



Security Audit Report

Router Protocol V2

Authors: Andrey Kuprianov, Darko Deuric, Marko Juric

Last revised 9 January, 2024

Table of Contents

Audit Overview	1
The Project	1
Scope of this audit	1
Conducted work	1
Conclusions	1
Audit Dashboard	3
Target Summary	3
Engagement Summary	3
Severity Summary	3
System Overview.....	4
Router Protocol	4
Workflow	4
Architectural components	5
Findings	7
Multiple panics in Relayer may disrupt relaying workflows	10
Dangerous error handling in Orchestrator and Relayer	13
Unchecked validator jailing: a ticking time bomb for chain stability	16
Orchestrators may permanently fail to deliver crosschain messages	19
Using WHERE without ORDER BY in Orchestrator and Relayer database queries may lead to out-of-order message passing	24
Excessive Relayer queries may cause Router chain DOS	27
Loss of crosschain messages/acks due to validator set updates	31
Multiple protocol-level issues inherited from Gravity Bridge design	34
Unstable chain operation due to gas consumption and panic handling in ExecuteInboundRequest function	36
Security concern: IBC permissionless interaction	39
Non-deterministic iteration of attmap in ClaimEventSlashing function	41
Outdated and deprecated versions of OpenZeppelin contracts used	43
Outdated version of Solidity compiler is accepted	45
Deficient decoding of contract metadata: lack of validation, error handling, and default values	46

Potential input issue: uninitialized price.PriceFeeder argument	48
Execution order inconsistency between pricefeed module and crosschain module endblockers	50
Efficiency and conditional sequence concerns in createValsets function: suboptimal ordering of conditions	52
Inefficient function call and redundant validation in tallyChainAttestations function: optimizing attestation processing	55
Redundant recalculation of total validator power in TryAttestation function	58
Redundant claimHash calculation in cross-chain request processing	60
Inefficient oracle tasks creating	62
Suboptimal iteration and switch-case mechanism in Execute and SettleFees functions	64
Inefficient validator search in nested loop	67
Inefficient validator set update in attestation processing	70
Lack of unit tests in Router chain modules	73
Possible unnecessary saving Crosschain(Ack) request to storage	75
Replace "require" with "revert" and custom errors to save gas	77
Various optimizations for Solidity contracts	79
Appendix: Vulnerability Classification	81
Impact Score	81
Exploitability Score	81
Severity Score	82
Disclaimer	84

Audit Overview

The Project

In July 2023, the Router development team engaged Informal Systems to conduct a security audit of the Router Protocol V2. Router protocol enables cross-chain communication and represents the solution to the blockchain interoperability problem. Simply put, the Router architecture allows contracts on one chain to interact with contracts on other chains in a secure and decentralized manner.

Scope of this audit

The audit was conducted by the following individuals:

- Andrey Kuprianov
- Darko Deuric
- Marko Juric

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

- Router chain modules: x/attestation, x/crosschain, x/metastore, x/multichain and x/pricefeed
- Gateway smart contracts (Solidity and wasm) - which are the interfaces for application contracts to interact with Router's bridging infrastructure
- Middleware contract - The application-specific bridge contract which is deployed on the Router chain that include the logic required to process the incoming request from a third-party chain and generate an outgoing request to another third-party blockchain.
- Orchestrator application - with main functionality to listen to incoming cross-chain requests from other chains, and post them on the Router chain
- Relayer application - permissionless entity that relay data from the Router chain to a specific destination chain

Conducted work

At the kickoff meeting, the Router team gave us a brief introduction to the Router protocol, and provided codebase which needs to be audited. Before starting the audit, our team needed time to study the Router protocol flow, as it is consisted of several critical components which also included off-chain components. The team reviewed all the existing specifications and technical diagrams for the Router protocol. Our team performed high-quality line-by-line manual code review with a focus mainly on code correctness and the critical points analysis specific to each component. Over the shared Slack channel, we discussed all the necessary information for sync meeting arrangements, the Github audit repository, etc. Also, the Router team provided additional documentation material such as video presentations which helped us better understand the project.

Conclusions

The overall assessment of the Router chain codebase indicates a commendable level of quality. The codebase is well-structured and exhibits a high degree of readability, facilitating ease of comprehension and follow-through.

Strengths:

- The codebase demonstrates a well-organized structure, contributing to its overall quality.
- The code is coherent and logical, enhancing its comprehensibility.

Areas for Improvement:

- **Unit Testing:** Notably, there is a notable deficiency of unit tests across various modules within the Router chain. Implementing comprehensive unit tests would significantly enhance the robustness and reliability of the codebase.
- **Orchestrator and Relayer Components:** These two components, Orchestrator and Relayer, require particular attention in terms of codebase and design improvements. Critical bugs are predominantly concentrated within these components.

Severity Classification of Issues:

- **Critical:** Eight critical issues have been identified, underscoring the need for urgent attention and resolution.
- **High:** Additionally, one issue has been categorized as high severity, warranting immediate attention.
- **Medium, Low, and Informational:** Several issues fall under the medium, low, and informational categories, each requiring appropriate prioritization and resolution based on their respective impact and relevance.

Audit Dashboard

Target Summary

- **Type:** Specification and Implementation
- **Platform:** Golang, Solidity, Rust
- **Artifacts:**
 - Router chain [repo](#), commit: [5e7859](#)
 - Gateway contract [repo](#), commit: [4a0abc3](#)
 - Bridge contract [repo](#), commit: [b03067b](#)
 - Relay [repo](#), between commits [f66f708c](#) and [f4281dee](#)
 - Orchestrator [repo](#), between commits [aa9ef5e6](#) and [5c23926f](#)

Engagement Summary

- **Dates:** 19.07.2023 to 12.09.2023
- **Method:** Manual code review, protocol analysis
- **Employees Engaged:** 3

Severity Summary

Finding Severity	#
Critical	6
High	1
Medium	4
Low	3
Informational	14
Total	28

System Overview

Router Protocol

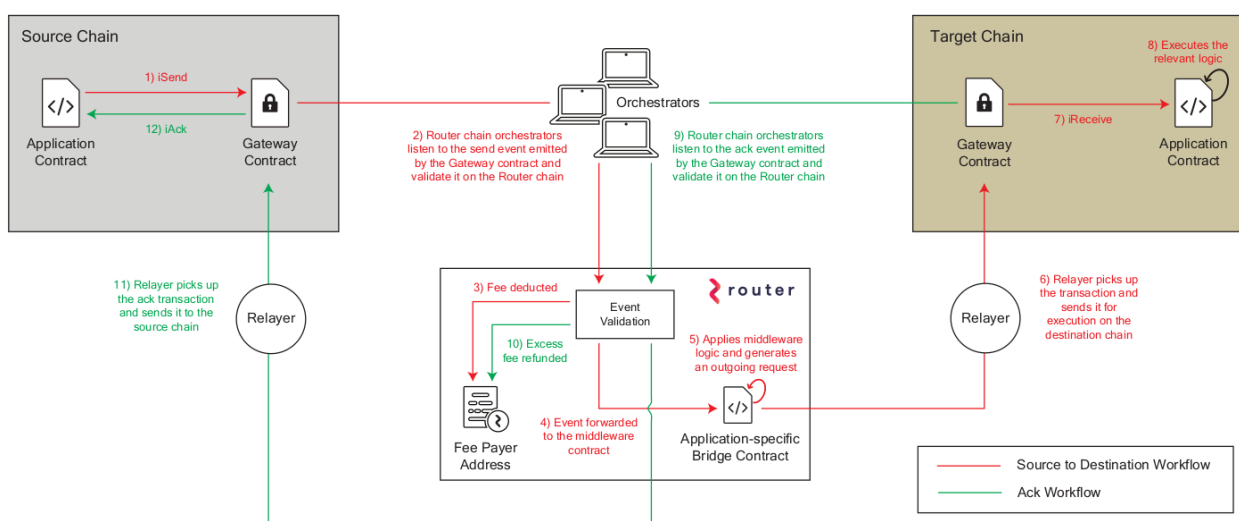
- powered by a tendermint-based Router chain, built using the Cosmos SDK
- enables state transitions across chains where Router chain sits as a hub between various EVM and non-EVM ecosystems

Features that set Router chain apart from other interoperability solutions include, but are not limited to:

1. **Support for middleware contracts:** Maintain states and implement custom business logic directly in the bridging layer.
2. **Plug-and-play for developers:** Router has a transcendent open-source developer tooling suite to assist with the continuous integration and development of cross-chain dApps.
3. **CrossTalk:** Developers looking to build cross-chain applications without any custom bridging logic can leverage Router's easy-to-integrate smart contract library, CrossTalk.
4. **Support for various kinds of use cases:** Batching, sequencing, and atomicity can be enforced directly from the Router chain.
5. **Flexibility:** Router provides developers with the utmost flexibility over their bridging model (stateless or stateful), security model, and smart contract platform (EVM or CosmWasm):
6. **Data aggregation:** Contracts on the Router chain can serve as data aggregation modules for various cross-chain and multi-chain applications.
7. **Cross-chain meta transactions:** By leveraging Router as their cross-chain infra provider, applications can enable gasless cross-chain transactions by delegating the execution of a request to a third-party service.
8. **Composability:** The Router chain will have inbuilt support for global applications such as oracles and liquidity pools/bridges, to name a few, which will help in easier integration of other applications.

Workflow

Generating and Sending a Cross-chain Request



step 1: a) A user initiates a cross-chain action on an application's smart contract on the source chain.
b) The application contract calls the `iSend()` function on the Router Gateway contract.

step 2: The Gateway contract on the source chain emits an event that is listened to by the orchestrators on the Router chain.

step 3: Once the event is validated, the Router chain will deduct the fee from the `feePayerAddress` for that dApp on the Router chain. A dApp's fee payer is its designated entity on the Router chain that is responsible for paying the fees for all of its cross-chain requests.

step 4: Once the fee is deducted successfully, the request is sent to the application's bridge contract on the Router chain.

step 5: After the request reaches the bridge contract on the Router chain, the bridge contract will validate if the source contract address from which the request has been generated is correct or not, following which it will apply its custom logic.

step 6: After the transaction initiated by the bridge contract is mined on the Router chain, the outgoing request is validated by the orchestrators.

step 7: Once the event is validated, a relayer picks up the request and forwards the event to the Router Gateway contract on the destination chain.

step 8: a) The Gateway contract on the destination chain calls the `iReceive()` function on the application contract on the destination chain.

b) The application contract on the destination chain will take appropriate actions based on the data transferred.

***note:** In case there is no need for application-specific bridging logic, applications do not need to include a bridge contract on the Router chain. They can use Router's CrossTalk framework

Generating and Sending an Acknowledgment

step 1: After the `iReceive()` function execution is complete on the destination chain, the destination chain's Gateway contract emits an acknowledgment event that is listened to by the orchestrators on the Router chain.

step 2: Once the orchestrators validate the ack request, the acknowledgment request is processed on the Router chain:

a) $(\text{RelayerIncentive} + \text{FeeConsumed})$ is transferred to the relayer address

b) $(\text{OutgoingTxFee} - \text{FeeConsumed})$ is refunded to the bridge contract on the Router chain.

step 3: Once the acknowledgment is processed on the Router chain, it is sent to the application's bridge contract.

step 4: If the dApp had opted to receive the ack back on the source chain, the relayers send it to the source chain's Gateway contract. If not, it is discarded.

step 5: The Gateway contract on the source chain sends the ack to the application's source chain contract.

Architectural components

- alongside the main Router protocol component, Router chain, there are also other important components of the protocol:

Application contracts:

- These are contracts deployed by applications on third-party chains and serve as the intermediary between end users of the application and the Router cross-chain infra.
- In the lifecycle of a cross-chain transaction, these contracts are responsible for making the `iSend()` function call to the Gateway contracts on the source chain by passing the address of the bridge contract on the Router chain as well as the relevant payload
- On the destination chain, application contracts will execute the instructions forwarded by the Gateway contract

Gateway contracts:

- serves as the interfaces for application contracts to interact with Router's bridging infrastructure
- Gateway contract functions include: `iSend()`, `iReceive()` and `setDappMetadata()`

Orchestrators

- Router orchestrators are entities that listen to incoming cross-chain requests from other chains, attest their validity, parse them into a unified format and post them on the Router chain.
- These attested requests can then be picked up by the relayers and forwarded to the destination chain.
- All validators must run an orchestrator instance to be a part of the Router chain ecosystem.

Application-specific Bridge Contracts

- The application-specific bridge contracts are middleware contracts deployed on the Router chain that include the logic required to process the incoming request from a third-party chain and generate an outgoing request to another third-party blockchain.
- These contracts can be written in either Rust (compiled using CosmWasm) or Solidity (compiled using the EVM compiler provided by Ethermint).
- To ensure that a faux contract doesn't execute any of the functions in these contracts, a bridge contract should always maintain a **mapping of the chainId and addresses of all the application contracts** (deployed on the third-party chains) that can execute its functions.

