Security Audit Report

Mars Protocol Envoy module: Source Code Analysis

03.02.2023
Last revised 2023/02/27

Authors: Ranadeep Biswas, Andrey Kuprianov

Contents

Audit Overview

The Project

The Delphi Labs LTD team engaged with Informal Systems to conduct a security audit of their implementation of *Envoy* module as part of their Mars Hub implementation.

This module is responsible for automated communication between Mars Hub - the sovereign blockchain implementing Mars Protocol; and its corresponding account, modules, or smart contracts deployed on other IBC-connected chains or *outposts*. It leverages Interchain Accounts (ICS-27) to make this possible.

The module is initiated with a module account. It has three ways to make changes to the blockchain state.

- Anyone can create an interchain account of the module account at a given IBC connection.
- The other two are done via governance proposal. Anyone can submit a governance proposal to execute a transaction in this module.
  - To send funds from its account or community fee pool to a remote interchain account over the `transfer` channel.
  - To execute some transactions at a remote interchain account over the `icacontroller-*` channel.

The module includes two query APIs that let anyone query the interchain account of the module at a given connection ID or all active interchain accounts of the module.

Scope of this report

The agreed-upon work plan was to audit the `/x/envoy` module in the Mars Hub blockchain at commit `c7795c`.

This report covers the work of the above task that was conducted from January 23, 2023, through February 3, 2023, by Informal Systems by the following personnel:

- Ranadeep Biswas
- Andrey Kuprianov

Conducted work

The Mars protocol team shared onboarding materials and a code walkthrough recording with the Informal Systems team. The Informal Systems team performed manual code analysis and improved the existing tests in the source code. Mars Protocol and Informal Systems teams met weekly to share the findings and the progress.

Timeline

- 17 January 2023: Kickoff meeting.
  - Attendees: Larry and Kris from Delphi Labs LTD, Ranadeep and Tesnim from Informal Systems
- 26 January 2023: Sync meeting. (Shared the found issues. Suggested to avoid iterating over all possible ICAs)
  - Attendees: Larry, Dane, and Kris from Delphi Labs LTD, Ranadeep and Tesnim from Informal Systems
- 3 February 2023: Sync meeting. (Patched found issues in a fork. Created an E2E test for the Envoy module)
  - Attendees: Larry and Kris from Delphi Labs LTD, Ranadeep and Tesnim from Informal Systems

Conclusions

The module source code is of high quality - concise and well-documented with explanation and design choices. One **High severity** issue was found during this audit; the rest were marked Medium or Informational severity. A solution is proposed in a pull request for the high-severity issue. For others, we recommended details that should be addressed to raise the code quality of the module.

Further Increasing Confidence

The scope of this audit was limited to manual code review and manual analysis and reconstruction of the protocols. To further increase confidence in the protocol and the implementation, we recommend following up with more rigorous formal measures, including automated model checking and model-based adversarial testing. Our experience shows that incorporating test suites driven by TLA+ models that can lead the implementation into suspected edge cases and error scenarios enable the discovery of issues that are unlikely to be identified through manual review.

System Overview

Mars Protocol is a credit protocol that is decentralized, interchain, non-custodial, transparent, algorithmic, and community governed.

The entire design of the protocol is out of the scope of this audit. We will only focus on the Envoy module that the Mars Hub (the sovereign blockchain that implements the Mars Protocol) will use to communicate among corresponding accounts, modules, or smart contracts deployed in other chains or *outposts* as part of the Mars Protocol.

It is assumed that the reader is familiar with the Cosmos-SDK and the IBC protocol, specifically IBC token transfer (ICS-20) and Interchain Accounts (ICS-27).

Behavior

The *Mars Hub* is the sovereign blockchain implementing the Mars Protocol. It controls the artifacts and assets deployed in other chains as part of the entire protocol. It leverages Interchain Accounts (ICS-27) to operate in a decentralized, non-custodial way. The implementation is seemingly adapted from the `intertx` module in the interchain-accounts-demo repo.

