Auditing Report

Hardening Blockchain Security with Formal Methods

FOR

ChainSafe

BTC Relayer

Veridise Inc.
July 9, 2024

► **Prepared For:**

ChainSafe
https://chainsafe.io/

► **Prepared By:**

Benjamin Mariano
Jacob Van Geffen
Vijay Singh

► **Contact Us:** contact@veridise.com

► **Version History:**

Jul. 9, 2024 V2
Jun. 19, 2024 V1

Contents

1 Executive Summary

From Jun. 10, 2024 to Jun. 18, 2024, ChainSafe engaged Veridise to review the security of their BTC Relayer. The review covered the implementation of a BTC relayer as well as an implementation of a FROST-based threshold-signature scheme. Veridise conducted the assessment over 3 person-weeks, with 3 engineers reviewing code over 1 week on commit `0d421d1`. The auditing strategy involved extensive manual review by Veridise engineers of the source code and related materials.

Project summary. The security assessment covered two additions to the existing `sygma-protocol` codebase. The first added support for BTC relaying logic and included listening logic for reading events off-chain and execution logic for completing cross-chain transactions. The second added support for a FROST-based threshold signature scheme (TSS) used for reaching consensus among relayers on cross-chain messages.

Code assessment. The BTC Relayer developers provided the source code of the BTC Relayer implementation for review. The code was largely original code written by the BTC Relayer developers. However, the TSS implementation depended heavily on the FROST implementation from the `multi-party-sig` library from Taurus SA. The BTC Relayer code contains some documentation in the form of READMEs and documentation comments on functions and storage variables. To facilitate the Veridise auditors' understanding of the code, the BTC Relayer developers additionally provided some high-level documentation explaining the broader protocol.

The source code contained a test suite, which the Veridise auditors noted covered some but not all critical paths in the code.

Summary of issues detected. The audit uncovered 14 issues, 1 of which is assessed to be of high severity by the Veridise auditors. Specifically, the highest severity issue involved potential dropped funds due to mismatched messages across relayers (V-BTC-VUL-001). The Veridise auditors also identified 4 medium-severity issues, including a denial of service attack (V-BTC-VUL-004), possible lost messages (V-BTC-VUL-003), and insufficient transaction fees (V-BTC-VUL-003) as well as 4 low-severity issues, 3 warnings, and 2 informational findings.

Disclaimer. We hope that this report is informative but provide no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the system is secure in all dimensions. In no event shall Veridise or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

2 Project Dashboard

Table 2.1: Application Summary.

Name	Version	Type	Platform
BTC Relayer	`0d421d1`	Go	Sygma Protocol

Table 2.2: Engagement Summary.

Dates	Method	Consultants Engaged	Level of Effort
Jun. 10 - Jun. 18, 2024	Manual	3	3 person-weeks

Table 2.3: Vulnerability Summary.

Name	Number	Acknowledged	Fixed
Critical-Severity Issues	0	0	0
High-Severity Issues	1	1	1
Medium-Severity Issues	4	4	3
Low-Severity Issues	4	4	4
Warning-Severity Issues	3	2	1
Informational-Severity Issues	2	2	2
TOTAL	14	13	11

Table 2.4: Category Breakdown.

Name	Number
Logic Error	9
Denial of Service	1
Race Condition	1
Unreliable Dependencies	1
Usability Issue	1
Maintainability	1

3 Audit Goals and Scope

3.1 Audit Goals

The engagement was scoped to provide a security assessment of BTC Relayer's implementation of the BTC listener and executor, as well as their FROST TSS implementation, which are in the folders `chains/btc` and `tss/frost` respectively. In our audit, we sought to answer questions such as:

- Can funds be double spent?
- Can a small number of malicious relayers block execution of valid transactions?
- Can user funds be lost (either maliciously or accidentally)?
- Can malicious users block execution of valid transactions?
- Can the price of cross-chain transactions be manipulated?
- Can funds be spent without validation from the expected number of relayers?

