Hypergrid SSN

Sonic

/ DRAFT /

HALBORN

Hypergrid SSN - Sonic

Prepared by: HALBORN

Last Updated 09/27/2024

Date of Engagement by: September 9th, 2024 - September 27th, 2024

Summary

0% ⓘ OF ALL REPORTED FINDINGS HAVE BEEN ADDRESSED

ALL FINDINGS	CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
9	2	1	1	1	4

TABLE OF CONTENTS

1. INTRODUCTION

`Sonic` engaged Halborn to conduct a security assessment on their app chain module beginning on 2024-09-09 and ending on 2024-09-27. The security assessment was scoped to the `hypergrid-ssn` Cosmos application scoped in the provided repository to the **Halborn team**.

2. ASSESSMENT SUMMARY

The team at Halborn was provided two weeks and four days for the engagement and assigned one full-time security engineer to assessment the security of the merge requests. The security engineers are blockchain and smart-contract security experts with advanced penetration testing, smart-contract hacking, and deep knowledge of multiple blockchain protocols.

The purpose of this assessment is to:

- Ensure that the **Golang components** operate as intended.
- Identify potential security issues with the Cosmos application.

In summary, Halborn identified several relevant issues that must be addressed by `Sonic team`. Most of the security concerns shared in this document reflect multiple logic implementations going against the determinism principle required by distributed ledgers and consensus mechanisms.

3. TEST APPROACH AND METHODOLOGY

Halborn performed a combination of manual and automated security testing to balance efficiency, timeliness, practicality, and accuracy in regard to the scope of the custom modules. While manual testing is recommended to uncover flaws in logic, process, and implementation; automated testing techniques help enhance coverage of structures and can quickly identify items that do not follow security best practices. The following phases and associated tools were used throughout the term of the assessment :

- Research into architecture and purpose.
- Static Analysis of security for scoped repository, and imported functions. (e.g., `staticcheck`, `gosec`, `unconvert`, `codeql`, `ineffassign` and `semgrep`)
- Manual Assessment for discovering security vulnerabilities on the codebase.
- Ensuring the correctness of the codebase.
- Dynamic Analysis of files and modules in scope.

4. RISK METHODOLOGY

Every vulnerability and issue observed by Halborn is ranked based on **two sets** of **Metrics** and a **Severity Coefficient**. This system is inspired by the industry standard Common Vulnerability Scoring System.

The two **Metric sets** are: **Exploitability** and **Impact**. **Exploitability** captures the ease and technical means by which vulnerabilities can be exploited and **Impact** describes the consequences of a successful exploit.

The **Severity Coefficients** is designed to further refine the accuracy of the ranking with two factors: **Reversibility** and **Scope**. These capture the impact of the vulnerability on the environment as well as the number of users and smart contracts affected.

The final score is a value between 0-10 rounded up to 1 decimal place and 10 corresponding to the highest security risk. This provides an objective and accurate rating of the severity of security vulnerabilities in smart contracts.

The system is designed to assist in identifying and prioritizing vulnerabilities based on their level of risk to address the most critical issues in a timely manner.

4.1 EXPLOITABILITY

ATTACK ORIGIN (AO):

Captures whether the attack requires compromising a specific account.

ATTACK COST (AC):

Captures the cost of exploiting the vulnerability incurred by the attacker relative to sending a single transaction on the relevant blockchain. Includes but is not limited to financial and computational cost.

ATTACK COMPLEXITY (AX):

Describes the conditions beyond the attacker's control that must exist in order to exploit the vulnerability. Includes but is not limited to macro situation, available third-party liquidity and regulatory challenges.

