Security Audit Report for Stratos Chain and Stratos Decentralized Storage (SDS)

Date: June 30, 2023

Version: 1.0

Contact: contact@blocksec.com

BlockSec Audit Team

Digitally signed by BlockSec Audit Team
Date: 2023.07.10 15:08:52 +08'00'

Contents

Report Manifest

Item	Description
Client	Stratos Network
Target	Stratos Chain and Stratos Decentralized Storage (SDS)

Version History

Version	Date	Description
1.0	June 30, 2023	First Release

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 5 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	Project
Language	Golang
Approach	Semi-automatic and manual verification

This audit primarily focuses on the Stratos Chain and Stratos Decentralized Storage (SDS), both of which are components of **Stratos**, a decentralized data architecture. Stratos provides scalable, reliable, and self-balanced storage, database, and computation networks, creating a robust foundation for data processing. The architecture of Stratos is divided into three distinct components:

1. *Stratos Chain*. This custom blockchain is based on the Cosmos-SDK and is responsible for defining various messages and implementing corresponding handlers to manage nodes and reward distribution within the network. By forking a custom Ethermint implementation, the Stratos Chain achieves full EVM compatibility.
2. *SP* (Meta Nodes). Within the Stratos Network, there are two node types: Meta Nodes (or SP in legacy terminology) and Resource Nodes (or SDS in legacy terminology). Meta Nodes are management nodes that connect storage nodes to the Stratos Chain and are responsible for volume reporting for reward distribution.
3. *SDS* (Storage Nodes). These lower-end nodes provide the actual storage for the entire network and form a P2P network to ensure high availability.

In this audit, only two of the three components, the Stratos Chain and Stratos Decentralized Storage (SDS), will be covered, with the SP being outside of scope [1]. Furthermore, it is important to note that this audit concentrates solely on the project's security aspects, while the project developers are responsible for ensuring functionality correctness.

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.

Project	Version	Commit Hash
Stratos Chain[2]	`Version 1`	`78e7317d24d69ad57b8f22f508e43446c25b1e30`
	`Version 2`	`9fb7f3d317859ada55a401cf4091f2af74e3cd48`
	`Version 3`	`9e234d20082a4fc213d8631cf8a8a400de4acff6`
SDS[3]	`Version 1`	`b94280b2b3b4823dccd0675418b50bd6814c00b0`
	`Version 2`	`a76c910d3cd97d1e595954323024f086198cc950`
	`Version 3`	`99ac9b69048b6cb0adabde664d547b8b3df72da2`

[1] Currently, the SP is not open-sourced, and Stratos Network are managing the Meta Nodes.

[2] https://github.com/stratosnet/stratos-chain

[3] https://github.com/stratosnet/sds

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 target project, 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 the project.

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 Golang 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 the project source code with automatic code analyzers, and then manually verify (reject or confirm) the issues reported by them.
- **Semantic Analysis** We study the business logic of the project 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 Software Security

* Reentrancy
* DoS
* Access control
* Data handling and data flow
* Exception handling
* Untrusted external call and control flow
* Initialization consistency
* Events operation
* Error-prone randomness
* Improper use of the proxy system

1.3.2 DeFi Security

* Semantic consistency
* Functionality consistency
* Permission management
* Business logic
* Token operation
* Emergency mechanism
* Oracle security
* Whitelist and blacklist
* Economic impact
* Batch transfer

1.3.3 NFT Security

* Duplicated item
* Verification of the token receiver
* Off-chain metadata security

1.3.4 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 [4] and Common Weakness Enumeration [5]. The overall *severity* of the risk is determined by *likelihood* and *impact*. 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.

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 four 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.

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

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

Table 1.1: Vulnerability Severity Classification

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

- **Fixed** The item has been confirmed and fixed by the client.

Chapter 2 Findings

In total, we find **eighteen** potential issues.

- High Risk: 12
- Medium Risk: 4
- Low Risk: 2

ID	Severity	Description	Category	Status
1	High	Insufficient access controls for privileged messages	Software Security	Fixed
2	High	Conflict logic in the `volumeReportRequestHandlerFn` function	Software Security	Fixed
3	Medium	Unchecked epoch field in the volume report	Software Security	Fixed
4	Low	Inconsistent token denoms	Software Security	Fixed
5	High	Incorrect `selfdestruct` logic in the EVM module	Software Security	Fixed
6	High	Complex and unstable logic in the `EndBlock` of the pot module	Software Security	Fixed
7	Medium	Deletion in iteration	Software Security	Fixed
8	Medium	Ignored error in reward distribution	Software Security	Fixed
9	High	Potential partial state write if `EndBlocker` panics	Software Security	Fixed
10	High	Potential concurrent-unsafe usage of a global variable	Software Security	Fixed
11	High	Potential loss of unbonding stake due to address overwriting	Software Security	Fixed
12	High	Potential locking of staked tokens if the creation vote fails	Software Security	Fixed
13	Low	Unremoved vote pool when the meta node is unbonded	Software Security	Fixed
14	High	Unverified message source	DeFi Security	Fixed
15	High	Unverified response messages	DeFi Security	Fixed
16	High	`ReqUploadFileSlice` allows arbitrary file writing	DeFi Security	Fixed
17	Medium	Potential DoS risk due to the absence of timeouts in message receiving and sending processes	DeFi Security	Fixed
18	High	Ignored error in authentication process	DeFi Security	Fixed

The details are provided in the following sections. Specifically, the findings of Stratos Chain and Stratos Decentralized Storage (SDS) are detailed in Section 2.1 and Section 2.2, respectively.

2.1 Stratos Chain

2.1.1 Insufficient access controls for privileged messages

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description Within the Stratos ecosystem, the chain serves as the on-chain ledger for all storage nodes and metadata nodes across the network, and it provides economic incentives to these nodes. Nodes interact with the chain via *messages*. In the context of Cosmos-SDK, all messages must be signed by a specific address (or a group of addresses) through the `GetSigners` method associated with the message. However, a prevalent issue in Cosmos-SDK-based chains is that the address used for signing the message is not employed in the message handler, meaning that the message sender's address is not involved in the access control process.

