



# **Security Audit Report**

Duality: Dex and Incentives modules

Authors: Andrey Kuprianov, Ivan Golubovic, Aleksandar Ljahovic

Last revised 16 May, 2023

# Table of Contents

<b>Audit Overview .....</b>	<b>1</b>
The Project	1
Scope of this audit	1
Conducted work	1
Conclusions	1
Further Increasing Confidence	2
<b>Audit Dashboard .....</b>	<b>3</b>
Target Summary	3
Engagement Summary	3
Severity Summary	3
<b>System Overview.....</b>	<b>4</b>
Modules	4
<b>Threat Inspection.....</b>	<b>7</b>
User categories	7
DEX related threats	7
Incentives related threats	8
<b>Findings .....</b>	<b>11</b>
Repetitive exponentiation in price computation may lead to DOS	13
A byzantine consumer can cause chain halt via pricing tick	15
Loss of user funds via shares rounding with large ticks	17
Stealing of user funds via negative deposits	19
Wasteful usage of storage creates a potential for DOS attacks	21
Denom collisions can be exploited to steal user funds	23
Stakes querying not handling stakes with multiple coins	27
Tick and Fee inputs are not validated	29
Incomplete validation of MsgDeposit	31
Optimize stakes querying	32
Number of epochs for gauge creation is not validated	34
Invalid Multi Hop Swap routes cause panic	36
Improvements for better efficiency	38

Large string for token causes panic	40
Minor code improvements for Incentives module	42
Minor code improvements for the DEX module	43
<b>Appendix: Vulnerability Classification .....</b>	<b>44</b>
Impact Score	44
Exploitability Score	44
Severity Score	45
<b>Disclaimer .....</b>	<b>47</b>

# Audit Overview

## The Project

Duality is a "concentrated liquidity exchange", which means that for a given trading pair, it relies on many different constant-sum AMM pools to fill orders. This is in contrast to simpler AMM exchanges, which rely on a single constant-*product* market maker pool to fill orders. The comparative advantage of the constant-sum AMM design is that it allows users who wish to LP to have total control over where they place their liquidity and how much they charge in fees. In its design, Duality combines two approaches to liquidity (thus the name): AMMs (liquidity pools), and Order books (limit orders).

## Scope of this audit

The audit took place from April 18, 2023 through May 12, 2023 by Informal Systems by the following personnel:

- Andrey Kuprianov
- Ivan Golubovic
- Aleksandar Ljahovic

During the audit, we focused on analyzing the x/dex and x/incentives modules; the remainder of the system was largely considered a "black box" during the audit. In particular, it has been explicitly negotiated that the mechanisms of integrating Duality with Cosmos Hub as a Replicated Security consumer chain is outside of the scope of this audit, and needs to be addressed separately.

## Conducted work

Within the scope of the audit project the following work has been conducted:

- Manual code inspection of the DEX and Incentives module. We have carefully inspected the code base of these modules, and documented our understanding in the section "System Overview"; manual code inspection allowed us to discover the majority of the findings in the present report.
- Reconstruction of the mathematical protocol employed by the DEX module, and modeling it in the Quint specification language. Due to the limited timeline of the project we've been able to complete only around 80% of the model (e.g. limit orders are not yet modeled, or not all error conditions are covered). Despite these limitations, the Quint DEX model allowed us to discover 2 of the 4 critical findings listed in the present report.
- Whenever possible, we've tried to reproduce the findings using Duality's integration test suite; the corresponding integration tests are included into the text of the respective findings.

## Conclusions

In general, we found the codebase to be of very high quality: the code is well-structured and easy to follow, and the test suite includes both unit and integration tests. Despite the high code quality we have discovered 4 Critical and 2 High severity findings, the rest of the findings being Medium, Low, or Informational severity. From our analysis, we would like to point to the following areas of immediate, relatively low effort improvements to the codebase:

- **Comprehensive validation of system inputs is needed.** Around half of the findings we've discovered in the course of this audit are due to either missing, or incomplete validation of transaction parameters.
- **The test infrastructure needs to be made more general and data-driven.** The current test suite, while being well thought-through and easy to use, is too limited in order to test all possible scenarios that are important for the system correctness. E.g., only two token types ("TokenA", "TokenB") can be supplied to the tests; or a very limited, set of users ("Alice", "Bob", "Carol") can be employed in the tests, and only via calling the appropriately named test functions (e.g. "aliceDeposits" or "assertBobBalances"), etc. These limitations need to be lifted, and the test infrastructure needs to be made data-driven, when the arbitrary

test inputs can be supplied as data, and not as Go code. This should allow to both reduce the amount of code in the test suite, and simultaneously to substantially increase its coverage.

## Further Increasing Confidence

Based on the modeling efforts in Quint for the DEX module, we can safely recommend to the Duality developers to continue this route. A formal model can serve both as a concise documentation of the system behavior, as well as a means of uncovering protocol level bugs in the mechanism design, which would be very difficult to find otherwise. As immediate next steps we recommend the following:

- Allocate the time to finalize the DEX Quint model. The goal here is to obtain the complete specification of the DEX protocol, as well as to formulate and validate the set of formal properties the system should satisfy. E.g., with the current model we've come very close to formally describing and validating the following (informal) English language property:
  - a. In any system state, when a liquidity provider deposits an amount of tokens to a liquidity pool;
  - b. Followed by any number of system actions;
  - c. When the liquidity provider withdraws all the shares obtained in step a), they obtain the amounts of tokens that are not smaller than the originally deposited token amounts, taking into account the exchange rate of the liquidity pool in question.
- Formally model the Incentives module, formulate and validate its properties. From our analysis we can say that formal modeling of this module in Quint should bring substantial benefits in understanding and validating the underlying protocols, as it includes non-trivial distribution logic involving the time axis.
- Implement the infrastructure for model-based testing (MBT) of DEX and Incentives modules, and integrate it into the project CI: when the aforementioned generalization of the test suite is performed, and the tests are data-driven, then performing MBT on the base of Quint models should dramatically increase the test coverage, via the systematic exploration of all possible scenarios of system evolution.

# Audit Dashboard

## Target Summary

- **Type:** Specification and Implementation
- **Platform:** Golang
- **Artifacts:**
  - Duality [DEX module at v0.2.0](#)
  - Duality [Incentives module at v 0.2.0](#)

## Engagement Summary

- **Dates:** 18.04.2023 to 12.05.2023
- **Method:** Manual code review, protocol analysis, formal modeling
- **Employees Engaged:** 3

## Severity Summary

Finding Severity	#
Critical	4
High	2
Medium	2
Low	5
Informational	3
<b>Total</b>	<b>16</b>

## System Overview

Duality's mission is to give liquidity providers superpowers by creating sustainable and powerful financial markets. Duality is introducing a novel mechanism design which combines the advantages of AMMs and order books. Duality can reach order-book levels of capital efficiency (zero price impact on trades, swaps, and limit orders) while still maintaining the computational efficiency and liveness properties of AMMs.

At the core of the AMM lays an incredibly simple swap mechanic: liquidity pools that allow traders to buy or sell tokens at a constant price.

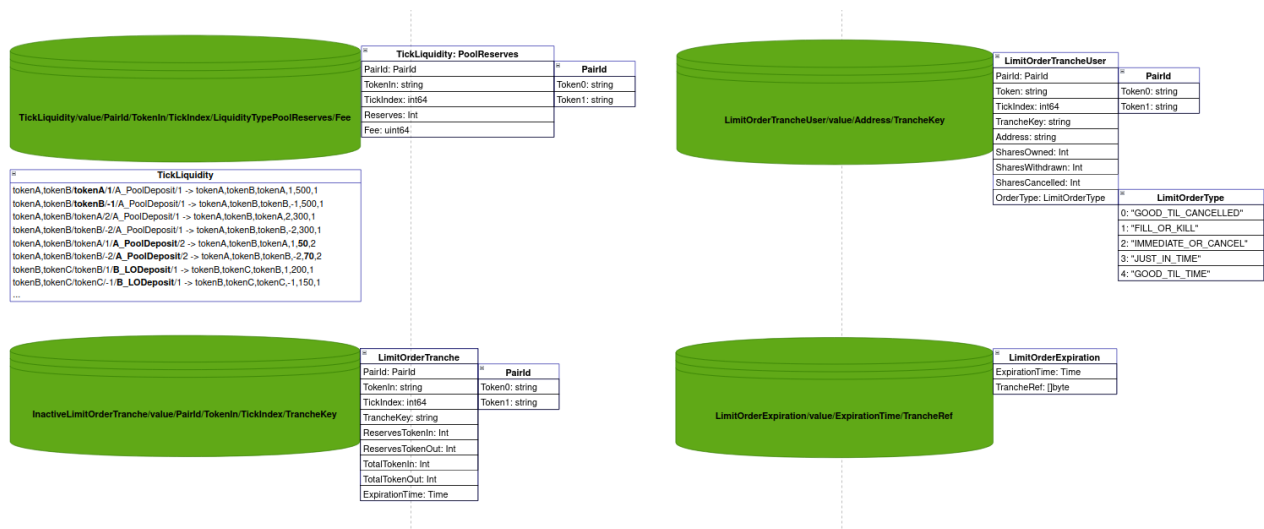
While two liquidity providers on Duality may prefer completely different liquidity distributions, they will still use the same underlying pools. The only difference is the amount deposited in each pool.