Relayers

- permissionless entities that relay executable proposals from the Router chain to a specific destination chain
- The Router chain has a set of relayers operated by various third parties, which distributes the responsibility.
- In the set, each relayer listens to the Router chain and relays data to the destination chains as and when required

Findings

Title	Type	Severity	Status
Multiple panics in Relayer may disrupt relaying workflows	IMPLEMENTATION PRACTICE	4 CRITICAL	RESOLVED
Dangerous error handling in Orchestrator and Relayer	IMPLEMENTATION PRACTICE	4 CRITICAL	RESOLVED
Unchecked validator jailing: a ticking time bomb for chain stability	IMPLEMENTATION PRACTICE	4 CRITICAL	RESOLVED
Orchestrators may permanently fail to deliver crosschain messages	IMPLEMENTATION	4 CRITICAL	RESOLVED
Using WHERE without ORDER BY in Orchestrator and Relayer database queries may lead to out-of-order message passing	IMPLEMENTATION	4 CRITICAL	RESOLVED
Loss of crosschain messages/acks due to validator set updates	PROTOCOL IMPLEMENTATION	4 CRITICAL	RESOLVED
Unstable chain operation due to gas consumption and panic handling in ExecuteInboundRequest function	IMPLEMENTATION	3 HIGH	RESOLVED
Security concern: IBC permissionless interaction	IMPLEMENTATION	2 MEDIUM	RESOLVED
Non-deterministic iteration of attmap in ClaimEventSlashing function	IMPLEMENTATION	2 MEDIUM	RESOLVED
Outdated and deprecated versions of OpenZeppelin contracts used	IMPLEMENTATION	2 MEDIUM	RESOLVED

Title	Type	Severity	Status
Outdated version of Solidity compiler is accepted	IMPLEMENTATION	2 MEDIUM	RESOLVED
Deficient decoding of contract metadata: lack of validation, error handling, and default values	IMPLEMENTATION	1 LOW	RESOLVED
Potential input issue: uninitialized price.PriceFeeder argument	IMPLEMENTATION	1 LOW	RESOLVED
Execution order inconsistency between pricefeed module and crosschain module endblockers	IMPLEMENTATION	1 LOW	RESOLVED
Efficiency and conditional sequence concerns in createValsets function: suboptimal ordering of conditions	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Inefficient function call and redundant validation in tallyChainAttestations function: optimizing attestation processing	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Redundant recalculation of total validator power in TryAttestation function	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Redundant claimHash calculation in cross-chain request processing	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Inefficient oracle tasks creating	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED

Title	Type	Severity	Status
Suboptimal iteration and switch-case mechanism in Execute and SettleFees functions	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Inefficient validator search in nested loop	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Inefficient validator set update in attestation processing	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Lack of unit tests in Router chain modules	PRACTICE	0 INFORMATIONAL	ACKNOWLEDGED
Possible unnecessary saving Crosschain(Ack) request to storage	IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Excessive Relayer queries may cause Router chain DOS	PROTOCOL IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED
Replace "require" with "revert" and custom errors to save gas	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Various optimizations for Solidity contracts	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Multiple protocol-level issues inherited from Gravity Bridge design	PROTOCOL IMPLEMENTATION	0 INFORMATIONAL	ACKNOWLEDGED

Multiple panics in Relayer may disrupt relaying workflows

Title	Multiple panics in Relayer may disrupt relaying workflows
Project	Router Protocol V2
Type	IMPLEMENTATION PRACTICE
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [Router Relayer at f66f708c](#)

Description

Router relayers are supposed to be run by Router validators, alongside with the main Router chain binary and the Router orchestrator. Relayers are a critical component of the Router infrastructure, which relay messages from the Router chain to the other chains that Router connects. As such, relayers need to operate continuously and uninterrupted for unlimited periods of time, irrespective of possible transient interruptions of other services they depend upon.

Unfortunately, the current relayer implementation contains a large number of `panic()` calls, which effectively shutdown the relayer; in total we have counted 43 `panic()` calls in non-testing code. While some of these panics may be considered legitimate, because they represent local conditions, like [inability to read configuration](#), or [database access errors](#), others shutdown the relayer upon inability to perform an operation on external service. Below are a few examples.

Panic on failure to obtain gas prices

In [chains/evm/validation/validation.go#L51-L54](#), the relayer panics when it fails to obtain a gas price estimate from an external service:

```
gasPrice, err := validation.client.SuggestGasPrice(ctx)
if err != nil {
    panic(err)
}
```

Panics on failure to access the Ethereum RPC client

In [chains/evm/executor/gateway.go#L98-L101](#), the relayer panics when it fails to obtain a transaction receipt:

```
receipt, err := gateway.client.WaitAndReturnTxReceipt(*hash)
if err != nil {
    panic(err)
}
```

The same behavior can be observed in:

- [chains/evm/executor/gateway.go#L145-L148](#)

Panics on failure to execute a Gateway contract action

In [chains/evm/validation/validation.go#L105-L108](#), the relayer panics on failure to invoke a function of the Ethereum Gateway contract:

```
currentValsetNonce, err := validation.gatewayContract.StateLastValsetNonce()
if err != nil {
    panic(err)
}
```

Similar behavior can be observed in:

- [chains/substrate/calls/contracts/contracts.go#L22-L25](#)
- [chains/substrate/calls/contracts/contracts.go#L31-L34](#)

Panics on failure to obtain information from the Router chain

In [routerlistener/routerlistener.go#L97-L103](#), the relayer panics when it fails to obtain validator sets from the Router chain:

```
valsetList, err := listener.routerChainClient.GetAllValsets(ctx, &query.PageRequest{
    Key:  nextKey,
    Limit: 50,
})
if err != nil {
    panic(err)
}
```

Similar behavior is observed in:

- [routerlistener/routerlistener.go#L119-L125](#)
- [routerlistener/routerlistener.go#L142-L148](#)
- [util/valsetUpdateConfirmations.go#L12-L15](#)

Problem Scenarios

As can be seen above, Router relayers panic on failures from the multitude of external services:

- Gas price oracle
- Router chain client
- External chain clients

All of the above are very likely to be executed on other machines, accessed via networks, and thus be subject to various kinds of service interruptions, including, but not limited to:

- target service outage
- network disruptions

As a result of transient inaccessibility of any of the external services, the relayers would panic and shutdown. The consequences are manifold:

- The Router infrastructure stops to function.
- The Router validators have to perform expensive maintenance, restarting the relayers.
- The restarted relayers have to go through the initialization phase, including e.g. (re-)initializing the databases, (re-)connecting to various clients, etc. This leads to the increased local computational load on the node running the relayer.
- The restarted relayers have to fetch the up-to-date information from all external clients. As all relayers are likely to experience the same service interruption simultaneously, their shutdown and restart will also happen synchronously, thus leading to a simultaneous increased service load on the external clients. This, in turn, may lead to the DOS from the side of the external clients, again shutting down the relayers, and putting the system into an unrecoverable, cascading failure loop.

Recommendation

The present finding demonstrates on a smaller scale the architectural problem that exists in the current relayer implementation, which doesn't seem to be designed for failure resiliency. Sufficient number of sources exist on the subject, as well as certain established architectural patterns; we can recommend e.g. to read the [Reliability documentation of Microsoft Azure Well-Architected Framework](#), and in particular the [Reliability design patterns](#). Out of those, the most applicable in the Router relayer context seem to be:

- [Bulkhead pattern](#): to isolate relayer parts wrt. failures happening on independent chains.
- [Retry pattern](#): to handle with transient service interruptions.
- [Circuit Breaker pattern](#): to avoid cascading failures on prolonged service interruptions.

As the first step towards correcting the problem, we recommend to apply the [Retry pattern](#) in all places where relayer currently panics on failures to access external services. As subsequent, more advanced steps, we recommend to apply the [Bulkhead pattern](#), and the [Circuit Breaker pattern](#).

Dangerous error handling in Orchestrator and Relayer

Title	Dangerous error handling in Orchestrator and Relayer
Project	Router Protocol V2
Type	IMPLEMENTATION PRACTICE
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [Router Orchestrator at aa9ef5e6](#)
- [Router Relayer at f66f708c](#)

Description

Both Router orchestrator and relayer contain a vast assortment of dangerous error handling patterns: in many cases errors are ignored, in many others the code path continues despite the error. We list below the main categories of such dangerous error handling patterns we have observed in the codebase.

Errors from sub-calls are ignored

In many cases we have observed the code like in [router-relayer/chains/evm/transformer/transformer.go#L89-L106](#):

```
id, _ := util.CreateCrosschainDBId(crosschainMessage)
...
msg := types.NewCrosschainMessage(id, sigs, RouterRequestPayload, currentValsetArgs,
metadata.DestGasPrice, gasLimit)
...
return msg
```

Below is the implementation of [CreateCrosschainDBId](#):

```
func CreateCrosschainDBId(crosschainRequest crosschainTypes.CrosschainRequest)
(string, error) {

    if len(crosschainRequest.SrcChainId) == 0 || crosschainRequest.RequestIdentifier
    == 0 {
```



```

        return "", fmt.Errorf("invalid ID creation due to data missing")
    }
    return fmt.Sprintf("%v:%v", crosschainRequest.SrcChainId,
        crosschainRequest.RequestIdentifier), nil
}

```

Notice that there are multiple cases when this function may return an error. When the error is ignored, as in the fragment above, the code will continue, and will happily construct a message with an empty `id`, which will be returned to the caller.

There are multiple other places in the codebase with the same behavior; the incomplete list follows:

- [router-orchestrator/attester/inbound/inbound.go#L81](#)
- [router-orchestrator/attester/outbound/outbound.go#L81](#)
- 22 cases in [router-orchestrator/listener/near/eventprocessor/eventprocessor.go](#)
- [router-orchestrator/listener/near/eventprocessor/transformer.go#L26](#)
- [router-orchestrator/listener/near/eventprocessor/transformer.go#L78](#)
- [router-orchestrator/store/txqStore.go#L23](#)
- [router-orchestrator/utls/crypto/sr25519/sr25519.go#L46](#)
- [router-relayer/chains/evm/transformer/transformer.go#L69](#)
- [router-relayer/chains/evm/transformer/transformer.go#L89](#)
- [router-relayer/chains/evm/transformer/transformer.go#L132](#)
- [router-relayer/chains/evm/transformer/transformer.go#L89](#)
- [router-relayer/crypto/sr25519/sr25519.go#L43](#)
- [router-relayerexample/app/app.go#L101](#)
- [router-relayerexample/app/app.go#L110](#)
- [router-relayer/store/txqStore.go#L23](#)
- [router-relayer/util/crosschainAckConfirmations.go#L16](#)
- [router-relayer/util/crosschainConfirmations.go#L17](#)

You may obtain other such places using the text search with the `, _ :=` search string. Other instances of this pattern are possible, which are not exhibited by such search.

The code path continues despite the error

Another dangerous error handling pattern we've observed frequently in the codebase is that the error from a sub-call is received, logged, but the execution continues despite the error, using the erroneous values obtained from the sub-call. An example of this pattern can be found in [router-relayer/chains/evm/transformer/transformer.go#L158-L161](#):

```

id, err := util.CreateValsetDBId(latestValset)
if err != nil {
    transformer.log.WithFields(logrus.Fields{"Error": err}).Debug("Error while
creating ID for DB")
}
...
msg := types.NewValsetMessageParams(id, filteredSignatures, latestValsetArs,
currentValsetArgs)
return msg

```

A similarity with the example in the previous section can be observed. In this example, the error is though not ignored completely, it is only logged, but the execution continues despite the error, and the message with the wrong `id` is returned.

There are also other instances of the same pattern scattered throughout the codebase, some instances of which are listed below. Notice though that no simple text search pattern exists to reveal all such cases.

- [router-orchestrator/attester/inbound/inbound.go#L50-L56](#)
- [router-orchestrator/attester/inbound/inbound.go#L92-L99](#)
- [router-orchestrator/attester/inbound/inbound.go#L147-L169](#)
- [router-orchestrator/attester/outbound/outbound.go#L151-L170](#)
- [router-orchestrator/health/health.go#L42-L49](#)
- [router-orchestrator/listener/evm/gatewayeventprocessor/transformer.go#L112-L128](#)
- [router-relayer/chains/evm/transformer/transformer.go#L194-L209](#)
- [router-relayer/routerlistener/routerlistener.go#L166-L169](#)
- [router-relayer/routerlistener/routerlistener.go#L185-L188](#)
- [router-relayer/routerlistener/routerlistener.go#L204-L207](#)
- at least 9 in [router-relayer/chains/evm/validation/validation.go](#)

Problem Scenarios

Some of the unhandled errors listed above may turn out to be harmless due to e.g. errors being filtered out by other conditions. Nevertheless, the whole spectrum of problems arising from such dangerous error handling patterns is possible, simply due to the number of places where errors are ignored or unhandled. Among the alternatives are:

- The least harmful scenario (which is still very dangerous) when the returned value being processed after the ignored error is such that it doesn't represent a valid one: e.g. a `null` pointer, or an element of an enumeration which represents an invalid value. The most probable consequence is that the code using this value down the stream will panic, e.g. on trying to dereference the `null` pointer, shutting down the respective binary (orchestrator or relayer).
- An error is ignored, and a result value which is a valid value for the given type is processed down the code path. Examples could be `0` for an integer type, an empty string, or a partially parsed type from its string representation, when parsing errors are ignored. The result value is processed, propagates to other parts of the code, and leads to an unknown cascade of potentially harmful effects.

Recommendation

There are standard error handling practices for the Go language that we recommend to follow; for the entry point see e.g.:

- <https://go.dev/blog/error-handling-and-go>
- <https://earthly.dev/blog/golang-errors/>

We recommend to thoroughly inspect the whole codebase of Router orchestrator and relayer, and refactor it to make sure all errors are handled correctly.

Status: Resolved

Error handling has undergone significant enhancement in various functions, including but not limited to:

```
CreateValsetConfirmRequest() , OutboundAttester.Start() ,
CreateCrosschainConfirmRequest() , InboundAttester.Start() ,
CreateCrosschainAckConfirmRequest() , ProcessInboundEvents() ,
SortAndTransformInboundEventsByEventNonce() , and TransformGatewayISendEvent() .
```

Unchecked validator jailing: a ticking time bomb for chain stability

Title	Unchecked validator jailing: a ticking time bomb for chain stability
Project	Router Protocol V2
Type	IMPLEMENTATION PRACTICE
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [x/attestation/abci.go](#)

Description

The [code in question](#) processes attestations and attempts to jail validators under certain conditions. However, it lacks proper checks to prevent the same validator from being jailed multiple times, which could result in critical issues within the blockchain network.

```
for _, attestation := range unObs {
    for _, valaddr := range attestation.Votes {
        validator, _ := sdk.ValAddressFromBech32(valaddr)
        val := k.StakingKeeper.Validator(ctx, validator)
        cons, _ := val.GetConsAddr()
        consPower := k.StakingKeeper.GetLastValidatorPower(ctx, validator)

        k.StakingKeeper.Slash(
            ctx, cons, ctx.BlockHeight(),
            consPower,
            slashFractionConflictingClaim,
        )

        k.StakingKeeper.Jail(ctx, cons)
    }

    k.DeleteAttestation(ctx, attestation)
}
```

The `Jail` function should not be called more than once for the same validator to avoid disrupting the network's stability.

Problem Scenarios

The problematic code snippet provided reveals a potential vulnerability in the current implementation of the blockchain protocol. Specifically, there is a risk of calling `k.StakingKeeper.Jail(ctx, cons)` multiple times for the same validator, which could lead to a critical issue within the blockchain network. This vulnerability arises from two distinct scenarios:

1. **Validator (Orchestrator) submits multiple conflicting votes for the same nonce:** In this case, if the same validator (orchestrator) submits two or more conflicting votes for the same nonce, the `Jail` function is called multiple times for that validator, which can lead to a panic in the Cosmos SDK's `Jail` implementation.
2. **Validator (Orchestrator) submits conflicting votes for different nonces:** If the validator (orchestrator) submits conflicting votes for two or more attestations with different nonces, it can also trigger multiple calls to the `Jail` function for the same validator.

