informal SYSTEMS

Security Audit Report

Router: Q4 2023

Authors: Darko Deuric, Josef Widder, Marko Juric

Last revised 9 January, 2024

Table of Contents

Title	Router: Q4 2023
Client	Router Protocol
Team	Darko Deuric Josef Widder Marko Juric
Start	09 Oct 2023
End	03 Nov 2023
Domains	CosmWasm Cosmos SDK EVM / Solidity Bridges
Methods	Code review Architecture design
Languages	Go Rust Solidity
Report	

Audit Overview

The Project

In October 2023, the Router development team collaborated with Informal Systems to initiate a security audit of the Router's Voyager product. Voyager is a permissionless cross-chain asset swap protocol, encompassing two distinct flows: the asset forwarder and the asset bridge.

The asset forwarder flow involves reverse verifications, while the asset bridge is designed for situations where there is minting capability present on either the source or destination contracts.

Scope of this audit

The audit was conducted by the following individuals:

- Darko Deuric
- Josef Widder
- Marko Juric

The agreed-upon work plan consisted of the following tasks which included evaluating and analyzing the code and specifications :

- Router chain module: x/voyager
- Asset forwarder contract (Solidity) - used for creating deposits and emitting events on source chain
- Middleware contract (asset forwarder flow) - with main purpose of processing the observed events emitted by asset forwarder contract from source chain and forwarder golang service and doing a reverse verification
- Voyager forwarder (asset forwarder flow) - off-chain service responsible for relaying cross-chain messages and handling associated token transactions across different blockchain networks.
- Asset bridge contract (Solidity) - used for creating deposits in asset bridge flow
- Middleware contracts (asset bridge and fee manager) - with main purpose of processing the observed events from gateway contract from source chain (which was previously called by asset bridge contract) and transmitting the requests to the destination chain

Conducted work

During the kickoff meeting, the Router team introduced us to the Router's Voyager protocol and provided the codebase that required auditing. Prior to commencing the audit, our team needed some time to thoroughly comprehend the Voyager flows, which encompass several critical components, including an off-chain element such as the Forwarder Golang service.

Our team conducted a meticulous, high-quality, line-by-line manual code review with a primary focus on code accuracy and critical point analysis specific to each component. We maintained communication through a shared Slack channel, discussing various details, coordinating sync meetings, sharing information on the GitHub audit repository, and more.

Additionally, the Router team supplied diagrams for both the asset forwarder and asset bridge flows, which greatly enhanced our understanding of the project.

Beyond the intensive code review, we dedicated our efforts to reconstruct the protocol with the main focus on checking the correctness of two main flows: the asset forwarder and the asset bridge. This involved an in-depth analysis of both successful and unsuccessful execution paths. We endeavored to identify and verify some of the system-level invariants and examine the liveness and design properties of the entire system to ensure a comprehensive understanding and validation of the protocol's operational integrity.

Conclusions

In the course of our comprehensive audit of the Router's Voyager protocol, we meticulously reviewed various components and processes, and here is the summary of our findings:

- **Critical Issues (3 Identified):** We have detected three critical issues that pose significant risks and should be rectified immediately to ensure the security and functionality of the system.
- **High Severity Issues (6 Identified):** Six high-severity issues have been flagged that, while not critical, still require urgent attention to mitigate potential threats and operational inefficiencies.
- **Medium, Low, and Informational Issues:** There is a spectrum of issues ranging from medium to informational severity. The priority of addressing these issues should be determined by their respective impact on the system and their likelihood of occurrence.

The bulk of the identified issues reside within the `asset-forwarder-middleware` contract, which is fundamental for the accurate accounting and bookkeeping of transactions involving users and forwarders. This contract, in particular, requires careful scrutiny and rectification.

Another vital component demanding thorough examination is the forwarder service. The forwarder's role is integral to the asset forwarding flow, and as such, its robustness and operational continuity are crucial.

The `asset bridge` middleware contract logic also necessitates close inspection, especially concerning the management of failed requests to ensure they are handled effectively and consistently, allowing for user funds' safety and system integrity.

Lastly, a critical problem which can cause a chain halt due to out-of-gas issues exists in the Router Voyager module, which is similar to an issue identified in our previous audit report.

It is also encouraging to note that we did not encounter significant issues with the other components included in our scope of review.

A special acknowledgment is extended to Router developer Gaurav Agarwal for his outstanding cooperation throughout this process. His prompt and knowledgeable responses greatly facilitated our review process, reflecting the team's commitment to quality and security.

Moving forward, we recommend that the Router team prioritize the remediation of identified issues, starting with the most severe, to enhance the security and reliability of the Voyager protocol. Our team remains at disposal for any further consultation or to assist with the implementation of our recommendations.

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Golang, Solidity, Rust
- **Artifacts:**
    - Router chain voyager module, commit: b982cd
    - Asset forwarder solidity contract repo, commit: 4db051
    - Asset forwarder middleware contract repo, commit: 0d9be1
    - Voyager asset bridge contracts repo, commit: 885b62
    - Voyager forwarder golang service repo, commit: e60d0e

Engagement Summary

- **Dates**: 09.10.2023 to 03.11.2023
- **Method**: Manual code review, protocol analysis
- **Employees Engaged**: 3

Severity Summary

Finding Severity	**#**
Critical	3
High	6
Medium	4
Low	2
Informational	3
Total	**18**

System overview

Asset Forwarder Flow

Components:

- **AssetForwarder EVM contract:** This is the main contract on both the source and destination chains with which either a user or a forwarder interacts.
- **AssetForwarderMiddleware (AFM) CW contract:** Deployed on the Router chain, this contract is primarily involved in the management of cross-chain communications and accounting checks.
- **Router chain - voyager module:** A core component that interacts with the AFM and ensures that the messages are relayed correctly between chains.
- **Forwarder:** Listens to events from the AssetForwarder EVM contract, performs validations, and sends data to the destination chain.
- **Orchestrator:** Listens to events from both source and destination chains and submits messages to the Router chain in a sequenced manner.
- **Relayer:** Relays messages together with orchestrators signatures.

Steps:

1. A user initiates a function like `iDeposit` on the source AF EVM contract, triggering an event.
2. Forwarders listen to this event. They transform the data, validate it, and conduct profitability checks.
3. If deemed suitable, the forwarder invokes `iRelay` to forward this data to the destination chain.
4. Meanwhile, Orchestrators listen to events on both the source and destination chains. They pass messages to the Router chain sequentially.
5. The Router's voyager module, upon detecting events like `FundDeposit` or `FundsPaid`, triggers the AFM CW contract.
6. The AFM manages accounting and verifies if the forwarder has performed its duties accurately.
7. Forwarder can invoke `WithdrawLiquidity` to retract its funds after successful transfer is been made , or users can activate `CreateRefundRequest` if there's a transfer delay.

Forwarder

The Voyager Forwarder component is a golang service responsible for relaying cross-chain messages and handling associated token transactions across different blockchain networks.

Service starts with `run` function. In essence, it sets up a comprehensive system to listen to multiple blockchains, detect relevant events, and then relay those events or data between the chains as needed. The system is modular, with each chain having its **listener** and **relayer**, and a centralized **dispatcher (processor)** managing the relay tasks. Proper logging, error handling, and health monitoring ensure the robustness and reliability of the relay system.

Listener
Its primary role is to set up the infrastructure to listen to and process events from different blockchains, specifically from the "Voyager" contract. `Start` function starts the `VoyagerListener` and:

1. fetch the last block that was processed by the listener from the database using the

`GetLastProcessedBlock` method. This is to ensure that the listener continues from where it left off, avoiding unnecessary reprocessing.

2. calls the `ProcessInboundEvents` method of the `eventProcessor`, passing in the last processed block height and event nonce fetched from local db. The `ProcessInboundEvents` method is responsible for continually fetching, transforming, and processing events from a source chain:
    a. an infinite loop to continuously fetch and process events.

b. calculates the starting and ending blocks to be queried
c. calls `FetchAndAddToDBVoyagerEvents` to query for events between the start and end blocks and adds them to the database
    i. retry event queries
    ii. After successfully querying the events, it processes two types of events: `fundDepositedEvents` and `fundDepositedWithMessageEvents` => creates a `FundRelay` struct, which represents a relay transaction, and adds it to the database (or transaction queue)
    iii. For each processed event, it updates the `lastProcessedEventNonce` to keep track of the highest processed event nonce
d. The `lastQueriedBlock` and `lastProcessedEventNonce` variables are updated based on the end block and new processed event nonce, respectively
e. note: if `ProcessInboundEvents` returns an error, the `Start` method will end, and the goroutine it's running in will also terminate.

Processor

Designed for continuous processing of requests from a database queue (`msgTxqArray`). `Start` function starts the `RequestProcessor` and:

1. starts a new goroutine with `go func() { ... }` that runs indefinitely due to the outer `for` loop.
2. Inside the goroutine:
    a. It queries the database for all unprocessed transactions using `GetTxqByStatus(store.Unprocessed)` . If there's an error fetching these unprocessed transactions, it panics, stopping the current goroutine
    b. For each unprocessed transaction status is updated to `Picked` in the database. This might be done to prevent other processors from picking up the same transaction if there are multiple instances running.
    c. Depending on the `TxType` of the transaction, a different method (`ProcessIRelayRequest` or `ProcessIRelayMessageRequest`) is called to process the transaction. It uses a registry to determine the appropriate destination chain handler, and then adds a message to a channel on that handler via the `AddIRelayFundsToExecutorChannel` method..

Relayer

Designed to continuously run and process messages from two channels (`IRelayRequestChannel` and `IRelayMsgRequestChannel`). It processes messages for relaying funds and relaying funds with messages. `Start` function starts the `ChainRelayer` and:

1. continuously monitor and process incoming messages.
2. Within this loop, the `select` statement is used to await messages from two channels. It processes the first available message and then continues to wait for the next.
3. When a message is received on `IRelayRequestChannel` , the function transforms and validates the message. If there's an error, it logs the error and continues to wait for the next message. If the message is valid, it invokes the `IRelay` method to process the message.
4. Similarly, when a message is received on `IRelayMsgRequestChannel` , the function processes it, but this time for relaying funds with a message.

Executor

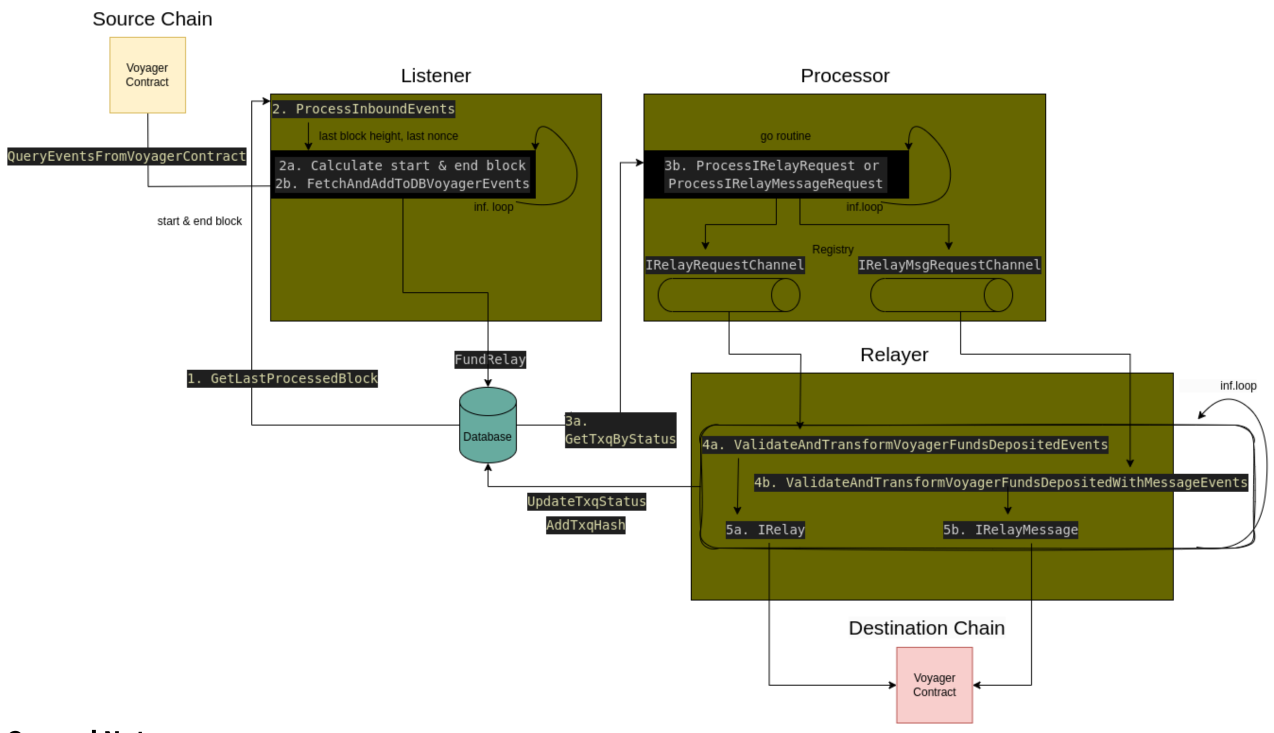
Part of chain's relayer which:

1. constructs the data for the transaction using ABI packed bytes.
2. It then estimates the gas price and gas limit for the upcoming transaction.
3. check if the transaction is profitable and hold off the transaction if it's not (which is commented-out!)
4. sends the transaction using the `assetForwarderWrapper`.
5. waits for the transaction to be mined and fetches the receipt.
6. Throughout the process, the function interacts with the database to update transaction statuses, errors, and hashes.

