Security Audit Report

Neutron:
Code Inspection and Protocol Analysis

17.03.2023
Last revised 27.03.2023

Authors: Dusan Maksimovic, Stana Miric

Contents

Audit overview

The Project

In January 2023, P2P staking engaged Informal Systems to conduct a security audit over the documentation and the current state of the implementation of the Neutron project.

Neutron is a proof-of-stake blockchain designed as a permissionless smart contracting platform which allows user-deployed smart contracts to easily interact with other Cosmos blockchains by leveraging Neutron's custom modules and other components. It is based on Tendermint, Cosmos SDK, IBC, CosmWasm and Interchain Security. Neutron blockchain consists of several custom modules:

Interchain Queries - allows smart contracts do define a customizable queries to obtain the states from other blockchains.

Interchain Transactions - introduces support for smart contracts to register interchain accounts on other blockchains and send transactions by using those accounts.

Transfer - a wrapper around IBC Transfer module that allows transfer results to be processed by the smart contracts that initiated the transfer.

Contract Manager - handles communication from the Neutron modules to the smart contracts when the results that were requested by the smart contracts have been received on Neutron blockchain. This is done by executing `sudo` calls on the receiving contracts. It also stores information about any contract failures that occurred in smart contracts during sudo calls execution.

Fee Refunder - smart contracts must pay certain fees when they want to execute some transactions or transfer tokens through IBC to other blockchains. These fees are stored on fee refunder module and paid out to the IBC relayers when they deliver the results.

Fee Burner - responsible for burning neutron tokens that have been collected through transaction fees. Works in conjunction with a treasury smart contract that will release certain number of tokens from the treasury based on how many tokens were burned by the fee burner module.

Neutron deploys two sets of smart contracts that are an integral part of the product. First set is used for the governance of the blockchain, and the second one handles the tokenomics by managing the neutron tokens treasury and distribution.

Neutron also has one off-chain component: Interchain Queries Relayer. This component is responsible for tracking the interchain queries on the Neutron chain, executing those queries on destination chains and submitting query results (together with proofs) back to Neutron blockchain.

Scope of this report

The agreed-upon workplan consisted of the following tasks:

1. Audit of the Neutron's custom modules and application source code.
2. Audit of the Neutron SDK, a CosmWasm bindings intended to be used by smart contracts to interact with other blockchains through Neutron's custom modules.
3. Audit of Neutron DAO set of governance smart contracts, and of tokenomics smart contracts.
4. Audit of WasmD fork adapted to meet specific needs of the Neutron blockchain.

This report covers the above tasks that were conducted from January 18 through March 10 by Informal Systems by the following personnel:

- Dusan Maksimovic
- Stana Miric

Conducted work

At the kick-off meeting the Neutron team gave us a brief introduction to the Neutron blockchain and Neutron DAO set of governance smart contracts. We agreed upon the exact revisions of each repository that we should audit.

The team first read through all the official documentation of the Neutron system overview, custom modules, governance and tokenomics smart contracts. After that, the team performed manual code review with a focus mainly on code correctness and the critical points analysis of Neutron's custom modules, smart contracts and CosmWasm bindings.

Timeline

- 18.01.2023: Kick-off meeting with the Neutron team.
- 19.01.2023: Sync meeting 1, Andrei and Mikhail did a short code walkthrouh of the Neutron chain modules and partially the governance contracts mechanism.
- 26.01.2023: Sync meeting 2, we went through the first findings, one of them was critical (ability for anyone to remove the interchain query immediately, before the first results were submitted, and take the smart contract's deposit)
- 02.02.2023: Sync meeting 3, we went through the new findings in interchaintxs and feerefunder modules and also discussed a potential problem with the way interchainqueries module stores and compares the heights of the submitted results. As a result, a new issue is reported.
- 09.02.2023: Sync meeting 4, we discussed the latest finding for whitelisting of the governance proposal types and two other issues that represent a collections of smaller issues in Neutron modules and CosmWasm bindings code, but do not pose a security threat.
- 23.02.2023: Sync meeting 5, we started the second part of the audit- the Neutron's governance and tokenomics smart contracts. No issues have been found so far.
- 02.03.2023: Sync meeting 6, we discussed smaller issues found in the governance and treasury smart contracts.
- 09.03.2023: Sync meeting 7, we presented the finding with collection of minor issues found in the smart contracts code.
- 10.03.2023: End of audit
- 10.03.2023: submission of first draft of this report

Conclusions

We found that the Neutron design and security model in general is well thought out. Despite the general high quality, we found some details that should be addressed to raise the quality of the code. One **Critical Severity** and one **High Severity** issues were found during this audit; the rest were marked as Low or Informational severity.

Further Increasing Confidence

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

It is our understanding that the P2P staking team intends to pursue such measures to further improve the confidence in their system.