The `panic` inside the `Jail` function is problematic because it can disrupt the normal operation of the blockchain, potentially causing the chain to halt.

```
if validator.Jailed {
    panic(fmt.Sprintf("cannot jail already jailed validator, validator: %v\n",
validator))
}
```

Recommendation

To address this issue and prevent the panic within the `Jail` function, it is essential to add a check to ensure that a validator is not already jailed before calling the `Jail` function. Here is a recommended code modification:

```
for _, attestation := range unObs {
    for _, valaddr := range attestation.Votes {
        validator, _ := sdk.ValAddressFromBech32(valaddr)
        val := k.StakingKeeper.Validator(ctx, validator)
        cons, _ := val.GetConsAddr()
        consPower := k.StakingKeeper.GetLastValidatorPower(ctx, validator)

        // Check if the validator is already jailed before attempting to jail them
again.
        if !val.IsJailed() {
            k.StakingKeeper.Jail(ctx, cons)
        }

        k.StakingKeeper.Slash(
            ctx, cons, ctx.BlockHeight(),
            consPower,
            slashFractionConflictingClaim,
        )
    }
}
```

```
k.DeleteAttestation(ctx, attestation)
}
```

Adding this check ensures that the `Jail` function is called only if the validator is not already jailed, thereby preventing the possibility of triggering a panic and halting the blockchain. This modification enhances the robustness and reliability of the blockchain network.

Orchestrators may permanently fail to deliver crosschain messages

Title	Orchestrators may permanently fail to deliver crosschain messages
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [Router Orchestrator at aa9ef5e6](#)

Description

On the high level, the crosschain request on its way from the source chain to the Router chain passes through the following stages:

1. The request originates at the source chain via calling the `iSend` method of the `GatewayUpgradeable` contract;
2. Each Router orchestrator listens for the `ISendEvent` emitted by the `GatewayUpgradeable` contract, and transforms them to the Router chain transaction message;
3. An orchestrator collects multiple transaction messages into a batch, and submits the resulting transaction to the Router chain.
4. The submitted transaction messages are processed on the Router chain by the `Crosschain` module, which invokes the `Attestation` module to attest the claims; *the latter, in particular, makes sure that all submitted event nonces from the orchestrator are consecutive.*

The approximate path of the crosschain request *within* the Orchestrator is as follows:

1. The processing commences in the function `listener/gatewaylistener.go:Start()`, which is executed once upon orchestrator (re-)start. The function compares the block heights of the last processed block from the source blockchain as stored in the internal database, and as retrieved from the Router chain, and proceeds as outlined below.

```

if lastProcessedBlockFromDB.BlockHeight >
lastProcessedBlockFromChain.EventHeight {

chainListener.eventProcessor.ProcessInboundEvents(lastProcessedBlockFromDB.BlockHeight, lastProcessedBlockFromDB.EventNonce)
} else {
    if contractConfigRes.ContractConfig.ContractHeight ==
lastProcessedBlockFromChain.EventHeight {
        fmt.Println("chainListener.ChainSpec.LastObservedEventNonce",
contractConfigRes.ContractConfig.LastObservedEventNonce)
        // The first event is at height where gateway contract is deployed.

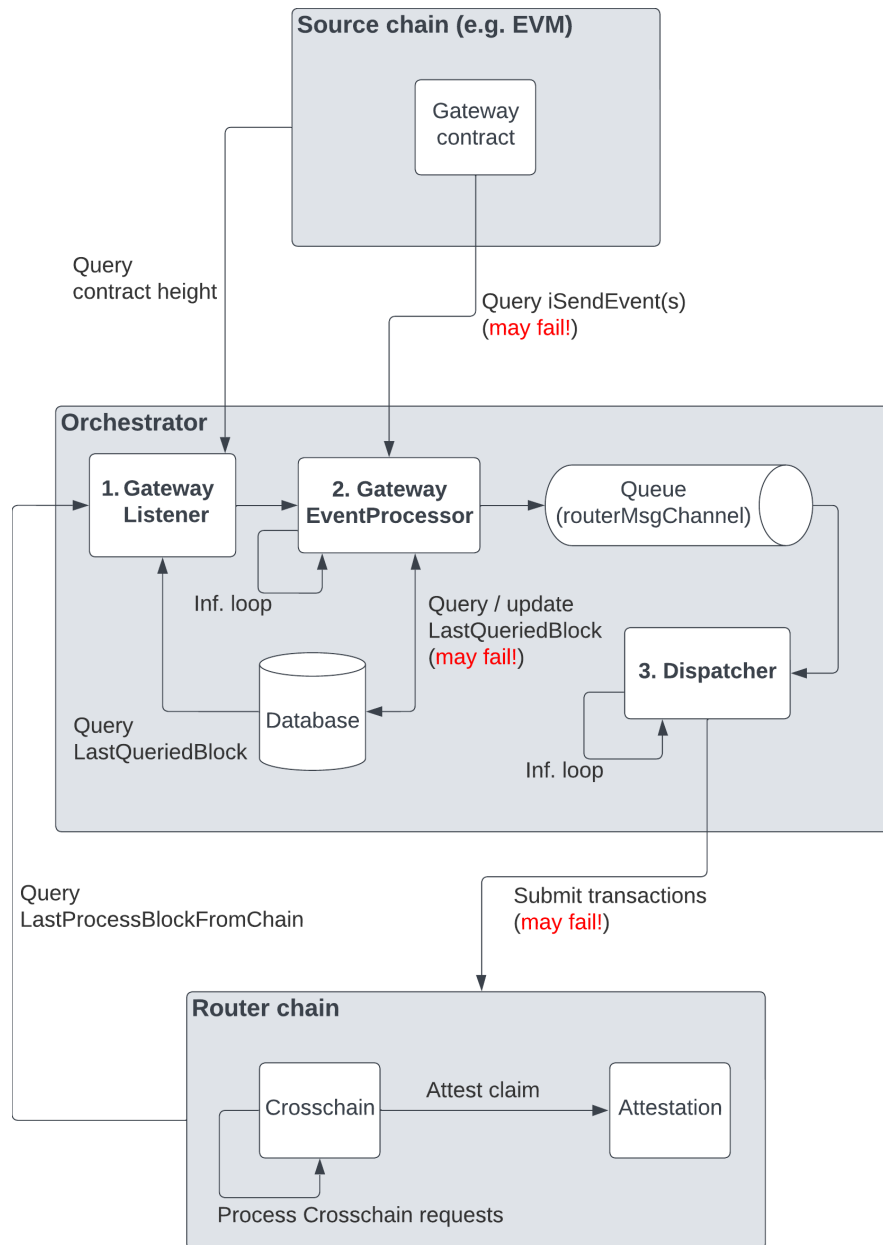
chainListener.eventProcessor.ProcessInboundEvents(contractConfigRes.ContractConfig.ContractHeight-2, lastProcessedBlockFromChain.EventNonce)
    } else {

chainListener.eventProcessor.ProcessInboundEvents(lastProcessedBlockFromChain.EventHeight-1, lastProcessedBlockFromChain.EventNonce)
    }
}

```

2. The processing continues in per-chain event processor, for our case we take `listener/evm/gatewayeventprocessor/gatewayeventprocessor.go:ProcessInboundEvents()`. As its first parameter this function takes `lastQueriedBlock`, passed from the previous function, and in the endless loop:
 - a. retrieves the events from the source chain (Ethereum in this case) starting from `lastQueriedBlock`;
 - b. sorts the events according to their event nonce;
 - c. transforms the events into transaction messages for the Router chain;
 - d. sends the messages to the internal queue for later processing;
 - e. updates the `lastQueriedBlock` stored in the local database to the height of the block last queried from the source chain.
3. The processing finishes in the function `dispatcher/dispatcher.go:Start()`, which, in the endless loop, fetches messages from the internal queue, and submits them to the Router chain.

We illustrate processing of crosschain events within Orchestrator graphically below.



The problems may occur at different stages of the event processing; in particular:

- Fetching of crosschain events in function `FetchAndTransformInboundEvents()` may panic after a fixed number of retries (3);
- Updates to the local database may fail in function `UpdateLastProcessedBlock()`; notice absence of error handling on the last line of that function.
- Submission of transactions, happening in function `dispatcher.go:Start()` may fail; and the errors are silently ignored (only the log message is added, but processing continues):

```
func (dispatcher *Dispatcher) Start(ctx context.Context) {
    for {
        select {
            case routerMsg := <-dispatcher.routerMsgChannel:
                //////////////////////////////////////
                // 1. SIGN AND BROADCAST //////////////////////////////////
```



```

// TODO - Add simulation to check if msg is already
processed.
// If so, Send Ack to Queue. Avoid sending tx to
Routerchain

    txResponse, err :=
dispatcher.routerChainClient.SyncBroadcastMsg(routerMsg...)
    if err != nil {
        log.WithFields(log.Fields{"error": err}).Info("
Error while submitting tx to routerchain")
        time.Sleep(3 * time.Second)
        continue
    }
    log.WithFields(log.Fields{"txHash":
txResponse.TxResponse.TxHash, "Code": txResponse.TxResponse.Code}).Info("Submit
ted tx to routerchain successfully")
    }
}

```

Please note that ignoring the errors as in the fragment above is one specific instance of the general problematic error handling patterns in orchestrator and relayer, as outlined in the finding [Dangerous error handling in Orchestrator and Relayer](#).

Problem Scenarios

As outlined in the above (non-exhaustive) list, there are multiple possibilities, how event processing may fail in the orchestrator. The consequences are:

- In the first case above (panic in [FetchAndTransformInboundEvents\(\)](#)), orchestrator binary will shutdown, and will have to be restarted. **Event processing by this orchestrator will stop completely.**
- In the second case (ignored failure in [UpdateLastProcessedBlock\(\)](#)), the database will be in a corrupted state. **All further processing with a corrupted database may lead to unpredictable failures.**
- In the last case (ignored failures to submit Router transactions in [dispatcher.go:Start\(\)](#)), some of the events will be lost and never arrive at the Router chain, but processing in the orchestrator will continue. Due to the requirement of consecutiveness of event nonces in the [Attestation](#) Router module, **after the first missing event, all subsequent forwarded events will be ignored**. Notice that this error is not correctable without a manual intervention to decrease the value of [lastQueriedBlock](#) in the local database.

Recommendation

The problem with event processing in orchestrators is manifold:

- There is no proper error handling, although the event processing may fail at different stages. We recommend to examine the whole event processing code, and carefully treat all possible error scenarios.
- The processing of events from the source chain is too optimistic. The [lastQueriedBlock](#) is updated always, irrespective of whether the subsequent stages succeeded or failed, thus preventing the system from refetching the events that have not been properly submitted.
- The checking of the crosschain requests that have reached the Router chain is done once upon orchestrator start, in function [listener/gatewaylistener.go:Start\(\)](#), and is never redone afterwards. Thus, even in the situation when orchestrator could detect that some event has not reached the Router chain, this is not done, thus effectively stopping the orchestrator from doing any meaningful work.

We recommend to rework the principles of orchestrator event processing, correcting the above problems. As to the more concrete recommendations, we could outline the following:

- Wrt. the unrecoverable errors that arise *locally* (like a possible failure to write to the local database), never ignore such errors, but report them with whatever means possible, maximal error level, and shutdown the operation. A separate system that reacts to such errors needs to be in operation: e.g. a system that monitors whether the process is running, and notifies the operator whenever it shuts down.
- Wrt. the errors that arise *due to distributed communication*, we recommend to:
 - perform a (configurable) number of silent retries, while logging the errors;
 - notify the operator afterwards about the errors, but continue retrying (do not shutdown the process);
 - properly implement the [Sliding window protocol](#) wrt. transmission of events from the source chain to the Router chain. While the current implementation in the orchestrator code suggests an attempt to implement a version of this protocol, the actual implementation deficiencies outlined above prevent it from functioning as expected.

Status: Resolved

In the `FetchAndTransformInboundEvents()` function, a `gatewayError` is now returned instead of causing a panic. Additionally, the interaction with the database in `UpdateLastProcessedBlock` still lacks proper error handling. However, there have been improvements in the `dispatcher.Start()` function.

Using WHERE without ORDER BY in Orchestrator and Relay database queries may lead to out-of-order message passing

Title	Using WHERE without ORDER BY in Orchestrator and Relay database queries may lead to out-of-order message passing
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [Router Orchestrator at 5c23926f](#)
- [Router Relay at f4281dee](#)

Description

In multiple places in Router orchestrator and relay, a local database is employed to store messages to be passed to/from the Router chain or Gateway contracts. The typical example can be found in [router-relayer/processor/outbound_processor.go#L54-L77](#):

```
    for {
        txqArray, err :=
outboundProcessor.dbHandler.GetTxqByStatus(types.Unprocessed)
        if err != nil {
            panic(err)
        }
        for _, txq := range txqArray {
            err = outboundProcessor.dbHandler.UpdateTxqStatus(txq.Id,
types.Picked)

            if err != nil {
                panic(err)
            }
            switch txq.TxType {
            case types.ValsetUpdate:

outboundProcessor.ProcessAndBroadcastOutboundRequest(ctx, txq.OutboundRequest)
```

```

                case types.Crosschain:
                    outboundProcessor.ProcessCrosschainRequest(ctx,
txq.OutboundRequest)
                case types.CrosschainAck:
                    outboundProcessor.ProcessCrosschainAckRequest(ctx,
txq.OutboundRequest)
            }
            time.Sleep(2 * time.Second)
        }
    }

```

with `dbHandler.GetTxqByStatus()` being

```

func (dataHandler DbHandler) GetTxqByStatus(status types.Status) ([]TxqStore, error)
{
    txqStoreArray := []TxqStore{}
    result := dataHandler.db.Where(&TxqStore{Status: status}).Find(&txqStoreArray)
    return txqStoreArray, result.Error
}

```

It can be seen that the `GetTxqByStatus()` retrieves records from the local database using SQL `WHERE` clause, without employing the `ORDER BY` clause. While in many cases, especially with smaller datasets during testing, the records may come from the database `WHERE` query in an expected and predictable order, under heavier load, with a larger number of records, they may come in the unpredictable order, which is influenced by internal database algorithms. For a more in-depth explanation we recommend reading e.g. this blog post: [Why Ordering Isn't Guaranteed Without an ORDER BY](#).

All `.Where()` calls in the below files employ WHERE clause without ORDER BY clause.

- [Orchestrator's store/txqStore.go](#)
- [Relayer's store/txqStore.go](#)

Clearly not all of them require a particular ordering, but some definitely do, as is demonstrated below.