Example:

Suppose a forwarder finalizes 100 requests, with each request transferring 50 tokens to the destination. In the middleware, the forwarder can then request 5,000 tokens to be released on the source chain and directed to its address. Notably, the destination's EVM Contract will compute and dispatch a message hash to the Router. Also, the AFM doesn't manage fees.

The simplified process can be seen on the diagram below.



General Notes:

- There can be numerous forwarders facilitating the cross-chain communication, leading to a decentralized and trust-less setup.
- Only a singular forwarder is permitted to deliver transfer data; any duplicate attempts, either by the same or different forwarders, will fail.
- Should a forwarder alter the data maliciously, the AFM will detect this and will not confirm or complete the tampered message.
- Forwarders might not always transfer data in sequence. They often prioritize more profitable transfers. Thus, some deposits might be overlooked if they're not deemed profitable.
- `depositNonce` is incremented post every deposit or relay, serving as a unique identifier for transactions.
- Orchestrators ensure that events are relayed in sequence to the AFM, guaranteeing order in requests received by the AFM.

Asset Bridge Flow

Components:

- **AssetBridge EVM contract:** It's the contract users interact with for initiating transfers and swaps.
- **RouterGateway EVM contract:** This contract serves as an intermediary, facilitating communication between the AssetBridge contract and the Router chain.
- **AssetBridge CW contract:** Deployed on the Router chain, it handles cross-chain requests and accounting.
- **Orchestrator**
- **Relayer**

Steps:

1. A user initiates a function, such as `transferToken`, on the source AssetBridge.
2. This calls the RouterGateway contract, emitting an event.
3. Orchestrators listen to these events and send messages to the Router chain in sequence.
4. The AssetBridge CW contract on the Router chain is then activated.
5. `HandleIReceive` is executed on the AssetBridge CW contract. Here, fees are deducted, and a cross-chain call is made.
6. The Router chain creates an outbound cross-chain request, which the relayer picks up.
7. The RouterGateway contract then receives this request and activates `iReceive` on the AssetBridge EVM destination contract.

Details:

1. **Initiation via AssetBridge.sol:**

The process commences with one of the following functions on the `AssetBridge.sol` contract:

- `transferToken`
- `transferTokenWithInstruction`
- `swapAndTransferToken`
- `swapAndTransferTokenWithInstruction`

Key points:

- **Token type differentiation:** The `transferToken` function in the AssetBridge contract differentiates between native and non-native tokens:
    - Non-native tokens:
        - If the token is non-burnable, it's **locked** in the AssetBridge contract.
        - If the token is burnable, it's **burned** from the sender's address.

    -
      ```
      if (!isSourceNative) {
          lockOrBurnToken(transferPayload.srcTokenAddress, msg.sender,
      transferPayload.srcTokenAmount);
      }
      ```
    - Native tokens (e.g., Ethereum):
        - The contract checks the sent native token amount.
        - It then wraps the native token into its ERC-20 equivalent (WETH) for consistent handling.

    -
      ```
      else {
      ```

```
    IWETH(WETH).deposit{ value: transferPayload.srcTokenAmount }();
    transferPayload.srcTokenAddress = WETH;
}
```

  - In short, assets are either locked or burned on the source chain during the bridging process, ensuring proper asset handling.
- **ABI encoded packet creation:** For the `transferToken` function, an ABI encoded packet is created containing:
  - `tx_code` : 0
  - `destChainIdBytes`
  - `srcTokenAddress`
  - `srcTokenAmount`
  - `sender`
  - `recipient`
  - `depositNonce` : A unique identifier for every packet sent from the `AssetBridge` contract.
  - `widgetId`
  - For the `transferTokenWithInstruction` function:
    - An additional `gasLimit` and `instruction` data are added.
    - `tx_code` : 1
    - All other fields remain analogous to the `transferToken` function.

2. **<u>Communication with the Gateway via gateway.iSend:</u>**

After packet creation, the `gateway.iSend` function is invoked. This function takes in the previously encoded packet, combined with the destination address, which is the `routerBridge` contract's address stored in the `AssetBridge` contract.

- **Event emission:** The `gateway.iSend` emits an event. This event is crucial because it's picked up by the orchestrators. The event includes:
  - `eventNonce` : Unique for every operation on the Gateway contract such as `updateValset` , `setDappMetadata` , `iReceive` , `iAck` , and `iSend` .
  - The unchanged packet received from `AssetBridge.sol` .

3. **<u>Orchestrator Processes Gateway Events</u>**

The Orchestrator is pivotal in ensuring that events between chains are monitored and acted upon. Here's a brief overview of its role, particularly in the context of the `iSend` event:

- **Event monitoring**: The Orchestrator constantly keeps tabs on the Gateway, specifically watching for events like the `GatewayUpgradeableISendEvent` associated with the `iSend` function.
- **Event sorting**: Once events are captured, they're sorted by their `EventNonce` . This order encompasses `iSendEvents` , `iReceiveEvents` , `iAckEvents` , `valsetUpdatedEvents` , and `setDappMetadataEvents` .
- **Event transformation**: After ordering the events, the Orchestrator processes the `GatewayUpgradeableISendEvent` to generate a `MsgCrosschainRequest` aimed at the Router chain. This request houses:

`orchestrator` : Orchestrator's address.

`srcChainId` : ID of the source chain.

`contract` : **Gateway Contract's address.**

`requestIdentifier` : Unique `EventNonce` .

`srcBlockHeight` : Height of the block in the source chain.

`sourceTxHash` : Hash of the transaction from the source chain.

`srcTimestamp` : Timestamp from the source transaction.

`srcTxOrigin` : Transaction's originator on the source chain.

`routeAmount` : Amount of the Route token being sent.

`routeRecipient` : Intended recipient of the Route token.

`destChainId` : ID of the destination chain (not the Router chain).

`requestSender` : **Address of the** `AssetBridge` **contract**

`requestMetadata` : ABI-encoded metadata pertaining to the request.

`requestPacket` : The **payload** from the initial `iSend` event, unchanged.

`srcChainType` : Type classification of the source chain.

`destChainType` : Type classification of the destination chain.

4. **Handling on the Router Chain**

Upon the Router chain receiving the `MsgCrosschainRequest` , the `HandleInbound` function is invoked. A `crosschainRequest` is then generated with the status set to `CROSSCHAIN_TX_CREATED` , and the orchestrator's vote is recorded for this request.

As more orchestrators submit their versions of the `MsgCrosschainRequest` , the system checks for consistency among the submissions. Once the submissions from the orchestrators reach a consensus (i.e., when 2/3 + 1 of the orchestrators have sent identical `MsgCrosschainRequest` entries with the same `RequestIdentifier`), the status of the request is updated to `CROSSCHAIN_TX_VALIDATED` .

During the next execution of `EndBlocker` , this validated request is then processed. Any associated fees are deducted, and the designated router bridge contract is invoked.

To determine the correct contract address on the router chain, the system extracts the `cwContractAddress` from the `packet.Handler` . It's important to note that this address was previously set to be the `routerBridge` contract's address (stored within the `AssetBridge` contract) before the `gateway's iSend` was called.

Lastly, the system forwards the `packet.Payload` to the `routerBridge` contract using a sudo message named `HandleIReceive` .

5. **Processing on the Router Bridge Contract**

- **Whitelist verification**: The first layer of validation is ensuring that the `request_sender` is part of a whitelist, using the `is_white_listed_modifier`. This step ensures that only authorized entities can interact with the bridge contract.
- **Chain type handling**: Depending on the `src_chain_id`, the system processes requests from different chains. For requests coming from EVM chains:
  - `handle_evm_inbound_request` is responsible for managing requests. It processes either:
    - Requests with instructions (`encode_type == 1`)
    - Requests without instructions (`encode_type == 0`)
- **Error handling**: If any error arises during data processing on the bridge contract, the system invokes `revert_inbound_to_src_chain`. This function forms a `CrosschainCall` with the `request_packet` encoded to have:
  - `destination_contract_address`: The address of the AssetBridge contract.
  - `contract_call_payload`: Which consists of `tx code = 2`, `dest_chain_id_bytes`, and `deposit_nonce`.
  - `request_metadata`: Includes various details, importantly the `ack_type` set to `AckOnBoth`, indicating that an acknowledgment is expected regardless of the success or failure status from the destination chain.
  - An `outbound_submessage` is created with an ID of `CREATE_OUTBOUND_REPLY_ID`.
- **Happy path execution**:
  - `srcTokenAmount` (originally sent from the source chain's AssetBridge contract) has fees deducted.
  - If the source token (`src_token`) is burnable, the fees get added to the `BURNED_GAS_TOKENS` storage.
  - `dest_token_amount` is adjusted based on the decimal precision difference between the source and destination chains and `srcTokenAmount`.
  - If the `destination_contract_address` is verified as correct, a `CrosschainCall` is assembled and an `outbound_submessage` is dispatched with the ID `CREATE_OUTBOUND_REPLY_ID`.
- **Intermediate actions on Router chain**:
  - The Router chain constructs an outbound `MsgCrosschainRequest` based on the `crosschainCall` assembled on the bridge contract.
  - Fees associated with this outbound request are deducted.
  - A `MsgCrosschainRequestResponse` is generated and dispatched back to the bridge contract, including the `RequestIdentifier` and the `feeDeducted`.
- **Reply mechanism on bridge contract**: Since submessages originate from the bridge contract, the reply mechanism exists to handle responses. After the Router chain completes the outbound request handling, it returns a `MsgCrosschainRequestResponse` to the bridge contract. The `handle_reply` function on the bridge contract processes this using the `CREATE_OUTBOUND_REPLY_ID` variant and archives the `CrosschainCall` into the `OUTBOUND_CALLS_STATE` storage. The storage key used is the `MsgCrosschainRequestResponse.RequestIdentifier`.