Audit Dashboard

Target Summary

- **Type**: Specification and Implementation
- **Platform**: Go, Rust
- **Artifacts**
  - neutron-org/neutron @ 64868908b21f648ad5e8a9b48179134619544e2a
  - neutron-org/neutron-dao @ 5a0ab5a60f7e1e3d9e532da0be8be3f57c7e16c7
  - neutron-org/neutron-sdk @ c19b40c024eeaa8733af9ddee94a52798d78f469
  - neutron-org/wasmd @ f37577b9c030221c40823916d47d94e8cd844fe3

Engagement Summary

- **Dates**: 18.01.2023 to 10.03.2023
- **Method**: Manual code review & protocol analysis
- **Employees Engaged**: 2

Severity Summary

Finding Severity	#
Critical	1
High	1
Medium	0
Low	2
Informational	11
Total	**15**

System Overview

In this section, we give a high-level overview of the system, which is useful for understanding the rest of the report. For more details on the system, see the project documentation.

Neutron introduces a permissionless smart contracting platform which offers support for any smart contract to easily obtain the values from the state of other Cosmos blockchains, as well as to register interchain accounts and send transactions to other blockchains by using those accounts. It also allows smart contracts to perform IBC transfers and, by using its custom IBC Transfer module, to inform the smart contracts about the outcome of those transfers. To be able to support all of these functionalities, Neutron has created a system which consists of multiple on-chain and off-chain components that are shown in Figure 1.

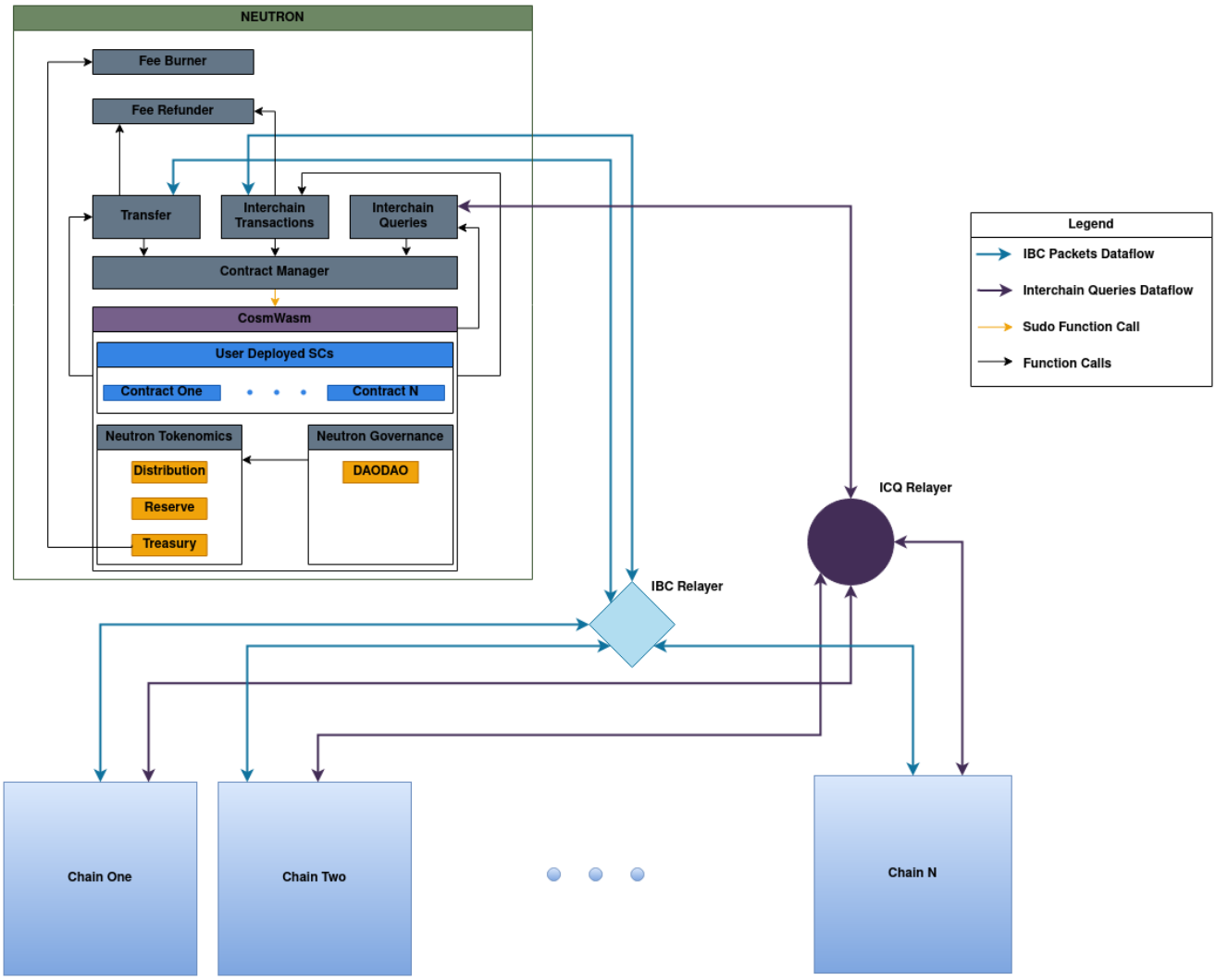

Figure 1: Neutron system overview

Custom modules

Neutron blockchain uses multiple Cosmos SDK modules, as well as the following custom modules:

1. Consumer module of the Interchain Security, since it will start as a consumer chain

2. CosmWasm module, which adds support for smart contracts
3. IBC module, which allows Neutron to communicate with other blockchains
4. Interchain Accounts module, which allows smart contracts to create accounts on other Cosmos blockchains
5. Admin module, which allows specified admin addresses to execute governance proposal messages once their proposals have passed

It also defines a couple of its own custom modules that are adding support for communication between the smart contracts deployed on the Neutron blockchain and other Cosmos blockchains. Below we present a brief overview of these custom modules.

