tss-lib Security Audit

Final Report, 2019-10-04

FOR PUBLIC RELEASE

Contents

1 Summary

Binance created a software library implementing threshold ECDSA signatures, and hired Kudelski Security to perform a security assessment of this library.

The repository concerned is:
https://github.com/binance-chain/tss-lib/
we specifically audited commit `31c67c55`.

This document reports the security issues identified and our mitigation recommendations, as well as some observations regarding the code base and general code safety. A "Status" section reports the feedback from Binance's developers, and includes a reference to the patches related to the reported issues. All changes have been reviewed by our team according to our standard audit methodology, and we believe the patches created by Binance adequately address the shortcomings reported.

We report:

- 7 security issues of medium severity
- 3 security issues of low severity
- 20 observations related to general code safety

All of which have already been fixed by the Binance team.

It is important to notice however that the severity of certain issues has been considered higher than we would usually do since this is a library meant to be reused by other developers. None of the issues found in the frame of this audit could be exploited *per se* to completely break the security of the scheme, or recover secret data.

2 Introduction

The tss-lib software implements threshold ECDSA signatures, based on a number of cryptographic components.

Said core components include commitment protocols, Paillier cryptosystem, randomness generation, range proofs, Schnorr proofs, and verifiable secret-sharing.

The actual threshold ECDSA signature scheme includes the three main required algorithms (key generation, signing, verification), as well as higher-level interface to simplify the usage of the threshold scheme.

Our assessment focused mostly on the cryptographic components (most of which are critical to the secure execution of the protocol). We aimed to find:

- Discrepancies between the specified protocol and its expected behavior.
- Insecure cryptographic components or parameters.
- Unsafe software patterns or components.
- Risk of software abuse from malicious input.
- Unsafe handling of errors and edge cases.

3 Methodology

We approached this engagement by performing the following activities:

1. review of the specification and related literature;
2. detailed review of the functional matching between the code and specified intended behavior;
3. assessment of the cryptographic primitives used;
4. software security code review.

This was done in a static way and no dynamic analysis has been performed on the codebase. We discuss our methodology in more detail in the following sections.

3.1 Code Safety

We reviewed the code for software defects, focusing on the handling of potentially untrusted inputs. We looked for:

- general code safety and susceptibility to known vulnerabilities;
- bad 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;
- safety against malformed or malicious input from other network participants.

3.2 Cryptography

We analyzed the cryptographic primitives and subprotocols used, with particular emphasis on randomness and hash generation, signatures, key management, zero-knowledge proofs, and encryption. We checked in particular:

- matching of the proper cryptographic primitives to the desired cryptographic functionality needed;
- security level of cryptographic primitives and of their respective parameters (key lengths, etc.);
- safety of the randomness generation in the general case and in case of failure;
- checking for known vulnerabilities in the primitives used.

3.3 Protocol Specification Matching

We analyzed the original paper, and checked that the code matches the given specification. We checked for things such as:

- proper implementation of the protocol phases;
- proper error handling;
- correct implementation of the zero-knowledge proofs;
- adherence to the protocol logical description.

4 Findings

This section reports security issues found during the audit. Notice that tss-lib being a library rather than an attacker-exposed application, we define severity assuming that public functions and methods can be reached by an attacker, and that we cannot expect one to perform security critical operations at the client level.

The “Status” section includes feedback from Binance developers received after reporting our findings.

KS-BTL-F-01: Signed message not hashed nor sanitized

Severity: Medium

Description

The message is not hashed in `signing/round_1.go` (this is even explicitly written in the comments on lines 21-22), instead it is simply passed as a `big.Int`, without even checking it's in $\mathbb{Z}_q$.

This means that it would be trivial to forge messages that match a signature for a previous message, shall the library be wrongly used without first hashing the messages.

```
21 // missing:
22 // line1: m = H(M) belongs to Zq
23 func (round *round1) Start() *tss.Error {
```

Recommendation

Hash the message before processing it, and make sure the hash is in $\mathbb{Z}_q$ (repeat hashing until it is).

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55. The check that the message is in $\mathbb{Z}_q$ has been added in #62.

