



Security Audit Report

Neutron: Duality v0.5.0 + integration of Cosmos SDK
0.47

Authors: Ivan Gavran, Andrey Kuprianov, Nikola Jovicevic

Last revised 11 November, 2023

Table of Contents

Audit overview.....	1
The Project	1
Conclusions	1
Audit dashboard.....	3
Target Summary	3
Engagement Summary	3
Severity Summary	3
System Overview and Checked Threats	4
Duality v0.5.0	4
Duality upgrade to Cosmos SDK v0.47.3	4
Axelar GMP middleware	5
Refactoring of cacheCtx logic in Transfer and InterchainTxs modules	5
Updates to x/adminmodule	6
Fork of the Cosmos SDK slashing module	7
Backport of bank-hooks from Osmosis	8
Findings	9
Unbounded iteration over Incentives stakes may lead to chain halt	11
Stealing of user funds via rounding errors in pool and limit order swaps	13
Stealing of arbitrary funds via IBC swaps	16
Quadratic scaling for multi-amount deposits may lead to DOS	20
Contract failures potentially remaining in the state indefinitely	24
Misleading Incentives module user-facing and developer documentation	26
Limit order tranche security rests on a fine interplay of implicit assumptions	28
Incomplete validation of MsgMultiHopSwap may lead to user fund loss and frustration	31
Incomplete validation of pool denominations	33
Miscellaneous security concerns	35
Tick and Fee inputs are not completely validated	37
Miscellaneous code improvements	40
Unutilized event collection	42
Error message referring to a wrong proposal id	43
Multi-message proposals stored as different proposals	45

Duality modules still use deprecated x/params	47
Appendix: Vulnerability Classification	48
Impact Score	48
Exploitability Score	48
Severity Score	49
Disclaimer	51

Audit overview

The Project

In October 2023, [Informal Systems](#) has conducted a security audit for [Neutron](#). The audit covered the following areas:

1. Duality [v0.5.0](#), focusing on changes since the last audit at the version [v0.2.0](#)
2. Duality upgrade to Cosmos SDK [v0.47.3](#), introduced in the [PR#423](#)
3. Integration of Axelar GMP middleware ([x/gmp](#) module at commit [9a208a52](#)) into Duality [v0.5.0](#).
4. Neutron update to Cosmos SDK [v0.47](#), which includes the refactoring of the `cachedCtx` logic, as well as the implementation of parameters handled by each module separately. These changes required checking all the modules of the [neutron](#) repository, especially `x/contractmanager`, `x/interchaintxs` and `x/transfer` due to the refactored logic. Commit hash was [261f47c3](#).
5. Updates to the admin module. Admin module is used for executing privileged actions, such as making software update proposals. The focus of the audit was on handling of the new admin proposals. The audited repository was [admin-module](#) at the commit hash [d5cd5ae2](#).
6. Fork of the Cosmos SDK slashing module, which improves efficiency of the validator's missed block bitmap, and is a backport from Cosmos SDK [v0.50](#). This implied auditing the [slashing module](#) repo at the commit hash [efa64930](#).
7. Backport of bank-hooks from Osmosis, using `TrackBeforeSend` and `BlockBeforeSend` hooks within the [tokenfactory](#) module. Additionally, we checked the [bank](#) module, in which the hooks are triggered.

The audit was performed from October 2, 2023 through October 31, 2023, by the following personnel:

- Ivan Gavran
- Andrey Kuprianov
- Nikola Jovicevic

Conclusions

We performed a thorough review of the project, and found it to be implemented carefully and equipped with a suite of integration tests. Nonetheless, some subtle mistakes were found. In our audit, we report the total of 16 findings: three of them of the [critical](#) severity, and one of the [high](#) severity.

The most crucial necessary improvements are to perform complete validation of Duality input messages, coming both via standard transactions, as well as via IBC: the most severe critical finding, allowing to steal arbitrary amounts of funds, is due to missing validation of inputs. Several other findings, some with undefined consequences, can be prevented when all inputs are properly validated. Another source of concerns seems to be the DOS attacks made possible to scaling the system in various dimensions, such as the number of stakes in the incentives program, or the number of pool shares obtained in the scope of a single transaction. To address this, we recommend to introduce a category of tests that specifically checks the system behavior when scaling various parameters. Furthermore, various mathematical operations, both with the native datatypes as well as with rational numbers, are a source of concern, to which we recommend to pay special attention. Finally, in some cases the unit tests are not kept up to date with the codebase changes, which reduces the overall robustness of the codebase.

Disclaimer wrt. Duality integration into Neutron

We should point out that *we have audited the Duality codebase as a separate project, at [v0.5.0](#)*. No integration of Duality into Neutron codebase was in the scope of the present audit. All findings regarding Duality have been acknowledged, and are valid wrt. [v0.5.0](#); but as has been pointed out by Neutron, two findings ([Unbounded iteration over Incentives stakes may lead to chain halt](#) and [Misleading Incentives module user-facing and developer documentation](#)) don't apply to the Neutron [v1.1.0](#) release, and have been therefore marked **Outdated** in this

report. Also all fixes wrt. other Duality findings are PRs against [neutron-org/neutron](https://github.com/neutron-org/neutron) repository, not against the audited [duality-labs/duality](https://github.com/duality-labs/duality) repository.

Informal Systems has not audited integration of Duality into Neutron v1.1.0, and can provide no guarantee of any kind wrt. this integration. Although in general we consider Neutron and Duality code to be of very high quality, we consider it our responsibility to point out that in particular the late, post-audit and pre-release changes are often the reasons of security vulnerabilities. We recommend Neutron to arrange a separate audit of Neutron v1.1.0 release.

Audit dashboard

Target Summary

- **Type:** Protocol and Implementation
- **Platform:** Go
- **Artifacts:** [admin-module](#), [neutron](#), [cosmos-sdk](#), [duality](#)

Engagement Summary

- **Dates:** 02.10.2023 -- 31.10.2023
- **Method:** Manual code review, protocol analysis

Severity Summary

Finding Severity	#
Critical	3
High	1
Medium	0
Low	5
Unknown	2
Informational	5
Total	16

System Overview and Checked Threats

Duality v0.5.0

Duality at v0.5.0 is about to be integrated into Neutron. We've been conducting a [previous audit of Duality at v0.2.0](#), thus the present audit is to a large degree based on the previous one, and we highlight only the important differences wrt. v0.2.0.

For the overview of the core Duality functionality and its security we refer the reader to the "System Overview" and "Threat Inspection" sections of the [Duality v0.2.0 audit report](#).

From the point of view of *external* system interfaces, the introduction of the [IBC Swap module](#) is a very important change, allowing anyone to piggyback a limit order in the memo field of the ICS-20 fungible token transfer IBC packet. The limit order will then be filled, if possible, immediately upon receiving of the ICS-20 packet: only Fill-Or-Kill limit orders are permitted currently.

The single most important *internal* change is that of **Introduction of Pool IDs**: while in the old version liquidity pools have been identified internally by the combination of pair id, pricing tick and the fee (see [deposit_denom.go at v0.2.0](#)), in the new version each liquidity pool is identified by an integer Pool ID (see [pool_denom.go at v0.5.0](#)): a new id is introduced if and only if tokens are deposited for a token pair at a specific tick and fee layer. **We can highly praise Duality developers for adding this additional level of indirection**, as it represents a significant line of defense wrt. possible attacks: it is now not possible to craft the system inputs (ticks and fees) in order to scoop into a different pool than intended.

Duality upgrade to Cosmos SDK v0.47.3

Duality upgraded its codebase to Cosmos SDK v0.47.3 in the [PR#423](#). Our approach to inspecting the upgrade was to check all the changes from the PR and validate their impact. Furthermore, we matched the changes with the SDK's [upgrade document](#). What made the upgrade easier in this case was that Duality is not a live chain (and thus no live upgrade was necessary) and that Duality's modules are set to be merged with the Neutron chain. Thus, we directed our attention to changes in Duality's modules.

Overall, we find the upgrade to be executed correctly. The only exception we noticed was that Duality did not migrate its params from the `x/params` module to be handled by individual modules, as reported [here](#).

The changes introduced are as follows:

- changes in the `math` module: replacing the member function `toDec` by `NewDecFromInt`
- replacing references to the `tendermint` repository by the ones to the `cometbft` repository
- removing legacy `Route` and `RouterQuerier` functions from `module.go` files
- removing now unnecessary `RandomizedParams` function
- removing writing events to the parent (a fix for this was introduced in SDK 0.46.6)
- replacing references to `gogo/protobuf` by `cosmos/gogoproto`
- using `IBC v7`, which also implied changing the signature of the `SendPacket` function

Unrelated to upgrades in the SDK version, there are a couple of more small changes in the PR:

- the function `GetRewardsEstimate` changes signature.
- slice as an argument to the `Sub` function (which accepts variadic arguments) is expanded
- re-formatting