Interchain Queries

This module allows smart contracts to create a customizable queries to obtain the desired state from other blockchains. Smart contract manages its interchain queries by using `MsgRegisterInterchainQuery`, `MsgUpdateInterchainQuery` and `MsgRemoveInterchainQuery` messages defined by this module. Currently, there are two types of supported interchain queries: key-value and transaction. Key-value queries allow for querying the state of the store on a destination blockchains by specifying the keys whose values they want to obtain. Transaction queries allow for querying the transactions on destination blockchains that satisfy the specified transaction filter conditions. To create an interchain query, smart contracts must pay the deposit whose amount is specified in the parameters of the module. This is used to protect the Neutron blockchain from a Byzantine smart contracts that could otherwise spam the network and force the blockchain nodes to store the queries which are not used by anyone. Only the smart contract that created the interchain query can remove it from the blockchain. But, if the results for a certain query haven't been submitted for longer than the `submit timeout` period of blocks, it becomes free for anyone to submit the `MsgRemoveInterchainQuery` to remove the obsolete query and claim the deposit that was paid by the smart contract who created it. This is used as an incentive for everyone, which helps with the clean-up of unused interchain queries from the blockchain. The module also defines a `MsgSubmitQueryResult` message which is used by the Interchain Query Relayer to submit the results of the interchain queries that were obtained from the destination chain. Together with the results, the ICQ Relayer submits the Merkle Proofs which prove that the results actually exist on the destination chain. These proofs are then verified against the IBC light clients stored on the Neutron blockchain. If the verification is successful, smart contract is notified about the results through the `sudo` calls managed by the `Contract Manager` module keeper. Additionally, for the interchain queries of key-value type, the results are stored in the interchain queries module store.

Interchain Transactions

This module allows smart contracts to register interchain accounts on destination blockchains and send transactions to those blockchains by using the created accounts. There is no limitation in number of interchain accounts that one smart contract can register. The module exposes two messages: `MsgRegisterInterchainAccount` and `MsgSubmitTx`. Smart contracts that wants to submit transactions to some destination blockchains will send the `MsgSubmitTx`. When submitting this message, a smart contract must pay acknowledgement and timeout fees that will be locked on the `Fee Refunder` module account until the results are delivered by the IBC relayer.

Transfer

This module is a wrapper around the IBC Transfer module, which adds better support for smart contracts to perform IBC token transfers. The main advantage of this module over the standard IBC Transfer module is that it will handle the acknowledgements and timeouts submitted by the IBC relayers and forward them to the smart contracts that initiated the transfer. That way, a smart contract can take proper action to successful or timed out IBC token transfers. Similarly to sending the transactions to destination blockchains through the `Interchain Transactions` module, smart contract must pay acknowledgement or timeout fees to IBC relayers that deliver the results.

Contract Manager

This module exposes the API for other Neutron modules to allow them to communicate with smart contracts by leveraging the `sudo` calls provided by the CosmWasm module. The module is used by `Interchain Queries`, `Interchain Transactions` and `Transfer` modules each time when IBC or ICQ relayers deliver some results to Neutron blockchain. After the results are verified by the modules, they are propagated to the smart contract which initiated transaction submission on destination chain, IBC token transfer, or when the results of interchain query are submitted by the ICQ relayer. It is the responsibility of the smart contract developers to implement corresponding `sudo` handlers in their smart contracts, which will then be called by the `Contract Manager` module. If an error occurs during any `sudo` handler execution in smart contract code, the module will record this contract failure in its

store, but it will not reject the transaction, which will be included in the block. This is a mechanism implemented to stop a potential Byzantine smart contracts from deliberately returning errors from `sudo` handlers and causing relayers to re-submit transactions, thus leading to drainage of relayers funds and spamming of the network.

Fee Refunder

This module is responsible for locking the acknowledgement and timeout fees that are paid by the smart contracts for each IBC packet they want to send to some destination chain. These fees remain locked on the module account until some IBC relayer delivers an acknowledgement or timeout result for the given IBC packet. The paid fees information is saved in module store under the key that is built by using port ID, channel ID and sequence number, which makes it possible to find the specific IBC packet fees that were paid by the smart contract when the IBC relayer delivers the results. When the IBC relayer submits the acknowledgement to the Neutron blockchain, the acknowledgement fees paid by the smart contract will be sent to the relayers account and the timeout fees are sent back to the smart contracts account. The smart contract also receives the results of the acknowledgement. In case of the IBC packet timeout, the relayer will receive the timeout fees while the acknowledgement fees are sent back to the smart contract address, and the smart contract gets notified about the packet timeout.