6. **<u>Processing acknowledgment and finalizing crosschain request</u>**

- **At the Bridge Contract:**
  - Two possible outcomes arise post-processing:

        - a. A `CrosschainCall` is successfully created (in both the happy scenario and the `revert_inbound_to_src_chain` situation).
        - b. No `CrosschainCall` is formed, and `HandleIReceive` ends with an error, such as when `is_chain_unpaused_modifier` returns an error.
- **At the Router:**
    - Following the processing on the bridge contract, the Router checks if there's a necessity to send an acknowledgment to the source chain. Based on the given `AssetBridge_REQUEST_METADATA` which is hardcoded value on the AssetBridge contract, after decoding it, `ackType = NO_ACK`, so no acknowledgment is dispatched.
- **Destination chain interaction:**
    - The relayer awaits signatures from 2/3 + 1 orchestrators. Upon gathering the necessary signatures, it forwards the payload to the gateway contract.
    - This triggers the `iReceive` function on the `AssetBridge` contract:
    - Depending on the decoded `txType`, the following actions take place:
        - a. Refund the depositor if `txType == 2`. A refund is processed to the `sender` address specified in the `depositData` structure decoded from the `packet`.
        - b. Mint tokens for `txType == 0`. The tokens are minted to the `recipient` specified in the `executeDetails` structure which is decoded from the `packet`.
        - c. Mint tokens and execute the instruction for `txType == 1`. As with `txType == 0`, tokens are minted to the `recipient` specified in the `executeDetails`.
    - Following these actions, an `ack` event is generated and forwarded to the Router chain via Orchestrators.
- **Router chain acknowledgment processing:**
    - The Router chain receives the `ack` event. After undergoing validation, the `ExecuteAck` function is invoked on the Router.
    - This leads to the triggering of the `HandleIAck` function on the bridge contract, marking the conclusion of the crosschain transaction process.

General Notes:

- AssetBridge is specifically used when there's minting capability on either the source or destination contracts since AssetForwarder can't perform reverse verifications in such cases.
- Tokens are either locked or burned on the source chain. This information is then transferred to the middleware contract via the RouterGateway.
- Depending on the capabilities of the destination contract, tokens are either minted afresh or existing tokens are unlocked and transferred.
- Liquidity management is essential in the AssetBridge flow, distinguishing it from the Asset Forwarder flow where multiple forwarders might operate.
- Transfer instructions can be coupled with the token transfer itself, offering more context for the transaction.
- The middleware contract performs various checks, including ensuring the source contract's legitimacy and validating data encoding.

Threat model

During our analysis, we inspected in particular the following general threats:

Asset Forwarder flow:

1. **System-level invariants:**
    - Verification that only one forwarder can relay a deposit at a time.
    - Assessment of forwarder behavior on relay for a timed-out deposit.
    - Ensuring if a forwarder gets refunded, it has performed the relay.
2. **Timeout and synchronization assumptions:**
    - Checking the robustness of the timeout mechanism and its dependency on synchronized clocks.
    - Analyzing the packet order and nonce system for reliability in detecting timeouts.
3. **Funds transfer validity:**
    - Confirmation that the transferred amount by a forwarder is accurate and no wrong data is used.
    - Ensuring no duplicate withdrawals occur on the source chain.
4. **Progress properties and liquidity:**
    - Review of the refund process and its conditions.
    - Verification of the router chain middleware's bookkeeping to prevent unfulfilled transactions from affecting liquidity.
    - Ensuring the forwarder's dependency on the router chain's communication for transaction completion.
5. **Design challenges:**
    - Ensuring every deposit and relay is properly recorded on the router chain.
    - Reviewing the router middleware for accuracy in liquidity management and the handling of "in-flight" funds.
    - Considering the potential halt of the system due to a stop in off-chain orchestrator operations.

Asset Bridge flow:

1. **Invariants:**
    - Confirmation that for each `transferToken` call, there is at most one `iReceive`.
    - Examination of the flow through the router chain, nonce management, and the checks on `iReceive`.
    - Ensuring that the tokens received match the sent amount, accounting for deducted fees.
2. **Transfer continuity:**
    - Verification that for each `transferToken`, there is either a successful `iReceive` or a refund on the source chain.
    - Assessing the procedures and error handling from the sender to the router chain, and from the router to the receiver chain.
3. **Progress and order reliance:**
    - Ensuring that a `transferToken` call leads to an eventual `iReceive` call under normal conditions.
    - Confirming the protocol for refunds and problem scenarios
    - Checking the reliance on ordered transmission and its implications on the protocol continuity.

In the audit process, we inspected the listed threats, resulting in the findings presented in the Findings section. Several invariant analyses are detailed in the System invariants section.

System invariants

For each deposit, at most one forwarder can do the relay.

Description:

1. **Deposit creation**: When a deposit is created on the source chain contract, the function `iDeposit` is called. This results in the emission of an event `FundsDeposited` which contains several parameters including the `partnerId`, `amount`, `destChainIdBytes`, `destAmount`, `depositNonce`, `srcToken`, `recipient`, and the sender of the deposit (`msg.sender`).
2. **Deposit nonce**: The `depositNonce` is a state variable in the contract that tracks the number of deposits (or related actions). It gets incremented every time one of the five mentioned functions is called: `iDeposit`, `iDepositInfoUpdate`, `iDepositMessage`, `iRelay`, and `iRelayMessage`.
3. **Forwarder action**: After the `FundsDeposited` event has been emitted, a Forwarder entity listens to this event, transforms it, applies its custom business logic (which might include checks for profitability), and then calls the `iRelay` function. The function `ValidateAndTransformVoyagerFundsDepositedEvents` is presumably where the Forwarder performs these transformations and checks.
4. **Duplicate relay prevention**: Within the contract, there's a mechanism to ensure that a specific relay (or deposit) cannot be executed more than once. This is achieved through:
   - **Message hash creation**: A unique identifier for each relay is generated using the `keccak256` hashing function. This hash (referred to as `messageHash`) is created by encoding several pieces of relay data like the `amount`, `srcChainId`, `depositId`, `destToken`, `recipient`, `depositor`, and the contract's address (`address(this)`).
   - **Execution check**: The contract checks whether this `messageHash` has already been executed by consulting the `executeRecord` mapping. If the relay has already been executed, the contract reverts with the error `MessageAlreadyExecuted`.
   - **Setting execution status**: If the relay hasn't been executed, the contract marks it as executed by setting `executeRecord[messageHash] = true`.

Assessment:

The described mechanism ensures that each deposit can only be relayed once. The combination of incrementing the `depositNonce` for every deposit-related action and the unique message hashing mechanism followed by the check against the `executeRecord` mapping helps maintain the stated invariant.

Forwarder is able to withdraw the corresponding amount after a successful relay.

Process flow:

1. **Initiation**: The process is initiated with the `iDeposit` function, where funds are deposited. This emits a `FundsDeposited` event which is monitored by both orchestrators and forwarders.

2. **Amounts**: The `relayData.amount` in `iRelay` should equal `IAssetDepositedData.DestAmount` sent in the `iDeposit` message. A unique `messageHash` is created during `iRelay`, which includes this amount.
3. **Orchestrator role**: Orchestrators transform the `FundsPaid` event into a Router message. If a matching request is found based on `messageHash`, then:
    - `request_info.is_paid` is set to the forwarder's address.
    - The `update_claimable_amount` function updates the amount claimable by the forwarder, which is the original amount minus fees.
4. **Risks for forwarders**: Forwarders face risks if they send incorrect amounts to the destination chain. Such payments are stored in `PENDING_FUND_PAID`, and without a matching event from the source chain, the forwarder risks losing funds.
5. **Conditions for** `update_claimable_amount`:
    - When a payment from the destination is matched with a deposit from the source.
    - When a deposit from the source is matched with a payment from the destination.
    - During forwarder withdrawal, where the claimable amount is reduced.
    - Following an acknowledgment error after a payment to the forwarder on the source chain.
6. **Withdrawal mechanism for forwarders**:
    - Forwarders can only withdraw after successful matching and accounting on the middleware.
    - In case of errors during the outbound request from the Router, a failed acknowledgment is processed. This updates the `claimable_amount`, allowing the forwarder to retry the withdrawal.

Conclusion: The system is designed to ensure that forwarders can withdraw their rightful amounts post-relay, factoring in potential errors and adjustments for fees. Proper matching and accounting are crucial for ensuring the forwarder's ability to withdraw.

For every chain, the total number of tokens `x` locked inside the contract must always exceed the liquidity recorded on the router middleware

This invariant is essential due to two primary reasons:

1. **Refund assurance:** Should an iDeposit timeout occur, it's crucial for the contract to have sufficient tokens available to process the refund.
2. **In-flight tokens:** Between the time a refund of amount `x` is initiated on the router chain and when the corresponding transaction gets executed on the destination chain, `x` tokens are effectively "in-flight." During this period, the contract's token amount should be at least `x` higher than the liquidity tracked on the middleware.

The process to ensure this invariant holds true is described as follows:

- **iDeposit:** When a user performs an iDeposit of amount `x` on chain A, this shouldn't influence the liquidity recorded on the middleware until a successful relay is achieved. Post successful relay by a forwarder, the middleware should log an increment in liquidity on chain A by `x`.
- **Refunds:** If a forwarder gets a refund on chain B for `y` tokens, the middleware should accordingly decrease the liquidity on chain B by `y`. Moreover, a forwarder's refund request of `z` tokens on chain C should only proceed if the liquidity in chain C is either equal to or exceeds `z`.

Considering the code situation:

The function `update_chain_liquidity_and_claimable_amount` is invoked exclusively in two cases:

1. During `handle_fund_deposit` if there exists a matching pending `FundPaidInfo` obtained from `handle_funds_paid` and stored in `PENDING_FUND_PAID`.
2. During `handle_funds_paid`, if there exists a matching `RequestInfo` obtained from `handle_fund_deposit` and saved in `MESSAGE_HASH_DATA`.

Both scenarios represent a "match" situation. This ensures that on the Router middleware, the deposit is being accurately transferred to the intended recipient at the destination. Subsequent steps involve updating the liquidity and claimable amounts for the involved parties.

In the code provided:

- The source chain's liquidity is correctly updated.
- The forwarder's claimable amount is set, taking into account any system and partner fees.
  - Note: Instead of incrementally updating, this operation overwrites the value, which has its related issue.
- Issue with extra fees found in `request_info` when updating the chain liquidity have been highlighted as well.
- Liquidity on the Router middleware appears to be properly managed overall. Both the chain's liquidity and the forwarder's claimable amount decrease whenever a forwarder opts to withdraw a portion or the entirety of their claimable amount.
- It's noteworthy that the withdrawal amount undergoes a sanitization process prior to updating the chain liquidity. The `sanitize_amount` function ensures amounts are precisely and consistently represented, factoring in decimal variations between input and output tokens.
- The final point of potential change for `update_chain_liquidity` is in `handle_sudo_ack`: `ClaimFunds` variant, if post-receipt of an acknowledgment contains an error. In such cases, both the chain's liquidity and the forwarder's claimable amount need upward adjustments, which is what's observed in the given code.

For every `transferToken` action initiated on the AssetBridge, there must be either a corresponding successful execution of `iReceive` on the destination chain or a refund executed on the source chain..

When a user initiates a `transferToken` call on the AssetBridge, the contract locks or burns the specified tokens and sends a packet to the gateway. The orchestrators are expected to deliver this packet to Router chain. On the Router chain, the bridge contract processes the packet. Now, let’s consider two scenarios:

1. **Successful transfer scenario**:
   - If there are no errors, the bridge contract creates outbound cross-chain call for destination chain. At the end of the flow, `iReceive` is triggered to mint or unlock tokens for the recipient or perform other instructed operations.
   - The successful execution of `iReceive` on the destination chain completes the token transfer process.
2. **Refund scenario**:
   - If the packet encounters an error during processing (e.g., a validation failure or an execution error), the bridge contract on the Router creates an outbound cross-chain call back to the source chain.
   - The call instructs the AssetBridge on the source chain to refund the locked or equivalent tokens to the user.
   - The AssetBridge then performs the refund, ensuring that the user's funds are not lost.

In both scenarios, the invariant ensures that the user's funds are either successfully transferred to the intended recipient on the destination chain or returned to the user on the source chain.
However, there exists a possibility that the execution of the bridge contract may fail without initiating a crosschain call. This potential issue is detailed in a specific finding.

There is no misdirection of funds in create_refund_request and create_withdraw_request functions

Background: The functions `create_withdraw_request` and `create_refund_request` play crucial roles in managing the fund withdrawal and refund processes, respectively. A primary concern is the potential for misdirection of funds aimed for the depositor or forwarder.

