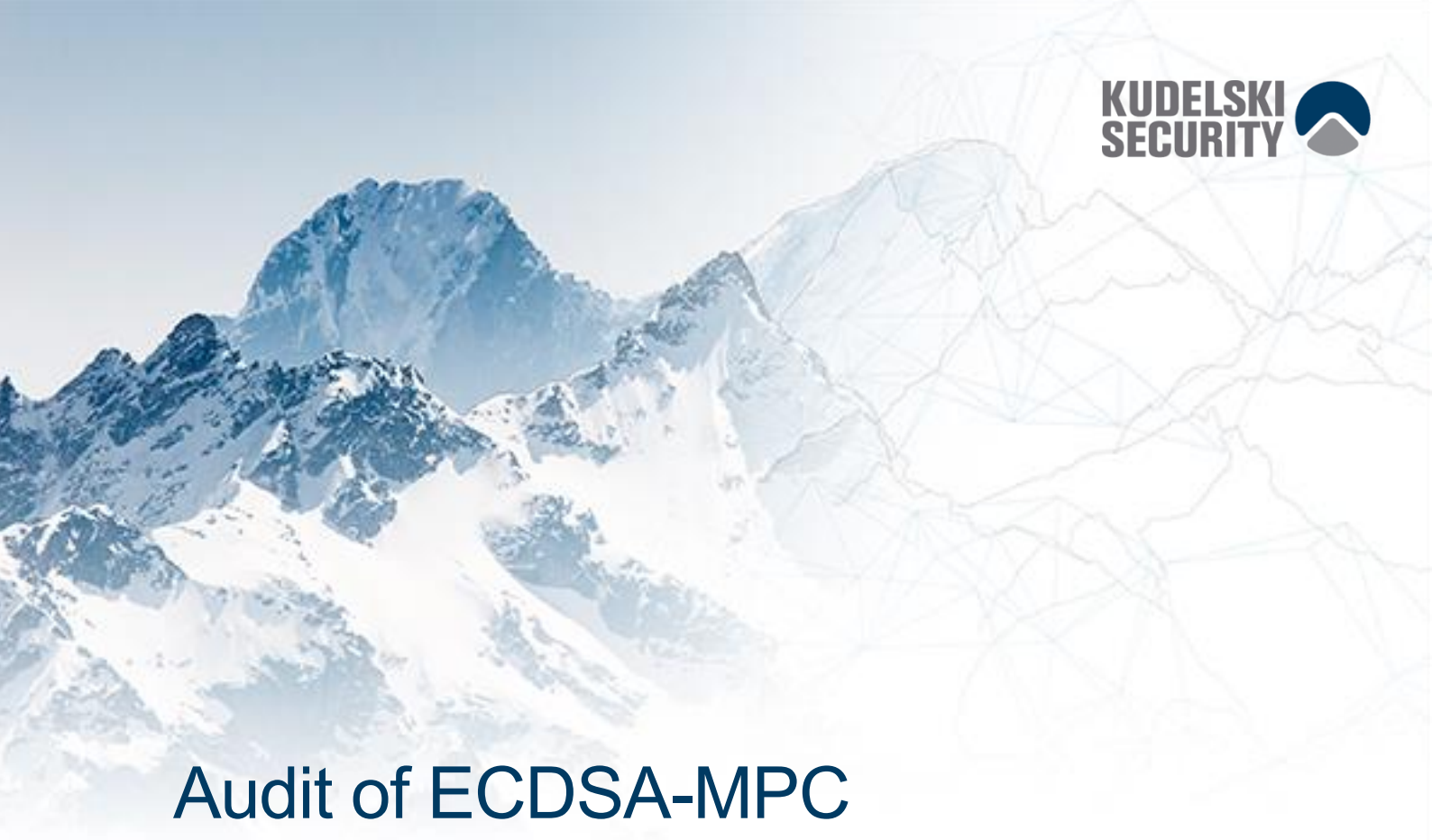

Audit of ECDSA-MPC

ING

26 March 2021
Version: 1.0

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

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

DOCUMENT PROPERTIES

Version:	1.0
File Name:	Audit_ING_TECDSA
Publication Date:	26 March 2021
Confidentiality Level:	For public release
Document Owner:	Tommaso Gagliardoni
Document Recipient:	Shariff Lutfi
Document Status:	Approved

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

Kudelski Security ("Kudelski", "we"), the cybersecurity division of the Kudelski Group, was engaged by ING ("the Client") to conduct an external security assessment in the form of a code audit of the cryptographic library `ecdsa-mpc` ("the Product").