Fee Burner

This module is responsible for handling the tokens collected through the transaction fees that belong to the consumer chain, after the consumer module handled the portion that belongs to the provider chain. The `Fee Burner` module logic runs at the end of each block. It goes through all denoms of the collected tokens, checks if the denom is Neutron denom and burns the collected amount. If it is some other denom, the collected amount will be sent to the `Treasury` smart contract address. The Neutron denom name and the address of the `Treasury` smart contract are stored in the module parameters. The module also records the total amount of burned Neutron tokens, which is later used by the `Treasury` smart contract to calculate how many tokens should be released from the treasury.

Neutron SDK

Neutron SDK is a set of structures and utility functions written in Rust, which provides support for smart contracts to communicate with Neutron custom modules. Smart contracts can import this SDK and write their custom logic which will eventually use the SDK structures to pack the data that will be sent to one of the Neutron custom modules for further processing. This way, the developers of smart contracts don't have to write this code by themselves and they can rely on the Neutron SDK instead. The SDK offers the following functionalities:

1. Structures and functions for registering interchain accounts on other Cosmos blockchains and sending transactions by using those accounts. This set of features allows communication with the `Interchain Transactions` module.
2. Structures and functions for registration, update and removal of the interchain queries on the Neutron blockchain. Currently, there is support for some of the most used queries like account balances, total supply of the tokens, querying the validators and delegations on other blockchains, etc. There is also support for obtaining the registered queries and their results from the Neutron blockchain. This set of features allows communication with the `Interchain Queries` module.

Smart contracts

Neutron blockchain will start with two sets of smart contracts that will be pre-deployed through the genesis file.

DAO smart contracts

This set of smart contracts is based on DAO DAO contracts and, with some modifications, it is used to handle the governance of the Neutron blockchain. It consists of two parts:

1. The Neutron DAO
2. Multiple subDAO-s

The `Neutron DAO` is a set of smart contracts which handles the governance of the Neutron blockchain. The main `Neutron DAO` contract address is added as the only admin of the `Admin` module, which allows it to submit messages to the `Admin` module in order to execute governance messages (e.g. parameter change, software upgrade, etc.) once the proposal has passed. One difference when compared to the original `DAO DAO` contracts is that the `Neutron DAO`

introduces a special type of overrule proposals. This type of proposals are voted on the `Neutron DAO`, and allow the main `Neutron DAO` to overrule any proposal that has already passed on any of the subDAO-s. This contract works with the `Timelock` contract which gets deployed as part of all subDAO-s. `Timelock` contract locks the passed subDAO proposals for a defined period of time. The overrule type of proposal has a shorter voting period than the lock period of the `Timelock` contract and a smaller threshold, which allows it to overrule already passed proposals on any subDAO that are still locked by the `Timelock` contract.

SubDAO set of contracts are intended to be instantiated after the Neutron blockchain has started, except for the special `Security subDAO` set of smart contracts that is also instantiated through the genesis file. The `Security subDAO` set of contracts has one special permission: to pause execution of all other subDAO-s as well as all tokenomics smart contracts. When the contract is paused, no other action can be executed, except for unpause, which can only be executed by the main `Neutron DAO` contract after the governance proposal to do so has successfully passed.

Tokenomics smart contracts

This set of smart contracts handles Neutron funds. It consists of 3 contracts:

1. Treasury
2. Distribution
3. Reserve

All three contracts will be pre-deployed through the genesis file. The main `Neutron DAO` and `Security subDAO` contract addresses must be provided in all three contract instantiation messages, since they can perform special actions on the tokenomics contracts. The `Treasury` contract will have certain amount of tokens asssigned to it in the genesis file. The contract exposes a permissionless method that performs the distribution of tokens from the treasury. The amount of tokens to be distributed is calculated by using the information about the amount of Neutron tokens that were burned by the `Fee Burner` module, counting from the last distribution. One portion of tokens to distribute goes to the `Distribution` smart contract, which distributes them among the shareholders whose addresses are tracked within the `Distribution` contract, and the other portion is sent to the `Reserve` contract which can later be used by the main `Neutron DAO` to pass governance proposals for one-off payments.

Interchain Queries Relayer

This is an off-chain component which handles execution of interchain queries against the destination blockchains and submission of the results to the Neutron blockchain. The ICQ relayer monitors events published by the `Interchain Queries` module when interchain queries are registered, updated or removed. It then uses information from those events to prepare and periodically execute queries and deliver the results, together with the Merkle Proofs of their existence, to the Neutron blockchain.