3.2 Audit Methodology & Scope

Audit Methodology. To address the questions above, our audit involved extensive manual review by our security experts.

Scope. The scope of this audit is limited to the code in the `chains/btc` and `tss/frost` folders, excluding mock implementations.

Methodology. Veridise auditors reviewed documentation for related protocols and audits, inspected the provided tests, and read the BTC Relayer documentation. They then began a manual review of the code. During the audit, the Veridise auditors met with the BTC Relayer developers to ask questions about the code and discuss issues. Veridise auditors also communicated asynchronously with developers to clarify questions and discuss issues over Telegram.

3.3 Classification of Vulnerabilities

When Veridise auditors discover a possible security vulnerability, they must estimate its severity by weighing its potential impact against the likelihood that a problem will arise. Table 3.1 shows how our auditors weigh this information to estimate the severity of a given issue.

Table 3.1: Severity Breakdown.

	Somewhat Bad	Bad	Very Bad	Protocol Breaking
Not Likely	Info	Warning	Low	Medium
Likely	Warning	Low	Medium	High
Very Likely	Low	Medium	High	Critical

In this case, we judge the likelihood of a vulnerability as follows in Table 3.2:

Table 3.2: Likelihood Breakdown

Not Likely	A small set of users must make a specific mistake
Likely	Requires a complex series of steps by almost any user(s) - OR - Requires a small set of users to perform an action
Very Likely	Can be easily performed by almost anyone

In addition, we judge the impact of a vulnerability as follows in Table 3.3:

Table 3.3: Impact Breakdown

Somewhat Bad	Inconveniences a small number of users and can be fixed by the user
Bad	Affects a large number of people and can be fixed by the user - OR - Affects a very small number of people and requires aid to fix
Very Bad	Affects a large number of people and requires aid to fix - OR - Disrupts the intended behavior of the protocol for a small group of users through no fault of their own
Protocol Breaking	Disrupts the intended behavior of the protocol for a large group of users through no fault of their own

4 Vulnerability Report

In this section, we describe the vulnerabilities found during our audit. For each issue found, we log the type of the issue, its severity, location in the code base, and its current status (i.e., acknowledged, fixed, etc.). Table 4.1 summarizes the issues discovered:

Table 4.1: Summary of Discovered Vulnerabilities.

ID	Description	Severity	Status
V-BTC-VUL-001	Dropped mismatched messages	High	Fixed
V-BTC-VUL-002	Mismatched resources	Medium	Fixed
V-BTC-VUL-003	Transactions blocked for low fees	Medium	Fixed
V-BTC-VUL-004	Unexecuted proposals may be lost	Medium	Fixed
V-BTC-VUL-005	Blocking transactions via small deposits	Medium	Acknowledged
V-BTC-VUL-006	Possible transaction nonce collision	Low	Fixed
V-BTC-VUL-007	Possible race conditions	Low	Fixed
V-BTC-VUL-008	Bad state on failure	Low	Fixed
V-BTC-VUL-009	Incorrectly handling empty msg.To	Low	Fixed
V-BTC-VUL-010	Unreliable/unstable dependencies	Warning	Acknowledged
V-BTC-VUL-011	Unclear BTC transaction assumptions	Warning	Fixed
V-BTC-VUL-012	Inefficiency on incomplete peers	Warning	Intended Behavior
V-BTC-VUL-013	Typos and incorrect comments	Info	Fixed
V-BTC-VUL-014	Bad non-zero check	Info	Fixed

4.1 Detailed Description of Issues

4.1.1 V-BTC-VUL-001: Dropped mismatched messages

Severity	High	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description.		
Confirmed Fix At	N/A		

In order to function properly, the protocol relies on the relayers communicating messages to one another that they have signed. Per the FROST algorithm, when a given message has reached enough signatures, it can be "executed" on the target chain. Proper functioning of this algorithm in the context of this application assumes that the same bridge transaction (captured as "proposals" in the implementation) is translated into the same message by each relayer independently. There are (at least) two cases that might cause this assumption to fail.

