Audit of bls_sig

Coinbase

13 July 2021
Version: 1.3

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

For public release

DOCUMENT PROPERTIES

Version:	1.3
File Name:	Audit_bls_sig
Publication Date:	13 July 2021
Confidentiality Level:	For public release
Document Owner:	Tommaso Gagliardoni
Document Recipient:	Jeff Barksdale
Document Status:	Approved

TABLE OF CONTENTS

TABLE OF FIGURES

EXECUTIVE SUMMARY

Kudelski Security ("Kudelski"), the cybersecurity division of the Kudelski Group, was engaged by Coinbase Inc. ("the Client") to conduct an external security assessment in the form of a code audit of their BLS signature library written in Golang.

The assessment was conducted remotely by Dr. Tommaso Gagliardoni, Cryptography Expert, and Dr. Marco Macchetti, Principal Engineer. The audit took place from August 17, 2020 to August 28, 2020 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 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 Limitations

The scope of the audit was twofold:

1. Complete code audit of the BLS signing library `bls_sig-master`, with a particular focus on integrity and security of cryptographic protocols and their compliance to existing IRTF draft standards, in particular:
    - https://tools.ietf.org/id/draft-irtf-cfrg-bls-signature-02.txt
    - https://tools.ietf.org/id/draft-irtf-cfrg-hash-to-curve-08.txt
2. For the low-level BLS pairing library `bls12-381-master`, verify that the following commits to the codebase (from the public git repository located at https://github.com/kilic/bls12-381) did not introduce vulnerabilities:
    - `ee2bda4265260796e7d8f0ff1c544d22845397ff`
    - `f7d8771bf0b5dba0d57bc94b8211cd23f0ce2ef7`
    - `4eac1ace18af102ea34135696a7d53a44a0ddd4a`
    - `89cca8641e3f543d582d9e0a15e068bc39aa47f9`

1.2 Engagement Analysis

The engagement consisted of a ramp-up phase where the necessary documentation about the technological standards and cryptographic know-how 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 High**, **2 Medium**, **9 Low**, and **4 Informational** findings.

Most of these findings and observation are related to deviations to the standard and poor coding, but some of them are related to missing data sanitization and correct implementation of cryptographic subroutines and could lead to exploitation.