Findings

Severity	Type	Finding
Critical	Implementation	Interchain Queries register and remove logic can be exploited to steal smart contract's deposit and stop it from creating the queries
High	Implementation	Non-validated IBC acknowledgement/timeout fees can lead to drainage of relayers funds and spamming of the network
Low	Implementation	InitGenesis() execution could cause ICQ-s to be overwritten in the store
Low	Implementation	Interchain Query keys are not properly validated
Informational	Implementation	Transactions stored for ICQ results of TX type can be removed when the query is removed
Informational	Implementation	Interchain Query of type TX can not be updated
Informational	Implementation	Heights for ICQ results submission are not properly checked
Informational	Implementation	Consumer governance proposal types whitelisting
Informational	Implementation	MsgSubmitQueryResult could be optimized to use less gas
Informational	Implementation	ValidateBasic() for the MsgRegisterInterchainAccount should validate maximum length of the InterchainAccountId field
Informational	Implementation	Introduce support for quering the minimum IBC fees from the smart contracts
Informational	Implementation	Various minor issues in the Neutron custom modules
Informational	Implementation	Various minor issues in the Neutron smart contracts
Informational	Implementation	Various minor issues in the Neutron SDK
Informational	Documentation	Various documentation inconsistencies

Interchain Queries register and remove logic can be exploited to steal smart contract's deposit and stop it from creating the queries

ID	IF-NEUTRON-01
Severity	Critical
Impact	High
Exploitability	High
Type	Implementation
Status	Resolved through PR 162

Involved artifacts

- interchainqueries/keeper/msg_server.go

Description

When a smart contract sends a message to register the interchain query (ICQ), a new `RegisteredQuery` structure is instantiated with its field `LastSubmittedResultLocalHeight` left uninitialized, which will default its value to 0. This field is later used to determine if it is allowed for anyone to remove the ICQ from the store if the result for the ICQ hasn't been submitted for more than a `SubmitTimeout` period. Leaving the `LastSubmittedResultLocalHeight` field initialized to 0 can lead to an exploit that would allow anyone to take the smart contract's deposit and remove its newly registered queries.

Problem Scenarios

1. Neutron chain has been running for at least 1036801 blocks (this is by 1 more from the default value of `QuerySubmitTimeout` parameter).
2. A smart contract registers a new KV ICQ in block 1036801. At this moment, `LastSubmittedResultLocalHeight` is set to 0 and `SubmitTimeout` to 1036800
3. A malicious software monitors the Neutron chain for the `query_updated` events and sees that the new query was registered and submits the `MsgRemoveInterchainQueryRequest` as a regular user in block 1036802, before any ICQ Relayer submits the first results.
4. If a malicious software submitted the message immediately, the following code would produce a value 1036800:

```
timeoutBlock := query.LastSubmittedResultLocalHeight + query.SubmitTimeout
```

Since the `ctx.BlockHeight()` would be 1036802, the following condition would not be satisfied:

```
if uint64(ctx.BlockHeight()) <= timeoutBlock && query.GetOwner() != msg.GetSender()
```

which would allow ICQ to be removed from the store and the deposit paid by the smart contract to be refunded to the malicious user address.

Recommendation

If possible, by setting the `LastSubmittedResultLocalHeight` to the `ctx.BlockHeight()` this issue would be resolved. The downside would be that ICQ relayer wouldn't submit the results for the newly created ICQ immediately, but only after `UpdatePeriod`.

Non-validated IBC acknowledgement/timeout fees can lead to drainage of relayers funds and spamming of the network

ID	IF-NEUTRON-02
Severity	High
Impact	Medium
Exploitability	High
Type	Implementation
Status	Resolved through PR 163

Involved artifacts

- interchaintxs/keeper/ibc_handlers.go
- feerefunder/keeper/keeper.go

Description

The fees paid by the smart contracts for the acknowledgement/timeout of the IBC events are not validated good enough, which can be exploited by a malicious smart contracts to cause an `ErrorOutOfGas` panics during the `OnAcknowledgementPacket()` and `OnTimeoutPacket()` messages processing. These panics would not be recovered through `outOfGasRecovery()` execution, which means that the transactions would not be included in a block, and would cause IBC relayer to submit the messages repeatedly and lead to drainage of relayers funds and spamming of the network.

Problem Scenarios

A malicious smart contract sends the `MsgSubmitTx` to the Neutron chain and specifies the `AckFee` and `TimeoutFee` as a very long slices of valid coins. This can be achieved by previously performing IBC transfers of small amount of coins from other chains to the smart contract's address on the Neutron chain. This would allow a smart contract to specify fees similar to the following ones, but with much more different coins:

```
let ack_fee = vec![
  CosmosCoin::new(1000u128, "neutron"),
  CosmosCoin::new(1u128,
  ↪  "IBC/27394FB092D2ECCD56123C74F36E4C1F926001CEADA9CA97EA622B25F41E5EB2"),
  CosmosCoin::new(1u128,
  ↪  "IBC/17394FB092D2ECCD56123C74F36E4C1F926001CEADA9CA97EA622B25F41E5EB3"),
  ...],
```

1. `MsgSubmitTx` would be successfully submitted since the smart contract would pay enough fees in at least one coin, as verified here.
2. After the transaction is executed successfully (or times out) on the destination chain, the IBC relayer submits `OnAcknowledgementPacket()` or `OnTimeoutPacket()`, respectively.
3. Let's assume that the gas limit is 150.000 units and, at this point of `HandleAcknowledgement()` execution, 50.000 units were already consumed. A `newGasMeter` would be created with a limit of 85.000 gas units.
4. The smart contract would write `SudoError()` and `SudoResponse()` so that they consume approximately 84.000 gas units, and does not go over the limit.
5. At this point, the main gas meter still has 100.000 unused gas units, since the gas used through the `newGasMeter` still hasn't been subtracted. The method `DistributeAcknowledgementFee()` would complete successfully, but it would use, for example, 20.000 gas units, due to the very large `Fee` object being read from the store.
6. At this point, the main gas meter has 80.000 gas units remaining and tries to subtract 84.000 gas units that were consumed by the `newGasMeter` and causes `panic` with `ErrorOutOfGas`. Notice that the `newGasMeter` is still not out of gas, since it has 1.000 gas units left.