First, each cross-chain message is assigned a nonce which is used as a unique identifier for that message. For events read from Bitcoin, the nonce is computed as a function of the current block number and "event number", which corresponds to the index of the on-chain event as it was queried from the Bitcoin node being read from. If this index varies across chain, the "event number" for the same transaction would vary across relayers, as shown in the snippet below

```
1  func (eh *FungibleTransferEventHandler) CalculateNonce(blockNumber *big.Int,
       evtNumber int) (uint64, error) {
2      // Convert blockNumber to string
3      blockNumberStr := blockNumber.String()
4
5      // Convert evtNumber to *big.Int
6      evtNumberBigInt := big.NewInt(int64(evtNumber))
7
8      // Convert evtNumberBigInt to string
9      evtNumberStr := evtNumberBigInt.String()
10
11     // Concatenate blockNumberStr and evtNumberStr
12     concatenatedStr := blockNumberStr + evtNumberStr
13
14     // Parse the concatenated string to uint64
15     result, err := strconv.ParseUint(concatenatedStr, 10, 64)
16     if err != nil {
17         return 0, err
18     }
19
20     return result, nil
21 }
```

Snippet 4.1: Implementation of `CalculateNonce()`

Second, when executing a set of proposals on Bitcoin, the proposals are converted into a Bitcoin transaction which gathers UTXOs from the specified resource address and uses them as the inputs for the transaction to handle the proposals received from other chains. These UTXOs are gathered by querying the mempool API and taking the first UTXOs which cover the needed

output amount. Because relayers may be asynchronous, if changes are made to the UTXOs between when relayers query the mempool API, the on-chain transaction (and thus the encoded message) could vary between relayers. The code used for fetching UTXOs is shown below.

```
func (e *Executor) inputs(tx *wire.MsgTx, address btcutil.Address, outputAmount int64
    ) (int64, []mempool.Utxo, error) {
    usedUtxos := make([]mempool.Utxo, 0)
    inputAmount := int64(0)
    utxos, err := e.mempool.Utxos(address.String())
    if err != nil {
        return 0, nil, err
    }
    for _, utxo := range utxos {
        previousTxHash, err := chainhash.NewHashFromStr(utxo.TxID)
        if err != nil {
            return 0, nil, err
        }
        outPoint := wire.NewOutPoint(previousTxHash, utxo.Vout)
        txIn := wire.NewTxIn(outPoint, nil, nil)
        tx.AddTxIn(txIn)

        usedUtxos = append(usedUtxos, utxo)
        inputAmount += int64(utxo.Value)
        if inputAmount > outputAmount {
            break
        }
    }
    return inputAmount, usedUtxos, nil
}
```

Snippet 4.2: Implementation of `inputs()`

Impact If relayers do not agree on the message produced for a given proposal, even a valid proposal with well-behaving relayers will still not be accepted. This could result in dropped transactions and lost funds.

Recommendation Use a scheme which guarantees that the same transaction will always be encoded as the same message.

Developer Response The developers implemented a fix which sorts UTXOs in order of block time. As a result, the same UTXOs should be returned even after new UTXOs are added. The nonce issue is solved in this PR by computing the nonce as a function of the event hash instead of the event number.