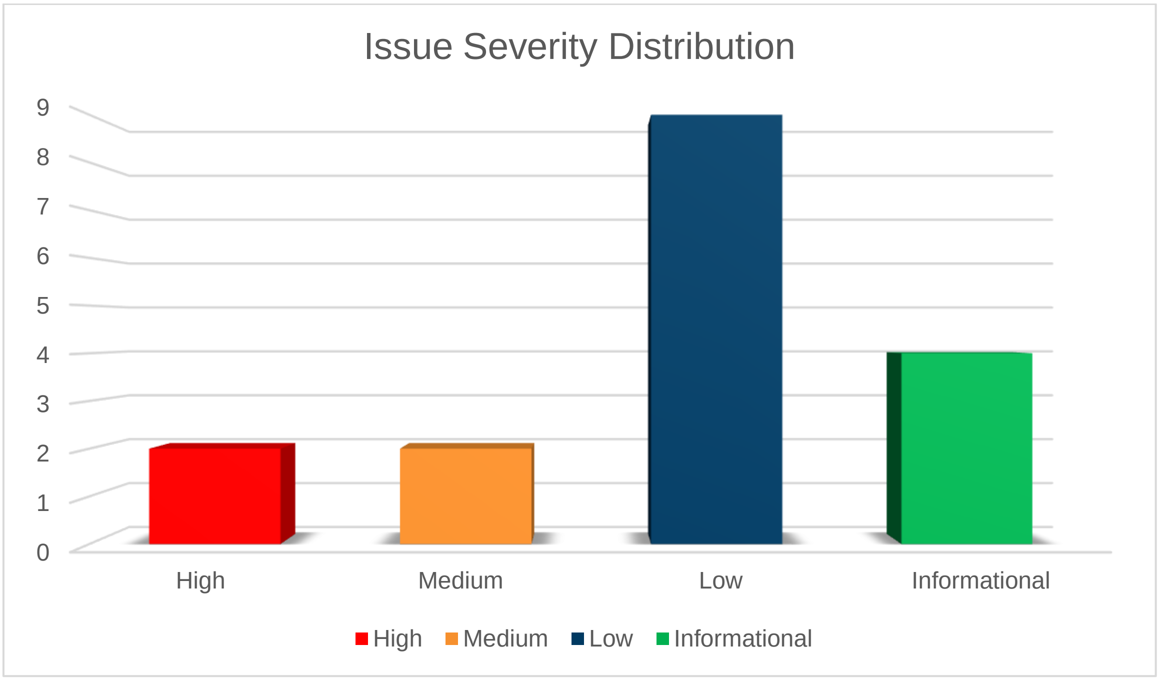

Figure 1 Issue Severity Distribution

1.3 Observations

The library we audited is the Client's implementation of the BLS12-381 pairing-friendly elliptic curve with good properties of signature and public key aggregation. The resulting cryptographic schemes are challenging to implement correctly and the fact that we identified issues in the current, non production-ready version of the code should not be seen as a surprise.

In general, we found the implementation 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.

1.4 Issue Summary List

ID	SEVERITY	FINDING	STATUS
KS-CBBLS-F-01	Low	Hash to curve parameter should be bound to curve in use	Remediated
KS-CBBLS-F-02	Low	Hashing of zero-length message	Remediated
KS-CBBLS-F-03	Low	Implementation of HKDF-SHA-256	Remediated
KS-CBBLS-F-04	Informational	Compliance of all-zero key	Remediated
KS-CBBLS-F-05	Low	Checking against all-zero key	Remediated
KS-CBBLS-F-06	High	Missing checks of subgroup membership	Remediated
KS-CBBLS-F-07	Low	Possible misuse of aggregating zero elements	Remediated
KS-CBBLS-F-08	High	Missing checks in `coreAggregateVerify`	Remediated
KS-CBBLS-F-09	Low	Redundant check in `coreAggregateVerify`	Remediated
KS-CBBLS-F-10	Informational	Implementation practice in `PopVerify`	Remediated
KS-CBBLS-F-11	Informational	Deviations from standard in `FastAggregateVerify`	Remediated
KS-CBBLS-F-12	Low	Return of nil value instead of error	Remediated
KS-CBBLS-F-13	Medium	Wrong variable in `UnmarshalBinary`	Remediated
KS-CBBLS-F-14	Low	Missing check on domain separation	Remediated
KS-CBBLS-F-15	Informational	Borderline case of 255 shares	Remediated
KS-CBBLS-F-16	Low	Missing check of number of total shares	Remediated
KS-CBBLS-F-17	Medium	Missing checks in `combineSigs`	Remediated

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.

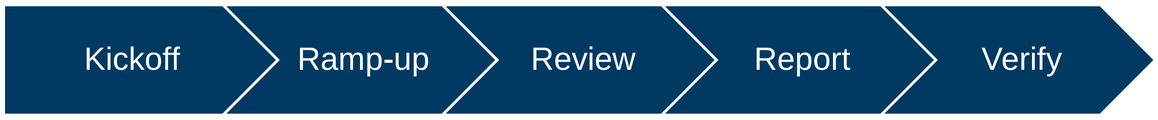

Figure 2 Methodology Flow

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

On August 31st 2020 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.

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 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 Golang 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 Golang

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

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

3.1 Hash to curve parameter should be bound to curve in use

Finding ID: KS-CBBLS-F-01

Severity: **Low**

Status: **Remediated**

Description

In committed change `4eac1ace18af102ea34135696a7d53a44a0ddd4a` of `bls12-381-master` a value is hashed to a curve point as from `draft-irtf-cfrg-hash-to-curve-08`. The expansion factor `lenInBytes` of the string before hashing is computed as follows:

Filename: `hash_to_field.go`

Beginning Line Number: 7

```
func hashToFpXMD(f func() hash.Hash, msg []byte, domain []byte, count int) ([]*fe, error) {
    h := f()
    lenPerElm := h.Size() * 2
    lenInBytes := count * lenPerElm
```

According to `draft-irtf-cfrg-hash-to-curve-08` this value should be 64 for the curve BLS12-381 instead.

Severity and Impact Summary

In this case the value still turns out to be 64 because of the particular hash algorithm used in the library (SHA-256) but could be more or less if the function is called with a different hash algorithm, which cannot be excluded since this is a library. This might lead in one case to excessive modulo bias, in the other case to unnecessary expansion and loss of performance.

