



# **Security Audit Report**

dYdX 2023 Q4: Modules and Conditional Orders

Authors: Aleksandar Ljahovic, Mirel Dalcekovic

Last revised 9 May, 2024

# Table of Contents

<b>Audit Overview .....</b>	<b>1</b>
The Project .....	1
Scope of this report .....	1
Conducted work .....	1
Conclusions .....	2
Further Increasing Confidence .....	2
<b>Audit Dashboard .....</b>	<b>3</b>
Severity Summary .....	3
<b>Findings .....</b>	<b>4</b>
Optimizing the Determination of User Fee Tiers .....	5
x/stats Module Minor Changes .....	7
x/clob Module Minor Changes .....	8
Potential Network Degradation Caused by Short-Term Order Spamming .....	9
Flaws in Subaccount Determination for Liquidation and Use of the math.rand Package .....	11
Deferred Liquidation for the Most Underwater Subaccount .....	14
Potential Chain Halt due to Panic in ABCI Phases .....	17
<b>Appendix: Vulnerability Classification .....</b>	<b>19</b>
Impact Score .....	19
Exploitability Score .....	19
Severity Score .....	20
<b>Disclaimer .....</b>	<b>22</b>

# Audit Overview

## The Project

In November 2023, dYdX has continued its engagement with [Informal Systems](#) to work on partnership and conduct security audit of selected v4 chain modules and Conditional Orders in dYdX's open source software. In this stage of our collaboration, the defined scope consisted of the following artifacts:

1. `x/fee-tiers` module
2. `x/stats` module
3. `x/block-time` module
4. Conditional Orders implemented in `x/clob` module.

## Scope of this report

The agreed-upon work plan consisted of the following tasks:

- Code inspection of `x/fee-tiers`, `x/stats`, `x/block-time` modules, primarily utilized as libraries within the dYdX chain, especially from other modules.
- Code inspection of the implemented Conditional Orders, categorized as Stateful (long-term) orders, along with an examination of potential issues arising from code changes introduced between the commit hash of the previous audit and the commit hash agreed upon for auditing during this project.

## Conducted work

During the kick-off meeting, representatives from Informal Systems and dYdX team agreed on the audit process and identified the best approach for this project's organization.

## Timeline

- 27.11.2023: Audit partnership Q4 2023 start: first overview of the potential scope and preparation of the questions for the kick off meeting scheduled with dYdX team
- 29.11.2023. Kick off meeting and high level code walkthrough by dYdX team.
- 07.12.2023. Sync meeting. The Auditing team shared the threat models created and presented the findings surfaced so far. The discussion revolved around determining the approach for analyzing liquidations and deleveraging code changes occurring between audits. There was consideration regarding whether to focus solely on assessing the impact of Conditional Orders or to include an examination of the introduced changes as well.
- 14.12.2023. Closure meeting. The Auditing team presented recent findings, with a particular emphasis on the evaluation of performance and stress testing conducted before configuring CometBFT parameters like mempool size and block size. The discussion delved into understanding potential network performance degradation that could arise from short-term order spamming and how these factors should be considered in the overall assessment. The draft report would be sent on December 15th. The dYdX engineering team agreed to examine the findings and furnish feedback and remediation plans to the auditing team. Uncertainty persisted regarding certain changes introduced in the liquidation mechanisms, as well as adjustments to mempool and consensus parameters. It was decided that final reports would be issued after comprehensive reviews by both the engineering and legal teams.
- 15.12.2023. Draft Report sent.
- 29.01.2024. - 07.02.2024. The Engineering team conducted a review and offered feedback to the Auditing team. Ongoing discussions took place on the Slack channel, prompting the Auditing team to reassess severity levels and consider additional scenarios that could result in these issues.

- 29.03.2024. Final reports were generated, offering comprehensive explanations for the disputed issues, along with detailed explanations of their respective severity assessments.
- 09.05.2024. New revision of final report audit restitution. Updated report to incorporate minor dYdX Trading legal team revisions for clarity and accuracy and updated finding status.