Axelar GMP middleware

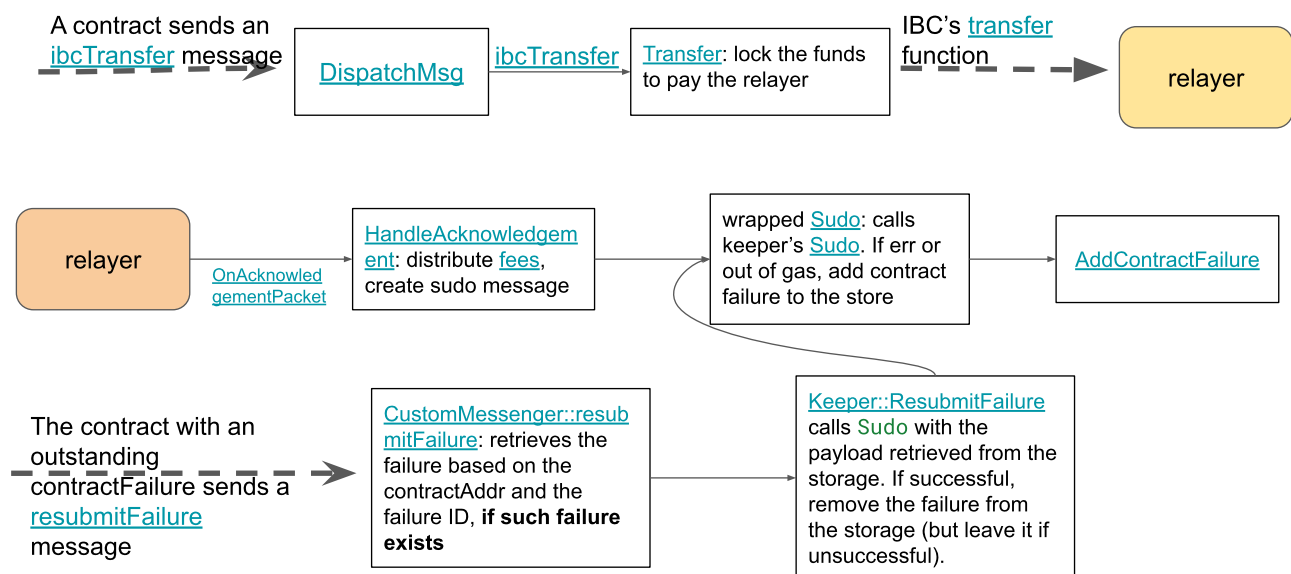
The Axelar GMP middleware is added for purposes of receiving `FungibleTokenPacketData` packets from the Axelar network. Axelar's authentication is not included, so it is basically adapting the packet data before passing it further down the stack. If the memo's payload field is empty, it indicates that the package should not be handled by the GMP middleware. Otherwise, it's all about assigning the memo field with the payload data before continuing with `OnRecvPacket` logic.

Refactoring of cacheCtx logic in Transfer and InterchainTx modules

In order to make the experience of relayers and contract developers easier, the way the gas is spent when smart contracts are handling IBC packets through `Sudo` calls has been refactored.

In a nutshell, every `Sudo` acknowledgement handler can spend at most `SudoCallGasLimit` gas. If the contract spends more gas than allowed (or returns an error), the acknowledgement is still marked as processed, and its payload is saved to the store. This allows the contract to get back to it and process successfully by sending a `ResubmitFailure` message. At the same time, it makes sure that the relayer is compensated fairly and its work is not abused by a smart contract's arbitrary behavior.

In the diagram below, we outlined the steps of the refactored workflow:



1 A workflow of handling the IBC acknowledgement in the transfer module

Inspected Threats

In the audit, we inspected the protocol and its implementation, focusing on these particular threats:

1. Is it possible that an out-of-gas error is raised outside of the wrapped call?
Resolution: Not a viable attack. We inspected all handling outside of the wrapped sudo call (e.g., [here](#)) and concluded that the relayer always has a predictable way of estimating the gas consumption.
2. Is it possible that the stored failures remain in memory indefinitely?
Resolution: This is indeed a possibility, as discussed [here](#). The severity is limited due to high-cost of

executing the attack (either paying for multiple IBC calls, or establishing an adversarial relayer to pocket the cost of IBC calls).

3. Is it possible to store the same failed acknowledgement under multiple `ContractFailure`s?

Resolution: Not a viable attack. The reason is that the iterated handling of the acknowledgement will take the error returned from the Sudo call ([here](#), the return is naked) and propagate it ([here](#), [here](#), and [here](#)), all until the `wasmd`'s `msg_dispatcher`, which will then ignore any state changes (see [here](#)), and thus won't store this new instance of the `ContractFailure`.

For comparison, when calling the same `Sudo` call from the `transfer` module, the error from `Sudo` is [ignored](#). (And the same is true for the `interchaintx` module [here](#).)

4. Sudo routing: could a relayer and a malicious contract steal locked funds of another contract? Concretely, could it happen that contract `A` locks its funds for `TimeoutFee` and `RecvFee`, but the unused part of it (eg, `TimeoutFee` in case of receiving acknowledgement) would be returned to some other contract `B`?
Resolution: No. The sender contract (to which funds need to be returned) is stored [when locking fees](#) under the `PacketId` key. Thus, a malicious relayer cannot manipulate that receiver, since `packetID` is derived from (`packet.SourcePort`, `packet.SourceChannel`, `packet.Sequence`), whose correctness is verified by the light client for the destination chain.

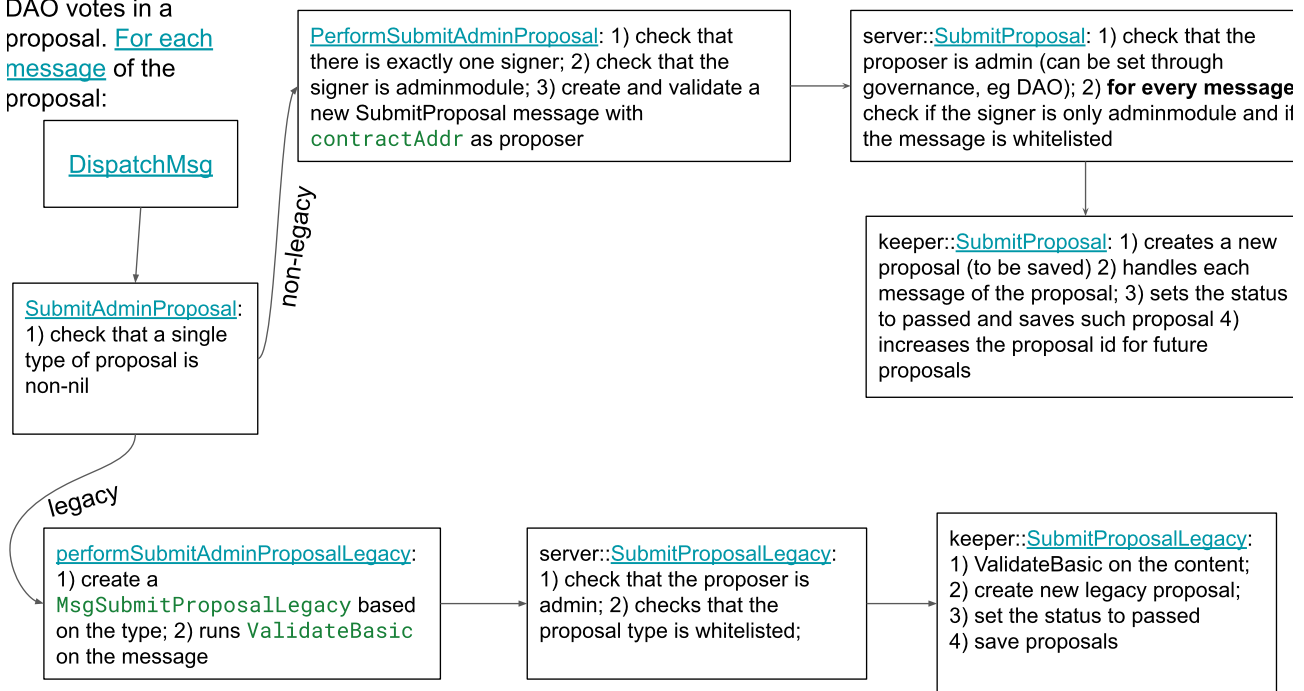
Updates to x/adminmodule

The `adminmodule` has been refactored to allow for handling more general `gov` proposals, which do not limit proposals to a pre-defined set, but rather allow it to be a sequence of (whitelisted) messages. At the same time, it still needs to support legacy proposals.

For a proposal that has passed on the contract side, its messages get executed within the `DispatchMsg` function as either legacy or non-legacy proposals. For legacy ones to pass, the proposer has to be an admin, and the proposal type has to be whitelisted. On the other side, non-legacy proposals are provided as JSON objects, and can be executed and stored only if the admin module is set as a signer, the proposer is an admin, and all the messages are whitelisted. In the end, if everything goes well, both approaches involve creating new proposals, marking them as passed before storing.

In the illustration below we show the workflow of how `adminmodule` handles an incoming proposal.

DAO votes in a proposal. For each message of the proposal:



2 Workflow of the updated adminmodule

During the audit, we inspected the following threats:

- Is it possible to submit an invalid JSON string, or to insert some kind of malicious data in order to trick the system:
 - in a way that an unauthorized user becomes authority;
Resolution: No. There validations for the authority field are in place.
 - if there is a possibility of constructing the JSON message with a legacy proposal pattern.
Resolution: Also no, because invalid JSON messages simply won't be unmarshalled;
- Is there any danger posed by the absence of the nil error check [here](#)?
Resolution: No, because Cosmos SDK ensures that a handler for the message exists.
- Is there a danger of the admin proposal spending too much gas during its execution?
Resolution: No, but even if it comes to 'out of gas' panic, transaction would be reverted without any further damage.

Fork of the Cosmos SDK slashing module

This feature has been backported from Cosmos SDK 0.50 and includes efficiency improvements to the slashing module. It contains a refactored logic for tracking the validator's missed block bitmap, more specifically, the way it is stored and represented.