Not hashing the message in the library is a design choice made by the developers and will not be changed, in order to be compatible with different sort of blockchain technologies, that require different type of hashes.

This means the clients are responsible for hashing correctly their messages before passing them to the library.

KS-BTL-F-02: Not fully committing ZKProof for Bob in MtAwc

Severity: Medium

Description

In `crypto/mta/proofs.go`, on lines 93 and 161, the following hash is computed:

```
93 eHash = common.SHA512_256i(append(pk.AsInts(), X.X(), X.Y(), c1, c2, z, zPrm, t, v,
   w)...)
```

However, one component in that hash is missing, namely $u = g^{\alpha}$.

This means the ZK proof does not fully commit to the consistency check as it should.

However, it does not allow to create a valid proof for a distinct set of data.

Recommendation

Include the parameter u in the hash computation.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/42 and has been fixed in #43.

KS-BTL-F-03: Not using safe primes

Severity: Medium

Description

The function `GetRandomGeneratorOfTheQuadraticResidue()` used in `GenerateNTildei()` works only if its input is the product of two *safe primes*, that is, primes of the form $p = 2q + 1$ where q is also prime.

However, the primes used in `GenerateNTildei()` are coming from its arguments and in `keygen/round_1.go`, it appears that the primes generated using `rsa.GenerateMultiPrimeKey()` are not safe primes.

```
// Return a random generator of RQn with high probability. THIS METHOD
// ONLY WORKS IF N IS THE PRODUCT OF TWO SAFE PRIMES!
// https://github.com/didiercrunch/paillier/blob/d03e8850a8e4c53d04e8016a2ce8762af3278b71/utils.go#L39
func GetRandomGeneratorOfTheQuadraticResidue(n *big.Int) *big.Int {
	r := GetRandomPositiveRelativelyPrimeInt(n)
	return new(big.Int).Mod(new(big.Int).Mul(r, r), n)
}
```

Recommendation

We recommend to check that the primes used are of the desired form. Also, during the generation of two 1024-bit RSA (safe) primes r and s meant to create a Paillier public key with $N = rs$, it should be checked that $r - s$ is also very large (1020 bits) in order to avoid square-root attacks.

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/55` and has been addressed in #63 and in #68.

KS-BTL-F-04: `MustGetRandomInt()` panics and DoS

Severity: Medium

Description

The `bits` argument in `MustGetRandomInt()` should be checked for acceptable values:

- Upon negative value, `rand.Int()` will panic
- Upon very long value rand.Int will take a long time

See PoC at `https://play.golang.org/p/y-wjDiIK374`.

Recommendation

We recommend to check that `bits` is strictly positive and below a certain bound defined as a constant.

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/28` and has been fixed in cb96dd6.

KS-BTL-F-05: `GetRandomPositiveInt()` infinite loop

Severity: Medium

Description

If the `lessThan` argument is negative in `GetRandomPositiveInt()`, then it will enter an infinite loop, as the condition `try.Cmp(lessThan) < 0` will never be satisfied.

See PoC at `https://play.golang.org/p/sqxDsvzM944`.

Recommendation

We recommend to check that `lessThan` is positive.

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/29` and has been fixed in cb96dd6.

KS-BTL-F-06: `GetRandomPositiveRelativelyPrimeInt()` infinite loop

Severity: Medium

Description

If the `n` argument is negative, then `GetRandomPositiveRelativelyPrimeInt` will loop forever, as the condition `v.Cmp(n) < 0` in `IsNumberInMultiplicativeGroup()` is never satisfied.

See PoC at `https://play.golang.org/p/NeejDhqUW0b`.

Recommendation

We recommend to check that `n` is positive, and also that `IsNumberInMultiplicativeGroup` checks the validity of its argument (being a method visible outside of the package).

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/30` and has been fixed in cb96dd6.

KS-BTL-F-07: Discrepancy in signing Finalize step

Severity: Medium

Description

In the last part of the signature generation protocol, each party is supposed to verify the final signature it computed verifies under the group public key. This is not the case currently in `finalize.go`, as such, any malicious party could make the group reconstruct invalid signatures.