Given this observation, the following issues arise concerning the access control of these messages:

1. `MsgRemoveResourceNode`. This message requires signing by the `OwnerAddress` specified within it. However, the handler for the `MsgRemoveResourceNode` does not validate the relationship between the `OwnerAddress` and the `resourceNode` (i.e., whether the `OwnerAddress` is indeed the owner of the `resourceNode`). Consequently, any account can delete a resource node without the appropriate authorization.

```
228    func (msg MsgRemoveResourceNode) GetSigners() []sdk.AccAddress {
229        addr, err := sdk.AccAddressFromBech32(msg.OwnerAddress)
230        if err != nil {
231            panic(err)
232        }
233        return []sdk.AccAddress{addr.Bytes()}
234    }
```

Listing 2.1: x/register/types/msg.go

```
132        func (k msgServer) HandleMsgRemoveResourceNode(goCtx context.Context, msg *types.
               MsgRemoveResourceNode) (*types.MsgRemoveResourceNodeResponse, error) {
133            ctx := sdk.UnwrapSDKContext(goCtx)
134            p2pAddress, err := stratos.SdsAddressFromBech32(msg.ResourceNodeAddress)
135            if err != nil {
136                return &types.MsgRemoveResourceNodeResponse{}, sdkerrors.Wrap(types.
                       ErrInvalidNetworkAddr, err.Error())
137            }
138            resourceNode, found := k.GetResourceNode(ctx, p2pAddress)
139            if !found {
140                return nil, types.ErrNoResourceNodeFound
141            }
142            if resourceNode.GetStatus() == stakingtypes.Unbonding {
143                return nil, types.ErrUnbondingNode
144            }
145
146            unbondingStake := k.GetUnbondingNodeBalance(ctx, p2pAddress)
147            availableStake := resourceNode.Tokens.Sub(unbondingStake)
148            if availableStake.LTE(sdk.ZeroInt()) {
149                return nil, types.ErrInsufficientBalance
150            }
151            ...
```

Listing 2.2: x/register/keeper/msg_server.go

2. `MsgRemoveMetaNode`. This message is required to be signed by the `OwnerAddress`. However, the handler for the `MsgRemoveMetaNode` fails to validate the relationship between the owner and the `metaNode`. As a consequence, any account can delete a meta node without obtaining the appropriate authorization.

```
277     func (msg MsgRemoveMetaNode) GetSigners() []sdk.AccAddress {
278         addr, err := sdk.AccAddressFromBech32(msg.OwnerAddress)
279         if err != nil {
280             panic(err)
281         }
282         return []sdk.AccAddress{addr.Bytes()}
283     }
```

Listing 2.3: x/register/types/msg.go

```
176     func (k msgServer) HandleMsgRemoveMetaNode(goCtx context.Context, msg *types.
            MsgRemoveMetaNode) (*types.MsgRemoveMetaNodeResponse, error) {
177         ctx := sdk.UnwrapSDKContext(goCtx)
178         p2pAddress, err := stratos.SdsAddressFromBech32(msg.MetaNodeAddress)
179         if err != nil {
180             return &types.MsgRemoveMetaNodeResponse{}, sdkerrors.Wrap(types.
                    ErrInvalidNetworkAddr, err.Error())
181         }
182         metaNode, found := k.GetMetaNode(ctx, p2pAddress)
183         if !found {
184             return nil, types.ErrNoMetaNodeFound
185         }
186
187         if metaNode.GetStatus() == stakingtypes.Unbonding {
188             return nil, types.ErrUnbondingNode
189         }
190
191         unbondingStake := k.GetUnbondingNodeBalance(ctx, p2pAddress)
192         availableStake := metaNode.Tokens.Sub(unbondingStake)
193         if availableStake.LTE(sdk.ZeroInt()) {
194             return nil, types.ErrInsufficientBalance
195         }
196         ...
```

Listing 2.4: x/register/keeper/msg_server.go

3. `MsgUpdateEffectiveStake`. This message requires signing by the `ReporterOwner` specified within it. However, the handler for the `MsgUpdateEffectiveStake` message in the register module fails to validate the relationship between the `ReporterOwner` array and the `Reporters` array. Consequently, any account can alter the effective stake of any resource node, potentially causing a significant impact on the reward distribution process.

```
662     func (m MsgUpdateEffectiveStake) GetSigners() []sdk.AccAddress {
663         var addrs []sdk.AccAddress
664         for _, owner := range m.ReporterOwner {
665             reporterOwner, err := sdk.AccAddressFromBech32(owner)
666             if err != nil {
667                 panic(err)
```

```
668             }
669             addrs = append(addrs, reporterOwner)
670         }
671         return addrs
672     }
```

Listing 2.5: x/register/types/msg.go

```
353     func (k msgServer) HandleMsgUpdateEffectiveStake(goCtx context.Context, msg *types.
            MsgUpdateEffectiveStake) (*types.MsgUpdateEffectiveStakeResponse, error) {
354         ctx := sdk.UnwrapSDKContext(goCtx)
355
356         for _, reporter := range msg.Reporters {
357             reporterSdsAddr, err := stratos.SdsAddressFromBech32(reporter)
358             if err != nil {
359                 return &types.MsgUpdateEffectiveStakeResponse{}, sdkerrors.Wrap(sdkerrors.
                        ErrInvalidAddress, err.Error())
360             }
361             if !(k.IsMetaNode(ctx, reporterSdsAddr)) {
362                 return &types.MsgUpdateEffectiveStakeResponse{}, sdkerrors.Wrap(sdkerrors.
                        ErrInvalidAddress, "MsgUpdateEffectiveStake is not sent by a meta node
                        ")
363             }
364         }
365         ...
```

Listing 2.6: x/register/keeper/msg_server.go

4. `MsgMetaNodeRegistrationVote`. This message requires signing by the `VoterOwnerAddress` specified within it. However, the handler for the `MsgMetaNodeRegistrationVote` message in the register module omits the validation of the relationship between the `VoterNetworkAddress` and `VoterOwnerAddress` specified in the message (the `VoterOwnerAddress` should be the owner of the meta node identified by the `VoterNetworkAddress`). Consequently, any account can cast a vote on behalf of a meta node, potentially leading to vote manipulation and undermining the integrity of the voting process.

```
598     func (msg MsgMetaNodeRegistrationVote) GetSigners() []sdk.AccAddress {
599         addr, err := sdk.AccAddressFromBech32(msg.VoterOwnerAddress)
600         if err != nil {
601             panic(err)
602         }
603         return []sdk.AccAddress{addr.Bytes()}
604     }
```

Listing 2.7: x/register/types/msg.go

```
220     func (k msgServer) HandleMsgMetaNodeRegistrationVote(goCtx context.Context, msg *
            types.MsgMetaNodeRegistrationVote) (*types.MsgMetaNodeRegistrationVoteResponse,
            error) {
221         ctx := sdk.UnwrapSDKContext(goCtx)
222
223         candidateNetworkAddress, err := stratos.SdsAddressFromBech32(msg.
                CandidateNetworkAddress)
224         if err != nil {
```

```
225             return &types.MsgMetaNodeRegistrationVoteResponse{}, sdkerrors.Wrap(types.
                    ErrInvalidCandidateNetworkAddr, err.Error())
226         }
227
228         nodeToApprove, found := k.GetMetaNode(ctx, candidateNetworkAddress)
229         if !found {
230             return nil, types.ErrNoMetaNodeFound
231         }
232         if nodeToApprove.OwnerAddress != msg.CandidateOwnerAddress {
233             return nil, types.ErrInvalidOwnerAddr
234         }
235
236         voterNetworkAddress, err := stratos.SdsAddressFromBech32(msg.VoterNetworkAddress)
237         if err != nil {
238             return &types.MsgMetaNodeRegistrationVoteResponse{}, sdkerrors.Wrap(types.
                    ErrInvalidVoterNetworkAddr, err.Error())
239         }
240         voter, found := k.GetMetaNode(ctx, voterNetworkAddress)
241         if !found {
242             return nil, types.ErrInvalidVoterAddr
243         }
244         ...
```