There is a bitmap for each validator which tells which blocks were missed within the current rolling window. The new approach avoids manipulations on entire bitmaps. It actually breaks them into chunks of fixed size of 1024 bits each. When determining whether a specific block was signed or missed by a validator, the system first **identifies the chunk** where evidence for that particular block is recorded. It then **calculates index** of the appropriate bit within the identified chunk to retrieve the relevant information. On the other hand, when it comes to setting the new bitmap values, it starts with **defining of the appropriate chunk**, and **setting the bit** on the **calculated index** to true.

We paid special attention to functions for setting the bitmap values, those retrieving information from them, as well as those for deleting entire bitmaps. The logic for storing, updating and removing the data is clean and understandable. No issues were found, except that this refactoring is not included in the Cosmos SDK documentation.

Backport of bank-hooks from Osmosis

This backport includes the usage of `BlockBeforeSend` and `TrackBeforeSend` hooks within the `tokenfactory` module triggered on `DelegateCoins`, `UndelegateCoins` and `SendCoin` actions inside the `bank` module. The difference between these two is that for the `TrackBeforeSend` hook a new context with limited gas amounts needs to be set, since it is used in module-to-module transfers. On the other hand, `BlockBeforeSend` is avoided when a transfer is happening between two modules.

Within the `tokenfactory` module, these hooks are implemented in the same function called `callBeforeSendListener`. This function iterates through each provided coin, sending the corresponding sudo messages to appropriate contract addresses. Depending on the `blockBeforeSend` boolean flag, it creates and sends either the `TrackBeforeSendSudoMsg` or `BlockBeforeSendSudoMsg`.

The primary focus during the audit of this part was on the fact that `TrackBeforeSendMsg` is designed for module-to-module transfers because it is using additional gas, while it is also used for `DelegateCoins` and `UndelegateCoins` functions, which are mostly oriented account-to-module and vice versa. This however does not pose a problem because there is a limit set within the child context.

Findings

Title	Type	Severity	Status
Unbounded iteration over Incentives stakes may lead to chain halt	IMPLEMENTATION	4 CRITICAL	OUTDATED
Stealing of user funds via rounding errors in pool and limit order swaps	PROTOCOL IMPLEMENTATION	4 CRITICAL	RESOLVED
Stealing of arbitrary funds via IBC swaps	IMPLEMENTATION	4 CRITICAL	RESOLVED
Quadratic scaling for multi-amount deposits may lead to DOS	IMPLEMENTATION	3 HIGH	RESOLVED
Contract failures potentially remaining in the state indefinitely	PROTOCOL	1 LOW	RISK ACCEPTED
Misleading Incentives module user-facing and developer documentation	DOCUMENTATION	1 LOW	OUTDATED
Limit order tranche security rests on a fine interplay of implicit assumptions	IMPLEMENTATION	1 LOW	RISK ACCEPTED
Incomplete validation of MsgMultiHopSwap may lead to user fund loss and frustration	IMPLEMENTATION	1 LOW	RESOLVED
Incomplete validation of pool denominations	IMPLEMENTATION	1 LOW	RESOLVED
Miscellaneous security concerns	IMPLEMENTATION	0X UNKNOWN	RESOLVED
Tick and Fee inputs are not completely validated	IMPLEMENTATION	0X UNKNOWN	RESOLVED
Miscellaneous code improvements	IMPLEMENTATION PRACTICE	0 INFORMATIONAL	RESOLVED
Unutilized event collection	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED
Error message referring to a wrong proposal id	PRACTICE	0 INFORMATIONAL	RESOLVED
Multi-message proposals stored as different proposals	IMPLEMENTATION	0 INFORMATIONAL	RISK ACCEPTED

Title	Type	Severity	Status
Duality modules still use deprecated x/ params	IMPLEMENTATION	0 INFORMATIONAL	RESOLVED

Unbounded iteration over Incentives stakes may lead to chain halt

Title	Unbounded iteration over Incentives stakes may lead to chain halt
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Threats	Expensive or unbounded computations Denial-Of-Service
Author(s)	Andrey Kuprianov
Status	OUTDATED
Link(s)	

Involved artifacts

- [Duality Incentives module at v0.5.0](#)
 - [msgs.go::MsgStake::ValidateBasic\(\)](#)
 - [msg_server.go::Stake\(\)](#)
 - [stake.go::CreateStake\(\)](#)
 - [hooks.go::AfterEpochEnd\(\)](#)
 - [distribute.go::Distribute\(\)](#)
 - [distributor.go::Distribute\(\)](#)
- [Duality Epochs module at v0.5.0](#)
 - [abci.go::BeginBlocker\(\)](#)

Description

Duality's incentives mechanism works by creating gauges, which describe reward amounts and distribution conditions. Users may then stake their LP tokens, to receive rewards during distribution events. While gauge creation is permissioned (only the admin multisig group can do that), **creation of stakes is permissionless**: all of [msgs.go::MsgStake::ValidateBasic\(\)](#), [msg_server.go::Stake\(\)](#), and [stake.go::CreateStake\(\)](#) check only that the owner field is a correct address, as well as that the supplied coins are valid.

On the other hand, distribution of rewards happens via a chain of calls originating from [abci.go::BeginBlocker\(\)](#), via [hooks.go::AfterEpochEnd\(\)](#), [distribute.go::Distribute\(\)](#), and ending in [distributor.go::Distribute\(\)](#), where, collectively, **iteration over all active gauges, for all stakes** happens. Notice that this includes a large number of operations involving store reads and writes, which consume a lot of both computing resources and gas.

Combination of these two factors (permissionless creation of stakes, and iteration over all stakes from within BeginBlocker), creates an easy possibility for a DOS attack on the chain. In particular, if a large number of stakes is created, and the iteration over them for distribution happens, the number of expensive store operations may be such that, depending on the configuration, either the panic occurs (from within BeginBlocker), leading to the chain halt, or the block gas limit is exceeded leading to rejection of all transactions in the block, or the block production slows down to an unacceptable speed.

Problem Scenarios

The following is a realistic scenario:

1. A large number of stakes are created:
 - either legitimately, by a large number of users interested to participate in the incentives distribution;
 - or maliciously, by a group of actors, seeking to undermine Neutron/Duality operation. Notice that very little funds are required for the attack, as stake creation is permissionless, and every stake may hold a very small number of coins.
2. During the distribution event, at the end of an epoch, iteration over the created stakes leads to one of the aforementioned catastrophic outcomes (chain halt / rejection of legitimate transactions / unacceptable slowdown).

Recommendation

We recommend employ preferably both of the following approaches:

- Limit the number of iterations happening in one block, e.g. via tracking of the gas consumed for distribution, and limiting it to a certain percentage of the block gas limit (e.g. 10%). Maintain the set of stakes to be distributed to in the next block, and update it, as more stakes get processed.
- Introduce a lower limit for accepted stakes (e.g. as number / value of coins). This should both reduce computational requirements for the nodes, as well as make possible attempts of a DOS attack more expensive to execute.

Clarification from Neutron regarding the finding

Not related anymore, since Incentives and Epochs module were removed from Neutron v1.1.0 release.

Stealing of user funds via rounding errors in pool and limit order swaps

Title	Stealing of user funds via rounding errors in pool and limit order swaps
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	PROTOCOL IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Threats	Wrong math calculations
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/344

Involved artifacts

- [Duality Dex module at v0.5.0](#)
 - `types/pool.go::Swap()`
 - `types/limit_order_tranche.go::Swap()`

Description

In both of the `Swap()` functions listed above, the following computation of the amount of the input token is used, as illustrated below on the [example from `pool.go::Swap\(\)`](#):

```
// outAmount will be the smallest value of:  
// a.) The available reserves1  
// b.) The most the user could get out given maxAmountIn0 (maxOutGivenIn1)  
// c.) The maximum amount the user wants out (maxAmountOut1)  
amountMakerOut = utils.MinIntArr(possibleAmountsMakerOut)  
amountTakerIn = math_utils.NewPrecDecFromInt(amountMakerOut).  
    Quo(makerReserves.PriceTakerToMaker).  
    Ceil().  
    TruncateInt()
```


In the above fragment, the amount of the output token is first chosen as the minimum among possible output amounts, and then the input amount as computed by dividing the output amount by `PriceTakerToMaker`, and taking the ceiling of it. The problem arises when the computed *fractional* input amount is small relative to the *integer* ceiling of that. For example:

- Having `amountMakerOut` equal to `1`, and `PriceTakerToMaker` equal to `0.99`, will produce the *fair* fractional input amount of approximately `1.01`, which will get rounded to the integer value of `2`, thus making the effective exchange price unexpected and unfair to the user.
- Even greater (in fact arbitrary) difference in the effective price can be achieved with `PriceTakerToMaker > 1`. E.g. having `amountMakerOut` equal to `1`, and `PriceTakerToMaker` equal to `10`, will produce the *fair* fractional input amount of `0.1`. With large output amounts, e.g. `1000` that would not be a problem, as the resulting input amount would have enough precision. With small amounts though, multiplying `1` by `0.1` will be `0.1`, which gets rounded to `1`, this producing the effective exchange price 10x higher than expected.