Notice that also the reference paper states that this check is necessary, as the signing procedure can potentially produce invalid signatures.

Recommendation

Perform the final check as described by the reference paper by verifying the reconstructed signature with the group's public key.

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/55` and has been fixed in 9f398c9.

KS-BTL-F-08: `PrepareForSigning()` panics on invalid inputs.

Severity: Low

Description

The input to `PrepareForSigning()` could be so that $pax > \mathrm{len}(ks) \,||\, pax > \mathrm{len}(Xs)$, which leads to a panic when iterating over the array `ks[j]` with $j \in [0, pax[$.

Here is the concerned code snippet, from `signing/prepare.go`:

```
12 func PrepareForSigning(i, pax int, xi *big.Int, ks []*big.Int, bigXs
    ↪  []*crypto.ECPoint) (wi *big.Int, bigWs []*crypto.ECPoint) {
13         modQ := common.ModInt(tss.EC().Params().N)
14 
15         // 2-4.
16         wi = xi
17         for j := 0; j < pax; j++ {
18                 if j == i {
19                         continue
20                 }
21                 kj, ki := ks[j], ks[i] // <-- This could panic if pax > len(ks)
22                 // big.Int Div is calculated as: a/b = a * modInv(b,q)
23                 coef := modQ.Mul(kj, modQ.ModInverse(new(big.Int).Sub(kj, ki)))
                    ↪  // This could panic if kj-ki == 0
```

Also, if `kj == ki`, notice the function will panic as it tries to compute the modular inverse of `new(big.Int).Sub(kj, ki)` on the next line, since 0 is not invertible.

A little bit further on line 36, if `ks[c]==ks[j]`, the `ModInverse` would panic as well.

Notice these should never happen, but this is an exported function, which means it is best to check from a defensive coding point of view, to reduce the attack surface. We report this as a finding since external input would lead to panics.

Recommendation

Add sanity checks on the input.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55 and was fixed in #56.

KS-BTL-F-09: `SHA512_256` interface prone to collisions

Severity: Low

Description

The `SHA512_256()` function takes of list of byte arrays as arguments and creates the data to be hashed by separating these data blocks with a $ character.

Since the byte arrays can also include this character, different sets of inputs can hash to the same value.

For example, if `in` is the single array `[a,$,b,$]`, then the hash result will be the same as when `in` is the two arrays `[a]` and `[b]`.

Recommendation

A non-ambiguous encoding should be used to prevent this, for example by adding an encoding of the block length for each of the blocks processed.

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/34` and has been fixed in #41.

KS-BTL-F-10: Undhandled errors can lead to panic

Severity: Low

Description

On line 38 of `share_protocol.go`, `err` is not checked, but if `EncryptAndReturnRandomness()` returns with the `ErrMessageTooLong` error, then both `cBetaPrm` and `cRand` would be nil, which it turns would make `pkA.HomoAdd(cB, cBetaPrm)` panic on nil pointer dereference.

Moreover, the results of `pkA.EncryptAndReturnRandomness()` and `ProveBob()` are not checked to be `nil` here:

```
38      cBetaPrm, cRand, err := pkA.EncryptAndReturnRandomness(betaPrm)
39      cB, err = pkA.HomoMult(b, cA)
40      if err != nil {
41              return
42      }
43      cB, err = pkA.HomoAdd(cB, cBetaPrm)
44      if err != nil {
45              return
46      }
47      beta = common.ModInt(q).Sub(zero, betaPrm)
48      piB, err = ProveBob(pkA, NTildeA, h1A, h2A, cA, cB, b, betaPrm, cRand)
49      return
```

Notice that if there are good reasons not to check these errors, it is recommended to make them explicit by setting the result to `something, _`.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/39 and has been fixed in #48.

5 Observations

This section reports various observations that are not security issues to be fixed, such as improvement or defense-in-depth suggestions.