Listing 2.8: x/register/keeper/msg_server.go

5. `MsgVolumeReport`. This message requires signing by the `ReporterOwner` specified within it. However, the handler fails to verify the relationship between the `ReporterOwner` and `Reporter` fields in the message. According to the design, the `ReporterOwner` field should be the owner of the meta node represented by the `Reporter` field. However, the current implementation does not enforce this relationship, enabling any account to impersonate a meta node and submit volume reports on its behalf.

```
49      func (msg MsgVolumeReport) GetSigners() []sdk.AccAddress {
50          var addrs []sdk.AccAddress
51          reporterOwner, err := sdk.AccAddressFromBech32(msg.ReporterOwner)
52          if err != nil {
53              panic(err)
54          }
55          addrs = append(addrs, reporterOwner)
56          return addrs
57      }
```

Listing 2.9: x/pot/types/msg.go

```
30      func (k msgServer) HandleMsgVolumeReport(goCtx context.Context, msg *types.
            MsgVolumeReport) (*types.MsgVolumeReportResponse, error) {
31          ctx := sdk.UnwrapSDKContext(goCtx)
32          reporter, err := stratos.SdsAddressFromBech32(msg.Reporter)
33          if err != nil {
34              return &types.MsgVolumeReportResponse{}, sdkerrors.Wrap(types.
                    ErrInvalidAddress, err.Error())
35          }
36          if !(k.IsMetaNode(ctx, reporter)) {
```

```
37              return &types.MsgVolumeReportResponse{}, sdkerrors.Wrap(types.
                    ErrInvalidAddress, "Volume report is not sent by a superior peer")
38          }
39          ...
```

Listing 2.10: x/pot/keeper/msg_server.go

6. `MsgSlashingResourceNode`. This message requires signing by the addresses in the `ReporterOwner` array. However, the handler fails to verify the relationship between the `ReporterOwner` and `Reporter` specified in the message. The `ReporterOwner` should correspond to the owner of the meta node associated with the `Reporter`, but this verification is absent. As a result, anyone can initiate a slash against any resource node in the name of a meta node, regardless of ownership. Furthermore, another issue in the handling of this message is the lack of a check for the length of the `ReporterOwner` array. It is deemed acceptable for there to be no addresses (i.e., `len(ReporterOwner) == 0`) signing this message.

```
321     func (m MsgSlashingResourceNode) GetSigners() []sdk.AccAddress {
322         var addrs []sdk.AccAddress
323         for _, owner := range m.ReporterOwner {
324             reporterOwner, err := sdk.AccAddressFromBech32(owner)
325             if err != nil {
326                 panic(err)
327             }
328             addrs = append(addrs, reporterOwner)
329         }
330         return addrs
331     }
```

Listing 2.11: x/pot/types/msg.go

```
178     func (k msgServer) HandleMsgSlashingResourceNode(goCtx context.Context, msg *types.
            MsgSlashingResourceNode) (*types.MsgSlashingResourceNodeResponse, error) {
179         ctx := sdk.UnwrapSDKContext(goCtx)
180
181         for _, reporter := range msg.Reporters {
182             reporterSdsAddr, err := stratos.SdsAddressFromBech32(reporter)
183             if err != nil {
184                 return &types.MsgSlashingResourceNodeResponse{}, sdkerrors.Wrap(types.
                        ErrInvalidAddress, err.Error())
185             }
186             if !(k.IsMetaNode(ctx, reporterSdsAddr)) {
187                 return &types.MsgSlashingResourceNodeResponse{}, sdkerrors.Wrap(sdkerrors.
                        ErrInvalidAddress, "Slashing msg is not sent by a meta node")
188             }
189         }
190         ...
```

Listing 2.12: x/pot/keeper/msg_server.go

7. `MsgFileUpload`. This message requires signing by the return value of `GetFrom()`. However, despite the validation of the "from" address signature, there is no permission control implemented. Furthermore, the "from" address is not incorporated into the message processing logic.

```
42      func (msg MsgFileUpload) GetSigners() []sdk.AccAddress {
43          accAddr, err := sdk.AccAddressFromBech32(msg.GetFrom())
44          if err != nil {
45              panic(err)
46          }
47          return []sdk.AccAddress{accAddr.Bytes()}
48      }
```

Listing 2.13: x/sds/types/msg.go

```
25      func (k msgServer) HandleMsgFileUpload(c context.Context, msg *types.MsgFileUpload)
            (*types.MsgFileUploadResponse, error) {
26          ctx := sdk.UnwrapSDKContext(c)
27
28          reporter, err := stratos.SdsAddressFromBech32(msg.GetReporter())
29          if err != nil {
30              return &types.MsgFileUploadResponse{}, sdkerrors.Wrap(sdkerrors.
                    ErrInvalidAddress, err.Error())
31          }
32
33          if _, found := k.registerKeeper.GetMetaNode(ctx, reporter); found == false {
34              return nil, sdkerrors.Wrapf(sdkerrors.ErrUnauthorized, "Reporter %s isn't an
                    SP node", msg.GetReporter())
35          }
36          height := sdk.NewInt(ctx.BlockHeight())
37          heightByteArr, _ := height.MarshalJSON()
38          var heightReEncoded sdk.Int
39          err = heightReEncoded.UnmarshalJSON(heightByteArr)
40          if err != nil {
41              return &types.MsgFileUploadResponse{}, sdkerrors.Wrap(sdkerrors.
                    ErrJSONUnmarshal, err.Error())
42          }
43
44          fileInfo := types.NewFileInfo(&heightReEncoded, msg.Reporter, msg.Uploader)
45          fileHashByte := []byte(msg.FileHash)
46          k.SetFileHash(ctx, fileHashByte, fileInfo)
47          ...
48      }
```

Listing 2.14: x/sds/keeper/msg_server.go

Impact Insufficient access controls enable malicious users to exploit privileged functions.

Suggestion Add sanity checks for privileged messages.

2.1.2 Conflict logic in the `volumeReportRequestHandlerFn` function

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description In `x/pot/client/rest/tx.go`, the `volumeReportRequestHandlerFn` function assembles a new volume report message struct with an empty `BLSSignatureInfo` struct as the `BLSSignature` field. Subsequently, the message struct is validated through the `ValidateBasic` method in the following line.

```
103    msg := types.NewMsgVolumeReport(walletVolumes, reporter, epoch, reportReference, reporterOwner
           , types.BLSSignatureInfo{})
104    err = msg.ValidateBasic()
```

Listing 2.15: x/pot/client/rest/tx.go

However, the `ValidateBasic` function returns an error if the `BLSSignature` field of the message is empty. Consequently, the `ValidateBasic` will fail in the above code snippet.

```
103    if len(msg.BLSSignature.Signature) == 0 {
104    return ErrBLSSignatureInvalid
105  }
```

Listing 2.16: x/pot/client/rest/tx.go

Impact The REST interface for sending the `MsgVolumeReport` message cannot be used.

Suggestion Fix the incorrect logic.

2.1.3 Unchecked epoch field in the volume report

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The original reward distribution implementation in Stratos can be summarized as follows: The `MsgVolumeReport` is submitted on-chain to record the traffic information for the entire Stratos network. At the end of the next block, all rewards are processed and transferred to the owner addresses of all nodes. If volume reports are present in a consecutive range of blocks, the volume report handling is postponed until an empty block without volume reports is encountered. The next handling of the volume reports starts from the last handled epoch (last block number) to the most recent reported epoch.