Problem Scenarios

The above rounding errors can be exploited maliciously, e.g. by depositing a large number of deposits with seemingly favorable prices, but small reserves. Liquidity iteration over these liquidity pools will exchange liquidity at the price which is completely off wrt. what user has requested, e.g. by supplying the `TickIndexInToOut` of `MsgPlaceLimitOrder`. This is illustrated by the integration tests below.

```
func (s *MsgServerTestSuite) TestLimitOrderSwapAtoBLargeDeposit() {
    s.fundAliceBalances(1000, 0)
    s.fundBobBalances(0, 1000)
    // GIVEN Bob deposits 1000 TokenB at tick 999
    s.bobDeposits(
        NewDeposit(0, 1000, 999, 1),
    )
    // WHEN Alice submits a limit order to swap 1000 TokenA at tick 1000
    s.aliceLimitSells("TokenA", 1000, 1000, types.LimitOrderType_GOOD_TIL_CANCELLED)
    // THEN it succeeds, is fulfilled via swap, and Alice gets back 904 TokenB
    s.assertAliceBalances(0, 904)
}

func (s *MsgServerTestSuite) TestLimitOrderSwapAtoBSmallDeposits() {
    s.fundAliceBalances(1000, 0)
    s.fundBobBalances(0, 1000)
    // GIVEN Bob deposits 1000 TokenB at ticks 0 to 999, 1 token each
    var deposits []*Deposit
    for i := 0; i < 999; i++ {
        deposits = append(deposits, NewDeposit(0, 1, i, 1))
    }
    s.deposits(s.bob, deposits)
    // WHEN Alice submits a limit order to swap 1000 TokenA at tick 1000
    s.aliceLimitSells("TokenA", 1000, 1000, types.LimitOrderType_GOOD_TIL_CANCELLED)
    // THEN it succeeds, is fulfilled via swap, and Alice gets back 500 TokenB
    s.assertAliceBalances(0, 500)
    // THUS The effective price of the swap for Alice was at tick 6920
    // 404 out of 904 of TokenB are stolen from Alice
}
```

```

    // (as 904 TokenB was the least she expected to get)
}

func (s *MsgServerTestSuite) TestLimitOrderSwapBtoALargeDeposit() {
    s.fundAliceBalances(0, 1000)
    s.fundBobBalances(100000, 0)
    // GIVEN Bob deposits 100000 TokenA at tick 30001
    s.bobDeposits(
        NewDeposit(100000, 0, 30001, 1),
    )
    // WHEN Alice submits a limit order to swap 1000 TokenB at tick 30000
    s.aliceLimitSells("TokenB", 30000, 1000, types.LimitOrderType_GOOD_TIL_CANCELLED)
    // THEN it succeeds, is fulfilled via swap, and Alice gets back 20082 TokenA
    s.assertAliceBalances(20082, 0)
}

func (s *MsgServerTestSuite) TestLimitOrderSwapBtoASmallDeposits() {
    s.fundAliceBalances(0, 1000)
    s.fundBobBalances(100000, 0)
    // GIVEN Bob deposits 1000 TokenA at ticks 30001 to 31000, 1 token each
    var deposits []*Deposit
    for i := 30001; i <= 31000; i++ {
        deposits = append(deposits, NewDeposit(1, 0, i, 1))
    }
    s.deposits(s.bob, deposits)
    // WHEN Alice submits a limit order to swap 1000 TokenB at tick 30000
    s.aliceLimitSells("TokenB", 30000, 1000, types.LimitOrderType_GOOD_TIL_CANCELLED)
    // THEN it succeeds, is fulfilled via swap, and Alice gets back 1000 TokenA
    s.assertAliceBalances(1000, 0)
    // THUS The effective price of the swap for Alice was at tick 0
    // 19082 out of 20082 of TokenA are stolen from Alice
    // (as 20082 TokenA was the least she expected to get)
}

```

Notice that the above scenarios represent a viable attack, when a user expects to get the price not worse than e.g. 30000 in the last case, and the attacker deposits a range of liquidity at seemingly more attractive price ticks, but the end result is that the user liquidity gets swapped at the effective price which is much worse than they expected; i.e. the funds are effectively stolen from them.

The exploit may happen also dynamically, via MEV, by monitoring the swap transactions in the mempool, and creating the ranges of liquidity with small amounts immediately before the target transaction gets processed.

Recommendation

The crux of the problem lies in the absence of checks that would compare the expected and the effective prices, thus giving possibility for wide deviations between them. We recommend to introduce the "*maximum spread*" parameter for swap-implying transactions, with reasonable defaults, e.g. similar to [how it's done in Astroport](#), and to control it when performing swaps.

Stealing of arbitrary funds via IBC swaps

Title	Stealing of arbitrary funds via IBC swaps
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	4 CRITICAL
Impact	3 HIGH
Exploitability	3 HIGH
Threats	Incomplete input validation
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/359

Involved artifacts

- [Duality IBCSwap module at v0.5.0](#)
 - [types/swap.go::Validate\(\)](#)

Description

[IBCSwap module](#) allows to piggyback a Fill-Or-Kill limit order on the IBC's ICS-20 token transfer message, with `MsgPlaceLimitOrder` embedded into `SwapMetadata`, in turn embedded into the `Memo` field of `FungibleTokenPacketData`. The data contained in the embedded `MsgPlaceLimitOrder` is validated via function `swap.go::Validate()`. The problem is, when `MsgPlaceLimitOrder` comes via IBC, there is no implicit connection between the originator of the message with the `Creator` field of that message, as opposed to it being maintained via `MsgPlaceLimitOrder::GetSigners()` when the message arrives via a standard transaction. The net result is that **any address can be put into the `Creator` field, allowing to steal funds of an arbitrary user.** But the problem is actually much more serious: **arbitrary amount of funds in any denomination can be stolen from any account.** The reason is that during the validation in `swap.go::Validate()`, no connection between the funds received via fungible token transfer and the piggybacked `MsgPlaceLimitOrder` is asserted, allowing to provide completely arbitrary values for all fields.

Problem Scenarios

The below test (to be placed into [tests/ibc/swap_test.go](#)) demonstrates how funds can be stolen from an arbitrary user account on Duality.

```
var AnyDualityAddress =
sdk.MustAccAddressFromBech32("dual1afk9zr2hn2jsac63h4hm60vl9z3e5u69gndzf7c99cqge3vzwj
zsyf0vr6")

func (s *IBCTestSuite) TestIBCSwapMiddleware_StealFunds() {
    // Some user sends funds to AnyDualityAddress
    s.IBCTransferProviderToDuality(
        s.providerAddr,
        AnyDualityAddress,
        nativeDenom,
        ibcTransferAmount,
        "",
    )

    // Send an IBC transfer from provider to Duality, so we can initialize a pool
    with the IBC denom token + native Duality token
    s.IBCTransferProviderToDuality(
        s.providerAddr,
        s.dualityAddr,
        nativeDenom,
        ibcTransferAmount,
        "",
    )

    // Assert that the funds are gone from the acc on provider and present in the acc
    on Duality
    newProviderBalNative :=
genesisWalletAmount.Sub(ibcTransferAmount).Sub(ibcTransferAmount)
    s.assertProviderBalance(s.providerAddr, nativeDenom, newProviderBalNative)

    s.assertDualityBalance(AnyDualityAddress, s.providerToDualityDenom,
ibcTransferAmount)
    s.assertDualityBalance(s.dualityAddr, s.providerToDualityDenom,
ibcTransferAmount)

    // deposit stake<>ibcTransferToken to initialize the pool on Duality
    depositAmount := sdk.NewInt(100_000)
    s.dualityDeposit(
        nativeDenom,
        s.providerToDualityDenom,
        depositAmount,
        depositAmount,
        0,
        1,
        s.dualityAddr)

    // Assert that the deposit was successful and the funds are moved out of the
    Duality user acc
    s.assertDualityBalance(s.dualityAddr, s.providerToDualityDenom, sdk.ZeroInt())
    postDepositDualityBalNative := genesisWalletAmount.Sub(depositAmount)
```

```

s.assertDualityBalance(s.dualityAddr, nativeDenom, postDepositDualityBalNative)

// Send an IBC transfer from providerChain to Duality with packet memo containing
the swap metadata
swapAmount := sdk.NewInt(100000)
expectedOut := sdk.NewInt(99_990)

metadata := swaptypes.PacketMetadata{
  Swap: &swaptypes.SwapMetadata{
    MsgPlaceLimitOrder: &dextypes.MsgPlaceLimitOrder{
      // use AnyDualityAddress as Creator to steal funds from it
      Creator:      AnyDualityAddress.String(),
      Receiver:     s.dualityAddr.String(),
      TokenIn:      s.providerToDualityDenom,
      TokenOut:     nativeDenom,
      AmountIn:     swapAmount,
      TickIndexInToOut: 1,
      OrderType:    dextypes.LimitOrderType_FILL_OR_KILL,
    },
    Next: nil,
  },
}

metadataBz, err := json.Marshal(metadata)
s.Require().NoError(err)

s.IBCTransferProviderToDuality(
  s.providerAddr,
  s.dualityAddr,
  nativeDenom,
  ibcTransferAmount,
  string(metadataBz),
)

// Check that the funds are moved out of the acc on providerChain
s.assertProviderBalance(
  s.providerAddr,
  nativeDenom,
  newProviderBalNative.Sub(ibcTransferAmount),
)

// Check that the IBC-transferred funds are present in the acc on Duality
s.assertDualityBalance(s.dualityAddr, s.providerToDualityDenom,
ibcTransferAmount)

// Check that the swap funds are also present in the acc on Duality (i.e. stolen)
s.assertDualityBalance(s.dualityAddr, nativeDenom,
postDepositDualityBalNative.Add(expectedOut))

// The funds are indeed stolen from AnyDualityAddress
s.assertDualityBalance(AnyDualityAddress, s.providerToDualityDenom,
sdk.ZeroInt())
}

```

For simplicity of integration, we slightly modified the already present test `TestIBCSwapMiddleware_Success`. The test demonstrates how the funds previously transferred via IBC can be stolen. As we have noted previously though, arbitrary amount of funds in any denomination can be stolen from any account.