1. create_withdraw_request vulnerabilities and safeguards:

- **Potential vulnerability**: The function `create_withdraw_request` can be invoked by any actor without any form of authentication. This poses the risk of a malicious entity falsely representing themselves as a forwarder.
- **Address mapping check**: To mitigate the above vulnerability, the function verifies the forwarder's address against an address mapping stored in the `FORWARDER_ADDRESS_MP` storage variable.
    - **Weakness**: The storage variable `FORWARDER_ADDRESS_MP` is populated/updated via the `update_forwarder_address` function, which can be invoked by any actor, potentially allowing for fraudulent address mappings.
- **Claimable amount defense**: The last line of defense is the `fetch_claimable_amount` function, ensuring that the purported forwarder has a legitimate claimable amount stored in the `CLAIMABLE_AMOUNT` storage. This is a critical safeguard since populating `CLAIMABLE_AMOUNT` is a strictly controlled process, and it prevents just any actor from wrongfully withdrawing funds.

2. create_refund_request safeguards:

- **Extraction from message_hash**: The actual depositor's address is extracted from the `message_hash` input. This ensures that only the genuine depositor can be the recipient of the refund. The `RequestInfo` for the associated `message_hash` is loaded from the `MESSAGE_HASH_DATA` storage.
- **Timed-out deposits control**: Only deposits that have timed out can be processed for refunds, which provides a safety measure against malicious refund requests.
- **Double payment prevention**: To avoid double payments, there's a check against the `request_info.is_paid` field. This ensures that once a request has been paid, it cannot be processed again for another payout.

Conclusion: Given the checks and balances in place, especially the claimable amount verification in `create_withdraw_request` and the message hash extraction in `create_refund_request`, it appears that there are robust measures to ensure that only the legitimate depositor or forwarder can receive their intended funds. Consequently, concerns about fund misdirection seem to be effectively addressed, and there are no apparent security issues with the current implementation regarding fund refunds for both users and forwarders.

Findings

Title	Type	Severity	Status
Possible leakage of Asset Forwarder contract funds on source chain	IMPLEMENTATION	4 CRITICAL	RESOLVED
Widespread absence of error handling	IMPLEMENTATION PRACTICE	4 CRITICAL	RESOLVED
Potential chain halt due to gas consumption panic	IMPLEMENTATION	4 CRITICAL	RESOLVED
Unhandled panic in requestProcessor.Start() & non-optimal sleep mechanism	IMPLEMENTATION PRACTICE	3 HIGH	RESOLVED
Potential Data Duplication in MESSAGE_HASH_DATA and BLOCKED_FUNDS	IMPLEMENTATION	3 HIGH	RESOLVED
Unacknowledged execution failures in bridge contract	IMPLEMENTATION	3 HIGH	RESOLVED
Race condition between user refund and forwarder transfer process	IMPLEMENTATION	3 HIGH	RESOLVED
Forwarder inability to withdraw cumulative funds due to overwritten claims	IMPLEMENTATION	3 HIGH	RESOLVED
Asset bridge middleware contract takes no action based on the acknlowledgments result	IMPLEMENTATION	3 HIGH	RESOLVED
Incorrect handling of extra fees impacting chain liquidity	IMPLEMENTATION	2 MEDIUM	RESOLVED

Title	Type	Severity	Status
Ignoring errors – a recipe for unpredictable failures and data inconsistencies	IMPLEMENTATION	2 MEDIUM	RESOLVED
Incorrect parameter passed for transaction expiry validation	IMPLEMENTATION	2 MEDIUM	RESOLVED
Possible failure of middleware contract to record processed invalid deposits	IMPLEMENTATION	2 MEDIUM	RESOLVED
Potential overflow and division by zero in isTransactionProfitable function	IMPLEMENTATION	1 LOW	RESOLVED
Middleware contract is missing full validation of evm type recipient addresses	IMPLEMENTATION	1 LOW	RESOLVED
Oversight in fee calculation considering chain decimal differences	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Ambiguous design in iDepositInfoUpdate and associated middleware handle_deposit_info_update function	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Miscellaneous: areas of improvement	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED

Possible leakage of Asset Forwarder contract funds on source chain

Title	Possible leakage of Asset Forwarder contract funds on source chain
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- contracts/asset-forwarder/src/execution.rs

Description

When processing FundDeposit and DepositInfoUpdate messages within the middleware contract, there is a chance of encountering errors during message handling. In such situations, the middleware contract will take action by adding the specified amount to the `BLOCKED_FUNDS` state for the given source chain, depositor, and token.

This action effectively categorizes the deposit as invalid, even though the depositor's funds are still held in escrow on the source chain. This measure is put in place to give depositor chance to withdraw the funds if some error in the processing happens. As a result, the depositor is only allowed to withdraw an amount equal to what is stored within the `BLOCKED_FUNDS` state.

Problem Scenarios

To facilitate the withdrawal of blocked funds, depositors must invoke the `withdraw_block_funds` method on the middleware smart contract within the Router chain. However, before initiating a cross-chain call to the source chain for withdrawing the funds to the depositor, the `withdraw_block_funds` method should also update the `BLOCKED_FUNDS` state by subtracting the amount specified in the cross-chain call.

As it stands, this **mechanism does not automatically update** `BLOCKED_FUNDS` **, which opens the door for depositors to potentially make multiple "withdraw_block_funds" requests and withdraw more funds than they are entitled to**.

On the other hand, the middleware contract is designed to await acknowledgment, potentially allowing for the updating of the `BLOCKED_FUNDS` state. In cases where the withdrawal process completes without any issues, no further action is required. However, if the acknowledgment returns an error, the following code will be executed:

```
MessageData::RefundBlockedFund(blocked_info) => {
            let info_str = format!("Setting refunded to false for {:?}",
request_identifier);
            deps.api.debug(&info_str);
            let prev_blocked = BLOCKED_FUNDS
                .load(
                    deps.storage,
                    (
                        &blocked_info.chain_id,
                        &blocked_info.depositor,
                        &blocked_info.token,
                    ),
                )
                .unwrap_or(Uint128::zero());
            BLOCKED_FUNDS.save(
                deps.storage,
                (
                    &blocked_info.chain_id,
                    &blocked_info.depositor,
                    &blocked_info.token,
                ),
                &(prev_blocked + blocked_info.amount),
            )?;
        }
```

The expression `&(prev_blocked + blocked_info.amount)` implies that the amount, which was not withdrawn due to an error, is being added to the previous `BLOCKED_FUNDS` value. This is problematic because the amount was never subtracted before sending the cross-chain request, which means that the amount will essentially be inserted into `BLOCKED_FUNDS` again in the event of an error.

This scenario could enable depositors to repeatedly attempt withdrawals, causing `BLOCKED_FUNDS` to accumulate and potentially resulting in them gaining access to more funds than originally intended. As long as there are available funds within the source chain contract for the specified token denomination, depositors could potentially acquire additional funds through this process.

Recommendation

In the `withdraw_block_funds` function, before making the cross-chain call, it's crucial to subtract the intended amount from the `BLOCKED_FUNDS`. This step serves two important purposes:

1. It prevents depositors from submitting multiple valid `withdraw_block_funds` requests for the same amount, ensuring fair and accurate withdrawals.
2. In the event of an error in the acknowledgment process, it enables the adjustment of the `BLOCKED_FUNDS` value without causing duplication issues.

Status: Resolved

Data is now removed from `blocked_funds` before initiating the withdrawal request from blocked funds. In the event of an error in the acknowledgment, it is reinserted.

Widespread absence of error handling

Title	Widespread absence of error handling
Project	Router: Q4 2023
Type	IMPLEMENTATION PRACTICE
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- listener/evm/eventprocessor/eventprocessor.go
- chains/evm/relayer/transformer.go

Description

The codebase demonstrates a recurring pattern of not handling potential errors, especially during database interactions. Such oversight can be detrimental to system stability and data integrity. This behavior is pervasive across multiple functions, including but not limited to `ProcessInboundEvents`, `TransformVoyagerDepositInfoUpdateEvent` and `FetchAndAddToDBVoyagerEvents`.

Problem Scenarios

1. The `chainListener` component, crucial for the forwarder's operation, is started in a goroutine using the `Start` method. Within this method, there's a call to `chainListener.eventProcessor.ProcessInboundEvents(...)`. If this call encounters an error (e.g. if `FetchAndAddToDBVoyagerEvents` returns an error), it does not appear to handle the error and will continue to the end of the `Start` function. As a result, the goroutine could potentially finish its execution prematurely, causing the forwarder to miss events or not function correctly without any clear indication of why it failed. The main thread (`run` function) will continue to execute independently of the finished goroutine. The end of the `chainListener.Start(ctx)` function's execution has no direct impact on the main thread or any other goroutines that may be running inside `run`.
2. The `ProcessInboundEvents` method of the `EvmEventProcessor` type contains a crucial section of code that interacts with the database to update the last processed block and event nonce. The method

`UpdateLastProcessedBlock` from `dbHandler` is used for this purpose. However, if an error occurs during the `UpdateLastProcessedBlock` call and goes unnoticed, it may lead to the database being in a corrupted state. Given that this method's purpose is to keep track of the blocks and events that have already been processed, any discrepancy or corruption in this data can be detrimental to the overall system.

3. In the `TransformVoyagerDepositInfoUpdateEvent` method, the call to `voyagerEventProcessor.dbHandler.UpdateTxq(*tx)` is made without handling potential errors. Should there be an issue with the update, it would go unnoticed, potentially leading to data inconsistencies.
4. The `FetchAndAddToDBVoyagerEvents` method contains another instance where the `voyagerEventProcessor.dbHandler.AddToTxq(*tx)` call is made without subsequent error handling. An unhandled failure here might cause certain transactions to be missed or not be processed properly.

Recommendation

1. **Never ignore errors:** Ignoring errors, especially in critical paths, is a cardinal sin in software development. Always handle errors appropriately by logging them and taking the necessary corrective action. If an error is genuinely insignificant and can be safely ignored, document the reason for doing so in comments.
2. **Recovery mechanism:** For the `chainListener`, consider introducing a recovery mechanism that can restart it in case of failure, ensuring that the forwarder remains operational and events aren't missed.
3. **Refactor and review:** Given the recurring nature of this issue, we strongly recommend reviewing the entire codebase for such instances and refactoring them. Review all instances in the codebase where the `dbHandler` is used. Ensure that every database interaction checks for potential errors and handles them adequately. Introducing standardized error handling will go a long way in making the software robust and resilient to failures.
4. **Unit and Integration tests:** Ensure that there are comprehensive unit and integration tests in place that specifically target error scenarios. These tests can catch instances where errors might be ignored or improperly handled, providing a safety net against potential oversights in the code.