7. The outOfGasRecovery() triggers, but since the error type is `ErrorOutOfGas`, but the `gasMeter` (which is actually the `newGasMeter`) is not out of gas, it will propagate the panic call, which means that the `OnAcknowledgementPacket()` would not be included in the block and would cause the relayer to submit the transaction over and over again.

Recommendation

Validate the acknowledgement/timeout fees so that only denoms specified in the parameters can be used, or at least check the length of the `AckFee` and `TimeoutFee` slices.

InitGenesis() execution could cause ICQ-s to be overwritten in the store

ID	IF-NEUTRON-03
Severity	Low
Impact	Low
Exploitability	Low
Type	Implementation
Status	Resolved through PR 136

Involved artifacts

- interchainqueries/genesis.go

Description

The InitGenesis() method assumes that the `RegisteredQueries` in the genesis file are ordered in ascending order by the query ID.

Problem Scenarios

If the `RegisteredQueries` in the genesis file are not ordered in ascending order by query ID, then the `SetLastRegisteredQueryKey()` would be called last for ID that is not the largest one, and later executions of `MsgRegisterInterchainQuery` would overwrite existing queries in the store.

Recommendation

Sort the `RegisteredQueries` slice by query ID before iteration.

Interchain Query keys are not properly validated

ID	IF-NEUTRON-04
Severity	Low
Impact	Low
Exploitability	Low
Type	Implementation
Status	Resolved through PR 137

Involved artifacts

- interchainqueries/types/tx.go

Description

When registering and updating Interchain Queries of KV type, the keys aren't checked for nil values before they are used. Smart contracts could send a `MsgRegisterInterchainQuery` where the keys would be defined as `Vec<Option<KVKey>>` and populate them with `None` values, that get converted into nil values on Go side (parsing is done successfully), which would cause a panic during `ValidateBasic()` execution.

Also, the keys are not checked for duplicated values so it is possible to create an ICQ with 32 identical keys.

Problem Scenarios

As per Description

Recommendation

Modify validateKeys() to check for nil and duplicate keys.

Transactions stored for ICQ results of TX type can be removed when the query is removed

ID	IF-NEUTRON-05
Severity	Informational
Impact	Low
Exploitability	None
Type	Implementation
Status	Resolved through PR 156

Involved artifacts

- interchainqueries/keeper/msg_server.go

Description

For the Interchain Queries (ICQ) of TX type, the ICQ relayers will submit the results that are stored in the `interchainqueries` module store. These transaction results, as stated in the comment, are not removed when such ICQ is removed from the store, but they actually can be removed without knowing the TX hashes.

Problem Scenarios

Transactions that are stored for removed ICQ-s of TX type will consume more and more space on the chain over time.

Recommendation

Here is the code which removes the stored TX results for the given query ID:

```
prefixStore := prefix.NewStore(ctx.KVStore(k.storeKey),
↪  iqtypes.GetSubmittedTransactionIDForQueryKeyPrefix(query.Id))
iterator := prefixStore.Iterator(nil, nil)
defer iterator.Close()
for ; iterator.Valid(); iterator.Next() {
  txHashKey := iterator.Key()
  prefixStore.Delete(txHashKey)
}
```

Interchain Query of type TX can not be updated

ID	IF-NEUTRON-06
Severity	Informational
Impact	Low
Exploitability	None
Type	Implementation
Status	Resolved through PR 141

Involved artifacts

- interchainqueries/types/tx.go

Description

It is not possible to update the ICQ of type TX by just specifying the `new_transactions_filter` without setting the `new_update_period` as well.

Problem Scenarios

The `ValidateBasic()` method of `MsgUpdateInterchainQueryRequest` will return an error if a smart contract sends message to only update the `new_transactions_filter`.

Recommendation

Allow updating ICQ of type TX by just specifying the `new_transactions_filter`.

Heights for ICQ results submission are not properly checked

ID	IF-NEUTRON-07
Severity	Informational
Impact	Low
Exploitability	None
Type	Implementation
Status	Unresolved

Involved artifacts

- proto/interchainqueries/genesis.proto
- interchainqueries/keeper/keeper.go

Description

When an ICQ relayer submits the results for queries of KV type, there is a check to verify if the submitted results are for the height that is larger than the one for which the results were already submitted before. However, it only checks the `RevisionHeight` value, and not the `RevisionNumber` value, which could lead to issues in some scenarios.

Problem Scenarios

As stated in the IBC documentation, it is possible that, during the upgrade, the chain resets its `RevisionHeight` to 0 and increases the `RevisionNumber` by one. If this happens, the ICQ relayer couldn't submit the query results anymore, until the destination chain reaches the `RevisionHeight` it used to have before the height-resetting upgrade.