The Envoy module in Mars Hub is responsible for communicating among different *outposts*. The module has a module account that can act as any other Cosmos-SDK bank account i.e., it can send and receive balances (on-chain or over IBC). The only way to perform these critical transactions for the Envoy module accounts is via the module itself. The Envoy module requires these critical transactions to be submitted via the Governance module account. So a user can only submit a governance proposal to execute an Envoy module account transactions that send funds or executes transactions on an outpost. Since the entire community will scrutinize the governance proposal, it is fair to assume that it is nearly impossible to execute a malicious critical transaction on behalf of the Envoy module account.

Although, the Envoy module allows anyone to register Interchain Accounts of its module account on the outposts. The registration is permissionless because it is harmless to register an interchain account on an open connection to an outpost. If there exists an interchain account already, the transaction does not do anything. Note the outpost must also implement the Interchain Account Host app.

Once the Envoy module account has Interchain Accounts registered on an outpost, it is ready to communicate with it. The Envoy module provides two major transaction APIs.

- To send funds from its module account to an interchain account at a given IBC channel ID. If the module account does not have enough balance, it will take from the community fee module. Otherwise, it fails. The IBC packets for this transaction are sent via the ibc-transfer channel.
- To execute transactions at an interchain account at a given IBC connection ID. The IBC packets for this transaction are sent via the interchain accounts channel.

As already mentioned before, these transactions can not be submitted directly. They are supposed to be executed by the Governance module when a governance proposal is passed.

Additionally, these behaviors are formally specified in TLA+ language. We verified the specification against few critical invariants. The specification and invariant will be discussed in following sections.

Implementation

In this section, we give a detailed analysis of the main components of the Enovy module. We look into the implementation of the application state of the module (`Keeper`), transaction validation (`ValidateBasic` and `GetSigners`), and application logic for executing transactions and querying the application state of the module. We will also describe how the TLA+ specification models the module state and the implementation logic.

Keeper

The module consists of a data structure called *Keeper*, a Cosmos-SDK convention that stores the application state. Usually, a module keeper consists of other module keepers (i.e., the current module is dependent on these other modules) and some other data specific to the module.

The following table lists the fields used in the Envoy module keeper and their purpose.

Field	Module	Purpose
`accountKeeper`	`auth`	Calculate the module account
`bankKeeper`	`bank`	Calculate the balance of the module account
`distrKeeper`	`distribution`	Perform bank transfer from Fee Pool
`channelKeeper`	`04-channel`	IBC information
`icaControllerKeeper`	`27-interchain-accounts/controller`	Interchain Account information
`scopedKeeper`	`capability`	Capability keeper
`authority`	-	The authorized address (`gov` module)

The formal specification uses the following state variable to model the keeper.

```
VARIABLES
    \* @type: $bankKeeper;
    bank_keeper,

    \* @type: $channelKeeper;
    channel_keeper,

    \* @type: $icaControllerKeeper;
    ica_controller_keeper,

    \* @type: Seq($ibcPacket);
    ibc_packets,

    \* @type: $accountId;
    authority,

    \* @type: {msg: $msg, success: Bool};
    action
```

- `bank_keeper` models the bank balances of different accounts including module accounts.
- `channel_keeper` models the keeper of the active channels.
- `ica_controller_keeper` models the keeper for ICA controller keeper.
- `ibc_packets` models the queue for IBC packets.
- `authority` models the authorized address.
- `action` models the executed transaction.

`accountKeeper`, `distrKeeper`, `scopedKeeper` are ignored for the sake of simplicity.

The following constants are used in the specification.

```
GOV_ACCOUNT == "gov"
FEE_POOL_ACCOUNT == "fee_pool"
IBC_ESCROW_ACCOUNT == "ibc_escrow"
ENVOY_ACCOUNT == "envoy"

ACCOUNTS == {GOV_ACCOUNT, FEE_POOL_ACCOUNT, IBC_ESCROW_ACCOUNT, ENVOY_ACCOUNT, "Alice", "Bob"}

DENOMS == {"umars", "uosmo"}
```

```
\* connection-0
CONNECTION_ID == 0

REMOTE_MSGS == {"bank/send", "cw/update"}

IBC_TRANSFER_PORT == "transfer"
ICA_CONTROLLER_PORT == "ica-controller"
```

Constant string IDs are used for module accounts for governance module, community fee pool, ibc escrow account, envoy module account.

Some additional accounts, `Alice` and `Bob`, are included as non-module accounts. Two denoms `umars` and `uosmo` are used to model multi-denom balances.