The current epoch number is provided by the `MsgVolumeReport`. If the epoch given by the volume report is `Int64.Max`, the `unhandledEpoch` is set to this value. In the subsequent call to `distributePotReward`, the `matureStartEpoch` in `rewardMatrueAndSubSlashing` is set to `-Int64.Max`. The following loop executes `Uint64.Max` times, and the previously calculated `matureTotalReward` is also included in the calculation.

```
200    // Iteration for mature rewards/slashing of all nodes
201    func (k Keeper) rewardMatureAndSubSlashing(ctx sdk.Context, currentEpoch sdk.Int) (
           totalSlashed sdk.Coins) {
202
203        matureStartEpoch := k.GetLastReportedEpoch(ctx).Int64() + 1
204        matureEndEpoch := currentEpoch.Int64()
205
206        totalSlashed = sdk.Coins{}
207
208        for i := matureStartEpoch; i <= matureEndEpoch; i++ {
209            k.IteratorIndividualReward(ctx, sdk.NewInt(i), func(walletAddress sdk.AccAddress,
                   individualReward types.Reward) (stop bool) {
```

```
210             oldMatureTotal := k.GetMatureTotalReward(ctx, walletAddress)
211             oldImmatureTotal := k.GetImmatureTotalReward(ctx, walletAddress)
212             immatureToMature := individualReward.RewardFromMiningPool.Add(individualReward.
                    RewardFromTrafficPool...)
213
214             //deduct slashing amount from upcoming mature reward, don't need to deduct slashing
                     from immatureTotal & individual
215             remaining, deducted := k.registerKeeper.DeductSlashing(ctx, walletAddress,
                    immatureToMature, k.RewardDenom(ctx))
216             totalSlashed = totalSlashed.Add(deducted...)
217
218             matureTotal := oldMatureTotal.Add(remaining...)
219             immatureTotal := oldImmatureTotal.Sub(immatureToMature)
220
221             k.SetMatureTotalReward(ctx, walletAddress, matureTotal)
222             k.SetImmatureTotalReward(ctx, walletAddress, immatureTotal)
223             return false
224         })
225     }
226     return totalSlashed
227 }
```

Listing 2.17: x/pot/keeper/distribute.go

Impact If the epoch specified in `MsgVolumeReport` is incorrect, it can trigger overflow and causes the duplication of reward accounting.

Suggestion Refactor the reward process logic.

2.1.4 Inconsistent token denoms

Severity Low

Status Fixed in `Version 2`

Introduced by `Version 1`

Description At the beginning of the `HandleMsgPrepay` function, it verifies whether the token for `DefaultBondDenom` is a `SendEnabledCoin`; however, the token utilized later is the `BondDenom`.

```
67  func (k msgServer) HandleMsgPrepay(c context.Context, msg *types.MsgPrepay) (*types.
        MsgPrepayResponse, error) {
68      ctx := sdk.UnwrapSDKContext(c)
69
70      if k.bankKeeper.IsSendEnabledCoin(ctx, sdk.NewCoin(types.DefaultBondDenom, sdk.OneInt()))
            == false {
71          return &types.MsgPrepayResponse{}, sdkerrors.ErrInvalidCoins
72      }
```

Listing 2.18: x/sds/keeper/msg_server.go

```
123 func (k Keeper) Prepay(ctx sdk.Context, sender sdk.AccAddress, coins sdk.Coins) (sdk.Int, error) {
124   for _, coin := range coins {
125     hasCoin := k.bankKeeper.HasBalance(ctx, sender, coin)
126     if !hasCoin {
```

```
127     return sdk.ZeroInt(), sdkerrors.Wrapf(sdkerrors.ErrInsufficientFunds, "Insufficient balance
            in the acc %s", sender.String())
128   }
129 }
130
131 prepay := coins.AmountOf(k.BondDenom(ctx))
132 return k.purchaseNozAndSubCoins(ctx, sender, prepay)
133}
```

Listing 2.19: x/sds/keeper/keeper.go

Impact Inconsistent usage of different denoms may introduce potential problems.

Suggestion Refactor the prepay handling logic.

2.1.5 Incorrect `selfdestruct` logic in the EVM module

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description In the EVM module, if a contract triggers the `selfdestruct` logic, then the `DeleteAccount` function would remove the code by the code hash. It means that other contracts with the same code hash would be unusable, causing a DoS attack [1].

```
67 func (k *Keeper) DeleteAccount(ctx sdk.Context, addr common.Address) error {
68   cosmosAddr := sdk.AccAddress(addr.Bytes())
69   acct := k.accountKeeper.GetAccount(ctx, cosmosAddr)
70   if acct == nil {
71     return nil
72   }
73
74   // NOTE: only Ethereum accounts (contracts) can be selfdestructed
75   ethAcct, ok := acct.(stratos.EthAccountI)
76   if !ok {
77     return sdkerrors.Wrapf(types.ErrInvalidAccount, "type %T, address %s", acct, addr)
78   }
79
80   // clear balance
81   if err := k.SetBalance(ctx, addr, new(big.Int)); err != nil {
82     return err
83   }
84
85   // remove code
86   codeHashBz := ethAcct.GetCodeHash().Bytes()
87   if !bytes.Equal(codeHashBz, types.EmptyCodeHash) {
88     k.SetCode(ctx, codeHashBz, nil)
89   }
```

Listing 2.20: x/evm/keeper/statedb.go

[1] https://github.com/evmos/ethermint/security/advisories/GHSA-f92v-grc2-w2fg

Impact DoS attacks can be initiated due to improper handling of code removal within the EVM module.

Suggestion Refactor the `selfdestruct` handling logic.

2.1.6 Complex and unstable logic in the `EndBlock` of the pot module

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description The pot module implements the logic for the `EndBlock` ABCI interface in the `EndBlocker` function. In this function, the `DistributePotReward` function implements a complex and unstable logic for the reward distribution process of the entire ecosystem, which may lead to the following issues:

1. There is no limit on the reward distribution process, meaning that the reward distribution (or the `EndBlock`) can be excessively lengthy. As the `EndBlock` ABCI interface has no gas limits, this situation can cause the entire chain to hang and lead to other potential problems.
2. Any panic in the `EndBlock` results in the entire node crashing, thus jeopardizing the entire network. For instance, in the `DistributePotReward` function, there is a `GetTotalConsumedNoz` function that aggregates the volume reports. If the sum overflows, the `EndBlocker` would panic. Coupled with the missing access control issue of the `MsgVolumeReport` (see Issue 2.1.1), a malicious actor can shut down the entire network.

```
18// EndBlocker called every block, process inflation, update validator set.
19func EndBlocker(ctx sdk.Context, req abci.RequestEndBlock, k keeper.Keeper) []abci.ValidatorUpdate
       {
20
21  // Do not distribute rewards until the next block
22  if !k.GetIsReadyToDistributeReward(ctx) && k.GetUnhandledEpoch(ctx).GT(sdk.ZeroInt()) {
23    k.SetIsReadyToDistributeReward(ctx, true)
24    return []abci.ValidatorUpdate{}
25  }
26
27  walletVolumes, found := k.GetUnhandledReport(ctx)
28  if !found {
29    return []abci.ValidatorUpdate{}
30  }
31  epoch := k.GetUnhandledEpoch(ctx)
32  logger := k.Logger(ctx)
33
34  //distribute POT reward
35  _, err := k.DistributePotReward(ctx, walletVolumes.Volumes, epoch)
36  if err != nil {
37    logger.Error("An error occurred while distributing the reward. ", "ErrMsg", err.Error())
38  }
39
40  k.SetUnhandledReport(ctx, types.WalletVolumes{})
41  k.SetUnhandledEpoch(ctx, sdk.ZeroInt())
42
43  return []abci.ValidatorUpdate{}
44}
```

Listing 2.21: x/pot/abci.go

Impact Malicious actors can launch DoS attacks or cause the shutdown of the entire network.