Recommendation

We recommend to extend validation of `SwapMetadata` in the following ways:

- Check that the address represented by the `Creator` field of `MsgPlaceLimitOrder` is equal to the address represented by the `Receiver` field of `FungibleTokenPacketData`
- Check that the `TokenIn` field of `MsgPlaceLimitOrder` is the same as the token denomination that is received (after ICS-20 encoding/decoding) by `Receiver` of `FungibleTokenPacketData`
- Check that the `AmountIn` of `MsgPlaceLimitOrder` doesn't exceed the `Amount` of `FungibleTokenPacketData`.

In general, we recommend taking extreme care with the messages received via IBC, and think through other possible ways to adversarially employ them, which are not necessarily covered by the above list of checks.

Quadratic scaling for multi-amount deposits may lead to DOS

Title	Quadratic scaling for multi-amount deposits may lead to DOS
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	3 HIGH
Impact	2 MEDIUM
Exploitability	3 HIGH
Threats	Expensive or unbounded computations
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/342

Involved artifacts

- [Duality Dex module at v0.5.0](#)
 - [keeper/core.go::DepositCore\(\)](#)

Description

The `DepositCore()` function is executed upon processing of `MsgDeposit`, which contains multiple amounts to deposit at different liquidity pools. The function iterates over all supplied amounts, and computes the issued pool shares to be sent back to the user. The offensive line is [L81](#):

```
sharesIssued = sharesIssued.Add(outShares)
```

In the above, `sharesIssued` is of type `sdk.Coins`, and its `Add` function is implemented as follows:

```
func (coins Coins) Add(coinsB ...Coin) Coins {  
    return coins.safeAdd(coinsB)  
}
```

```

func (coins Coins) safeAdd(coinsB Coins) (coalesced Coins) {
    if !coins.isSorted() {
        panic("Coins (self) must be sorted")
    }
    if !coinsB.isSorted() {
        panic("Wrong argument: coins must be sorted")
    }

    uniqCoins := make(map[string]Coins, len(coins)+len(coinsB))
    // Traverse all the coins for each of the coins and coinsB.
    for _, cL := range []Coins{coins, coinsB} {
        for _, c := range cL {
            uniqCoins[c.Denom] = append(uniqCoins[c.Denom], c)
        }
    }

    for denom, cL := range uniqCoins { //nolint
        comboCoin := Coin{Denom: denom, Amount: NewInt(0)}
        for _, c := range cL {
            comboCoin = comboCoin.Add(c)
        }
        if !comboCoin.IsZero() {
            coalesced = append(coalesced, comboCoin)
        }
    }
    if coalesced == nil {
        return Coins{}
    }
    return coalesced.Sort()
}

func (coins Coins) isSorted() bool {
    for i := 1; i < len(coins); i++ {
        if coins[i-1].Denom > coins[i].Denom {
            return false
        }
    }
    return true
}

```

As can be seen, every addition happening on L81 results in multiple traversals over all coins in `sharesIssued`, and also involves allocation of additional data structures, thus resulting in **quadratic scaling of the `DepositCore` runtime** (because it invokes `sdk.Coins Add()` for each issued share). We have demonstrated this with a small test:

```

func (s *MsgServerTestSuite) TestDepositManyAmounts() {
    s.fundAliceBalances(100000, 0)
    // Alice deposits many amounts TokenA, 1 token each
    var deposits []*Deposit
    for i := 1; i <= 1000; i++ {
        deposits = append(deposits, NewDeposit(1, 0, i, 1))
    }
    s.deposits(s.alice, deposits)
}

```



```
}

```

Running the above test with the number of amounts in `MsgDeposit` doubling each time clearly demonstrates the described quadratic behavior:

Num amounts	Runtime (sec)
1000	0.20
2000	0.67
4000	2.54
8000	10.29
16000	42.67

Problem Scenarios

Duality is expected to operate in automated setups where users or tools can spread their liquidity over multiple pools, and the amounts of tokens in a single `MsgDeposit` may be quite large. In the moderate case, having such quadratic behavior will slow down the network; in the worst case the slowdown may become so substantial that it may lead to DOS.

Recommendation

We recommend to implement an easy fix which replaces the quadratically scaling solution using `sdk.Coins.Add()` with a single sorting and addition pass over all issued shares at the end of `DepositCore` function. The prototype of such a solution is outlined below.

- After L37 add

```
shares := make([]sdk.Coin, 0, len(amounts0))
```

- Replace L82, `sharesIssued = sharesIssued.Add(outShares)` with

```
shares = append(shares, outShares)
```

- After L99, add

```
sort.SliceStable(shares, func(i, j int) bool {
    return shares[i].Denom < shares[j].Denom
})

lastShareDenom := ""
```

```

for _, share := range shares {
    if lastShareDenom != share.Denom {
        sharesIssued = append(sharesIssued, share)
    } else {
        sharesIssued[len(sharesIssued)-1].Add(share)
    }
    lastShareDenom = share.Denom
}

```

Implementing the above changes results in reducing quadratic scaling of the `DepositCore` runtime to linear with a small coefficient, as is shown in the table below.

Num amounts	Old runtime (sec)	New runtime (sec)
1000	0.20	0.07
2000	0.67	0.11
4000	2.54	0.22
8000	10.29	0.42
16000	42.67	0.86

Contract failures potentially remaining in the state indefinitely

Title	Contract failures potentially remaining in the state indefinitely
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	PROTOCOL
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Threats	Statebloat
Author(s)	Ivan Gavran
Status	RISK ACCEPTED
Link(s)	

Involved artifacts

- [neutron/x/contractmanager/ibcmiddleware.go](https://github.com/neutron/x/contractmanager/ibcmiddleware.go)

Description

If handling of a Sudo call fails, [a new contract failure is added](#). It is [removed](#) only upon a successful resubmission of the failure and at no other place.

Problem Scenarios

The only contract that can execute the resubmission is the offending contract. If it happens that the contract does not implement/support `ResubmitFailure` or it repeatedly runs out of gas and gives up, the state space will get used without a chance of clearing it. A malicious contract could bloat the state space and make the sync/state export/state import slow.

This kind of attack is limited by the high cost the contract has to pay for submitting multiple IBC calls and failing to execute them.

Recommendation

We recommend to give authority to a pre-defined contract to clear:

- a) all failures connected to a particular `contractAddr`

b) all failures older than a particular block. This would require to group failures in block buckets (where a bucket would contain multiple blocks) and reflect the bucket structure in the way they are stored.

Clarification from Neutron regarding the finding

Since contract must pay fee to issue txs from Neutron's ICTX and Transfer module, we consider a risk of this attack as very low.

Misleading Incentives module user-facing and developer documentation

Title	Misleading Incentives module user-facing and developer documentation
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	DOCUMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	3 HIGH
Threats	Code-documentation alignment
Author(s)	Andrey Kuprianov
Status	OUTDATED
Link(s)	

Involved artifacts

- [Duality Incentives module at v0.5.0](#)
 - [README GAUGES.md](#)
 - [msg_server.go](#)

Description

Examples from user-facing documentation, [README GAUGES.md](#):

Anyone can create gauge and add rewards to the gauge, there is no way to take it out other than distribution.

- Not true see [msg_server.go::CreateGauge\(\)](#): only the configured authority (Duality admin multisig) can create gauges;

Staked tokens can be of any denomination, including LP tokens (gamm/pool/x), IBC tokens (tokens sent through IBC such as ibc/27394FB092D2ECCD56123C74F36E4C1F926001CEADA9CA97EA622B25F41E5EB2), and native tokens (such as ATOM or LUNA).

- Not true; only Duality LP tokens are accepted: [stake.go::CreateStake\(\)](#) calls [stake.go::ValidateBasic\(\)](#), which makes sure that the coins have Duality pool denomination.

In general, the documentation in the Incentives README looks to be copied over from Osmosis Incentives module without any adaptations, which manifests itself in many Osmosis-specific references and names.

Examples from developer-facing documentation, [msg_server.go](#)

// StakeTokens stakes tokens in either two ways.

// 1. Add to an existing stake if a stake with the same owner and same duration exists.

```
// 2. Create a new stake if not.  
// A sanity check to ensure given tokens is a single token is done in ValidateBaic.  
// That is, a stake with multiple tokens cannot be created.  
• 1 is not true; a new stake is always created, see stake.go::CreateStake\(\)  
• "a stake with multiple tokens cannot be created": not true; msgs.go::MsgStake::ValidateBasic\(\) checks only  
  that all coins in the stake are positive, while stake.go::ValidateBasic\(\) checks only coin denominations for all  
  coins in the stake.
```

Problem Scenarios

Misleading documentation can be dangerous. In the above:

- misleading user-facing documentation may lead to user frustration the least (e.g. when an unauthorized user tries to create a gauge), or, worse, to wrong usage of the system.
- misleading developer-facing documentation may provide the developers with false assumptions on the behavior of other components, and as a result, lead to them writing the code resting on those incorrect assumptions; in the worst case leading to security vulnerabilities. E.g. in the example above, a developer may write a code that assumes that only a single coin can be sent for a stake.

Recommendation

We recommend carefully examine all module documentation, both user- and developer-facing, and to make sure it accurately reflects what happens in reality.

Clarification from Neutron regarding the finding

Not related anymore, since Incentives and Epochs module were removed from Neutron v1.1.0 release.