Potential chain halt due to gas consumption panic

Title	Potential chain halt due to gas consumption panic
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- x/voyager/keeper/fund_deposit_execution.go
- x/voyager/keeper/fund_paid_execution.go
- x/voyager/keeper/update_deposit_info_execution.go

Description

The code provided demonstrates an attempt to manage gas consumption within a smart contract execution environment by creating a limited gas meter and employing a cache context to prevent state commitment in the event of errors. However, the implementation of gas consumption tracking and panic recovery could lead to critical issues.

```
// use cache context so that state is not committed in case of errors.
    contractGasLimit := params.FundPaidGasLimit
    limitedMeter := sdk.NewGasMeter(contractGasLimit)
    em := ctx.EventManager()
    cacheCtx, commit := ctx.CacheContext()
    cacheCtx = cacheCtx.WithEventManager(em).WithGasMeter(limitedMeter)

    defer func() {
        // Refund fees
        gasConsumed := cacheCtx.GasMeter().GasConsumed()
        gasUsed := big.NewInt(0).SetUint64(gasConsumed)
        gasPrice := big.NewInt(0).SetUint64(params.VoyagerGasPrice)
        feeConsumed := new(big.Int).Mul(gasUsed, gasPrice)
        feeConsumedInRoute = sdk.NewCoin(chaintypes.RouterCoin,
sdk.NewIntFromBigInt(feeConsumed))
```

```
        k.Logger(ctx).Debug("Gas consumed by bridge contract execution", "address",
cwContractAddress.String(), "gasUsed", gasConsumed, "gasLimit", contractGasLimit,
"feefeeConsumedInRoute", feeConsumedInRoute)
        // catch out of gas panic
        if r := recover(); r != nil {
            switch rType := r.(type) {
            case sdk.ErrorOutOfGas:
                err := sdkerrors.Wrapf(sdkerrors.ErrOutOfGas, "out of gas in
location: %v", rType.Descriptor)
                k.Logger(ctx).Error("Error out of gas", err)
                execFlag = false
                execData = []byte(err.Error())
            default:
                err = sdkerrors.Wrapf(sdkerrors.ErrIO, "Unknown error with contract
execution: %v", rType)
                k.Logger(ctx).Error("Unknown Error", err)
            }
        }
        // Push gas consumed to parent context. This is needed so that the
RawContractExecutionParams execution can be stopped by parent context if cumulative
gas consumed exceeds MaxBeginBlockTotalGas
        ctx.GasMeter().ConsumeGas(gasConsumed, "consume gas for contract execution in
end blocker")
    }()
```

A deferred function is used for cleanup operations, including the refund of fees and logging of gas consumption. Crucially, the deferred function includes a call to `ctx.GasMeter().ConsumeGas(gasConsumed, "consume gas for contract execution in begin blocker")`. This line is problematic as it is located within a panic recovery block, implying that an out-of-gas situation could potentially trigger a panic which, due to the context of execution (i.e., within an endBlocker), might result in a chain halt. This is akin to an issue previously identified and remedied in another part of the codebase.

Problem Scenarios

The core issue arises if the gas limit set by `contractGasLimit` is exceeded, which would trigger a panic due to an out-of-gas error. Normally, the recovery block should handle such a scenario gracefully. However, if the call to `ctx.GasMeter().ConsumeGas()` itself causes a panic (due to insufficient gas), this would not be caught by the existing recovery logic and could halt the entire blockchain, as this operation is conducted within a critical block of the chain's operation (endBlocker). This represents a significant risk to the blockchain's stability and security.

Please be aware that the identical issue occurs across three separate functions: `ExecuteFundDepositRequest`, `ExecuteFundPaidRequest` and `ExecuteUpdatedDepositInfoRequest`.

Recommendation

To address this vulnerability, the code should be refactored to include a check before gas consumption that ensures there is sufficient gas remaining. Specifically, it should use `ctx.GasMeter().GasRemaining()` to ascertain the remaining gas. If `gasRemaining` is greater than `gasConsumed`, the contract should consume the `gasConsumed` amount. Otherwise, it should only consume what is left (`gasRemaining`). This conditional approach prevents the possibility of a panic due to an out-of-gas error within the cleanup and panic recovery process.

This solution aligns with the corrective action taken in a similar situation, as detailed in the previous audit report and the corresponding commit `f797850d64eede793116e116e2768ffd263aab67`. Implementing this strategy will ensure that gas consumption accounting does not inadvertently cause a chain halt, thus enhancing the robustness and reliability of the chain operations.

Status: Resolved

The approach involving `gasRemaining` appears to be satisfactory.

Unhandled panic in requestProcessor.Start() & non-optimal sleep mechanism

Title	Unhandled panic in requestProcessor.Start() & non-optimal sleep mechanism
Project	Router: Q4 2023
Type	IMPLEMENTATION PRACTICE
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- processor/processor.go

Description

The function `requestProcessor.Start()` is invoked inside a goroutine and contains code sections that can potentially panic, particularly when there are database interactions that fail. If a panic occurs, the go runtime will then terminate the entire program, not just the goroutine where the panic occurred (there is no `recover` in place to catch the panic). In addition, there's a `time.Sleep(1 * time.Second)` mechanism that pauses the function between processing transactions.

Problem Scenarios

1. **Unhandled panics**: If there's an issue with the database, for instance, if it becomes temporarily unavailable or if there's a network glitch, a panic could be triggered, leading to the termination of the program. Notice that the function will panic even if the `TxType` is not the expected one. This would cause the `requestProcessor.Start()` function to stop processing, possibly leading to a system-wide halt in processing transactions or other related tasks. Using `panic` without `recover` is generally considered a bad practice for expected error conditions in Go. The `panic` and `recover` mechanisms are reserved for unexpected errors from which the program cannot continue, or for "fail-fast" error handling scenarios during the program's initialization phase.
   It's usually recommended to handle errors gracefully by checking the returned error and taking the appropriate action, such as logging the error, cleaning up resources, and continuing operation or shutting down gracefully as needed.

2. **Non-optimal sleep**: The static sleep interval of 1 second (`time.Sleep(1 * time.Second)`) between processing transactions might not be the best approach. If the system is under high load, this could introduce unnecessary delays. On the other hand, if there are no transactions to process, the system is idling inefficiently.

Recommendation

1. **Panic handling**: Implement a panic recovery mechanism using `recover()` to handle unexpected panics. This ensures that the goroutine isn't terminated unexpectedly. However, this is a short-term solution. The root causes of the panics should be investigated and addressed.
2. **Retry mechanism**: Instead of using a panic-recovery loop, consider implementing a more robust retry mechanism, especially for database operations. This could involve exponential backoff strategies where the system waits increasingly longer periods between retries, which can be more effective when dealing with temporary outages or network glitches.
3. **Optimizing sleep**: Reevaluate the need for the static 1-second sleep. If the aim is to reduce load on the system, consider implementing a dynamic sleep or back-off mechanism that adapts based on the system load or the number of transactions awaiting processing.
4. **Monitoring and logging**: Ensure adequate logging is in place, especially around error scenarios. This will assist in quickly diagnosing issues that lead to panics or system slowdowns. Monitoring tools can then be used to set alerts based on these logs, ensuring timely intervention.
5. **Database health checks**: Implement periodic health checks for the database, ensuring its availability. If the database is detected as being down, the system can switch to a "degraded mode" of operation or halt certain functionalities until the database is restored, rather than continuing to try operations that are likely to fail.

Potential Data Duplication in MESSAGE_HASH_DATA and BLOCKED_FUNDS

Title	Potential Data Duplication in MESSAGE_HASH_DATA and BLOCKED_FUNDS
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- asset-forwarder-middleware

Description

Middleware contract processes the observed events received from Router chain and checks whether received request is “valid” or “invalid”. `handle_fund_deposit` method calls `fund_deposit_post_processing` and in case of an error while processing the input parameters inside `fund_deposit_post_processing`, caller function (`handle_fund_deposit`) executes `BLOCKED_FUNDS.save` and saves the deposit as invalid, with depositor being able to withdraw the funds based on the information stored in `BLOCKED_FUNDS`.

Middleware contract flow:

- The middleware contract processes events from the Router chain.
- It evaluates incoming requests to determine if they are "valid" or "invalid."

Fund deposit handling:

- The `handle_fund_deposit` method is responsible for processing fund deposit requests.
- This method invokes `fund_deposit_post_processing` for additional processing.
- In case of an error during parameter processing in `fund_deposit_post_processing`, error is propagated to the caller function (`handle_fund_deposit`).

Handling invalid deposits:

- When an error in `fund_deposit_post_processing` occurs, `handle_fund_deposit` triggers `BLOCKED_FUNDS.save` to mark the deposit as invalid.
- Depositors have the ability to withdraw their funds based on information stored in `BLOCKED_FUNDS`.

Problem Scenarios

Fund deposit post processing function part:

```
  // store message_hash vs encoded data
    MESSAGE_HASH_DATA.save(deps.storage, &message_hash_hex_str, &request_info)?;
    MESSAGE_HASH_BY_DEPOSIT_INFO.save(
        deps.storage,
        (
            &request_info.src_chain_id,
            &request_info.src_contract,
            request_info.deposit_id,
        ),
        &message_hash_hex_str,
    )?;
    let info_str: String = format!("message_hash {:?}",
hex::encode(message_hash.clone()));
    deps.api.debug(&info_str);
    fund_paid_post_processing(deps.branch(), env, &request_info,
&message_hash_hex_str)?;
    let res = Response::new().add_attribute("message_hash",
hex::encode(message_hash));
    Ok(res)
```

Within the `fund_deposit_post_processing` function, it's possible for the `MESSAGE_HASH_DATA.save` operation to succeed while subsequent processing step, `fund_paid_post_processing(deps.branch(), env, &request_info, &message_hash_hex_str)?`, may encounter error. However, it's important to note that the success of the initial save operation doesn't imply that state changes will be automatically reverted in the event of an error.

Handle funds deposit function part:

```
    let fund_deposit_result = fund_deposit_post_processing(
        deps.branch(),
        env,
        src_chain_id.clone(),
        src_contract.clone(),
        dest_chain_id_bytes,
        src_token.clone(),
        depositor.clone(),
        deposit_id,
        dest_amount,
        amount,
        recipient,
        partner_id,
        message,
    );
    //update last info received timestamp
    update_last_info_received(deps.branch(), &env, &src_chain_id)?;
    let info_str: String = format!("fund_deposit_result {:?}",
fund_deposit_result.is_ok());
    deps.api.debug(&info_str);
    if fund_deposit_result.is_err() {
        let blocked_amount = BLOCKED_FUNDS
            .load(deps.storage, (&src_chain_id, &depositor, &src_token))
            .unwrap_or(Uint128::zero());
        BLOCKED_FUNDS.save(
            deps.storage,
            (&src_chain_id, &depositor, &src_token),
            &(blocked_amount + amount),
        )?;
```

Caller function could then execute `BLOCKED_FUNDS.save` based on `fund_deposit_result.is_err()` which returned true.

That indicates there is a possibility of duplicate refund from a depositor: from both normal refund and withdraw from blocked funds!

Recommendation

This could be prevented if outer function only called `MESSAGE_HASH_DATA.save` if all checks were performed beforehand.

Status: Resolved

The issue is resolved, but there is uncertainty regarding the rationale behind the initial insertion into `MESSAGE_HASH_DATA` and `MESSAGE_HASH_BY_DEPOSIT_INFO`, followed by a potential removal in the event of an error in `fund_paid_post_processing`. It may be more coherent to consider inserting only when `fund_paid_post_processing` is completed without errors.

Unacknowledged execution failures in bridge contract

Title	Unacknowledged execution failures in bridge contract
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- middleware/contracts/asset-bridge/src/handle_inbound.rs
- x/crosschain/abci.go

Description

In the current implementation of the bridge contract on the Router chain, a critical gap exists in the error handling logic. While `revert_inbound_to_src_chain` is effectively utilized to manage errors during `handle_reserve_tokens` or `fetch_and_validate_dest_contract`, other failure modes are not captured, leaving no recourse for error acknowledgment or refund initiation.

Particularly, errors stemming from store interaction issues, `is_chain_unpaused_modifier` checks, or incorrect `src_chain_type` values do not trigger any form of acknowledgment back to the source chain. Given the `AssetBridge_REQUEST_METADATA` configuration, which defaults to `NO_ACK`, the system is predisposed to silence on failures, potentially leading to stranded assets without automated recovery or notification to the original chain.

Problem Scenarios

In the event of a contract execution failure on the Router chain that falls outside the predefined error handling paths, the current logic will neither revert the source chain state nor dispatch an acknowledgment of failure. This situation arises because the metadata encoded within `AssetBridge_REQUEST_METADATA` results in an `ackType` of `0` (NO_ACK), implying no acknowledgment is needed, regardless of success or failure.

Consequently, if an error occurs within `HandleIReceive` — for instance, during store access or if `is_chain_unpaused_modifier` returns an error — the contract execution halts. Without an acknowledgment, the source chain remains uninformed, and the locked or burned funds designated for crosschain transfer cannot be refunded or released, effectively being locked indefinitely.