## Modules

### DEX module

The **DEX** (Decentralized Exchange) module holds the core functionality for token swaps, limit orders, and liquidity provisioning

Data structures (KV Stores):



### Messages:

#### 1. Deposit

Enables tokens to be deposited in different pools for the same pair of denoms. Based on the amount of deposited funds, shares tokens are calculated and sent to the receiver, while the deposited funds are transferred to the dex module. Also, data structures (pool, tick, pair) are checked, initialized and updated in this message.

#### 2. Withdrawal

Enables the withdrawal of funds for the same pair of denoms from different pools, by specifying ticks and fees. Also, for each pool, it is possible to specify the number of shares tokens that you want to withdraw. The calculation determines the amount of certain tokens that corresponds to the given amount of share tokens. If everything goes as expected, shares tokens are withdrawn from the caller's account and burned, and the calculated amount of tokens are sent to the receiver's account.

### 3. Swap

Allows the user to exchange a certain amount of a token with a certain amount of another token.

It is iterated by the Liquidities corresponding to the tokens participating in the swap. In each iteration, a swap is performed over the current liquidity, and as a result, the amount of in/out tokens used from that liquidity for the swap is returned. The iteration ends either when the amount of input tokens reaches the specified maximum or when all iterations are finished.

The Liquidity interface has two implementations: PoolLiquidity and LimitOrderTranche

Input tokens are sent from the caller's account to the dex module, and output tokens are sent from the dex module to the receiver's account.

### 4. MultiHopSwap

Allows specifying multiple possible routes to swap tokenA for tokenB:

OSMO -> ATOM -> ETH

OSMO -> ATOM -> USDC -> ETH

OSMO -> USDC -> ETH

For each of the routes, the best price for the swap between two tokens is calculated, until the end is reached.

There is an option to specify whether to calculate the best route or swap according to the first one from the collection.

Input tokens are sent from the caller's account to the dex module, and the calculated output tokens are sent from the dex module to the receiver's account.

### 5. PlaceLimitOrder

Enables adding a limit order for a certain pair of denoms, the amount of input token, tick and order type.

### 6. WithdrawFilledLimitOrder

Calculates and sends filled liquidity from module to user for a limit order based on amount wished to receive.

### 7. CancellLimitOrder

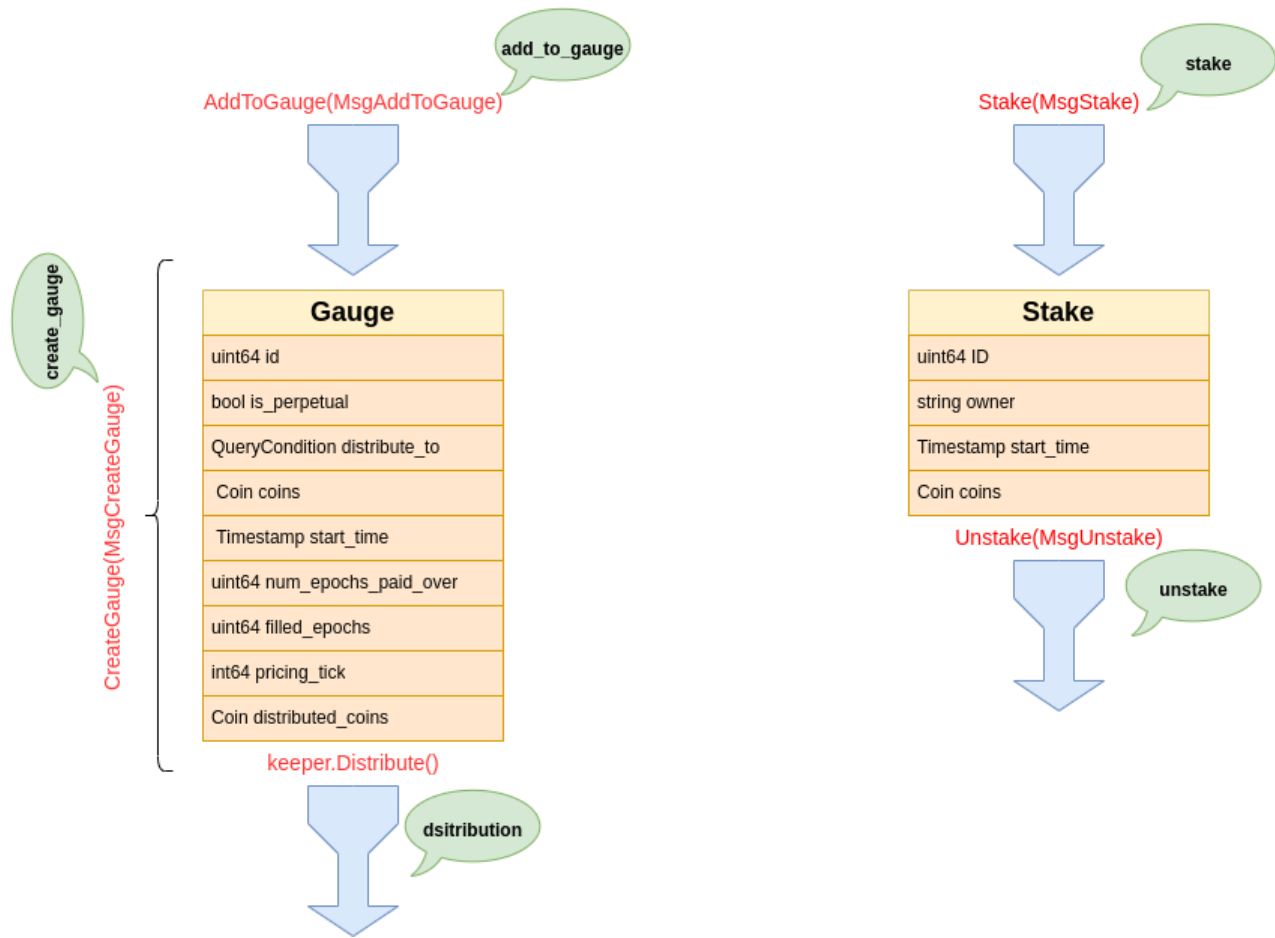
Removing a specified number of shares from a limit order and returning the respective amount in terms of the reserve to the user

## Incentives module

Incentive module represents Duality's take on rewards program. The intention is to reward users providing liquidity for trading. In Duality's terms this means providing liquidity on particular positions where the position is defined by token pair and chosen tick range.

From high level perspective, the module is organized around two main types: gauges and stakes. The following diagram explains the structure of the types and basic operations:





As it is shown above, basic operations include four messages:

1. **CreateGauge** for constructing the gauge itself and adding coins to it. The operations includes storing gauge to appropriate KV store, it's indexing and emitting of event. After the gauge is created, the response is returned.
2. **AddToGauge** is designed to handle increasing of coin amount in the gauge. It includes validation of gauge's activity (coins cannot be added to finished gauges) as well as event emitting and response when the message is processed.
3. **Stake** message is created to support creating of new stake or adding stakes to already created ones. The most important parts of this operation include owner validation and denom calculation. It is not allowed to stake multiple tokens. Event emitting and response also included.
4. **Unstake** comprises of iterating through unstaking array and finding and executing eligible unstakes. Eligibility depends on coin amount, owner and time of unstaking action. It also includes event emitting and response.

Apart from four messages described above, probably the most important action is the reward distribution. It is executed in **Distribute** function inside keeper. It strongly depends on epochs (time period for rewards distribution), as it can be executed only upon *active gauges* which are determined by comparing gauge time and running block time. All eligible gauges are looped with their respective stakes. To perform the reward distribution, coins must be extracted from stake, and rewards have to be properly calculated. The concept used to calculate the reward is the **pricing\_tick**. It is designed to support fair distribution and stop reward increase in cases of users staking greater amounts on specific positions. The distribution event is triggered when all the rewards are delivered to addresses that "deserved" their reward.

# Threat Inspection

## User categories

- **Duality stakeholders:** Duality owners, or whoever has an economic upside from the Duality itself
- **Liquidity providers (LPs):** Users who provide their tokens, and thus give liquidity for trades. Their main incentive is to get upside from trades via fees
- **Traders:** Users who perform trades on Duality. Their incentive is to extract value via trading tokens at different exchanges at varying valuations.

## DEX related threats

### Threat: Traders' tokens are lost while swapping

The DEX module has certain super-powers (e.g. transferring tokens from a user account via Bank module's `SendCoinsFromAccountToModule`). If not used carefully, this may have disastrous consequences for users.

- Impacted users:
  - Traders
- Possible impacts:
  - Loss of funds
- Possible ways of exploitation:
  - Submitting a transaction with specially crafted inputs (e.g. such that the coins with the wrong denoms are used)
- Audit actions: examine all places where superpowers (Bank module methods) are used:
  - from where these methods are used
  - How their inputs are formed
  - Whether those inputs can be maliciously crafted to perform an exploit.