Limit order tranche security rests on a fine interplay of implicit assumptions

Title	Limit order tranche security rests on a fine interplay of implicit assumptions
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	1 LOW
Impact	2 MEDIUM
Exploitability	1 LOW
Threats	Implicit security assumptions
Author(s)	Andrey Kuprianov
Status	RISK ACCEPTED
Link(s)	

Involved artifacts

- [Duality Dex module at v0.5.0](#)
 - [keeper/limit_order_tranche.go](#)
 - [types/limit_order_tranche.go](#)

Description

When processing `MsgPlaceLimitOrder` :

- function `PlaceLimitOrderCore()` calls function `GetOrInitPlaceTranche()`,
- which in the default branch calls function `GetPlaceTranche()`,
- which iterates over all liquidity for the provided trade pair and tick, and determines whether there is already a tranche by calling function `IsPlaceTranche()`, which is this curious little beast below:

```
func (t LimitOrderTranche) IsPlaceTranche() bool {  
    return t.ReservesMakerDenom.Equal(t.TotalMakerDenom)  
}
```

The first thing to notice in the above sequence of calls is the completely misleading naming, like `GetPlaceTranche()` and `IsPlaceTranche()` which, from their names, seem to indicate it's a simple matter of whether smth. with the provided parameters exists or not in the state. What they do in fact is return the existing tranche only if its components `ReservesMakerDenom` and `TotalMakerDenom` are equal.

The net result is that when the above condition holds, the limit orders of different users are coalesced into the same limit order tranche; and when not, a new tranche is created for the newly placed limit order.

Having investigated how the variables `ReservesMakerDenom` and `TotalMakerDenom` are used, the following conclusions can be made:

- `TotalMakerDenom` is only incremented, when new funds are added to the limit order tranche. It is also heavily used in computations via functions `RatioFilled()` and `AmountUnfilled()`, when e.g. the limit order position of a user is canceled (function `RemoveTokenIn()`), or funds are withdrawn (function `Withdraw()`).
- `ReservesMakerDenom` is incremented in synchrony with `TotalMakerDenom` when new funds are added to the limit order tranche, and is decremented when the limit order position of a user is canceled (function `RemoveTokenIn()`), or a swap via limit order tranche is performed (function `Swap()`).

Thus, *currently the logic of updating the above two variables doesn't seem to pose an immediate security risk*. We should note though that:

- The fact that funds of multiple users are coalesced into the same limit order tranche makes it an attractive target for attacks.
- **The heavy (implicit!) security assumption is that no new funds can be added to the tranche after either a swap or a cancel operation is performed by any user participating in this tranche.**
- The above security assumption rests on how the (inadequately named) functions `GetPlaceTranche()` and `IsPlaceTranche()` behave, and in particular the really crucial part of limit tranche security is the interplay between `IsPlaceTranche()` being implemented as `t.ReservesMakerDenom.Equal(t.TotalMakerDenom)`, and of the two variables involved being updated and used in a very specific way.

Problem Scenarios

It seems entirely possible that within the normal system evolution, e.g. when new developers take care of the system maintenance, the following may happen:

- Either the behavior of `GetPlaceTranche()` and `IsPlaceTranche()` is changed in such a way that it corresponds to the (inadequate) naming; e.g. by simply returning the limit order tranche with the given parameters from the state.
- Or the limit order tranche mathematical logic is updated in such a way that `ReservesMakerDenom` and `TotalMakerDenom` get out of sync, and allowing to attack the system by a sequence of calls in which e.g. `alice` first places some funds into a limit order tranche, then `bob` places funds, then `alice` cancels her position, or swaps and withdraws, and newly deposits, to manipulate the internal tranche variables in such a way that `bob`'s funds can be stolen.

Recommendation

No immediate action seems to be necessary for the present release. Before the next release, we recommend to:

- Explicitly document the security assumptions wrt. limit order tranches; e.g. that it's crucial to maintain the invariant **"No new funds should be added to the limit order tranche after any swap or cancel"**

- Fix the naming and usage of the functions `GetPlaceTranche()` and `IsPlaceTranche()` , as well as write a proper documentation for them in order to reflect the crucial role they play in enforcing the above security assumption.

Clarification from Neutron regarding the finding

Decided not to implement, since it's not a security concern for the current release.

Incomplete validation of MsgMultiHopSwap may lead to user fund loss and frustration

Title	Incomplete validation of MsgMultiHopSwap may lead to user fund loss and frustration
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	2 MEDIUM
Threats	Incomplete input validation
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/357

Involved artifacts

- Duality Dex module at v0.5.0
 - `types/message_multi_hop_swap.go::ValidateBasic()`
 - `keeper/core.go::MultiHopSwapCore()`

Description

`ValidateBasic()` of `MsgMultiHopSwap` performs a very incomplete validation of its inputs wrt. supplied routes. In particular:

- Empty routes are accepted, and `MultiHopSwapCore()` panics:

```
func (s *MsgServerTestSuite) TestMultiHopNoTokens() {
    s.fundAliceBalances(100, 0)

    // WHEN alice mistakenly multihopswaps with no tokens,
    route := [][]string{{}}
    s.aliceMultiHopSwaps(route, 100, math_utils.MustNewPrecDecFromStr("0.1"),
false)
```

```

    // THEN the swap proceeds, and panics in core.go:241
    // runtime error: index out of range [0] with length 0
    // initialInCoin := sdk.NewCoin(routes[0].Hops[0], amountIn)
}

```

- Routes with a single token (no hops) are accepted; `MultiHopSwapCore()` proceeds and fails when attempting to send empty coins:

```

func (s *MsgServerTestSuite) TestMultiHopNoHops() {
    s.fundAliceBalances(100, 0)

    // WHEN alice mistakenly multihopswaps A (no hops),
    route := [][]string{{"TokenA"}}
    s.aliceMultiHopSwaps(route, 100, math_utils.MustNewPrecDecFromStr("0.1"),
false)

    // THEN the swap proceeds, and fails deep inside with the following error
    // Error: Expected nil, but got: <nil>: invalid coins [cosmos/cosmos-
sdk@v0.47.4/x/bank/keeper/send.go:236]
}

```

- Routes with cycles (which are nonsensical) are accepted, proceed to completion, and the user ends up with less of the same tokens they've had before:

```

func (s *MsgServerTestSuite) TestMultiHopCycle() {
    s.fundAliceBalances(100, 0)

    // GIVEN liquidity in pool A<>B,
    s.SetupMultiplePools(
        NewPoolSetup("TokenA", "TokenB", 100, 100, 10000, 200),
    )

    // WHEN alice mistakenly multihopswaps A -> B -> A,
    route := [][]string{{"TokenA", "TokenB", "TokenA"}}
    s.aliceMultiHopSwaps(route, 100, math_utils.MustNewPrecDecFromStr("0.1"),
false)

    // THEN alice gets back 95 TokenA
    s.assertAccountBalanceWithDenom(s.alice, "TokenA", 95)
}

```

Problem Scenarios

It is possible that users may mistype the inputs, or that invalid inputs are generated automatically by some frontend; accepting nonsensical data, and returning back incomprehensible errors will lead to user frustration the least, or even to loss of user funds, such as the case of cyclic routes.

Recommendation

We recommend to perform as complete validation of the user inputs as possible, and to return informative error messages in case of invalid inputs.

Incomplete validation of pool denominations

Title	Incomplete validation of pool denominations
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	1 LOW
Impact	1 LOW
Exploitability	1 LOW
Threats	Incomplete input validation
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/356

Involved artifacts

- [Duality Dex module at v0.5.0](#)
 - `types/pool_denom.go::ValidatePoolDenom()`
- [\[Duality Incentives module at v0.5.0\]](#)
 - `types/stake.go::ValudateBasic()`

Description

In Duality, liquidity pools are identified by an integer id of type `uint64`, and for each pool its shares are issued as coins with the denomination `duality/pool/{id}`, as defined by [PoolDenomRegexp](#):

```
PoolDenomPrefix    = "duality/pool/"
PoolDenomRegexpStr = "^" + PoolDenomPrefix + `(\d+)` + "$"
```

When receiving from outside of Duality coins which are supposed to represent pool shares, their validation is performed in two ways:

- either by function [ParsePoolIDFromDenom\(\)](#), which is called from [GetPoolMetadataByDenom\(\)](#); the latter one is used in many places throughout the codebase;
- or by function [ValidatePoolDenom\(\)](#), which is used in a few queries, but also most notably from [incentives/types/stake.go::ValudateBasic\(\)](#).

The problem is that while the first function performs complete validation, including validating the type bounds on the pool identifier, the second one validates only that the denomination adheres to the regular expression, ignoring the type bounds:

```
func ValidatePoolDenom(denom string) error {  
    if !PoolDenomRegexp.MatchString(denom) {  
        return ErrInvalidPoolDenom  
    }  
    return nil  
}
```

This provides an opportunity for incorrect shares denominations to pass through the system boundary, and potentially cause harm.

Problem Scenarios

We have identified one case when the above incorrect denomination does indeed enter the system, as exposed by the below small test (to be placed in [incentives/keeper/stake_test.go](#)); here we provide the pool id a value that is 1 greater than the maximum of type `uint64`:

```
func (suite *KeeperTestSuite) TestStakeInvalidDenom() {  
    shares := sdk.Coins{sdk.NewInt64Coin("duality/pool/18446744073709551616", 1)}  
    suite.SetupStake(suite.SetupAddr(0), shares, 0)  
}
```

The above test fails with `spendable balance is smaller than 1duality/pool/18446744073709551616: insufficient funds` in [incentives/keeper/stake.go#L210](#):

```
if err := k.bk.SendCoinsFromAccountToModule(ctx, owner, types.ModuleName,  
    stake.Coins); err != nil {  
    return nil, sdkerrors.Wrap(sdkerrors.ErrInvalidRequest, err.Error())  
}
```

I.e. it fails *after* passing validation for the `Stake` message, *deep inside* the function `CreateStake`, when trying to transfer funds from the user account to the system.