Problem Scenarios

In many cases it is supposed that messages arrive in a particular order: e.g. crosschain messages, or validator set updates. In case an orchestrator or a relayer happen to transfer messages out of order, it may lead e.g. to a message with a nonce larger than the expected one being rejected by a component down the stream. One particular instance of such problem seems to be in the orchestrator's function `SubscribeToMsgsFromQueue()`:

```

func (mqConsumer *MqConsumer) SubscribeToMsgsFromQueue() error {
    //////////////////////////////////////
    // Subscribe msgs from queue //
    //////////////////////////////////////
    for {
        var txqArray []store.TxqStore
        for i := uint64(0); i <= mqConsumer.batchWaitTime; i++ {
            txqArray, _ = mqConsumer.dbHandler.GetTxqByStatus(store.Unprocessed,
mqConsumer.batchSize)
            if uint64(len(txqArray)) < mqConsumer.batchSize {
                time.Sleep(time.Second)
                continue
            } else {
                break
            }
        }
    }
}

```

```

    }
}
var msgArray []sdk.Msg
var sdkMsg sdk.Msg
for _, msg := range txqArray {
    err := mqConsumer.dbHandler.UpdateTxqStatus(msg.Id, store.Picked)
    if err != nil {
        return err
    }
    err =
mqConsumer.routerChainClient.ClientContext().Codec.UnmarshalInterface(msg.MsgRequest,
&sdkMsg)
    if err != nil {
        log.WithFields(log.Fields{"error": err}).Error("Error while decoding
Message from MQ")
        return err
    }
    msgArray = append(msgArray, sdkMsg)
}
if len(msgArray) > 0 {
    fmt.Println("SqQueue:", "BatchSizeFromCfg", mqConsumer.batchSize,
"BatchSize", len(msgArray), "BatchWaitTime", mqConsumer.batchWaitTime)
    mqConsumer.routerMsgChannel <- msgArray
}
}
}

```

Notice that `dbHandler.GetTxqByStatus()` retrieves a set of records from the database, and uses WHERE clause without ORDER BY under the hood. The messages are then submitted into

`mqConsumer.routerMsgChannel`, and, eventually, sent to the Router chain. The processing sequence involving this function as one of the steps is the same as outlined already in the finding [Orchestrators may permanently fail to deliver crosschain messages](#), so we don't repeat it here. What's important to understand though is that down the stream, the Router chain `Attestation` module is invoked to attest the claims, and it makes sure that all submitted event nonces from the orchestrator are consecutive. If any message arrives out of order, that message will be dropped.

Recommendation

For all cases where ordering of messages is important, we recommend to complement the database employing WHERE clause with the ORDER BY clause, specifying the order that is expected due to the semantics of message ordering (e.g. by their nonces).

Status: Resolved

On the Orchestrator's side, `dbHandler.GetTxqByStatus` is utilizing `OrderBy`. To adhere to best practices for handling sensitive data, even in development and test environments, it is recommended to use environment variables or `.env` files for storing sensitive information. Avoid hardcoding sensitive details such as `ethPrivateKey`, `ethPassphrase`, `cosmosPrivateKey`, etc., directly in configuration files.

Excessive Relay queries may cause Router chain DOS

Title	Excessive Relay queries may cause Router chain DOS
Project	Router Protocol V2
Type	PROTOCOL IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	3 HIGH
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- Router relayer at f4281dee
- Router chain at f9ff8b44

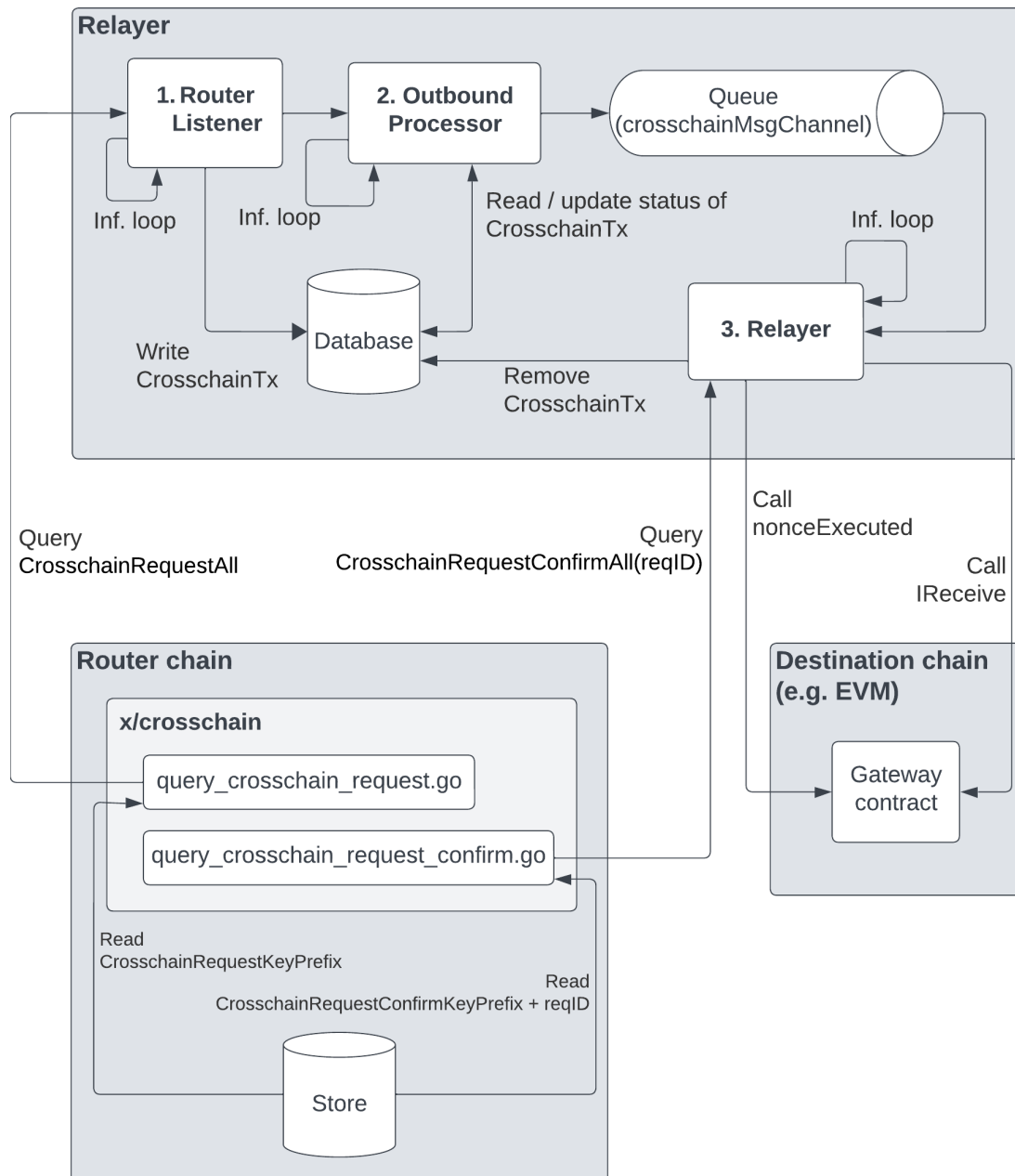
Description

In the current Router relayer and chain implementation, their interactions follow the same scheme for different kinds of messages, including crosschain requests, request acknowledgements, and validator set updates. We illustrate this scheme below on the example of crosschain requests.

- The processing starts in function `routerlistener.go:ListenAndProcessCrosschainReq()`, which in an endless loop:
 - Fetches crosschain requests from the Router chain in `FetchCrosschainRequests()`
 - This eventually comes to the Router chain as `CrosschainRequestAll`, and is processed in `query_crosschain_request.go:CrosschainRequestAll()`;
 - The above function retrieves from the chain storage all entries prefixed with `CrosschainRequestKeyPrefix`, and returns them to the querier.
 - Adds retrieved crosschain requests to the internal database for later processing via `AddtoCrosschainTxQueue()`, with the status `Unprocessed`
- The processing continues in `outbound_processor.go:Start()`, which, in an endless loop, fetches from the database all entries with the status `Unprocessed`, and distributes them to other functions according to their type. In the case of crosschain requests, `outbound_processor.go:ProcessCrosschainRequest()` is called, which eventually puts the request into Go's `crosschainMessageChannel`.
- The message is further processed in an endless loop of function `relayer.go:Start()`, which, in case of crosschain requests:
 - Retrieves the message from `crosschainMessageChannel`
 - Validates it in function `validation.go:ValidateCrosschainCall()`, which:

- i. In `validation.go:checkIfCrosschainNonceProcessed()` calls into the Gateway contract, and checks if the particular nonce has been already processed by it; if it is was, the validation fails.
 - ii. Via `validation.go:checkMajorityCrosschainConfirmations()`, eventually calls `CrosschainRequestConfirmAll` on the Router chain, which in `query_crosschain_request_confirm.go:CrosschainRequestConfirmAll()` collects all orchestrator confirmations for a particular crosschain request;
 - iii. if $2/3 + 1$ orchestrator votes are not collected by this crosschain request, `validation.go:checkMajorityCrosschainConfirmations()` removes the message from the internal database, and fails validation as a whole.
- c. If validation succeeds, the message is submitted for execution in the gateway contract on the destination chain via calling `gateway.go:IReceive()`.

The processing steps are also illustrated graphically below.



One of the main problems with the above processing is that it's hugely redundant and inefficient, in particular:

- In step 1, all crosschain requests are fetched. There are going to be many crosschain requests; **all existing crosschain requests will be fetched multiple times, by multiple relayers**. Notice that `FetchCrosschainRequests()` retrieves all entries from the Router chain store with the specific prefix each time it is executed. A crosschain request is going to live in the store for a long time before it is completely processed and subsequently purged.
- In step 3, for a particular crosschain request, **all request confirmations will be fetched multiple times, by multiple relayers**. Notice that in `validation.go:checkMajorityCrosschainConfirmations()` a particular crosschain request is going to be discarded if $< 2/3$ confirmations for it are received. This leads to the repetition of all processing steps multiple times until enough confirmations are received.

Problem Scenarios

The redundancy in processing crosschain requests, as well as other types of messages will become a problem as soon as the Router chain starts processing any non-trivial number of messages. Notice that in the above processing steps a multiplication of multiple scaling factors happens:

- number of messages of a particular type
- number of confirmations for each such message
- number of relayers
- number of orchestrators

This guarantees exponential growth of computational requirements for Router chain nodes, and will quickly lead to them being unable to either process queries, or to continue generating and validating new blocks.

Recommendation

We recommend to rethink and refactor the protocol and implementation of interactions between the Router chain and relayers. Below we outline some possible principles for such refactoring:

- Make Router chain generate specific `events` for cases when some message is ready to be relayed. For crosschain messages, in particular, this means accumulating all orchestrator confirmations for a message, and then issuing a designated event when $> 2/3$ of confirmations are received; let's call such event `CrosschainRequestEvent`; it may also contain all information needed to relay a particular crosschain message (such as the request content and all orchestrator signatures).
- Router relayer doesn't need to repeatedly query the chain first for crosschain requests, and then separately for confirmations for those requests, as well as there is no need to perform any expensive validation of requests from the relayer side.
- A relayer will need only to "listen" for occurrence of `CrosschainRequestEvent` on the Router chain, and relay the contained crosschain message (with $> 2/3$ signatures already), or any other type of message, to the destination chain.

We recommend studying how relaying is implemented e.g. by the [Hermes IBC relayer](#); the [Relayer specification](#) as well as [IBC packet handling](#) are also very relevant for possible refactoring of Router relayer protocol and implementation.

Status: Acknowledged

The Router team has acknowledged the status of the matter indicating that the relayer will be equipped with a distinct router read node.

The initially Critical finding regarding the potential for a DOS attack on the Router chain due to excessive relayer queries has been reassessed and marked as Informational by the Router team.

Their explanation for this reassessment is that the relayers in the system are designed to interact with public sentry nodes rather than the validator nodes directly. Sentry nodes act as a protective layer for validator nodes, handling incoming traffic and queries. This architecture ensures that even if the relayers generate a high volume of queries, these queries are directed at the sentry nodes and not at the validator nodes themselves.

This setup effectively shields the validator nodes from being overwhelmed by such queries, addressing the initial critical concern.

Loss of crosschain messages/acks due to validator set updates

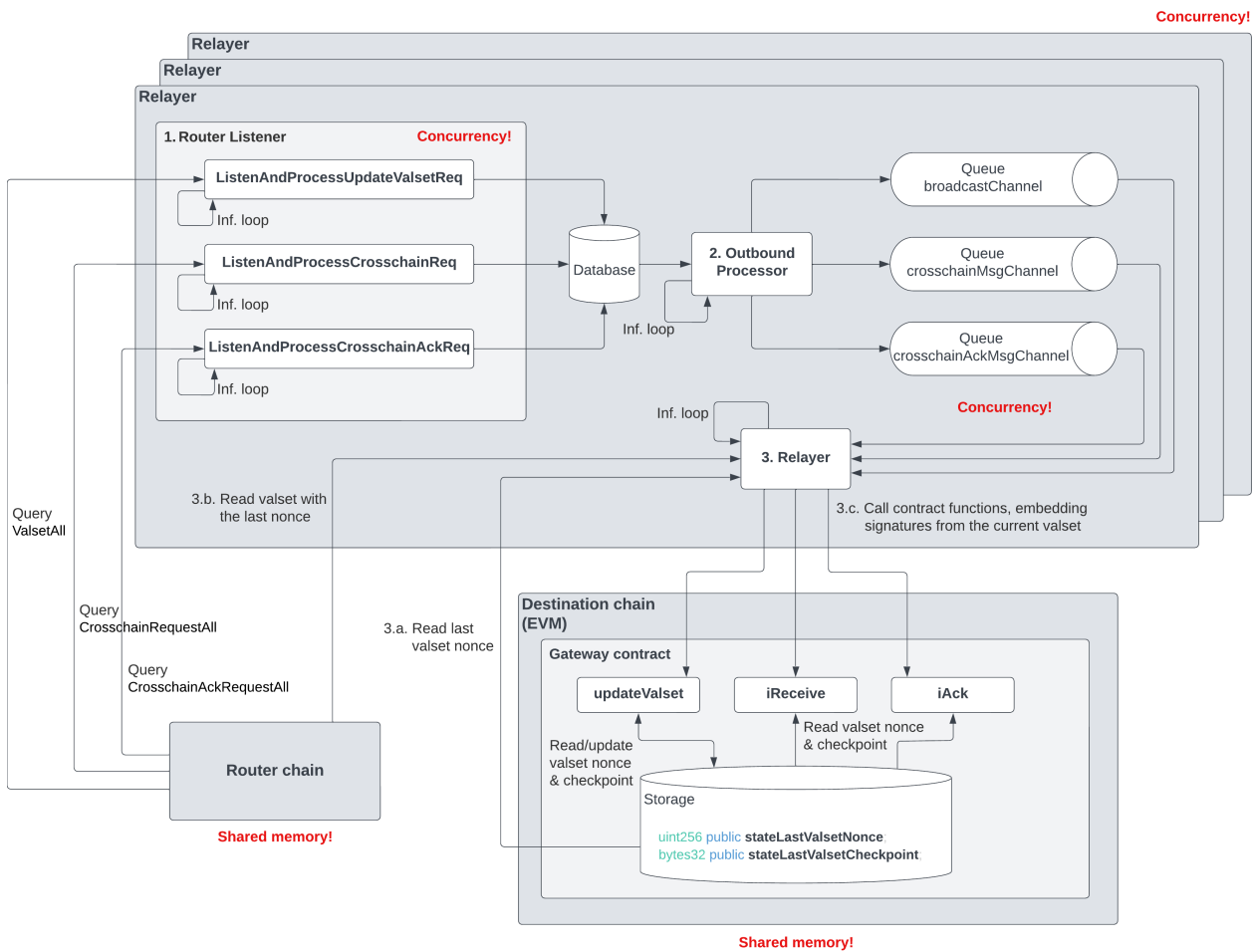
Title	Loss of crosschain messages/acks due to validator set updates
Project	Router Protocol V2
Type	PROTOCOL IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [Router EVM Gateway contract at 4cbae6ee](#) (informal-audit branch)
- [Router relayer at f4281dee](#)
- [Router chain at f9ff8b44](#)

Description

The present Router protocol and implementation represents a hugely concurrent arrangement of orchestrators and relayers, and with Router chain and other chains (e.g. EVM) acting as a shared memory. Below we illustrate one scenario where this leads to problems; but other scenarios are definitely possible.