The assessment was conducted remotely by the Kudelski Security Team and coordinated by Dr. Tommaso Gagliardoni, Senior Cryptography Expert, Yolan Romailler Cryptography Expert and Nathan Hamiel, Head of Cybersecurity Research. 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 Rust, 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 code located in the sub-branches `/src/algorithms` and `/src/ecdsa` at https://github.com/ing-bank/threshold-signatures.

We audited the commit number: `cc86590a2fbc8ee41b6cede2bfbc48c03a0f4da5`.

Particular attention was given to side-channel attacks, in particular constant timeness and secure erasure of secret data from memory.

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 identified **6 Medium**, **3 Low**, and **7 Informational** findings.

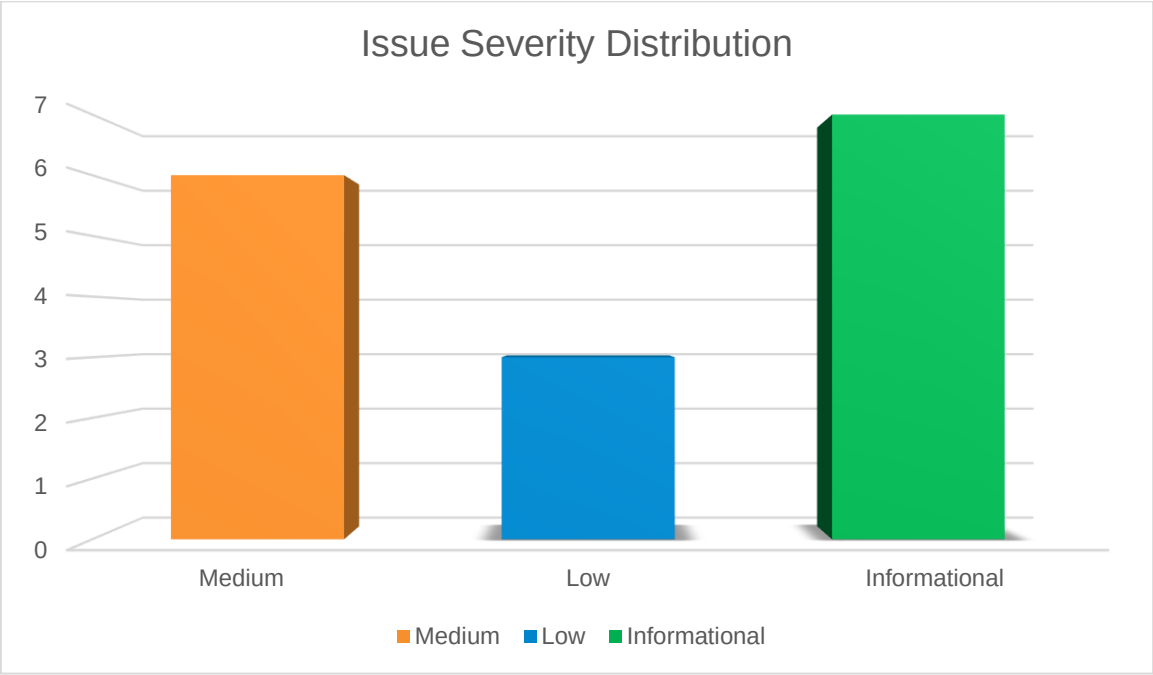

Figure 1 Issue Severity Distribution

1.3 Observations

The Product is one of the most advanced libraries we are aware of implementing ECDSA threshold signing and offers interesting features in terms of flexibility. Comments in the code help pointing out to the scientific literature where specific algorithms are taken from.