KS-BTL-O-01: The `IsOnCurve()` method is not called during unmarshal/new

The structure ECPoint represents a point on a given curve. Currently a method exists such that one can:

```
point := NewECPoint(P256(), x, y)
if !point.IsOnCurve() {
    // report error
}
```

This relies on the programmer remembering to call `IsOnCurve`. We see such checks in the code:

```
tss-lib/crypto/schnorr/schnorr_proof_test.go:20:
assert.True(t, proof.Alpha.IsOnCurve())

tss-lib/crypto/mta/proofs.go:173:
if !gS1.IsOnCurve() || !xEU.IsOnCurve() || !gS1.Equals(xEU) {

tss-lib/crypto/vss/feldman_vss_test.go:36:
assert.True(t, pg.IsOnCurve())

tss-lib/ecdsa/keygen/round_3.go:127:
if !ecdsaPubKey.IsOnCurve() {
```

Recommendation

For defensive coding purposes we recommend not allowing the construction of an invalid curve point. This can be done by changing `NewECPoint` to validate and return an error:

```
func isOnCurve(c Curve, x, y *big.Int) bool {
    if x == nil || y == nil {
        return false
    }
    return c.IsOnCurve(x,y)
}

func (p *ECPoint) IsOnCurve() bool {
    return isOnCurve(p, p.coords[0], p.coords[1])
}

func NewECPoint(curve elliptic.Curve, X, Y *big.Int) *ECPoint, error {
    if !isOnCurve(curve, X, Y) {
        return nil, fmt.Errorf("Your error message here")
    }
    return &ECPoint{curve, [2]*big.Int{X, Y}}, nil
}
```

The unmarshalling function `UnmarshalJSON()` can likewise call `isOnCurve()` to validate the point.

The above is just a suggestion: given that structure fields are only private at a package level in Go, this may or not make sense to implement. In particular, if consuming packages are free to modify the internal fields of the structure at any point, then `IsOnCurve()` may need to be called regularly by the programmers to check the structure has not been altered into an invalid state.

Also notice that invalid curve attacks are not a known problem for ECDSA, unlike for TLS, hence this does not appear to be a security issue.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/46 and has been fixed in #48.

KS-BTL-O-02: Inconsistent arguments validation in range proofs

The verification function checks that arguments are not `nil` in `mta/range_proof.go`:

```
func (pf *RangeProofAlice) Verify(pk *paillier.PublicKey, NTilde, h1, h2, c *big.Int)
↪ bool {
    if pf == nil || pk == nil || NTilde == nil || h1 == nil || h2 == nil || c == nil {
        return false
    }
    ...
```

However, the proof creation does not perform any such checks (and might fail upon `nil` values, when performing arithmetic operations):

```
func ProveRangeAlice(pk *paillier.PublicKey, c, NTilde, h1, h2, m, r *big.Int)
↪ *RangeProofAlice {
    q := tss.EC().Params().N
    q3 := new(big.Int).Mul(q, q)
    q3 = new(big.Int).Mul(q, q3)
    ...
```

Recommendation

We recommend to add `nil` checks for extra safety.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/47 and has been fixed in #50.

KS-BTL-O-03: Accumulators can spare `big.Exp()` computations in `for` loops

In the `feldman_vss.go` file, in the `evaluatePolynomial()` function, we have the following:

```
        q := tss.EC().Params().N
        modQ := common.ModInt(q)
        result = new(big.Int).Set(v[0])
        for i := 1; i <= threshold; i++ {
                ai := v[i]
                aiXi := new(big.Int).Mul(ai, modQ.Exp(id, big.NewInt(int64(i))))
                result = modQ.Add(result, aiXi)
        }
```


But instead of computing `modQ.Exp(id, big.NewInt(int64(i)))` at each iteration, it would be faster to accumulate the value of x^i.

