Audit of tECDSA

Coinbase

26 February 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_coinbase_tECDSA
Publication Date:	26 February 2021
Confidentiality Level:	Confidential
Document Owner:	Tommaso Gagliardoni
Document Recipient:	Luis Ocegueda
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 Coinbase ("the Client") to conduct an external security assessment in the form of a code audit of the cryptographic library implementing threshold ECDSA (tECDSA) developed by the Client.

The assessment was conducted remotely by Dr. Tommaso Gagliardoni, Cryptography Expert and Dr. Marco Macchetti, Principal Engineer. The audit took place from January 18th, 2021 to February 5th, 2021 and 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 literature.
- 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 a cryptographic library (tECDSA) written in Golang, with a particular attention to safe implementation of hashing, proof verification, and potential for misuse and leakage of secrets.

The target of the audit was the code branch in the private Git repository provided by the Client.

Additional technical documentation and specs of the solution were provided by the Client.

tECDSA implements a multi-round protocol for computing an ECDSA signature where private key material is split between multiple participants. A quorum (threshold) of participants communicates in a secure multi-party computation (SMC) protocol to compute a valid signature over a designated message. This signature can be verified using the traditional, composite, public key and so this protocol is backwards compatible with existing ECDSA verification implementations, such as the Bitcoin blockchain.

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 **2 Medium**, **9 Low**, and **11 Informational** findings.

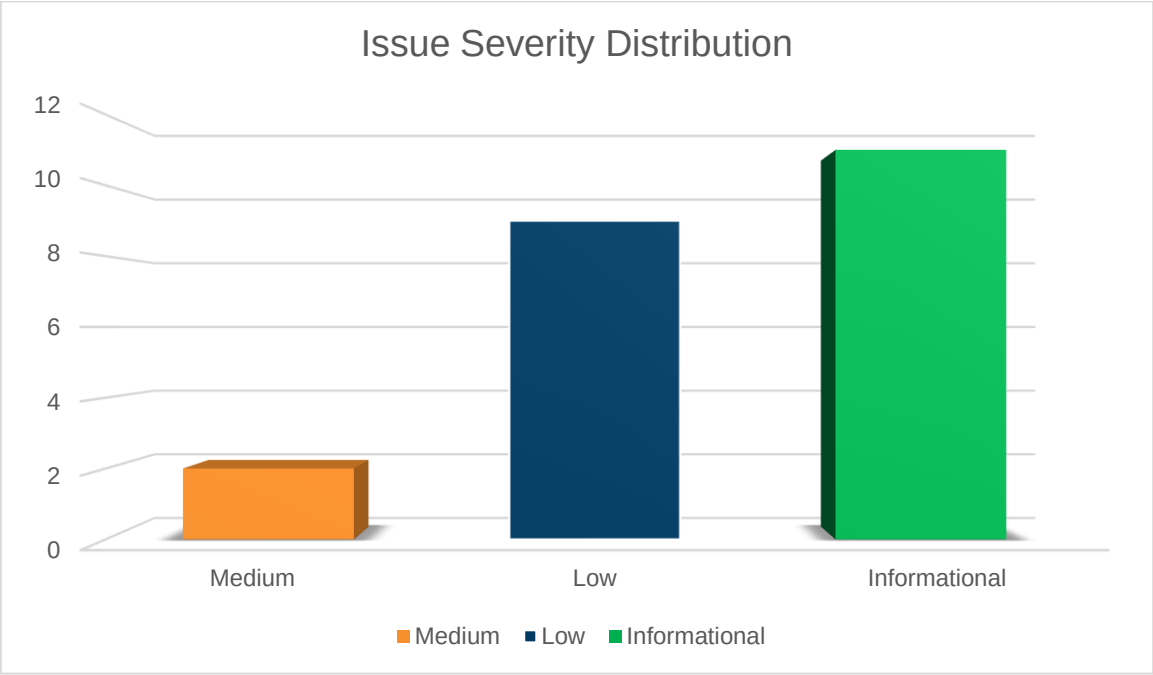

Figure 1 Issue Severity Distribution

1.3 Observations

Threshold ECDSA is a complex cryptographic protocol, so a few challenges in its implementation have to be expected. In general, we found the code quality to be of high standard and we believe that all the identified vulnerabilities can be easily addressed. Moreover, we did not find evidence of any hidden backdoor or malicious intent in the code.