The specification models the application behavior on a single connection. It assumes the connection is already established with ID `connection-0`.

Two different strings are used to model multiple types of sdk messages to execute on the interchain account.

Lastly, two unique strings are used to model the port IDs for transfer channel and interchain account channel.

The model state is initialized with the following predicate.

```
Init ==
    \E _channel_id \in Nat:
    \E _bank_keeper \in [ACCOUNTS -> [DENOMS -> 0..10]]:
        /\ bank_keeper = _bank_keeper
        /\ channel_keeper = SetAsFun({<<_channel_id, [connection_id |-> CONNECTION_ID, port |->
↪  IBC_TRANSFER_PORT]>>})
        /\ ica_controller_keeper = SetAsFun({})
        /\ ibc_packets = <<>>
           \* Envoy authority is set to gov module account
        /\ authority = GOV_ACCOUNT
        /\ action = [msg |-> Variant("Genesis", 0), success |-> TRUE]
```

- `bank_keeper` is initialized with accounts with multiple denoms with arbitrary balances.
- `channel_keeper` is initialized with a single token transfer channel (ICS20) with an arbitrary channel ID.
- `ica_controller_keeper` is initialized as empty.
- `ibc_packets` is initialized as empty.
- `authority` is initialized to the governance module account.
- `action` is initialized with an empty action called *Genesis*.

Transaction

There are three transaction types for the Envoy module.

- `MsgRegisterAccount`
- `MsgSendFunds`
- `MsgSendMessages`

`ValidateBasic` and `GetSigners` methods are implemented for these transactions. These interface methods are responsible for validating submitted transactions during blockchain runtime.

`ValidateBasic` implementations check if the provided account addresses are valid. Also, they check if the sent fund is non-empty for `MsgSendFunds` and if the sent list of messages is non-empty for `MsgSendMessages`, and contains valid messages.

`GetSigners` implementations return the `Sender` field for `MsgRegisterAccount` and the `Authority` field for `MsgSendFunds` and `MsgSendMessages`.

The application logic for these transaction types is implemented at `msg_server.go`.

In the specification, the transaction effects are modeled as a disjunction of three different operators corresponding to different transactions `RegisterAccountNext`, `SendFundsNext` and `SendMessagesNext`.

```
Next ==
    \/ RegisterAccountNext
    \/ SendFundsNext
    \/ SendMessagesNext
```

Each operator is described along with the corresponding transactions.

`MsgRegisterAccount`

This takes an IBC connection ID as input. The app logic uses the connection ID to register the interchain account of the Envoy module account at that connection ID.

An icacontroller transaction is created using the input and then executed. The IBC events are emitted for relayers to listen and act accordingly.

The effect of this transaction is modeled as follows.

```
RegisterAccountNext ==
    \E _connection_id \in Nat:
    \E _new_channel_id \in Nat:
        LET
        _msg == Variant("RegisterAccount", [connection_id |-> _connection_id])
        _is_success ==
                \* the connection ID must be active
            /\ _connection_id \in {CONNECTION_ID}
                \* there should not be an existing ICA
            /\ _connection_id \notin DOMAIN ica_controller_keeper
                \* the new channel ID must be unused
            /\ _new_channel_id \notin DOMAIN channel_keeper
        IN
            IF _is_success THEN
                /\ channel_keeper' = channel_keeper @@ (_new_channel_id :> [connection_id |->
↪  _connection_id, port |-> ICA_CONTROLLER_PORT])
                /\ ica_controller_keeper' = ica_controller_keeper @@ (_connection_id :>
↪  _new_channel_id)
                /\ action' = [msg |-> _msg, success |-> TRUE]
                /\ UNCHANGED <<bank_keeper, ibc_packets, authority>>
            ELSE
                /\ action' = [msg |-> _msg, success |-> FALSE]
                /\ UNCHANGED <<bank_keeper, channel_keeper, ica_controller_keeper, ibc_packets,
↪  authority>>
```

An arbitrary connection ID is chosen as input. `_is_success` is defined to be the precondition for a successful execution. The transaction is successful when

- The connection ID is active.
- There is no other ICA on the same connection ID.
- There is a new channel ID to create a new ICA channel over the connection.

