BLOCKSEC

Security Audit
Report for Fast Finality Network

Date: August 11, 2025 **Version:** 1.0

Contact: contact@blocksec.com

Contents

Chapter 1 Introduction **1**

1.1 About Target Contracts . . . 1
1.2 Disclaimer . . . 2
1.3 Procedure of Auditing . . . 2
1.3.1 Security Issues . . . 3
1.3.2 Additional Recommendation . . . 3
1.4 Security Model . . . 3

Chapter 2 Findings **5**

2.1 Security Issue . . . 6
2.1.1 Incorrect verification mechanism for collected responses . . . 6
2.1.2 Circumvention of the slashing mechanism . . . 9
2.1.3 Lack of an exit condition in the function `work()` . . . 10
2.1.4 Inconsistent validations of the stake amount . . . 11
2.1.5 Incorrect value assignments during header constructions . . . 12
2.1.6 Incorrect construction of the variable `voteStateRoot` . . . 13
2.1.7 Improper error handling logic in the file `msc.go` . . . 13
2.1.8 Incorrect calculation of `signerApk` in the function `checkSignatures()` . . . 15
2.1.9 Potential data race due to the improper use of the mutex . . . 16
2.1.10 Lack of initialization for the field `MaxSubmissionRetries` . . . 18
2.1.11 Ungraceful shutdown due to the uninitialized field `cancel` of the struct `Node` 18
2.1.12 Unsafe confirmation depth . . . 19
2.1.13 Improper validation design for checking sync status . . . 22
2.1.14 Lack of validation for minimum valid signatures threshold . . . 22
2.1.15 Lack of proper implementations in the function `Close()` . . . 23
2.1.16 Lack of closing the channel `stopChan` in the function `Stop()` . . . 24
2.1.17 Inconsistent error handling logic . . . 27
2.1.18 Lack of proper resource cleanup . . . 28
2.1.19 Fail to start the file `manager.go` . . . 29
2.1.20 Potential duplicate node registration leading to incorrect `currentApk` calculation . . . 29
2.1.21 Lack of non zero check for `_operatorName` in the function `registerOperator()` 30
2.1.22 Potential DoS due to the improper use of `ERC20` interfaces . . . 31
2.1.23 Lack of checks for the registration status of `nonSignerPubkeys` . . . 32
2.1.24 Lack of checks for the inputs `operator` and `pubkeyRegistrationMessageHash` 32
2.1.25 Ungraceful shutdown due to the lack of invoking the function `m.wg.Wait()` 34
2.2 Recommendation . . . 35
2.2.1 Use struct to unmarshal http response body . . . 35
2.2.2 Redundant Code . . . 37
2.2.3 Non zero address checks . . . 39

2.2.4 Handle the error in the function `handleSign()` 40
2.3 Note 40
2.3.1 The slashing mechanism has not yet been implemented 40
2.3.2 Potential centralization risks 41
2.3.3 Security audit assumptions on BLS signatures 41

Report Manifest

Item	Description
Client	Manta Network
Target	Fast Finality Network

Version History

Version	Date	Description
1.0	August 11, 2025	First release

Signature

Digitally signed by: BlockSec (https://blocksec.com)

SHA1 digest of public key: BAB106D3708BC6DE0963CE05E9478C24B326AE1C

Date: 2025-08-12 02:42:18+0000

About BlockSec BlockSec focuses on the security of the blockchain ecosystem and collaborates with leading DeFi projects to secure their products. BlockSec is founded by top-notch security researchers and experienced experts from both academia and industry. They have published multiple blockchain security papers in prestigious conferences, reported several zero-day attacks of DeFi applications, and successfully protected digital assets that are worth more than 14 million dollars by blocking multiple attacks. They can be reached at Email, Twitter and Medium.

Chapter 1 Introduction

1.1 About Target Contracts

Information	Description
Type	Smart Contract
Language	Solidity
Approach	Semi-automatic and manual verification

The target of this audit is the code repository [1][2][3][4][5] of Fast Finality Network of Manta Network.

The Fast Finality Network of the Manta Network is a protocol designed to enable fast block finality on the Manta Layer 2 network and to support rapid withdrawals to Layer 1. In the project, there are three key roles: Finality Provider (FP), Node, and Manager. Finality Providers (FPs) are responsible for submitting finality signatures for proposed state roots (i.e., blocks). These signatures are submitted to selected networks (e.g., the Celestia chain) for further verification. The Manager tracks state roots and sends corresponding voting requests to Nodes. Upon receiving a request, Nodes verify the corresponding finality signatures and cast their votes. Only signatures from valid FPs (i.e., those with sufficient active staked amounts) are recorded in the Nodes responses. Finally, the Manager finalizes the block (i.e., reducing the finalizing time) once two-thirds of Nodes have cast an agreed vote. It is worth noting that both the Manager and Nodes are operated by the project, while anyone can participate in the Fast Finality Network as a Finality Provider.

Note this audit only focuses on the smart contracts in the following directories/files:

- manta-fp-contracts/src/core/FinalityRelayerManager.sol
- manta-fp-contracts/src/bls/BLSApkRegistry.sol
- manta-staking-contracts/src/MantaStakingMiddleware.sol
- manta-staking-contracts/src/TokenDistributor.sol
- celestia-bedrock-manta/packages/contracts-bedrock/contracts/L1/L2OutputOracle.sol
- celestia-bedrock-manta/packages/contracts-bedrock/contracts/L1/OptimismPortal.sol
- celestia-bedrock-manta/packages/contracts-bedrock/scripts/UpgradeL2OutputOracle.s.sol
- celestia-bedrock-manta/packages/contracts-bedrock/scripts/UpgradeOptimismPortal.s.sol
- manta-fp/symbiotic-fp/mantastaking/msc.go
- manta-fp-aggregator/manager/manager.go
- manta-fp-aggregator/node/node.go
- manta-fp-aggregator/synchronizer/eth_synchronizer.go
- manta-fp-aggregator/synchronizer/celestia_synchronizer.go

[1] https://github.com/Manta-Network/manta-fp-contracts.git

[2] https://github.com/Manta-Network/manta-staking-contracts.git

[3] https://github.com/Manta-Network/celestia-bedrock-manta.git

[4] https://github.com/Manta-Network/manta-fp.git

[5] https://github.com/Manta-Network/manta-fp-aggregator.git

Other files are not within the scope of the audit. Additionally, all dependencies of the smart contracts within the audit scope are considered reliable in terms of both functionality and security, and are therefore not included in the audit scope.

The auditing process is iterative. Specifically, we would audit the commits that fix the discovered issues. If there are new issues, we will continue this process. The commit SHA values during the audit are shown in the following table. Our audit report is responsible for the code in the initial version (`Version 1`), as well as new code (in the following versions) to fix issues in the audit report. Code prior to and including the baseline version (`Version 0`), where applicable, is outside the scope of this audit and assumes to be reliable and secure.

Project	Version	Commit Hash
manta-fp-contracts	Version 1	aee8feb7890d5fbe62a2e2cc0a23d73040818fef
	Version 2	044e890e589b57e4ac35c2c4ef29a49963c8eb76
manta-staking-contracts	Version 1	01bb95446df046c7ac8c643a590ef9dffa1388f1
	Version 2	6fc7546ce97bea4846ac580c55d8f1a0ebb59fd9
celestia-bedrock-manta	Version 1	d23284fbba3373a36ccaef30e9e3a673afb6c70b
manta-fp	Version 1	42ea2aef5f51ffd2f1ee86ac5e815f008fbddf7c
	Version 2	24da135b4fa4fcee9738aa6dea03f0e175ec2bec
manta-fp-aggregator	Version 1	68660cd499c83e09ca768d9119a71ad17c81ddcf
	Version 2	685b901fd8abf048d4177a6f91eb1d3dcdd7501b

1.2 Disclaimer

This audit report does not constitute investment advice or a personal recommendation. It does not consider, and should not be interpreted as considering or having any bearing on, the potential economics of a token, token sale or any other product, service or other asset. Any entity should not rely on this report in any way, including for the purpose of making any decisions to buy or sell any token, product, service or other asset.

This audit report is not an endorsement of any particular project or team, and the report does not guarantee the security of any particular project. This audit does not give any warranties on discovering all security issues of the smart contracts, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues. As one audit cannot be considered comprehensive, we always recommend proceeding with independent audits and a public bug bounty program to ensure the security of smart contracts.

The scope of this audit is limited to the code mentioned in Section 1.1. Unless explicitly specified, the security of the language itself (e.g., the solidity language), the underlying compiling toolchain and the computing infrastructure are out of the scope.

1.3 Procedure of Auditing

We perform the audit according to the following procedure.

- **Vulnerability Detection** We first scan smart contracts with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.

- **Semantic Analysis** We study the business logic of smart contracts and conduct further investigation on the possible vulnerabilities using an automatic fuzzing tool (developed by our research team). We also manually analyze possible attack scenarios with independent auditors to cross-check the result.
- **Recommendation** We provide some useful advice to developers from the perspective of good programming practice, including gas optimization, code style, and etc.

We show the main concrete checkpoints in the following.

1.3.1 Security Issues

* Access control
* Permission management
* Whitelist and blacklist mechanisms
* Initialization consistency
* Improper use of the proxy system
* Reentrancy
* Denial of Service (DoS)
* Untrusted external call and control flow
* Exception handling
* Data handling and flow
* Events operation
* Error-prone randomness
* Oracle security
* Business logic correctness
* Semantic and functional consistency
* Emergency mechanism
* Economic and incentive impact

1.3.2 Additional Recommendation

* Gas optimization
* Code quality and style

Note *The previous checkpoints are the main ones. We may use more checkpoints during the auditing process according to the functionality of the project.*

1.4 Security Model

To evaluate the risk, we follow the standards or suggestions that are widely adopted by both industry and academy, including OWASP Risk Rating Methodology [6] and Common Weakness Enumeration [7]. The overall *severity* of the risk is determined by *likelihood* and *impact*.

[6] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

[7] https://cwe.mitre.org/

Specifically, likelihood is used to estimate how likely a particular vulnerability can be uncovered and exploited by an attacker, while impact is used to measure the consequences of a successful exploit.

In this report, both likelihood and impact are categorized into two ratings, i.e., *high* and *low* respectively, and their combinations are shown in Table 1.1.

Table 1.1: Vulnerability Severity Classification

Impact \ Likelihood	*High*	*Low*
High	High	Medium
Low	Medium	Low

Accordingly, the severity measured in this report are classified into three categories: **High**, **Medium**, **Low**. For the sake of completeness, **Undetermined** is also used to cover circumstances when the risk cannot be well determined.