### Threat: Traders get unfair swap price

The main incentive for traders to participate in Duality is to trade their tokens at the expected prices. We should make sure that the core Duality swapping logic is correct

- Impacted users:
  - Traders
  - Liquidity providers
- Possible impacts:
  - Traders receive an unfair amount of swapped tokens
    - receiving less means traders lose their funds
    - receiving more means the system is exploitable, and liquidity providers lose their funds.
- Possible ways of exploitation:
  - Discovering a flaw in the economic mechanism and submitting a crafted transaction to exploit it
- Audit actions: validate Duality core swap protocols
  - reconstruct the protocol from code, and model them in Quint
  - formulate desirable properties of the protocol (invariants)
  - (partially) validate invariants via Quint simulation
  - (for the future): validate reconstructed protocols with the mechanism design expert

## Threat: Liquidity providers get unfair shares

When LPs submit their funds into liquidity pools, they receive shares instead. We should make sure that the shares they receive are fair wrt. all parameters.

- Impacted users:
  - Liquidity providers
- Possible impacts:
  - LPs submitting funds receive an unfair amount of shares
    - receiving less means they lose their funds
    - receiving more means the system is exploitable, and other liquidity providers lose their funds.
- Possible ways of exploitation:
  - Discovering a flaw in the economic mechanism and submitting a crafted transaction to exploit it
- Audit actions: validate Duality core LP protocols
  - reconstruct the protocols from code, and model them in Quint
  - formulate desirable properties of the protocols (invariants)
  - (partially) validate invariants via Quint simulation
  - (for the future): validate reconstructed protocols with the mechanism design expert

## Threat: Out-of-thin-air token generation due to swap rounding

Calculations when doing swapping are non-trivial, with multiplication and truncation, and multiple cases. (see e.g. [Pool::Swap0To1\(\)](#)). Rounding errors may lead to generation of residual tokens; however small, this may lead to depleting the reserves.

- Impacted users:
  - Duality stakeholders
  - Liquidity providers
- Possible impacts:
  - Loss of funds
- Possible ways of exploitation:
  - Submitting multiple transactions with specially crafted inputs (e.g. very large values)
- Audit actions: precisely understand the logic there, and how it will behave:
  - under various combinations of conditions
  - when some of the inputs/state variables are very small or very large
  - whether the rounding errors can be exploited in user's favor.

## Incentives related threats

- **Miscalculation that can be abused to lead to unfair allocation of rewards.**  
Miscalculation in terms of amount can harm both users and chain holders. Apart from amount, the denomination (also a result of some sort of calculation) should always be correct in order not to reward arbitrary tokens.
- **Routing tokens to incorrect user.**  
Since the number of users that are involved in staking is presumably arbitrary, it should be inspected that rewards distribution is correctly routed and that rewards always end up on accounts that really did the staking and “deserved” the reward.
- **Abuse the time concept of rewarding mechanism - stop or delay reward distribution.**  
The rewards are distributed using epochs module which is responsible for creating particular time periods in which the distribution is happening. The mechanism should be resistant to any abuse that some rewards are not distributed until the end of period.
- **Malicious validators with significant portion of power trying to control incentives mechanism.**  
This threat is in some cases known as 66% attack and it implies there are validators with voting power exceeding 66% and that their intentions are malicious in any way. In this particular case, the goal would be

to misuse incentives program (create any of the previous threats maybe) or to take control of it in a way that no other validators can be rewarded.

- **Intentionally a malicious user defines incentives programs that slows down the chain (by exceeding the block time)**

It should be inspected that rewards are always distributed to the accounts on time without possibility of slowing down the chain with user created malicious gauges or stakes.

Another part of the Incentives module is staking/unstaking of tokens which also could be point of attack:

- **Staking to specific tick position or trading pair to abuse trading frequency on that pair.**

There should be a mechanism (*pricing\_tick* in distribution) stopping a user of a group of users creating stakes at such positions and in such quantity that get rewarded more in a sense that rewards are distributed in proportion to the number of deposit shares staked within the gauge's qualifying tick range.

The results of the threat inspection with findings that could influence the probability of threat actually being materialized are shown in the following table:

Threat	Risk	Findings
<b>Miscalculation that can be abused to lead to unfair allocation of rewards.</b>	Medium risk	1. Stakes querying not handling stakes with multiple coins
<b>Routing tokens to incorrect user.</b>	Low risk	-
<b>Abuse the time concept of rewarding mechanism - stop or delay reward distribution.</b>	Low risk	1. NumOfEpochsPaidOver not validated
<b>Malicious validators with significant portion of power trying to control incentives mechanism.</b>	High risk	1. NumOfEpochsPaidOver not validated 2. Stakes querying not handling stakes with multiple coins 3. A byzantine consumer can cause chain halt via pricing tick
<b>Intentionally a malicious user defines incentives programs that slows down the chain (by exceeding the block time)</b>	High risk	1. A byzantine consumer can cause chain halt via pricing tick
<b>Staking to specific tick position or trading pair to abuse trading frequency on that pair.</b>	Low risk	-

It could be noticed that some findings are repeated, and some of them do not maybe resemble the threat the exact way it was earlier defined. For example, the threat is defined as “...**slows down the chain**”, whereas the finding linked in the table causes chain halt. This is due to the fact that threats are defined prior to detailed code inspection, and as the system understanding is improved, and findings are created, the threats also evolve. Wrt. previous, the table can be used to show how one finding can influence more than one threat, or more than the threat's original scope.

The level of risk shown in the table is not formed out of any strict rules. It is created from auditor's impression on how the software is being defended against particular threats, what parts of code are included or what code is missing regarding this defense. Also, the findings greatly influence the risk evaluation and since there is a critical

finding that influences two threats possibilities to be materialized (or any kind of extension to this threats), these are evaluated as high risk.

The first threat in the table where the risk is evaluated as medium, is also greatly influenced by the finding linked in the table. The calculation of rewards itself is pretty well thought of, the operations with KV Stores look safe, but there is a possibility of skipping coins that are supposed to be distributed due to the error in collection definition, as explained with more details in the finding. Also, it should be noted, that calculations regarding rewards are not trivial, and could be more thoroughly tested using Quint (formal verification language from Informal Systems).

Other threats noted in the table are evaluated as low risk. This is due to the impression that current codebase includes all the necessary defense mechanisms against these threats, or the process is designed in a way that does not leave much room for potential attackers to create such harmful scenario. Some useful notes regarding these threats are:

**Routing tokens to incorrect user:** All important routing of funds is performed using `SendCoinsFromAccountToModule` or `SendCoinsFromModuleToAccount` from BankKeeper.

**Staking to specific tick position or trading pair to abuse trading frequency on that pair:** Duality's idea of pricing tick completely removes possibility of naive distribution due to stake positions, as it was clearly explained in their documentation.

**Abuse the time concept of rewarding mechanism - stop or delay reward distribution:** The finding linked in the table does in some way influence the time user has to, for example, wait to create a gauge. However, after discussing with Duality team, we agreed that this scenario represents a scam, and the idea for the incentives program to be permissionless, there is no need to limit some of the input in the current state, or to change the whole mechanism (switch to governance proposals). Also, the motive for such an attack is hard to be found, so the risk is evaluated as low.

Generally speaking, the most obvious flaw we saw during the audit is the lack of input validation throughout the codebase. There is a number of user created data that is not valuated, but can lead to harmful scenario. However, it will be visible in findings later themselves what could be improved in this area also.

## Findings

Title	Type	Severity	Impact	Exploitability
Repetitive exponentiation in price computation may lead to DOS	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH
A byzantine consumer can cause chain halt via pricing tick	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH
Loss of user funds via shares rounding with large ticks	PROTOCOL	4 CRITICAL	3 HIGH	3 HIGH
Stealing of user funds via negative deposits	IMPLEMENTATION	4 CRITICAL	3 HIGH	3 HIGH
Wasteful usage of storage creates a potential for DOS attacks	IMPLEMENTATION	3 HIGH	2 MEDIUM	3 HIGH
Denom collisions can be exploited to steal user funds	IMPLEMENTATION	3 HIGH	3 HIGH	2 MEDIUM
Stakes querying not handling stakes with multiple coins	IMPLEMENTATION	2 MEDIUM	2 MEDIUM	2 MEDIUM
Tick and Fee inputs are not validated	IMPLEMENTATION	2 MEDIUM	1 LOW	3 HIGH
Incomplete validation of MsgDeposit	IMPLEMENTATION	1 LOW	1 LOW	2 MEDIUM
Optimize stakes querying	IMPLEMENTATION	1 LOW	1 LOW	1 LOW
Number of epochs for gauge creation is not validated	IMPLEMENTATION	1 LOW	1 LOW	2 MEDIUM
Invalid Multi Hop Swap routes cause panic	IMPLEMENTATION	1 LOW	1 LOW	2 MEDIUM
Large string for token causes panic	IMPLEMENTATION	1 LOW	1 LOW	1 LOW
Improvements for better efficiency	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE

Title	Type	Severity	Impact	Exploitability
Minor code improvements for Incentives module	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE
Minor code improvements for the DEX module	IMPLEMENTATION	0 INFORMATIONAL	0 NONE	0 NONE