If the precondition is met, the state variables are updated accordingly. The `msg` field of `action` variable is updated to be the transaction message and the `success` field is set accordingly. For a failure, the state variables are unchanged except the `action` variable.

`MsgSendFunds`

As inputs, this takes an IBC channel ID, an account address as the authority that submitted this transaction, and a set of coins to be sent from the module account.

IBC channel ID is required because the IBC denoms sent over different channels are non-fungible. So a specific channel ID is necessary.

The application logic checks and rejects the transaction if the submitter address is not equal to the authority address maintained by the module keeper - which is set to the Governance module account.

The transaction is also rejected if the IBC channel is multi-hop. Mars Hub does not support multi hop channels and intends to connect to their outposts directly.

The app logic performs a successful IBC transfer to the interchain account if the bag of coins is non-empty. If the module account does not have the required minimum balance for the transfer, the remaining balance is funded from the Community Fee pool.

The application logic uses ibc-transfer app to send the tokens. Unfortunately, it does not support multi-coin transfers. So the coins are sent one by one iteratively.

For each ibc-transfer execution, the IBC events are emitted for relayers to listen and act accordingly.

The effect of this transaction is modeled as follows.

```
SendFundsNext ==
    \E _account \in DOMAIN bank_keeper:
    \E _channel_id \in Nat:
    \E _amount \in [DENOMS -> Nat]:
        LET
        _msg == Variant("SendFunds", [authority |-> _account, channel_id |-> _channel_id,
↪  amount |-> _amount])
        _old_envoy_balance == bank_keeper[ENVOY_ACCOUNT]
        _new_envoy_balance == [_d \in DENOMS |-> Max(_old_envoy_balance[_d] - _amount[_d], 0)]
        _envoy_short_fall == [_d \in DENOMS |-> Min(_old_envoy_balance[_d] - _amount[_d], 0)]
        _old_fee_pool_balance == bank_keeper[FEE_POOL_ACCOUNT]
        _new_fee_pool_balance == [_d \in DENOMS |-> Max(_old_fee_pool_balance[_d] +
↪  _envoy_short_fall[_d], 0)]
        _fee_pool_short_fall == [_d \in DENOMS |-> Min(_old_fee_pool_balance[_d] +
↪  _envoy_short_fall[_d], 0)]
        _old_ibc_escrow_balance == bank_keeper[IBC_ESCROW_ACCOUNT]
        _new_ibc_escrow_balance == [_d \in DENOMS |-> _old_ibc_escrow_balance[_d] +
↪  _amount[_d]]
        _is_success ==
               \* the sent funds must be non-empty
            /\ ~IsEmpty(_amount)
               \* the accounts must have enough balance for the transfer
            /\ AllPositiveAmount(_fee_pool_short_fall)
               \* the submitter must be the authority
            /\ _account = authority
               \* the ICAccount must exist
            /\ channel_keeper[_channel_id].connection_id \in DOMAIN ica_controller_keeper
               \* the channel_id must be active
            /\ _channel_id \in DOMAIN channel_keeper
               \* the channel id must be for ibc-transfer
            /\ channel_keeper[_channel_id].port = IBC_TRANSFER_PORT
        IN
        IF _is_success THEN
            /\ bank_keeper' = [bank_keeper EXCEPT
                ![ENVOY_ACCOUNT] = _new_envoy_balance,
                ![FEE_POOL_ACCOUNT] = _new_fee_pool_balance,
                ![IBC_ESCROW_ACCOUNT] = _new_ibc_escrow_balance
                ]
            /\ action' = [msg |-> _msg, success |-> TRUE]
            /\ ibc_packets' = Append(ibc_packets, Variant("SendFunds", [remote |->
↪  ENVOY_ACCOUNT, amount |-> _amount]))
```

```
            /\ UNCHANGED <<channel_keeper, ica_controller_keeper, authority>>
        ELSE
            action' = [msg |-> _msg, success |-> FALSE]
            /\ UNCHANGED <<bank_keeper, channel_keeper, ica_controller_keeper, ibc_packets,
↪  authority>>
```

An account, multi-denom amount and a channel ID are chosen arbitrarily. The transaction is successful when,

- The multi-denom amount is non-empty.
- The updated envoy module account and community fee pool balance is positive.
- The performing account is authorized.
- The channel ID must be active and for ibc token transfer.