Recommendation

We propose to use an accumulator instead of re-computing the exponentiation in each iteration of the loop:

```
        q := tss.ec().params().n
        modq := common.modint(q)
        result = new(big.int).set(v[0])
        x := big.newint(int64(1)) // we need to have our accumulator outside of the loop
        for i := 1; i <= threshold; i++ {
            ai := v[i]
            x = modq.mul(x, id) // so we have that x = 1*id*id*...*id = id^i
            aixi := new(big.int).mul(ai, x)
            result = modq.add(result, aixi)
        }
```

The same holds for the `verify()` function in `vss/feldman_vss.go#L75`, proposed patch:

```
func (share *Share) Verify(threshold int, vs Vs) bool {
    if share.Threshold != threshold {
        return false
    }
    modN := common.ModInt(tss.EC().Params().N)
    v := vs[0]
    t := new(big.Int).SetInt64(int64(1)) // we need to have our accumulator outside of the loop
    for j := 1; j <= threshold; j++ {
        // t = ki^j
        t = modN.Mul(t, share.ID)  // here we can use just Mul instead of Exp.
        // v = v * vj^t
        vjt := vs[j].SetCurve(tss.EC()).ScalarMult(t)
        v = v.SetCurve(tss.EC()).Add(vjt)
    }
    sigmaGi := crypto.ScalarBaseMult(tss.EC(), share.Share)
    if sigmaGi.Equals(v) {  // could be simplified to "return sigmaGi.Equals(v)"
        return true         // *
    }                       // *
    return false            // *
}
```

Also notice the above function could directly return the value of `sigmaGi.Equals(v)` instead of having a conditional return statement.

The same holds for the `Start()` function in `regroup/round_4_new_step_2.go`.

And the same holds for the `Start()` function in `keygen/round_3.go`.

For instance, using 512 bits values for `q` and `id`, here are some benchmark results that show a 5x speed increase using an accumulator, as well as a significant decrease in the number of allocations required:

```
BenchmarkModMul-4    1000     2188456 ns/op    1135169 B/op    8996 allocs/op
BenchmarkExpMod-4     100    10746285 ns/op    1987063 B/op   13186 allocs/op
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/51 and has been fixed in #53.

KS-BTL-O-04: Storage of unnecessary data in signing round 1

In round 1, in the case where j == i, the `mta.AliceInit()` function is still called, and the message is even stored in `round.temp.signRound1MtAInitMessages[i]`, whereas it is always skipped in the next rounds.

Recommendation

This could be skipped completely in round 1 as well as per the specification.

```go
for j, Pj := range round.Parties().IDs() {
    c, pi, err := mta.AliceInit(round.key.PaillierPks[i], k, round.key.NTildej[j], round.key.H1j[j], round.key.H2j[j])
    if err != nil {
        return round.WrapError(fmt.Errorf("failed to init mta: %v", err))
    }
    r1msg1 := NewSignRound1MtAInitMessage(Pj, round.PartyID(), c, pi)
    if j == i { // this could be at the start of the loop to avoid the AliceInit computations
        round.temp.signRound1MtAInitMessages[j] = &r1msg1 // <-- unnecessary
        continue
    }
    round.temp.signRound1SentMtaInitMessages[j] = &r1msg1
    round.out <- r1msg1
}
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55 and has been fixed in #56.

KS-BTL-O-05: Extra memory allocations in `ModInt` methods

The modular arithmetic defined on top of big.Int does a lot of `new(big.Int)` in order to use big.Int methods, however these incur superfluous memory allocations, which may have an impact on performance when many operations are done.

For example in:

```
func (mi *modInt) Add(x, y *big.Int) *big.Int {
    i := new(big.Int).Add(x, y)
    return new(big.Int).Mod(i, mi.i())
}
```

Here, three `big.Int` structs are allocated whereas one would be sufficient, for example by doing:

```
func (mi *modInt) Add(x, y *big.Int) *big.Int {
    i := new(big.Int).Add(x, y)
    return i.Mod(i, mi.i())
}
```

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/33` and has been fixed in 60f53b9.

KS-BTL-O-06: Extra big int allocation done in proofs

In `mta/proofs.go`, there are extra allocation of `big.Int` that could be spared:

```
// 14.
s1 := new(big.Int).Mul(e, x)
s1 = new(big.Int).Add(s1, alpha)

// 15.
s2 := new(big.Int).Mul(e, rho)
s2 = new(big.Int).Add(s2, rhoPrm)

// 16.
t1 := new(big.Int).Mul(e, y)
t1 = new(big.Int).Add(t1, gamma)

// 17.
t2 := new(big.Int).Mul(e, sigma)
t2 = new(big.Int).Add(t2, tau)
```

which all allocate a temporary `big.Int` to perform the additions.