Recommendation

The value `lenInBytes` should be 1) bound to the curve type rather than the hash function, for portability, and 2) capped to a max value (e.g. 64) in order to not waste performance overexpanding the message.

Status

This has been fixed by making sure that exactly 64 bytes are used for each elements, which is consistent with the structure of the curve in use.

3.2 Hashing of zero-length message

Finding ID: KS-CBBLS-F-02

Severity: **Low**

Status: **Remediated**

Description

The signing functions do not allow to sign an empty message. However, notice that the zero size for message signature is actually IRTF compliant: it should be possible to sign an empty message, especially since this is supported by `hash_to_curve.go`.

Filename: `usual_bls-sig.go`

Beginning Line Number: 155

```
func (sk SecretKey) createSignature(message []byte, signDst string) (*Signatur
e, error) {
    if len(message) == 0 {
        return nil, fmt.Errorf("message cannot be empty or nil")
    }
```

Severity and Impact Summary

Even if not critical, not allowing the zero message to be signed could cause interoperability issues for certain applications, and in any case it would go against compliance to the IRTF draft.

Recommendation

If possible, allow the empty message to be signed.

Status

This has been fixed as from our recommendations.

3.3 Implementation of HKDF-SHA-256

Finding ID: KS-CBBLS-F-03

Severity: **Low**

Status: **Remediated**

Description

In `lib.go` the `hkdf` function is meant to implement HKDF as specified by RFC 5869, but limited to SHA-256, i.e. HKDF-SHA-256. It is called only once at line 88 to generate a secret key with pre-defined salt and length = 48.

Filename: `lib.go`

Beginning Line Number: 31

```
func hkdf(ikm, salt, info []byte, length int) []byte {
    // HKDF-Extract
    mac := hmac.New(sha256.New, salt)
    mac.Write(ikm)
    prk := mac.Sum(nil)
```

Filename: `lib.go`

Beginning Line Number: 88

```
    okm := hkdf(ikm, []byte(hkdfKeyGenSalt), []byte{0, 48}, 48)
    x := new(big.Int).SetBytes(okm)
    v := new(big.Int).Mod(x, blsEngine.G1.Q())
    return &SecretKey{value: *v}, nil
```

Severity and Impact Summary

The risk is a mismanagement of the salt value in case the `hkdf` function is called somewhere else in future versions of the code without proper parameter handling.

Recommendation

Even if it is not a problem given this only reference in the code, in order to make this function generic we suggest at least two improvements which would be necessary to be compliant with RFC 5869: first, if the provided salt is `nil`, then the default 32-byte string of 0x00 bytes should be used; second, the length must not exceed 255*32.

Also note that HKDF is available in go as `golang.org/x/crypto/hkdf` so one could simply use this implementation.

Status

This has been fixed with the use of the native HKDF implementation.

3.4 Compliance of all-zero key

Finding ID: KS-CBBLS-F-04

Severity: **Informational**

Status: **Remediated**

Description

The UnmarshalBinary function converts bytes to a secret key but also checks against the all 0 key. This goes against `draft-irtf-cfrg-bls-signature-02` where Section 2.3 seems to allow a zero secret key.

Filename: `lib.go`

Beginning Line Number: 99

```
// Deserialize a secret key from raw bytes
// Cannot be zero. Must be 32 bytes and cannot be all zeroes.
// https://tools.ietf.org/html/draft-irtf-cfrg-bls-signature-02#section-2.3
func (sk *SecretKey) UnmarshalBinary(data []byte) error {
    if len(data) != SecretKeySize {
        return fmt.Errorf("secret key must be %d bytes", SecretKeySize)
    }
    nonzero := byte(0)
    for _, b := range data {
        nonzero |= b
    }
    if nonzero == 0 {
        return fmt.Errorf("secret key cannot be zero")
    }
    sk.value.SetBytes(data)
    return nil
}
```

Severity and Impact Summary

This is actually a good behavior in the code and a misinterpretation of `draft-irtf-cfrg-bls-signature-02` as also acknowledged by the authors of the IRTF draft in private communication.