Furthermore, the status of a discovered item will fall into one of the following five categories:

- **Undetermined** No response yet.
- **Acknowledged** The item has been received by the client, but not confirmed yet.
- **Confirmed** The item has been recognized by the client, but not fixed yet.
- **Partially Fixed** The item has been confirmed and partially fixed by the client.
- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we found **twenty five** potential security issues. Besides, we have **four** recommendations and **three** notes.

- High Risk: 8
- Medium Risk: 6
- Low Risk: 11
- Recommendation: 4
- Note: 3

ID	Severity	Description	Category	Status
1	High	Incorrect verification mechanism for collected responses	Security Issue	Fixed
2	High	Circumvention of the slashing mechanism	Security Issue	Fixed
3	High	Lack of an exit condition in the function `work()`	Security Issue	Fixed
4	High	Inconsistent validations of the stake amount	Security Issue	Fixed
5	High	Incorrect value assignments during header constructions	Security Issue	Fixed
6	High	Incorrect construction of the variable `voteStateRoot`	Security Issue	Fixed
7	High	Improper error handling logic in the file `msc.go`	Security Issue	Fixed
8	High	Incorrect calculation of `signerApk` in the function `checkSignatures()`	Security Issue	Fixed
9	Medium	Potential data race due to the improper use of the mutex	Security Issue	Fixed
10	Medium	Lack of initialization for the field `MaxSubmissionRetries`	Security Issue	Fixed
11	Medium	Ungraceful shutdown due to the uninitialized field `cancel` of the struct `Node`	Security Issue	Fixed
12	Medium	Unsafe confirmation depth	Security Issue	Fixed
13	Medium	Improper validation design for checking sync status	Security Issue	Fixed
14	Medium	Lack of validation for minimum valid signatures threshold	Security Issue	Fixed
15	Low	Lack of proper implementations in the function `Close()`	Security Issue	Fixed
16	Low	Lack of closing the channel `stopChan` in the function `Stop()`	Security Issue	Fixed
17	Low	Inconsistent error handling logic	Security Issue	Fixed

18	Low	Lack of proper resource cleanup	Security Issue	Fixed
19	Low	Fail to start the file `manager.go`	Security Issue	Fixed
20	Low	Potential duplicate node registration leading to incorrect `currentApk` calculation	Security Issue	Fixed
21	Low	Lack of non zero check for `_operatorName` in the function `registerOperator()`	Security Issue	Fixed
22	Low	Potential DoS due to the improper use of `ERC20` interfaces	Security Issue	Fixed
23	Low	Lack of checks for the registration status of `nonSignerPubkeys`	Security Issue	Fixed
24	Low	Lack of checks for the inputs `operator` and `pubkeyRegistrationMessageHash`	Security Issue	Partially Fixed
25	Low	Ungraceful shutdown due to the lack of invoking the function `m.wg.Wait()`	Security Issue	Fixed
26	-	Use struct to unmarshal http response body	Recommendation	Fixed
27	-	Redundant Code	Recommendation	Fixed
28	-	Non zero address checks	Recommendation	Fixed
29	-	Handle the error in the function `handleSign()`	Recommendation	Fixed
30	-	The slashing mechanism has not yet been implemented	Note	-
31	-	Potential centralization risks	Note	-
32	-	Security audit assumptions on BLS signatures	Note	-

The details are provided in the following sections.

2.1 Security Issue

2.1.1 Incorrect verification mechanism for collected responses

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `manager.go`, the manager collects responses (i.e., voting details) from nodes (i.e., via the function `sign()` in the file `sign.go`) and invokes the `VerifyFinalitySignature()` function in the contract `FinalityRelayerManager` to complete the finality process. However, in the file `sign.go`, the function `sign()` only verifies the total number of responses (i.e., `respNumber`) against the threshold (i.e., `len(ctx.AvailableNodes())*2/3`). This flawed design leads to two critical issues:

1. Lack of agreed vote verification. The responses verification process does not check whether the number of agreed votes meets the required threshold. As a result, a state root could be finalized on-chain without sufficient agreed votes.

2. Single point of failure. If a node is offline but the total number of responses passes the threshold check, the list `NonSignerPubKeys` becomes incomplete. As a result, the incomplete list `NonSignerPubKeys` leads to on-chain signature verification failure, as the verification process requires full participation from all registered operations (i.e., nodes).

```
42 go func() {
43   cctx, cancel := context.WithTimeout(context.Background(), m.cfg.Manager.SignTimeout)
44   defer func() {
45     m.log.Info("exit signing process")
46     cancel()
47     close(stopChan)
48     wg.Done()
49   }()
50   for {
51     select {
52     case <-errSendChan:
53       return
54     case resp := <-respChan:
55       m.log.Info(fmt.Sprintf("signed response: %s", resp.RpcResponse.String()), "node", resp.
           SourceNode)
56       if !ExistsIgnoreCase(ctx.AvailableNodes(), resp.SourceNode) { // ignore the message which
           the sender should not be involved in approver set
57         continue
58       }
59       respNumber++
60       func() {
61         defer func() {
62           responseNodes[resp.SourceNode] = struct{}{}
63         }()
64         if resp.RpcResponse.Error != nil {
65           m.log.Error("Unrecognized error code",
66             "err_code", resp.RpcResponse.Error.Code,
67             "err_data", resp.RpcResponse.Error.Data,
68             "err_message", resp.RpcResponse.Error.Message)
69           return
70         } else {
71           var signResponse types.SignMsgResponse
72           if err = tmjson.Unmarshal(resp.RpcResponse.Result, &signResponse); err != nil {
73             m.log.Error("failed to unmarshal sign response", "err", err)
74             return
75           }
76
77           if signResponse.Vote != uint8(common.AgreeVote) {
78             g1Point, err = new(sign.G1Point).Deserialize(signResponse.NonSignerPubkey)
79             if err != nil {
80               m.log.Error("failed to deserialize g1Point", "err", err)
81               return
82             }
83             NonSignerPubkeys = append(NonSignerPubkeys, g1Point)
```

```
84            return
85          }
86          dG2Point, err := g2Point.Deserialize(signResponse.G2Point)
87          if err != nil {
88            m.log.Error("failed to deserialize g2Point", "err", err)
89            return
90          }
91
92          dSign, err := g1Point.Deserialize(signResponse.Signature)
93          if err != nil {
94            m.log.Error("failed to deserialize signature", "err", err)
95            return
96          }
97          g2Points = append(g2Points, dG2Point)
98          g1Points = append(g1Points, dSign)
99          return
100       }
101     }()
102
103   case <-cctx.Done():
104     m.log.Warn("wait for signature timeout", "requestId", ctx.RequestId(), "received responses
          len", respNumber)
105     return
106   default:
107     if respNumber == len(ctx.AvailableNodes()) {
108       m.log.Info("received all signing responses", "requestId", ctx.RequestId(), "received
            responses len", respNumber)
109       return
110     }
111   }
112 }
113 }()
```

Listing 2.1: manta-fp-aggregator/manager/sign.go

```
128   function checkSignatures(bytes32 msgHash, uint256 referenceBlockNumber,
        FinalityNonSignerAndSignature memory params)
129     public
130     view
131     returns (StakeTotals memory, bytes32)
132   {
133     require(
134       referenceBlockNumber < uint32(block.number), "BLSSignatureChecker.checkSignatures:
            invalid reference block"
135     );
136     BN254.G1Point memory signerApk = BN254.G1Point(0, 0);
137     bytes32[] memory nonSignersPubkeyHashes;
138     if (params.nonSignerPubkeys.length > 0) {
139       nonSignersPubkeyHashes = new bytes32[](params.nonSignerPubkeys.length);
140       for (uint256 j = 0; j < params.nonSignerPubkeys.length; j++) {
141         nonSignersPubkeyHashes[j] = params.nonSignerPubkeys[j].hashG1Point();
142         signerApk = currentApk.plus(params.nonSignerPubkeys[j].negate());
143       }
```

```
144         } else {
145             signerApk = currentApk;
146         }
147         (bool pairingSuccessful, bool signatureIsValid) =
148             trySignatureAndApkVerification(msgHash, signerApk, params.apkG2, params.sigma);
```

Listing 2.2: manta-fp-contracts/src/bls/BLSApkRegistry.sol

Impact The flawed design could result in a state root being finalized on-chain without sufficient agreed votes or cause an on-chain verification failure.

Suggestion Revise the logic accordingly.

2.1.2 Circumvention of the slashing mechanism

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the project, a malicious FP's (i.e., Finality Provider) vault (i.e., created via the contract `MantaStakingMiddleware`) will be slashed. In the files `msc.go`, `node.go` and `manager.go`, the function `getSymbioticOperatorStakeAmount()` is implemented to retrieve the total active staked amount (i.e., the variable `vaultTotalActiveStaked`) for a given FP (i.e., the variable `operator`) for the state root finalizing. Specifically, this data fetching is achieved using a query service (i.e., `m.cfg.SymbioticStakeUrl`), which is maintained by the project. However, the function `getSymbioticOperatorStakeAmount()` retrieves the total active staked amount across the FP's all vaults in the Symbiotic protocol, leading the FP to circumvent the slashing mechanism. For example, a malicious FP can create additional vaults via the Symbiotic protocol directly and cast a malicious vote with a large staked amount in their external vaults. As a result, the malicious FP can interrupt the system without being slashed.

```
623 query := fmt.Sprintf(`{"query":"query {\n vaultUpdates(first: 1, where: {operator: \"%s\"},
        orderBy: timestamp, orderDirection: desc) {\n vaultTotalActiveStaked\n }\n}"}`, operator)
```

Listing 2.3: manta-fp/symbiotic-fp/mantastaking/msc.go

```
805func (m *Manager) getSymbioticOperatorStakeAmount(operator string) (*big.Int, error) {
806 query := fmt.Sprintf(`{"query":"query {\n vaultUpdates(first: 1, where: {operator: \"%s\"},
        orderBy: timestamp, orderDirection: desc) {\n vaultTotalActiveStaked\n }\n}"}`, operator)
807 jsonQuery := []byte(query)
808
809 req, err := http.NewRequest("POST", m.cfg.SymbioticStakeUrl, bytes.NewBuffer(jsonQuery))
```

Listing 2.4: manta-fp-aggregator/manager/manager.go

```
469func (n *Node) getSymbioticOperatorStakeAmount(operator string) (*big.Int, error) {
470 query := fmt.Sprintf(`{"query":"query {\n vaultUpdates(first: 1, where: {operator: \"%s\"},
        orderBy: timestamp, orderDirection: desc) {\n vaultTotalActiveStaked\n }\n}"}`, operator)
471 jsonQuery := []byte(query)
472
473 req, err := http.NewRequest("POST", n.cfg.SymbioticStakeUrl, bytes.NewBuffer(jsonQuery))
```

Listing 2.5: manta-fp-aggregator/node/node.go

Impact The malicious FP can interrupt the system without being slashed.