Most of the issues we identified concern the way secret values are erased from memory after use ("zeroization"). Zeroization is a tricky subject even in Rust, which is one of the few modern programming languages offering such feature (thorough the `zeroize()` crate). Some examples of misuse and correct implementation can be found for example at https://benma.github.io/2020/10/16/rust-zeroize-move.html .

The only high severity problem we found (KS-INGT-O-07) is about the possibility of a single malicious member in the resharing protocol to lock or delete funds. This dangerous vulnerability, as far as we are aware of, is first reported in this document. However, it is a problem of the protocol itself rather than the implementation, and it is therefore not possible to address it by patching the code we audited. As far as we can tell, the Product follows the protocol correctly, but this vulnerability arises from lack of validation of the underlying security assumptions, namely the need for a trusted broadcast channel. This is not provided by the Product (as the implementation of the network and authentication layer is left to the application) and should be implemented at a higher level in the software stack. It is therefore duty of the application using the Product to mitigate the attack, so we included this issue (and discussion of possible remediation strategies) as an observation rather than a finding.

In general, we found the implementation to be of high standard and we believe that all the identified issues can be easily addressed. Moreover, we did not find evidence of any hidden backdoor or malicious intent in the code.

1.4 Issue Summary List

The following security issues were found:

ID	SEVERITY	FINDING	STATUS
KS-INGT-F-01	Low	Security parameter check not enforced	Remediated
KS-INGT-F-02	Medium	Paillier secret key not zeroized after use	Remediated
KS-INGT-F-03	Medium	Secret share not zeroized upon error	Remediated
KS-INGT-F-04	Medium	Variable shadowing prevents zeroization	Remediated
KS-INGT-F-05	Medium	Paillier secret key not zeroized upon errors	Remediated
KS-INGT-F-06	Medium	Dlog signature nonce not zeroized	Remediated
KS-INGT-F-07	Medium	Intermediate signing values not zeroized	Remediated
KS-INGT-F-08	Low	Incorrect upper bound in range sampling	Remediated
KS-INGT-F-09	Low	Use of `powm()`	Remediated

The following are observations related to general design and improvements:

ID	SEVERITY	FINDING	STATUS
KS-INGT-O-01	Informational	Extra checks in `get_rho_vec()`	Remediated
KS-INGT-O-02	Informational	Proof of Knowledge is not a Signature	Remediated
KS-INGT-O-03	Informational	Commented out code	Remediated
KS-INGT-O-04	Informational	Lack of code coverage on error handling	Acknowledged
KS-INGT-O-05	Informational	Copy-paste error	Remediated
KS-INGT-O-06	Informational	Use of an unmaintained dependency	Remediated
KS-INGT-O-07	Informational	Single party attack on key resharing	Acknowledged

2. 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 2 Methodology Flow

2.1 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

2.2 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

2.3 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

2.4 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 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.

2.5 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.

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. Since this is a library, we ranked the severity of 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.

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.

3. TECHNICAL DETAILS OF SECURITY FINDINGS

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

3.1 Security parameter check not enforced

Finding ID: KS-INGT-F-01

Severity: Low

Status: Remediated

Location: `src/algorithms/dlog_signature.rs @ line 47`

Description and Impact Summary

The function `verify()` checks that a provided proof of discrete logarithm knowledge is valid. It also checks that the validity holds for a given security parameter.

```
    pub fn verify(&self, N: &BigInt, g: &BigInt, V: &BigInt, security_param: u
32) -> bool {
        let x = g.powm(&self.y, N) * V.powm(&self.c, N) % N;
        let c = HSha512Trunc256::create_hash(&[N, g, V, &x]);

        c == self.c && self.security_param == security_param
    }
```

However, this last check is useless because it is not enforceable. The value `security_param` is simply provided as input by the (possibly dishonest) proving party, so it could be anything and disconnected from the statement in exam.

Recommendation

Verifying that a given instance/statement respects a certain security parameter is highly non-trivial. To be strict, one should re-compute the security parameter from the public key directly, this possibly considering not only the bitsize, but also verifying that the modulus is of the right form, etc. Depending on the context (including this use case) this might be definitely overkill.

In order to strengthen the robustness of the proof, we recommend either or both of the following two modifications:

1) Compute the value `security_param` from the bitsize of the statement `V`; or
2) Include `security_param` in the Fiat-Shamir hash at lines 33 and 45.