In the scenario, illustrated graphically above, the following is happening:

1. The processing starts in function `routerlistener.go:Start()`, which spawns 3 **concurrent** goroutines, for processing validator set updates, crosschain requests, and crosschain acknowledgements. Each of the goroutines, in an infinite loop, requests corresponding data from the Router chain, and commits the transactions of a respective type into the local database.
2. The processing continues in `outbound_processor.go:Start()`, which, in an endless loop, fetches from the database all entries with the status `Unprocessed`, and distributes them to other functions according to their type. These functions put the messages into three separate queues, namely `broadcastChannel` for validator set updates, and `crosschainMsgChannel` and `crosschainAckMsgChannel` for crosschain requests and acknowledgments respectively.
3. The message is further processed in an endless loop of function `relayer.go:Start()`, which performs Golang's `select` statement to retrieve the messages from one of the channels. Notice that according to the [Golang spec](#), "If one or more of the communications can proceed, a single one that can proceed is chosen via a uniform pseudo-random selection"; thus, this operation is also **concurrent**. Besides that, while performing validation and transformation of the messages, this function performs accesses to remote components, namely the Router chain and other chains; in our case it's EVM. Notice that wrt. the the whole Router system these chains perform the role of a **shared memory** wrt. multiple concurrent components accessing them (orchestrators and relayers). Focusing on the concurrency aspects, the following is happening:
 - a. Relayer **calls into the Gateway contract** to determine the nonce of the last validator set stored in the contract.
 - b. Relayer **calls into the Router chain** to retrieve the validator set corresponding to the last valset nonce from the Gateway contract.

- c. Relay [filters the message orchestrator signatures](#) according to the validator set retrieved from the Router chain, [embeds these signatures into the parameters](#), and [calls the corresponding Gateway function](#) (`updateValset` , `iReceive` , `iAck`) with the constructed parameters.
4. Finally, the Gateway contract functions are executed on EVM, and they validate that the call parameters correspond to the last stored validator set, [here](#) and [here](#), like this (where `_currentValset` is one of the call parameters):

```
// Check that the supplied current validator set matches the saved checkpoint
if (makeCheckpoint(_currentValset) != stateLastValsetCheckpoint) {
    revert Utils.IncorrectCheckpoint();
}

// Check that enough current validators have signed off on the new validator set
bytes32 newCheckpoint = makeCheckpoint(_newValset);
bytes32 digest = _make_digest(newCheckpoint);
SignatureUtils.checkValidatorSignatures(_currentValset, _sigs, digest,
Utils.CONSTANT_POWER_THRESHOLD);
```

Notice that besides the concurrency happening *in a single relay*, as outlined above, there are also **multiple concurrent relayers** operating at any given point in time.

Problem Scenarios

There are multiple problems in the scenario outlined above:

- Gateway contracts track a single validator set, that they deem the only source of truth from the time of the last update, up to the next one. But messages may arrive, which have been signed by one of the previous validator sets, or one of the future ones (if the chain in question lags behind), and these messages are perfectly valid
 - In the scenario above, **the messages that do not correspond to a single stored validator set, will be rejected by the Gateway contract.**
- Router components (relayers in this scenario) perform multiple concurrent accesses to other remote components in the same function. Due to the nature of distributed systems (e.g. messages in flight, or message delays), the information a relay acts upon will never be valid: one or other parts of it will be outdated.
 - In this particular scenario, between the time point 3.a (when relay asks the Gateway contract for the last valset nonce), and the time point 4 (when the Gateway contract checks the call validity), **the validator set stored in the Gateway may be updated due to concurrent calls from the same relay, or from other relayers**, leading to the subsequent rejection of the Gateway call, as outlined above.

Recommendation

- To properly act on the messages received from other chains, the blockchain in question should track not a single valid state, but a set of valid states of other blockchains. We recommend reworking both the Router protocols and their implementation to properly account for the evolving nature of its constituent components (blockchains). A particular approach we may recommend to employ is the [Light client paradigm](#) for bridging messages between independent blockchains, as implemented for example in the [IBC light clients](#).
- We further recommend reworking the Router protocols and their implementation to avoid performing concurrent accesses to evolving remote components. The only valid way for a component such as relay to act is to base its decisions on the locally available information only (i.e. the state of the local memory, and the content of messages received so far).

Multiple protocol-level issues inherited from Gravity Bridge design

Title	Multiple protocol-level issues inherited from Gravity Bridge design
Project	Router Protocol V2
Type	PROTOCOL IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	3 HIGH
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [Router chain at f9ff8b44](#)
- [Router EVM Gateway contract at 4cbae6ee \(informal-audit branch\)](#)
- [Router Relayer at f4281dee](#)
- [Router Orchestrator at 5c23926f](#)

Description

During our investigation of the Router codebase, we could not help but notice that substantial parts of Router design have been influenced by the [Gravity Bridge project by Althea](#) (also [here](#)), which we have audited around 2 years ago. Some code has also migrated from that project, in particular Router's [attestation module](#) has substantial overlaps with the Gravity Bridge's [gravity module](#), or Router's [Solidity contracts](#) have substantial overlap with Gravity Bridge's [Solidity contracts](#).

But even more important than code similarities are similarities in protocols and architecture. Due to these, we would like to point out to the [protocol-level Gravity Bridge issues](#) we've discovered during its audit. Out of those many are directly applicable to Router due to the design similarities, in particular:

- [No Independent Validation in Ethereum](#)
- [No Trusting Period Check in Ethereum](#)
- [No Independent Validation in Cosmos](#)
- [Unclear Rules to Issue Validator Set Updates](#)

Recommendation

We recommend to thoroughly examine the above listed issues of the Gravity Bridge audit, and refactor Router's protocols, design, and implementation, to address these protocol-level issues.

Status: Acknowledged

The finding has been acknowledged and status changed to Informational after further clarification from the Router team. The Router team's response specifically addresses the concerns raised about the similarities in design and protocols between the Router and the Gravity Bridge project.

Regarding the specific issues of "No Independent Validation in Ethereum" and "No Trusting Period Check in Ethereum," the Router team clarified that these aspects do not apply to their Gateway contracts on the EVM side. Also, regarding "Unclear Rules to Issue Validator Set Updates", all changes to the validator set are initiated from the Router chain, which is considered the definitive source of truth in this architecture. Consequently, there are no additional checks on the EVM side regarding independent validation or trusting periods, as any data or instructions emanating from the Router chain are deemed trustworthy and safe.

Unstable chain operation due to gas consumption and panic handling in ExecuteInboundRequest function

Title	Unstable chain operation due to gas consumption and panic handling in ExecuteInboundRequest function
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	3 HIGH
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [x/crosschain/keeper/inbound_execution.go](https://github.com/crosschain/keeper/inbound_execution.go)

Description

The provided [code snippet](#) is part of a function named `ExecuteInboundRequest`.

```
defer func() {  
    // Refund fees  
    gasConsumed := cacheCtx.GasMeter().GasConsumed()  
    gasUsed := big.NewInt(0).SetUint64(gasConsumed)  
    gasPrice := big.NewInt(0).SetUint64(params.InboundGasPrice)  
    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,  
    "feeConsumedInRoute", 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
begin blocker")
}()

```

This function is responsible for executing an inbound cross-chain request by decoding a packet, fetching contract-related metadata, and executing a Cosmwasm bridge contract on a chain. One problematic aspect of this function is the placement of `ctx.GasMeter().ConsumeGas(gasConsumed, "consume gas for contract execution in begin blocker")` within a `defer` statement. This could lead to unintended consequences if a panic occurs within the function.

Problem Scenarios

The problematic part of the code is the line `ctx.GasMeter().ConsumeGas(gasConsumed, "consume gas for contract execution in begin blocker")`, which is placed inside a `defer` function that is intended to catch panics. However, if this line triggers a panic, it can cause a chain halt due to out-of-gas issues. This situation could arise under the following scenarios:

1. **Large DestGasLimit Value:** If the `DestGasLimit` field is set to an excessively large `uint64` value, it could result in excessive gas consumption during contract execution, causing the gas meter to exceed its limit and trigger an `ErrorOutOfGas` or an `ErrorGasOverflow` panic.
2. **Multiple Crosschain Requests:** If there are multiple cross-chain requests executing the same contract within the same block, the `ConsumeGas` calls could accumulate enough gas consumption to trigger either an `ErrorOutOfGas` or an `ErrorGasOverflow` panic.

Recommendation

1. **Prevent Excessive Gas Limit:** Implement validation checks on the `DestGasLimit` field to prevent excessively large values. Ensure that the gas limit is set to a reasonable value that corresponds to the anticipated gas consumption during contract execution.
2. **Gas Estimation:** Implement gas estimation mechanisms to accurately predict the gas consumption of contract execution. This approach will help prevent unexpected out-of-gas scenarios, ensuring that the provided gas limit is adequate for the execution.
3. **Panics Handling:** Consider adding a mechanism to catch panics from the `ctx.GasMeter().ConsumeGas` call. You can use the `recover` keyword to catch the panic, log the relevant information, and gracefully handle the situation. This will prevent panics from interrupting the normal execution flow and contribute to a more stable chain operation.
4. **Thorough Testing:** Rigorously test the `ExecuteInboundRequest` function across various gas limit values, contract execution scenarios, and cross-chain request volumes. This testing will identify potential vulnerabilities and verify that gas consumption remains within acceptable limits.

Status: Resolved

The approach involving `gasRemaining` appears to be satisfactory.

Security concern: IBC permissionless interaction

Title	Security concern: IBC permissionless interaction
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [x/pricefeed/module_ibc.go](#)

Description

The provided [code snippet](#) is part of the `OnRecvPacket` function implementation in the context of the IBC module. This function is responsible for processing incoming packets from other chains and handling Oracle response data that is sent to the `pricefeed` module. The code extracts and validates the Oracle response packet data, and then stores it. However, there are potential security concerns related to the IBC permissionless property.

```
func (im IBCModule) OnRecvPacket(
    ctx sdk.Context,
    modulePacket channeltypes.Packet,
    relayer sdk.AccAddress,
) ibcexported.Acknowledgement {
    var packet bandtypes.OracleResponsePacketData

    // Unmarshal the data from the module packet into the OracleResponsePacketData
    // object.
    if err := types.ModuleCdc.UnmarshalJSON(modulePacket.GetData(), &packet); err !=
nil {
        return channeltypes.NewErrorAcknowledgement(err)
    }

    // Request has been resolved and relayed to pricefeed module
    // ctx.EventManager().EmitEvent(sdk.NewEvent(
    //     types.EventTypeResponsePacket,
```

```
// sdk.NewAttribute(types.AttributeKeyRequestID, fmt.Sprintf("%d",
packet.RequestID)),
// ))

if packet.ResolveStatus != bandtypes.RESOLVE_STATUS_SUCCESS {
    return channeltypes.NewErrorAcknowledgement(types.ErrResolveStatusNotSuccess)
}

if err := im.keeper.StoreOracleResponsePacket(ctx, packet); err != nil {
    return channeltypes.NewErrorAcknowledgement(err)
}

return channeltypes.NewResultAcknowledgement(nil)
}
```

Problem Scenarios

The IBC protocol is designed to be permissionless, allowing any two IBC-enabled chains to establish connections and create channels for communication. This means that both legitimate (e.g. [Bond Protocol](#)) and potentially malicious actors, including other chains or solo machines, can initiate connections and establish channels to interact with the `pricefeed` module.

In the provided code, there is no explicit verification of the source channel, assuming the counterparty chain is trusted.

Recommendation

To address the potential security risks associated with the IBC permissionless property, it's important to implement additional safeguards and checks within the `OnRecvPacket` function. One solution could be to maintain Whitelist (a list of trusted channels that are allowed to interact with the `pricefeed` module).

This could involve maintaining a whitelist of channel identifiers or IDs that are known to be legitimate and safe. Only packets originating from these trusted channels should be processed.

Status: Resolved

`CheckValidPath` has been updated in comparison to the commit hash mentioned in the comment. The whitelist approach now accommodates both scenarios, whether the Router is the IBC sender (`OnAck`) or receiver (`OnReceive`).

Non-deterministic iteration of attmap in ClaimEventSlashing function

Title	Non-deterministic iteration of attmap in ClaimEventSlashing function
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [x/attestation/keeper/attestation.go](#)
- [x/attestation/abci.go](#)

Description

In the `ClaimEventSlashing` function, the attestation mapping (`attmap`) retrieved from the `GetAttestationMappingByChainAndContract` function is iterated without enforcing any specific order.

```
func ClaimEventSlashing(ctx sdk.Context, k keeper.Keeper, chainId string, contract
string, claimsSlashWindow uint64, slashFractionConflictingClaim sdk.Dec,
slashFractionMissingClaim sdk.Dec) {
    // #3 condition
    // Oracle events MsgDepositClaim, MsgWithdrawClaim

    attmap, _ := k.GetAttestationMappingByChainAndContract(ctx, chainId, contract)
    for _, attestations := range attmap {

        currentBondedSet := k.StakingKeeper.GetBondedValidatorsByPower(ctx)

        .....
    }
```

This could lead to non-deterministic behavior in the code, as the order of iteration over the map may vary between different executions of the function. The [comment](#) above the `GetAttestationMappingByChainAndContract` function suggests that the map should be iterated using ordered keys to ensure deterministic behavior.