## Conclusions

The overall evaluation of the dYdX v4 chain codebase reveals a commendable level of quality, with well-structured and readable code that enhances comprehension and continuity. It's important to note, however, that the `x/clob` module is notably complex, and the logic within it can be challenging to follow. It is once again concluded that enriching the documentation with precise specifications of existing implementation protocol mechanisms could be beneficial. Providing concise specifications for each existing algorithm, including a clear explanation of behavior, expected inputs, and outputs, would be highly beneficial. While the current documentation is extensive, it primarily emphasizes implementation details (both current and planned) rather than specifying the intended behavior.

While we identified several minor issues that do not require immediate resolution, one issue pertaining to liquidations necessitates clarification.

In the course of evaluating correctness of spam protection mechanisms, it was determined that optimizing CometBFT's consensus and mempool parameters, as well as conducting an analysis of network performance under high load, could potentially offer advantages. It is important to note that the scope of our audit did not explicitly include an in-depth examination of these aspects, and thus, we did not have access to information regarding any prior efforts or performance benchmarks conducted by the dYdX team in this specific domain. Any potential improvements in the mentioned areas would necessitate a dedicated investigation beyond the parameters of this audit time-frame.

## Further Increasing Confidence

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

As a possible way forward in our partnership, we propose conducting thorough end-to-end testing and/or creating quint models for complex flows to identify more intricate protocol issues. The main focus should be on the mechanisms implemented within the `x/clob` module, specifically the lifecycle of short-term and stateful orders, orders matching and liquidation and deleveraging processes. The analysis should prioritize examining potential vulnerabilities that could arise in live deployments of dYdX open source software, including those that could pertain to user funds, the chain itself, and insurance funds.

# Audit Dashboard

## Target Summary

- **Type:** Specification and Implementation
- **Platform:** Go
- **Artifacts**
  - `x/clob` : `x/clob` [code](#)
  - `x/block-time` : [code](#)
  - `x/fee-tiers` : [code](#)
  - `x/stats` : [code](#)
- **Commit hash/tag:** [protocol/v1.0.1](#)

## Engagement Summary

- **Dates:** 27.11.2023 until 15.12.2023.
- **Number of person weeks on project:** 6
- **Method:** Manual code review & protocol analysis
- **Employees Engaged:** 2

## Severity Summary

Finding Severity	#
Critical	1
High	0
Medium	2
Low	1
Informational	3
<b>Total</b>	7

## Findings

Title	Type	Severity	Status	Issue
Potential Chain Halt due to Panic in ABCI Phases	IMPLEMENTATION	CRITICAL	RESOLVED	<a href="https://github.com/dydxprotocol/v4-chain/pull/1342">https://github.com/dydxprotocol/v4-chain/pull/1342</a>
Deferred Liquidation for the Most Underwater Subaccount	PROTOCOL IMPLEMENTATION	MEDIUM	ACKNOWLEDGED	
Flaws in Subaccount Determination for Liquidation and Use of the math.rand Package	IMPLEMENTATION PROTOCOL	MEDIUM	ACKNOWLEDGED	
Optimizing the Determination of User Fee Tiers	IMPLEMENTATION	LOW	ACKNOWLEDGED	
x/stats Module Minor Changes	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
x/clob Module Minor Changes	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Potential Network Degradation Caused by Short-Term Order Spamming	PROTOCOL	INFORMATIONAL	ACKNOWLEDGED	

## Optimizing the Determination of User Fee Tiers

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	IMPLEMENTATION
Severity	LOW
Impact	LOW
Exploitability	LOW
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- [x/feetiers/keeper/keeper.go](https://x/feetiers/keeper/keeper.go)

### Description

In `getUserFeeTier` function, when determining fee-tier for a user there are two ways of optimizing the speed and memory usage:

1. Avoid redundant assignments: `this` could be moved outside of the loop, speeding up the execution of determining the appropriate fee tier for certain user stats.
2. Precompute values outside the loop: values like `bigUserTotalNotional` and `len(tiers)` will not change within the loop. It's more efficient to compute these values once before entering the loop.