On the choice of the trusted dealer model

In academic literature there are various different proposed schemes that compute threshold ECDSA signatures, some of them very recent, with different tradeoffs in terms of security and speed. The cryptographic scheme implemented by this library follows the recent proposal by Gennaro and Goldfeder (GG20) but using a setup phase where a trusted dealer performs a ceremony where the parameters of the scheme, as well as private Paillier keys for each participant, are generated in a centralized way. This introduces a single point of failure, albeit only in the setup phase, and a trusted authority is a commonly accepted assumption for many custody schemes. Furthermore, the presence of this trusted dealer allows the participants to avoid a decentralized trustless setup phase, which is an extremely complex cryptographic procedure. In addition to introducing a considerable performance overhead, such decentralized phase is so complex that it is very prone to implementation errors, as also shown by recent vulnerability disclosures on similar libraries. Therefore, we consider the choice of using a trusted dealer a reasonable tradeoff in security.

On side-channel attacks

The security model assumes a trusted dealer and honest parties using adequately isolated protocol instances. However, constant-timeness of operations has been considered where possible. Zeroization of values from memory cannot be enforced in Go, so it’s not in scope.

1.4 Issue Summary List

The following security issues were found:

ID	SEVERITY	FINDING	STATUS
KS-CBTSS-F-01	Medium	Paillier keys generation not enforced	Open
KS-CBTSS-F-02	Medium	Use of arbitrary curves not recommended	Open
KS-CBTSS-F-03	Low	Modulo bias in `hashModQ`	Open
KS-CBTSS-F-04	Low	Prime generation of inconsistent size	Open
KS-CBTSS-F-05	Low	Missing loop exit in Round 6	Open
KS-CBTSS-F-06	Low	Salt is zero	Open
KS-CBTSS-F-07	Low	Integer comparison not constant time	Open
KS-CBTSS-F-08	Low	Possible time leak in `Add`	Open
KS-CBTSS-F-09	Low	Round counter prone to mishandling	Open
KS-CBTSS-F-10	Low	Curve equality testing misbehaviour	Open
KS-CBTSS-F-11	Low	Missing modulo reduction	Open

The following are non-security observations related to general design and optimization:

ID	SEVERITY	FINDING	STATUS
KS-CBTSS-O-01	**Informational**	Use of HKDF instead of HMAC	**Informational**
KS-CBTSS-O-02	**Informational**	Use of deprecated proof specs	**Informational**
KS-CBTSS-O-03	**Informational**	Behaviour of `ScalarMult()`	**Informational**
KS-CBTSS-O-04	**Informational**	Misleading comment in `paillier.go`	**Informational**
KS-CBTSS-O-05	**Informational**	Function name in `paillier.go`	**Informational**
KS-CBTSS-O-06	**Informational**	Random group element in `mod.go`	**Informational**
KS-CBTSS-O-07	**Informational**	Variable call in `mta.go`	**Informational**
KS-CBTSS-O-08	**Informational**	Equality check in `mod.go`	**Informational**
KS-CBTSS-O-09	**Informational**	Misleading comment in `mta.go`	**Informational**
KS-CBTSS-O-10	**Informational**	Misleading comment in `round1.go`	**Informational**
KS-CBTSS-O-11	**Informational**	Typo in specs document	**Informational**

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.

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

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 Paillier keys generation not enforced

Finding ID: KS-CBTSS-F-01

Severity: Medium

Status: Open

Location: `crypto/dealer.go`

Description and Impact Summary

The function `NewDealerShares()` implements the DealerKeyGen functionality of the trusted dealer as from Fig. 3 of the specs. However, one critical step is missing: the generation of Paillier keys for each participant.

6. Compute $sk_i, pk_i = \mathsf{PaillierKeyGen}(1^\kappa)$ // generate a Paillier key pair for each player

In the codebase we audited, this step is left to the single participants as a local computation, as in `participant/round_test.go @ line 136`:

```
for i := range sharesMap {
    playerKeysMap[i], err = paillier.NewSecretKey(keyPrimesArray[i-1].p, keyPrimesArray[i-1].q)
    tt.AssertNoError(t, err)
    pubkeys[i] = &playerKeysMap[i].PublicKey
}
```

This behavior has two issues: first of all, it is extremely susceptible to misuse, as now the correct generation of Paillier keys is left to the user. Second, it is crucial that the users' Paillier keys are generated in a trusted way, otherwise this introduces vulnerabilities in the protocol. For example, in the original GG20 protocol every participant has to check that each other participants' Paillier keys are well-formed by using zero-knowledge proofs.