Problem Scenarios

Imagine a scenario where the `attmap` contains attestations with conflicting votes for the same event nonce. Since the iteration order is not deterministic, the function might behave differently based on the order in which the attestations are iterated. This could result in uneven slashing of validators and inconsistency in how conflicting claims are handled.

Recommendation

To ensure deterministic behavior and prevent potential issues related to non-determinism, it is recommended to iterate over the `attmap` using the ordered keys provided by the

`GetAttestationMappingByChainAndContract` function, as demonstrated in the `tallyChainAttestations` function:

```
attmap, keys := k.GetAttestationMappingByChainAndContract(ctx, chainId, contract)

for _, nonce := range keys {
    for _, att := range attmap[nonce] {
        // Process each attestation following the sorted order
    }
}
```

Status: Resolved

The second returned value of `GetAttestationMappingByChainAndContract` (`keys`) is now utilized, addressing the non-determinism issue.

Outdated and deprecated versions of OpenZeppelin contracts used

Title	Outdated and deprecated versions of OpenZeppelin contracts used
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [router-gateway-contracts/evm at 4cbae6ee \(informal-audit branch\)](#)

Description

Both `@openzeppelin/contracts` and `@openzeppelin/contracts-upgradeable` are fixed to `v.4.8.0`, which is at the time of writing `10 months old`, from November 2022.

Moreover, in the `AssetVault` contract, OpenZeppelin's `ERC20PresetMinterPauser` is used for the route token, which has been deprecated since the release `4.5.0` (from February 2022) in favor of the `Contracts Wizard`, and will be completely removed in the upcoming release `v.5.0`.

Problem Scenarios

Using significantly outdated or even deprecated versions of the dependences poses security risks that grow as time passes: the old code doesn't receive proper attention anymore, and even known security vulnerabilities may not be fixed for longer time periods. Besides, when the new release comes and the deprecated contracts are removed, no security guarantees can be provided.

Recommendation

We recommend to:

- upgrade the latest release versions of OpenZeppelin contracts (`v.4.9.2` from `June 2023` at the time of writing)
- Replace the deprecated `ERC20PresetMinterPauser` contract with the one generated by `Contracts Wizard` using the "Mineable" and "Pausable" features.

Status: Resolved

Issue is resolved as recommended.

Outdated version of Solidity compiler is accepted

Title	Outdated version of Solidity compiler is accepted
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	2 MEDIUM
Impact	2 MEDIUM
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [router-gateway-contracts/evm at 4cbae6ee \(informal-audit branch\)](#)

Description

All Router EVM contracts, e.g. [GatewayUpgradeable.sol](#) fix the Solidity compiler version to

```
pragma solidity >=0.8.0 <0.9.0;
```

Version [v.0.8.0](#) is already almost 3 years old; having such statement in the contracts leaves open the possibility to (accidentally) compile the contracts with a very old version of the Solidity compiler.

Problem Scenarios

Using an old version of the Solidity compiler may be problematic due to various reasons:

- Unfixed security issues
- Absence of support for more modern language features
- Absence of many optimizations, and as a result suboptimal bytecode generated from the contracts.

Recommendation

We recommend to update the contract requirements wrt. to the Solidity compiler to use one of the most recent versions (the latest release is [v.0.8.21](#) from July 2023).

Status: Resolved

Issue is resolved as recommended.

Deficient decoding of contract metadata: lack of validation, error handling, and default values

Title	Deficient decoding of contract metadata: lack of validation, error handling, and default values
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [x/crosschain/types/metadata.go](https://github.com/crosschain/types/metadata.go)

Description

The [provided code](#) presents the `DecodeEvmContractMetadata` function, which aims to decode and extract metadata from a given contract message implementing the `IContractMetadata` interface. The function converts hexadecimal encoded metadata strings into various data types such as `uint64`, `*big.Int`, `uint8`, `bool`, and bytes, ultimately constructing an instance of the `EvmContractMetadata` struct.

Problem Scenarios

1. **Lack of Validation:** The code does not validate the incoming `requestMetadataStr` before attempting to decode it. If the provided `requestMetadataStr` does not adhere to the expected format, it can lead to issues during decoding and potential errors, especially in scenarios where the decoding operations expect specific lengths or data types.
2. **Error Handling Missing:** The code does not include error handling for potential failures during decoding operations, such as those related to hexadecimal decoding or conversion of data types. In the absence of proper error handling, unexpected data or decoding failures can result in undefined behavior and runtime errors.
3. **Missing Default Values:** If the decoding process fails or if certain fields are not provided after decoding, the function does not provide any fallback or default values. This can lead to inconsistencies in the metadata structure and create difficulties in downstream processes that rely on consistent metadata.

Recommendation

To improve the robustness and reliability of the `DecodeEvmContractMetadata` function, consider the following recommendations:

1. **Validate** `requestMetadataStr` : Before proceeding with decoding operations, validate the length and format of the `requestMetadataStr` . Ensure that it adheres to the expected length and data type requirements before attempting any decoding.
2. **Implement Error Handling**: Introduce error handling mechanisms for decoding operations that can potentially fail, such as hexadecimal decoding or data type conversions. Handle these errors gracefully by returning appropriate errors or fallback values.
3. **Default Values**: In case decoding fails or certain fields are missing, provide default values for the `EvmContractMetadata` struct fields. This will ensure a consistent metadata structure even in scenarios where decoding or field extraction encounters issues.

Optimized Implementation:

```
func DecodeEvmContractMetadata(msg IContractMetadata) *EvmContractMetadata {
    metadata := EvmContractMetadata{}
    requestMetadataStr := hex.EncodeToString(msg.GetRequestMetadata())

    if len(requestMetadataStr) < 100 {
        return &metadata
    }

    // ... Existing code for slicing and decoding

    // Implement validation checks, error handling, and default values
    // before performing decoding operations.

    // Validate requestMetadataStr length and format
    if len(requestMetadataStr) != 200 {
        // Return error or handle appropriately
        return &metadata
    }

    // ... Implement additional validation checks and error handling

    // ... Existing code for decoding

    return &metadata
}
```

Status: Resolved

The code has been enhanced by incorporating error handling and incrementally constructing the 'metadata' struct.

Potential input issue: uninitialized price.PriceFeeder argument

Title	Potential input issue: uninitialized price.PriceFeeder argument
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [x/pricefeed/keeper/keeper.go](#)
- [x/pricefeed/keeper/price.go](#)

Description

The provided [code snippets](#) involve the `UpdatePrice` function within the context of a `pricefeed` module. This function aims to update a price based on new data.

```
func (k Keeper) UpdatePrice(ctx sdk.Context, price types.Price) bool {
    old, found := k.GetPrice(ctx, price.PriceFeeder, price.Symbol)
    if !found || old.ResolveTime < price.ResolveTime {
        k.SetPrice(ctx, price)
        return true
    }
    return false
}
```

The `UpdatePrice` function is called [within a loop](#) over a list of results. It seems that the input argument `price.PriceFeeder` might not be properly initialized when invoking the `UpdatePrice` function, potentially leading to issues.

```
changed := k.UpdatePrice(ctx, types.Price{
    Symbol:      r.Symbol,
    Price:       math.NewIntFromUint64(r.Rate),
    ResolveTime: res.ResolveTime,
})
```

Problem Scenarios

The `UpdatePrice` function relies on the `price.PriceFeeder` argument to determine whether to update the price. If the `price.PriceFeeder` is not initialized or is an empty string, it could lead to unexpected behavior. Specifically, if `price.PriceFeeder` is not provided or is an empty string, it might result in incorrect or unintended price updates or retrievals.

Recommendation

To address the potential issues related to the uninitialized or empty `price.PriceFeeder` argument, consider the following steps:

1. **Input Validation:** Before calling the `UpdatePrice` function, ensure that the `price.PriceFeeder` argument is properly initialized and contains valid data. This can help prevent unexpected behavior due to uninitialized or empty values.
2. **Guard Clause:** Within the `UpdatePrice` function, add a guard clause to handle cases where the `price.PriceFeeder` argument is empty or uninitialized. If the `price.PriceFeeder` is not valid, you might choose to return an error or take appropriate action based on the context.
3. **Documentation:** Clearly document the expected behavior and requirements for input arguments like `price.PriceFeeder` in the relevant parts of your codebase. This will help other developers understand the proper usage and initialization of these variables.

Status: Resolved

`PriceFeeder` is initialized with `BAND_PRICE_FEEDER`.

Execution order inconsistency between pricefeed module and crosschain module endblockers

Title	Execution order inconsistency between pricefeed module and crosschain module endblockers
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Status	RESOLVED
Issue	

Involved artifacts

- [app/app.go](#)

Description

During the review, an inconsistency was identified in the execution order of endblockers between the Pricefeed module and the Crosschain module. The Crosschain module's endblocker is currently set to execute **before** the Pricefeed module's endblocker.

However, it has been observed that the Crosschain module actively utilizes functionalities from the Pricefeed module during its execution, such as `GetGasPriceState`, `GetTokenPriceState`, and `ConvertNativeTokenFeeToRouter`. This ordering mismatch could lead to unintended consequences and inconsistencies in the module's behavior.

Problem Scenarios

The current execution order places the Crosschain module's endblocker execution before the Pricefeed module's endblocker. This order is problematic because the Crosschain module relies on data and functionalities from the Pricefeed module. As a result, when the Crosschain module executes, it depends on information from the Pricefeed module that may not yet be updated due to the Pricefeed module's endblocker execution being scheduled later.

Recommendation

To ensure consistency and accurate functionality, it is recommended to adjust the execution order of endblockers as follows:

Execute Pricefeed Endblocker Before Crosschain Endblocker: Change the execution order so that the Pricefeed module's endblocker (`SetOrderEndBlockers : pricefeedmoduletypes.ModuleName`) is executed before the Crosschain module's endblocker (`SetOrderEndBlockers : crosschainmoduletypes.ModuleName`).

Status: Resolved

`pricefeedmoduletypes.ModuleName` is executed prior to `crosschainmoduletypes.ModuleName` .

Efficiency and conditional sequence concerns in createValsets function: suboptimal ordering of conditions

Title	Efficiency and conditional sequence concerns in createValsets function: suboptimal ordering of conditions
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- x/attestation/abci.go

Description

The provided [code snippet](#) is a function named `createValsets`, which is responsible for automating the creation of validator set requests. The function utilizes a combination of checks to determine whether a new validator set request should be generated and submitted to Ethereum. Of particular interest is the section:

```
if (latestValset == nil) || (lastUnbondingHeight == uint64(ctx.BlockHeight())) ||
significantPowerDiff {
    // if the conditions are true, put in a new validator set request to be signed
    and submitted to Ethereum
    k.SetValsetRequest(ctx)
}
```

The logic of the conditional block revolves around three conditions, with the intent to initiate the `SetValsetRequest` function if any one of the conditions evaluates to true. The sequence of these conditions and their potential impacts on the execution flow raises considerations for optimizing the efficiency and accuracy of the `createValsets` function.

Problem Scenarios

Early Termination Opportunity: If the condition `(latestValset == nil) || (lastUnbondingHeight == uint64(ctx.BlockHeight()))` evaluates to true, the function immediately calls `SetValsetRequest` and doesn't proceed further. However, if `significantPowerDiff` is checked afterward, it might involve an unnecessary, potentially [costly calculation](#), impacting performance:

```
func (b InternalBridgeValidators) PowerDiff(c InternalBridgeValidators) float64 {
    powers := map[string]int64{}
    // loop over b and initialize the map with their powers
    for _, bv := range b {
        powers[bv.EthereumAddress.GetAddress().Hex()] = int64(bv.Power)
    }

    // subtract c powers from powers in the map, initializing
    // uninitialized keys with negative numbers
    for _, bv := range c {
        if val, ok := powers[bv.EthereumAddress.GetAddress().Hex()]; ok {
            powers[bv.EthereumAddress.GetAddress().Hex()] = val - int64(bv.Power)
        } else {
            powers[bv.EthereumAddress.GetAddress().Hex()] = -int64(bv.Power)
        }
    }

    var delta float64
    for _, v := range powers {
        // NOTE: we care about the absolute value of the changes
        delta += math.Abs(float64(v))
    }

    return math.Abs(delta / float64(math.MaxUint32))
}
```

Recommendation

To enhance the efficiency and accuracy of the `createValsets` function, consider the following sequencing and optimizations:

1. **Sequential Conditional Check:** Arrange the conditional checks in sequential order according to their priority and potential impact. Since `(latestValset == nil) || (lastUnbondingHeight == uint64(ctx.BlockHeight()))` has a definitive outcome and doesn't involve costly calculations, it should be evaluated first. If it evaluates to true, immediately call `SetValsetRequest`.
2. **Avoid Unnecessary Calculation:** If `(latestValset == nil) || (lastUnbondingHeight == uint64(ctx.BlockHeight()))` is false, proceed to the `significantPowerDiff` calculation only if necessary. This optimization prevents performing a potentially expensive calculation when its outcome won't affect the decision to call `SetValsetRequest`.

Optimized Sequence and Implementation:

```
if (latestValset == nil) || (lastUnbondingHeight == uint64(ctx.BlockHeight())) {
```



```
    // Immediate call if conditions are met
    k.SetValsetRequest(ctx)
} else if significantPowerDiff {
    // Proceed to power difference calculation and call if needed
    k.SetValsetRequest(ctx)
}
```

By structuring the conditions in this optimized sequence, you ensure that the function proceeds efficiently and avoids unnecessary calculations when the outcome has already been determined by earlier conditions. This approach helps in achieving accurate and efficient validator set request creation while maintaining code readability and performance.

Status: Resolved

Issue is resolved as recommended.