4.1.2 V-BTC-VUL-002: Mismatched resources

Severity	Medium	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	`btc/executor/executor.go`		
Location(s)	rawTx()		
Confirmed Fix At	N/A		

The function `rawTx()` is used to construct the BTC transaction which will be used to complete the Bitcoin side of a cross-chain transaction.

```
1  func (e *Executor) rawTx(proposals []*proposal.Proposal) (*wire.MsgTx, []mempool.Utxo
       , error) {
2      resourceAddress, ok := e.resourceAddresses[proposals[0].Data.(
       BtcTransferProposalData).ResourceId]
3      if !ok {
4          return nil, nil, fmt.Errorf("no address for resource")
5      }
6
7    ...
8
9      return tx, utxos, err
10 }
```

Snippet 4.3: Snippet from `rawTx()`

As shown, the address for the resource to use when constructing the BTC transaction is retrieved by getting the resource address associated with the first proposal in the set. It is then assumed that all proposals share that same resource address.

Impact If the proposals do not share the same resource, amounts of resources will essentially be exchanged at a 1:1 ratio, which could lead to significant lost funds.

Recommendation Ensure all proposals share the same resource ID.

4.1.3 V-BTC-VUL-003: Transactions blocked for low fees

Severity	Medium	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	`btc/executor/executor.go`		
Location(s)	rawTx()		
Confirmed Fix At	N/A		

When constructing a transaction to execute on the Bitcoin network, input UTXOs are selected which cover at least as much as the output amount. Any additional amounts are used towards fees (up to a given maximum amount that is computed), as is shown in the snippet below.

```
outputAmount, err := e.outputs(tx, proposals)
...
inputAmount, utxos, err := e.inputs(tx, resourceAddress, outputAmount)
...
if inputAmount < outputAmount {
    return nil, nil, fmt.Errorf("utxo input amount %d less than output amount %d",
    inputAmount, outputAmount)
}
fee, err := e.fee(int64(len(utxos)), int64(len(proposals))+1)
...
returnAmount := int64(inputAmount) - fee - outputAmount
if returnAmount > 0 {
    // return extra funds
    returnScript, err := txscript.PayToAddrScript(resourceAddress)
    if err != nil {
        return nil, nil, err
    }
    txOut := wire.NewTxOut(returnAmount, returnScript)
    tx.AddTxOut(txOut)
}
return tx, utxos, err
```

Snippet 4.4: Snippet from `rawTx()`

As shown, there is no guarantee that the input amount computed will have enough to cover any fees.

Impact If very low fees are provided, the transaction may take an extremely long time to be completed (or might be entirely rejected).

Recommendation Add some additional amount for the input calculation to account for fees.

4.1.4 V-BTC-VUL-004: Unexecuted proposals may be lost

Severity	Medium	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

When executing a cross-chain proposal (whose target is BTC), a proposal is marked as "pending" and is then signed by the relayer and communicated to the other relayers. Proposals which are "pending" will not be considered by the relayer again (they are filtered out by the call to `proposalsForExecution()` in the function `Execute()`). This means that if there is any failure during the signing process (e.g., a timeout occurs on signing, a network failure, etc.) the proposal will be lost and not considered again (even though it does in fact correspond to a valid transaction).

Impact This could lead to the funds associated with a proposal being lost by the user.

Recommendation Implement some backup (perhaps even manual) mechanism for fulfilling these potentially dropped proposals.

Developer Response The developers indicated that "The execution addresses have a spending condition that an admin multisig can unlock funds in case of (dropped proposals)".

4.1.5 V-BTC-VUL-005: Blocking transactions via small deposits

Severity	Medium	**Commit**	0d421d1
Type	Denial of Service	**Status**	Acknowledged
File(s)	`chains/btc/executor/executor.go`		
Location(s)	inputs()		
Confirmed Fix At	N/A		