Technical breakdown:

- The `AssetBridge_REQUEST_METADATA` constant is set to a value where the extracted `chunk6` (from the metadata string) is `00`, equating to an `ackType` of `NO_ACK`.
- The `isAckRequired` function on the Router chain does not trigger an acknowledgment in case of execution failure (`execStatus == false`) due to the `NO_ACK` configuration.
- The `HandleIReceive` function on the Router chain is devoid of a comprehensive error broadcasting mechanism that covers all potential errors outside of `handle_reserve_tokens` and `fetch_and_validate_dest_contract`.

Recommendation

A robust error recovery and acknowledgment system should be implemented to address this vulnerability:

- **Extended error handling**: Broaden the error handling logic in `HandleIReceive` to cover all possible execution paths, including interactions with store and other modifiers.
- **Acknowledgment configuration**: Revise the `AssetBridge_REQUEST_METADATA` to enforce an acknowledgment on error (`ACK_ON_ERROR`) or in all cases (`ACK`), ensuring that all states of execution are communicated back to the source chain.
- **Enhanced acknowledgment logic**: Modify the `isAckRequired` logic to account for all failure conditions, ensuring that any execution status resulting in failure triggers an appropriate acknowledgment back to the source chain.
- **Dynamic metadata adjustment**: Implement mechanisms that allow for the dynamic updating of acknowledgment policies within the contract metadata to cater to evolving needs without the need for contract migration.

Race condition between user refund and forwarder transfer process

Title	Race condition between user refund and forwarder transfer process
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

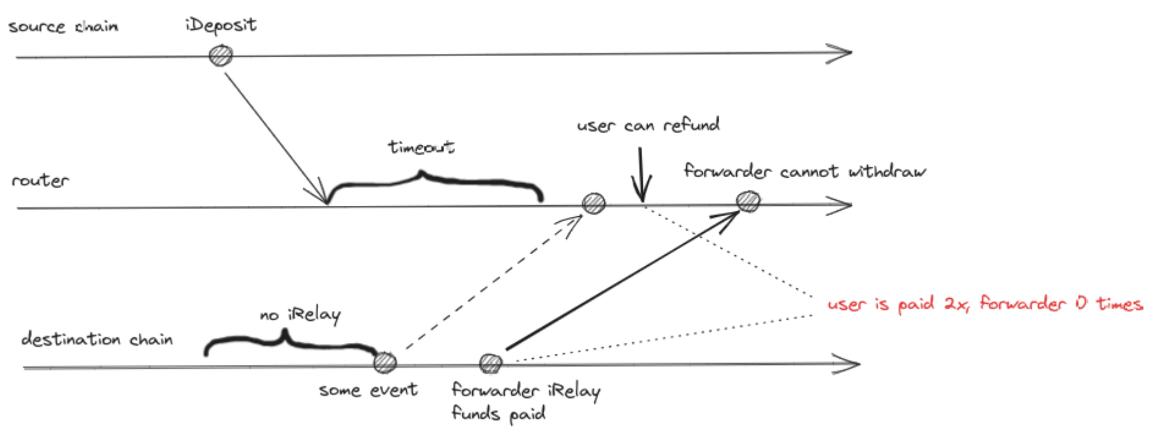
Involved artifacts

- voyager-forwarder
- asset-forwarder-middleware

Description

In the cross-chain transfer mechanism, a potential race condition exists between the user's ability to refund and the forwarder's transfer process. When a deposit (`iDeposit`) event occurs on the source chain, it is propagated to the Router via Orchestrators, ultimately reaching the middleware smart contract. From this point, a timeout mechanism is activated, monitoring for any delay or lack of confirmation of the funds' transfer.

There's a vulnerability window where forwarders might momentarily ignore such a request, potentially due to prioritizing higher fee requests. If the forwarder hasn't executed `iRelay` within the defined timeout duration, or the relayed request hasn't been validated on the Router, the user becomes eligible for a refund. Consequently, the user might initiate `CreateRefundRequest` to obtain their refund. Meanwhile, there exists a latent risk that the destination chain might eventually recognize and confirm the transfer, indicating the forwarder has moved the funds.



Problem Scenarios

Consequences:

1. **Double payment to user**: The user could be compensated twice—once on the destination chain by the forwarder and again through their refund on the source chain.
2. **Forwarder's funds locked**: Due to the premature user refund, the forwarder could be denied the ability to withdraw its funds, leading to a potential loss or lockup of assets.

Implication:

This race condition can compromise the integrity and fairness of the transfer mechanism, potentially resulting in financial losses for the forwarders and unintended gains for the users.

Additional insights from Router dev:

1. **Timestamp discrepancy**: The intended approach was to use timestamps from the source/destination chain when updating `LAST_INFO_RECEIVED_TIMESTAMP`. However, the timestamp currently used is from the Router: `LAST_INFO_RECEIVED_TIMESTAMP.save(deps.storage, chain_id, &env.block.time.seconds())?;`.
2. **Forwarder behavior**: If the forwarder doesn't intend to complete the transfer within the initial hour, it is unlikely to process it at all, ensuring the expiry timestamp remains unthreatened. Forwarders also assess the expiration time of requests, ignoring those past this timeframe.
3. **Orchestrator latency**: There's a potential for Orchestrators to delay in relaying requests, which could be a primary cause for request latency from the destination side. This factor significantly influences the timeout risk.
4. **Preventative measures**: Though the likelihood of this race condition occurring is low, it is still vital to address. Suggestions include using the timestamp from the source chain or the Router chain with some added buffer time.

Recommendation

The timeout check should be implemented on the destination side's smart contract to avert transfers to the Router. However, it is crucial that the Router still remains updated with this information. Proper measures, including the usage of accurate timestamps and consistent checks, should be instituted to ensure equitable treatment for all involved parties.

Status: Resolved

`LAST_INFO_RECEIVED_TIMESTAMP` no longer stores the `env.block.time.seconds()` timestamp; instead, it is now updated with `src_timestamp`.

Forwarder inability to withdraw cumulative funds due to overwritten claims

Title	Forwarder inability to withdraw cumulative funds due to overwritten claims
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- asset-forwarder-middleware

Description

The middleware contract seems to have an oversight where a forwarder's accumulated claimable amounts may be potentially overwritten, instead of being cumulatively added. This arises when the `update_claimable_amount` function is invoked.

The `CLAIMABLE_AMOUNT` storage item, designed to store the amount claimable by the forwarder, appears to overwrite the claim amount for each request rather than summing up. This means the forwarder can only claim the amount from their last request, leading to potential financial losses for the forwarder, especially if they are batching multiple requests for a single withdrawal.

Problem Scenarios

Root Cause: The core of the issue seems to be in the way `CLAIMABLE_AMOUNT` is being updated inside the `update_claimable_amount` function. Instead of incrementally adding to the existing claimable amount (using `+=`), it overwrites with the new amount.

Potential Impact:

Forwarders might not be able to claim their entire rightful amount, leading to potential losses.

Additional comment:

There's a potential catch with the `CLAIMABLE_AMOUNT`. Its uniqueness is determined using a composite key comprised of the chain_id, token, and forwarder_address. Now, while this might seem sufficient for individual requests, there's a caveat: if a forwarder sends multiple requests that share the same key before calling `create_withdraw_request`, only the latest request's amount will be recorded. Previous requests with the same key will be inadvertently overwritten.

Furthermore, the `create_withdraw_request` function has implemented an additional layer of check. If a forwarder attempts to process withdrawals for multiple requests that target the same chain_id, the function won't execute. Specifically, it collects all the chain_ids from the `withdraw_info`, checks for any repetitions, and if found, it promptly returns an error stating that chain_ids must be unique.

This combination of logic means that forwarders need to be cautious. If they send multiple requests with identical keys and then attempt to collectively process them, they'll face hurdles, potentially leaving funds unclaimed.

Recommendation

Exploring the possibility of restricting cumulative fund withdrawals for forwarders, enabling them to withdraw all available funds in a single transaction. This not only resolves the issue at hand but also streamlines the process, as multiple smaller withdrawals offer minimal benefits compared to a single comprehensive withdrawal.

Status: Resolved

The update logic (`+=` and `-=`) has been relocated within `update_chain_liquidity()`. Nonetheless, it seems that the amount is sanitized twice, both within and outside the `update_claimable_amount` function, as observed in this section: link to the code.

Asset bridge middleware contract takes no action based on the acknlowledgments result

Title	Asset bridge middleware contract takes no action based on the acknlowledgments result
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- middleware/contracts/asset-bridge/src/handle_acknowledgement.rs

Description

In the asset bridge flow, the Router chain's asset bridge middleware carries out cross-chain calls with two potential outcomes: one, completing the token transfer on the destination chain, and two, reverting the transaction on the source chain in case of processing errors. In both scenarios, the middleware contract awaits acknowledgments to provide a response, which can be either success or failure.

It is of utmost importance for the middleware to be informed about the results of both transactions. This knowledge is vital, as the middleware must be prepared to take the necessary actions in the event of any request failure.

Problem Scenarios

```
use crate::state::{ACK_STATUS, FEE_REFUND_ADDRESS};
use cosmwasm_std::{Binary, Coin, DepsMut, Env, Response, StdResult};
use router_wasm_bindings::{RouterMsg, RouterQuery};

pub fn handle_sudo_ack(
    deps: DepsMut<RouterQuery>,
    _env: Env,
    request_identifier: u64,
    exec_flag: bool,
    exec_data: Binary,
    refund_amount: Coin,
) -> StdResult<Response<RouterMsg>> {
    let ack_status_key: String = request_identifier.to_string();

    let info_str: String = format!(
        "handle_out_bound_ack_request-- request_identifier: {}, exec_flag: {},
exec_data: {}, exec_data {:?}, refund_amount {:?}",
        request_identifier, exec_flag, exec_data.clone(), exec_data, refund_amount
    );
    ACK_STATUS.save(deps.storage, &ack_status_key, &info_str)?;
    deps.api.debug(&info_str);

    let response =
        Response::new().add_attribute("outbound_batch_nonce",
request_identifier.to_string());

    match FEE_REFUND_ADDRESS.load(deps.storage, request_identifier) {
        Ok(_fee_payer) => {
            // TODO: handle excess fee
        }
        Err(_) => {}
    }

    Ok(response)
}
```

The `handle_sudo_ack` function presented appears to be incomplete. It records the acknowledgement status within the code but lacks any executed actions based on the transaction's result. Furthermore, there is no other method that utilizes the `ACK_STATUS` storage to trigger actions according to the acknowledgement states.

This situation leaves room for the following possible scenarios:

1. In the event of a failed revert, there is no mechanism in place for retrying the operation, resulting in the depositor losing any opportunity to refund the deposited funds on the source chain.
2. If the transfer from the middleware to the destination chain encounters an issue, the contract does not initiate a revert transaction or attempt a retry.

Addressing these concerns is essential to ensure the proper functioning and reliability of the system.

Recommendation

We recommend implementing mechanisms that can respond to acknowledgments' outcomes effectively, thereby preventing funds from becoming trapped in case of issues. This proactive approach will enhance the system's reliability and resilience.

Status: Resolved

The `execute_failed_request` function has been added.

Incorrect handling of extra fees impacting chain liquidity

Title	Incorrect handling of extra fees impacting chain liquidity
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- contracts/asset-forwarder/src/execution.rs

Description

The `update_chain_liquidty_and_claimable_amount` function is designed to update both the chain's liquidity and claimable amounts when a match is identified on the Router middleware.

Within this function, there's a distinct section dedicated to handling `extra_fee` which is encapsulated within the `request_info` structure. If `extra_fee` is present, the chain liquidity is incremented again by `request_info.src_amount`, even though the logical intent appears to be to only consider the actual fee amount present in the `ExtraFeeInfo`.

Problem Scenarios

When the `extra_fee` exists, the system erroneously increases the chain's liquidity using the entire source amount (`request_info.src_amount`) as opposed to just the specified extra fee. This leads to a potential inflation of the liquidity amount beyond the actual funds present. Consequently, this can paint an inaccurate picture of the available liquidity, which might lead to unforeseen consequences.