Inefficient function call and redundant validation in tallyChainAttestations function: optimizing attestation processing

Title	Inefficient function call and redundant validation in tallyChainAttestations function: optimizing attestation processing
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [x/attestation/abci.go](#)

Description

The provided [code snippet](#) corresponds to a function named `tallyChainAttestations`, which is designed to tally and process attestations for a specific chain and contract. This function iterates through attestation event nonces and checks whether the attestation should be processed based on certain conditions. The focus lies on the following segment:

```
if nonce == uint64(k.MultichainKeeper.GetLastObservedEventNonce(ctx, chainId,
contract))+1 {
    k.TryAttestation(ctx, att)
}
```

This segment is located within a nested double for loop. Additionally, within the `TryAttestation` function, there are checks related to the observed event nonce:

```
lastEventNonce := k.MultichainKeeper.GetLastObservedEventNonce(ctx,
claim.GetChainId(), claim.GetContract())
if claim.GetEventNonce() != lastEventNonce+1 {
    panic("attempting to apply events to state out of order")
}
```

It is worth noting that the current implementation invokes `GetLastObservedEventNonce` multiple times within the double for loop, which might introduce unnecessary computational overhead. Moreover, the validity of the last event nonce is checked again within the `TryAttestation` function, even though it is already validated at a higher level.

Problem Scenarios

1. **Redundant Function Calls:** The repeated invocation of `GetLastObservedEventNonce` for each attestation within the double for loop could lead to redundant function calls, which may affect the performance of the `tallyChainAttestations` function.
2. **Redundant Validation:** The revalidation of the event nonce validity within the `TryAttestation` function adds an extra layer of redundancy, as the calling logic ensures that only valid event nonces are processed.

Recommendation

To optimize the `tallyChainAttestations` function and eliminate unnecessary function calls and validations, consider the following recommendations:

1. **Local Variable for Last Observed Event Nonce and Handling Updates:** Before entering the double for loop, retrieve the value of `GetLastObservedEventNonce` once and store it in a local variable. Use this local variable within the loop to avoid multiple calls to the same function. Since `TryAttestation` can change `LastObservedEventNonce` only if there is a $2/3 + 1$ voting power majority, you can handle this by returning a value from `TryAttestation` to inform the outer block of code to update the local `LastObservedEventNonce` if needed.

Optimized Implementation:

```
lastObservedNonce := k.MultichainKeeper.GetLastObservedEventNonce(ctx, chainId,
contract)
for _, nonce := range keys {
    for _, att := range attmap[nonce] {
        if nonce == lastObservedNonce+1 {
            updated := k.TryAttestation(ctx, att)
            if updated {
                lastObservedNonce++ // Increment local variable
            }
        }
    }
}
```

2. **Optimize Validation:** Since the higher-level logic already validates the validity of event nonces before calling `TryAttestation`, you can omit the redundant validation within the `TryAttestation` function.

Status: Resolved

Efficiency is improved by reducing the number of calls to

`MultichainKeeper.LastObservedEventNonce` and by minimizing the processing of non-relevant nonces.

Redundant recalculation of total validator power in TryAttestation function

Title	Redundant recalculation of total validator power in TryAttestation function
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- x/attestation/keeper/attestation.go

Description

The `TryAttestation` function, as provided in the [code snippet](#), is responsible for processing attestations based on specific conditions, including the calculation of `totalPower` and `requiredPower` within a double for loop. The focus is on the lines:

```
totalPower := k.StakingKeeper.GetLastTotalPower(ctx)
requiredPower :=
types.AttestationVotesPowerThreshold.Mul(totalPower).Quo(sdk.NewInt(100))
```

It's important to note that the calculation of `totalPower` remains constant across the iterations, as it is independent of the loop variables. However, in the current implementation, these calculations are performed repeatedly for each attestation within the loop, which could lead to unnecessary computation and decreased efficiency.

Problem Scenarios

Redundant Recalculation: The recalculation of `totalPower` and `requiredPower` for every attestation within the loop is redundant, as these values remain constant throughout the loop iterations. This repetition of calculations can result in unnecessary computational overhead and decreased performance.

Recommendation

To optimize the `TryAttestation` function and avoid redundant calculations of `totalPower` and `requiredPower`, consider the following recommendation:

Move Calculation Out of the Loop: Since the values of `totalPower` and `requiredPower` remain unchanged within the loop, calculate them once outside the loop and pass them as parameters to the `TryAttestation` function. This optimization eliminates the need for repeated calculations and contributes to improved efficiency.

Optimized Implementation:

```
totalPower := k.StakingKeeper.GetLastTotalPower(ctx)
requiredPower :=
types.AttestationVotesPowerThreshold.Mul(totalPower).Quo(sdk.NewInt(100))

// ... (loop initialization)

for _, nonce := range keys {
    for _, att := range attmap[nonce] {
        if nonce == lastObservedNonce+1 {
            updated := k.TryAttestation(ctx, att, totalPower, requiredPower) // Pass
totalPower and requiredPower as parameters
            if updated {
                lastObservedNonce++
            }
        }
    }
}
```

Status: Resolved

The `requiredPower` is computed outside the loop, employing the "multiplication before division" pattern.

Redundant claimHash calculation in cross-chain request processing

Title	Redundant claimHash calculation in cross-chain request processing
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [x/crosschain/abci.go](#)

Description

The provided [code segment](#) illustrates the `HandleFees` function responsible for handling various types of cross-chain requests by deducting fees and setting corresponding statuses. Within this function, the calculation of `claimHash` using the `ClaimHash` method is performed multiple times for different branches of the switch statement, even though changes within the `crosschainRequest` instance do not affect the `claimHash` value.

Additionally, the same redundancy issue is found in the `ProcessBlockedCrosschainRequests` function.

Problem Scenarios

Redundant `ClaimHash` Calculation: The `claimHash` is recalculated multiple times within the same function based on the same static fields of the `CrosschainRequest`. This redundancy can lead to unnecessary computational overhead and reduced performance.

Recommendation

To address the issue of redundant `claimHash` calculations, consider the following recommendation:

Calculate `claimHash` Once: Calculate the `claimHash` value once for each branch of the switch statement where it is required, and store the calculated value in a local variable. Reuse this local variable for subsequent uses of `claimHash` within the same branch, rather than recalculating it.

Optimized Implementation:

```
for _, crosschainRequest := range crosschainRequests {
    claimHash, _ := crosschainRequest.ClaimHash() // Calculate claimHash once for
    each request

    workflowType := crosschainRequest.WorkflowType()
    switch workflowType {
    case types.INBOUND:
        // Deduct fees and handle INBOUND requests
        // Use the previously calculated claimHash value
        // ...
    case types.OUTBOUND:
        // Handle OUTBOUND requests
        // ...
    case types.CROSSTALK:
        // Deduct fees and handle CROSSTALK requests
        // Use the previously calculated claimHash value
        // ...
    }
}
```

Status: Resolved

Issue is resolved as recommended.

Inefficient oracle tasks creating

Title	Inefficient oracle tasks creating
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [x/pricefeed/types/helpers.go](#)

Description

The `ComputeOracleTasks` function currently uses three separate loops to create Oracle tasks from a list of symbol requests. This methodology raises concerns regarding both runtime performance and memory utilization, especially for larger symbol request lists. The three loops encompass the creation of a map, population of an ID slice, and assembly of the Oracle tasks list:

```
func ComputeOracleTasks(symbols []SymbolRequest, blockHeight int64) []OracleTask {
    symbolsOsMap := make(map[uint64][]string)
    for _, symbol := range symbols {
        if symbol.BlockInterval != 0 && blockHeight%int64(symbol.BlockInterval) == 0
        {
            symbolsOsMap[symbol.OracleScriptID] =
append(symbolsOsMap[symbol.OracleScriptID], symbol.Symbol)
        }
    }

    ids := make([]uint64, 0, len(symbolsOsMap))
    for id := range symbolsOsMap {
        ids = append(ids, id)
    }
    sort.Slice(ids, func(i, j int) bool { return ids[i] < ids[j] })

    tasks := make([]OracleTask, len(symbolsOsMap))
    for i, id := range ids {
```

```

        tasks[i] = OracleTask{OracleScriptID: id, Symbols: symbolsOsMap[id]}
    }

    return tasks
}

```

Problem Scenarios

In the initial implementation of the `ComputeOracleTasks` function, three separate loops are used to achieve the desired outcome. This approach can lead to inefficiencies in terms of both runtime and memory usage, especially when dealing with larger symbol request lists. The three loops involve creating a map, populating an ID slice, and then creating the Oracle tasks list.

Recommendation

To improve the efficiency of the code, consider adopting the alternative implementation provided in the updated `ComputeOracleTasks` function:

```

func ComputeOracleTasks1(symbols []SymbolRequest, blockHeight int64) []OracleTask {
    symbolMap := make(map[uint64][]string)

    // stays the same
    for _, symbol := range symbols {
        if symbol.BlockInterval != 0 && blockHeight%int64(symbol.BlockInterval) == 0
    {
        symbolMap[symbol.OracleScriptID] =
        append(symbolMap[symbol.OracleScriptID], symbol.Symbol)
    }
    }

    tasks := make([]OracleTask, 0, len(symbolMap))
    // directly create tasks
    for oracleScriptID, symbols := range symbolMap {
        tasks = append(tasks, OracleTask{
            OracleScriptID: oracleScriptID,
            Symbols:         symbols,
        })
    }
    // sort them
    sort.Slice(tasks, func(i, j int) bool {
        return tasks[i].OracleScriptID < tasks[j].OracleScriptID
    })

    return tasks
}

```

This refined version utilizes only two loops, streamlining the process of directly generating Oracle tasks within the same loop. While the sorting step remains separate, this approach still helps minimize memory usage and potentially enhance runtime performance, especially when managing substantial symbol request lists.

Status: Resolved

Issue is resolved as recommended.

Suboptimal iteration and switch-case mechanism in Execute and SettleFees functions

Title	Suboptimal iteration and switch-case mechanism in Execute and SettleFees functions
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- x/crosschain/abci.go

Description

In the provided code snippets, there is an opportunity for optimization in both the `Execute` and `SettleFees` functions. The current implementations involve retrieving cross-chain requests and acknowledgment requests with specific statuses, and then iterating over them to process them based on their properties. This approach can be optimized to reduce redundant iterations and streamline the processing of different types of requests.

```
func Execute(ctx sdk.Context, k keeper.Keeper) {
    ///////////////////////////////////////////////////
    // 1. Query Crosschain Requests which are Validated //
    ///////////////////////////////////////////////////
    crosschainRequests := k.GetCrosschainRequestsByStatus(ctx,
types.CROSSCHAIN_TX_READY_TO_EXECUTE)
    for _, crosschainRequest := range crosschainRequests {
        workflowType := crosschainRequest.WorkflowType()
        switch workflowType {
        case types.INBOUND:
            // implementation logic
        case types.CROSSTALK, types.OUTBOUND:
            // Executed by Router relayer
        }
    }
}
```

```

func SettleFees(ctx sdk.Context, k keeper.Keeper) {
    ///////////////////////////////////////////////////
    // 1. Query Crosschain Ack Requests which are Validated //
    ///////////////////////////////////////////////////
    crosschainAckRequests := k.GetCrosschainAckRequestsByStatus(ctx,
types.CROSSCHAIN_ACK_TX_VALIDATED)
    for _, crosschainAckRequest := range crosschainAckRequests {
        switch crosschainAckRequest.WorkflowType() {
            case types.CROSSTALK_ACK, types.INBOUND_ACK:
                // implementation logic
            case types.OUTBOUND_ACK:
                // Already fee is settled when outbound ack is created.
        }
    }
}

```

Problem Scenarios

The current code retrieves cross-chain requests and acknowledgment requests based on their statuses and iterates through them to process them individually. This can lead to inefficiencies due to multiple iterations and switch-case checks for different request types.

Recommendation

1. **Optimize Execution of Cross-Chain Requests and Acknowledgments:** Instead of obtaining cross-chain requests and acknowledgment requests with specific statuses and then iterating over them separately, consider combining these steps into single iterations. For both the `Execute` and `SettleFees` functions, query the requests by both status and request type. By doing so, the switch-case mechanisms can be removed, and processing can be streamlined.
2. **Refactor Function Names:** With the optimizations in place, you can rename the functions to reflect their specific purposes. For example, `Execute` -> `ExecuteInboundRequests` and `SettleFees` -> `SettleCrossTalkAndInboundAcks` would be appropriate to clarify their functionalities.

Updated Code Snippets:

ExecuteInboundRequests Function:

```

func ExecuteInboundRequests(ctx sdk.Context, k keeper.Keeper) {
    crosschainRequests := k.GetCrosschainRequestsByStatusAndType(ctx,
types.CROSSCHAIN_TX_READY_TO_EXECUTE, types.INBOUND)
    for _, crosschainRequest := range crosschainRequests {
        // Process inbound requests
        // ...
    }
}

```

SettleCrossTalkAndInboundAcks Function:

```

func SettleCrossTalkAndInboundAcks(ctx sdk.Context, k keeper.Keeper) {
    crosschainAckRequests := k.GetCrosschainAckRequestsByStatusAndType(ctx,
types.CROSSCHAIN_ACK_TX_VALIDATED, types.INBOUND_ACK, types.CROSSTALK_ACK)
    for _, crosschainAckRequest := range crosschainAckRequests {
        // Process cross-chain acknowledgment requests
        // ...
    }
}

```

}

Inefficient validator search in nested loop

Title	Inefficient validator search in nested loop
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- x/attestation/abci.go

Description

The provided [code snippet](#) exhibits an inefficient search operation within a nested loop structure. The code's objective is to identify and process validators based on their presence in the `attestation.Votes` slice compared to the `currentBondedSet`.

However, the current implementation employs a nested loop, resulting in a time complexity of $O(k * n)$, where 'k' represents the length of `currentBondedSet`, and 'n' represents the length of `attestation.Votes`.

```
for _, bv := range currentBondedSet {
    found := false
    for _, val := range attestation.Votes {
        confVal, _ := sdk.ValAddressFromBech32(val)
        if confVal.Equals(bv.GetOperator()) {
            found = true
            break
        }
    }

    if !found {
        cons, _ := bv.GetConsAddr()
        consPower := k.StakingKeeper.GetLastValidatorPower(ctx, bv.GetOperator())

        k.StakingKeeper.Slash(
            ctx, cons, ctx.BlockHeight(),
```

```
        consPower, slashFractionMissingClaim,
    )

    if !bv.IsJailed() {
        k.StakingKeeper.Jail(ctx, cons)
    }
}
```

Problem Scenarios

The issue arises from the nested loop structure, which can lead to significant performance degradation when processing a large number of validators and votes. The nested loop checks each validator in `currentBondedSet` against every vote in `attestation.Votes`. This results in a quadratic time complexity, which can become highly inefficient as the data scales up.

Recommendation

To improve efficiency and reduce the time complexity to $O(k + n) \rightarrow O(n)$, consider the following optimized approach:

1. Create a map of validator addresses from `attestation.Votes` for quick and efficient validation checks.
2. Iterate through the `currentBondedSet` once, checking if each validator's address exists in the map created in step 1.