In summary, the issue is that KeyGen as currently specified in the IRTF draft can (with infinitesimal probability) select a 0 secret key, which would result in a public key equal to the identity point. This can be wrongly interpreted as if the draft *requires* implementations to accept the identity point as a valid public key. The authors of the IRTF acknowledged that they will most likely clarify that such identity point should instead be considered an *invalid* public key, and modify hence KeyGen to ensure that it will never return a 0 value as a secret key, as it is currently done in the Client's implementation.

The problem with allowing the identity point as a public key is that every signature under this public key is also the identity point, which means (for example) that a signer can equivocate about the message they signed. This is not true for any other public key (equivocating would require finding a collision in the hash-to-curve function, which is infeasible). Formally this would not break the security of the scheme, but it is an exceptional behavior that might cause problems in protocols relying on informal or intuitive properties of BLS signatures.

Recommendation

We recommend leaving the check as it is now at least until further clarification is provided in a new version of the IRTF draft.

Status

This behavior has been kept as from our recommendations.

3.5 Checking against all-zero key

Finding ID: KS-CBBLS-F-05

Severity: **Low**

Status: **Remediated**

Description

Checking against the all-zero key is done by checking byte-per-byte the key array.

Filename: `lib.go`

Beginning Line Number: 107

```
    for _, b := range data {
        nonzero |= b
    }
    if nonzero == 0 {
        return fmt.Errorf("secret key cannot be zero")
    }
```

Filename: `lib.go`

Beginning Line Number: 132

```
    nonzero := byte(0)
    for _, b := range data {
        nonzero |= b
    }
    if nonzero == 0 {
        return fmt.Errorf("secret key share cannot be zero")
    }
```

Severity and Impact Summary

A byte-scan of a secret key material is always to avoid, if possible, because of the potential leakage in side-channel attacks.

Recommendation

As a possible in-depth mitigation, it would be better to perform the bit collection loop using larger type variables, such as uint32 or uint64.

Status

This has been fixed as from our recommendations, with `subtle.ConstantTimeCompare`.

3.6 Missing checks of subgroup membership

Finding ID: KS-CBBLS-F-06

Severity: **High**

Status: **Remediated**

Description

`verifySignature` is the implementation of the CoreVerify function described in section 2.7 of `draft-irtf-cfrg-bls-signature-02`. However, the at point 3 of the pseudocode the function should check that the signature is lying in the correct subgroup before computing the pairing. This check seems to be missing, both in `usual_bls_sig.go` and in `verifySignatureVt` of `tiny_bls_sig.go`.

Severity and Impact Summary

The hardness of the discrete logarithm problem is not guaranteed outside of the large curve subgroup, therefore it could be possible that a signature outside of that subgroup is actually forged by a malicious adversary.

Recommendation

Always check that signatures lie in the correct subgroup before validating.

Status

This has been fixed as from our recommendations, by introducing proper subgroup membership when required.

3.7 Possible misuse of aggregating zero elements

Finding ID: KS-CBBLS-F-07

Severity: **Low**

Status: **Remediated**

Description

The two functions `aggregatePublicKeys` in `usual_bls_sig` and `aggregatePublicKeysVt` in `tiny_bls_sig` aggregate an array of public keys passed as input into a compact aggregated public key. However, these functions do not seem to check whether at least one element is passed as input: if zero elements can be passed, these functions return the point at infinity, while they should return INVALID (this is stated as precondition check in the IRTF document spec of Aggregate function, section 2.8.)

Severity and Impact Summary

Even if detecting a public key aggregated by zero elements should be easy to check by an honest party, having the infinity point as a valid public key is not compliant.

Recommendation

Enforce a minimum of one key to aggregate, and check that all keys belong to the large underlying subgroup.

Status

This has been fixed as from our recommendations.

3.8 Missing checks in `coreAggregateVerify`

Finding ID: KS-CBBLS-F-08

Severity: **High**

Status: **Remediated**

Description

The two functions `coreAggregateVerify` in `usual_bls_sig` and `coreAggregateVerifyVt` in `tiny_bls_sig` are the implementation of the pseudocode in section 2.9 of the IRTF draft. It seems that the following necessary checks are missing:

1. a check that at least 1 (message, public key) pair is received (precondition);
2. a check that the signature is in the correct subgroup (step 3 in the pseudocode, see also section 5.2 of the draft);
3. validation of all public keys (step 6 in the pseudocode, see also section 5.1).

Moreover, check of unicity of all messages in the draft is not part of coreAggregateVerify (see also KS-CBBLS-F-09 below).