Recommendation

Modify the last_submitted_result_remote_height in `RegisteredQuery` to use a structure that keeps track of both `RevisionNumber` and `RevisionHeight` and compare them properly when the new query results are being submitted.

Consumer governance proposal types whitelisting

ID	IF-NEUTRON-08
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PRs 152, 83 and 41

Involved artifacts

- app/proposals_whitelisting.go

Description

Parameters of the `feeburner` module are not whitelisted for the consumer governance, and some other proposal types are whitelisted, but without proper CosmWasm binding support.

Problem Scenarios

1. Parameters defined by the `feeburner` module can't be changed through the governance by submitting the `MsgSubmitProposal` to the `adminmodule`.
2. Also, some other proposal types have been whitelisted, but there are no corresponding CosmWasm bindings that would allow these types of proposals to be submitted by the governance smart contracts.

Recommendation

1. Whitelist the `feeburner` module parameters by extending the whitelist here.
2. Remove the unnecessary proposal types or provide the appropriate CosmWasm bindings that would allow them to be submitted by the governance smart contracts.

MsgSubmitQueryResult could be optimized to use less gas

ID	IF-NEUTRON-09
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PR 140

Involved artifacts

- interchainqueries/keeper/msg_server.go
- interchainqueries/keeper/keeper.go

Description

`MsgSubmitQueryResult` could be optimized to use less gas.

Problem Scenarios

1. If the ICQ results are valid they will be saved into the store by calling the SaveKVQueryResult() method. This method calls `k.UpdateLastRemoteHeight()` and `k.UpdateLastLocalHeight()` which both load the registered query from the store, modify one field and store the query back to the store. Less gas would be consumed if both fields were updated at once since the query would be loaded/saved only once. Since this message is called many times by ICQ relayers, it could save a lot of gas over a long distance.
2. `MsgSubmitQueryResult` should check in the beginning of the execution if the KV ICQ results for the given (or higher) remote height were already submitted so that less gas is spent for the relayer that tries to submit the same or older results.

Recommendation

Perform an early check of the results remote height and load/save query just once per message execution.

ValidateBasic() for the MsgRegisterInterchainAccount should validate maximum length of the InterchainAccountId field

ID	IF-NEUTRON-10
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PR 157

Involved artifacts

- interchaintxs/types/tx.go

Description

The `ValidateBasic()` method for the `MsgRegisterInterchainAccount` should also check the maximum length of the `InterchainAccountId` field.

Problem Scenarios

If the smart contract sends the `InterchainAccountId` value that is too long, it will cause a panic in the Interchain Accounts module code when the controller keeper tries to bind such port. The maximum port identifier length is 128 characters, so the `InterchainAccountId` maximum length must be less than: 128 - len("icacontroller-") - len(smart_contract_address) - len(".")

Recommendation

Validate the maximum length of the `InterchainAccountId` field in `ValidateBasic()` method.

Introduce support for quering the minimum IBC fees from the smart contracts

ID	IF-NEUTRON-11
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PR 164

Involved artifacts

- wasmbinding/custom_querier.go

Description

Introduce a query in the `feerefunder` module that would allow smart contracts to obtain the minimum fees for IBC acknowledgements/timeouts stored in the module's parameters.

Problem Scenarios

Since a smart contract doesn't have a way to find out what are the minimum IBC fees during its messages execution, it is forced to hard-code these values. The problem arises if the governance changes these parameters to higher values or different tokens. In this case, a smart contracts would need to be updated with a new code so that they can continue sending IBC messages.

Recommendation

Introduce a feerefunder module query to allow smart contracts to obtain the minimum IBC fees values.

Various minor issues in the Neutron custom modules

ID	IF-NEUTRON-12
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PR 149

Involved artifacts

- x/interchainqueries/types/genesis.go
- x/interchainqueries/keeper/msg_server.go
- x/interchainqueries/keeper/keeper.go
- x/interchainqueries/types/message_remove_interchain_query.go
- x/interchaintxs/types/tx.go
- x/interchaintxs/keeper/msg_server.go
- x/feerefunder/keeper/keeper.go
- wasmbinding/bindings/msg.go

Description

This is a list of various minor issues that have been noticed in the Neutron modules code, but they do not pose a security threat.

1. ValidateGenesis() doesn't validate registered queries from the genesis file, if there are any.
2. The error message isn't properly formatted. It should be `len(query.Keys)` instead of just `query.Keys`.
3. clearQueryResult() doesn't copy the `Revision` field from the input `QueryResult`. This field is later used by this query to return results to the smart contracts, which means they will always get 0 as a `Revision`.
4. Returning an error here would produce the same result as the `panic()` call.
5. There is no reason to implement `sdk.Msg` for `MsgRemoveInterchainQueryRequest` and `MsgSubmitTx` with a pointer receiver. Either remove the pointer or check the receiver for nil.
6. The length of the messages slice is already checked in `MsgSubmitTx.ValidateBasic()`.
7. Panics on this line and this line should use `feeInfo.Payer` instead of `receiver`.
8. CosmWasm binding structures AddAdmin and AddAdminResponse should be removed since they are not used anywhere.