Status Details

In PR #22 the check for `security_param` has been removed, but it has been made implicit by versioning the protocol. More specifically: the subroutines in `dlog_proof.rs` are still called by passing `security_param` as an argument, but the value itself is hardcoded as '128' in the high-level calls from `zkp.rs` and a salt string representing an identifier for the protocol version is included in the Fiat-Shamir hash generation. This is consistent with other default parameters in the code (including the use of curve `secp256k1` as used in Bitcoin) targeting a security level of 128 bits.

3.2 Paillier secret key not zeroized after use

Finding ID: KS-INGT-F-02

Severity: Medium

Status: Remediated

Location: `src/ecdsa/keygen.rs @ line 362`

Description and Impact Summary

During Phase 1 of the MPC key generation, each party fetches a copy of their own Paillier secret key from the trusted vault in order to generate a zero-knowledge proof. However, this copy is not zeroized after the generation of the proof.

```
        let dk = secret_key_loader
            .get_paillier_secret()
            .map_err(|e| KeygenError::ProtocolSetupError(e.0))?;
        if !PaillierKeys::is_valid(&init_keys.paillier_encryption_key, &dk) {
            return Err(KeygenError::ProtocolSetupError(
                "invalid own Paillier key".to_string(),
            ));
        }
        let proof = nizk_rsa::gen_proof(dk);
```

Recommendation

We recommend performing zeroization on the secret key to protect the value from compromise after the key is used.

Status Details

PR #21 has introduced “boxing” for variables containing secret values, so that they are erased before the object gets out of scope.

3.3 Secret share not zeroized upon error

Finding ID: KS-INGT-F-03

Severity: Medium

Status: Remediated

Location: `src/ecdsa/keygen.rs @ line 653`

Description and Impact Summary

At the end of Phase 2 of the MPC key generation, in preparation for the next phase, each party loads a copy of their own secret share from the trusted vault in order to generate the commitment broadcast for Phase 3. It is checked that this loading procedure does not return error, otherwise the state machine transition to an error state and the protocol is aborted. However, a generic error does not automatically imply that the secret share (or part of it) was not fetched from the vault. This could leave sensitive information in memory.

```
        let sk = self.secret_key_loader.get_initial_secret();
        if let Err(e) = &sk {
            errors.push(KeygenError::GeneralError(e.0.clone()));
        }

        if !errors.is_empty() {
            let error_state = ErrorState::new(errors);
            log::error!("Phase2 returns {:?}", error_state);
            return Transition::FinalState(Err(error_state));
        }
```

Recommendation

We recommend to zeroize the secret share copy before proceeding to error management.

Status Details

PR #21 has introduced “boxing” for variables containing secret values, so that they are erased before the object gets out of scope.

3.4 Variable shadowing prevents zeroization

Finding ID: KS-INGT-F-04

Severity: Medium

Status: Remediated

Location: `src/ecdsa/keygen.rs @ line 664`

Description and Impact Summary

During Phase 2 of the MPC key generation, the secret share variable `let sk` is defined and loaded at line 653. At line 664 a new variable `let mut sk` is defined, thereby shadowing the old variable. The zeroization at line 668 only affects the inner variable, but the shadowed one remains in memory, with potential leakage of secrets.

```
        let (vss_scheme, outgoing_shares) = {
            let mut sk = sk.unwrap();
            let vss_sharing =
                VerifiableSS::share(self.params.threshold, self.params.share_c
ount, &sk);
            sk.zeroize();
            vss_sharing
        };
```

Recommendation

We recommend avoiding shadowing, also for a matter of cleanliness of the code, and add a zeroization for the outer variable.

Status Details

PR #21 has introduced “boxing” for variables containing secret values, so that they are erased before the object gets out of scope. Moreover, the variable name has been changed for readability.