One direct ramification could be incorrect liquidity checks; the following code segment might pass without error:

```
let liquidity: Uint128 = fetch_token_liquidity(
            deps.as_ref(),
            &dest_chain_id,
            &voyager_gateway,
            &dest_token.clone(),
        );
        if liquidity < dest_sanitize_amount {
            return Err(StdError::GenericErr {
                msg: format!(
                    "liquidty not present on chain: {} for token: {}",
                    dest_chain_id.clone(),
                    dest_token.clone()
                ),
            });
        }
```

Despite there being insufficient actual liquidity, due to the exaggerated liquidity values caused by the double addition of `request_info.src_amount`.

Despite the severity of the issue being mitigated to 'Medium' due to other checks in place, such as ensuring the withdrawal amount does not exceed the claimable amount, the core issue still poses a significant risk. These additional checks may prevent the most severe scenarios, but they do not address the underlying cause of the liquidity inflation.

Recommendation

1. Revise the logic within the `extra_fee` handling block to ensure that the chain's liquidity is incremented only by the specific fee amount, not by the entire `request_info.src_amount`.
2. Implement unit tests specifically designed to verify the correctness of liquidity updates when `extra_fee` is present. This will help identify if such discrepancies occur in the future.
3. Review any historical transactions processed with `extra_fee` to determine if there were any instances where the liquidity was erroneously updated, and determine potential remedies for affected parties if needed.
4. Going forward, maintain rigorous documentation detailing the logic behind liquidity calculations, especially in edge cases like `extra_fee`, to prevent similar oversight in the future.

Status: Resolved

In the scenario involving `extra_fee`, the chain liquidity is now accurately updated to include: `extra_fee.sys_fee + extra_fee.partner_fee + extra_fee.forwarder_fee`.

Ignoring errors – a recipe for unpredictable failures and data inconsistencies

Title	Ignoring errors – a recipe for unpredictable failures and data inconsistencies
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- chains/evm/relayer/transformer.go
- listener/evm/eventprocessor/transformer.go
- chains/evm/relayer/executor.go

Description

Apart from the observed lack of proper error handling during database interactions, there's a concerning pattern of explicitly ignoring returned errors in the codebase. This can result in the software behaving unpredictably and could further escalate to more severe issues like data corruption or system malfunctions.

Problem Scenarios

1. In the `TransformVoyagerFundDepositEvent` function, the `ValidateTxExpiry` method is called to validate the expiration of the transaction (deposit). However, the result or error of the validation is not being checked or handled. This omission potentially allows transactions that have already expired to be processed further, resulting in unintended and possibly problematic behavior.
2. In the `TransformVoyagerFundDepositEvent` method of the `EvmEventProcessor` struct, the error returned by `voyagerEventProcessor.GetBlockTime(fundsDepositedEvent.Raw.BlockNumber)` is intentionally ignored using the `_` syntax. Consequently, the `timeStamp` variable might contain an invalid or zero value, which can then be saved to the database, leading to data inconsistencies. The same problem repeats in `TransformVoyagerFundDepositWithMessageEvent` and `TransformVoyagerDepositInfoUpdateEvent`.

3. Similarly, in another instance, the line `data, _ := methods.IRelay.GetAbiPackBytes(AssetForwarder.AssetForwarderABI, relayData, m.ForwarderAddress)` from the `IRelay` method discards a potentially vital error message. If there's an issue with `GetAbiPackBytes`, the resultant `data` might be incomplete, corrupt, or altogether invalid. Such corrupted data, when propagated through the system, can lead to unpredictable behavior or even crashes.
4. In the function `isTransactionProfitable`, the conversion of the `feesInUsd` string to a `big.Int` is performed without handling potential failures. Specifically, the code `num2, _ := new(big.Int).SetString(feesInUsd, 10)` disregards the result indicating the success of the conversion. If the conversion fails, the function could mistakenly assume the fees are zero, leading to erroneous assessments of profitability.

Recommendation

The same as the Widespread absence of error handling .

Status: Resolved

All four problematic scenarios previously identified have now been resolved.

Incorrect parameter passed for transaction expiry validation

Title	Incorrect parameter passed for transaction expiry validation
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- chains/evm/relayer/transformer.go

Description

The `TransformVoyagerFundDepositEvent` function calls the method `ValidateTxExpiry` with the `BlockHeight` parameter, specifically `tx.Raw.BlockHeight`. However, upon reviewing the implementation of `ValidateTxExpiry`, it is evident that the **function expects a** `txTimestamp` (indicative of an actual timestamp) as its parameter **and not a block height**.

Using block height in place of a timestamp for expiration validation can lead to inaccurate results, making transactions appear valid or expired incorrectly.

Problem Scenarios

Given the current implementation:

1. The `ValidateTxExpiry` method compares the sum of the received timestamp and a fixed expiry time (in seconds) against the current time's seconds. Using a block height instead of a timestamp for this comparison will yield incorrect results.
2. If a block height is significantly different from an actual timestamp (which is very likely), transactions can be prematurely marked as expired or remain unexpired well past their actual expiry time.
3. This can potentially lead to acceptance of transactions that should have been rejected or rejection of valid transactions, posing both usability and security risks.

Recommendation

1. **Correct parameter usage**: Modify the call to `ValidateTxExpiry` in the `TransformVoyagerFundDepositEvent` function to pass the correct `Timestamp` parameter: `tx.Raw.Timestamp`.
2. **Function signature clarity**: Rename the parameter of the `ValidateTxExpiry` method from `txTimestamp` to something more descriptive, like `txExpiryTimestamp`, to avoid confusion in future development.
3. **Robust error handling**: Ensure that error handling is in place for the `GetBlockTime` function and that it doesn't silently ignore errors.
4. **Testing**: Implement unit tests and integration tests to confirm that the expiration mechanism works as expected and that transactions are correctly marked as expired or not based on their timestamps.

Status: Resolved

Resolved, with `tx.Raw.Timestamp` being passed.

Possible failure of middleware contract to record processed invalid deposits

Title	Possible failure of middleware contract to record processed invalid deposits
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- asset-forwarder-middleware

Description

The middleware contract plays a crucial role in the validation of deposits received from the Router chain. It empowers depositors to request refunds if the deposited event is considered invalid, with the amount info stored in BLOCKED_FUNDS. The process is facilitated by the `handle_fund_deposit` function, which, in turn, calls `fund_deposit_post_processing` to thoroughly scrutinize the deposit data. In the event of any data-related errors, the deposit is marked as invalid and securely stored in BLOCKED_FUNDS.

What makes this process particularly critical is that, regardless of the deposit's status (valid or invalid), the funds are already deposited on the source chain. The middleware contract verifies deposits only after they have occurred. This underscores the significance of the contract's ability to accurately determine the validity of deposits, as it directly impacts whether depositors will be permitted to withdraw from BLOCKED_FUNDS.

Problem Scenarios

Inside the `fund_deposit_post_processing` function, there are instances where `unwrap()` is used for error handling. If an error occurs during `unwrap()`, it will lead to a middleware contract panic, causing it to fail saving deposit information in BLOCKED_FUNDS.

```
  convert_address_from_bytes_to_string(&request_info.recipient,
dest_chain_type).unwrap();
```

A potential error in `unwrap()` could leave the funds on the source chain deposited, while preventing depositors from refunding.

Similar usage of `unwrap()` can be found in other parts of the code, such as:

```
if request_info.instruction.is_some() {
        let instruction_token =
Token::Bytes(request_info.instruction.clone().unwrap().0);
        tokens.push(instruction_token);
    }
```

And also in the `handle_deposit_info_update` method:

```
let message_hash: String = fetch_message_hash_from_deposit_info(
        deps.as_ref(),
        &src_chain_id,
        Some(src_contract.clone()),
        deposit_id,
    )
    .unwrap();
    let mut deposit_info: RequestInfo =
        fetch_request_details(deps.as_ref(), &message_hash).unwrap();
```

Recommendation

In Rust, the term "unwrap" means requesting the result of a computation, and if an error occurs during the computation, it leads to a program panic, which stops the program.

In the context of the `fund_deposit_post_processing` function, all possible errors encountered indicate that the deposit should be treated as invalid. It's crucial not to overlook storing the deposit in BLOCKED_FUNDS to allow depositors to withdraw the funds later.

However, it's worth noting that some calls within the `fund_deposit_post_processing` function handle errors correctly. In these cases, if an error occurs, it will be passed up to the caller function, ensuring that the data is stored in BLOCKED_FUNDS.

For instance, consider the following example with appropriate error handling:

`let addr_bytes = convert_address_from_string_to_bytes(dest_token.clone(), dest_chain_type)?;`

In this example, `convert_address_from_string_to_bytes` handles errors, allowing for correct error propagation and, ultimately, the proper handling of deposit data in BLOCKED_FUNDS.

Status: Resolved

The invocation of `convert_address_from_string_to_bytes(dest_token.clone(), dest_chain_type)` now eliminates the possibility of panicking by removing the `unwrap` operation.

Potential overflow and division by zero in isTransactionProfitable function

Title	Potential overflow and division by zero in isTransactionProfitable function
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Status	RESOLVED
Issue	

Involved artifacts

- chains/evm/relayer/executor.go

Description

The `isTransactionProfitable` function within the ChainRelayer component is responsible for determining the profitability of a transaction based on provided gas prices, limits, and token prices. On analysis, it was found that the function could be susceptible to overflow scenarios, as well as a potential division by zero.

Problem Scenarios

1. **Overflow**: When calculating the value of `gasinUsd` , the function multiplies `gasInNative` (which is the product of `gasLimit` and `gasPrice`) with `tokenPrice` . Given sufficiently large values of `gasLimit` , `gasPrice` , and `tokenPrice` , this multiplication could result in an overflow. This overflow could lead to an incorrect evaluation of transaction profitability, which in turn could affect the forwarder's decision-making process regarding transaction relays.
2. **Division by zero**: The function divides by `e.tokenPriceFetcher.GasPrice.Decimals` without any explicit check to ensure that `GasPrice.Decimals` is non-zero. A division by zero would cause a panic and disrupt the service.
3. **Commented out calls**: It was also observed that the calls to `isTransactionProfitable` from the `IRelay` and `IRelayMessage` functions are currently commented out. This implies that, even if the function is perfectly coded, it isn't currently being used to validate transaction profitability. In a production

environment, this oversight means the forwarder will proceed without any profitability checks, potentially leading to economic losses.

Recommendation

1. **Addressing overflow**:
    - Ensure that the multiplication of values in `gasinUsd` is safeguarded against overflow scenarios. Consider using specialized libraries or checking the sizes of numbers before performing the multiplication.
    - Add checks to ensure the provided values (like `gasLimit`, `gasPrice`, `tokenPrice`) are within acceptable bounds.
2. **Addressing division by zero**:
    - Before the division operation, add a check to ensure that `e.tokenPriceFetcher.GasPrice.Decimals` is not zero. If it is zero, return a meaningful error or handle the scenario gracefully.
3. **Uncomment and integrate profitability check**:
    - Ensure that `isTransactionProfitable` is actively called from `IRelay` and `IRelayMessage` to validate transaction profitability. This is vital to ensure that only profitable transactions are relayed, safeguarding economic interests.
    - Assess the reason for commenting out the calls. If they were commented out for debugging or other temporary purposes, they should be reintegrated. If there are underlying issues, they need to be addressed before reintegrating.

Status: Resolved

The updated version of `isTransactionProfitable()` appears to be more computationally efficient, focusing solely on checking if the provided fees are non-negative.

Middleware contract is missing full validation of evm type recipient addresses

Title	Middleware contract is missing full validation of evm type recipient addresses
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	1 LOW
Status	RESOLVED
Issue	

Involved artifacts

- asset-forwarder-middleware

Description

Within the middleware contract, all three `sudo` messages include the recipient's address as an argument, which is received in the form of bytes. It is crucial to validate this recipient address because if the contract does not detect the possibility of an invalid address format, deposit information will not be stored within the `BLOCKED_FUNDS`. This could potentially prevent depositors from withdrawing their funds at a later stage.