Severity and Impact Summary

As usual, since it is important that all keys and signatures lie in the correct subgroup, the risk is aggregating a set which includes maliciously generated public keys or accepting a maliciously generated aggregated signature.

Recommendation

Always perform the above checks as from draft.

Status

This has been fixed as from our recommendations.

3.9 Redundant check in `coreAggregateVerify`

Finding ID: KS-CBBLS-F-09

Severity: **Informational**

Status: **Remediated**

Description

The goal of the two functions `AggregateVerify` in `usual_bls_sig` and `AggregateVerifyVt` in `tiny_bls_sig` is only to check that all messages are unique before calling their respective core aggregate/verify functions.

Severity and Impact Summary

This seems a bit redundant: for example, if the basic scheme is used this check is already done at line 150 in `usual_bls.go` so effectively it's done twice. If the augmented or the PoP schemes are used, it is not necessary to do it.

Recommendation

These functions should probably be removed and the three schemes AggregateVerify should eventually call only `coreAggregateVerify`.

Status

This has been fixed by calling `coreAggregateVerify` at a low level and `AggregateVerify` as an alias.

3.10 Implementation practice in `PopVerify`

Finding ID: KS-CBBLS-F-10

Severity: **Informational**

Status: **Remediated**

Description

The two functions `Verify` in `usual_bls_sig` and `VerifyVt` in `tiny_bls_sig` are the implementation of the PopVerify function described in section 3.3.3 of the IRTF draft. It seems that the subgroup check for the signature is missing (step 3 of the draft pseudocode).

Recommendation

In this case we do not consider it as a vulnerability, as this check should be performed in the verify function called inside this one. We do however suggest changing the last parameter name from `signDst` to `popDst` as the domain separation tag used is the one for PoP (to be coherent with the `PopProve` function).

Status

This has been fixed as from our recommendations.

3.11 Deviations from standard in `FastAggregateVerify`

Finding ID: KS-CBBLS-F-11

Severity: **Informational**

Status: **Remediated**

Description

We note that `FastAggregateVerify` differs from the description in the IRTF draft (section 3.3.4) in the sense that the implementation also aggregates the signatures.

Status

This has been fixed to be consistent with the IRTF description, now there is only one signature to be verified.

3.12 Return of `nil` value instead of error

Finding ID: KS-CBBLS-F-12

Severity: **Low**

Status: **Remediated**

Description

In the following two locations, when an invalid condition occurs the code returns `nil` instead of raising an error.

Filename: `tiny_bls_sig.go`

Beginning Line Number: 181

```
    p1, err := engine.G1.HashToCurve(sha256.New, message, []byte(signDstVt))
    if err != nil {
        return false, nil
    }
```

Filename: `usual_bls_sig.go`

Beginning Line Number: 273

```
    msg, err := pk.MarshalBinary()
    if err != nil {
        return false, nil
    }
```

Severity and Impact Summary

Proper error handling is important in order to catch unwanted behavior and not create interoperability issues.

Recommendation

Use proper error handling rather than returning a `nil` value.

Status

This has been fixed as from our recommendations.

3.13 Wrong variable in `UnmarshalBinary`

Finding ID: KS-CBBLS-F-13

Severity: Medium

Status: Remediated

Description

In `tiny_bls_sig.go` line 348 the point `p2` is initialized as a new signature but it should rather be a public key.

Filename: `tiny_bls_sig.go`

Beginning Line Number: 344

```
func (pk *MultiPublicKeyVt) UnmarshalBinary(data []byte) error {
    if len(data) != PublicKeyVtSize {
        return fmt.Errorf("multi public key must be %v bytes", PublicKeySize)
    }
    p2 := new(Signature)
    err := p2.UnmarshalBinary(data)
    if err != nil {
        return err
    }
    pk.value = p2.value
    return nil
}
```

Severity and Impact Summary

This is a logical error in the code that can easily cause interoperability issues.

Recommendation

The line should read `p2 := new(PublicKeyVt)` instead of `p2 := new(Signature)`.

Status

This has been fixed as from our recommendations.

3.14 Missing check on domain separation

Finding ID: KS-CBBLS-F-14

Severity: **Low**

Status: **Remediated**

Description

The two functions `NewSigPopWithDST` in `usual_bls.go` and `NewSigPopVtWithDST` in `tiny_bls.go` do not verify that the two domain separation strings are indeed different as from section 3.3.1 of the IRTF draft.