Recommendation

We recommend enforcing Paillier key generation within `NewDealerShares()`.

Status Details

Waiting for feedback from Client.

3.2 Use of arbitrary curves not recommended

Finding ID: KS-CBTSS-F-02

Severity: Medium

Status: Open

Location: `Readme.md @ line 71`

Description and Impact Summary

According to the description of the tECDSA library (for example in the readme) the rationale behind its architecture design is to be used with a flexible choice of curves.

```
tECDSA works with any elliptic curve that supports the Digital Signature Algorithm (DSA). T
he library is designed such that any curve that is compatible with **golang**'s `elliptic.C
urve` interface can be used.
The next choice is to decide how many signers are required to complete a signature and how
many total signers to allow. For our examples going forward, we use **secp256k1** and 3 min
imum signers but 10 total signers.
```

However, in the context of the GG20 threshold ECDSA scheme, this is extremely unsafe unless precautions are taken in carefully tuning the other parameters of the scheme as well. The GG20 scheme (as well as the related specs document) focuses mainly on the choice of curves with a 256-bit field representation (targeting 128 bits of security). As a consequence of this, other parameters (such as 2048-bit Paillier keys and 256-bit hashing) are chosen accordingly. We notice that these are typical parameters deployed in many traditional blockchain solutions (for example, Bitcoin uses SHA-256 and curve secp256k1) so we think that these default choices are reasonable. However, if the tECDSA library is deployed in applications that require different curves (or different hash functions), there is a serious risk of misconfiguration that can lead not merely to a reduced bit-security, but even to exploitation.

As an example, consider what would happen if SHA-256 is replaced by SHA-512 but leaving all other parameters unchanged. In the MtAProveRange1 function (Fig. 7 in the specs) an initiator wants to start the multiplicative-to-additive share protocol using their own 256-bit secret share *a* as an input and sending a “blinded” version of this share to the recipient so that the value of *a* cannot be unmasked.

11. Compute $s_1 = ea + \alpha$ // computed over the integers
12. Compute $s_2 = e\rho + \gamma$ // computed over the integers
13. Set $\pi = [z, e, s, s_1, s_2]$
14. Return π

Now, the share *a* is 256 bits, while the value α is 256*3=768 bits for a curve defined over a field having a 256 bits representation. However, the (public) value *e* is the result of the hash function evaluation, so 512 bits. This means that the product *ea* is 256+512=768 bits, which in turn means that the blinding factor α is not large enough to hide the most significant bit of the secret share. If using a smaller curve field (or a larger share), the end result would be even more catastrophic.

On the other hand, using a larger curve field without tuning other parameters also impacts security negatively: the GG20 protocol relies on computational assumptions that makes the security proof go through only if N (the Paillier modulus) is at least as large as q^8 where q is the order of the curve. This is true for a 256-bit curve since N is 2048 bits, but not for larger q.

Recommendation

We recommend clarifying the intended use of the tECDSA library and evaluate carefully the implications. If the intended use is mainly for Bitcoin transactions, then it is OK to leave the default parameters as they are, but it would be better to explicitly warn in the documentation that these parameters should only be tuned at the user's own risk. If the goal is to provide greater flexibility, then at least minimal sanity checks regarding the bitsize of hash function, curve field, and Paillier modulus should be enforced.

Status Details

Waiting for feedback from Client.

3.3 Modulo bias in `hashModQ`

Finding ID: KS-CBTSS-F-03

Severity: Low

Status: Open

Location: `participant/round6.go @ line 152`

Description and Impact Summary

This function hashes a message to a point on the curve. The distribution should be uniformly random, but the reduction modulo *q* is not corrected for bias. This is even commented in the code but yet unaddressed.

```
// hashModQ takes a message and hashes it to a value less than Q
// TODO: test for bias in the hashed result
func hashModQ(f func() hash.Hash, msg []byte, curve elliptic.Curve) (*big.Int, error) {
    h := f()
    w, err := h.Write(msg)
    if w != len(msg) {
        return nil, fmt.Errorf("bytes written to hash doesn't match expected")
    } else if err != nil {
        return nil, err
    }
    m := new(big.Int).SetBytes(h.Sum(nil))
    return m.Mod(m, curve.Params().N), nil
}
```

Recommendation

The modulo bias should be corrected, either by performing appropriate, deterministic rejection sampling, or by using standard hash-to-curve standards also described in existing IRTF drafts.