Recommendation

This could be instead:

```
s1 := new(big.Int).Mul(e, x)
s1.Add(s1, alpha)
...
```

which achieves the same outcome.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55 and has been fixed in #56.

KS-BTL-O-07: `GenerateNTildei()` incomplete arguments check

The function requires a slice with exactly two `*big.Int`'s, however the function will not reject slices with three or more values:

```
func GenerateNTildei(rsaPrimes []*big.Int) (NTildei, h1i, h2i *big.Int, err error) {
    if len(rsaPrimes) < 2 {
        return nil, nil, nil, fmt.Errorf("GenerateNTildei: needs two primes, got %d",
        ↪  len(rsaPrimes))
    }
    NTildei = new(big.Int).Mul(rsaPrimes[0], rsaPrimes[1])
    h1 := random.GetRandomGeneratorOfTheQuadraticResidue(NTildei)
    h2 := random.GetRandomGeneratorOfTheQuadraticResidue(NTildei)
    return NTildei, h1, h2, nil
}
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/35 and has been fixed in #49.

KS-BTL-O-08: Discrepancy in the re-sharing protocol

In `round_1_old_step_1.go`, the "big Xjs" are included in the commitments made using the $\mathrm{Com}()$ function, whereas in the specification itself, only commitments for the v_{ij} from the Feldman's VSS scheme are included.

Similarly, the use of X_j is omitted from the decommitment in $\mathrm{NewCommitteeStep2}$ step 6, but unwrapped in the code and stored in `round.temp.OldBigXj` and round.temp.OldKs in round_4_new_step_2.go. They both do not appear to be reused after that, and take no further part in the specification.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/60 and has been fixed in #60.

KS-BTL-O-09: Unnecessary re-computation of stored value

In the signing round 1, the value round.temp.point is stored, but in round 4 and in round 5, the bigGamma value and the RX, RY values respectively are re-computed from the round.temp.gamma value, instead of being retrieved from the round.temp.point variable.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/61 and has been fixed in #62.

KS-BTL-O-10: Redundant check in GetRandomPrimeInt()

In common/random.go, in the function GetRandomPrimeInt(), there is a conditional check where try == nil can only be true if err != nil, so it is sufficient to check that err != nil.

```
func GetRandomPrimeInt(bits int) *big.Int {
	try, err := rand.Prime(rand.Reader, bits)
	if err != nil ||
		try == nil ||
		try.Cmp(zero) == 0 {
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/31 and has been fixed in a897100.

KS-BTL-O-11: Redundant check in `ECpoint.ValidateBasic()`

An `ECPoint` struct is defined as:

```
type ECPoint struct {
        curve   elliptic.Curve
        coords  [2]*big.Int
}
```

Hence the `coords` array will always be of size 2, as enforced by the Go language. The condition `len(p.coords) == 2` below is therefore always true, thus can be omitted:

```
func (p *ECPoint) ValidateBasic() bool {
        return p != nil && len(p.coords) == 2 && p.coords[0] != nil && p.coords[1] !=
        ↪  nil
}
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/36 and has been fixed in #48.

KS-BTL-O-12: `big.Int` is not constant time

Golang's `big` package provides multiprecision integer arithmetic similar to `libgmp` or `libmpir`. However, `big.Int` does not operate in constant time. Therefore, all `big.Int.Exp()` implementations (https://golang.org/pkg/math/big/#Int.Exp) potentially leak timing information.

Golang's `RSA` package is similarly vulnerable, and in order to mitigate this problem, it applies *blinding* (e.g. in crypto/rsa/rsa.go) in decryption operations. Decrypt functionality in the Paillier cryptosystem may apply similar techniques. An issue (https://github.com/golang/go/issues/20654) is open in the Go repository to support constant-time arithmetic but there is little progress.

Notice this is not necessarily something we recommend trying to fix, but which might be useful depending on your threat model.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/44 and does not necessarily need to be fixed.

KS-BTL-O-13: Non-constant time commitment verification

If an attacker's goal is to find the hash corresponding to some (unknown) secret, they may leverage the variable-time comparator in `Verify()`. The execution time of `Verify()` can also leak the result of the function to an observer. However this is only in theory, and unlikely to represent a security issue in practice.