Severity and Impact Summary

Although this does not immediately lead to vulnerabilities, at a minimum it is not spec compliant and might invalidate the cryptographic security proof.

Recommendation

We suggest verifying that the two domain separation strings are indeed different as required by section 3.3.1 of the IRTF draft.

Status

This has been fixed as from our recommendations.

3.15 Borderline case of 255 shares

Finding ID: KS-CBBLS-F-15

Severity: **Informational**

Status: **Remediated**

Description

The function `thresholdizeSecretKey` rejects the case of 255 shares.

Filename: `lib.go`

Beginning Line Number: 159

```
    if total >= 255 || threshold >= 255 {
        return nil, fmt.Errorf("cannot have more than 255 shares")
    }
```

Severity and Impact Summary

This is not a vulnerability per se, but our understanding is that the protocol should support *up to and including* 255 shares, as also confirmed by the returned error message.

Recommendation

Clarify whether 255 shares are supported or not and include the 255 value in case.

Status

This has been fixed so that the code actually supports the borderline case of 255 shares.

3.16 Missing check of number of total shares

Finding ID: KS-CBBLS-F-16

Severity: **Low**

Status: **Remediated**

Description

In `thresholdizeSecretKey` the array of `shares` is initialized based on the output returned by `shamir.NewDealer` without proper sanitization.

Filename: `lib.go`

Beginning Line Number: 167

```
    shareSet, err := shamir.NewDealer(finitefield.New(q)).Split(sk, int(thresh
old), int(total))
    if err != nil {
        return nil, err
    }
    shares := shareSet.Shares
    secrets := make([]*SecretKeyShare, len(shares))
    for i, s := range shares {
        sks := &SecretKeyShare { value: *s }
        secrets[i] = sks
    }
```

Severity and Impact Summary

This might cause issues in case `shamir.NewDealer` returns a number of shares different from total.

Recommendation

As a defense-in-depth, we suggest double-checking that `shares` equals `total` before initializing the `shares` array.

Status

This has been fixed as from our recommendations.

3.17 Missing checks in `combineSigs`

Finding ID: KS-CBBLS-F-17

Severity: **Medium**

Status: **Remediated**

Description

The two functions `combineSigs` in `usual_bls_sig.go` and `combineSigsVt` in `tiny_bls_sig.go` gather partial signatures and yield a complete signature. However, the following checks are missing:

1. That `partials` lies between 2 and 255;
2. That all the input partial signatures lie in the correct subgroup.
3. That the obtained combined signature lies in the correct subgroup.

Severity and Impact Summary

The IRTF draft specifies that the subgroup check must be performed, while the limit of 255 signatures is due to the current implementation. Missing checks 1) and 2) above could result in interoperability issues and the possibility of recombining forged signatures, while 3) is a defense-in-depth against misuse.

Recommendation

We recommend performing the above checks.

Status

This has been fixed as from our recommendations.

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	28 August 2020	Tommaso Gagliardoni	
1.0	Proposal	10 September 2020	Tommaso Gagliardoni	
1.1	Proposal	15 September 2020	Tommaso Gagliardoni	Corrected status label
1.2	Proposal	30 September 2020	Tommaso Gagliardoni	Integrated Coinbase fixes to 3.10 and 3.11
1.3	Final	13 July 2021	Tommaso Gagliardoni	Integrated Coinbase fixes to remaining issues

REVIEWER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	31 August 2020	0.1	
Nathan Hamiel	Head of Security Research	14 September 2020	1.0	
Nathan Hamiel	Head of Security Research	15 September 2020	1.1	
Nathan Hamiel	Head of Security Research	30 September 2020	1.2	
Nathan Hamiel	Head of Security Research	14 July 2021	1.3	

APPROVER	POSITION	DATE	VERSION	COMMENTS
Nathan Hamiel	Head of Security Research	31 August 2020	0.1	
Nathan Hamiel	Head of Security Research	14 September 2020	1.0	
Nathan Hamiel	Head of Security Research	15 September 2020	1.1	
Nathan Hamiel	Head of Security Research	30 September 2020	1.2	
Nathan Hamiel	Head of Security Research	14 July 2021	1.3	

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.