Suggestion Revise the logic accordingly.

Clarification from BlockSec Finality providers should check the staked amount of their vaults before submitting a finality signature. Signatures submitted by providers without a sufficient active staked amount will be discarded by the manager and nodes.

2.1.3 Lack of an exit condition in the function `work()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `node.go`, the function `work()` contains a loop that processes messages received from the channel `n.txMsgChan`. However, the loop lacks an exit condition, preventing the corresponding wait group from being closed (i.e., `n.wg.Done()`). As a result, the function `Stop()` becomes ineffective due to the code `n.wg.Wait()`, and the node cannot shut down gracefully.

```
175func (n *Node) work() {
176  defer n.wg.Done()
177  for {
178    select {
179    case txMsg := <-n.txMsgChan:
180      if err := n.babylonSynchronizer.ProcessNewFinalityProvider(txMsg); err != nil {
181        n.log.Error("failed to process NewFinalityProvider msg", "err", err)
182        continue
183      }
184      if err := n.babylonSynchronizer.ProcessCreateBTCDelegation(txMsg); err != nil {
185        n.log.Error("failed to process CreateBTCDelegation msg", "err", err)
186        continue
187      }
188      if err := n.babylonSynchronizer.ProcessCommitPubRandList(txMsg); err != nil {
189        n.log.Error("failed to process CommitPubRandList msg", "err", err)
190        continue
191      }
192      if err := n.babylonSynchronizer.ProcessBTCUndelegate(txMsg); err != nil {
193        n.log.Error("failed to process BTCUndelegate msg", "err", err)
194        continue
195      }
196      if err := n.babylonSynchronizer.ProcessSelectiveSlashingEvidence(txMsg); err != nil {
197        n.log.Error("failed to process SelectiveSlashingEvidence msg", "err", err)
198        continue
199      }
200      if err := n.babylonSynchronizer.ProcessSubmitFinalitySignature(txMsg); err != nil {
201        n.log.Error("failed to process SubmitFinalitySignature msg", "err", err)
202        continue
203      }
```

```
204    }
205  }
206}
```

Listing 2.6: manta-fp-aggregator/node/node.go

```
156func (n *Node) Stop(ctx context.Context) error {
157  n.cancel()
158  close(n.done)
159  n.wg.Wait()
160  n.babylonSynchronizer.Close()
161  n.celestiaSynchronizer.Close()
162  if n.metricsServer != nil {
163    if err := n.metricsServer.Close(); err != nil {
164      n.log.Error("failed to close metrics server", "err", err)
165    }
166  }
167  n.stopped.Store(true)
168  return nil
169}
```

Listing 2.7: manta-fp-aggregator/node/node.go

Impact The node cannot shut down gracefully due to the lack of an exit condition in the function `work()`.

Suggestion Revise the logic accordingly.

2.1.4 Inconsistent validations of the stake amount

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The files `node.go` and `msc.go` employ inconsistent validation for the stake amount of the Finality Providers (i.e., FPs). Specifically, in the file `msg.go`, a FP can only submit signatures when its stake amount is greater than the required minimum amount (i.e., `stakeAmount >= stakeLimit`). However, the file `node.go` (i.e., line 447) only verifies if a FP's staking amount is greater than zero. As a result, the inconsistent validations allow FPs to submit signatures with an insufficient stake amount, potentially affecting the finality result with low costs.

```
441     if symbioticFpSignCache[sfs.SignRequests.SignAddress] == "" {
442       amount, err := n.getSymbioticOperatorStakeAmount(strings.ToLower(sfs.SignRequests.
              SignAddress))
443       if err != nil {
444         n.log.Error("failed to get operator stake amount", "address", sfs.SignRequests.
              SignAddress, "err", err)
445         continue
446       }
447       if amount.Cmp(big.NewInt(0)) > 0 {
448         stateRootCountCache[sfs.SignRequests.StateRoot]++
449         symbioticFpSignCache[sfs.SignRequests.SignAddress] = sfs.SignRequests.StateRoot
```

```
450       }
451     }
```

Listing 2.8: manta-fp-aggregator/node/node.go

```
613 stakeLimit, _ := new(big.Int).SetString(msm.Cfg.StakeLimit, 10)
614 if stakeAmount.Cmp(stakeLimit) < 0 {
615   msm.log.Error("the total stake amount is insufficient", zap.String("staked", stakeAmount.
          String()), zap.String("required", msm.Cfg.StakeLimit))
616   return false, nil
617 }
```

Listing 2.9: manta-fp/symbiotic-fp/mantastaking/msc.go

Impact The inconsistent validations allow FPs to submit signatures with an insufficient stake amount, potentially affecting the finality result with low costs.

Suggestion Revise the logic accordingly.

2.1.5 Incorrect value assignments during header constructions

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `processBatch()` of the `celestia_synchronizer.go` and `eth_synchronizer.go` files, newly fetched block headers are processed and stored in the database (via the functions `db.SetEthBlockHeaders()` and `db.SetCelestiaBlockHeaders()`). However, in the header construction process, incorrect values are assigned to the fields `CelestiaBlockHeader.ParentHash` and `EthBlockHeader.Hash`. Specifically, the field `EthBlockHeader.Hash` should be `headers[i].Hash()` and the field `CelestiaBlockHeader.ParentHash` should be `headers[i].LastHeader()`. As a result, headers with incorrect information are stored in the database, potentially affecting further operations.

```
162   eHeader := store.EthBlockHeader{
163     Hash:      headers[i].TxHash,
164     ParentHash: headers[i].ParentHash,
165     Number:    headers[i].Number.Int64(),
166     Timestamp: int64(headers[i].Time),
167   }
```

Listing 2.10: manta-fp-aggregator/synchronizer/eth_synchronizer.go

```
154   cHeader := store.CelestiaBlockHeader{
155     Hash:      headers[i].Hash(),
156     ParentHash: headers[i].LastResultsHash.Bytes(),
157     Number:    headers[i].Height(),
158     Timestamp: headers[i].Time().Unix(),
159   }
```

Listing 2.11: manta-fp-aggregator/synchronizer/celestia_synchronizer.go

Impact Headers with incorrect information are stored in the database, potentially affecting further operations.

Suggestion Revise the logic accordingly.

2.1.6 Incorrect construction of the variable `voteStateRoot`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `manager.go`, the function `processStateRoot()` finalizes unprocessed state roots by sending sign requests to nodes. This function begins by constructing a sign request using the function `getMaxSignStateRoot()` to retrieve the unprocessed state root (i.e., `voteStateRoot`). However, in the function `getMaxSignStateRoot()`, the function incorrectly uses the variables `op.L1BlockNumber` and `op.L2BlockNumber` of the processed state root (i.e., line 734 and 743) to select the finality signatures, resulting in an incorrect construction of the unprocessed state root (i.e., `voteStateRoot`) for the signing process. As a result, the intended unprocessed state root cannot be finalized, causing the program to behave incorrectly.

```
473func (m *Manager) processStateRoot(op *store.OutputProposed) error {
474  voteStateRoot, err := m.getMaxSignStateRoot(op.Timestamp.Uint64())
475  m.log.Info("success to count fp signatures", "result", voteStateRoot)
476  if err != nil {
477    m.log.Error("failed to get max sign state root", "err", err)
478    return err
479  }
```

Listing 2.12: manta-fp-aggregator/manager/manager.go

```
717  op, err := m.db.GetLatestProcessedStateRoot()
```

Listing 2.13: manta-fp-aggregator/manager/manager.go

```
734    if bfs.SubmitFinalitySignature.L2BlockNumber == op.L2BlockNumber.Uint64() {
```

Listing 2.14: manta-fp-aggregator/manager/manager.go

```
743    if sfs.SignRequests.L1BlockNumber == op.L1BlockNumber && sfs.SignRequests.L2BlockNumber == op.
           L2BlockNumber.Uint64() {
```

Listing 2.15: manta-fp-aggregator/manager/manager.go

Impact The intended unprocessed state root cannot be finalized, causing the program to behave incorrectly.

Suggestion Revise the logic accordingly.

2.1.7 Improper error handling logic in the file `msc.go`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `msc.go`, the function `SubmitBatchFinalitySignatures()` submits the finality signature to the Celestia chain within a nested `if` block but forgets to return errors when the submission fails. This improper error handling logic prevents the retry mechanism from being triggered. As a result, the finality signature may be dropped, and the program continues processing new state roots.

```
376 if msm.DAClient != nil && msm.DAClient.Client != nil {
377   commit, err := celestia.CreateCommitment(data, msm.DAClient.Namespace)
378   if err == nil {
379     ctx2, cancel := context.WithTimeout(ctx, msm.DAClient.GetTimeout)
380     ids, err := msm.DAClient.Client.Submit(ctx2, [][]byte{data}, -1, msm.DAClient.Namespace)
381     cancel()
382     if err == nil && len(ids) == 1 && len(ids[0]) == 40 && bytes.Equal(commit, ids[0][8:]) {
383       msm.log.Info("celestia: blob successfully submitted", zap.String("id", hex.EncodeToString(
              ids[0])))
384       ctx2, cancel := context.WithTimeout(ctx, msm.DAClient.GetTimeout)
385       proofs, err := msm.DAClient.Client.GetProofs(ctx2, ids, msm.DAClient.Namespace)
386       cancel()
387       if err == nil && len(proofs) == 1 {
388         ctx2, cancel := context.WithTimeout(ctx, msm.DAClient.GetTimeout)
389         valids, err := msm.DAClient.Client.Validate(ctx2, ids, proofs, msm.DAClient.Namespace)
390         cancel()
391         if err == nil && len(valids) == 1 && valids[0] == true {
392           msm.log.Info("success to send finality signature to celestia")
393         } else {
394           msm.log.Error("celestia: failed to validate proof",
395             zap.String("err", err.Error()),
396             zap.Any("valid", valids))
397         }
398       } else {
399         msm.log.Error("celestia: failed to get proof", zap.String("err", err.Error()))
400       }
401     } else {
402       msm.log.Info("celestia: blob submission failed; falling back to eth",
403         zap.String("err", err.Error()),
404         zap.Any("ids", ids),
405         zap.ByteString("commit", commit))
406     }
407   } else {
408     msm.log.Info("celestia: failed to create commitment", zap.String("err", err.Error()))
409   }
410 }
411
412 return nil
```

Listing 2.16: manta-fp/symbiotic-fp/mantastaking/msc.go

```
312   case <-time.After(msm.SubmissionRetryInterval):
313     // error will be returned if max retries have been reached
314     var err error
```

```
315     var ctx = context.Background()
316     err = msm.SubmitBatchFinalitySignatures(ctx, targetBlocks)
317     if err != nil {
318       msm.log.Debug(
319         "failed to submit finality signature to the consumer chain",
320         zap.String("address", msm.WalletAddr.String()),
321         zap.Uint32("current_failures", failedCycles),
322         zap.Uint64("target_start_height", targetBlocks[0].Height),
323         zap.Uint64("target_end_height", targetHeight),
324         zap.Error(err),
325       )
326
327       failedCycles++
328       if failedCycles > msm.MaxSubmissionRetries {
329         return fmt.Errorf("reached max failed cycles with err: %w", err)
330       }
331     } else {
332       // the signature has been successfully submitted
333       return nil
334     }
```

Listing 2.17: manta-fp/symbiotic-fp/mantastaking/msc.go

Impact The finality signature may be dropped, and the program continues processing new state roots. due to the improper error handling logic.