While, to the best of our understanding, all current usages of the aforementioned incompletely validating function `ValidatePoolDenom()` seem not to pose security risks, the presence of such function in the codebase poses significant future risks, when it might be used on a more critical code path, creating to a security vulnerability.

Recommendation

We recommend to modify the function `ValidatePoolDenom()` in order to also validate type bounds on the embedded pool id, as it is done in the function `ParsePoolIDFromDenom()`; ideally both functions should reuse the same core parsing and validation functionality.

Miscellaneous security concerns

Title	Miscellaneous security concerns
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	0X UNKNOWN
Impact	0X UNKNOWN
Exploitability	0X UNKNOWN
Threats	Incomplete input validation
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/361

We list here various security-related concerns that, to the best of our limited knowledge (due to the limited project timeline), don't pose immediate security threats, but may lead to security vulnerabilities or problems if not taken care of properly.

Duality integration (duality-labs / duality repo at v.0.5.0)

1. `AppAuthority` is hard-coded in `app/app.go` to the “address of the admin multisig group”; it is used throughout many module instantiations. When integrating with Neutron, this needs to be taken care of, in particular because it plays a crucial role in some cases (e.g. in the Incentives module, only this account is allowed to create gauges).
2. In the `epochs` module, **all tests that check hooks panicking or erroring are disabled**. While the panic/error catching functionality of epoch hooks seems to work as of now, having these tests disabled poses significant future security risks: if this functionality stops to work, and this goes unnoticed, the chain may stop in case of a hook panic (which will happen in the `BeginBlocker`).
3. Function `GetLimitOrderTrancheByKey()` may potentially return `nil`, `true`, on **line 104**, in case function `GetLimitOrderTranche()` returns `nil`. This may happen if the liquidity at the specified tick contains not a limit order tranche, but other kind of liquidity. If this happens, it may lead to severe consequences, one example being that the function `PurgeExpiredLimitOrders()` will panic on **line 147**. To the best of our knowledge, it's not possible for other kind of liquidity to be present when the liquidity is requested via function `GetLimitOrderTrancheByKey()`, but if we are mistaken, or the situation changes in the future, this may lead to security vulnerabilities. We recommend to change **line 104** of that function to return `nil`, `false`, in case `GetLimitOrderTranche()` returns `nil`.

4. Function `SwapExactAmountIn()` doesn't behave as its name indicates: it doesn't swap the exact amount in, but the max amount such that the swap would not give any additional out coins. The remaining amount of in-coins, which is not swapped, is silently ignored; only the amount of out-coins is returned. The function is used in `MultihopStep()` to perform a single step of a multi-hop route; this function is in turn invoked from `RunMultihopRoute()`. Notice that in that function the route price is calculated by dividing the total amount of out-coins by the total amount of in-coins, ignoring the unswapped coins. The price is compared to the limit price provided by the user, so the constraints provided by the user are still satisfied. The user could have obtained a better price if only the swapped coins are taken from their account; instead the unswapped rest of the in-coins are transferred to the system, which is slightly unfair to the end user, but not critical. What's more important, is that the presence of a function with misleading name (`SwapExactAmountIn()`) creates a danger of employing it in the future in other scenarios, which may lead to security vulnerabilities.
5. In our previous audit report of Duality v.0.2.0, there is a finding "Incomplete validation of `MsgDeposit`": `MsgDeposit` contains 5 repeated fields, in particular `Options`, with the length of all arrays supposed to be equal, but the validation code of `MsgDeposit::ValidateBasic` checks that only 4 of those are of equal length (the length of `Options` array is not validated). This finding is still valid for the present v.0.5.0.

Tick and Fee inputs are not completely validated

Title	Tick and Fee inputs are not completely validated
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	0X UNKNOWN
Impact	3 HIGH
Exploitability	0X UNKNOWN
Threats	Incomplete input validation
Author(s)	Andrey Kuprianov
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/360

Involved artifacts

- [Duality Dex module at v0.5.0](#)
 - `tx.proto`
 - `types/message_deposit.go`
 - `types/message_withdrawl.go`
 - `types/message_place_limit_order.go`
- [Duality Incentives module at v0.5.0](#)
 - `types/msgs.go`

Description

The present finding is tightly related to the finding “*Tick and Fee inputs are not validated*” from [our previous audit report of Duality v.0.2.0](#); while some improvements have been made wrt. the previous finding, we should state that the problem still persists to the large degree.

Many Duality transactions accept tick indexes (as `int64`) and fees (as `uint64`): see e.g. `MsgDeposit` and `MsgWithdrawal` in `tx.proto`. In the implementation, semantically, these values are assumed to belong to the range `[-559680, 559680]`, see [price.go](#). The problem is that these constraints are either not enforced at all at the system boundary, or not validated completely, and thus the values that enter the system may be anything in range `[-9223372036854775808, 9223372036854775807]` for `int64`, and `[0, 18446744073709551615]` for `uint64`.

Allowing incorrect values to pass the system boundary creates a high risk of security attacks.

Below are some of places in the codebase where the inputs are either not validated, or validated incompletely.

Function `IsTickOutOfRange()` performs incorrect validation

`IsTickOutOfRange()` is implemented as follows:

```
func IsTickOutOfRange(tickIndex int64) bool {
    return tickIndex > 0 && uint64(tickIndex) > MaxTickExp
}
```

As can be seen, the function doesn't properly validate that the ticks belong to the range `[-559680, 559680]`: it accepts any negative `tickIndex`, i.e. returns `false` for any value in the range `[-9223372036854775808, 559680]`.

This function is used in function `CalcPrice()`, which is in turn employed in many places to validate correctness of inputs, as well as directly in `MsgCreateGauge::ValidateBasic()`.

MsgDeposit

[Validation of MsgDeposit](#) doesn't check constraints on either ticks or fees. Incorrect values enter the system, and are later presumably are filtered out deep inside of `DepositCore()`. In particular, the fees are checked via [invoking the function ValidateFee](#), which checks that fees belong to one of fee tiers. Ticks, on the contrary, are not validated at all, and enter the computations like in function `GetPool()`:

```
feeInt64 := utils.MustSafeUint64ToInt64(fee)

id0To1 := &types.PoolReservesKey{
    TradePairID:      types.NewTradePairIDFromMaker(pairID, pairID.Token1),
    TickIndexTakerToMaker: centerTickIndexNormalized + feeInt64,
    Fee:              fee,
}
```

Notice that the expression `centerTickIndexNormalized + feeInt64` may overflow.

MsgWithdrawal

[Validation of MsgWithdrawal](#) doesn't impose any constraints on either tick or fee values, which enter the system, and are involved in numerical computations with the possibility of over/underflow. The main (and probably the only) barrier that stands on the path of exploiting these for attack purposes is the right decision to [track pools via pool IDs](#), not via a combination of tick index and fees, as in v.0.2.0. With that addition, withdrawing from a pool requires that this pool has been previously added via `MsgDeposit`; without tracking pool IDs it might be possible to withdraw from a pool that the user doesn't own. Still, with a large number of corner cases, it's not possible to guarantee the absence of attacks involving out-of-bounds tick and fee values.

MsgPlaceLimitOrder

[Validation of MsgPlaceLimitOrder](#) doesn't impose any constraints on the `TickIndexInToOut` value, which thus enters the system. A partial validation is performed via [calling the function CalcPrice\(\)](#), which calls `IsTickOutOfRange()`, which, [as we've already explained above](#), performs validation incorrectly.

MsgCreateGauge

`MsgCreateGauge::ValidateBasic()` performs validation of tick indices using the aforementioned function `IsTickOutOfRange()`, which is incorrect.

Problem Scenarios

Whenever incorrect data enters the system, it's hard to predict how it will interact with the other system data present, or with the algorithms employed. For example, `PoolReservesKey::Counterpart()` looks like this:

```
func (p PoolReservesKey) Counterpart() *PoolReservesKey {
    feeInt64 := utils.MustSafeUint64ToInt64(p.Fee)
    return &PoolReservesKey{
        TradePairID:      p.TradePairID.Reversed(),
        TickIndexTakerToMaker: p.TickIndexTakerToMaker*-1 + 2*feeInt64,
        Fee:               p.Fee,
    }
}
```

I.e. it employs quite heavy numerical computations with native types, all subexpressions of which may over-/underflow when the constituent variables are close to type bounds. This function is used e.g. in `GetPool()`, or in `liquidity_iterator.go::WrapTickLiquidity()`; and this is only one of many examples of native arithmetics in the codebase involving ticks of fees. When performed over incorrect data, these computations may lead to unpredictable results, and also to security vulnerabilities.

Recommendation

Carefully inspect all transactions that accept `tickIndex` and `fee` inputs, and add checks to their respective `ValidateBasic()` functions that would ensure that all of the following are in the acceptable range of `[-559680, 559680]` for tick indices:

- `tickIndex`
- `fee`
- `tickIndex + fee`
- `tickIndex - fee`

It is much easier and failproof to reject incorrect inputs at the system boundary, than to try to fight it off at dozens of places inside the system.