## Repetitive exponentiation in price computation may lead to DOS

<b>Title</b>	Repetitive exponentiation in price computation may lead to DOS
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	4 CRITICAL
<b>Impact</b>	3 HIGH
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	

### Involved artifacts

- [duality/price.go at v0.2.0](#)

### Description

In the current implementation, the `Price` datastructure has a single field, `RelativeTickIndex int64`, which is used in 2 places in `price.go` in the same way, illustrated here on the example of the `MulInt()` / `Mul()` functions:

```
func (p Price) MulInt(other sdk.Int) sdk.Dec {
    return p.Mul(other.ToDec())
}

// We are careful not to use negative-valued exponents anywhere
func (p Price) Mul(other sdk.Dec) sdk.Dec {
    if p.RelativeTickIndex >= 0 {
        return other.Quo(utils.BasePrice().Power(uint64(p.RelativeTickIndex)))
    }

    return other.Mul(utils.BasePrice().Power(uint64(-1 * p.RelativeTickIndex)))
}
```

The above function is used in multiple (at least 10) places throughout the codebase to perform conversion from a certain amount of one token to the amount of another token when these tokens constitute a trading pair; an example can be found in `pool.go::Swap0To1()`:

```
maxOutGivenIn1 := p.Price0To1Upper.MulInt(maxAmount0).TruncateInt()
```



The above computation is problematic because of two reasons:

1. Exponentiation of decimals, which is used in `MulInt()` / `Mul()` via `Power()` function, is computationally expensive: it involves a logarithmic number of multiplication of big integers; with `MaxTickExp` being 352437, means that a single `Power()` computation may involve up to 19 big integer multiplications.
2. The above functions are used repeatedly when iterating through liquidity: a single swap order will iterate through all available tick indexes for the desired token pair, and perform this computation over and over again.

## Problem Scenarios

- The least problematic, though still very expensive scenario is that the nodes will perform lots of unnecessary computations, thus wasting resources and energy
- The most problematic scenario is that this can be abused by unprivileged attackers to stop the network. All they need to do is:
  - create many liquidity pools at different tick indices, holding very little reserves each
  - submit a `Swap` transaction for a moderate amount of tokens that falls within the scope of these pools, thus triggering iteration over all tick indices, with the swap not fulfilled till the very end.

## Recommendation

In the `Price` datastructure, store not the tick index, but the corresponding decimal, and precompute it once when creating the datastructure.

## A byzantine consumer can cause chain halt via pricing tick

<b>Title</b>	A byzantine consumer can cause chain halt via pricing tick
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	4 CRITICAL
<b>Impact</b>	3 HIGH
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	

### Involved artifacts

- [duality/gauge.go at v0.2.0](#)
- [duality/disitrbutor.go at v0.2.0](#)
- [duality/cli\\_test.go at v0.2.0](#)

### Description

There is no validation of pricing tick property of a gauge present at the moment, so the user can specify any value(int64). Pricing tick is used in `price1To0Center` calculation in `ValueForShares`. This function gets called from `Distribute`, which is originally invoked from the hook in `BeginBlocker` of epochs module.

During the calculation in which the mentioned tick is used, a panic occurs if tick is outside of range ( `[-352437, 352437]` ). Since the panic then can be reached from `BeginBlocker`, there is a risk of halting the chain.

### Problem Scenarios

A Byzantine user can specify any int64 value for pricing tick when creating a gauge, thus it can be set to 360000 for example. To confirm the gauge creation process would not detect any abuse, a test can be modified:

```
func TestNewCreateGaugeCmd(t *testing.T) {
    testTime := time.Unix(1681505514, 0).UTC()
    desc, _ := cli.NewCreateGaugeCmd()
    tcs := map[string]osmocli.TxCliTestCase[*types.MsgCreateGauge]{
        "basic test": {
            Cmd: fmt.Sprintf("TokenA<>TokenB 0 100 100TokenA,100TokenB 10 360000 --
from %s", testAddresses[0]),
            ExpectedMsg: &types.MsgCreateGauge{
                IsPerpetual: false,
                Owner:        testAddresses[0].String(),
                DistributeTo: types.QueryCondition{
                    PairID: &dextypes.PairID{Token0: "TokenA", Token1: "TokenB"},
```

```

        StartTick: 0,
        EndTick: 100,
    },
    Coins: sdk.NewCoins(
        sdk.NewCoin("TokenA", sdk.NewInt(100)),
        sdk.NewCoin("TokenB", sdk.NewInt(100)),
    ),
    StartTime:      time.Unix(0, 0).UTC(),
    NumEpochsPaidOver: 10,
    PricingTick:     360000,
},
},
...

```

Since no error or panic occurs, the gauge is created and added to incentives program. When the distribution is invoked from `BeginBlocker`, gauge can eventually become active and included in the calculation. Upon including its pricing tick in the calculation, a panic occurs. It can be caused also using one of the already present tests:

```

func (suite *KeeperTestSuite) TestDistribute() {
    ...
    gaugeSpecs: []gaugeSpec{
        {
            isPerpetual: false,
            rewards:     sdk.Coins{sdk.NewInt64Coin("reward", 3000)},
            startTick:    -10,
            endTick:      10,
            paidOver:     1,
            pricingTick:  360000,
        },
    },
}

```

After running a test, the following stack trace is shown:

```

--- FAIL: TestKeeperTestSuite (0.00s)
    --- FAIL: TestKeeperTestSuite/TestDistribute (0.00s)
        --- FAIL: TestKeeperTestSuite/TestDistribute/one_gauge (0.00s)
panic: supplying a tick outside the range of [-352437, 352437] is not allowed
[recovered]
    panic: supplying a tick outside the range of [-352437, 352437] is not allowed

```

Another possible scenario for a pricing tick set to such value to cause chain halt is a mistake in user input, or a result of automated or code generated input.

## Recommendation

Validation of pricing tick should be introduced to return an error when it is specified out of range.

## Loss of user funds via shares rounding with large ticks

<b>Title</b>	Loss of user funds via shares rounding with large ticks
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	PROTOCOL
<b>Severity</b>	4 CRITICAL
<b>Impact</b>	3 HIGH
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	

### Involved artifacts

- [duality/pool.go at v0.2.0](#)

### Description

When a user deposits funds to a liquidity pool, the amount of shares they get is calculated in the function `CalcSharesMinted()` as follows:

```
valueMintedToken0 := CalcAmountAsToken0(amount0, amount1, *price1To0Center)

valueExistingToken0 := CalcAmountAsToken0(p.LowerTick0.Reserves,
p.UpperTick1.Reserves, *price1To0Center)
var sharesMintedAmount sdk.Int
if valueExistingToken0.GT(sdk.ZeroDec()) {
    sharesMintedAmount =
valueMintedToken0.MulInt(existingShares).Quo(valueExistingToken0).TruncateInt()
} else {
    sharesMintedAmount = valueMintedToken0.TruncateInt()
}
```

Thus, for the shares issued, the `amount1` of `Token1` is recalculated into the amount of `Token0`. The problem with this approach is when the price of `Token1` is very low wrt. the price of `Token0`, the truncation can substantially reduce the shares a user gets. This may potentially lead to loss or stealing of user funds. The issue has been first discovered in a [Quint](#) model of the DEX module, and then confirmed via the following integration test.

```

func (s *MsgServerTestSuite) TestDepositLossViaSharesRounding() {
    s.fundAliceBalances(0, 1000000)
    s.fundBobBalances(0, 1200000)

    s.assertPoolLiquidity(0, 0, -1000000, 1)
    s.assertAliceShares(-1000000, 1, 0)

    s.aliceDeposits(NewDeposit(0, 1000000, -1000000, 1))
    s.assertAliceShares(-1000000, 1, 4)
    s.assertAliceBalances(0, 0)

    s.bobDeposits(NewDeposit(0, 1200000, -1000000, 1))
    s.assertBobShares(-1000000, 1, 4)
    s.assertBobBalances(0, 0)

    s.aliceWithdraws(NewWithdrawal(4, -1000000, 1))
    s.assertAliceBalances(0, 1100000)

    s.bobWithdraws(NewWithdrawal(4, -1000000, 1))
    s.assertBobBalances(0, 1100000)
}

```

As can be seen, user Alice creates a one-sided pool with very large tick of `-1000000`, and gets only 4 shares for `1000000` of `TokenB`. When later user Bob deposits `1200000` of `TokenB` into the same pool, he also gets 4 shares. As a result, when withdrawing the funds from the pool, Bob obtains `100000` of `TokenB` less, and Alice – `100000` of `TokenB` more, compared to their original contributions.

## Problem Scenarios

The above integration test demonstrates one problematic scenario, when a user loses funds when depositing to, and withdrawing from a liquidity pool. Other problematic scenarios with the same root cause (shares rounding) seem possible.

## Recommendation

Rework the mathematics of shares calculation: always calculating the shares as the amount of `Token0`, as can be seen, poses problems with large tick values. One possibility would be to always calculate the amount of shares as the amount of the cheapest token from the trading pair. Another more precise but also more complicated approach would be e.g. to represent the conversion factor  $p$  between two tokens in the equation  $R_0 + pR_1$  as a rational number  $a/b$ , and then calculate the amount of shares as  $bR_0 + aR_1$ .