Suggestion Revise the logic accordingly.

2.1.8 Incorrect calculation of `signerApk` in the function `checkSignatures()`

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `checkSignatures()` in the contract `BLSApkRegistry` contains a critical flaw in its calculation of the aggregate public key (`signerApk`). Currently, the function attempts to compute the signer's aggregate key by subtracting non-signers' public keys from `currentApk`. However, the implementation erroneously overwrites `signerApk` in each iteration of the loop, resulting in only the last non-signer's key being subtracted. This incorrect calculation leads to faulty signature verification and may cause DoS.

```
128   function checkSignatures(bytes32 msgHash, uint256 referenceBlockNumber,
          FinalityNonSignerAndSignature memory params)
129       public
130       view
131       returns (StakeTotals memory, bytes32)
132   {
133       require(
134           referenceBlockNumber < uint32(block.number), "BLSSignatureChecker.checkSignatures:
                invalid reference block"
135       );
136       BN254.G1Point memory signerApk = BN254.G1Point(0, 0);
```

```
137       bytes32[] memory nonSignersPubkeyHashes;
138       if (params.nonSignerPubkeys.length > 0) {
139           nonSignersPubkeyHashes = new bytes32[](params.nonSignerPubkeys.length);
140           for (uint256 j = 0; j < params.nonSignerPubkeys.length; j++) {
141               nonSignersPubkeyHashes[j] = params.nonSignerPubkeys[j].hashG1Point();
142               signerApk = currentApk.plus(params.nonSignerPubkeys[j].negate());
143           }
144       } else {
145           signerApk = currentApk;
146       }
```

Listing 2.18: manta-fp-contracts/src/bls/BLSApkRegistry.sol

Impact Potential DoS due to the incorrect update for the variable `signerApk`.

Suggestion Revise the code logic accordingly.

2.1.9 Potential data race due to the improper use of the mutex

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `manager.go`, the function `resetState()` uses a mutex (i.e., `m.mu`) to protect access to the database (i.e., `m.db.SetLatestProcessedStateRoot`) and the variables (i.e., `m.windowPeriodStartTime` and `m.tickerController`). However, the function `processStateRoot()` is still able to access or modify protected data (i.e., line 521, 591 and 596 in the file `manager.go`) without using the mutex. As a result, this design may lead to race conditions in a concurrent execution environment.

```
461func (m *Manager) resetState(op *store.OutputProposed) {
462  m.mu.Lock()
463  defer m.mu.Unlock()
464
465  if err := m.db.SetLatestProcessedStateRoot(*op); err != nil {
466    m.log.Error("failed to set latest processed state root")
467  }
468  m.windowPeriodStartTime = op.Timestamp.Uint64()
469  m.tickerController = true
470
471}
```

Listing 2.19: manta-fp-aggregator/manager/manager.go

```
717  op, err := m.db.GetLatestProcessedStateRoot()
```

Listing 2.20: manta-fp-aggregator/manager/manager.go

```
371     op, err := m.db.GetLatestUnprocessedStateRoot(m.windowPeriodStartTime, m.
            outputSubmissionInterval)
372     if err != nil {
373       m.log.Error("failed to get latest unprocessed state root", "err", err)
374       continue
```

```
375     }
376
377     if op == nil {
378       if m.ethSynchronizer.HeaderTraversal.LastTraversedHeader().Time > m.windowPeriodStartTime+m
              .outputSubmissionInterval {
379         m.windowPeriodStartTime = m.windowPeriodStartTime + m.outputSubmissionInterval - 1
380         m.log.Warn("no more state root need to processed, skip", "next_start", m.
                windowPeriodStartTime)
381       }
382       m.log.Warn("no more state root need to processed", "start", m.windowPeriodStartTime, "end",
              m.windowPeriodStartTime+m.outputSubmissionInterval)
383       continue
384     }
```

Listing 2.21: manta-fp-aggregator/manager/manager.go

```
521 err = m.db.SetBatchStakeDetails(m.batchId, voteStateRoot, m.windowPeriodStartTime, op.Timestamp.
        Uint64())
```

Listing 2.22: manta-fp-aggregator/manager/manager.go

```
591 if err = m.db.DeleteStakeDetailsByTimestamp(m.babylonSynchronizer.StartTimestamp, m.
        windowPeriodStartTime); err != nil {
592   m.log.Error("failed to delete old stake details data", "err", err)
593   return err
594 }
595
596 if err = m.db.SetLatestProcessedStateRoot(*op); err != nil {
597   m.log.Error("failed to set latest processed state root", "err", err)
598   return err
599 }
```

Listing 2.23: manta-fp-aggregator/manager/manager.go

```
390     if m.isFirstBatch {
391       m.windowPeriodStartTime = op.Timestamp.Uint64()
392       if err = m.db.SetLatestProcessedStateRoot(*op); err != nil {
393         m.log.Error("failed to set latest processed state root", "err", err)
394         continue
395       }
396       m.isFirstBatch = false
397       continue
398     }
399
400     m.tickerController = false
401     m.metrics.RecordWindowPeriodStartTime(m.windowPeriodStartTime)
```

Listing 2.24: manta-fp-aggregator/manager/manager.go

```
364     if !m.tickerController {
365       m.log.Warn("the previous state root has not been processed yet")
366       continue
367     }
```

Listing 2.25: manta-fp-aggregator/manager/manager.go

Impact The improper use of the mutex may lead to race conditions in a concurrent execution environment.

Suggestion Revise the logic accordingly.

2.1.10 Lack of initialization for the field `MaxSubmissionRetries`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `msc.go`, the struct `MantaStakingMiddleware` includes a `MaxSubmissionRetries` field, which is used to limit the number of retry attempts for signature submissions. However, the field `MaxSubmissionRetries` remains uninitialized (i.e., remains as zero). As a result, the lack of initialization for the field `MaxSubmissionRetries` disables the retry mechanism for the signature submission.

```
57  MaxSubmissionRetries     uint32
```

Listing 2.26: manta-fp/symbiotic-fp/mantastaking/msc.go

```
328       if failedCycles > msm.MaxSubmissionRetries {
329         return fmt.Errorf("reached max failed cycles with err: %w", err)
330       }
```

Listing 2.27: manta-fp/symbiotic-fp/mantastaking/msc.go

Impact The lack of initialization for the field `MaxSubmissionRetries` of the `MantaStakingMiddleware` struct disables the retry mechanism for the signature submission.

Suggestion Initialize the field `MaxSubmissionRetries` of the `MantaStakingMiddleware` struct with a proper value.

2.1.11 Ungraceful shutdown due to the uninitialized field `cancel` of the struct `Node`

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `node.go`, the function `Stop()` invokes the pre-defined cancel function (i.e., `n.cancel()`) to exit the context. However, there is a lack of initialization for the field `cancel` of the struct `Node`, leading to a runtime panic when the function `Stop()` is invoked. As a result, the program can not shut down gracefully.

```
54  cancel   context.CancelFunc
```

Listing 2.28: manta-fp-aggregator/node/node.go

```
156func (n *Node) Stop(ctx context.Context) error {
157  n.cancel()
158  close(n.done)
159  n.wg.Wait()
160  n.babylonSynchronizer.Close()
161  n.celestiaSynchronizer.Close()
162  if n.metricsServer != nil {
163    if err := n.metricsServer.Close(); err != nil {
164      n.log.Error("failed to close metrics server", "err", err)
165    }
166  }
167  n.stopped.Store(true)
168  return nil
169}
```

Listing 2.29: manta-fp-aggregator/node/node.go

Impact Unfraceful shutdown due to the uninitialized field `cancel` of the struct `Node`.

Suggestion Initialize the field `cancel` of the struct `Node`.

2.1.12 Unsafe confirmation depth

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the files `eth_synchronizer.go` and `celestia_synchronizer.go`, the functions `NewEthSynchronizer()` and `NewCelestiaSynchronizer()` use a confirmation depth of zero when constructing header traverser (i.e., `headerTraversal`). This is unsafe due to the risk of chain reorganizations that the fetched latest block may later be replaced and is therefore not final. As a result, using an unsafe confirmation depth may lead to the storage of incorrect state roots, affecting the subsequent verifications.

```
40func NewEthSynchronizer(cfg *config.Config, db *store.Storage, ctx context.Context, logger log.
      Logger, shutdown context.CancelCauseFunc, contractEventChan chan store.ContractEvent, metricer
       metrics.Metricer) (*EthSynchronizer, error) {
41  client, err := node.DialEthClient(ctx, cfg.EthRpc)
42  if err != nil {
43    return nil, err
44  }
45
46  dbLatestHeader, err := db.GetEthScannedHeight()
47  if err != nil {
48    return nil, err
49  }
50  var fromHeader *types.Header
51  if dbLatestHeader != 0 {
52    logger.Info("eth: sync detected last indexed block", "number", dbLatestHeader)
53    header, err := client.BlockHeaderByNumber(big.NewInt(int64(dbLatestHeader)))
54    if err != nil {
55      logger.Error("failed to get eth block header", "height", dbLatestHeader)
```

```
56    }
57    fromHeader = header
58  } else if cfg.EthStartingHeight > 0 {
59    logger.Info("eth: no sync indexed state starting from supplied ethereum height", "height", cfg
          .EthStartingHeight)
60    header, err := client.BlockHeaderByNumber(big.NewInt(cfg.EthStartingHeight))
61    if err != nil {
62      return nil, fmt.Errorf("could not fetch eth starting block header: %w", err)
63    }
64    fromHeader = header
65  } else {
66    logger.Info("no ethereum block indexed state")
67  }
68
69  headerTraversal := node.NewEthHeaderTraversal(client, fromHeader, big.NewInt(0))
70
71  var contracts []common.Address
72  contracts = append(contracts, common.HexToAddress(cfg.Contracts.FrmContractAddress))
73  contracts = append(contracts, common.HexToAddress(cfg.Contracts.L2ooContractAddress))
74
75  resCtx, resCancel := context.WithCancel(context.Background())
76  return &EthSynchronizer{
77    HeaderTraversal: headerTraversal,
78    ethClient:       client,
79    latestHeader:    fromHeader,
80    db:              db,
81    blockStep:       cfg.EthBlockStep,
82    contracts:       contracts,
83    resourceCtx:     resCtx,
84    resourceCancel:  resCancel,
85    log:             logger,
86    contractEventChan: contractEventChan,
87    metrics:         metricer,
88    tasks: tasks.Group{HandleCrit: func(err error) {
89      shutdown(fmt.Errorf("critical error in eth synchronizer: %w", err))
90    }},
91  }, nil
92}
```