Suggestion Refactor the `EndBlock` and reward distribution logic.

2.1.7 Deletion in iteration

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description There is an issue with the code implementation in the `DequeueAllMatureUBDQueue` function of the register module. During the iterator traversal process, keys are being deleted, which can lead to unknown potential problems. Deletion during iteration is generally not recommended in software development.

```
184 func (k Keeper) DequeueAllMatureUBDQueue(ctx sdk.Context, currTime time.Time) (matureUnbonds []
        string) {
185   store := ctx.KVStore(k.storeKey)
186   // gets an iterator for all timeslices from time 0 until the current Blockheader time
187   unbondingTimesliceIterator := k.UnbondingNodeQueueIterator(ctx, ctx.BlockHeader().Time)
188   defer unbondingTimesliceIterator.Close()
189
190   for ; unbondingTimesliceIterator.Valid(); unbondingTimesliceIterator.Next() {
191     timeSliceVal := stratos.SdsAddresses{} //[]stratos.SdsAddress{}
192     value := unbondingTimesliceIterator.Value()
193     k.cdc.MustUnmarshalLengthPrefixed(value, &timeSliceVal)
194     timeSlice := timeSliceVal.GetAddresses()
195     matureUnbonds = append(matureUnbonds, timeSlice...)
196     store.Delete(unbondingTimesliceIterator.Key())
197   }
198   ctx.Logger().Debug(fmt.Sprintf("DequeueAllMatureUBDQueue, %d matured unbonding nodes detected",
          len(matureUnbonds)))
199   return matureUnbonds
200 }
```

Listing 2.22: x/register/keeper.go

Impact Deletion in iteration may cause potential problems.

Suggestion Revise the corresponding code logic.

2.1.8 Ignored error in reward distribution

Severity Medium

Status Partially Fixed in `Version 3`

Introduced by `Version 1`

Description There is an issue with the code implementation in the `EndBlocker` ABCI function of the pot module. Specifically, the code fails to check for errors returned by the `DistributePotReward` function, which

can result in the clearing of all volume reports, even when reward distribution fails. This could also result in the loss of a portion of the rewards. However, processing the error is also incorrect. It is recommended to implement the reward distribution logic as a regular message.

```
18 // EndBlocker called every block, process inflation, update validator set.
19 func EndBlocker(ctx sdk.Context, req abci.RequestEndBlock, k keeper.Keeper) []abci.
       ValidatorUpdate {
20
21   // Do not distribute rewards until the next block
22   if !k.GetIsReadyToDistributeReward(ctx) && k.GetUnhandledEpoch(ctx).GT(sdk.ZeroInt()) {
23     k.SetIsReadyToDistributeReward(ctx, true)
24     return []abci.ValidatorUpdate{}
25   }
26
27   walletVolumes, found := k.GetUnhandledReport(ctx)
28   if !found {
29     return []abci.ValidatorUpdate{}
30   }
31   epoch := k.GetUnhandledEpoch(ctx)
32   logger := k.Logger(ctx)
33
34   //distribute POT reward
35   _, err := k.DistributePotReward(ctx, walletVolumes.Volumes, epoch)
36   if err != nil {
37     logger.Error("An error occurred while distributing the reward. ", "ErrMsg", err.Error())
38   }
39
40   k.SetUnhandledReport(ctx, types.WalletVolumes{})
41   k.SetUnhandledEpoch(ctx, sdk.ZeroInt())
42
43   return []abci.ValidatorUpdate{}
44 }
```

Listing 2.23: x/pot/abci.go

Impact The ignored error can cause potential problems.

Suggestion Refactor the `EndBlock` logic in the pot module.

Additional Comments The original logic for the `DistributePotReward` function is complex and prone to error. In `Version 3`, this portion of the logic has been entirely refactored, ensuring that errors can occur only due to transfer failures.

2.1.9 Potential partial state write if `EndBlocker` panics

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In `Version 2` of the codebase, a fix has been applied to the `EndBlock` of the pot module that introduces a recovery mechanism to recover from panics and avoid halting the entire chain. However, this mechanism is susceptible to partially written states. Consequently, if the function panics while processing

volume report requests, the processed requests are saved to the state, while the remaining states are not. For instance, if the function panics while invoking the `DistributePotReward` function.

```
18 // EndBlocker called every block, process inflation, update validator set.
19 func EndBlocker(ctx sdk.Context, req abci.RequestEndBlock, k keeper.Keeper) []abci.
       ValidatorUpdate {
20
21   logger := k.Logger(ctx)
22
23   defer func() {
24     if r := recover(); r != nil {
25       logger.Error("Recovered from panic. ", "ErrMsg", r)
26     }
27   }()
28
29   // Do not distribute rewards until the next block
30   if !k.GetIsReadyToDistribute(ctx) && k.GetUnDistributedEpoch(ctx).GT(sdk.ZeroInt()) {
31     k.SetIsReadyToDistribute(ctx, true)
32   } else {
33     // Start distribute reward if report found
34     walletVolumes, found := k.GetUnDistributedReport(ctx)
35     if found {
36       epoch := k.GetUnDistributedEpoch(ctx)
37
38       //distribute POT reward
39       err := k.DistributePotReward(ctx, walletVolumes.Volumes, epoch)
40       if err != nil {
41         logger.Error("An error occurred while distributing the reward. ", "ErrMsg", err.Error())
42       }
43
44       // reset undistributed info after distribution
45       k.SetUnDistributedReport(ctx, types.WalletVolumes{})
46       k.SetUnDistributedEpoch(ctx, sdk.ZeroInt())
47     }
48   }
49
50   // mature reward
51   err := k.RewardMatureAndSubSlashing(ctx)
52   if err != nil {
53     logger.Error("An error occurred while distributing the reward. ", "ErrMsg", err.Error())
54   }
55
56   return []abci.ValidatorUpdate{}
57 }
```

Listing 2.24: x/pot/abci.go

Impact Potential partial state write if `EndBlocker` panics.

Suggestion Refactor the `EndBlocker` logic in the pot module.