METRICS:

EXPLOITABILIY METRIC (M_E)	METRIC VALUE	NUMERICAL VALUE
Attack Origin (AO)	Arbitrary (AO:A) Specific (AO:S)	1 0.2
Attack Cost (AC)	Low (AC:L) Medium (AC:M) High (AC:H)	1 0.67 0.33
Attack Complexity (AX)	Low (AX:L) Medium (AX:M) High (AX:H)	1 0.67 0.33

Exploitability E is calculated using the following formula:

$$E = \prod m_e$$

4.2 IMPACT

CONFIDENTIALITY (C):

Measures the impact to the confidentiality of the information resources managed by the contract due to a successfully exploited vulnerability. Confidentiality refers to limiting access to authorized users only.

INTEGRITY (I):

Measures the impact to integrity of a successfully exploited vulnerability. Integrity refers to the trustworthiness and veracity of data stored and/or processed on-chain. Integrity impact directly affecting Deposit or Yield records is excluded.

AVAILABILITY (A):

Measures the impact to the availability of the impacted component resulting from a successfully exploited vulnerability. This metric refers to smart contract features and functionality, not state. Availability impact directly affecting Deposit or Yield is excluded.

DEPOSIT (D):

Measures the impact to the deposits made to the contract by either users or owners.

YIELD (Y):

Measures the impact to the yield generated by the contract for either users or owners.

METRICS:

IMPACT METRIC (M_I)	METRIC VALUE	NUMERICAL VALUE
Confidentiality (C)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Integrity (I)	None (I:N) Low (I:L) Medium (I:M) High (I:H) Critical (I:C)	0 0.25 0.5 0.75 1
Availability (A)	None (A:N) Low (A:L) Medium (A:M) High (A:H) Critical (A:C)	0 0.25 0.5 0.75 1
Deposit (D)	None (D:N) Low (D:L) Medium (D:M) High (D:H) Critical (D:C)	0 0.25 0.5 0.75 1
Yield (Y)	None (Y:N) Low (Y:L) Medium (Y:M) High (Y:H) Critical (Y:C)	0 0.25 0.5 0.75 1

Impact I is calculated using the following formula:

$$I = max(m_I) + \frac{\sum m_I - max(m_I)}{4}$$

4.3 SEVERITY COEFFICIENT

REVERSIBILITY (R):

Describes the share of the exploited vulnerability effects that can be reversed. For upgradeable contracts, assume the contract private key is available.

SCOPE (S):

Captures whether a vulnerability in one vulnerable contract impacts resources in other contracts.

METRICS:

SEVERITY COEFFICIENT (C)	COEFFICIENT VALUE	NUMERICAL VALUE
Reversibility (r)	None (R:N) Partial (R:P) Full (R:F)	1 0.5 0.25
Scope (s)	Changed (S:C) Unchanged (S:U)	1.25 1

Severity Coefficient C is obtained by the following product:

$$C = rs$$

The Vulnerability Severity Score S is obtained by:

$$S = min(10, EIC * 10)$$

The score is rounded up to 1 decimal places.

SEVERITY	SCORE VALUE RANGE
Critical	9 - 10
High	7 - 8.9
Medium	4.5 - 6.9

SEVERITY	SCORE VALUE RANGE
Low	2 - 4.4
Informational	0 - 1.9

5. SCOPE

FILES AND REPOSITORY

(a) Repository: hypergrid-ssn

(b) Assessed Commit ID: 0f5d345

(c) Items in scope:

- cmd/hypergrid-ssnd/cmd/config.go
- cmd/hypergrid-ssnd/cmd/commands.go
- cmd/hypergrid-ssnd/cmd/root.go
- cmd/hypergrid-ssnd/main.go
- go.mod
- tools/tools.go
- tools/solana.go
- app/export.go
- app/app.go
- app/genesis.go
- app/app_config.go
- app/genesis_account.go
- proto/buf.gen.gogo.yaml
- hypergrid-aide/go.mod
- hypergrid-aide/tools/solana.go
- hypergrid-aide/tools/sys.go
- hypergrid-aide/tools/cosmos.go
- hypergrid-aide/go.sum
- hypergrid-aide/main.go
- go.sum
- x/hypergridssn/types/messages_grid_inbox.go
- x/hypergridssn/types/params.go
- x/hypergridssn/types/key_hypergrid_node.go
- x/hypergridssn/types/query.pb.gw.go
- x/hypergridssn/types/types.go
- x/hypergridssn/types/keys.go
- x/hypergridssn/types/expected_keepers.go
- x/hypergridssn/types/messages_hypergrid_node.go
- x/hypergridssn/types/messages_grid_block_fee.go
- x/hypergridssn/types/messages_solana_account.go
- x/hypergridssn/types/messages_fee_settlement_bill.go
- x/hypergridssn/types/genesis.go
- x/hypergridssn/types/msg_update_params.go
- x/hypergridssn/types/key_solana_account.go
- x/hypergridssn/types/codec.go
- x/hypergridssn/types/errors.go
- x/hypergridssn/module/simulation.go
- x/hypergridssn/module/genesis.go
- x/hypergridssn/module/autocli.go
- x/hypergridssn/module/module.go
- x/hypergridssn/simulation/solana_account.go
- x/hypergridssn/simulation/fee_settlement_bill.go

- x/hypergridssn/simulation/grid_inbox.go
- x/hypergridssn/simulation/helpers.go
- x/hypergridssn/simulation/hypergrid_node.go
- x/hypergridssn/simulation/grid_block_fee.go
- x/hypergridssn/keeper/query.go
- x/hypergridssn/keeper/solana_account.go
- x/hypergridssn/keeper/query_grid_block_fee.go
- x/hypergridssn/keeper/query_hypergrid_node.go
- x/hypergridssn/keeper/params.go
- x/hypergridssn/keeper/keeper.go
- x/hypergridssn/keeper/msg_server_grid_inbox.go
- x/hypergridssn/keeper/query_grid_inbox.go
- x/hypergridssn/keeper/msg_server_hypergrid_node.go
- x/hypergridssn/keeper/fee_settlement_bill.go
- x/hypergridssn/keeper/msg_server_grid_block_fee.go
- x/hypergridssn/keeper/msg_server_solana_account.go
- x/hypergridssn/keeper/msg_server_fee_settlement_bill.go
- x/hypergridssn/keeper/query_params.go
- x/hypergridssn/keeper/msg_server.go
- x/hypergridssn/keeper/grid_inbox.go
- x/hypergridssn/keeper/query_fee_settlement_bill.go
- x/hypergridssn/keeper/msg_update_params.go
- x/hypergridssn/keeper/hypergrid_node.go
- x/hypergridssn/keeper/grid_block_fee.go
- x/hypergridssn/keeper/query_solana_account.go

Out-of-Scope: Third party dependencies and economic attacks.

Out-of-Scope: New features/implementations after the remediation commit IDs.

6. ASSESSMENT SUMMARY & FINDINGS OVERVIEW

CRITICAL	HIGH	MEDIUM	LOW	INFORMATIONAL
2	1	1	1	4

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-01 - INDETERMINISTIC SOLANA ACCOUNT INFORMATION RETRIEVAL	CRITICAL	-
HAL-02 - RANDOM SOLANA ACCOUNT GENERATION CAUSES INDETERMINISM	CRITICAL	-

SECURITY ANALYSIS	RISK LEVEL	REMEDIATION DATE
HAL-03 - RESOURCE EXHAUSTION IN MAKE USAGE	HIGH	-
HAL-04 - SORT MAPPINGS BEFORE USAGE	MEDIUM	-
HAL-05 - EXTERNAL API/RPC CALLS CAN LEAD TO INDETERMINISM	LOW	-
HAL-06 - USE OF MAGIC VALUES	INFORMATIONAL	-
HAL-07 - INCONSISTENT LOGGING MECHANISM	INFORMATIONAL	-
HAL-08 - OPEN TO-DOS	INFORMATIONAL	-
HAL-09 - USAGE OF HARDCODED PATHS	INFORMATIONAL	-

7. FINDINGS & TECH DETAILS

7.1 (HAL-01) INDETERMINISTIC SOLANA ACCOUNT INFORMATION RETRIEVAL

// CRITICAL

Description

The functions `CreateSolanaAccount` and `UpdateSolanaAccount` interact with the Solana blockchain to verify account details via RPC calls, specifically using the `solana.GetAccountInfo` function. This interaction is crucial for validating the state of Solana accounts before they are mirrored or updated in the Cosmos chain.