## Stealing of user funds via negative deposits

<b>Title</b>	Stealing of user funds via negative deposits
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	4 CRITICAL
<b>Impact</b>	3 HIGH
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	

### Involved artifacts

- [duality/tx.proto at v0.2.0](#)
- [duality/message\\_deposit.go at v0.2.0](#)

### Description

Some Duality transactions accept amounts as unbounded integers: see e.g. `MsgDeposit` and `MsgWithdrawal` in [duality/tx.proto at v0.2.0](#), but don't validate that the amounts are non-negative; see e.g. the `ValidateBasic()` of `MsgDeposit`. As a result, negative amounts are accepted, and pass all the way through transaction processing. The issue has been originally discovered by exploring the DEX model in the [Quint](#) specification language, and later confirmed via the integration test below.

```
func (s *MsgServerTestSuite) TestDepositNegative() {
    s.fundAliceBalances(1000, 10000)

    s.aliceDeposits(NewDeposit(-1000, 10000, 0, 1))

    s.assertAliceBalances(1000, 0)
    s.assertDexBalances(0, 10000)
    s.assertAliceShares(0, 1, 9000)

    s.aliceWithdraws(NewWithdrawal(9000, 0, 1))
    s.assertAliceBalances(1000, 10000)
    s.assertDexBalances(0, 0)
    s.assertAliceShares(0, 1, 0)
}
```

As can be seen, the negative amount of `-1000` for `TokenA` is accepted: the integration test passes. Initially there doesn't seem to be any issues; only the amount of shares `Alice` gets is reduced accordingly to the negative amount of `TokenA`.

## Problem Scenarios

The problem arises when the above acceptance of negative amounts is combined with the finding [Loss of user funds via shares rounding with large ticks](#), discovered independently in the course of this audit. In that finding we've demonstrated that a user may lose their tokens when they deposit to liquidity pools with large tick values: this somehow diminished the scope of the aforementioned finding. But as the present problem allows a malicious user to arbitrarily reduce the number of shares they get for a pool *with any ticks*, not only large ones, the combination of the two findings leads to an easily exploitable opportunity to steal funds from any pool, as the below integration test demonstrates.

```
func (s *MsgServerTestSuite) TestStealViaNegativeDepositAndSharesRounding() {
    s.fundAliceBalances(0, 10000)
    s.fundBobBalances(0, 19000)

    // Alice "sets the stage", by creating a pool
    // with a share that costs lots of coins
    s.aliceDeposits(NewDeposit(-9999, 10000, 0, 1))
    s.assertDexBalances(0, 10000)
    s.assertAliceBalances(0, 0)
    s.assertAliceShares(0, 1, 1) // 1 share costs 10000 TokenB

    s.bobDeposits(NewDeposit(0, 19000, 0, 1))
    s.assertDexBalances(0, 29000)
    s.assertBobBalances(0, 0)
    s.assertBobShares(0, 1, 1) // due to rounding, only 1 share for 19000 of TokenB

    s.aliceWithdraws(NewWithdrawal(1, 0, 1))
    s.assertDexBalances(0, 14500)
    s.assertAliceBalances(0, 14500) // Alice pockets extra 4500 of TokenB

    s.bobWithdraws(NewWithdrawal(1, 0, 1))
    s.assertDexBalances(0, 0)
    s.assertBobBalances(0, 14500) // 4500 of Bob's TokenB are stolen
}
```

The above is only one possible scenario; other scenarios due to accepting negative token amounts seem possible.

## Recommendation

We recommend to thoroughly examine all accepted transactions, and add checks that would reject transactions with negative token amounts.

## Wasteful usage of storage creates a potential for DOS attacks

<b>Title</b>	Wasteful usage of storage creates a potential for DOS attacks
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	3 HIGH
<b>Impact</b>	2 MEDIUM
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	<a href="https://github.com/duality-labs/duality/issues/397">https://github.com/duality-labs/duality/issues/397</a>

### Involved artifacts

- [duality/pool\\_reserves.go](#) at v0.2.0
- [duality/limit\\_order\\_tranche.go](#) at v0.2.0

### Description

Current implementation of `PoolReserves` and `LimitOrderTranches` store them in the Cosmos SDK store in the following way (demonstrated here on the example of `PoolReserves`, see `SetPoolReserves()`):

```
func (k Keeper) SetPoolReserves(ctx sdk.Context, pool types.PoolReserves) {
    // Wrap pool back into TickLiquidity
    tick := types.TickLiquidity{
        Liquidity: &types.TickLiquidity_PoolReserves{
            PoolReserves: &pool,
        },
    }

    store := prefix.NewStore(ctx.KVStore(k.storeKey),
types.KeyPrefix(types.TickLiquidityKeyPrefix))
    b := k.cdc.MustMarshal(&tick)
    store.Set(types.TickLiquidityKey(
        pool.PairID,
        pool.TokenIn,
        pool.TickIndex,
        types.LiquidityTypePoolReserves,
        pool.Fee,
    ), b)
}
```



with the `PoolReserves` type being

```
message PoolReserves {
  PairID pairID = 1;
  string tokenIn = 2;
  int64 tickIndex = 3;
  string reserves = 4 [
    (gogoproto.moretags) = "yaml:\"reserves\"",
    (gogoproto.customtype) = "github.com/cosmos/cosmos-sdk/types.Int",
    (gogoproto.jsontag) = "reserves",
    (gogoproto.nullable) = false
  ];
  uint64 fee = 5;
}
```

The implementation of `SetPoolReserves()` means that out of 5 fields, 4 are used as the store key, and all 5 are used as the store value. Instead, only the `reserves` could be stored as the value under the given key, and the `PoolReserves` data structure recreated on the fly by combining the key and the value.

## Problem Scenarios

Taking into account that a `PairID` contains 2 token IDs, with an additional token ID in `tokenIn`, each one possibly up to 127 bytes, this means that for any `PoolReserves` potentially more than 400 bytes can be stored as a value, not counting the space needed to store the key. As an unlimited number of `PoolReserves` can be created at almost no cost, this can be abused by an attacker to slow down the nodes, or even stop the blockchain if the storage requirements surpass the node capabilities.

## Recommendation

Store in the Cosmos SDK store only those parts of both `PoolReserves` and `LimitOrderTranches` data structures that not employed as a key, and recreate the data structures on the fly when reading them from the store by combining the key and the value parts.

## Denom collisions can be exploited to steal user funds

<b>Title</b>	Denom collisions can be exploited to steal user funds
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	3 HIGH
<b>Impact</b>	3 HIGH
<b>Exploitability</b>	2 MEDIUM
<b>Issue</b>	

### Involved artifacts

- [duality/deposit\\_denom.go at v0.2.0](#)

### Description

When forming deposit denom string for LP shares from a [DepositDenom](#) structure, the [following code](#) transforms the structure into a string representation:

```
func (d DepositDenom) String() string {
    // TODO: Revisit security of this.
    prefix := DepositDenomPairIDPrefix(d.PairID.Token0, d.PairID.Token1)
    return fmt.Sprintf("%s-t%d-f%d", prefix, d.Tick, d.Fee)
}

func DepositDenomPairIDPrefix(token0, token1 string) string {
    t0 := strings.ReplaceAll(token0, "-", "")
    t1 := strings.ReplaceAll(token1, "-", "")
    return fmt.Sprintf("%s-%s-%s", DepositSharesPrefix, t0, t1)
}
```

At the same time, the denom in Cosmos SDK is described by the [following regular expression](#):

```
// Denominations can be 3 ~ 128 characters long and support letters, followed by
either
// a letter, a number or a separator ('/', ':', '.', '_' or '-').
reDnmString = `[a-zA-Z][a-zA-Z0-9/[:-_]{2,127}`
```

As can be seen, the `-` symbol is allowed in a denom, but it's stripped from the deposit denom by replacing all `-` with empty strings. This creates a possibility for collisions, when two separate Cosmos SDK denoms are mapped to the same deposit denom, e.g. when `Token-A` denom is stripped into `TokenA`.