2.1.10 Potential concurrent-unsafe usage of a global variable

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the updated version of the keeper for the register module, there is a (module) global variable `metaNodeBitMapIndexCacheStatus` that indicates the current status of the `metaNodeBitMapIndexCache` within the keeper. Additionally, a set of functions are provided to load and refresh the cache, as demonstrated below.

```
18 func (k Keeper) AddMetaNodeToBitMapIdxCache(networkAddr stratos.SdsAddress) {
19   k.metaNodeBitMapIndexCache[networkAddr.String()] = -1
20   metaNodeBitMapIndexCacheStatus = types.CACHE_DIRTY
21 }
22
23 func (k Keeper) RemoveMetaNodeFromBitMapIdxCache(networkAddr stratos.SdsAddress) {
24   delete(k.metaNodeBitMapIndexCache, networkAddr.String())
25   metaNodeBitMapIndexCacheStatus = types.CACHE_DIRTY
26 }
27
28 func (k Keeper) UpdateMetaNodeBitMapIdxCache(ctx sdk.Context) {
29   if metaNodeBitMapIndexCacheStatus == types.CACHE_NOT_DIRTY {
30     return
31   }
32   if len(k.metaNodeBitMapIndexCache) == 0 {
33     k.ReloadMetaNodeBitMapIdxCache(ctx)
34     return
35   }
36
37   keys := make([]string, 0)
38   for key, _ := range k.metaNodeBitMapIndexCache {
39     keys = append(keys, key)
40   }
41   sort.Slice(keys, func(i, j int) bool {
42     return keys[i] < keys[j]
43   })
44   for index, key := range keys {
45     k.metaNodeBitMapIndexCache[key] = index
46   }
47   metaNodeBitMapIndexCacheStatus = types.CACHE_NOT_DIRTY
48 }
49
50 func (k Keeper) ReloadMetaNodeBitMapIdxCache(ctx sdk.Context) {
51   if metaNodeBitMapIndexCacheStatus == types.CACHE_NOT_DIRTY {
52     return
53   }
```

Listing 2.25: x/register/meta_node.go

However, this implementation is not concurrent-safe. Due to the concurrent native features of Golang and Cosmos-SDK, all usage of global variables must be concurrent-safe. One potential race condition is the concurrent usage of the `CheckTx` and `DeliverTx` callbacks. For instance, consider a scenario where there is a `CheckTx` for adding a meta node and a `DeliverTx` for file upload. In this case, the file upload request might read an incorrect index of the reporter because `AddMetaNodeToBitMapIdxCache` has written

to the index, but the `metaNodeBitMapIndexCacheStatus` has not been set. Consequently, an incorrect value would be returned by the `GetMetaNodeBitMapIndex` function. Under specific conditions, this could lead to inconsistency and halt the entire chain.

Impact Concurrent access to the module's global cached index can lead to race conditions and produce unexpected consequences.

Suggestion Implement an `RWLock` for the cached index.

2.1.11 Potential loss of unbonding stake due to address overwriting

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description The unbonding stakes of both meta nodes and resource nodes are maintained in a structure called `UnbondingNode` (UBD), which contains a list of unbonding record entries. The UBD is stored with the network address serving as the key for the map. However, when a resource node is created using a network address that already exists for a meta node, the UBD entry for the meta node is replaced by the newly arriving resource node. As a result, all unbonding stakes of the meta node would be lost.

```
55  // SetUnbondingNode sets the unbonding node
56  func (k Keeper) SetUnbondingNode(ctx sdk.Context, ubd types.UnbondingNode) {
57    store := ctx.KVStore(k.storeKey)
58    bz := k.cdc.MustMarshalLengthPrefixed(&ubd)
59    networkAddr, err := stratos.SdsAddressFromBech32(ubd.GetNetworkAddr())
60    if err != nil {
61      return
62    }
63    key := types.GetUBDNodeKey(networkAddr)
64    store.Set(key, bz)
65  }
```

Listing 2.26: x/register/keeper/store.go

Impact Unbonding stakes can be lost due to the potential overwriting of the same address.

Suggestion Ensure the uniqueness of addresses for all nodes.

2.1.12 Potential locking of staked tokens if the creation vote fails

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description A newly created meta node is designated as suspended and becomes active if it passes the registration vote before the expiration time. There are two ways to remove a meta node: first, through the `MsgRemoveMetaNode` message, and second, via the `MsgUpdateMetaNodeStake` message. However, both methods invoke the `UnbondMetaNode` function, which returns immediately when the meta node is suspended. Consequently, if new meta nodes fail to pass the registration vote, they cannot be removed, and the staked assets become permanently locked in the pool.

```
339 func (k Keeper) UnbondMetaNode(ctx sdk.Context, metaNode types.MetaNode, amt sdk.Int,
340 ) (ozoneLimitChange sdk.Int, unbondingMatureTime time.Time, err error) {
341   if metaNode.GetStatus() == stakingtypes.Unbonding {
342     return sdk.ZeroInt(), time.Time{}, types.ErrUnbondingNode
343   }
344   networkAddr, err := stratos.SdsAddressFromBech32(metaNode.GetNetworkAddress())
345   if err != nil {
346     return sdk.ZeroInt(), time.Time{}, errors.New("invalid network address")
347   }
348   ownerAddr, err := sdk.AccAddressFromBech32(metaNode.GetOwnerAddress())
349   if err != nil {
350     return sdk.ZeroInt(), time.Time{}, errors.New("invalid wallet address")
351   }
352   ownerAcc := k.accountKeeper.GetAccount(ctx, ownerAddr)
353   if ownerAcc == nil {
354     return sdk.ZeroInt(), time.Time{}, types.ErrNoOwnerAccountFound
355   }
356   // suspended node cannot be unbonded (avoid dup stake decrease with node suspension)
357   if metaNode.Suspend {
358     return sdk.ZeroInt(), time.Time{}, types.ErrInvalidSuspensionStatForUnbondNode
359   }
360   // check if node_token - unbonding_token > amt_to_unbond
361   unbondingStake := k.GetUnbondingNodeBalance(ctx, networkAddr)
362   availableStake := metaNode.Tokens.Sub(unbondingStake)
363   if availableStake.LT(amt) {
364     return sdk.ZeroInt(), time.Time{}, types.ErrInsufficientBalance
365   }
366   if k.HasMaxUnbondingNodeEntries(ctx, networkAddr) {
367     return sdk.ZeroInt(), time.Time{}, types.ErrMaxUnbondingNodeEntries
368   }
369   unbondingMatureTime = calcUnbondingMatureTime(ctx, metaNode.Status, metaNode.CreationTime, k.
          UnbondingThreasholdTime(ctx), k.UnbondingCompletionTime(ctx))
370   bondDenom := k.GetParams(ctx).BondDenom
371   coin := sdk.NewCoin(bondDenom, amt)
372   if metaNode.GetStatus() == stakingtypes.Bonded {
373     // to prevent remainingOzoneLimit from being negative value
374     if !k.IsUnbondable(ctx, amt) {
375       return sdk.ZeroInt(), time.Time{}, types.ErrInsufficientBalance
376     }
377     // transfer the node tokens to the not bonded pool
378     k.bondedToUnbonding(ctx, metaNode, true, coin)
379     // adjust ozone limit
380     ozoneLimitChange = k.DecreaseOzoneLimitBySubtractStake(ctx, amt)
381   }
382   // change node status to unbonding if unbonding all available tokens
383   if amt.Equal(availableStake) {
384     metaNode.Status = stakingtypes.Unbonding
385     // decrease meta node count
386     v := k.GetBondedMetaNodeCnt(ctx)
387     count := v.Sub(sdk.NewInt(1))
388     k.SetBondedMetaNodeCnt(ctx, count)
389     // set meta node
```

```
390     k.SetMetaNode(ctx, metaNode)
391   }
392   // Set the unbonding mature time and completion height appropriately
393   unbondingNode := k.SetUnbondingNodeEntry(ctx, networkAddr, true, ctx.BlockHeight(),
          unbondingMatureTime, amt)
394   // Add to unbonding node queue
395   k.InsertUnbondingNodeQueue(ctx, unbondingNode, unbondingMatureTime)
396   ctx.Logger().Info("Unbonding meta node " + unbondingNode.String() + "\n after mature time" +
          unbondingMatureTime.String())
397   return ozoneLimitChange, unbondingMatureTime, nil
398 }
```