```
func (cmt *HashCommitDecommit) Verify() bool {
	C, D := cmt.C, cmt.D
	if C == nil || D == nil {
		return false
	}
	hash := common.SHA512_256i(D...)
	if hash.Cmp(C) == 0 {
		return true
	} else {
		return false
	}
}
```

Here `hash.Cmp(C)` calls `big/nat.go`'s `cmp()`, which compares `bigInt`'s value `Word` by `Word`.

Notice this is not necessarily something we recommend trying to fix, but which might be useful depending on your threat model.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/37 and does not necessarily need to be fixed.

KS-BTL-O-14: `RejectionSample` superfluous loop condition

Here the `for` loop depends on the condition `zero.Cmp(q) == -1`, however neither `q` nor `zero` are modified inside the loop, so this check can be moved out of the loop:

```
	qBits := q.BitLen()
	// e = the first |q| bits of e'
	e := firstBitsOf(qBits, eHash)
	// while e is not between 0-q
	for !(e.Cmp(q) == -1 && zero.Cmp(q) == -1) {
		eHash := SHA512_256i(eHash)
		e = firstBitsOf(qBits, eHash)
	}
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/52 and has been fixed in f890595.

KS-BTL-O-15: Unhandled errors

In `ecpoint.go`, there are unhandled errors on `Read`, `Write`, and `(*Int) GobDecode`. This should not be a security concern, but it would be better to handle them. Especially for the `GobDecode` one, as the encoding version can change, causing it to be incompatible.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/38 and has been fixed in #48.

KS-BTL-O-16: Use of multiple channels in concurrency situations

In the ECDSA signing rounds, there are multiple instances of $2n$ unbuffered channels being created for n parties. But one unbuffered channel should be enough to gather all errors from all go routines, especially since we know that we are writing $2n - 2$ things to the channel at most, so it won't block.

In `round_3.go` in particular, two arrays of channels are created, two for each remote party:

```go
// it's concurrency time...
errChs1 := make([]chan *tss.Error, len(round.Parties().IDs()))
for j := range errChs1 {
    if j == i {
        errChs1[j] = nil
        continue
    }
    errChs1[j] = make(chan *tss.Error)
}
errChs2 := make([]chan *tss.Error, len(round.Parties().IDs()))
for j := range errChs2 {
    if j == i {
        errChs2[j] = nil
        continue
    }
    errChs2[j] = make(chan *tss.Error)
}
```

Goroutines are then launched in order to receive messages and report errors from this round, which are passed into each channel. The main thread continues to “drain” error messages (if any) from these channels. Channels are not immediately closed (they do not have to be) and are exposed in a bidirectional fashion to the goroutines.

This works; however, it allocates many individual channels, and ranges iterating over the array of channels will block on any channel that has not had a message sent – that is, if `channel[2]` is encountered first and is blocked but `channel[3]` is ready, the loop will wait until `channel[2]` is ready.

For further reading you may refer to “closing channels in golang”.

Status

This is tracked in `https://github.com/binance-chain/tss-lib/issues/55` and has been fixed in #56.

KS-BTL-O-17: Wrong value gets saved in signing round 2

In the loop at the end of the signing round 2 responsible for sending the messages, it appears only the latest message meant for the last party is saved in the local `temp` variable :

```
// create and send messages
for j, Pj := range round.Parties().IDs() {
    if j == round.PartyID().Index {
        continue
    }
    r2msg := NewSignRound2MtAMidMessage(
        Pj, round.PartyID(), round.temp.c1jis[j], round.temp.pi1jis[j],
         ↪ round.temp.c2jis[j], round.temp.pi2jis[j])
    round.temp.signRound2MtAMidMessages[round.PartyID().Index] = &r2msg
    round.out <- r2msg
}
```

Also notice that `round.PartyID().Index` is actually `i`.