## Problem Scenarios

An adversary may abuse the above collision to steal funds of other users, via intentionally providing liquidity with a denom that will collide with another denom already present in some pool. This scenario is demonstrated by the below integration test.

```
func (s *MsgServerTestSuite) TestStealDeposit() {
    s.fundAliceBalances2(100, 0, 0)
    s.fundBobBalances2(100, 0, 100)

    // Alice deposits 100 TokenA @ tick0 => 100 shares
    s.aliceDeposits(NewDeposit(100, 0, 0, 10))
    s.assertAliceShares(0, 10, 100)
    s.assertLiquidityAtTick(sdk.NewInt(100), sdk.ZeroInt(), 0, 10)

    // Bob deposits 100 Token-A @ tick0 => 100 shares
    s.bobDeposits2([]*Deposit{NewDeposit(100, 0, 0, 10)},
        types.PairID{Token0: "Token-A", Token1: "TokenB"})
    s.assertLiquidityAtTick(sdk.NewInt(100), sdk.ZeroInt(), 0, 10)
    s.assertAliceShares2("TokenA", "TokenB", 0, 10, 100)
    s.assertBobShares2("TokenA", "TokenB", 0, 10, 100)
    s.assertBobShares2("Token-A", "TokenB", 0, 10, 100)

    // Bob withdraws 100 shares of (TokenA, TokenB) @ tick0 => 50 TokenA tokens
    s.bobWithdraws(NewWithdrawal(100, 0, 10))

    // Bob now has 150 TokenA
    s.assertBobBalances(150, 0)

    // The pool now has only 50 TokenA remaining
    s.assertLiquidityAtTick(sdk.NewInt(50), sdk.ZeroInt(), 0, 10)
}
```

To make this test work, we've had to introduce additional functions allowing to operate with denoms different from `TokenA` or `TokenB`:

```
func NewA2Coin(amt sdk.Int) sdk.Coin {
    return sdk.NewCoin("Token-A", amt)
}
```

```

func (s *MsgServerTestSuite) fundAccountBalances2(account sdk.AccAddress, aBalance,
bBalance, a2Balance int64) {
    aBalanceInt := sdk.NewInt(aBalance)
    bBalanceInt := sdk.NewInt(bBalance)
    a2BalanceInt := sdk.NewInt(a2Balance)
    balances := sdk.NewCoins(NewACoin(aBalanceInt), NewBCoin(bBalanceInt),
NewA2Coin(a2BalanceInt))
    err := FundAccount(s.app.BankKeeper, s.ctx, account, balances)
    s.Assert().NoError(err)
    s.assertAccountBalances(account, aBalance, bBalance)
}

func (s *MsgServerTestSuite) fundAliceBalances2(a, b, a2 int64) {
    s.fundAccountBalances2(s.alice, a, b, a2)
}

func (s *MsgServerTestSuite) fundBobBalances2(a, b, a2 int64) {
    s.fundAccountBalances2(s.bob, a, b, a2)
}

func (s *MsgServerTestSuite) bobDeposits2(deposits []*Deposit, pairID types.PairID) {
    s.deposits(s.bob, deposits, pairID)
}

func (s *MsgServerTestSuite) assertAliceShares2(token0 string, token1 string, tick
int64, fee, sharesExpected uint64) {
    s.assertAccountShares2(s.alice, token0, token1, tick, fee, sharesExpected)
}

func (s *MsgServerTestSuite) assertBobShares2(token0 string, token1 string, tick
int64, fee, sharesExpected uint64) {
    s.assertAccountShares2(s.bob, token0, token1, tick, fee, sharesExpected)
}

```

After discussing the finding with the developers and doing the additional research, we concluded that currently the denom collisions should not happen, because:

- Duality doesn't include the mechanism to introduce native chain denominations
- All denominations arriving over IBC, should have a specific structure, with the prefix `ibc/` followed by a bounded HEX string.

Nevertheless, we would like to point out that the current approach of stripping unwanted `-` characters when forming a denomination string is fragile, and rests on undocumented assumptions: that all denoms come over IBC, and have a specific structure. If these assumptions are not explicitly documented and tested against, then in the future either some code on the Duality side may introduce the unwanted denominations, or the IBC denomination format may change, and start allowing them.

## Recommendation

- **Explicitly document the assumptions** about acceptable token denominations on Duality.
- When forming a deposit denom string, check that the provided denomination follows the assumptions, and **reject the input if the assumptions are violated**.

- Make sure to test the code with various values for denominations, covering all possible denomination inputs to the system: both positive as well as negative cases wrt. the assumptions.

## Stakes querying not handling stakes with multiple coins

<b>Title</b>	Stakes querying not handling stakes with multiple coins
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	2 MEDIUM
<b>Impact</b>	2 MEDIUM
<b>Exploitability</b>	2 MEDIUM
<b>Issue</b>	

### Involved artifacts

- [duality/stake.go at v0.2.0](#)

### Description

Inside one of the stakes queries, complete [default clause](#) is being ignored since it mainly consists of iterating through coins in stake, but the loop is not going to be executed ever. The loop expression is `for low < high`, where `low` is set to 0, and `high` is set to length of coins array which is defined just a few lines above as an empty [array of coins](#). So, low and high are always 0, and the whole code inside the loop is not going to be executed.

### Problem Scenarios

Possible consequences of this issue include skipping coins(not including them in rewards distribution) from the stake even though they pass the condition for distribution. [The call to the function](#) is inside the distribution logic, thus making it critical part of the calculation of incentives program.

There is already a test that can be changed to confirm this, it can be rewritten in the following way(add coins to stake 4):

```
func TestDistributor(t *testing.T) {
    app := app.Setup(false)
    ctx := app.BaseApp.NewContext(false, tmtypes.Header{Height: 1, ChainID:
"duality-1", Time: time.Now().UTC()})

    gauge := types.NewGauge(
        1,
        false,
        types.QueryCondition{
            PairID: &dextypes.PairID{
                Token0: "TokenA",
                Token1: "TokenB",
```

```

        },
        StartTick: -10,
        EndTick: 10,
    },
    sdk.Coins{sdk.NewCoin("coin1", sdk.NewInt(100))},
    ctx.BlockTime(),
    10,
    0,
    sdk.Coins{},
    0,
)
rewardedDenom := dextypes.NewDepositDenom(&dextypes.PairID{Token0: "TokenA",
Token1: "TokenB"}, 5, 1).String()
nonRewardedDenom := dextypes.NewDepositDenom(&dextypes.PairID{Token0: "TokenA",
Token1: "TokenB"}, 12, 1).String()
allStakes := types.Stakes{
    {1, "addr1", ctx.BlockTime(), sdk.Coins{sdk.NewCoin(rewardedDenom,
sdk.NewInt(50))}},
    {2, "addr2", ctx.BlockTime(), sdk.Coins{sdk.NewCoin(rewardedDenom,
sdk.NewInt(25))}},
    {3, "addr2", ctx.BlockTime(), sdk.Coins{sdk.NewCoin(rewardedDenom,
sdk.NewInt(25))}},
    {4, "addr3", ctx.BlockTime(), sdk.Coins{sdk.NewCoin(nonRewardedDenom,
sdk.NewInt(50)), sdk.NewCoin(rewardedDenom, sdk.NewInt(25))}},
}

...

```

and [here](#), the function `CoinsPassingQueryCondition` will return 0 coins.

## Recommendation

The part of code that conditions the processing of multiple coins from stakes: `for low < high` either has to be changed, or the definition of array inside the mentioned function: `coins := sdk.Coins{}` has to be changed.

## Tick and Fee inputs are not validated

<b>Title</b>	Tick and Fee inputs are not validated
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	2 MEDIUM
<b>Impact</b>	1 LOW
<b>Exploitability</b>	3 HIGH
<b>Issue</b>	

### Involved artifacts

- [duality/tx.proto at v0.2.0](#)
- [duality/message\\_deposit.go at v0.2.0](#)

### Description

Many Duality transactions accept tick indexes (as `int64`) and fees (as `uint64`): see e.g. `MsgDeposit` and `MsgWithdrawal` in [duality/tx.proto at v0.2.0](#). In the implementation, semantically these values are assumed to belong to the range  $[-352437, 352437]$ , see [price.go](#). The problem is that these constraints are not enforced anywhere on the transaction inputs; see e.g. the `ValidateBasic()` of `MsgDeposit`. As a result, out-of-range values for ticks and fees can be passed, and enter transaction processing with the consequences that can't be fully predicted. As an example see the [below fragment](#) of `DepositCore()`:

```
feeUInt := utils.MustSafeUint64(fee)
lowerTickIndex := tickIndex - feeUInt
upperTickIndex := tickIndex + feeUInt
```

Variables `tickIndex` and `fee` come directly from transaction parameters, and because of not being validated can be in the range  $[-9223372036854775808, 9223372036854775807]$  for `tickIndex`, and  $[0, 9223372036854775807]$  for `fee` (the last one because of `utils.MustSafeUint64` check). Notice, for example, that supplying both `tickIndex` and `fee` to be  $9223372036854775807$ , will compute `lowerTickIndex` and `upperTickIndex` as  $0$  and  $-2$ , respectively, due to overflows of the type `int64`. Notice that as a result of overflows we have:

1. Both values `lowerTickIndex` and `upperTickIndex` are the acceptable range of tick indices.



2. The value of `lowerTickIndex` is greater than that of `upperTickIndex`, i.e. opposite to what the system semantics normally expects.

This particular transaction will fail later in the shares calculation, but we believe the example still demonstrates well enough the dangers of accepting unvalidated inputs.

## Problem Scenarios

We rate the severity of this finding as **Medium** only because we have not been able to identify in the limited time scope of the projects the concrete values that would have severe system impact. There might be still circumstances, now or at a later stage, when a particular combination of large, unvalidated inputs for `tickIndex` and `fee` will pass all processing of a certain transaction, and would allow an attacker to create the system state in which they can e.g. steal user funds. In that case the severity of the present finding would immediately become **Critical**.