In the context of the Ethereum Virtual Machine (EVM), all public addresses adhere to the same format. They begin with the prefix "0x" and consist of 40 alphanumeric characters, comprising a total of 42 characters in length.

In terms of bytes, EVM public addresses represent 20 bytes, and when prefixed with "0x," they become 42 characters long. This consistent format ensures the proper functioning of the contract and the secure handling of recipient addresses.

Problem Scenarios

unit tests `test_evm_to_evm_fund_deposit_18_decimal` part:

```
let recipient_bytes: Vec<u8> = convert_address_from_string_to_bytes(
        EVM_RECIPIENT.to_string().clone(),
        ChainType::ChainTypeEvm.get_chain_code(),
    )
    .unwrap();
    let msg: SudoMsg = SudoMsg::HandleFundDeposit {
        src_chain_id: SRC_CHAIN_ID.to_string(),
        voyager_contract_address: SRC_GATEWAY.to_string(),
        dest_chain_id_bytes:
Binary(hex::decode(REMOTE_CHAIN_BYTES32.to_string()).unwrap()),
        src_token: SRC_TOKEN.to_string(),
        depositor: EVM_SENDER.to_string(),
        deposit_id: 1,
        dest_amount: Uint128::from(90000000u128), //fee_amount = 10000000
        amount: Uint128::from(100000000u128),
        recipient: Binary(recipient_bytes.clone()),
        partner_id: Uint128::new(0u128),
        message: Binary::default(),
        deposit_with_message: false,
    };
    let res = sudo(deps.as_mut(), env.clone(), msg);
    assert!(res.is_ok());
```

During testing, `recipient_bytes` are generated using a function `convert_address_from_string_to_bytes` to ensure that the provided string contains only alphanumeric characters and numerals, thus maintaining a valid format. However, in practice, it's important to note that the `recipient_bytes` received as arguments are not rigorously validated for their format, potentially allowing for the inclusion of non-standard characters.

`fn fund_deposit_post_processing` part:

```
if dest_chain_type == ChainType::ChainTypeEvm.get_chain_code() {
        let addr_bytes = convert_address_from_string_to_bytes(dest_token.clone(),
dest_chain_type)?;
        let address_h160 = H160::from_slice(&addr_bytes);
        dest_token_token = Token::Address(address_h160);

        let addr_bytes =
            convert_address_from_string_to_bytes(dest_contract.clone(),
dest_chain_type)?;
        let address_h160 = H160::from_slice(&addr_bytes);
        dest_contract_token = Token::Address(address_h160);
        let address_h160 = H160::from_slice(&request_info.recipient);
        recipient_token = Token::Address(address_h160);
    } else {
        dest_token_token = Token::String(dest_token.clone());
        dest_contract_token = Token::String(dest_contract);
        let str_address: String =
            convert_address_from_bytes_to_string(&request_info.recipient,
dest_chain_type).unwrap();
        recipient_token = Token::String(str_address);
    }
```

In a situation where destination chain type is evm, only validation of recipient address happens using H160 struct which can only contain 20 bytes.

This checks the evm address length is properly validated but it doesn't check whether address could contain some special characters besides numerals and letters.

Recommendation

It is crucial to ensure that EVM addresses undergo thorough validation to prevent the contract from incorrectly categorizing deposit requests as regular deposits, rather than storing them in the `BLOCKED_FUNDS`. Furthermore, for other chain types, the validation of recipient addresses should be tailored to match the specific address format requirements of the destination chain. This approach guarantees accurate and secure handling of deposit requests across diverse blockchain environments.

Status: Resolved

Issue is resolved as recommended.

Oversight in fee calculation considering chain decimal differences

Title	Oversight in fee calculation considering chain decimal differences
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- utils/utils.go

Description

In the system, there's a potential oversight that might allow for negative fees to be calculated due to decimal differences between source and destination chains, especially if `destAmount` is greater than `amount`. However, according to Router dev, this is attributed to the different number of decimals on source/destination chains: e.g., Tron has 6 decimals, and Polygon has 18, and both have USDT. Although the negative fee might be stored in the database, the forwarder intentionally retains the request, expecting a possible fee update via `HandleDepositInfoUpdate`.

Problem Scenarios

- In the Solidity function `iDeposit`, the parameters `amount` and `destAmount` are emitted in the `FundsDeposited` event.
- On the Forwarder side, `FundDepositEvent` will be listened to, and upon reception, it triggers the `TransformVoyagerFundDepositEvent` operation.
- Inside `TransformVoyagerFundDepositEvent`, the function `FetchDestTokenAndCalculateFee` is invoked.
- A potentially problematic line of code is found in the function: `providedFees := big.NewInt(0).Sub(amount, destAmount)`. If `destAmount` is larger than `amount` due to chain decimal differences, this results in a negative value for `providedFees`.
- This negative `providedFees` value is stored in the database without validation.

Developer insights:

- The Forwarder is designed to incorporate a profitability check. After normalizing both amounts to account for the chain decimal differences, fees are calculated. Negative fees are intentionally ignored, but they might still be stored in the database.
- The forwarder will save the request, even if there's a negative fee, because there's a possibility of updating the fees later through `HandleDepositInfoUpdate`.
- The fee should always exceed the Forwarder's cost of execution on the destination chain.

Recommendation

1. Continue to ensure the normalization of amounts considering the decimal differences of the respective chains.
2. While it's understood that the Forwarder saves requests with negative fees for later adjustments, it would be prudent to add safeguards to prevent storing excessive negative fees which might raise alarms or cause potential financial implications.
3. Implement a mechanism that ensures the total fee is always greater than the Forwarder's execution cost on the destination chain. If it's not, consider raising alerts or blocking such transactions.

Status: Resolved

Issue is resolved as recommended.

Ambiguous design in iDepositInfoUpdate and associated middleware handle_deposit_info_update function

Title	Ambiguous design in iDepositInfoUpdate and associated middleware handle_deposit_info_update function
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- evm/src/AssetForwarder.sol
- contracts/asset-forwarder/src/sudo.rs

Description

The `DepositInfoUpdate` functions, both in the source chain smart contract and the middleware, present an unclear design that intertwines the process of deposit information updates with withdrawal initiations.

In the smart contract's `iDepositInfoUpdate` function, the presence of the following code snippet is counterintuitive:

```
if (initiatewithdrawal) {
    assert(msg.value == 0);
    emit DepositInfoUpdate(
        srcToken,
        0,
        depositId,
        ++depositNonce,
        initiatewithdrawal,
        msg.sender
    );
    return;
}
```

Given the function's name, there's no clear indication that it also initiates a withdrawal process. Such a mixture of functionalities may lead to misunderstandings and potential misuses.

Additionally, on the middleware side, the `handle_deposit_info_update` function breaks the single responsibility principle. Instead of exclusively updating a deposit's `extra_fee`, it also manages the `initiate_withdrawal` process. This fusion of responsibilities can make future maintenance and updates to the codebase more challenging.

Rationale

Despite the unclear design, the Router developer has highlighted some potential justifications:

1. **Cost efficiency**: Implementing a dedicated message for the withdrawal initiation might result in higher costs. This is due to the necessity to update orchestrators, the router chain, and potentially other components.
2. **User considerations**: The current design provides a solution for users who may not have coins on the middleware and therefore won't interact with the middleware contract. It caters to a hybrid solution where users can initiate withdrawals without engaging the middleware contract.

Recommendation

While the current design might cater to certain use-cases and potentially save costs, it's crucial to prioritize clarity and maintainability:

1. **Clear function naming**: Function names should clearly depict their primary functionality. If a function is meant to handle withdrawals, its name should reflect that.
2. **Break functions for clarity**: Instead of amalgamating multiple functionalities into one, it might be better to separate them for better clarity, even if there's a slight overhead in terms of development or execution costs.
3. **Documentation**: If certain design choices are made for cost efficiency or specific use-cases, they should be well-documented. This ensures that any developer, whether they're from the core team or an external contributor, understands the rationale behind the design choices.

Miscellaneous: areas of improvement

Title	Miscellaneous: areas of improvement
Project	Router: Q4 2023
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- voyager-forwarder
- asset-bridge-contracts

Description

Upon reviewing the provided code snippets, multiple areas of improvement have been identified:

1. **Cohesion in methods**: The function `FetchDestTokenAndCalculateFee` contains logic that can be further separated based on different responsibilities. Specifically, it fetches the destination token and calculates the fee, which are two separate tasks.
2. **Magic numbers & hardcoded values**: There are instances in the codebase where magic numbers and hardcoded values are present, for example:
    - Using the hardcoded chain ID `"2494104990"`.
    - Assigning an expiry time with the magic number `uint64(50000)`.
    - values for transaction types, routeRecipient hash comparisons, and gas limits/prices
3. **Unused parameters**: In several functions (`ChainRelayer.Start, VoyagerListener.Start, RequestProcessor.ProcessIRelayRequest, NearChainRelayer.Start` to name a few), parameters such as `ctx` and `sysErr` have been declared but are not utilized within the code, potentially indicating missed functionality or oversight.
4. **Transaction status oversight**: In a portion of the code where transaction receipts are fetched, the transaction's execution status, a critical attribute, is not checked or considered, leading to potential misinterpretations of transaction outcomes.
5. **Redundant branch conditions:** The logic branches for `dest_chain_type` 1 and 5 are identical, invoking `send_evm_reserve_outbound_request` with the same parameters. This redundancy could be streamlined into a single case to improve code clarity and maintainability.

6. **Duplicated code:** Within the `handle_evm_inbound_request` function, duplication exists between the handling of `encode_type` 0 and 1. This duplication increases the codebase's size and complexity unnecessarily and may introduce maintenance issues. A refactoring effort is recommended to abstract the common logic and reduce repetition.
7. **Error handling concerns:** The usage of `unwrap_or_default()` has been noted in multiple places. This practice could lead to silent failures where defaults are not intended, thus masking potential errors. It would be prudent to handle errors explicitly, logging them and returning early where defaults are not an acceptable fallback.
8. **Avoidable cloning:** It's generally more efficient to pass variables by reference if they are not being mutated to avoid the computational cost of cloning.
9. **Variable naming:** The usage of type names as variable identifiers, such as `u256`, is not advisable. It detracts from the readability and does not communicate the variable's role within the context. It is preferable to use descriptive names that reflect the content or purpose of the variable.
10. **Unused response initialization:** An instance of `Response<RouterMsg>` is created and named `temp_response`, but it's only used to call `.add_submessages(temp_response.messages)`, which is empty. This appears to be a redundant operation and may indicate an oversight or incomplete implementation.

Appendix: 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.

Impact Score	**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:

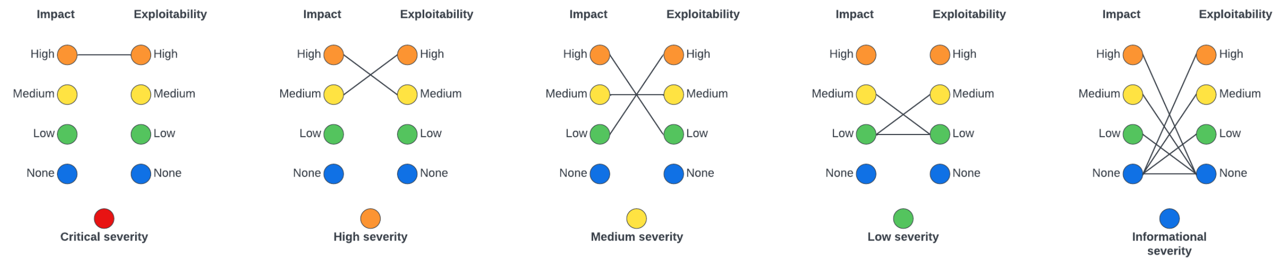
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

    - *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)

Exploitability Score	**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.



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.

Severity Score	**Examples**
🔴 **Critical**	Halting of chain via a submission of a specially crafted transaction

Severity Score	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

Disclaimer