Listing 2.27: x/register/keeper/keeper.go

Impact Unbonding stakes can be lost if the creation vote fails or expires for meta nodes.

Suggestion Implement a refunding logic for meta nodes if the creation vote fails.

2.1.13 Unremoved vote pool when the meta node is unbonded

Severity Low

Status Fixed in `Version 3`

Introduced by `Version 2`

Description During the meta node removal process, the corresponding vote pool is not removed. Consequently, if a meta node is removed, the owner can create another meta node with the same address, and the `HandleVoteForMetaNodeRegistration` call would directly succeed due to the legacy vote pool not being removed. This issue can be mitigated by setting an expiration time (`votingValidityPeriodInSecond`).

```
319 func (k Keeper) HandleVoteForMetaNodeRegistration(ctx sdk.Context, candidateNetworkAddr stratos.
        SdsAddress, candidateOwnerAddr sdk.AccAddress,
320 opinion types.VoteOpinion, voterNetworkAddr stratos.SdsAddress, voterOwnerAddr sdk.AccAddress) (
        nodeStatus stakingtypes.BondStatus, err error) {
321
322 // voter validation
323 voterNode, found := k.GetMetaNode(ctx, voterNetworkAddr)
324 if !found {
325   return stakingtypes.Unbonded, types.ErrNoVoterMetaNodeFound
326 }
327 if voterNode.GetOwnerAddress() != voterOwnerAddr.String() {
328   return stakingtypes.Unbonded, types.ErrInvalidVoterOwnerAddr
329 }
330 if voterNode.Status != stakingtypes.Bonded || voterNode.Suspend {
331   return stakingtypes.Unbonded, types.ErrInvalidVoterStatus
332 }
333
334 // candidate validation
335 candidateNode, found := k.GetMetaNode(ctx, candidateNetworkAddr)
336 if !found {
337   return stakingtypes.Unbonded, types.ErrNoCandidateMetaNodeFound
338 }
339 if candidateNode.GetOwnerAddress() != candidateOwnerAddr.String() {
340   return candidateNode.Status, types.ErrInvalidCandidateOwnerAddr
```

```
341  }
342
343  // vote validation and handle voting
344  votePool, found := k.GetMetaNodeRegistrationVotePool(ctx, candidateNetworkAddr)
345  if !found {
346    return stakingtypes.Unbonded, types.ErrNoRegistrationVotePoolFound
347  }
348  if votePool.ExpireTime.Before(ctx.BlockHeader().Time) {
349    return stakingtypes.Unbonded, types.ErrVoteExpired
350  }
351  if hasStringValue(votePool.ApproveList, voterNetworkAddr.String()) || hasStringValue(votePool.
         RejectList, voterNetworkAddr.String()) {
352    return stakingtypes.Unbonded, types.ErrDuplicateVoting
353  }
```

Listing 2.28: x/register/meta_node.go

Impact If the owner of a removed meta node registers again, the creation vote will pass immediately.

Suggestion Eliminate the vote pool after the creation vote is completed for meta nodes.

2.2 Stratos Decentralized Storage (SDS)

2.2.1 Unverified message source

Severity High

Status Fixed in `Version 3`

Introduced by `Version 1`

Description An issue exists in the SDS implementation where message handlers cannot verify the source of messages, including the identity and type of the peer. This arises because the connection component does not provide any information about the peer to the handlers. As a result, all messages lack appropriate authentication and authorization mechanisms. For instance, the `RspGetPPList` message is assumed to originate from the SP node or be routed by other PP nodes; however, this assumption is neither verified nor accurate in the context of the P2P network.

```
14   func RspGetPPList(ctx context.Context, conn core.WriteCloser) {
15       var target protos.RspGetPPList
16       if !requests.UnmarshalData(ctx, &target) {
17           utils.ErrorLog("Couldn't unmarshal protobuf to protos.RspGetPPList")
18           return
19       }
20
21       if target.Result.State != protos.ResultState_RES_SUCCESS {
22           utils.Log("failed to get any network")
23           return
24       }
25
26       err := p2pserver.GetP2pServer(ctx).SavePPList(ctx, &target)
27       if err != nil {
28           utils.ErrorLog("Error when saving PP List", err)
```

```
29      }
```

Listing 2.29: pp/event/get_pplist.go

Impact Insufficient authorization can result in exploits targeting privileged operations.

Suggestion Implement access controls for privileged functions.

2.2.2 Unverified response messages

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description A vulnerability exists in the code implementation where response messages are not verified to have corresponding request messages. Malicious nodes can exploit this issue by directly sending response messages to other nodes without proper validation, potentially leading to unexpected consequences. For instance, the `RspGetSPList` message unconditionally accepts the list of SPs specified in the message, without checking for a corresponding `ReqGetSPList` request. This vulnerability allows a malicious node to add fake or malicious SP nodes to the PP node, potentially compromising the network's security. The impact of this issue is that response functions can be invoked without receiving requests from the node.

Impact Response functions can be invoked without receiving requests from the node.

Suggestion Ensure that each response message corresponds to a request message.

2.2.3 `ReqUploadFileSlice` allows arbitrary file writing

Severity High

Status Fixed in `Version 2`

Introduced by `Version 1`

Description During the `ReqUploadFileSlice` process, the SP's signature is verified to confirm agreement to store the file with a file hash on the node. The `SaveUploadFile` operation then calculates the path using the `SliceHash` in the message and writes the slice content to that location. However, since the `SliceHash` is not checked during signature verification, the sender can specify an arbitrary `SliceHash`, allowing them to manipulate the `SliceSize` and bypass subsequent checks. This enables the sender to alter the `SliceHash` to achieve arbitrary file writing on the recipient's node. Therefore, an attacker can exploit this vulnerability to write a malicious script and overwrite key files in the victim's filesystem.

```
530   func SaveUploadFile(target *protos.ReqUploadFileSlice) error {
531       return file.SaveSliceData(target.Data, target.SliceInfo.SliceHash, target.SliceInfo.
              SliceOffset.SliceOffsetStart)
532   }
```

Listing 2.30: pp/task/upload_task.go

```
157  func SaveSliceData(data []byte, sliceHash string, offset uint64) error {
158      wmutex.Lock()
159      defer wmutex.Unlock()
160      slicePath, err := getSlicePath(sliceHash)
161      if err != nil {
162          return errors.Wrap(err, "failed getting slice path")
163      }
164      fileMg, err := os.OpenFile(slicePath, os.O_CREATE|os.O_RDWR, 0777)
165      defer func() {
166          _ = fileMg.Close()
167      }()
168      if err != nil {
169          return errors.Wrap(err, "failed opening a file")
170      }
171      _, err = fileMg.WriteAt(data, int64(offset))
172      if err != nil {
173          utils.ErrorLog("error save file")
174          return errors.Wrap(err, "failed writing data")
175      }
176      return nil
177  }
```

Listing 2.31: pp/file/file.go

Impact Nodes can be compromised due to arbitrary file writing.