`MsgSendMessages`

As inputs, this takes an IBC connection ID, an account address as the authority that submitted this transaction, and a list of messages to be executed at the interchain account.

A connection ID is enough, as the effect of ICA transaction execution is fungible over different channels. The application logic uses icacontroller app to send the list of messages to the outpost.

As `MsgSendFunds`, the transaction is rejected if the authority does not match. The transaction is also rejected when the submitted messages are invalid Cosmos-SDK messages.

As before, IBC events from the ICA transaction submission are emitted for relayers to listen and act accordingly.

The effect of this transaction is modeled as follows.

```
SendMessagesNext ==
    \E _account \in DOMAIN bank_keeper:
    \E _connection_id \in Nat:
    \E _remote_msgs_set \in SUBSET REMOTE_MSGS:
        LET
        _remote_msgs == SetToSeq(_remote_msgs_set)
        _msg == Variant("SendMessages", [authority |-> _account, connection_id |->
↪  _connection_id, messages |-> _remote_msgs])
        _is_success ==
                \* the submitter must be the authority
            /\ _account = authority
                \* the ICAccount must exist
            /\ _connection_id \in DOMAIN ica_controller_keeper
                \* the channel id must be for ICA
            /\ channel_keeper[ica_controller_keeper[_connection_id]].port = ICA_CONTROLLER_PORT
        IN
        IF _is_success THEN
            /\ action' = [msg |-> _msg, success |-> TRUE]
            /\ ibc_packets' = Append(ibc_packets, Variant("SendMessages", [remote |->
↪  ENVOY_ACCOUNT, messages |-> _remote_msgs]))
            /\ UNCHANGED <<bank_keeper, channel_keeper, ica_controller_keeper, authority>>
        ELSE
            action' = [msg |-> _msg, success |-> FALSE]
            /\ UNCHANGED <<bank_keeper, channel_keeper, ica_controller_keeper, ibc_packets,
↪  authority>>
```

An account, a sequence of remote messages and a connection ID are chosen arbitrarily. The transaction is successful when,

- The account is authorized.
- The connection ID is active.
- The connection ID has an active interchain account.

Query

The Envoy module provides two query APIs. One is to query the interchain account of the module account corresponding to an IBC connection. The other one lists all active interchain accounts of the Envoy module account.

The application logic for these query types is implemented at `query_server.go`.

`QueryAccountRequest`

This takes an IBC connection ID as input. It computes the interchain account port ID of the module account. Then it uses `GetInterchainAccountAddress` from `icaControllerKeeper` to prepare the information for the corresponding interchain account.

`QueryAccountsRequest`

This does not take any input. It uses `GetAllInterchainAccounts` from `icaControllerKeeper` to iterate over all interchain accounts and filters matching the correct interchain account port ID for the Envoy module account.

Afterward, it prepares information for each interchain account, similar to the single account query.

Invariants

The formal specification includes some critical invariants.

Operator ID	Invariant	Description
`invariant-CS`	`ConstantSupply`	Token supply on the chain never changes.
`invariant-PB`	`PositiveBalance`	Balances are always non-negative.
`invariant-VA`	`ValidAuthority`	SendFunds and SendMessages have the correct authority.
`invariant-IE`	`ICAExists`	SendFunds and SendMessages are submitted to an existing ICAccount.
`invariant-PL`	`NoPacketLoss`	IBC packets of SendFunds and SendMessages are submitted in the IBC queue.

The specification is verified against the above invariants using the Apalache model checker.

Methodology

Vulnerability classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

ImpactScore	Examples
High	Halting of the chain; loss, locking, or unauthorized withdrawal of funds of many users; arbitrary transaction execution; forging of user messages / circumvention of authorization logic
Medium	Temporary denial of service / substantial unexpected delays in processing user requests (e.g. many hours/days); loss, locking, or unauthorized withdrawal of funds of a single user / few users; failures during transaction execution (e.g. out of gas errors); substantial increase in node computational requirements (e.g. 10x)
Low	Transient unexpected delays in processing user requests (e.g. minutes/a few hours); Medium increase in node computational requirements (e.g. 2x); any kind of problem that affects end users, but can be repaired by manual intervention (e.g. a special transaction)
None	Small increase in node computational requirements (e.g. 20%); code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation

Exploitability Score

The Exploitability score reflects the ease and technical means by which the vulnerability can be exploited; it represents the characteristics of the vulnerable component. In the below table we list, for each category, examples of actions by actors that are enough to trigger the exploit. In the examples below:

- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be
  - *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
  - *illegitimate*, e.g. submission of a specially crafted transaction (not following the protocol, or e.g. with large/incorrect values); voting on two different alternative blocks; alteration of relayed messages.
- We employ also a *qualitative measure* representing the amount of certain class of power (e.g. possessed tokens, validator power, relayed messages): *small* for $< 3\%$; *medium* for 3-10%; *large* for 10-33%, *all* for $>33\%$. We further quantify this qualitative measure as relative to the largest of the system components. (e.g. when two blockchains are interacting, one with a large capitalization, and another with a small capitalization, we employ *small* wrt. the number of tokens held, if it is small wrt. the large blockchain, even if it is large wrt. the small blockchain)

ExploitabilityScore	**Examples**
High	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
Medium	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
Low	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
None	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

The severity score combines the above two sub-scores into a single value, and roughly represents the probability of the system suffering a severe impact with time; thus it also represents the measure of the urgency or order in which vulnerabilities need to be addressed. We assess the severity according to the combination scheme represented graphically below.

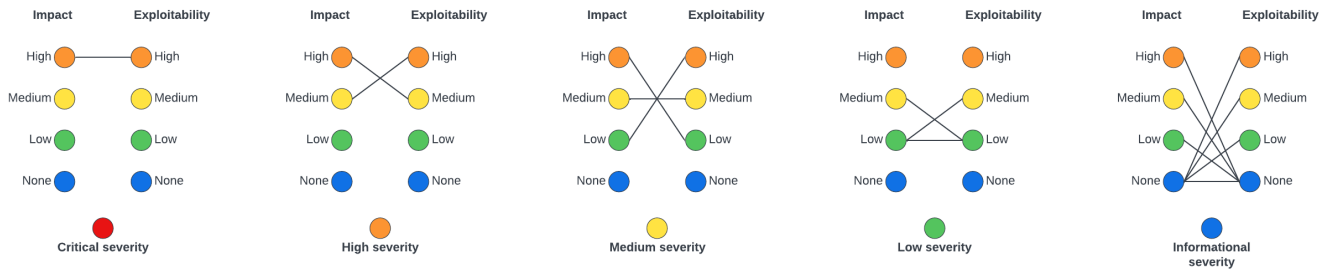

Figure 1: Severity classification

As can be seen from the image above, only a combination of high impact with high exploitability results in a Critical severity score; such vulnerabilities need to be addressed ASAP. Accordingly, High severity score receive vulnerabilities with the combination of high impact and medium exploitability, or medium impact, but high exploitability.

SeverityScore	**Examples**
Critical	Halting of chain via a submission of a specially crafted transaction

SeverityScore	Examples
High	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
Medium	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
Low	2x increase in node computational requirements via coordinated withdrawal of all user tokens
Informational	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Golang
- **Artifacts**
  - `/x/envoy` module at commit `c7795c`

Engagement Summary

- **Dates**: 23.01.2023 to 03.02.2023
- **Method**: Manual code review & automated testing
- **Employees Engaged**: 2

Severity Summary

Finding Severity	#
Critical	0
High	1
Medium	1
Low	0
Informational	3
Total	**5**

Findings

Severity	Finding
High	Iterate over all Interchain Accounts
Medium	Mars Hub as Interchain Account Host chain
Informational	Redundant use of ScopedKeeper
Informational	Inconsistent Gov module CLI help
Informational	Non-exhaustive unit tests

Contributions

For the **High Severity** finding, we contributed a pull request which fixes the issue.

Apart from these findings, we identified the module did not have an end-to-end test. So we also contributed another pull request to introduce an end-to-end test to be run at GitHub action workflow.

Iterate over all Interchain Accounts

ID	IF-MARSPROTOCOL-01
Severity	High
Impact	Medium
Exploitability	High
Type	Implementation
Issue	#41
Status	Resolved

Involved artifacts

- Multiple interchain accounts query

Description

The Envoy module offers a query API where one can fetch all available interchain accounts of the module.