Status Details

Waiting for feedback from Client.

3.4 Prime generation of inconsistent size

Finding ID: KS-CBTSS-F-04

Severity: Low

Status: Open

Location: `crypto/primes.go @ line 37`

Description and Impact Summary

The function `GenerateSafePrime` generates a prime *p* of a given bitsize such that (*p*-1)/2 is also a prime. In order to speed up generation, it uses a known technique that tries to "recycle" discarded candidate primes if they are "off by one bit" only.

```
        // https://eprint.iacr.org/2003/186.pdf
        // a safe prime is congruent to 2 mod 3
        // A known exception to this is 7
```

However, this has the disadvantage that the resulting bitsize could be larger (by one bit). Since the Paillier modulus is a product of such two primes, that means that the output could be Paillier key moduli of 2048, 2049, or 2050 bits. Although probably not impacting security directly (but see, e.g., finding KS-CBTSS-F-02) we believe that this might lead to interoperability issues, and possibly exploitation by memory overflow in the case of third parties' implementations of the library. Namely, when receiving a Paillier key by the trusted dealer, if a client application is expecting a modulus of 2048 bits as from specs, the received value might be larger than the memory buffer allocated for.

Recommendation

We recommend enforcing a desired bitsize for the output safe primes, even at the expense of some performance degradation. In any case, according to our preliminary tests, modifying the `GenerateSafePrime` function according to the following pseudocode yields the desired result and at the same time actually boosts performance by a factor of roughly 2.

```
1. let T be 3*5*7*... the product of the first, say, 100 small primes, except 2
   (this should be precomputed and hardcoded as a constant)
2. generate a 1023 bit prime q
3. define p = LeftShift(q,1) + 1
4. if p % T == 0 then go back to 2
5. if ProbablyPrime(p) != true then go back to 2
6. return p
```

Status Details

Waiting for feedback from Client.

3.5 Missing loop exit in Round 6

Finding ID: KS-CBTSS-F-05

Severity: Low

Status: Open

Location: `participant/round6.go @ line 46`

Description and Impact Summary

In the original pseudocode from specs (Fig. 6) there is a loop exit instruction intended to skip the verification of the proof of correctness for the signer's own share, which is missing from the code.

```
for j, value := range in {
        // 3. If i = j, Continue

        // 4. If VerifyPDL(πkCONSIST,g,q,R,pk_j,N,h1,h2,cj,Rj) = False, Abort
```

Recommendation

Although in this implementation this is not a problem (because the array of proofs already excludes the participant's own partial signature) we think it might be better from an in-depth defense perspective to include the check, also considering that the performance impact would probably be negligible.

Status Details

Waiting for feedback from Client.

3.6 Salt is zero

Finding ID: KS-CBTSS-F-06

Severity: Low

Status: Open

Location: `crypto/proof/util.go @ line 91`

Description and Impact Summary

The Fiat-Shamir hashing is done using a zero salt.

```
    salt := make([]byte, 32)
```

Recommendation

Although we do not think in this case it can lead to practical exploitation given the way the concatenated hash is computed, as an in-depth precaution and for good practice we suggest anyway to use a nothing-up-my-sleeves, application-descriptive string in place of a zero salt.

Status Details

Waiting for feedback from Client.

3.7 Integer comparison not constant time

Finding ID: KS-CBTSS-F-07

Severity: Low

Status: Open

Location: `mod/mod.go @ line 70`

Description and Impact Summary

Large integer comparisons are not always performed using the `subtle` (constant time) method. This is also pointed out in this comment.

```
    // TODO: Should this and all big.Int.Cmps be moved to subtle?
    // x ∈ Z_m ⇔ 0 ≤ x < m
    if x.Cmp(Zero) != -1 && x.Cmp(m) == -1 {
        return nil
    }
```

Recommendation

We recommend moving all these comparisons to subtle as suggested in the comment.