Various minor issues in the Neutron smart contracts

ID	IF-NEUTRON-13
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PRs 166 and 43

Involved artifacts

- contracts/dao/voting/neutron-vault/src/contract.rs
- contracts/dao/voting/neutron-voting-registry/src/contract.rs
- contracts/tokenomics/distribution/src/contract.rs
- contracts/tokenomics/distribution/src/msg.rs
- contracts/subdaos/cwd-subdao-timelock-single/src/contract.rs
- contracts/tokenomics/treasury/src/contract.rs

Description

This is a list of various minor issues that have been noticed in the Neutron smart contracts code, but they do not pose a security threat.

1. The query_description() method of the `neutron-vault` smart contract reads data from the `DESCRIPTION` field that is never populated, which would cause an error when the query gets executed. It should read the `Config.description` from the `CONFIG` field instead. This query is called from the `neutron-voting-registry` smart contract by the query_voting_vaults() method, so this query would also return an error.
2. The `neutron-vault` contract uses the wrong name.
3. Both `neutron-vault` and `neutron-voting-registry` contracts allow for easy misconfiguration. For example, if both `owner` and `manager` fields in `InstantiateMsg` are set to `None`, then it would be impossible to update the configuration and, in case of `neutron-voting-registry`, even to execute any contract action.
4. The `neutron-voting-registry` contract should use `checked_add()` instead of `+=` for voting power calculations in both query_voting_power_at_height() and query_total_power_at_height() methods.
5. The `distribution` contract should use `checked_add()` instead of `+=` when calculating the total_shares.
6. `StdError::Overflow` would make more sense here if the `total_shares` were checked for zero value before the start of iteration.
7. StatsResponse and ShareResponse structures are not used anywhere by the contract.
8. Regarding the comment for updating the `subdao` in the `CONFIG` field of the `timelock` smart contract, if this is implemented in the future, it could happen that the new subDAO (which would have its own new proposal module) sends `execute_timelock_proposal()` messages with `proposal_ids` that already exist in the `PROPOSALS` store of the `timelock` contract, which would overwrite the existing proposals that were already passed/overruled/failed. Since there are two queries in the `timelock` contract which are using the `PROPOSALS` field, their results would, in that case, contain mixed results from two different subDAOs.
9. There are some input arguments like `denom`, `min_period` and `vesting_denominator` that are not validated good enough in the instantiate() and execute_update_config() methods of the `treasury` contract.

Various minor issues in the Neutron SDK

ID	IF-NEUTRON-14
Severity	Informational
Impact	None
Exploitability	None
Type	Implementation
Status	Resolved through PR 81

Involved artifacts

- interchain_queries/helpers.rs
- interchain_queries/types.rs

Description

This is a list of various minor issues that have been noticed in the Neutron SDK code, but they do not pose a security threat.

1. get_total_supply_denom() should use the `SUPPLY_PREFIX` constant instead of assuming that the prefix is 1 byte long.
2. KVReconstruct for `Delegations` doesn't check if the `storage_values` length is greater than 0 before accessing the slice.

Various documentation inconsistencies

ID	IF-NEUTRON-15
Severity	Informational
Impact	None
Exploitability	None
Type	Documentation
Status	Unresolved

Involved artifacts

- Offical documentation
- x/feerefunder/keeper/keeper.go

Description

1. The official documentation of the `x/feerefunder` module doesn't state that the module emits any events, but there are events emitted in the following methods: `LockFees()`, `DistributeAcknowledgementFee()` and `DistributeTimeoutFee()`.
2. The documentation for the Remove Interchain Query message doesn't mention that the interchain query can be removed by anyone if the timeout period for results submission has expired.

Recommendation

Update the official documentation.

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.

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

Exploitability Score

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

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

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

Severity Score

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

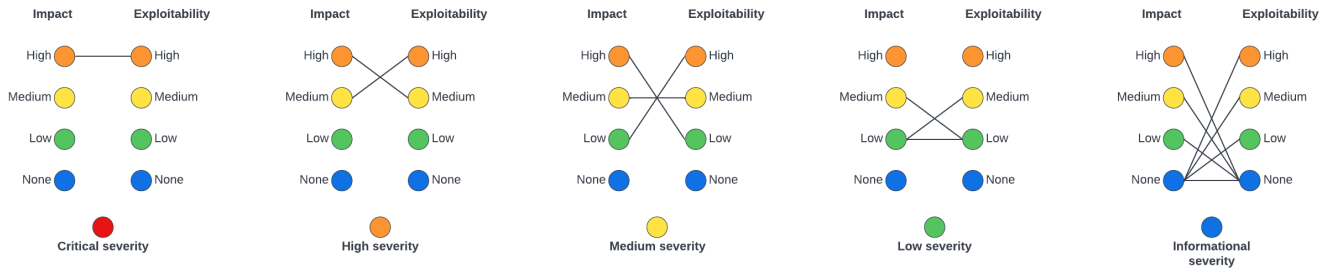

Figure 2: Severity classification

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

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