A cross-chain transaction with BTC as a target is fulfilled by the construction of a BTC transaction which covers the expected output amounts. The input UTXOs for this BTC transaction are constructed via the `inputs()` function shown below.

```
1  func (e *Executor) inputs(tx *wire.MsgTx, address btcutil.Address, outputAmount int64
       ) (int64, []mempool.Utxo, error) {
2      usedUtxos := make([]mempool.Utxo, 0)
3      inputAmount := int64(0)
4      utxos, err := e.mempool.Utxos(address.String())
5      if err != nil {
6          return 0, nil, err
7      }
8      for _, utxo := range utxos {
9          previousTxHash, err := chainhash.NewHashFromStr(utxo.TxID)
10         if err != nil {
11             return 0, nil, err
12         }
13         outPoint := wire.NewOutPoint(previousTxHash, utxo.Vout)
14         txIn := wire.NewTxIn(outPoint, nil, nil)
15         tx.AddTxIn(txIn)
16
17         usedUtxos = append(usedUtxos, utxo)
18         inputAmount += int64(utxo.Value)
19         if inputAmount > outputAmount {
20             break
21         }
22     }
23     return inputAmount, usedUtxos, nil
24 }
```

Snippet 4.5: Implementation of `inputs()`

This function constructs the inputs to the transaction by querying the `mempool` for UTXOs associated with the given BTC address and then iterates through those UTXOs, saving each one until the amount of the UTXOs gathered exceeds the desired output amount.

On BTC, fees are used as an incentive for miners to mine a given transaction. The fees a miner will expect to mine a given block are a function of (among other things) the number of input UTXOs used to construct a transaction. Thus, an attacker could increase the fees needed to send a transaction by increasing the number of UTXOs associated with a given BTC address, which they could achieve by simply flooding the BTC address with very low (even single Satoshi) deposits.

Impact This could drastically slow down the protocol and/or require users to pay significantly more fees to cover cross-chain transactions.

Recommendation One option that could help mitigate this attack is requiring a minimum amount of BTC for a cross-chain transaction. With this, at least the attacker would just have to lose whatever small deposits they make to increase the number of UTXOs.

Developer Response The developers have acknowledged such an attack might be viable. They are considering charging a fee per output that would reduce the economic viability of the attack — in other words, while such a charge would not technically prohibit such an attack, it would make it economically infeasible/impractical for an attacker.

4.1.6 V-BTC-VUL-006: Possible transaction nonce collision

Severity	Low	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	`btc/listener/event-handlers.go`		
Location(s)	CalculateNonce()		
Confirmed Fix At	N/A		

The function `CalculateNonce()` is used to compute a nonce for a given event that is used as the unique identifier for that event when it is processed in the executor. The logic for computing this nonce is given in the code block below.

```
func (eh *FungibleTransferEventHandler) CalculateNonce(blockNumber *big.Int,
    evtNumber int) (uint64, error) {
    // Convert blockNumber to string
    blockNumberStr := blockNumber.String()

    // Convert evtNumber to *big.Int
    evtNumberBigInt := big.NewInt(int64(evtNumber))

    // Convert evtNumberBigInt to string
    evtNumberStr := evtNumberBigInt.String()

    // Concatenate blockNumberStr and evtNumberStr
    concatenatedStr := blockNumberStr + evtNumberStr

    // Parse the concatenated string to uint64
    result, err := strconv.ParseUint(concatenatedStr, 10, 64)
    if err != nil {
        return 0, err
    }

    return result, nil
}
```

Snippet 4.6: Implementation of `CalculateNonce()`

Here, the nonce is computed by concatenating a string representation of the block number and event number as strings. Thus, it is not guaranteed that the resulting nonce is unique. For instance, the nonce for block number `1` and event number `11` is `111` , as is the nonce for block number `11` and event `1`.

Impact In the event that multiple transactions shared the same nonce, the second transaction to get that nonce could not be executed by the relayer, as a transaction associated with that nonce would already be executed and stored by the relayer. As a result, funds associated with the transaction would be lost.

It should be noted that this is not likely to happen due to the fact that the current block height is in the 800,000's, meaning this type of collision is unlikely to occur for a very long time.

Recommendation Use technique for deriving a nonce for an event that is less likely to result in a collision.