By implementing this optimized approach, you can significantly improve the performance of the code, making it more scalable and efficient when dealing with a large number of validators and votes.

```

// 1. Create a map of validator addresses
validatorMap := make(map[string]bool)
for _, val := range attestation.Votes {
    confVal, _ := sdk.ValAddressFromBech32(val)
    validatorMap[confVal.String()] = true
}

// 2. Iterate through the currentBondedSet once
for _, bv := range currentBondedSet {
    operator := bv.GetOperator().String()
    if _, found := validatorMap[operator]; !found {
        cons, _ := bv.GetConsAddr()
        consPower := k.StakingKeeper.GetLastValidatorPower(ctx, bv.GetOperator())

        k.StakingKeeper.Slash(
            ctx, cons, ctx.BlockHeight(),
            consPower, slashFractionMissingClaim,
        )

        if !bv.IsJailed() {
            k.StakingKeeper.Jail(ctx, cons)
        }
    }
}
k.DeleteAttestation(ctx, attestation)

```

Status: Resolved

Issue is resolved as recommended.

Inefficient validator set update in attestation processing

Title	Inefficient validator set update in attestation processing
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [x/attestation/abci.go](#)

Description

The provided code snippet is responsible for processing attestations, including slashing validators in certain conditions and managing the validator set using the `GetBondedValidatorsByPower` function.

However, the current approach pessimistically [updates the validator set](#) on each iteration through attestations, even if no validators are jailed during that iteration. This approach can result in unnecessary and potentially costly updates to the validator set.

```
for _, attestations := range attmap {  
    // calculate for every "attestations"  
    currentBondedSet := k.StakingKeeper.GetBondedValidatorsByPower(ctx)  
  
    // maybe jail some validator(s)  
}
```

Problem Scenarios

The issue arises from the pessimistic assumption that validators will be slashed frequently. As a result, the code calls `GetBondedValidatorsByPower` for every attestation, which involves iterating through the validator set based on their power ranking.

```

func (k Keeper) GetBondedValidatorsByPower(ctx sdk.Context) []types.Validator {
    maxValidators := k.MaxValidators(ctx)
    validators := make([]types.Validator, maxValidators)

    iterator := k.ValidatorsPowerStoreIterator(ctx)
    defer iterator.Close()

    i := 0
    for ; iterator.Valid() && i < int(maxValidators); iterator.Next() {
        address := iterator.Value()
        validator := k.mustGetValidator(ctx, address)

        if validator.IsBonded() {
            validators[i] = validator
            i++
        }
    }

    return validators[:i] // trim
}

```

This operation can be computationally expensive and is not efficient for cases where there are no jailed validators during an iteration.

Recommendation

To optimize the processing of attestations and reduce unnecessary calls to `GetBondedValidatorsByPower`, consider the following approach:

Optimistic validator set update: Only update the validator set using `GetBondedValidatorsByPower` when there is a confirmed need to do so. In other words, update the validator set if and only if there are validators jailed during the current iteration.

Here's a modified code snippet illustrating this approach:

```

// Initialize a flag to track whether any validators were jailed.
validatorsJailed := false

for _, attestation := range attmap {
    // ... (other code)

    if len(attestations) > 1 {
        // ... (other code)

        // Check if there are jailed validators during this iteration.
        if oneObserved {
            validatorsJailed = true
        }
    }

    if len(attestations) == 1 {
        // ... (other code)

        // Check if there are jailed validators during this iteration.
    }
}

```

```
        if windowPassed && attestation.Observed {
            validatorsJailed = true
        }
    }

    // Update the validator set only if there are jailed validators.
    if validatorsJailed {
        currentBondedSet := k.StakingKeeper.GetBondedValidatorsByPower(ctx)
        // ... (other code)
    }
```

Status: Resolved

Issue is resolved as recommended.

Lack of unit tests in Router chain modules

Title	Lack of unit tests in Router chain modules
Project	Router Protocol V2
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	04 UNKNOWN
Exploitability	04 UNKNOWN
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- all the modules on Router chain

Description

In our audit of the Router chain codebase, a concerning observation has emerged - the majority of modules are notably devoid of unit tests. Even in instances where unit tests exist, they are few and far between, often compounded by build problems. This dearth of unit testing is a matter of significant concern and warrants immediate attention.

Unit testing is an indispensable aspect of software development, particularly in the context of blockchain technology. Here's why:

Robustness and Reliability:

Unit tests verify the correctness of individual code components in isolation, ensuring that they perform as expected. This promotes code robustness and reliability, reducing the likelihood of bugs and vulnerabilities creeping into the system.

Early Bug Detection:

Unit tests allow for the early detection of issues during development, preventing them from escalating into more complex and costly problems later in the development cycle.

Maintainability:

Well-maintained unit tests serve as documentation, providing insights into how each module is intended to function. This is especially valuable for future developers who need to understand and modify the code.

Security:

In the context of blockchain, where security is paramount, unit tests play a crucial role in identifying vulnerabilities that could otherwise lead to exploits and financial losses.

While end-to-end (e2e) tests have their merits, they should not be viewed as a replacement for unit tests. E2E tests primarily validate the system's overall functionality, ensuring that different components interact correctly. However, they are less effective at pinpointing the exact source of a problem within a specific module.

Therefore, while we commend the presence of e2e tests in the Router Chain codebase, we emphasize that unit testing should not be omitted. Instead, it should be treated as an essential and complementary part of the testing strategy.

In conclusion, the absence of unit tests in Router Chain modules is a critical issue that demands immediate rectification. We strongly recommend prioritizing the development and maintenance of unit tests to bolster the reliability, security, and maintainability of the system.

Possible unnecessary saving Crosschain(Ack) request to storage

Title	Possible unnecessary saving Crosschain(Ack) request to storage
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- [x/crosschain/abci.go](#)

Description

Inside func Execute SetCrosschainRequest could potentially be called several times in order to change crosschain request status which could lead to unnecessary accessing of storage.

```

crosschainRequest.Status = types.CROSSCHAIN_TX_EXECUTED
claimHash, _ := crosschainRequest.ClaimHash()
k.SetCrosschainRequest(ctx, claimHash, crosschainRequest)

if execFlag == true {
    k.EmitCrosschainExecutedEvent(ctx, &crosschainRequest, execData)
} else {
    k.EmitCrosschainExecutionFailedEvent(ctx, &crosschainRequest, execData)
}

//////////
///// Settle fees /////
//////////
feeDeducted := crosschainRequest.DestTxFeeDeducted
relayerIncentive := crosschainRequest.RelayIncentive
amountAfterRelayerIncentive := feeDeducted.Sub(relayerIncentive)
refundAmount := sdk.NewCoin(chaintypes.RouterCoin, sdk.NewInt(0))
relayerFees := sdk.NewCoin(chaintypes.RouterCoin, sdk.NewInt(0))
if amountAfterRelayerIncentive.IsGTE(feeConsumedInRoute) {
    relayerFees = feeConsumedInRoute.Add(relayerIncentive)
    refundAmount = amountAfterRelayerIncentive.Sub(feeConsumedInRoute)
}

```

```

} else {
    relayerFees = amountAfterRelayerIncentive.Add(relayerIncentive)
}

k.Logger(ctx).Info("Total fee deducted for crosschain request", "srcChainId",
crosschainRequest.SrcChainId, "RequestIdentifier",
crosschainRequest.RequestIdentifier, "totalFees", feeDeducted, "feeConsumed",
feeConsumedInRoute, "refundAmount", refundAmount)
err = k.BankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, feePayer,
sdk.NewCoins(refundAmount))
if err != nil {
    k.Logger(ctx).Error("Error deducting fee for crosstalk request from FeePayer",
"srcChainId", crosschainRequest.SrcChainId, "RequestIdentifier",
crosschainRequest.RequestIdentifier, "feePayer", feePayer)
    continue
}

k.Logger(ctx).Info("Incentivize relayer for crosstalkrequest and refund sender and
Set crosschain request status to fees_settled", "srcChainId",
crosschainRequest.SrcChainId, "RequestIdentifier",
crosschainRequest.RequestIdentifier, "txFeeInRoute", feeDeducted,
"feeConsumedInRoute", feeConsumedInRoute, "relayerFee", relayerFees, "refundAmount",
refundAmount, "feepayer", feePayer)
crosschainRequest.Status = types.CROSSCHAIN_TX_FEES_SETTLED
k.SetCrosschainRequest(ctx, claimHash, crosschainRequest)
k.EmitCrosschainRequestFeeSettlementEvent(ctx, &crosschainRequest, relayerFees,
refundAmount)

```

Since func `ExecuteAck` is written similarly to func `Execute`, the same type of issue is also present in that function.

Problem Scenarios

Changing the status `crosschainRequest.Status = types.CROSSCHAIN_TX_EXECUTED` and calling `k.SetCrosschainRequest(ctx, claimHash, crosschainRequest)` is unnecessary at this point since later on there is another change to different status `crosschainRequest.Status = types.CROSSCHAIN_TX_FEES_SETTLED` and also another saving to the same storage.

Recommendation

Consider changing the status to `types.CROSSCHAIN_TX_EXECUTED` and saving it in the store only if there is an error after `err = k.BankKeeper.SendCoinsFromModuleToAccount(ctx, types.ModuleName, feePayer, sdk.NewCoins(refundAmount))` before calling “continue” to go to the next iteration. That would make sure status is only changed if there is necessity to do it.

Replace "require" with "revert" and custom errors to save gas

Title	Replace "require" with "revert" and custom errors to save gas
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	3 HIGH
Status	RESOLVED
Issue	

Involved artifacts

- [router-gateway-contracts/evm at 4cbae6ee \(informal-audit branch\)](#)

Description

All of Router Solidity contracts use `require` with strings for errors. Here are some examples from [GatewayUpgradeable.sol](#):

```
require(keccak256(bytes(chainId)) == keccak256(bytes(requestPayload.destChainId)),
"wrong dest ID");

// "cross-chain request message already handled" => "C06"
require(nonceExecuted[requestPayload.srcChainId][requestPayload.requestIdentifier] ==
false, "C06");

if (requestPayload.routeAmount > 0) {
    //caution: make sure recipient should be correct
    require(requestPayload.routeRecipient != address(0), "Recipient = addr(0)");
    vault.handleWithdraw(requestPayload.routeAmount, requestPayload.routeRecipient);
}
```

In the more recent versions of Solidity compiler there is a possibility to use `revert` with custom error types. Besides providing better and more informative possibilities for handling errors (e.g. via passing error parameters), this also allows to reduce gas usage.

Recommendation

We recommend introduce custom error types using the `error` statement, and replace all `require` statements employing strings with `revert` statements with custom errors.

Status: Resolved

Issue is resolved as recommended.

Various optimizations for Solidity contracts

Title	Various optimizations for Solidity contracts
Project	Router Protocol V2
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Status	RESOLVED
Issue	

Involved artifacts

- [router-gateway-contracts/evm at 4cbae6ee](#) (informal-audit branch)

Description

- Remove unneeded `keccak256(bytes(...))` calls in [GatewayUpgradeable.sol#L366](#):

```
require(keccak256(bytes(chainId)) ==  
keccak256(bytes(requestPayload.destChainId)), "wrong dest ID");
```

- `_setupRole` is a deprecated function of `AccessControlUpgradeable` ; `_grantRole` should be used instead, in [GatewayUpgradeable.sol#L217](#).
- In [GatewayUpgradeable.sol#L220-L223](#), replace double assignment to `eventNonce`

```
eventNonce = 0;  
stateLastValsetNonce = valsetNonce;  
stateLastValsetCheckpoint = newCheckpoint;  
eventNonce++;
```

with

```
stateLastValsetNonce = valsetNonce;  
stateLastValsetCheckpoint = newCheckpoint;  
eventNonce = 1;
```

Status: Resolved

Issue is resolved as recommended.


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:

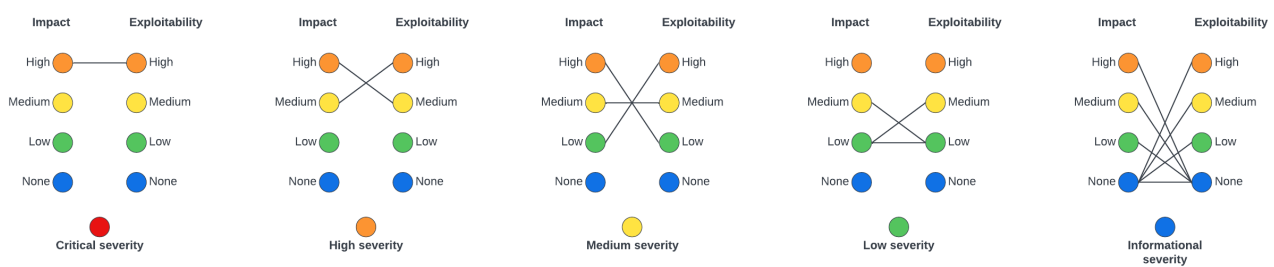
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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

This report is subject to the terms and conditions (including without limitation, description of services, confidentiality, disclaimer and limitation of liability, etc.) set forth in the associated Services Agreement. This report provided in connection with the Services set forth in the Services Agreement shall be used by the Company only to the extent permitted under the terms and conditions set forth in the Agreement.

This audit report is provided on an “as is” basis, with no guarantee of the completeness, accuracy, timeliness or of the results obtained by use of the information provided. Informal has relied upon information and data provided by the client, and is not responsible for any errors or omissions in such information and data or results obtained from the use of that information or conclusions in this report. Informal makes no warranty of any kind, express or implied, regarding the accuracy, adequacy, validity, reliability, availability or completeness of this report. This report should not be considered or utilized as a complete assessment of the overall utility, security or bugfree status of the code.

This audit report contains confidential information and is only intended for use by the client. Reuse or republication of the audit report other than as authorized by the client is prohibited.

This report is not, nor should it be considered, an “endorsement”, “approval” or “disapproval” of any particular project or team. This report is not, nor should it be considered, an indication of the economics or value of any “product” or “asset” created by any team or project that contracts with Informal to perform a security assessment. This report does not provide any warranty or guarantee regarding the absolute bug-free nature of the technology analyzed, nor does it provide any indication of the client’s business, business model or legal compliance. This report should not be used in any way to make decisions around investment or involvement with any particular project. This report in no way provides investment advice, nor should it be leveraged as investment advice of any sort.

Blockchain technology and cryptographic assets in general and by definition present a high level of ongoing risk. Client is responsible for its own due diligence and continuing security in this regard.