Thorchain TSS Security Audit

Final Report, 2020-06-19

FOR PUBLIC RELEASE

Contents

1 Summary

Thorchain is a decentralized liquidity network to facilitate cross-chain liquidity pools. It is built on Cosmos SDK and Tendermint, and uses threshold signatures (TSS) to authorize transactions.

Lunar 8 Global Services hired Kudelski Security to perform a security assessment of Thorchain's implementation of TSS, based on Binance's TSS Go library `tss-lib`, providing access to source code and documentation.

The repository concerned is:
`https://gitlab.com/thorchain/tss/go-tss`
we specifically audited commit `0401859996f39ba36fd618658ee710c369a72954`.

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 Lunar 8 Global Services'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 usual audit methodology.

We report:

- 1 security issue of medium severity
- 2 security issues of low severity
- 1 observation related to general code safety

The audit was performed jointly by Dr. Tommaso Gagliardoni – Cryptography Expert, and Yolan Romailler – Senior Cryptography Engineer, with support of Dr. Jean-Philippe Aumasson – VP of Technology.

2 Methodology

In this engagement, we performed three main tasks:

1. code review of the core cryptographic components with code safety issues in mind;
2. assessment of the cryptographic primitives used;
3. verification on the patching of known vulnerabilities in `tss-lib`.

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

2.1 Code Safety

We analyzed the provided code, checking for issues related to:

- general code safety and susceptibility to vulnerabilities;
- poor coding practices and unsafe behavior;
- leakage of secrets or other sensitive data through memory mismanagement;
- leakage of secrets of other sensitive data through timing attacks;
- susceptibility to misuse and system errors;
- safety against malformed or malicious input from other network participants;
- error management and logging.

2.2 Cryptography

We analyzed the cryptographic primitives and components, and their correct implementation from `tss-lib`. 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.);
- assessment of proper security definitions and compliance to the use cases;
- checking for known vulnerabilities in the primitives used.

2.3 Patching Status of `tss-lib`

The core cryptographic components of Thorchain are managed by Binance's `tss-lib`. Kudelski Security has previously audited this component, the report can be found at https://github.com/binance-chain/tss-lib/releases/download/v1.0.0/audit-binance-tss-lib-final-20191018.pdf . We double-checked that our mitigation recommendations have been correctly applied, as well as patching of two other important vulnerabilities:

- the first vulnerability was fixed in Binance's `tss-lib` Security Release v1.1.0 (https://github.com/binance-chain/tss-lib/releases/tag/v1.1.0 .) The vulnerability applies to the re-sharing protocol only. It allows for a malicious actor to cause a new committee member to abort the protocol, unable to write a valid share to disk. The other participants would continue as normal and overwrite their share data.
- The second vulnerability was fixed in Binance's `tss-lib` Security Release v1.2.0 (https://github.com/binance-chain/tss-lib/releases/tag/v1.2.0 .) The vulnerability applied when the keygen protocol was compromised. It allowed for a malicious actor to generate bad parameters h_1, h_2 that are shared with other parties during keygen. This allowed such an actor to compromise signing rounds and potentially expose the private data of other parties. Existing save data for signing groups that have already completed keygen should be considered secure if the parties were trusted at the time of keygen. This vulnerability could only have been exploited when a malicious actor with prior knowledge of this vulnerability had participated in keygen.

As Thorchain is based on Binance's `tss-lib` v.1.3.1, it is not affected by any of the above vulnerabilities.

2.4 Notes

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, and in the scope of the agreement between Lunar 8 Global Services and Kudelski Security.

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.

3 Findings

This section reports security issues found during the audit. The "Status" section includes feedback from the developers received after delivering our draft report.

KS-TC-F-02: Insufficient Public-Key Validation

Severity: Medium

Description

In `tss.go` the `GetTssPubKey()` function forms a public key object from an elliptic curve point, checking only that the pointer is not `nil`:

```
func GetTssPubKey(pubKeyPoint *crypto.ECPoint) (string, types.AccAddress, error) {
	if pubKeyPoint == nil {
		return "", types.AccAddress{}, errors.New("invalid points")
	}
	tssPubKey := btcec.PublicKey{
		Curve: btcec.S256(),
		X:     pubKeyPoint.X(),
		Y:     pubKeyPoint.Y(),
	}
```

However, it is not verified that the point is a valid public key: the point should be on the curve, and not the point to infinity. Otherwise, the point decompression will fail to recover the correct point, and signature verification will fail.

Recommendation

Verify the coordinates correspond to a point on secp256k1.

Status

This has been fixed in commit `dc5fe0332a5992579117376834b9500f6d6d7639`.

KS-TC-F-03: Potential String Format Vulnerability

Severity: Low

Description

In `localstate_mgr.go` a `FileStateMgr` creates a path name from a public key as a string and the `fsm.folder` path:

```
func (fsm *FileStateMgr) getFilePathName(pubKey string) string {
	localFileName := fmt.Sprintf("localstate-%s.json", pubKey)
	if len(fsm.folder) > 0 {
		return filepath.Join(fsm.folder, localFileName)
	}
	return localFileName
}
```

However here a `pubKey` could itself contain a path, such as `../../filename`. If an attacker can somehow control `pubKey`, they could therefore make the application attempt to write in any place in the filesystem.

Recommendation

`pubKey` should be validated to be of the format (length, character set) expected for a public key.

Status

This has been fixed in commit `e484e836cfc0444a69061972a517b267d9a0b123`.

KS-TC-F-04: Unbalanced Leader Election

Severity: Low

Description

In `p2p/leader_provider.go` a modulo bias in the leader election occurs if the `numNodes` value is not a power of 2.

```go
func LeaderNode(buf []byte, numNodes int32) (int32, error) {
        h := fnv.New32()
        if _, err := h.Write(buf); err != nil {
                return -1, err
        }
        result := int32(h.Sum32())
        if result < 0 {
                result = result * -1
        }
        return result % numNodes, nil
}
```

The value `result` is reduced modulo `numNodes`, which is introducing the modulo bias on the final output. While not necessarily dangerous, this might cause imbalanced leader elections which could undermine the trust of the users.

Recommendation

We recommend not using a modulo to reduce a random value to a given range. Rejection sampling with bit masking and continuous hashing might be a better solution there.

Status

In the latest version of the code the role of the leader has been removed and replaced by a permissionless join party scheme.

4 Observation

This section reports an observation that is not a security issue, but rather an improvement or defense-in-depth suggestion.

KS-TC-O-01: Possible Large Values Produced by `hashToInt`

In `common/tss_helper.go` the function `hashToInt` maps a given hash value to a big integer in the field of the elliptic curve:

```go
func hashToInt(hash []byte, c elliptic.Curve) *big.Int {
	orderBits := c.Params().N.BitLen()
	orderBytes := (orderBits + 7) / 8
	if len(hash) > orderBytes {
		hash = hash[:orderBytes]
	}

	ret := new(big.Int).SetBytes(hash)
	excess := len(hash)*8 - orderBits
	if excess > 0 {
		ret.Rsh(ret, uint(excess))
	}
	return ret
}
```

However, this is only guaranteed to produce an integer of the same bitsize as the underlying curve field, not necessarily an integer smaller than that. This is not a security problem per se, but depending on the use of the output of the function a modulo reduction might be required (in the case where this does not introduce a modulo bias.)

5 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