Listing 2.30: manta-fp-aggregator/synchronizer/eth_synchronizer.go

```
42func NewCelestiaSynchronizer(ctx context.Context, cfg *config.Config, db *store.Storage, shutdown
      context.CancelCauseFunc, logger log.Logger, authToken string, metricer metrics.Metricer) (*
      CelestiaSynchronizer, error) {
43  cli, err := client.NewClient(ctx, cfg.CelestiaConfig.DaRpc, authToken)
44  if err != nil {
45    return nil, err
46  }
47
48  nsBytes, err := hex.DecodeString(cfg.CelestiaConfig.Namespace)
49  if err != nil {
50    return nil, err
51  }
```

```
52  if len(nsBytes) != 10 {
53    return nil, errors.New("wrong namespace length")
54  }
55
56  namespace, err := share.NamespaceFromBytes(append(make([]byte, 19), nsBytes...))
57  if err != nil {
58    return nil, err
59  }
60
61  dbLatestHeader, err := db.GetCelestiaScannedHeight()
62  if err != nil {
63    return nil, err
64  }
65  var fromHeader *header.ExtendedHeader
66  if dbLatestHeader != 0 {
67    logger.Info("celestia: sync detected last indexed block", "number", dbLatestHeader)
68    header, err := cli.Header.GetByHeight(ctx, dbLatestHeader)
69    if err != nil {
70      logger.Error("failed to get celestia header", "height", dbLatestHeader)
71    }
72    fromHeader = header
73  } else if cfg.CelestiaStartingHeight > 0 {
74    logger.Info("celestia: no sync indexed state starting from supplied celestia height", "height"
          , cfg.CelestiaStartingHeight)
75    header, err := cli.Header.GetByHeight(ctx, uint64(cfg.CelestiaStartingHeight))
76    if err != nil {
77      return nil, fmt.Errorf("could not fetch celestia starting block header: %w", err)
78    }
79    fromHeader = header
80  } else {
81    logger.Info("no celestia block indexed state")
82  }
83
84  headerTraversal := node.NewCelestiaHeaderTraversal(cli, fromHeader, big.NewInt(0))
85
86  resCtx, resCancel := context.WithCancel(context.Background())
87  return &CelestiaSynchronizer{
88    client:         cli,
89    blockStep:      cfg.CelestiaBlockStep,
90    HeaderTraversal: headerTraversal,
91    LatestHeader:   fromHeader,
92    db:             db,
93    resourceCtx:    resCtx,
94    resourceCancel: resCancel,
95    log:            logger,
96    namespace:      namespace,
97    metrics:        metricer,
98    tasks: tasks.Group{HandleCrit: func(err error) {
99      shutdown(fmt.Errorf("critical error in celestia synchronizer: %w", err))
100   }},
101 }, nil
102}
```

Listing 2.31: manta-fp-aggregator/synchronizer/celestia_synchronizer.go

Impact Using a confirmation depth of zero may lead to the storage of incorrect state roots, affecting the subsequent verifications.

Suggestion Revise the logic accordingly.

2.1.13 Improper validation design for checking sync status

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the files `manager.go` and `node.go`, the verifying and signing processes begin only after both the Babylon and Celestia chains are synchronized, as determined by the function `checkSyncStatus()`. However, this design introduces a potential single point of failure. Specifically, if one of the chains experiences a network issue (e.g., delay or disconnection), the function `checkSyncStatus()` will not return `true`, preventing the verifying and signing processes from proceeding. As a result, the files `manager.go` and `node.go` may become stuck, and the state root cannot be verified.

```
386     if !m.checkSyncStatus(op) {
387       continue
388     }
```

Listing 2.32: manta-fp-aggregator/manager/manager.go

```
276     if !n.checkSyncStatus(requestBody.EndTimestamp) {
277       continue
278     }
```

Listing 2.33: manta-fp-aggregator/node/node.go

Impact The improper validation design may lead the files `manager.go` and `node.go` to become stuck, and the state root cannot be verified.

Suggestion Revise the logic accordingly.

2.1.14 Lack of validation for minimum valid signatures threshold

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The function `checkSignatures()` lacks validation for the minimum number of required valid signatures, which poses risks of insufficient authorization. The function may accept messages signed by fewer signers than required.

```
128   function checkSignatures(bytes32 msgHash, uint256 referenceBlockNumber,
          FinalityNonSignerAndSignature memory params)
129       public
130       view
131       returns (StakeTotals memory, bytes32)
132   {
133       require(
134           referenceBlockNumber < uint32(block.number), "BLSSignatureChecker.checkSignatures:
                invalid reference block"
135       );
136       BN254.G1Point memory signerApk = BN254.G1Point(0, 0);
137       bytes32[] memory nonSignersPubkeyHashes;
138       if (params.nonSignerPubkeys.length > 0) {
139           nonSignersPubkeyHashes = new bytes32[](params.nonSignerPubkeys.length);
140           for (uint256 j = 0; j < params.nonSignerPubkeys.length; j++) {
141               nonSignersPubkeyHashes[j] = params.nonSignerPubkeys[j].hashG1Point();
142               signerApk = currentApk.plus(params.nonSignerPubkeys[j].negate());
143           }
144       } else {
145           signerApk = currentApk;
146       }
147       (bool pairingSuccessful, bool signatureIsValid) =
148           trySignatureAndApkVerification(msgHash, signerApk, params.apkG2, params.sigma);
149       require(pairingSuccessful, "BLSSignatureChecker.checkSignatures: pairing precompile call
              failed");
150       require(signatureIsValid, "BLSSignatureChecker.checkSignatures: signature is invalid");
151
152       bytes32 signatoryRecordHash = keccak256(abi.encodePacked(referenceBlockNumber,
              nonSignersPubkeyHashes));
153
154       StakeTotals memory stakeTotals =
155           StakeTotals({totalBtcStaking: params.totalBtcStake, totalMantaStaking: params.
                totalMantaStake});
156
157       return (stakeTotals, signatoryRecordHash);
158   }
```

Listing 2.34: manta-fp-contracts/src/bls/BLSApkRegistry.sol

Impact Setting the finalization period seconds for `outputRoot` could be approved without sufficient consensus.

Suggestion Implement checks for the minimum number of required valid signatures.

2.1.15 Lack of proper implementations in the function `Close()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the files `eth_synchronizer.go` and `celestia_synchronizer.go`, the `Close()` function is responsible for closing the synchronization process. However, the function `Close()` only

returns `nil` without implementing any shutdown logic. As a result, the synchronization process is always forcefully terminated, leading to memory leaks. Moreover, the error handling logic on lines 288, 292, and 296 in the file `manager.go` cannot be triggered to output proper logs. This lack of proper implementations in the function `Close()` may result in resource leaks and unintended system behavior.

```
198func (syncer *EthSynchronizer) Close() error {
199  return nil
200}
```

Listing 2.35: manta-fp-aggregator/synchronizer/eth_synchronizer.go

```
201func (syncer *CelestiaSynchronizer) Close() error {
202  return nil
203}
```

Listing 2.36: manta-fp-aggregator/synchronizer/celestia_synchronizer.go

```
281func (m *Manager) Stop(ctx context.Context) error {
282  close(m.done)
283  if err := m.httpServer.Shutdown(ctx); err != nil {
284    m.log.Error("http server forced to shutdown", "err", err)
285    return err
286  }
287  if err := m.babylonSynchronizer.Close(); err != nil {
288    m.log.Error("babylon synchronizer server forced to shutdown", "err", err)
289    return err
290  }
291  if err := m.ethSynchronizer.Close(); err != nil {
292    m.log.Error("eth synchronizer server forced to shutdown", "err", err)
293    return err
294  }
```

Listing 2.37: manta-fp-aggregator/manager/manager.go

Impact This lack of proper implementations in the function `Close()` may result in resource leaks and unintended system behavior.

Suggestion Revise the logic accordingly.

2.1.16 Lack of closing the channel `stopChan` in the function `Stop()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `node.go`, the function `sign()` starts a goroutine to process the sign request received from the file `manager.go`. The goroutine leverages the channel `n.stopChan` to determine when to exit. However, the program lacks logic to close `n.stopChan` (i.e., via the invocation of `close(n.stopChan)`) in the function `Stop()`, potentially leading to the ungraceful shutdown and goroutine leaks. The similar issue exists in the function `ProcessMessage()` of the file `deal_msg.go` as well.

```
208func (n *Node) sign() {
209  defer n.wg.Done()
210
211  n.log.Info("start to sign message")
212
213  go func() {
214    defer func() {
215      n.log.Info("exit sign process")
216    }()
217    for {
218      select {
219      case <-n.stopChan:
220        return
221      case req := <-n.signRequestChan:
222        var resId = req.ID.(tdtypes.JSONRPCStringID).String()
223        n.log.Info(fmt.Sprintf("dealing resId (%s) ", resId))
224
225        var nodeSignRequest types.NodeSignRequest
226        rawMsg := json.RawMessage{}
227        nodeSignRequest.RequestBody = &rawMsg
228
229        if err := json.Unmarshal(req.Params, &nodeSignRequest); err != nil {
230          n.log.Error("failed to unmarshal ask request")
231          RpcResponse := tdtypes.NewRPCErrorResponse(req.ID, 201, "failed", err.Error())
232          if err := n.wsClient.SendMsg(RpcResponse); err != nil {
233            n.log.Error("failed to send msg to manager", "err", err)
234          }
235          continue
236        }
237        var requestBody types.SignMsgRequest
238        if err := json.Unmarshal(rawMsg, &requestBody); err != nil {
239          n.log.Error("failed to unmarshal asker's params request body")
240          RpcResponse := tdtypes.NewRPCErrorResponse(req.ID, 201, "failed", err.Error())
241          if err := n.wsClient.SendMsg(RpcResponse); err != nil {
242            n.log.Error("failed to send msg to manager", "err", err)
243          }
244          continue
245        }
246
247        if requestBody.StartTimestamp <= 0 || requestBody.EndTimestamp <= 0 {
248          n.log.Error("start timestamp and end timestamp must not be nil or negative")
249          RpcResponse := tdtypes.NewRPCErrorResponse(req.ID, 201, "failed", "start timestamp and
                 end timestamp must not be nil or negative")
250          if err := n.wsClient.SendMsg(RpcResponse); err != nil {
251            n.log.Error("failed to send msg to manager", "err", err)
252          }
253          continue
254        }
255
256        nodeSignRequest.RequestBody = requestBody
257
258        go n.handleSign(req.ID.(tdtypes.JSONRPCStringID), nodeSignRequest)
```

```
259     }
260   }
261 }()
262}
```