Suggestion Restrict the file space writable by the program.

2.2.4 Potential DoS risk due to the absence of timeouts in message receiving and sending processes

Severity Medium

Status Fixed in `Version 2`

Introduced by `Version 1`

Description The P2P networking code implementation in SDS contains a vulnerability within the `readLoop` and `writeLoop` processes, which lack timeouts. This issue can cause connections to the same peer to become stuck, potentially leading to a DoS attack. While there may be arguments in favor of not having timeouts, it is crucial to recognize that this ideal scenario may not always be the case, and the vulnerability should be addressed.

```
582  func writeLoop(c WriteCloser, wg *sync.WaitGroup) {
583      var (
584          sendCh chan *message.RelayMsgBuf
585          cDone <-chan struct{}
586          sDone <-chan struct{}
587          packet *message.RelayMsgBuf
588          sc     *ServerConn
589      )
590
591      sendCh = c.(*ServerConn).sendCh
```

```
592         cDone = c.(*ServerConn).ctx.Done()
593         sDone = c.(*ServerConn).belong.ctx.Done()
594         sc = c.(*ServerConn)
595         defer func() {
596             if p := recover(); p != nil {
597                 Mylog(sc.belong.opts.logOpen, LOG_MODULE_WRITELOOP, fmt.Sprintf("panics: %v", p))
598             }
599             // drain all pending messages before exit
600         OuterFor:
601             for {
602                 select {
603                 case packet, ok := <-sendCh:
604                     // selected, not received: break from the loop
605                     if !ok {
606                         break OuterFor
607                     }
608                     // drain pending messages
609                     if packet != nil {
610                         if err := sc.writePacket(packet); err != nil {
611                             utils.ErrorLog(err)
612                             break OuterFor
613                         }
614                     }
615                 default:
616                     break OuterFor
617                 }
618             }
619             wg.Done()
620             GoroutineMap.Delete(sc.GetName() + "write")
621             c.Close()
622         }()
623 
624         for {
625             select {
626             case <-cDone: // connection closed
627                 Mylog(sc.belong.opts.logOpen, LOG_MODULE_WRITELOOP, "closes by conn")
628                 return
629             case <-sDone: // server closed
630                 Mylog(sc.belong.opts.logOpen, LOG_MODULE_WRITELOOP, "closes by server")
631                 return
632             case packet = <-sendCh:
633                 if packet != nil {
634                     if err := sc.writePacket(packet); err != nil {
635                         Mylog(sc.belong.opts.logOpen, LOG_MODULE_WRITELOOP, "write packet err", err.
                                Error())
636                         return
637                     }
638                 }
639             }
640         }
641     }
```

Listing 2.32: framework/core/conn.go

Impact The absence of timeouts could result in a DoS attack.

Suggestion Integrate timeout logic into the relevant functions.

2.2.5 Ignored error in authentication process

Severity High

Status Fixed in `Version 3`

Introduced by `Version 2`

Description In the `Version 2` of SDS, a verification callback is set for each message using the `RegisterAllEventHandl` function. When the message handler receives a message, it calls `VerifyMessage` to invoke the registered verification callback. However, the error returned by `VerifyMessage` is only logged, and not processed, which introduces a potential vulnerability.

```
74  func RspDeleteSlice(ctx context.Context, conn core.WriteCloser) {
75      var target protos.RspDeleteSlice
76      if err := VerifyMessage(ctx, header.RspDeleteSlice, &target); err != nil {
77          utils.ErrorLog("failed verifying the message, ", err.Error())
78      }
79      p2pserver.GetP2pServer(ctx).TransferSendMessageToSPServer(ctx, core.MessageFromContext(ctx)
            )
80  }
```

Listing 2.33: pp/event/delete_file.go

Impact An unhandled error makes the verification process ineffective.

Suggestion Take actions when the processes throw errors.

Chapter 3 Appendix: Detection Results from In-house Tools

The below table summarizes the findings of the customized in-house tools.

Tool Name	Scanned Item	Issues Found
CodeQL	UntrustedDataToAPI	None
	Uncontrolled data used in path expression	Yes (Section 2.2.3)
	InsecureRandomness	Yes [1]
	Incorrect conversion between integer types	Yes (Section 2.1.3)
	CleartextLogging	None
	XPath injection	None
	Insecure HostKeyCallback implementation	None
	Use of constant `state` value in OAuth 2.0 URL on qldatabase	None
	Use of a weak cryptographic key	None
	Suspicious characters in a regular expression	None
	Stored cross-site scripting	None
	Size computation for allocation may overflow	None
	Reflected cross-site scripting	None
	Potentially unsafe quoting	None
	Open URL redirect	None
	Missing regular expression anchor	None
	Log entries created from user input	None
	Insecure TLS configuration	None
	Information exposure through a stack trace	None
	Incomplete URL scheme check	None
	Incomplete regular expression for hostnames	None
	Hard-coded credentials	None
	Email content injection	None
	Disabled TLS certificate check	None
	Database query built from user-controlled sources	None
	Command built from user-controlled sources	None

	Command built from stored data	None
	Bad redirect check	None
	Arbitrary file write extracting an archive containing symbolic links	None
	Arbitrary file write during zip extraction ("zip slip")	None
GoSec	Use of weak cryptographic primitive [G401 (CWE-326)]	Yes [2]
	Deferring unsafe method [G307 (CWE-703)]	None
	Errors unhandled [G104 (CWE-703)]	Yes (Section 2.1.8)
	Expect WriteFile permissions to be 0600 or less [G306 (CWE-276)]	None
	Expect directory permissions to be 0750 or less [G301 (CWE-276)]	None
	Expect file permissions to be 0600 or less [G302 (CWE-276)]	None
	Private key file permissions need to be restricted / Potential Slowloris Attack	None
	Potential file inclusion via variable [G304 (CWE-22)]	None
	Subprocess launched with variable [G204 (CWE-78)]	None
	Use of net/http serve function that has no support for setting timeouts [G114 (CWE-676)]	Yes (Section 2.2.4)
	Use of unsafe calls should be audited [G103 (CWE-242)]	None

[1]The insecure randomnesses are not exploitable.

[2]The use of weak cryptographic primitive (i.e., `crypto/md5`) is only present internally and are not exploitable externally.