Status Details

Client already answered in Slack, waiting for official statement.

3.8 Possible time leak in `Add`

Finding ID: KS-CBTSS-F-08

Severity: Low

Status: Open

Location: `crypto/paillier/paillier.go @ line 186`

Description and Impact Summary

The function `Add` adds homomorphically (multiplies) two Paillier ciphertexts. As a first sanitization step, it is checked that both ciphertexts are in the correct range. However, this is done in a sequential way, which might reveal which one of the two is not in the range (in the case that one of the two is not).

```
// Add combines two Paillier ciphertexts
func (pk *PublicKey) Add(c, d Ciphertext) (Ciphertext, error) {
    if c == nil || d == nil {
        return nil, internal.ErrNilArguments
    }
    // Ensure c,d ∈ Z_N²
    if err := mod.In(c, pk.N2); err != nil {
        return nil, err
    }

    if err := mod.In(d, pk.N2); err != nil {
        return nil, err
    }
```

In general, this behavior should be avoided: by measuring abort time an adversary might infer which of two possibly unknown ciphertext is ill-formed, which could reveal side information from an application perspective.

Recommendation

We recommend checking the return value of both `mod.In` invocations at the same time, in a constant-time way.

Status Details

Waiting for feedback from Client.

3.9 Round counter prone to mishandling

Finding ID: KS-CBTSS-F-09

Severity: Low

Status: Open

Location: `participant/round3.go @ line 93`

`participant/round3.go @ line 107`

Description and Impact Summary

The variable `signer.round` keeps track of the round number we are in. Usually, this variable starts with a value “n”, and is incremented by setting it to the value “n+1” at the end of round number “n”. However, in rounds 3 and 6 this is done by incrementing it as a counter.

```
    signer.round++
```

Recommendation

Although this does not lead to exploitation in this particular case, for code safety and consistency we recommend setting this variable in the specific way used in other rounds.

Status Details

Waiting for feedback from Client.

3.10 Curve equality testing misbehaviour

Finding ID: KS-CBTSS-F-10

Severity: Low

Status: Open

Location: `crypto/ec_point.go @ line 207`

Description and Impact Summary

The function `SameCurve` checks whether two curves are the same by parsing some attributes.

```
    // Use curve order and name
    return a.Curve.Params().P == b.Curve.Params().P &&
        a.Curve.Params().N == b.Curve.Params().N &&
        a.Curve.Params().Name == b.Curve.Params().Name
}
```

However, the structure `CurveParams` contains other fields too:

```
type CurveParams struct {
    P       *big.Int // the order of the underlying field
    N       *big.Int // the order of the base point
    B       *big.Int // the constant of the curve equation
    Gx, Gy  *big.Int // (x,y) of the base point
    BitSize int      // the size of the underlying field
    Name    string   // the canonical name of the curve; added in Go 1.5
}
```

Recommendation

For correctness and robustness, we recommend checking equality among all fields.

Status Details

Waiting for feedback from Client.

3.11 Missing modulo reduction

Finding ID: KS-CBTSS-F-11

Severity: Low

Status: Open

Location: `participant/participant.go @ line 195, 211`

Description and Impact Summary

The routine `convertToAdditive` follows Fig. 4 from the specs, but two intermediate values are not reduced modulo *q* as from the pseudocode. In line 195:

```
            l = l.Mul(x[k].Div(den))
```

and 211:

```
            wI := l.Mul(field.ElementFromBytes(p.Share.Value.Bytes()))
```

This can lead to potential performance loss and, in the second case, also to leakage of some high-order secret bits.

Recommendation

We recommend performing the reduction modulo *q* as from specs for these two values.

Status Details

Waiting for feedback from Client.

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 Use of HKDF instead of HMAC

Observation ID: KS-CBTSS-O-01

Location: `hash/kmac.go`

Description and Impact Summary

The way SHA-256 is used to compute the Fiat-Shamir output of concatenated values is by using a HKDF construction instead of a standard HMAC. Such choice makes sense if we want to have complete freedom over the output size (the concept of Extract-then-Expand). However, in our case, the output will be anyway limited to the hash function's bitsize (256 bit by default), because that's the specified size for the Fiat-Shamir hash. One good reason in support of using HKDF, instead, would be to differentiate this function from the one used to compute commitments (although this is not required by the spec).