Listing 2.38: manta-fp-aggregator/node/node.go

```
156func (n *Node) Stop(ctx context.Context) error {
157  n.cancel()
158  close(n.done)
159  n.wg.Wait()
160  n.babylonSynchronizer.Close()
161  n.celestiaSynchronizer.Close()
162  if n.metricsServer != nil {
163    if err := n.metricsServer.Close(); err != nil {
164      n.log.Error("failed to close metrics server", "err", err)
165    }
166  }
167  n.stopped.Store(true)
168  return nil
169}
```

Listing 2.39: manta-fp-aggregator/node/node.go

```
 13func (n *Node) ProcessMessage() {
 14  n.log.Info("process websocket message")
 15  defer n.wg.Done()
 16  reqChan := make(chan tmtypes.RPCRequest)
 17  stopChan := make(chan struct{})
 18  if err := n.wsClient.RegisterResChannel(reqChan, stopChan); err != nil {
 19    n.log.Error("failed to register request channel with websocket client", "err", err)
 20    return
 21  }
 22
 23  go func() {
 24    defer func() {
 25      close(stopChan)
 26    }()
 27    for {
 28      select {
 29      case rpcReq := <-reqChan:
 30        reqId := rpcReq.ID.(tdtypes.JSONRPCStringID).String()
 31        n.log.Info(fmt.Sprintf("receive request method : %s", rpcReq.Method), "reqId", reqId)
 32        if rpcReq.Method == types.SignMsgBatch.String() {
 33          if err := n.writeChan(n.signRequestChan, rpcReq); err != nil {
 34            n.log.Error("failed to write msg to sign channel,channel blocked ", "err", err)
 35          }
 36        } else {
 37          n.log.Error(fmt.Sprintf("unknown rpc request method : %s ", rpcReq.Method))
 38        }
 39      }
 40
 41    }
```

```
42 }()
43}
```

Listing 2.40: manta-fp-aggregator/node/deal_msg.go

Impact The ungraceful shutdown and goroutine leaks due to the lack of closing the channel `stopChan` in the function `Stop()`.

Suggestion Close the channel `stopChan` in the function `Stop()`.

2.1.17 Inconsistent error handling logic

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the file `eth_synchronizer.go`, the function `NewEthSynchronizer()` fetches block headers using either `dbLatestHeader` or `cfg.ethStartingHeight` in two branches to construct the header traverser (i.e., `headerTraversal`). However, the error handling logic between two branches is inconsistent. Specifically, the former branch uses the nil value to construct header traverser when error is thrown in the invocation of the function `client.BlockHeaderByNumber()`, the latter one returns an error to break the invocation of the function `NewEthSynchronizer()` directly. As a result, this inconsistent error handling logic may break the intended design. The similar issue happens in the file `celestia_synchronizer.go` as well.

```
50 var fromHeader *types.Header
51 if dbLatestHeader != 0 {
52   logger.Info("eth: sync detected last indexed block", "number", dbLatestHeader)
53   header, err := client.BlockHeaderByNumber(big.NewInt(int64(dbLatestHeader)))
54   if err != nil {
55     logger.Error("failed to get eth block header", "height", dbLatestHeader)
56   }
57   fromHeader = header
58 } else if cfg.EthStartingHeight > 0 {
59   logger.Info("eth: no sync indexed state starting from supplied ethereum height", "height", cfg
         .EthStartingHeight)
60   header, err := client.BlockHeaderByNumber(big.NewInt(cfg.EthStartingHeight))
61   if err != nil {
62     return nil, fmt.Errorf("could not fetch eth starting block header: %w", err)
63   }
64   fromHeader = header
65 } else {
66   logger.Info("no ethereum block indexed state")
67 }
```

Listing 2.41: manta-fp-aggregator/synchronizer/eth_synchronizer.go

```
65 var fromHeader *header.ExtendedHeader
66 if dbLatestHeader != 0 {
67   logger.Info("celestia: sync detected last indexed block", "number", dbLatestHeader)
68   header, err := cli.Header.GetByHeight(ctx, dbLatestHeader)
69   if err != nil {
```

```
70      logger.Error("failed to get celestia header", "height", dbLatestHeader)
71    }
72    fromHeader = header
73  } else if cfg.CelestiaStartingHeight > 0 {
74    logger.Info("celestia: no sync indexed state starting from supplied celestia height", "height"
          , cfg.CelestiaStartingHeight)
75    header, err := cli.Header.GetByHeight(ctx, uint64(cfg.CelestiaStartingHeight))
76    if err != nil {
77      return nil, fmt.Errorf("could not fetch celestia starting block header: %w", err)
78    }
79    fromHeader = header
80  } else {
81    logger.Info("no celestia block indexed state")
82  }
```

Listing 2.42: manta-fp-aggregator/synchronizer/celestia_synchronizer.go

Impact The inconsistent error handling logic may break the intended design.

Suggestion Revise the logic accordingly.

2.1.18 Lack of proper resource cleanup

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the project, many tickers are instantiated but lack proper cleanup. Specifically, tickers are created without being explicitly stopped using `ticker.Stop()`. As a result, the failure to stop these tickers may lead to resource leaks over the long term. Below are some examples:

1. The ticker `tickerSyncer` in the function `Start()` of the file `celestia_synchronizer.go`.
2. The ticker `tickerSyncer` in the function `Start()` of the file `eth_synchronizer.go`.
3. The ticker `waitNodeTicker` in the function `Start()` of the file `manager.go`.
4. The ticker `fpTicker` in the function `work()` of the file `manager.go`.

```
104func (syncer *CelestiaSynchronizer) Start() error {
105  tickerSyncer := time.NewTicker(time.Second * 2)
```

Listing 2.43: manta-fp-aggregator/synchronizer/celestia_synchronizer.go

```
94func (syncer *EthSynchronizer) Start() error {
95  tickerSyncer := time.NewTicker(time.Second * 2)
```

Listing 2.44: manta-fp-aggregator/synchronizer/eth_synchronizer.go

```
351func (m *Manager) work() {
352  fpTicker := time.NewTicker(m.cfg.Manager.FPTimeout)
```

Listing 2.45: manta-fp-aggregator/manager/manager.go

```
221func (m *Manager) Start(ctx context.Context) error {
222  waitNodeTicker := time.NewTicker(5 * time.Second)
```

Listing 2.46: manta-fp-aggregator/manager/manager.go

Impact The failure to stop these tickers may lead to resource leaks over the long term.

Suggestion Revise the logic accordingly.

2.1.19 Fail to start the file `manager.go`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `Start()` of the file `manager.go`, it initializes the `m.windowPeriodStartTime` variable by querying the latest output index from the contract `L2OutputOracle` (via the function `L2OutputOracle.latestOutputIndex()`). However, when there are no proposed outputs (i.e., `l2Outputs.length == 0`), the invocation reverts, causing a fetch error in the function `Start()`. As a result, the file `manager.go` cannot start when there are no proposed outputs in the contract `L2OutputOracle`.

```
259 if err := m.getWindowPeriodStartTime(); err != nil {
260   m.log.Error("could not get window period start time", "err", err)
261   return err
262 }
```

Listing 2.47: manta-fp-aggregator/manager/manager.go

```
334   index, err := m.l2oo.LatestOutputIndex(cOpts)
335   if err != nil {
```

Listing 2.48: manta-fp-aggregator/manager/manager.go

```
293   function latestOutputIndex() external view returns (uint256) {
294       return l2Outputs.length - 1;
295   }
```

Listing 2.49: celestia-bedrock-manta/packages/contracts-bedrock/contracts/L1/L2OutputOracle.sol

Impact The file `manager.go` cannot start when there are no proposed outputs in the contract `L2OutputOracle`.

Suggestion Revise the logic accordingly.

2.1.20 Potential duplicate node registration leading to incorrect `currentApk` calculation

Severity Low

Status Fixed in `Version 2`

Introduced by Version 1

Description The function registerOperator() in the contract BLSApkRegistry lacks protection against duplicate registrations of the same node (operator address). Each registration adds the operator's public key to the current aggregated public key (i.e., currentApk) without checking whether the operator was previously registered. This could lead to incorrect currentApk calculation, due to the same public key being counted multiple times in currentApk, distorting the true aggregate value.

```
56  function registerOperator(address operator) public onlyFinalityRelayerManager {
57      (BN254.G1Point memory pubkey,) = getRegisteredPubkey(operator);
58
59      _processApkUpdate(pubkey);
60
61      emit OperatorAdded(operator, operatorToPubkeyHash[operator]);
62  }
```

Listing 2.50: manta-fp-contracts/src/bls/BLSApkRegistry.sol

```
187 function _processApkUpdate(BN254.G1Point memory point) internal {
188     BN254.G1Point memory newApk;
189
190     uint256 historyLength = apkHistory.length;
191     require(historyLength != 0, "BLSApkRegistry._processApkUpdate: quorum does not exist");
192
193     newApk = currentApk.plus(point);
194     currentApk = newApk;
```

Listing 2.51: manta-fp-contracts/src/bls/BLSApkRegistry.sol

Impact This could cause miscalculations in BLS signature verification that rely on accurate currentApk values.

Suggestion Implement a mapping to enforce one-time registration.