## 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  $[-352437, 352437]$  for tick indices:

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

## Incomplete validation of MsgDeposit

<b>Title</b>	Incomplete validation of MsgDeposit
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	1 LOW
<b>Impact</b>	1 LOW
<b>Exploitability</b>	2 MEDIUM
<b>Issue</b>	<a href="https://github.com/duality-labs/duality/issues/396">https://github.com/duality-labs/duality/issues/396</a>

### Involved artifacts

- [duality/message\\_deposit.go](#) at v0.2.0
- [duality/tx.proto](#) at v0.2.0

### Description

`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).

### Problem Scenarios

A user might submit a `deposit` transaction with unbalanced length of the `Options` array; the transaction will then pass the `ValidateBasic check`, and will be thus admitted to the block, but will fail during processing in `DepositCore`.

### Recommendation

Extend `ValidateBasic checks` to validate also the length of the `Options` array.

## Optimize stakes querying

<b>Title</b>	Optimize stakes querying
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	1 LOW
<b>Impact</b>	1 LOW
<b>Exploitability</b>	1 LOW
<b>Issue</b>	

### Involved artifacts

- [duality/stake.go at v0.2.0](#)

### Description

1. There is room for improvement in terms of optimization in [stakes querying code](#). First, [the creation of `idMemo` map](#) can be enhanced with map size, since it is a known value at coding time - the length of `tickStakeIds`. That way, growing a map in multiple steps is avoided.
2. Another point to improve in this function is to remove one [unnecessary loop](#). Array `resultIds` does not need to be created since it is used just to iterate overlapping stake ids. These ids are identified in the first loop and can be used right away.

### Problem Scenarios

Obvious issue with this is that unnecessary computations are being executed and valuable energy and resources being wasted.

Another scenario, since the loops are called from distribution logic, if the number of distribution executions increases overtime, the resource usage is increased also.

### Recommendation

The optimization explained in the description under 1. could be rewritten in the following way:

```
idMemo := make(map[uint64]bool, len(tickStakeIds))
for _, id := range tickStakeIds {
    idMemo[id] = true
}
```

and the code noted in under 2. in the description as:

```
results := make([]*types.Stake, 0)
for _, id := range timeStakeIds {
    if _, ok := idMemo[id]; ok {
        stake, err := k.GetStakeByID(ctx, id)
        if err != nil {
            // This represents a db inconsistency
            panic(err)
        }
        results = append(results, stake)
    }
}
```

## Number of epochs for gauge creation is not validated

<b>Title</b>	Number of epochs for gauge creation is not validated
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	1 LOW
<b>Impact</b>	1 LOW
<b>Exploitability</b>	2 MEDIUM
<b>Issue</b>	

### Involved artifacts

- [duality/gauge.go at v0.2.0](#)
- [duality/cli\\_test.go at v0.2.0](#)
- [duality/messages\\_test.go at v0.2.0](#)

### Description

When creating a gauge, a user specifies number of epochs over which the rewards distribution from created gauge is being performed: `numEpochsPaidOver`. While there is a [limitation on number of created gauges](#) (default value is 20 by the current design), [there is no validation or limitation of number of epochs](#) a user inputs.

The distribution logic is based on iterating over all active gauges when rewards calculation is executed. Also, the current flow is that the rewards are distributed once per day.

### Problem Scenarios

The core distribution logic is based on iterating over all active gauges and, among other things, checking if there are epochs remaining to be distributed over ([calculation of remaining epochs](#)). Since the `numEpochsPaidOver`, which is initially set by user, is not validated nor limited, a malicious user can set a large number, thus requiring a large number of days (epochs) to pass until there is no more remaining epochs in the specific gauge.

Moreover, since the maximum number of gauges is set to 20 by default, there is also a possibility of a group of malicious users creating multiple gauges with large number of epochs where the possible, worst case scenario is an incentive program with stopped creation of gauges for a long period (maximum number of created gauges is reached, and until all gauges “run out of epochs”, no new ones can be created), and in those gauges the rewards distributed are of a small amount since the amount of coins in non perpetual gauge is divided by number of epochs.

To confirm the validation is not present, some existing tests are modified in the following way:

```
//cli_test.go
//NumEpochsPaidOver set to maximum uint64 value
func TestNewCreateGaugeCmd(t *testing.T) {
```

```

testTime := time.Unix(1681505514, 0).UTC()
desc, _ := cli.NewCreateGaugeCmd()
tcs := map[string]osmocli.TxCliTestCase[*types.MsgCreateGauge]{
    "basic test": {
        Cmd: fmt.Sprintf("TokenA<>TokenB 0 100 100TokenA,100TokenB
18446744073709551615 0 --from %s", testAddresses[0]),
        ExpectedMsg: &types.MsgCreateGauge{
            IsPerpetual: false,
            Owner:      testAddresses[0].String(),
            DistributeTo: types.QueryCondition{
                PairID: &dextypes.PairID{Token0: "TokenA", Token1: "TokenB"},
                StartTick: 0,
                EndTick: 100,
            },
            Coins: sdk.NewCoins(
                sdk.NewCoin("TokenA", sdk.NewInt(100)),
                sdk.NewCoin("TokenB", sdk.NewInt(100)),
            ),
            StartTime:      time.Unix(0, 0).UTC(),
            NumEpochsPaidOver: 18446744073709551615,
            PricingTick:      0,
        },
    },
},

```

```

//msgs_test.go
//NumEpochsPaidOver set to 1000000
func TestMsgCreatePool(t *testing.T) {
    .
    .
    .
    {
        name: "invalid num epochs paid over",
        msg: createMsg(func(msg MsgCreateGauge) MsgCreateGauge {
            msg.NumEpochsPaidOver = 1000000
            return msg
        }),
        expectPass: true,
    },
}

```

Similar scenario can be created with start and end tick in mind (from QueryCondition in gauges). The unwanted ending scenario would be an area of ticks with a lot of gauges for rewards there, but others not being rewarded.

## Recommendation

There should be a reasonable limitation created for a number of epochs per gauge, and then a user selected value should be validated against that creation.

## Invalid Multi Hop Swap routes cause panic

<b>Title</b>	Invalid Multi Hop Swap routes cause panic
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	1 LOW
<b>Impact</b>	1 LOW
<b>Exploitability</b>	2 MEDIUM
<b>Issue</b>	

### Involved artifacts

- [duality/core.go at v0.2.0](#)
- [duality/multihop\\_swap.go at v0.2.0](#)

### Description

If the routes (`[]MultiHopRoute`) are filled so that the last `MultiHopRoute` contains only the exit denom, the execution of the `MultiHopSwap` message will cause panic. More precisely, the panic occurs during the [Amounts comparison in the `MultiHopSwapCore\(\)`](#) function because the value of `routeCoinOut.Amount` is nil.

### Problem Scenarios

This case was confirmed by an integration test with the following routes:

```
routes := [][]string{
    {"TokenA", "TokenB", "TokenC", "TokenX"},
    {"TokenA", "TokenB", "TokenD", "TokenX"},
    {"TokenX"},
}
```

### Corresponding stack trace:

```
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/suite.go:63: test panicked: runtime error: invalid memory address or nil pointer dereference
goroutine 8 [running]:
runtime/debug.Stack()
/home/aleksandar/go/src/runtime/debug/stack.go:24 +0x65
github.com/stretchr/testify/suite.failOnPanic(0xc000f076c0)
/home/aleksandar/go/pkg/mod/github.com/stretchr/testify@v1.7.1/suite/suite.go:63 +0x3e
panic({0x18583e0, 0x321c750})
```

```

/home/aleksandar/.go/src/runtime/panic.go:884 +0x212
math/big.(*Int).Cmp(0x24e11d0?, 0xc00049cbc0?)
/home/aleksandar/.go/src/math/big/int.go:324 +0x25
github.com/cosmos/cosmos-sdk/types.lt(...)
/home/aleksandar/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.45.11-ics/types/int.go:23
github.com/cosmos/cosmos-sdk/types.Int.LT(...)
/home/aleksandar/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.45.11-ics/types/int.go:224
github.com/duality-labs/duality/x/dex/keeper.Keeper.MultiHopSwapCore({{0x24e11d0, 0xc00049cbc0},
{0x24c0a00, 0xc0004e1200}, {0x24c0a00, 0x0}, {{0x24e11d0, 0xc00049cbc0}, 0xc000122900, {0x24c0a00, ...}, ...}, ...})
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/core.go:303 +0x46d
github.com/duality-labs/duality/x/dex/keeper.msgServer.MultiHopSwap({{{0x24e11d0, 0xc00049cbc0},
{0x24c0a00, 0xc0004e1200}, {0x24c0a00, 0x0}, {{0x24e11d0, 0xc00049cbc0}, 0xc000122900, {0x24c0a00, ...}, ...}, ...})
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/msg_server.go:179 +0x159

```

## Recommendation

It should add validation to ValidateBasic that verifies that all MultiHopRoute contains 2 or more elements.