Developer Response The developers replaced this function with a function that takes the block number and event hash, concatenates their strings together *separated by a dash*, hashes that string, and folds with XOR to get the necessary bytes. While there is technically a chance of collision in this technique, it is equivalent to the chance of a hash collision, which is very unlikely.

4.1.7 V-BTC-VUL-007: Possible race conditions

Severity	Low	**Commit**	0d421d1
Type	Race Condition	**Status**	Fixed
File(s)	`btc/executor/executor.go`		
Location(s)	storeProposalsStatus		
Confirmed Fix At	N/A		

In the function `proposalsForExecution()`, the function uses `e.propMutex` to attempt to avoid race condition issues for the entire function (presumably to avoid race conditions on reads/writes from the `propStorer`). There is no such use of the mutex in the function `storeProposalsStatus()` which writes to the `propStorer`.

Impact Because the `propStorer` is used to store information about past proposals, failure to handle race conditions in this case could lead to invalid data about proposals being stored.

Recommendation Use the `e.propMutex` in the function `storeProposalsStatus`.

4.1.8 V-BTC-VUL-008: Bad state on failure

Severity	Low	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	`tss/frost/resharing/resharing.go`		
Location(s)	Run()		
Confirmed Fix At	N/A		

In the function `Run()` for resharing, the line `defer r.Stop()` is used to ensure that appropriate cleanup processes happen when the resharing TSS process is ended, regardless of the reason. This line appears in `Run()` as is shown below.

```
...

outChn := make(chan tss.Message)
msgChn := make(chan *comm.WrappedMessage)
r.subscriptionID = r.Communication.Subscribe(r.SessionID(), comm.TssReshareMsg,
    msgChn)

r.key.Key.PublicKey = params
r.Handler, err = protocol.NewMultiHandler(
    frost.RefreshTaproot(
        r.key.Key,
        common.PartyIDSFromPeers(append(r.Host.Peerstore().Peers(), r.Host.ID()))),
    []byte(r.SessionID()))
if err != nil {
    return err
}

defer r.Stop()

...

return p.Wait()
```

Snippet 4.7: Snippet from `Run()`

The `Stop()` function is responsible for a number of important operations, including unsubscribing from the communication channel and unlocking the keyshare. However, as shown in the snippet above, if an error is encountered in the call to construct the `NewMultiHandler`, `Stop()` will not be called.

Impact This could lead to a locked state for a relayer where the keyshare is locked. It could also lead to an unclosed communication channel that could block future attempts to start a channel.

Recommendation Move the `defer r.Stop()` to the beginning of the logic to ensure it always runs.

Developer Response The developers have fixed the issue by enabling the coordinator to stop the resharing process, thus removing the need for `defer r.Stop()` within the `Run()` function.

4.1.9 V-BTC-VUL-009: Incorrectly handling empty msg.To

Severity	Low	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	`tss/frost/common/base.go`		
Location(s)	BroadcastPeers()		
Confirmed Fix At	N/A		

When processing outbound messages, the `BroadcastPeers()` function is used to determine the list of recipients for the message, as specified by `msg`. The documentation for `protocol.Message` states the following for the `To` field:

> `To` is the intended recipient for this message. If `To == ""`, then the message should be sent to all.

However, when determining the peers receiving the message, the `To == ""` case is handled by returning an empty list.

```
func (k *BaseFrostTss) BroadcastPeers(msg *protocol.Message) ([]peer.ID, error) {
    if msg.Broadcast {
        return k.Peers, nil
    } else {
        if string(msg.To) == "" {
            return []peer.ID{}, nil
        }
    ...
}
```

Snippet 4.8: Snippet from `BroadcastPeers()`

Impact If the `frost` implementation sets the `msg.To` field to `""`, outbound messages will be handled incorrectly, sending the messages to no peers instead of all peers. This may result in the protocol failing to successfully communicate messages, even when all relayers are acting appropriately.

Recommendation Handle the `To == ""` case as a broadcast message.