3.5 Paillier secret key not zeroized upon errors

Finding ID: KS-INGT-F-05

Severity: Medium

Status: Remediated

Location: `src/ecdsa/keygen.rs @ line 861`

Description and Impact Summary

During Phase 3 of the MPC key generation, the secret Paillier key is loaded, and if an error occurs the protocol aborts. However, the key is not zeroized before exiting. This could leave sensitive traces in memory.

```
        if !errors.is_empty() {
            log::error!("Phase3 returns errors {:?}", errors);
            return Transition::FinalState(Err(ErrorState::new(errors)));
        }
        let mut dk = dk.expect("invalid paillier decryption key");
```

Recommendation

We recommend zeroizing the secret key before exit.

Status Details

PR #21 has introduced “boxing” for variables containing secret values, so that they are erased before the object gets out of scope.

3.6 Dlog signature nonce not zeroized

Finding ID: KS-INGT-F-06

Severity: Medium

Status: Remediated

Location: `src/algorithms/dlog_signature.rs @ line 35`

Description and Impact Summary

During generation of the signature (proof of discrete logarithm knowledge), the nonce value `r` is not zeroized after use. Notice that schemes such as ECDSA are extremely sensitive to leakage of even just fractions of the bits of `r`.

```
        let y = r - c.borrow() * s;
        Self {
            security_param,
            y,
            c,
        }
```

Recommendation

We recommend zeroizing the nonce after use.

Status Details

Fixed in PR #19.

3.7 Intermediate signing values not zeroized

Finding ID: KS-INGT-F-07

Severity: Medium

Status: Remediated

Location: `src/ecdsa/signature.rs @ line 1234`

Description and Impact Summary

During Phase 4 of signing, the intermediate temporary secret values `k_i` and `gamma_i` are not zeroized before proceeding to Phase 5.

```
            let g_gamma_sum = responses
                .iter()
                .fold(g_gamma_i, |acc, msg| acc + msg.1.g_gamma_i);

            let R = g_gamma_sum * self.delta_inv;
            let local_sig =
                LocalSignature::new(&self.params.message_hash, &R, &self.k_i,
&self.sigma_i);
            let (p5_commit, p5_decommit) = local_sig.phase5b_proof();

            Transition::NewState(Box::new(Phase5ab {
```

Recommendation

We recommend zeroizing these intermediate values as soon as they are not required anymore.

Status Details

PR #21 has introduced “boxing” for variables containing secret values, so that they are erased before the object gets out of scope.

3.8 Incorrect upper bound in range sampling

Finding ID: KS-INGT-F-08

Severity: Low

Status: Remediated

Location: various, for example: `src/algorithms/dlog_signature.rs @ line 30`

Description and Impact Summary

The function `sample_below`() samples at random an integer strictly below a given bound. However, throughout the code it is often called as if the bound is included. This slightly reduces the entropy of the sample. For example, in the code below it is not necessary to subtract `BigInt::one()` from the bound.

```
        let R = BigInt::from(2).pow(log_r) - BigInt::one();
        let r = BigInt::sample_below(&R);
```

Recommendation

We recommend checking carefully the upper bound.

Status Details

Fixed in PR #17.

3.9 Use of `powm()`

Finding ID: KS-INGT-F-09

Severity: Low

Status: Remediated

Location: various

Description and Impact Summary

The function `powm`() is used extensively throughout the codebase. This function computes a modular exponentiation not in constant time, which might leak information about the arguments.

Recommendation

We recommend using `powm_sec()` instead, whenever there is secret parameters involved in the call.

Status Details

Fixed in PR #16.

4. 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.

4.1 Extra checks in `get_rho_vec()`

Observation ID: KS-INGT-O-01

Location: `src/algorithms/nizk_rsa.rs @ line 95`

Description and Impact Summary

The function here follows the pseudocode of https://eprint.iacr.org/2018/057.pdf , section C.4. However, two additional checks for each vector element are introduced in the code:

- The check that every *rho* is nonzero; and
- The check that *GCD(rho,n)=1*.

Recommendation

We do not see a potential vulnerability here, but we recommend anyway documenting this choice.

Notes

This deviation was initially an explicit design choice from the Client. The original algorithm actually requires that all the *rho* vector components lie in Z_n^* rather than just Z_n, where *n* is a product of two safe primes *p* and *q*. In order to ensure this, the two additional checks are required. However, from a practical perspective they can be removed because:

1) The probability of picking a candidate in Z_n but not in Z_n^* is negligible;
2) Even if this happens, that's not a security risk, it will just make an iteration of the algorithm fail and cause a false negative that will require sampling new components;
3) The performance loss of wasting some cycles by (rarely) sampling new *rho* components is nothing compared to the performance gain of (always) skipping an expensive GCD check.