## Improvements for better efficiency

<b>Title</b>	Improvements for better efficiency
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	0 INFORMATIONAL
<b>Impact</b>	0 NONE
<b>Exploitability</b>	0 NONE
<b>Issue</b>	

## Involved artifacts

- [duality/core.go](#) at v0.2.0
- [duality/pair\\_helper.go](#) at v0.2.0

## Description

There are several places in the code that affect the efficiency of transaction execution:

- The **for loop** for initializing *amounts0Deposited* and *amounts1Deposited* in the `DepositCore()` function is not needed. These arrays get **calculated values** in the *for* loop below, which also iterates through *amount0*.
- No need to continue execution if a `TrancheUser` exists, but a `Tranche`, active or inactive, does not exist in `WithdrawFilledLimitOrderCode()`. In that case, `TrancheUser` will be **saved** without changes to the object, and then **an error will be returned** because both *amountOutTokenOut* and *remainingTokenIn* are zero.
- There is no need to create a new `PairID` in the *for* loop in the `WithdrawCore()` function. The corresponding **pairID** is created before the *for* loop and can be used to create a new *sharesId*.
- The equality of *baseToken* and *token0* should be checked before the *for* loop in the `NormalizeAllTickIndexes()` function. If they are equal, neither *tickIndexes* should be changed.

## Problem Scenarios

Unnecessary iterations in the *for* loop and the creation of new elements negatively affect the consumption of resources and the time of execution of transactions.

## Recommendation

`NormalizeAllTickIndexes` can be modified as follows:

```
func NormalizeAllTickIndexes(baseToken, token0 string, tickIndexes []int64) []int64 {
    if baseToken != token0 {
        for i, idx := range tickIndexes {
```

```
        tickIndexes[i] = idx * -1
    }
}

return tickIndexes
}
```

## Large string for token causes panic

<b>Title</b>	<a href="#">Large string for token causes panic</a>
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	1 LOW
<b>Impact</b>	1 LOW
<b>Exploitability</b>	1 LOW
<b>Issue</b>	

### Involved artifacts

- [duality/core.go at v0.2.0](#)
- [duality/liquidity.go at v0.2.0](#)

### Description

Several tx messages have string tokens as parameters. As there is no check of the length of that string, nor of the validity of the token, it is possible to cause a panic during the execution of the transaction.

### Problem Scenarios

This case was tested with an integration test for the PlaceLimitOrder message. A string of length 35700 characters is set for the *tokenIn* value. Panic occurs when validating the coin created in the **Swap()** function:

```
func NewCoin(denom string, amount Int) Coin {
    coin := Coin{
        Denom:  denom,
        Amount: amount,
    }

    if err := coin.Validate(); err != nil {
        panic(err)
    }

    return coin
}
```

Also, it was established that an invalid denom of standard length does not lead to panic.

Corresponding stack trace:

```

/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/suite.go:63: test panicked: invalid denom: *LARGE STRING*
goroutine 15 [running]:
runtime/debug.Stack()
/home/aleksandar/.go/src/runtime/debug/stack.go:24 +0x65
github.com/stretchr/testify/suite.failOnPanic(0xc000502ea0)
/home/aleksandar/go/pkg/mod/github.com/stretchr/testify@v1.7.1/suite/suite.go:63 +0x3e
panic({0x1821c80, 0xc000e316f0})
/home/aleksandar/.go/src/runtime/panic.go:884 +0x212
github.com/cosmos/cosmos-sdk/types.NewCoin({0x1b344de, 0x8bb3}, {0x24c4fe0?})
/home/aleksandar/go/pkg/mod/github.com/cosmos/cosmos-sdk@v0.45.11-ics/types/coin.go:23 +0x51
github.com/duality-labs/duality/x/dex/keeper.Keeper.Swap({{0x24e57b0, 0xc0003ba320}, {0x24c4fe0, 0xc000d52210}, {0x24c4fe0, 0x0}, {{0x24e57b0, 0xc0003ba320}, 0xc000126b90, {0x24c4fe0, ...}, ...}, ...)
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/liquidity.go:172 +0x43d
github.com/duality-labs/duality/x/dex/keeper.Keeper.SwapWithCache({{0x24e57b0, 0xc0003ba320}, {0x24c4fe0, 0xc000d52210}, {0x24c4fe0, 0x0}, {{0x24e57b0, 0xc0003ba320}, 0xc000126b90, {0x24c4fe0, ...}, ...}, ...)
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/liquidity.go:204 +0x1f8
github.com/duality-labs/duality/x/dex/keeper.Keeper.PlaceLimitOrderCore({{0x24e57b0, 0xc0003ba320}, {0x24c4fe0, 0xc000d52210}, {0x24c4fe0, 0x0}, {{0x24e57b0, 0xc0003ba320}, 0xc000126b90, {0x24c4fe0, ...}, ...}, ...)
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/core.go:384 +0x573
github.com/duality-labs/duality/x/dex/keeper.msgServer.PlaceLimitOrder({{{0x24e57b0, 0xc0003ba320}, {0x24c4fe0, 0xc000d52210}, {0x24c4fe0, 0x0}, {{0x24e57b0, 0xc0003ba320}, 0xc000126b90, {0x24c4fe0, ...}, ...}, ...})
/home/aleksandar/Projects/Duality/Main/duality/x/dex/keeper/msg_server.go:118 +0x35e

```

## Recommendation

It is necessary to add validation related to the correctness of the token - whether it exists, length or format string. It is necessary to cover TokenA and TokenB from MsgDeposit, TokenA and TokenB from MsgWithdrawal, TokenIn and TokenOut from MsgPlaceLimitOrder and all values of Routes from MsgMultiHopSwap with some validation.

## Minor code improvements for Incentives module

<b>Title</b>	Minor code improvements for Incentives module
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	0 INFORMATIONAL
<b>Impact</b>	0 NONE
<b>Exploitability</b>	0 NONE
<b>Issue</b>	

### Involved artifacts

- [duality/gauge.go](#) at v0.2.0
- [duality/stake.go](#) at v0.2.0
- [duality/distributor.go](#) at v0.2.0
- [duality/msg\\_server.go](#) at v0.2.0

### Description

- [Here](#), `k.GetLastGaugeID(ctx)` can be changed with `numGauges` from few lines before, it is one less load from the store;
- Consider if `SetLastStakeID` should be called after [deleting](#) stake. If there is a case of unstaking before new stake, and during unstaking, stake with `LastStakeID` was deleted, then when creating a new stake, one of the IDs would be left without actual stake. So the store would keep a key with no purpose.
- There is a room for improvement in [this](#) loop:
  - `stakesSumCache` can be sized just like `gaugeStakes` to avoid expensive resizing inside loop
- Since the owner address is changed to `AccAddress` [here](#), you could pass it as an argument [here](#) and avoid creating it the same way [here](#).

### Problem Scenarios

### Recommendation

## Minor code improvements for the DEX module

<b>Title</b>	Minor code improvements for the DEX module
<b>Project</b>	Duality: Dex and Incentives modules
<b>Type</b>	IMPLEMENTATION
<b>Severity</b>	0 INFORMATIONAL
<b>Impact</b>	0 NONE
<b>Exploitability</b>	0 NONE
<b>Issue</b>	

### Reversed argument order in related functions

In [limit\\_order\\_tranche\\_user.go](#), the order of arguments in closely related functions is reversed:

- [GetLimitOrderTrancheUser\(address string, trancheKey string\)](#)
- [RemoveLimitOrderTrancheUserByKey\(trancheKey string, address string\)](#)

This is confusing for a developer; as both arguments are plain strings supplying arguments in the reversed order won't be caught by a type checker, and may lead to unneeded complications and debugging.


## 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
<b>High</b>	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
<b>Medium</b>	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)
<b>Low</b>	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)
 <b>None</b>	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:

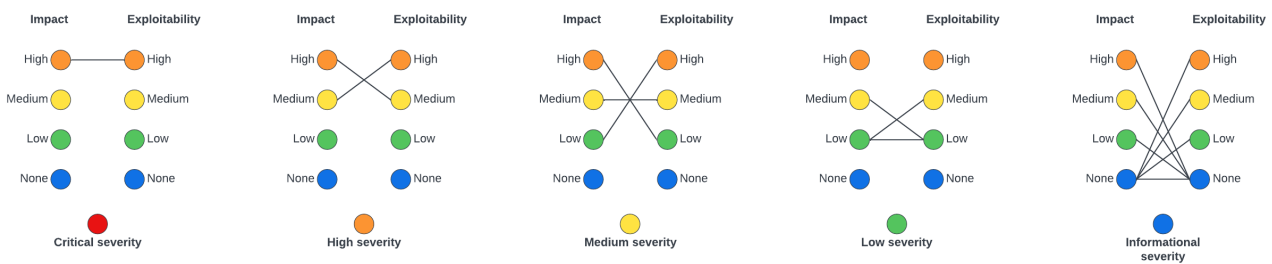
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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
<b>High</b>	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
<b>Medium</b>	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
<b>Low</b>	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
● <b>None</b>	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
● <b>Critical</b>	Halting of chain via a submission of a specially crafted transaction



Severity Score	Examples
<b>High</b>	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
<b>Medium</b>	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
<b>Low</b>	2x increase in node computational requirements via coordinated withdrawal of all user tokens
 <b>Informational</b>	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 Systems 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 Systems 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 Systems 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.