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/87507ac08c24922d235ec4aba3e4a2db4251fd50/x/hypergridssn/keeper/msg_server_solana_account.go#L36-L58

```
resp, err := solana.GetAccountInfo(node.Rpc, msg.Address)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

value, err := cmtjson.Marshal(resp.Value)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

var solanaAccount = types.SolanaAccount{
    Creator: msg.Creator,
    Address: msg.Address,
    Version: msg.Version,
    Source:  node.Pubkey,
    Slot:    strconv.FormatUint(resp.RPCContext.Context.Slot, 10),
    Value:   string(value),
}

k.SetSolanaAccount(
    ctx,
    solanaAccount,
)
```

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/87507ac08c24922d235ec4aba3e4a2db4251fd50/x/hypergridssn/keeper/msg_server_solana_account.go#L89-L108

```
resp, err := solana.GetAccountInfo(node.Rpc, msg.Address)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

value, err := cmtjson.Marshal(resp.Value)
```

```
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

var solanaAccount = types.SolanaAccount{
    Creator: msg.Creator,
    Address: msg.Address,
    Version: msg.Version,
    Source:  valFound.Source,
    Slot:    strconv.FormatUint(resp.RPCContext.Context.Slot, 10),
    Value:   string(value),
}

k.SetSolanaAccount(ctx, solanaAccount)
```

However, due to the asynchronous nature of these RPC calls and the continuous block production on Solana, there is a significant risk of state indeterminacy.
Different nodes in the Cosmos network may receive varying account states depending on the exact moment their RPC call is processed by the Solana network. This variation can occur because account balances, ownership details, or other state-relevant information on Solana might change between the issuance of these calls across different Cosmos nodes.

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/97e4c3474c07fa559e881bdc3b6dd5279f3a8429/tools/solana.go#L20-L55

```
func GetAccountInfo(rpcUrl string, address string) (*rpc.GetAccountInfoResult, error)
    // endpoint := rpc.DevNet_RPC //MainNetBeta_RPC
    client := rpc.New(rpcUrl)
    pubKey := solana.MustPublicKeyFromBase58(address) // serum token

    // Get the account
    return client.GetAccountInfoWithOpts(
        context.TODO(),
        pubKey,
        // You can specify more options here:
        &rpc.GetAccountInfoOpts{
            Encoding:   solana.EncodingBase64Zstd,
            Commitment: rpc.CommitmentFinalized,
            // You can get just a part of the account data by specify a DataSlice:
            // DataSlice: &rpc.DataSlice{
            //  Offset: pointer.ToUint64(0),
            //  Length: pointer.ToUint64(1024),
            // },
        },
    )

    // ...

}
```

go/pkg/mod/github.com/gagliardetto/solana-go@v1.10.0/rpc/types.go:

```
type GetAccountInfoResult struct {
    RPCContext
    Value *Account `json:"value"`
}
```

go/pkg/mod/github.com/gagliardetto/solana-go@v1.10.0/rpc/types.go:

```
type Account struct {
    // Number of lamports assigned to this account
    Lamports uint64 `json:"lamports"`

    // Pubkey of the program this account has been assigned to
    Owner solana.PublicKey `json:"owner"`

    // Data associated with the account, either as encoded binary data or JSON format
    Data *DataBytesOrJSON `json:"data"`

    // Boolean indicating if the account contains a program (and is strictly read-only
    Executable bool `json:"executable"`

    // The epoch at which this account will next owe rent
    RentEpoch *big.Int `json:"rentEpoch"`
}
```