For these reasons, the redundant checks have been removed with PR #23.

4.2 Proof of Knowledge is not a Signature

Observation ID: KS-INGT-O-02

Location: `src/algorithms/dlog_signature.rs @ line 16`

Description and Impact Summary

The function here is called “signature”. This can be misleading, as there is no message to sign. What the function actually does is computing a “proof of knowledge of discrete logarithm” for the DSA signature scheme (so, basically signing the public key itself).

```
/// Signature scheme for DL proof in a composite group with unknown modulo
///
/// "Composite discrete logarithm and secure authentication" , D. Pointcheval
, pp 3.2
#[allow(clippy::many_single_char_names)]
impl DlogSignature {
```

Recommendation

Just in order to improve readability, we suggest changing the function’s name.

Notes

This has been fixed (both at the code and filename level) in PR #18.

4.3 Commented out code

Observation ID: KS-INGT-O-03

Location: `src/ecdsa/resharing.rs @ line 1366`

Description and Impact Summary

A line of code has been left commented out, its role unclear.

```
        //assert!(shares.len() >= self.reconstruct_limit());
```

Recommendation

We recommend checking the need for the additional check and removing the code altogether if not necessary.

Notes

Commented out code has been removed in PR #18.

4.4 Lack of code coverage on error handling

Observation ID: KS-INGT-O-04

Location: various

Description and Impact Summary

There is no test code coverage on code that handles errors.

Recommendation

Tests that exclusively take a positive program-flow into account often lead to a false sense of security. Negative test cases are a good support for assessing if possible errors are handled correctly, even if the implemented error handling is thorough and robust.

Notes

Expanding test case coverage is planned in the next major release of the Product.

4.5 Copy-paste error

Observation ID: KS-INGT-O-05

Location: `src/ecdsa/resharing.rs @ line 494`

Description and Impact Summary

There is a mismatch for the reference to the new VS old committee between the code and the comment and error message.

```
            // check if new committee has duplicates
            let old_parties_as_set = BTreeSet::from_iter(old_committee.iter().
cloned());
            if old_parties_as_set.len() != old_committee.len() {
                return Err(ResharingError::ProtocolSetupError(
                    "duplicate entries in new committee's list".to_string(),
                ));
            }
```

Recommendation

Fix the typo.

Notes

This has been fixed in PR #18.

4.6 Use of an unmaintained dependency

Observation ID: KS-INGT-O-06

Location: `Cargo.lock @ line 322`

`Cargo.toml @ line 24`

Description and Impact Summary

The `failure` dependency is unmaintained and has a Rustsec advisory recommending alternatives: https://rustsec.org/advisories/RUSTSEC-2020-0036.html

Recommendation

Do not use unmaintained dependencies and use `cargo-audit` to monitor for issues and Rustsec advisory notices in the codebase: https://github.com/RustSec/cargo-audit

Notes

This has been fixed in PR #15.

4.7 Single party attack on key resharing

Finding ID: KS-INGT-O-07

Location: `src/ecdsa/resharing.rs @ line 917`

Description and Impact Summary

The key resharing protocol is a delicate procedure that has been covered by many recent attacks. One of these attacks, the "forget-and-forgive", is presented in the public report "Attacking Threshold Wallets" by J.P. Aumasson and O. Shlomovits, and is described as:

> **3.2 The Attack**
>
> The attack aims to prevent parties running the reshare protocol from holding valid shares (both from the old and new set), and therefore prevent them from collectively issue transactions, thereby locking the group of parties out of their wallet.

The recommended mitigation is:

> The maintainers who implemented the fix described it as follows: "a final round has been added to the re-sharing protocol where the new committee members send ACK messages to members of both the old and new committees. Each participant must receive ACK messages from n members of the new committee (excluding themselves) before they save any data to disk."