4.1.10 V-BTC-VUL-010: Unreliable/unstable dependencies

Severity	Warning	**Commit**	0d421d1
Type	Unreliable Dependenc	**Status**	Acknowledged
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

The dependency `rpcclient` uses an API which is not yet stable according to their documentation. Additionally, the dependency `multi-party-sig` used to implement much of the security-critical TSS is unaudited. In particular, in their documentation they write "DISCLAIMER: Use at your own risk, this project needs further testing and auditing to be production-ready."

Impact Unstable and un-audited dependencies are potential sources of bugs and errors. If possible, these dependencies should be avoided.

Recommendation Avoid use of unstable and un-audited dependencies as possible, especially for critical cryptographic operations. If this is not possible, seeking out additional auditing for cryptographic operations is strongly recommended.

Developer Response The developers indicated that `Multi-party-sig` will be audited separately. They also indicated that the `rpcclient` library has been stable for a while, even though it is technically in active development for the RPCs used by the project.

4.1.11 V-BTC-VUL-011: Unclear BTC transaction assumptions

Severity	Warning	**Commit**	0d421d1
Type	Usability Issue	**Status**	Fixed
File(s)		N/A	
Location(s)		N/A	
Confirmed Fix At		N/A	

There are a number of assumptions made about BTC transactions which are not clearly documented. Consider the following loop from `DecodeDepositEvent()` from `chains/btc/listener/util.go`.

```
for _, vout := range evt.Vout {
    // read the OP_RETURN data
    if vout.ScriptPubKey.Type == OP_RETURN {
        opReturnData, err := hex.DecodeString(vout.ScriptPubKey.Hex)
        if err != nil {
            return Deposit{}, true, err
        }
        // Extract OP_RETURN data (excluding OP_RETURN prefix)
        data = string(opReturnData[2:])
    }

    if resource.Address.String() == vout.ScriptPubKey.Address {
        isBridgeDeposit = true
        resourceID = resource.ResourceID
        if vout.ScriptPubKey.Type == PubKeyHash || vout.ScriptPubKey.Type ==
    ScriptHash || vout.ScriptPubKey.Type == WitnessV0KeyHash {
            amount.Add(amount, big.NewInt(int64(vout.Value*1e8)))
        }
    }
}
```

Snippet 4.9: Snippet from `DecodeDepositEvent()`

The loop iterates through the outputs for a given transaction. It computes the total output value (i.e., `amount`) by summing the amounts for any output where the resource address matches the script address (assuming the script type is one of the script types considered). Additionally, if the script type stores data (using `OP_RETURN`), those data are fetched and stored in the `data` field. In this case, it is assumed there is only one output with type `OP_RETURN` (otherwise, the `data` field will be overwritten).

Furthermore, the `data` from above is processed further in the following snippet from `HandleDeposit()` in `chains/btc/listener/deposit-handler.go`.

As indicated by the comment, the `data` is assumed to have a very specific format containing the receiver EVM address and destination chain ID.

Impact If these assumptions are intended, a lack of clarity could lead to lost user funds if they do not submit transactions in the desired format. If they are not intended, funds could be lost.

Recommendation Clarify all assumptions around the format of BTC transactions.

```
// data is composed of recieverEVMAddress_destinationDomainID
parsedData := strings.Split(data, "_")
evmAdd := common.HexToAddress(parsedData[0]).Bytes()
destDomainID, err := strconv.ParseUint(parsedData[1], 10, 8)
if err != nil {
    return nil, err
}
```

Snippet 4.10: Snippet from `HandleDeposit()`

4.1.12 V-BTC-VUL-012: Inefficiency on incomplete peers

Severity	Warning	**Commit**	0d421d1
Type	Logic Error	**Status**	Intended Behavior
File(s)	`tss/frost/signing/signing.go`		
Location(s)	Run()		
Confirmed Fix At	N/A		