Such discrepancies can lead to relevant consistency issues within the consensus during block production. For example, if different nodes validate and record different states for the same Solana account, this could lead to a state split or even halt the blockchain if consensus mechanisms detect irreconcilable differences between node states.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:N/Y:N (10.0)

Recommendation

Implement a mechanism where data validity is agreed upon at the data source level, by using a specific Solana block threshold height or timestamp to query account information. This approach ensures that all nodes in the Cosmos network refer to the same state information for Solana accounts since it should allow specifying from which valid block number to retrieve this information.
One approach would be to include a block value in the message used in both functions, this parameter would allow setting a specific block number from which retrieve the information, this number must be found inside a range of values in order to guarantee the block is not too old and including additional checks to verify it was finalized.

7.2 (HAL-02) RANDOM SOLANA ACCOUNT GENERATION CAUSES INDETERMINISM

// CRITICAL

Description

The functions `CreateSolanaAccount` and `UpdateSolanaAccount` interact with the Solana blockchain to verify account details via RPC calls, specifically using the `solana.GetAccountInfo` function. This interaction is crucial for validating the state of Solana accounts before they are mirrored or updated in the Cosmos chain.

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/87507ac08c24922d235ec4aba3e4a2db4251fd50/x/hypergridssn/keeper/msg_server_solana_account.go#L36-L58

```
resp, err := solana.GetAccountInfo(node.Rpc, msg.Address)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

value, err := cmtjson.Marshal(resp.Value)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

var solanaAccount = types.SolanaAccount{
    Creator: msg.Creator,
    Address: msg.Address,
    Version: msg.Version,
    Source:  node.Pubkey,
    Slot:    strconv.FormatUint(resp.RPCContext.Context.Slot, 10),
    Value:   string(value),
}

k.SetSolanaAccount(
    ctx,
    solanaAccount,
)
```

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/87507ac08c24922d235ec4aba3e4a2db4251fd50/x/hypergridssn/keeper/msg_server_solana_account.go#L89-L108

```
resp, err := solana.GetAccountInfo(node.Rpc, msg.Address)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
}

value, err := cmtjson.Marshal(resp.Value)
if err != nil {
    return nil, errorsmod.Wrap(sdkerrors.ErrUnauthorized, err.Error())
```

```
}

var solanaAccount = types.SolanaAccount{
    Creator: msg.Creator,
    Address: msg.Address,
    Version: msg.Version,
    Source:  valFound.Source,
    Slot:    strconv.FormatUint(resp.RPCContext.Context.Slot, 10),
    Value:   string(value),
}

k.SetSolanaAccount(ctx, solanaAccount)
```

However, due to the asynchronous nature of these RPC calls and the continuous block production on Solana, there is a significant risk of state indeterminacy.
Different nodes in the Cosmos network may receive varying account states depending on the exact moment their RPC call is processed by the Solana network. This variation can occur because account balances, ownership details, or other state-relevant information on Solana might change between the issuance of these calls across different Cosmos nodes.

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/97e4c3474c07fa559e881bdc3b6dd5279f3a8429/tools/solana.go#L20-L55

```
func GetAccountInfo(rpcUrl string, address string) (*rpc.GetAccountInfoResult, error)
    // endpoint := rpc.DevNet_RPC //MainNetBeta_RPC
    client := rpc.New(rpcUrl)
    pubKey := solana.MustPublicKeyFromBase58(address) // serum token

    // Get the account
    return client.GetAccountInfoWithOpts(
        context.TODO(),
        pubKey,
        // You can specify more options here:
        &rpc.GetAccountInfoOpts{
            Encoding:   solana.EncodingBase64Zstd,
            Commitment: rpc.CommitmentFinalized,
            // You can get just a part of the account data by specify a DataSlice:
            // DataSlice: &rpc.DataSlice{
            //  Offset: pointer.ToUint64(0),
            //  Length: pointer.ToUint64(1024),
            // },
        },
    )

    // ...

}
```