2.1.21 Lack of non zero check for _operatorName in the function registerOperator()

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description The function registerOperator() does not check whether _operatorName is a non-empty string, while the off-chain FP aggregator (i.e., manager.go) explicitly checks for empty names. This inconsistency could lead to invalid operator registrations where operators with empty names are recognized as not operators.

```
86  function registerOperator(
87      bytes memory _operatorPublicKey,
88      string memory _operatorName,
89      address _rewardAddress,
90      uint48 _commission
```

```
91    ) external override nonReentrant {
```

Listing 2.52: manta-staking-contracts/src/MantaStakingMiddleware.sol

```
754     if operator.OperatorName == "" {
755       m.log.Warn(fmt.Sprintf("node %s is not operator", sfs.SignRequests.SignAddress))
756       continue
```

Listing 2.53: manta-fp-aggregator/manager/manager.go

Impact This inconsistency could lead to invalid operator registrations where operators with empty names are recognized as not operators in the off-chain logic.

Suggestion Add a non zero check for `_operatorName` in the function `registerOperator()`.

2.1.22 Potential DoS due to the improper use of `ERC20` interfaces

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the contract `TokenDistributor`, the functions `claimToken()` and `withdrawToken()` transfer the `distributeToken` using `IERC20` interface. Specifically, it relies on the `transfer()` function, which returns a boolean value in the `IERC20` interface. However, this interface is incompatible with certain tokens (e.g., `USDT`) that do not return a value from their `transfer()` function. As a result, when such tokens are used as `distributeToken`, the functions will revert.

```
 99        bool success = IERC20(distributeToken).transfer(
100            _receiver,
101            claimableAmount
102        );
```

Listing 2.54: manta-staking-contracts/src/TokenDistributor.sol

```
186    function withdrawToken(
187        address _token,
188        address _account,
189        uint256 _amount
190    ) external onlyRole(DEFAULT_ADMIN_ROLE) {
191        bool success = IERC20(_token).transfer(_account, _amount);
192        require(success, "TokenDistributor: Failed to transfer token");
193    }
```

Listing 2.55: manta-staking-contracts/src/TokenDistributor.sol

Impact The functions `claimToken()` and `withdrawToken()` revert when the `distributeToken` is incompatible with the interface `IERC20`.

Suggestion Use the function `safeTransfer()` of the `SafeERC20` library.

2.1.23 Lack of checks for the registration status of nonSignerPubkeys

Severity Low

Status Fixed in Version 2

Introduced by Version 1

Description In the contract BLSApkRegistry, it assumes that all nodes corresponding to the list nonSignerPubkeys have been properly registered. However, there is no validation to confirm this registration status. If a node operator fails to register via the function registerOperator() but is still included in the list nonSignerPubkeys, it will lead to incorrect signerApk calculations. This occurs because the currentApk does not actually contain the unregistered nonSignerPubkeys components. This oversight could lead to incorrect signerApk calculation, which will finally cause DoS.

```
138       if (params.nonSignerPubkeys.length > 0) {
139           nonSignersPubkeyHashes = new bytes32[](params.nonSignerPubkeys.length);
140           for (uint256 j = 0; j < params.nonSignerPubkeys.length; j++) {
141               nonSignersPubkeyHashes[j] = params.nonSignerPubkeys[j].hashG1Point();
142               signerApk = currentApk.plus(params.nonSignerPubkeys[j].negate());
143           }
144       } else {
```

Listing 2.56: manta-fp-contracts/src/bls/BLSApkRegistry.sol

Impact This oversight could lead to incorrect signerApk calculation, which will finally cause DoS.

Suggestion Revise the logic accordingly.

2.1.24 Lack of checks for the inputs operator and pubkeyRegistrationMessageHash

Severity Low

Status Partially Fixed in Version 2

Introduced by Version 1

Description The function registerBLSPublicKey() does not check whether the msg.sender is the input operator. A malicious operator Alice can generate valid parameters and register it for another operator Bob, then operatorToPubkeyHash[Bob] will be set to a public key by Alice, thus preventing Bob from registering its own public key. Furthermore, the function registerBLSPublicKey() takes pubkeyRegistrationMessageHash as an input but fails to verify whether this hash was correctly generated by the function getPubkeyRegMessageHash() based on the input operator. This could potentially lead to a rogue-key attack risk.

```
71    function registerBLSPublicKey(
72        address operator,
73        PubkeyRegistrationParams calldata params,
74        BN254.G1Point calldata pubkeyRegistrationMessageHash
75    ) external returns (bytes32) {
76        require(
77            blsRegisterWhitelist[msg.sender],
```

```
78          "BLSApkRegistry.registerBLSPublicKey: this address have not permission to register bls
                key"
79      );
80
81      bytes32 pubkeyHash = BN254.hashG1Point(params.pubkeyG1);
82
83      require(pubkeyHash != ZERO_PK_HASH, "BLSApkRegistry.registerBLSPublicKey: cannot register
            zero pubkey");
84      require(
85          operatorToPubkeyHash[operator] == bytes32(0),
86          "BLSApkRegistry.registerBLSPublicKey: operator already registered pubkey"
87      );
88
89      require(
90          pubkeyHashToOperator[pubkeyHash] == address(0),
91          "BLSApkRegistry.registerBLSPublicKey: public key already registered"
92      );
93
94      uint256 gamma = uint256(
95          keccak256(
96              abi.encodePacked(
97                  params.pubkeyRegistrationSignature.X,
98                  params.pubkeyRegistrationSignature.Y,
99                  params.pubkeyG1.X,
100                 params.pubkeyG1.Y,
101                 params.pubkeyG2.X,
102                 params.pubkeyG2.Y,
103                 pubkeyRegistrationMessageHash.X,
104                 pubkeyRegistrationMessageHash.Y
105             )
106         )
107     ) % BN254.FR_MODULUS;
108
109     require(
110         BN254.pairing(
111             params.pubkeyRegistrationSignature.plus(params.pubkeyG1.scalar_mul(gamma)),
112             BN254.negGeneratorG2(),
113             pubkeyRegistrationMessageHash.plus(BN254.generatorG1().scalar_mul(gamma)),
114             params.pubkeyG2
115         ),
116         "BLSApkRegistry.registerBLSPublicKey: either the G1 signature is wrong, or G1 and G2
                private key do not match"
117     );
118
119     operatorToPubkey[operator] = params.pubkeyG1;
120     operatorToPubkeyHash[operator] = pubkeyHash;
121     pubkeyHashToOperator[pubkeyHash] = operator;
122
123     emit NewPubkeyRegistration(operator, params.pubkeyG1, params.pubkeyG2);
124
125     return pubkeyHash;
126 }
```

Listing 2.57: manta-fp-contracts/src/bls/BLSApkRegistry.sol

```
218   function getPubkeyRegMessageHash(address operator) public view returns (BN254.G1Point memory)
          {
219       return BN254.hashToG1(_hashTypedDataV4(keccak256(abi.encode(PUBKEY_REGISTRATION_TYPEHASH,
              operator))));
220   }
```

Listing 2.58: manta-fp-contracts/src/bls/BLSApkRegistry.sol

Impact The normal operator's registration process can be blocked and can potentially lead to a rogue-key attack risk.

Suggestion Add the check logic accordingly.

Clarification from BlockSec The project is aware of the potential risk and decides not to add the check for the input `pubkeyRegistrationMessageHash` of the function `registerBLSPublicKey()`, as only whitelisted operators are permitted to register. The risk of a rogue-key attack can be mitigated by the whitelist design.

2.1.25 Ungraceful shutdown due to the lack of invoking the function `m.wg.Wait()`

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the files `manager.go`, the function `Stop()` does not invoke the function `m.wg.Wait()` after closing the channel `m.done`. As a result, the lack of invoking the function `m.wg.Wait()` potentially leads to ungraceful shutdown.

```
281func (m *Manager) Stop(ctx context.Context) error {
282  close(m.done)
283  if err := m.httpServer.Shutdown(ctx); err != nil {
284    m.log.Error("http server forced to shutdown", "err", err)
285    return err
286  }
287  if err := m.babylonSynchronizer.Close(); err != nil {
288    m.log.Error("babylon synchronizer server forced to shutdown", "err", err)
289    return err
290  }
291  if err := m.ethSynchronizer.Close(); err != nil {
292    m.log.Error("eth synchronizer server forced to shutdown", "err", err)
293    return err
294  }
295  if err := m.celestiaSynchronizer.Close(); err != nil {
296    m.log.Error("celestia synchronizer server forced to shutdown", "err", err)
297    return err
298  }
299  if m.metricsServer != nil {
300    if err := m.metricsServer.Close(); err != nil {
301      m.log.Error("failed to close metrics server", "err", err)
```

```
302   }
303 }
304 m.stopped.Store(true)
305 m.log.Info("Server exiting")
306 return nil
307}
```

Listing 2.59: manta-fp-aggregator/manager/manager.go

Impact The lack of invoking the function `m.wg.Wait()` potentially leads to ungraceful shutdown.

Suggestion Invoke the function `m.wg.Wait()` in the function `Stop()`.

2.2 Recommendation

2.2.1 Use struct to unmarshal http response body

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the following code segments, the HTTP response body is unmarshaled using type assertion without sufficient error handling. To improve robustness and readability, it is recommended to define a dedicated struct to unmarshal the HTTP response body.

```
649 var result map[string]interface{}
650 if err := json.Unmarshal(body, &result); err != nil {
651   msm.log.Error("Error parsing JSON response:", zap.Error(err))
652   return nil, err
653 }
654
655 var totalStaked = big.NewInt(0)
656 if data, exists := result["data"]; exists {
657   if vaultUpdates, exists := data.(map[string]interface{})["vaultUpdates"]; exists {
658     if len(vaultUpdates.([]interface{})) > 0 {
659       vaultTotalActiveStaked := vaultUpdates.([]interface{})[0].(map[string]interface{})["
              vaultTotalActiveStaked"]
660       totalStaked, _ = new(big.Int).SetString(vaultTotalActiveStaked.(string), 10)
661       msm.log.Info(fmt.Sprintf("operator %s vaultTotalActiveStaked: %s", operator,
              vaultTotalActiveStaked))
662     } else {
663       msm.log.Warn(fmt.Sprintf("operator %s no vault updates found", operator))
664     }
665   } else {
666     msm.log.Warn(fmt.Sprintf("operator %s no vaultUpdates field found in response data",
            operator))
667   }
668 } else {
669   msm.log.Warn(fmt.Sprintf("operator %s no data field found in JSON response", operator))
670 }
```