Recommendation

The chosen approach is non-standard but probably fine. We would anyway suggest using a HMAC cascade to compute the Fiat-Shamir hash of concatenated values and at the same time avoiding the issue of domain separators. For instance, if computing `Hash(a,b,c)` one could compute a HMAC of `a`, `b`, and `c` separately, then concatenating together the resulting (fixed size) bitstrings, and then applying the HMAC again. This is also the approach recommended in the specs.

Notes

This seems to be an explicit design choice from the Client.

4.2 Use of deprecated proof specs

Observation ID: KS-CBTSS-O-02

Location: `crypto/proof/pdl.go`

`crypto/proof/mta.go`

Description and Impact Summary

The code for the zero-knowledge proofs of knowledge of discrete logarithm (Fig. 11 in the specs) and MtA shares (Fig. 7 and 9) follow an old version of the GG20 specs document, where the Fiat-Shamir hash is computed locally, some of the input components are sent and their correctness verified. The new version uses a different approach: the underlying components are processed directly to obtain a Fiat-Shamir output, and only the correctness of this output is checked. This boosts performances without sacrificing security.

Recommendation

We recommend following the updated version of the specs.

Notes

TBD.

4.3 Behaviour of `ScalarMult()`

Observation ID: KS-CBTSS-O-03

Location: `crypto/ec_point.go @ line 119, 134`

Description and Impact Summary

The functions `ScalarMult` and `NewScalarBaseMult` automatically reduce the input scalar modulo *q* (the curve order, which is denoted by `N` in the code). However, both `ScalarMult` and `ScalarBaseMult` in `elliptic` support scalars of arbitrary size.

Recommendation

We do not see a concrete reason to reduce the input scalar modulo *q* by default. According to the specific program, this can either improve slightly or reduce slightly performance, so this behaviour should be evaluated and, we think, avoided in case of doubt benefit.

Moreover, for consistency, the function `ScalarMult` should actually be renamed `NewScalarMult`.

Notes

This seems to be an explicit design choice from the Client.

4.4 Misleading comment in `paillier.go`

Observation ID: KS-CBTSS-O-04

Location: `crypto/paillier/paillier.go @ line 207`

Description and Impact Summary

The comment about the `Mul` function states:

```
// Mul combines two Paillier ciphertexts
```

However, this is not correct: in Paillier one can only multiply a ciphertext by a *scalar* (which is what the code actually does).

Recommendation

For clarity we recommend fixing the comment.

Notes

TBD

4.5 Function name in `paillier.go`

Observation ID: KS-CBTSS-O-05

Location: `crypto/paillier/paillier.go @ line 223`

Description and Impact Summary

Regarding the function `Encrypt`:

```
// Encrypt produces a ciphertext on input message, public key.
func (pk *PublicKey) Encrypt(msg *big.Int) (Ciphertext, *big.Int, error) {
```

However, this implementation also returns the randomness (Cfr. Fig. 1 in the specs, function PaillierEncryptAndReturnRandomness).

Recommendation

We recommend following the specs for clarity, and rename the function to something similar to “EncryptAndReturnRandomness”.

Notes

TBD

4.6 Random group element in `mod.go`

Observation ID: KS-CBTSS-O-06

Location: `mod/mod.go @ line 144`

Description and Impact Summary

The function `Rand` generates a uniformly random element from a certain range, typically the multiplicative group of integers modulo a prime *p*. This would be any integer between 1 and *p*-1, extreme values included. However, the code also excludes the value 1.

```
    // Select a random element, but not zero or one
    for {
        result, err := crand.Int(crand.Reader, m)
        if err != nil {
            return nil, err
        }

        if result.Cmp(One) == 1 { // result > 1
            return result, nil
        }
```

Recommendation

This choice does not introduce vulnerabilities, and might be recommendable from an in-depth perspective. However, from a cryptographic point of view there is no need of excluding the value 1. We think there is no need to change the code, but the choice should be documented.

Notes

This seems to be an explicit design choice from the Client.