go/pkg/mod/github.com/gagliardetto/solana-go@v1.10.0/rpc/types.go:

```
type GetAccountInfoResult struct {
    RPCContext
    Value *Account `json:"value"`
}
```

go/pkg/mod/github.com/gagliardetto/solana-go@v1.10.0/rpc/types.go:

```
type Account struct {
    // Number of lamports assigned to this account
    Lamports uint64 `json:"lamports"`

    // Pubkey of the program this account has been assigned to
    Owner solana.PublicKey `json:"owner"`

    // Data associated with the account, either as encoded binary data or JSON format
    Data *DataBytesOrJSON `json:"data"`

    // Boolean indicating if the account contains a program (and is strictly read-only
    Executable bool `json:"executable"`

    // The epoch at which this account will next owe rent
    RentEpoch *big.Int `json:"rentEpoch"`
}
```

Such discrepancies can lead to relevant consistency issues within the consensus during block production. For example, if different nodes validate and record different states for the same Solana account, this could lead to a state split or even halt the blockchain if consensus mechanisms detect irreconcilable differences between node states.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:C/I:C/D:N/Y:N (10.0)

Recommendation

To resolve the issue, the private key generation process must be made deterministic, ensuring that all nodes generate/use the same key for the same transaction. In this case, as the random generation is linked to a private key used in Solana, it isn't convenient to set a deterministic key generation, for example by using a common seed, as all nodes and users will have access to every account's private key generated by nodes.
Therefore, one approach is to have the public key passed as part of the transaction message (`MsgCreateGridInbox`) rather than generating it within the `SendTxInbox` function. This way, the public key is known in advance and consistent across all nodes.

7.3 (HAL-03) RESOURCE EXHAUSTION IN MAKE USAGE

// HIGH

Description

The `CreateGridBlockFee` function in the Cosmos SDK message server is designed to process grid block fee entries from incoming messages. Each message can include multiple items, each specifying details about a grid block fee such as grid ID, slot, block hash, block time, and the associated fee.
However, there is a significant risk associated with the way memory allocation is handled in this function.
https://github.com/mirrorworld-universe/hypergrid-ssn/blob/a468c2365b89c9d225ff195d89bb7a88a41ae19b/x/hypergridssn/keeper/msg_server_grid_block_fee.go#L11-L37

```
func (k msgServer) CreateGridBlockFee(goCtx context.Context, msg *types.MsgCreateGridB
    ctx := sdk.UnwrapSDKContext(goCtx)

    ids := make([]uint64, len(msg.Items))
    for _, item := range msg.Items {
        var gridBlockFee = types.GridBlockFee{
            Creator:   msg.Creator,
            Grid:      item.Grid,
            Slot:      item.Slot,
            Blockhash: item.Blockhash,
            Blocktime: item.Blocktime,
            Fee:       item.Fee,
        }

        //todo: make blockhash unique

        id := k.AppendGridBlockFee(
            ctx,
            gridBlockFee,
        )
        ids = append(ids, id)
    }

    return &types.MsgCreateGridBlockFeeResponse{
        Id: ids[len(ids)-1],
    }, nil
}
```

The function initializes a slice with a length based on the number of items in the incoming message (`make([]uint64, len(msg.Items))`), but then it appends to this slice rather than assigning values to existing indices. This can lead to an oversized slice with a capacity much larger than needed, resulting in unnecessary memory allocation.
Moreover, there is no check on the length of `msg.Items`, which means a malicious user could craft a message with a very large number of items, potentially leading to resource exhaustion. This could cause the function to allocate excessive amounts of memory, leading to system slowdown, crashes, or in worst cases, halting the node entirely.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:H/I:M/D:N/Y:N (8.8)

Recommendation

The following changes should be made in order to fix the aforementioned issues:

1. **Limit the Number of Items**: Implement a maximum limit on the number of items that can be included in a `MsgCreateGridBlockFee` message. This limit should be based on realistic usage scenarios and the capacity of the system. By enforcing a limit, the function can prevent excessive resource allocation that could lead to denial of service.
2. **Properly Size the Slice:** Instead of using `append` on a pre-sized slice, initialize the slice with a capacity of zero and let it grow as needed. This can be done by changing `make([]uint64, len(msg.Items))` to `make([]uint64, 0, len(msg.Items))`. This approach ensures that the slice only allocates memory for actual data being used, avoiding the creation of empty slots.
3. **Input Validation**: Add validation checks for each item in the message to ensure all provided data is within acceptable bounds and conforms to expected formats. This includes validating the grid ID, slot, block hash, block time, and fee to prevent any malformed or malicious data from being processed.

7.4 (HAL-04) SORT MAPPINGS BEFORE USAGE

// MEDIUM

Description

In the provided code, a `map[string]uint64` named `bills` is used to accumulate fees, and the map is later iterated over to create `SettlementBillParam` objects:
https://github.com/mirrorworld-universe/hypergrid-ssn/blob/c22f5d98b6597ca0fa89b3b318acbee9a263d5d2/x/hypergridssn/keeper/msg_server_fee_settlement_bill.go#L35-L44

```
bills := make(map[string]uint64)
for i := startId; i < msg.EndId; i++ {
    item, found := k.GetGridBlockFee(ctx, i)
    if !found {
        break
    }

    fee, _ := strconv.ParseUint(item.Fee, 10, 64)
    bills[item.Grid] += fee
}
```

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/97e4c3474c07fa559e881bdc3b6dd5279f3a8429/tools/solana.go#L237-L249

```
for key, value := range bills {
    Bills = append(Bills, SettlementBillParam{
        Key:    solana.MustPublicKeyFromBase58(key),
        Amount: value,
    })
}

instructionData := SettleFeeBillParams{
    Instruction: 1,
    FromID:      FromId,
    EndID:       EndID,
    Bills:       Bills,
}
```

Using Go maps in Cosmos SDK chains can introduce indeterminism, because Go maps do not guarantee any specific order when iterating over their elements. This leads to different nodes potentially processing transactions in different orders, which would result in diverging blockchain states across validators, risking a **network split** or chain halting due to inconsistent state updates.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:M/I:M/D:N/Y:N (6.3)

Recommendation

To avoid indeterminism, the iteration over `bills` should be done in a deterministic manner by sorting the keys before processing. This ensures that all nodes will handle the map entries in the same order, preserving consensus.

7.5 (HAL-05) EXTERNAL API/RPC CALLS CAN LEAD TO INDETERMINISM

// LOW

Description

In functions like `SendTxFeeSettlement` and `SendTxInbox`, external API or RPC calls are used to execute transactions. These calls interact with external services, such as external endpoints or blockchain RPC nodes, to submit transactions or fetch data. This introduces a source of indeterminism in the Cosmos SDK chain, as the responses from these external services may vary across different nodes due to factors like network latency, timeouts, service availability, or differences in response data.

Since Cosmos SDK chains rely on all validators processing the same transaction in exactly the same way to reach consensus, any variation in responses from these external calls can lead to inconsistent states across nodes. This can result in diverging blockchain states, potentially causing **network splits** or a chain halt due to the inability of nodes to reach consensus on the transaction outcome.

BVSS

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:L/I:L/D:N/Y:N (3.1)

Recommendation

To ensure deterministic execution of transactions, external API and RPC calls should be avoided in consensus-critical logic.