Listing 2.60: manta-fp/symbiotic-fp/mantastaking/msc.go

```
496 var result map[string]interface{}
497 if err := json.Unmarshal(body, &result); err != nil {
498   n.log.Error("Error parsing JSON response:", "err", err)
499   return nil, err
500 }
501
502 var totalStaked = big.NewInt(0)
503 if data, exists := result["data"]; exists {
504   if vaultUpdates, exists := data.(map[string]interface{})["vaultUpdates"]; exists {
505     if len(vaultUpdates.([]interface{})) > 0 {
506       vaultTotalActiveStaked := vaultUpdates.([]interface{})[0].(map[string]interface{})["
              vaultTotalActiveStaked"]
507       totalStaked, _ = new(big.Int).SetString(vaultTotalActiveStaked.(string), 10)
508       n.log.Info(fmt.Sprintf("operator %s vaultTotalActiveStaked: %s", operator,
              vaultTotalActiveStaked))
509     } else {
510       n.log.Warn(fmt.Sprintf("operator %s no vault updates found", operator))
511     }
512   } else {
513     n.log.Warn(fmt.Sprintf("operator %s no vaultUpdates field found in response data", operator)
          )
514   }
515 } else {
516   n.log.Warn(fmt.Sprintf("operator %s no data field found in JSON response", operator))
517 }
```

Listing 2.61: manta-fp-aggregator/node/node.go

```
838 var totalStaked = big.NewInt(0)
839 if data, exists := result["data"]; exists {
840   if vaultUpdates, exists := data.(map[string]interface{})["vaultUpdates"]; exists {
841     if len(vaultUpdates.([]interface{})) > 0 {
842       vaultTotalActiveStaked := vaultUpdates.([]interface{})[0].(map[string]interface{})["
              vaultTotalActiveStaked"]
843       totalStaked, _ = new(big.Int).SetString(vaultTotalActiveStaked.(string), 10)
844       m.log.Info(fmt.Sprintf("operator %s vaultTotalActiveStaked: %s", operator,
              vaultTotalActiveStaked))
845     } else {
846       m.log.Warn(fmt.Sprintf("operator %s no vault updates found", operator))
847     }
848   } else {
849     m.log.Warn(fmt.Sprintf("operator %s no vaultUpdates field found in response data", operator)
          )
850   }
851 } else {
852   m.log.Warn(fmt.Sprintf("operator %s no data field found in JSON response", operator))
853 }
```

Listing 2.62: manta-fp-aggregator/manager/manager.go

Suggestion Use struct to unmarshal the HTTP response body.

2.2.2 Redundant Code

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There are several unused or redundant variables, functions, and validations. It is recommended to remove them for better code readability. Specifically, the following code should be removed or revised.

1. The validation for the length of the variable `targetBlocks` is redundant.

```
301 if len(targetBlocks) == 0 {
302   return fmt.Errorf("cannot send signatures for empty blocks")
303 }
```

Listing 2.63: manta-fp/symbiotic-fp/mantastaking/msc.go

2. The following fields are redundant or unused.

```
61 signTimeout       time.Duration
62 waitScanInterval  time.Duration
```

Listing 2.64: manta-fp-aggregator/node/node.go

```
71 metrics       metrics.Metricer
```

Listing 2.65: manta-fp-aggregator/node/node.go

```
30 startHeight     *big.Int
31 confirmationDepth *big.Int
32 resourceCtx     context.Context
33 resourceCancel  context.CancelFunc
```

Listing 2.66: manta-fp-aggregator/synchronizer/eth_synchronizer.go

```
32 startHeight     *big.Int
33 confirmationDepth *big.Int
34 resourceCtx     context.Context
35 resourceCancel  context.CancelFunc
```

Listing 2.67: manta-fp-aggregator/synchronizer/celestia_synchronizer.go

1. The `handleSymbioticSign()` function in the file `node.go` and the `IsMaxPriorityFeePerGasNotFoundError()` function in the file `msc.go` are unused.

```
343func (n *Node) handleSymbioticSign(resId tdtypes.JSONRPCStringID, req types.NodeSignRequest) error
        {
344  var err error
345  var bSign *sign.Signature
346  requestBody := req.RequestBody.(types.SignMsgRequest)
347  bSign, err = n.SignMessage(requestBody)
348  if bSign != nil {
349    signResponse := types.SignMsgResponse{
350      G2Point:  n.keyPairs.GetPubKeyG2().Serialize(),
351      Signature: bSign.Serialize(),
352      Vote:     uint8(common2.AgreeVote),
```

```
353   }
354   RpcResponse := tdtypes.NewRPCSuccessResponse(resId, signResponse)
355   n.log.Info("node agree the msg, start to send response to finality manager")
356
357   err = n.wsClient.SendMsg(RpcResponse)
358   if err != nil {
359     n.log.Error("failed to sendMsg to finality manager", "err", err)
360     return err
361   } else {
362     n.log.Info("send sign response to finality manager successfully ")
363     return nil
364   }
365 }
366 return nil
367}
```

Listing 2.68: manta-fp-aggregator/node/node.go

```
450func (msm *MantaStakingMiddleware) IsMaxPriorityFeePerGasNotFoundError(err error) bool {
451  return strings.Contains(
452    err.Error(), common2.ErrMaxPriorityFeePerGasNotFound.Error(),
453  )
454}
```

Listing 2.69: manta-fp/symbiotic-fp/mantastaking/msc.go

1. The function `SignMessage()` in the file `node.go`, always returns an error of nil. Therefore, there is no need to catch the error thrown when invoking the function `SignMessage()`.

```
369func (n *Node) SignMessage(requestBody types.SignMsgRequest) (*sign.Signature, error) {
370  var bSign *sign.Signature
371  bSign = n.keyPairs.SignMessage(crypto.Keccak256Hash(common.Hex2Bytes(requestBody.StateRoot)))
372  n.log.Info("success to sign SubmitFinalitySignatureMsg", "signature", bSign.String())
373
374  return bSign, nil
375}
```

Listing 2.70: manta-fp-aggregator/node/node.go

```
300       bSign, err = n.SignMessage(requestBody)
```

Listing 2.71: manta-fp-aggregator/node/node.go

```
347  bSign, err = n.SignMessage(requestBody)
```

Listing 2.72: manta-fp-aggregator/node/node.go

5.The modifier `operatorNotPaused` in contract `MantaStakingMiddleware` is unused.

```
46    modifier operatorNotPaused(address operator) {
47        require(!operators[operator].paused, "MantaStakingMiddleware: operator is paused");
48        _;
49    }
```

Listing 2.73: manta-staking-contracts/src/MantaStakingMiddleware.sol

Suggestion Remove or revise the redundant code accordingly.

2.2.3 Non zero address checks

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the functions `initialize()`, `withdrawToken()`, and `claimToken()`, several address variables (e.g., `_disputeGameFactory`) are not checked to ensure they are not zero. It is recommended to add such checks to prevent potential mis-operations.

```
38  function initialize(address _initialOwner, address _finalityRelayerManager, address
        _relayerManager)
39      external
40      initializer
41  {
42      _transferOwnership(_initialOwner);
43      finalityRelayerManager = _finalityRelayerManager;
44      relayerManager = _relayerManager;
45      _initializeApk();
46  }
```

Listing 2.74: manta-fp-contracts/src/bls/BLSApkRegistry.sol

```
39  function initialize(
40      address _initialOwner,
41      bool _isDisputeGameFactory,
42      address _blsApkRegistry,
43      address _l2OutputOracle,
44      address _disputeGameFactory,
45      address _operatorWhitelistManager
46  ) external initializer {
47      _transferOwnership(_initialOwner);
48      blsApkRegistry = IBLSApkRegistry(_blsApkRegistry);
49      l2OutputOracle = _l2OutputOracle;
50      disputeGameFactory = _disputeGameFactory;
51      isDisputeGameFactory = _isDisputeGameFactory;
52      operatorWhitelistManager = _operatorWhitelistManager;
53
54      confirmBatchId = 0;
55      TARGET_MANTA = 1000000 * 10e17;
56      TARGET_BITCOIN = 1000 * 10e7;
57  }
```

Listing 2.75: manta-fp-contracts/src/core/FinalityRelayerManager.sol

```
65  function claimToken(
66      address _receiver,
67      bytes32 _ownerAddressDigest,
68      uint256 _totalTokenRewardAmount,
69      uint8 _v,
70      bytes32 _r,
71      bytes32 _s
72  ) public whenNotPaused nonReentrant {
73      require(
```

```
74          _totalTokenRewardAmount > claimedAmount[_ownerAddressDigest],
75          "TokenDistributor: Already claimed"
76      );
77
78      address signer = _recoverSigner(
79          _receiver,
80          _ownerAddressDigest,
81          _totalTokenRewardAmount,
82          _v,
83          _r,
84          _s
85      );
```

Listing 2.76: manta-staking-contracts/src/TokenDistributor.sol

```
186   function withdrawToken(
187       address _token,
188       address _account,
189       uint256 _amount
190   ) external onlyRole(DEFAULT_ADMIN_ROLE) {
191       bool success = IERC20(_token).transfer(_account, _amount);
192       require(success, "TokenDistributor: Failed to transfer token");
193   }
```

Listing 2.77: manta-staking-contracts/src/TokenDistributor.sol

Suggestion Add non-zero address checks accordingly.

2.2.4 Handle the error in the function `handleSign()`

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In the function `handleSign()` of the file `node.go`, the error returned by the function `getMaxSignStateRoot()` is not properly handled. It is recommended to handle the returned error for better code readability.

```
280     maxSignStateRoot, err := n.getMaxSignStateRoot(requestBody)
281     if maxSignStateRoot != requestBody.StateRoot {
```

Listing 2.78: manta-fp-aggregator/node/node.go

Suggestion Handle the error in the function `handleSign()`.

2.3 Note

2.3.1 The slashing mechanism has not yet been implemented

Introduced by `Version 1`

Description The audited version of the code does not implement the slashing mechanism. This feature must be correctly implemented in a future update to ensure protocol security.

2.3.2 Potential centralization risks

Introduced by Version 1

Description In this project, several privileged roles (e.g., DEFAULT_ADMIN_ROLE) can conduct sensitive operations, which introduces potential centralization risks. For example, the role DEFAULT_ADMIN_ROLE can withdraw tokens of the contract TokenDistributor through the function withdrawToken(). If the private keys of the privileged accounts are lost or maliciously exploited, it could pose a significant risk to the protocol.

Feedback from the project The project stated that they will use a multisig wallet for role management to mitigate potential risks during the early stage and plan to transition role control to governance in the future.

2.3.3 Security audit assumptions on BLS signatures

Introduced by Version 1

Description The current security assessment of the protocol operates under the critical assumption that the BLS cryptographic signature scheme used in the contract BLSApkRegistry is fully trusted. This foundational premise directly impacts all subsequent security evaluations and risk assessments conducted within the audit scope.


BOOST WEB3 THROUGH NEXT-GENERATION SECURITY & USABILITY INNOVATIONS