As a consequence, in the `Update()` function, message on index `i` is the one meant to be sent to the latest party. This seems wrong as there is no need to store the latest message sent on index `i` of that table. If the goal is simply to have a valid message on index `i`, so that the `Update()` still verifies, it seems a bit too much to overwrite the value on each loop iteration to just retain the latest one in the end.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55 and has been fixed in #56.

KS-BTL-O-18: Unnecessary initialization in loop

In `signing/prepare.go`, line 21, the value `ki := ks[i]` is initialized in every loop iteration, whereas it could be initialized just once outside of the loop.

```
12 func PrepareForSigning(i, pax int, xi *big.Int, ks []*big.Int, bigXs
    ↪ []*crypto.ECPoint) (wi *big.Int, bigWs []*crypto.ECPoint) {
13         modQ := common.ModInt(tss.EC().Params().N)
14 
15         // 2-4.
16         wi = xi // ki could be initialized here
17         for j := 0; j < pax; j++ {
18                 if j == i {
19                         continue
20                 }
21                 kj, ki := ks[j], ks[i]
                    ↪ // ki doesn't need to be initialized every time
```

The same hold in `prepare.go`, line 36: the value `ks[c]` is accessed at each iteration, it could be copied to a local variable in the outer loop:

```
30 bigWj := bigXs[j]
31 for c := 0; c < pax; c++ {
32     if j == c {
33         continue
34     }
35     // big.Int Div is calculated as: a/b = a * modInv(b,q)
36     iota := modQ.Mul(ks[c], modQ.ModInverse(new(big.Int).Sub(ks[c], ks[j])))
37     bigWj = bigWj.ScalarMult(iota)
38 }
```

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55 and has been fixed in #56.

KS-BTL-O-19: Implementation does not support more parties than the threshold

It seems the current signing algorithm is assuming as many parties as required by the threshold, and if there are more parties, with indexes larger than `Threshold + 1`, they would panic during signing.

See for instance, how `pax` and `len(bigWs)` are not related in `PrepareForSigning()` (lines 28, 29, and 39, in `prepare.go`), which means `bigWs[pax:]` is containing null pointers that would be used by the parties with indexes greater than `pax` in `round_2`'s `Start()`.

Notice that this is consistent with the specification that says *"We assume that [the set of player participating]* $= t + 1$. However this does not seem to be checked within the library; instead it is assumed, and it could panics if someone where to run the protocol with more players than the threshold $+1$.

Status

This is tracked in https://github.com/binance-chain/tss-lib/issues/55 and has been fixed in #56.

Notice that it is considered the client's responsibility to make sure that `len(ks) == len(bigXs) == pax`, however some checks have been added, since this is an exported function.

KS-BTL-O-20: Linters & good practices for Go

We want to stress that the usage of linters in the CD/CI pipeline would help increase code quality, and could have avoided some of the observation we had, such as the presence of deadcode in `signing/round5.go`, where the `NextRound()` function returns always before the `return nil` at its end, on line 67.

Another example that would have been detected by linting: in `keygen/round_1.go`, line 88, the variable `cmt` is named like the package `cmt`, which hurts readability and maintainability.

```
88 cmt := cmt.NewHashCommitment(pGFlat...)
```

Also, linters could have complained regarding the use of a type identifier as a variable identifier in `hash.go`, where `int` is used as an identifier in `for i, int := range in`. Although `int` is not a reserved keyword in the Go language, such a mix between type and variable identifier can be confusing and is not recommended.

It is also probable that KS-BTL-O-15 would have been detected using linters.

Also, the Go good practices require exported functions to be commented as per the GoDoc format. This is especially important in a library and is not the case currently.

Furthermore, the Go good practices also require variables to be named using CamelCase, which is not always enforced currently.

Recommendation

We recommend using `golangci-lint` with the basic linters enabled, and following the Effective Go guidelines.

Status

Some of these issues are tracked in https://github.com/binance-chain/tss-lib/issues/32, https://github.com/binance-chain/tss-lib/issues/40, and have been fixed in #41, #45 .

6 About

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 or https://kudelski-blockchain.com/.

Kudelski Security
Route de Genève, 22-24
1033 Cheseaux-sur-Lausanne
Switzerland