7.6 (HAL-06) USE OF MAGIC VALUES

// INFORMATIONAL

Description

Magic numbers are direct numerical or string values used in the code without any explanation or context. This practice can lead to code maintainability issues, potential errors, and difficulty in understanding the code's logic.
By using a constant state variable in config files or custom Cosmos SDK parameters, you can enhance code clarity, reduce the potential for errors, and facilitate easier updates. If the value needs to be changed, you can modify it in a single location, rather than searching for and updating each instance of the magic number. This practice promotes better code quality and simplifies the maintenance process.
https://github.com/mirrorworld-universe/hypergrid-ssn/blob/97e4c3474c07fa559e881bdc3b6dd5279f3a8429/tools/solana.go#L138-L139

```
const SonicFeeProgramID = "SonicFeeSet1ement11111111111111111111111111"
const L1InboxProgramID = "5XJ1wZkTwAw9mc5FbM3eBgAT83TKgtAGzKos9wVxC6my"
```

https://github.com/mirrorworld-universe/hypergrid-ssn/blob/863f60583f1707ba609d4a52cba6001609a58fa3/hypergrid-aide/main.go#L18-L25

```
const SOLANA_RPC_ENDPOINT = "<http://localhost:8899>" //"<https://devnet1.sonic.game>"
const COSMOS_RPC_ENDPOINT = "<http://172.31.10.244:26657>"
const COSMOS_ADDRESS_PREFIX = "cosmos"
const COSMOS_HOME = ".hypergrid-ssn"
const COSMOS_KEY = "my_key"
const COSMOS_GAS = "100000000"

const AIDE_GET_BLOCKS_COUNT_LIMIT = uint64(200)
```

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To improve code maintainability, readability, and reduce the risk of potential errors, it is recommended to replace magic numbers in code files with well-defined constants in configuration files in a structured way. By using constants in configuration files, developers can provide clear and descriptive names for specific values, making the code easier to understand and maintain. Additionally, updating the values becomes more straightforward, as changes can be made **in a single location**, reducing the risk of errors and inconsistencies.

7.7 (HAL-07) INCONSISTENT LOGGING MECHANISM

// INFORMATIONAL

Description

In the entire project, it has been identified the mixed usage of `fmt.Print` related functions and a custom logging mechanism per module in order to print different information though the program's execution. This methodology does not follow any format standard and could lead to confusing outputs from the application.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider implementing a logging mechanism to print relevant information in each output, for example, the module where it was executed or the function which is triggering that output. Also, including different log levels.

7.8 (HAL-08) OPEN TO-DOS

// INFORMATIONAL

Description

It has been identified Open TO-DOs in the scoped files. Open TO-DOs can point to architecture or programming issues that still need to be resolved. Often these kinds of comments indicate areas of complexity or confusion for developers. This provides value and insight to an attacker who aims to cause damage to the protocol.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

Consider resolving the TO-DOs before deploying code to a production context. Use an independent issue tracker or other project management software to track development tasks.

7.9 (HAL-09) USAGE OF HARDCODED PATHS

// INFORMATIONAL

Description

In the `getLocalPrivateKey` function, the path to the user's private key is hardcoded: https://github.com/mirrorworld-universe/hypergrid-ssn/blob/97e4c3474c07fa559e881bdc3b6dd5279f3a8429/tools/solana.go#L141-L149

```
func getLocalPrivateKey() (solana.PrivateKey, error) {
    //get home path "~/"
    home, err := os.UserHomeDir()
    if err != nil {
        // panic(err)
        return nil, err
    }
    // Load the account that you will send funds FROM:
    accountFrom, err := solana.PrivateKeyFromSolanaKeygenFile(home + "/.config/solana/
```

This hardcoding of the key file path (`~/.config/solana/id.json`) does not allow users to configure where they want to store their private keys. This creates poor user experience (UX) as users cannot customize the location of their keys based on their system or preferences. Additionally, hardcoded "magic values" make the code less flexible and harder to maintain, as any changes to file paths require code modifications.

Score

AO:A/AC:L/AX:L/R:N/S:U/C:N/A:N/I:N/D:N/Y:N (0.0)

Recommendation

To improve UX and maintainability, allow the private key path to be configurable. This can be achieved by passing the key file path as a parameter, or by using environment variables or configuration files.

Halborn strongly recommends conducting a follow-up assessment of the project either within six months or immediately following any material changes to the codebase, whichever comes first. This approach is crucial for maintaining the project's integrity and addressing potential vulnerabilities introduced by code modifications.