When calling `Run()` to perform signing on a particular relayer, the following check is used to ensure that the current relayer is included in the chosen peer set.

```
if !util.IsParticipant(s.Host.ID(), peerSubset) {
    return &errors.SubsetError{Peer: s.Host.ID()}
}
```

Snippet 4.11: Snippet from `Run()`

The peer subset is chosen based on the starting parameters chosen, which are either chosen from the start message from the coordinator, or, if the current relayer is the coordinator, from the function `StartParams()` shown below.

```
func (s *Signing) StartParams(readyMap map[peer.ID]bool) []byte {
    readyMap = s.readyParticipants(readyMap)
    peers := []peer.ID{}
    for peer := range readyMap {
        peers = append(peers, peer)
    }

    sortedPeers := util.SortPeersForSession(peers, s.SessionID())
    peerSubset := []peer.ID{}
    for _, peer := range sortedPeers {
        peerSubset = append(peerSubset, peer.ID)
        if len(peerSubset) == s.key.Threshold+1 {
            break
        }
    }

    paramBytes, _ := json.Marshal(peerSubset)
    return paramBytes
}
```

Snippet 4.12: Implementation of `StartParams()`

As is shown, `StartParams()` retrieves the first `s.key.Threshold+1` "ready" peers, but does not necessarily include the current relayer. This means that there is a chance that the current relayer will not be chosen and the `SubsetError` will be triggered when the current relayer calls `Run()`.

Impact This will require retrying the signing process, which is slow and consumes unnecessary resources.

Recommendation Have the starting parameters include the current relayer.

Developer Response Per the developers, the coordinator is always set as ready to go in `coordinator.go` and will always be chosen as the first after sorting in both static and bully elections. Thus, the coordinator will always be the first `peer` in `sortedPeers` and will thus always be added to the `peerSubset`.

4.1.13 V-BTC-VUL-013: Typos and incorrect comments

Severity	Info	**Commit**	0d421d1
Type	Maintainability	**Status**	Fixed
File(s)	See issue description		
Location(s)	See issue description		
Confirmed Fix At	N/A		

Description In the following locations, the auditors identified minor typos and potentially misleading comments:

- `chains/btc/listener/event-handlers.go`:
  - `struct Deposit`: The comment for `ResourceID` states that the resource ID used to determine the handler for the deposit. However, this does not seem to be true for the logic below, where the `FungibleTransferEventHandler` seems to be used regardless of resource ID.
- `tss/frost/common/util.go`:
  - `PartyIDSFromPeers()`:
    - The call `sort.Sort(peers)` is not necessary as the values are immediately added to a set (and are thus unordered thereafter).
    - The function `.Pretty()` is deprecated in favor of `.String()`.
- `tss/frost/signing/resharing.go`:
  - `Ready()`: The `Ready()` function does not use the truth values in the `readyMap` (nor does the `Ready()` function in `keygen.go`). However, the `Ready()` function in `signing.go` does. As best we can tell, values in the ready map are always set to `true`, meaning the code need not track the additional truth value.

Impact These minor errors may lead to future developer confusion.

4.1.14 V-BTC-VUL-014: Bad non-zero check

Severity	Info	**Commit**	0d421d1
Type	Logic Error	**Status**	Fixed
File(s)	`btc/config.go`		
Location(s)	Validate()		
Confirmed Fix At	N/A		

The `Validate()` function is used to validate various aspects of the configuration provided for BTC. It includes the following check for block confirmations.

```
1 if c.BlockConfirmations != 0 && c.BlockConfirmations < 1 {
2     return fmt.Errorf("blockConfirmations has to be >=1")
3 }
```

Snippet 4.13: Snippet from `Validate()`

Despite the error message, this does not disallow a value for `c.BlockConfirmations` of `0`.

Impact Having a block confirmations value of `0` could be dangerous, as it would mean all transactions are immediately processed when seen.

Recommendation Make the check disallow block confirmations of `0`.