Miscellaneous code improvements

Title	Miscellaneous code improvements
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION PRACTICE
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Threats	Best Coding Practices
Author(s)	Andrey Kuprianov Ivan Gavran Nikola Jovicevic
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/pull/355 https://github.com/neutron-org/neutron/pull/345

Description

In this issue we describe a number of improvements to the code that do not affect the functionality, improve readability or are a good practice.

Cosmos SDK 0.47 integration (neutron-org / neutron and admin-module repos)

1. There is a wrong log message in `HandleTimeout` (it refers to acknowledgement instead of the timeout): [here](#)
2. The entire logic of the `DispatchMessage` function is contained within the if-branch. A more elegant approach would involve placing the opposite condition inside the if statement.
3. Inside of the each module's `module.go` file, there is a `ConsensusVersion()` function which returns a sequence number for state-breaking change of the module which should be incremented for each consensus-breaking change introduced by the module, such as [here](#). The current implementation directly returns the consensus version as a hard-coded value, which could potentially lead to inconsistencies if not updated properly during migrations or code updates. We recommend introducing a constant to track the version.
4. The current implementation hard-codes the global fees and denom values [here](#) and [here](#), which might not be optimal for maintainability and flexibility. Consider introducing constant values for fees, and if possible, add denominations dynamically.
5. There is a [superfluous assignment](#) of the memo field. After creating the new `transferMsg`, there is no need to assign its memo field again.

6. Unclear and copy-pasted [descriptions](#) of `TrackBeforeSend` and `BlockBeforeSend` hooks.
7. `callBeforeSendListener` function refactoring:
 - a. This should be implemented within two separate functions, which would be a much more elegant approach due to these two if-else statements ([here](#) and [here](#));
 - b. Entire logic is stored inside of [this](#) if-branch. The opposite condition inside the if statement should be placed;
 - c. Error handling [here](#) is superfluous because error is not returned from the `TrackBeforeSend` hook.
8. Cosmos SDK maintains two versions of their `gov` module: `v1` and `v1beta1` (for legacy reasons). When both need to be used (as is the case for the `adminmodule` and `neutron` codebase), a descriptive name for the import is used to disambiguate between them.
 In the codebase under audit, we noticed inconsistent usages. These never cause an error, but might be a source of confusion when reading the code.
 In particular:
 - the name `govtypes` is used to mean `gov/types/v1beta1` in `adminmodule/types/codec.go` and in `neutron neutron/app/proposals_allowlisting.go`. However, it means `gov/types` in `neutron/app/app.go`
 - `gov/types/v1beta` carries different names throughout the codebase: `govv1beta1`, `govtypes`, `govv1types`, `govv1beta1types`, `govtypesv1b1`, and `v1beta1`
9. Tests are not maintained: some of them do not even compile and some fail unexpectedly (e.g., [TestSoftwareUpgradeProposal](#)).

Duality integration (duality-labs / duality repo at v.0.5.0)

1. Instead of using deprecated `sdk.Int`, import `sdkmath "cosmosdk.io/math"`
2. There is an unnecessary declaration of `AxelarGMPAcc` constant since Axelar authentication is not included. Additionally, its value is incorrect, as it does not match the one provided in the [Axelar documentation](#) (the 's' letter is missing at the very end).
3. Function `stake.go:SingleCoin()` is unused.
4. Functions `GetFillTranche()` and `GetAllLimitOrderTrancheAtIndex()` are used only in tests. We recommend moving test-only functions to the files with test utilities, such that they don't clutter the production code.
5. Remove self-assignment in `core.go#L327`.
6. Function `core.go::Swap()` has a single return case, which always returns `error == nil`. Thus, you can eliminate returning of errors from that function, and, by transitivity, from the function `core.go::SwapWithCache()`; as well as handling of errors from all locations where these functions are invoked.

Unutilized event collection

Title	Unutilized event collection
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	Best Coding Practices
Author(s)	Nikola Jovicevic
Status	RESOLVED
Link(s)	https://github.com/neutron-org/admin-module/pull/6

Involved artifacts

- [x/adminmodule/keeper/proposal.go](#)

Description

Within the `SubmitProposal` function, handlers are appropriately executed, and subsequent events are gathered. However, these events are neither published nor passed elsewhere. Consequently, potentially valuable information remains unutilized and obscured within the codebase.

Problem Scenarios

1. In scenarios where other neutron modules rely on these events, their absence could lead to incorrect behavior due to a lack of necessary information.
2. Additionally, if debugging or monitoring the proposal handling process becomes necessary, these events would remain unnoticed, potentially hindering the resolution of issues.
3. Collecting events without utilizing them introduces unnecessary computational overhead, potentially impacting the performance of the application.

Recommendation

To address this issue, it is recommended to emit the events immediately after they are gathered within the `SubmitProposal` function.

Error message referring to a wrong proposal id

Title	Error message referring to a wrong proposal id
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	PRACTICE
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	Best Coding Practices
Author(s)	Ivan Gavran
Status	RESOLVED
Link(s)	https://github.com/neutron-org/admin-module/pull/5

Involved artifacts

- [x/adminmodule/keeper/proposal.go::SubmitProposal](#)

Description

When the admin module executes proposals, it does so in the following way:

```
proposalID, err := k.GetProposalID(ctx)
...
for idx, msg := range msgs {
    handler := ...
    res, err := handler(ctx, msg)
    if err != nil {
        return proposal, fmt.Errorf("failed to handle %d msg in proposal %d: %w",
idx, proposal.Id, err)
    }
}
```

The error message refers to the id `proposal.Id`. However, given that this value is just created in the function and is never saved, it is not known outside of the function.

Problem Scenarios

Referring to the `proposalId` is useless to callees. At worst, it might be confusing if the error is inspected: it refers to a proposal id that will be used for the next proposal (which might be a successfully executed one).

Recommendation

It might make sense to store the information about proposals that were not handled successfully. Alternatively, one could simply remove mentioning of the `proposal.Id`.

Multi-message proposals stored as different proposals

Title	Multi-message proposals stored as different proposals
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Threats	Code Clarity
Author(s)	Ivan Gavran
Status	RISK ACCEPTED
Link(s)	

Description

A proposal sent to the adminmodule to execute could, in principle, consist of multiple messages. These messages will be stored as different proposals within `adminmodule`.

Problem Scenarios

Assume a proposal consisting of two SDK messages, `m1` and `m2` was passed in `neutron-DAO`.

Subsequently, a call to `execute the proposal` was sent to the contract which adds messages `msgs = [m1, m2]`.

In `wasmd`, the call to execute is handled in sequence by `ExecuteContract`, `execute`, and finally

`handleContractResponse`, which handles the returned `msgs`. Handling is done through `DispatchSubmessages`, which iterates over all messages in `msgs` and for each of them calls `DispatchMsg`. Each of these messages will be assigned a separate `proposalId` ([here](#)), under which it will be stored.

This does not have correctness implications, but is confusing towards the outside: a number adminmodule's stored proposals may in fact correspond to a single actual proposal. Another (potential) point of confusion is that those IDs do not correspond to the IDs of the actual proposals at `neutron-DAO`.

Recommendation

One way to remove the confusion would be to have adminmodule's actions have the same ID as the original DAO proposals. That could be accomplished by adding the additional `ID` field to the `ProposalExecuteMessage` struct.

However, in our discussion with the dev team, we learned that they don't consider the mapping between original proposals and `adminmodule`'s proposals significant. Thus, we suggest changing the naming inside the `adminmodule` to make it explicit that what is being executed is not a proposal, but something different (eg, *proposal action*, *admin_proposal*, or similar).

Duality modules still use deprecated x/params

Title	Duality modules still use deprecated x/params
Project	Neutron: Duality v0.5.0 + integration of Cosmos SDK 0.47
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Threats	Best Coding Practices
Author(s)	Ivan Gavran Nikola Jovicevic
Status	RESOLVED
Link(s)	https://github.com/neutron-org/neutron/blob/923584c62074e2188ebc5f5aedafad6d289ed45e/x/dex/keeper/msg_server.go#L187

Involved artifacts

- `x/incentives/types/params.go`
- `x/dex/types/params.go`

Description

During the inspection of Duality [PR#423](#), we noticed that parameters are still all handled by the legacy `x/params` module, instead of being handled by each module individually.

Problem Scenarios

There is no immediate problem since v0.47 still contains `x/params`, but this module will be removed in one of the future releases.

Recommendation

Given that the Neutron codebase migrated to params being handled by individual modules, we suggest the same be done for Duality modules. Here is a [PR](#) you can take a look at as an example.





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:

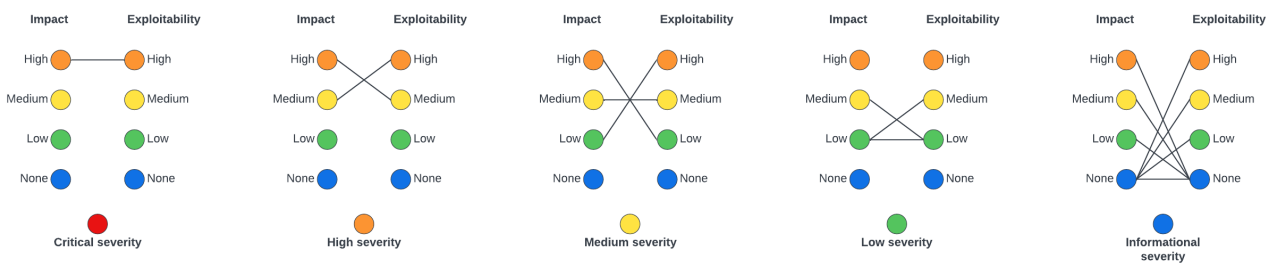
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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.