4.7 Variable call in `mta.go`

Observation ID: KS-CBTSS-O-07

Location: `crypto/proof/mta.go @ line 605`

Description and Impact Summary

Fiat-Shamir is called using the original input variables.

```
        challenge, err = fiatShamir(pp.curve.Params().Gx, pp.curve.Params().Gy
, pp.curve.Params().N, pp.pk.N, pp.dealerParams.N, pp.dealerParams.H1, pp.deal
erParams.H2, pp.X.X, pp.X.Y, pp.c1, pp.c2, u.X, u.Y, z, zTick, t, v, w)
```

However these variables are cloned at the beginning and the clone handle is used elsewhere in the code, for example at line 611:

```
        challenge, err = fiatShamir(curveParams.Gx, curveParams.Gy, curveParam
s.N, pp.pk.N, pp.dealerParams.N, pp.dealerParams.H1, pp.dealerParams.H2, pp.c1
, pp.c2, z, zTick, t, v, w)
```

Recommendation

For clarity and ease of reading, we recommend being consistent with the way these variables are called.

Notes

TBD

4.8 Equality check in `mod.go`

Observation ID: KS-CBTSS-O-08

Location: `mod/mod.go`

Description and Impact Summary

The function `ConstantTimeEqByte` compares two integers in constant time. It returns 1 if the two integers have same byte and sign representation, zero otherwise. However, its behavior is not consistent in case one or both values are `nil`.

```
// ConstantTimeEqByte determines if a, b have identical byte serialization
// and signs. It uses the crypto/subtle package to get a constant time comparison
// over byte representations. Return value is a byte which may be
// useful in bitwise operations. Returns 0x1 if the two values have the
// identical sign and byte representation; 0x0 otherwise.
func ConstantTimeEqByte(a, b *big.Int) byte {
    if a == nil && a == b {
        return 0
    }
    if a == nil || b == nil {
        return 1
    }
```

Recommendation

We think this was a typo in the code, in that case one should just swap the 0 and 1 return values on the above lines. Otherwise, if this behaviour is intended, it should be documented.

Notes

TBD

4.9 Misleading comment in `mta.go`

Observation ID: KS-CBTSS-O-09

Location: `crypto/proof/mta.go @ line 555`

Description and Impact Summary

The comment states:

```
        // u = g ^\alpha mod q
        ux, uy := pp.curve.ScalarBaseMult(rp.alpha.Bytes())
```

However, this could be interpreted as reduction of integers modulo *q* (the order of the curve), which is not correct: what is happening here is that *u* is a point of the curve obtained by performing the curve operation α times on the generator *g*.

Recommendation

We recommend keeping the notation consistent with the specs and removing the “mod q” part of the comment for clarity.

Notes

TBD

4.10 Misleading comment in `round1.go`

Observation ID: KS-CBTSS-O-10

Location: `participant/round1.go @ line 66`

Description and Impact Summary

There is a typo in the comment:

```
    // 6. \pi_i^{\Range1} = MtAProveRange1(g,q,pk+i,N~,h_1,h_2,k_i,c_i,r_i)
    pp := proof.Proof1Params{
```

In fact, `pk+1` should rather be `pk_i`.

Recommendation

Fix the comment for clarity.

Notes

TBD

4.11 Typo in specs

Observation ID: KS-CBTSS-O-11

Location: `docs/Coinbase_pseudocode.pdf @ Fig. 5`

Description and Impact Summary

There is a typesetting error in the pseudocode in Fig. 5. Namely, steps 4. and 6. of function SigRound6() seem to be executed outside of the main For loop but this is clearly not the case.

$s_i \leftarrow$ SignRound6$(M, [R_j]_{j \neq i})$

1. Set $V = \bar{R}_i$
2. For $j = [1, \ldots, t+1]$
3. If $i = j$, Continue
4. If VerifyPDL$(\pi_j^{k^{\text{CONSIST}}}, g, q, R, pk_j, \tilde{N}, h_1, h_2, c_j, \bar{R}_j) =$ False, Abort
5. Compute $V = V \cdot \bar{R}_j$ in $\mathcal{G}$
6. If $V \neq g$, Abort

The error does not appear in the code, which implements the algorithm correctly.

Recommendation

We recommend fixing the pseudocode in the draft in order to avoid misinterpretations.

Notes

TBD

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	5 February 2021	Tommaso Gagliardoni	Initial draft
1.0	Final	26 February 2021	Tommaso Gagliardoni	No changes requested, marked as Final

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

APPROVER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	8 February 2021	0.1	
Nathan Hamiel	Head of Security Research	26 February 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.