To find them, the implementation loops over all existing interchain accounts on the chain at x/envoy/keeper/query_server.go

```
allAccounts := qs.k.icaControllerKeeper.GetAllInterchainAccounts(ctx)
accounts := []*types.AccountInfo{}
for _, account := range allAccounts {
    if account.PortId == portID {
        account, err := qs.queryAccount(ctx, account.ConnectionId, portID)
```

Problem Scenarios

Since interchain account creation is permissionless, a malicious actor may create thousands of empty wallets and register interchain accounts for them.

Then, they can query the API that will iterate over all interchain accounts, including the ones created by the actor. This allows the malicious actor to enforce an arbitrarily long computation at a public RPC node.

Recommendation

- `ICAControllerKeeper` should provide a prefix or account filter to iterate over existing interchain accounts.
- Meanwhile, a similar method `GetAllChannelsWithPortPrefix`, from `ChannelKeeper`, may be used to find the corresponding connections of the interchain accounts of the Envoy module.

Mars Hub as Interchain Account Host chain

ID	IF-MARSPROTOCOL-02
Severity	Medium
Impact	Medium
Exploitability	Medium
Type	Protocol
Status	Acknowledged

Involved artifacts

- Mars Hub application

Description

Interchain Account has two components - the Controller chain and the Host chain.

- A chain is a Controller if it allows its accounts to register their corresponding interchain accounts on a Host chain.
- A chain is a Host if it allows the accounts from a Controller chain to execute transactions on itself.

A Cosmos-SDK chain may choose to implement either the Controller or the Host chain, or both.

The Envoy module requires Mars Hub to implement the Controller chain.

But additionally, Mars Hub also implements the Host chain.

Problem Scenarios

A Host chain maintains a whitelist of the transactions that are allowed to be executed as interchain account messages.

It is best to avoid transactions that also touch the interchain account capabilities - such as the Envoy module or the Interchain Account module itself.

Another example is whitelisting MsgDelegate messages will enable liquid staking - which may or may not be a desirable effect.

Recommendation

Mars Hub should add a well-thought-out transaction whitelist in the Interchain Account Host implementation.

Redundant use of ScopedKeeper

ID	IF-MARSPROTOCOL-03
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Commit	`@f89a2c5`
Status	Resolved

Involved artifacts

- Envoy module keeper

Description

`ScopedKeeper` is used to claim capabilities. This is usually done to avoid malicious application modules hijacking the ports of a different application and sending spoofed packets.

It is used in the Envoy module keeper at x/envoy/keeper/keeper.go.

```
type Keeper struct {
    cdc codec.Codec

    accountKeeper       authkeeper.AccountKeeper
    bankKeeper          bankkeeper.Keeper
    distrKeeper         distrkeeper.Keeper
    channelKeeper       ibcchannelkeeper.Keeper
    icaControllerKeeper icacontrollerkeeper.Keeper
    scopedKeeper        capabilitykeeper.ScopedKeeper

    // The baseapp's message service router.
    // We use this to dispatch messages upon successful governance proposals.
    router *baseapp.MsgServiceRouter

    // The account who can execute envoy module messages.
    // Typically, this should be the x/gov module account.
    authority string
}
```

Problem Scenarios

The Envoy module does not create new IBC ports. The Envoy module uses ibc-transfer and ica-controller to send tokens and messages over IBC channels indirectly.

So it is redundant for the Envoy module keeper.

Recommendation

The dependency on the `ScopedKeeper` can be removed in the Envoy module keeper.

Inconsistent Gov module CLI help

ID	IF-MARSPROTOCOL-04
Severity	Informational
Impact	None
Exploitability	None
Type	Documentation
Issue	#15032
Status	Resolved

Involved artifacts

- Mars Hub's custom `Metadata` parsing
- `ProposalMetadata` struct
- Gov module CLI help

Description

The Envoy module whitelists only the Governance module account address to submit `MsgSendFunds` and `MsgSendMessages`. So it is in the Envoy module's interest that the Governance module has proper documentation.

Mars Hub modifies the `metadata` field of the governance proposal transaction from a base64 encoded string to a JSON string with compulsory values.