This mitigation is correctly implemented by the Client in the Product:

```
    /// Last phase of the protocol
    ///
    /// * sends `FinalAck` messages to all parties, including members of old a
nd new committees
    /// * collects `FinalAck` from membeers of new committee and exits
```

However, we found the mitigation insufficient.

A new, malicious committee member is able to create a split of the new committee into two sets by simply sending crafted ACK messages in phase 5:

- A set believing the resharing failed.
- A set believing the resharing succeeded.

This has the consequence that this single party can put themself into a position of force that allows them to either cause a complete loss of funds (not enough valid shares having been written to disk after phase 5 to carry out with the old committee, nor with the new committee); or to cause a blackmail situation where her share is mandatory for the new committee to produce valid signatures.

Recommendation

Protecting the integrity of the resharing protocol and ensuring that it is completed successfully is highly non-trivial. The obvious naïve solution of ending the protocol with a final phase (where a threshold signature for a dummy "OK" message is generated and verified) does only

guarantee that a large enough quorum exists somewhere such that a signature is possible, but does not address other important issues:

1. Does the quorum only include non-malicious parties?
2. Do all the parties of the new committee yield valid shares?
3. Can we identify misbehaving parties?

The best solution (which completely mitigates the attack) would be to ensure that the parties have access to a *robust broadcast channel*, which guarantees integrity and availability of all broadcast messages for all parties. This way no malicious party can send different broadcast ACK messages to different parties. Implementing such a broadcast channel however should be provided by the network layer at an application level, and is therefore not enforceable in the Product. We recommend adding a warning in the documentation of the Product.

Given that the recommended fix is not enforceable by the Product, we classify this issue as an "observation" rather than a "finding", despite its severity. We provide below other possible mitigation strategies that *can* be enforced in the Product's code directly instead, but offer lesser guarantees and come with some drawbacks. We recommend adopting the strategies below only if the availability of a robust broadcast channel as described above is not possible.

A possible fix which provides weaker guarantees is as follows: at the end of the resharing protocol, each party acknowledges success (and therefore erases the old share) if and only if *at least* a threshold number of "ACK" messages is received among the members of the new committee. More specifically: if an old committee consisting of n_o parties and threshold t_o (such that t_o+1 parties are required for signing) enters the resharing protocol (removing old members and adding new ones) into a new committee of n_n parties and threshold t_n, then each party of the old committee who is becoming a member of the new committee only does so (and erases the old share) if at least t_n ACKs are received from the other member candidates of the new committee. This ensures that (notwithstanding accidental network errors or other non-malicious errors in the protocol) at least t_n+1 members of the new committee can recover the secret. However, this does not ensure that the other (n_n-t_n-1) members of the new committee have valid shares, nor it ensures that there are enough (t_n+1) non-malicious parties in the new committee. In other words, the new committee must anyway include at least (t_n+1) total honest parties for the funds not to be locked in a blackmail scheme. This is anyway a much better situation than in the original fix proposed in the "Attacking Threshold Wallets" paper, where even just a *single* malicious party can blackmail the whole committee.

We suggest, in addition to the above, the two following countermeasures:

1) Parties should retain a history of the old shares, as a backup precaution; and
2) Never increase the threshold value t unless absolutely necessary (the Product should forbid this unless an explicit warning flag is set by the user).

More effective mitigation probably requires additional research.

Status Details

The Client acknowledges the potential for a threat and agrees that successful mitigation is only possible at the application layer. Therefore, no fix is being implemented in the Product.

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

APPENDIX B: DOCUMENT HISTORY

VERSION	STATUS	DATE	AUTHOR	COMMENTS
0.1	Draft	8 March 2021	Tommaso Gagliardoni	
0.2	Draft	17 March 2021	Tommaso Gagliardoni	Added finding and moved finding into observations
0.3	Draft	25 March 2021	Tommaso Gagliardoni	Final draft with feedback from Client
1.0	Final Version	26 March 2021	Tommaso Gagliardoni	Final Version

REVIEWER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	8 March 2021	0.1	
Nathan Hamiel	Head of Security Research	26 March 2021	1.0	

APPROVER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	8 March 2021	0.1	
Nathan Hamiel	Head of Security Research	26 March 2021	1.0	

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.