At x/gov/types/metadata.go,

```
// ProposalMetadata defines the required schema for proposal metadata.
type ProposalMetadata struct {
    Title             string   `json:"title"`
    Authors           []string `json:"authors"`
    Summary           string   `json:"summary,omitempty"`
    Details           string   `json:"details"`
    ProposalForumURL  string   `json:"proposal_forum_url,omitempty"`
    VoteOptionContext string   `json:"vote_option_context,omitempty"`
}
```

At x/gov/keeper/msg_server.go,

```
func (ms msgServer) SubmitProposal(goCtx context.Context, msg *govv1.MsgSubmitProposal)
↪  (*govv1.MsgSubmitProposalResponse, error) {
    // the metadata string must not be empty. attempt to deserialize it using
    // the given schema return error if fails.
    if _, err := types.UnmarshalProposalMetadata(msg.Metadata); err != nil {
        return nil, sdkerrors.Wrap(types.ErrInvalidMetadata, err.Error())
    }

    // if metadata is good, we just hand over the rest to the vanilla msgServer
    return govkeeper.NewMsgServerImpl(ms.k.Keeper).SubmitProposal(goCtx, msg)
}
```

Problem Scenarios

Although Mars Hub modifies the governance proposal, the CLI help message for the governance module remains unchanged.

`marsd tx gov submit-proposal --help` prints the following.

```
...
Example:
$ marsd tx gov submit-proposal path/to/proposal.json

Where proposal.json contains:

{
  // array of proto-JSON-encoded sdk.Msgs
  "messages": [
    {
      "@type": "/cosmos.bank.v1beta1.MsgSend",
      "from_address": "cosmos1...",
      "to_address": "cosmos1...",
      "amount":[{"denom": "stake","amount": "10"}]
    }
  ],
  "metadata: "4pIMOgIGx1vZGU=", // base64-encoded metadata
  "deposit": "10stake"
...
```

A user may be confused using a base64 string for the `metadata` field - even with base64 encoding of a valid JSON string.

Recommendation

Mars Hub should update the CLI help message for the governance module to reflect the changes for the `metadata` field.

Non-exhaustive unit tests

ID	IF-MARSPROTOCOL-05
Severity	Informational
Impact	None
Exploitability	None
Type	Practice
Commit	`@8b64dad`
Status	Resolved

Involved artifacts

- Unit tests

Description

The unit tests for `MsgSendFunds` and `MsgSendMessages` always use the Envoy module account for the `Authority` field.

At `x/envoy/types/tx_test.go`

```
testAuthority    = authtypes.NewModuleAddress(types.ModuleName)
```

At `TestValidateBasic` in `x/envoy/types/tx_test.go`

```
{
    "MsgSendMessages - success",
    &types.MsgSendMessages{
        Authority:    testAuthority.String(),
        ConnectionId: testConnectionId,
        Messages:     []*codectypes.Any{testValidMsg},
    },
    true,
},
{
    "MsgSendMessages - messages is empty",
    &types.MsgSendMessages{
        Authority:    testAuthority.String(),
        ConnectionId: testConnectionId,
        Messages:     []*codectypes.Any{},
    },
    false,
},
{
    "MsgSendMessages - message does not implement sdk.Msg interface",
    &types.MsgSendMessages{
        Authority:    testAuthority.String(),
        ConnectionId: testConnectionId,
        Messages:     []*codectypes.Any{testInvalidMsg},
    },
    false,
},
```

At TestGetSigners in x/envoy/types/tx_test.go

```
{
    "MsgRegisterAccount",
    &types.MsgRegisterAccount{
        Sender:       testSender.String(),
        ConnectionId: testConnectionId,
    },
    testSender,
},
{
    "MsgSendFunds",
    &types.MsgSendFunds{
        Authority: testAuthority.String(),
        ChannelId: testChannelId,
        Amount:    sdk.NewCoins(),
    },
    testAuthority,
},
{
    "MsgSendMessages",
    &types.MsgSendMessages{
        Authority:    testAuthority.String(),
        ConnectionId: testConnectionId,
        Messages:     []*codectypes.Any{},
    },
    testAuthority,
},
```

These unit tests also pass with other non-authorized and authorized accounts.

Problem Scenarios

The tests are not exhaustive. Multiple addresses should be used for the authority field.

Recommendation

The tests should include more cases with different account addresses.