### Problem Scenarios

This function `getUserFeeTier` is called from:

- `UserFeeTier` query and this is not an issue considering that there won't be a large number of fee-tiers defined in the system and query is executed per user.
- `GetPerpetualFeePpm` function when iterating through all open orders for a subaccount [here](#).  
`GetMEVDataFromOperations` function when iterating through all operations. Recording MEV metrics is done (if enabled for a node) from `ProcessProposal`, which should not last long.  
`ProcessSingleMatch` function when determining the fee tier for maker and taker fee `ppms` [here](#).  
Places that could call `getUserFeeTier` multiple times which could impact on more significant memory and time consumption.

## Recommendation

As suggested above. Consider removing all the computation operations in the flows listed above outside the loops.



## x/stats Module Minor Changes

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- [x/stats/keeper/keeper.go](https://xstats/keeper/keeper.go)

### Description

- Minor code improvement: no need for reading the `epochInfo` from the store if there are no fills made in the block. Move this [line](#) after this [condition](#) is checked.
- Defensive coding practice: During the execution of `EndBlocker` there are unsafe arithmetic operations and possibility of `uint` wrap-around: [here](#). Even though it seems this is highly unlikely to happen (since it would consider large trading volumes), we suggest adding a check. If not triggering a panic, at the very least, logging the error is advised.

### Recommendation

As explained above.

## x/clob Module Minor Changes

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- [x/clob/keeper/](#)

### Description

- There is no need to convert slice collections to a set [here](#) and [here](#). It's more efficient to just iterate through the slice and check if `orderId` exists. Slice uniqueness is better checked when adding elements to that collection.
- `isTakerLiquidation` is already defined at the beginning of the `persistMatchedOrders` function, so that it can be used at places [process\\_single\\_match.go#L323](#) and [process\\_single\\_match.go#L359](#).

### Recommendation

As explained above.

It could also be useful to consider the benefits of switching from slice collections to sets in

`ProcessProposerMatchesEvents`. Depending on the usage of this struct, slices, and size, it could be concluded, after benchmark testing, that other data structures could be more useful.

## Potential Network Degradation Caused by Short-Term Order Spamming

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	PROTOCOL
Severity	INFORMATIONAL
Impact	UNKNOWN
Exploitability	MEDIUM
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- </protocol/app/ante.go>
- [/protocol/x/clob/rate\\_limit](/protocol/x/clob/rate_limit)
- [/protocol/x/clob/types/block\\_rate\\_limit\\_config.go](/protocol/x/clob/types/block_rate_limit_config.go)
- [/protocol/x/clob/keeper/equity\\_tier\\_limit.go](/protocol/x/clob/keeper/equity_tier_limit.go)
- [/protocol/x/clob/types/equity\\_tier\\_limit\\_config.go](/protocol/x/clob/types/equity_tier_limit_config.go)

### Description

There are several mechanisms of spam protection, built in by CometBFT and Cosmos SDK - and custom dYdX ones. All of them are allowing a user to place 400 transactions containing placement or cancellation of short term orders, per block in the mempool without any fees paid or amount of balance on a subaccount (for `Immediate-or-Cancel` and `Fill-or-Kill` orders). Also, one user could sent transactions to more nodes at a same time.

There is also a possibility for user to place additional short term transactions, free but in limited number depending on the notional value of the subaccount.

Facts:

- `x/clob` transactions do not require gas fees - those types of tx are `skipped in antehandler processing`
- sequence numbers `are not checked` for short term orders
- short term order transactions placing and cancellations `are rate limited` per 1 block to 200 transactions.
- equity tier limiting `does not exist` for short term orders of type `Immediate-or-Cancel` and `Fill-or-Kill` - those orders are allowed regardless of the net collateral of the subaccount.

## Problem Scenarios

Short-term orders have the potential to saturate the node's receiving mempool, consuming a substantial amount of bandwidth as they are propagated throughout the network. This could cause a strain on the affected nodes. Given that short-term orders persist for a duration of 20 blocks and are propagated through the network, they could degrade the performances of nodes affected.

## Recommendation

In conclusion, it is crucial to conduct stress testing to determine the viability of spamming and evaluate the potential consequences. Creating scripts and placing large number of short term orders over time to multiple nodes and monitor the behavior of the chain.

Additionally, careful adjustment of network parameters is necessary to achieve the desired system behavior. Parameters such as block size, mempool size, and fees should be considered as an additional way of protecting from spam:

- The parameter `max_txs_bytes` could be used to limit the size of mempools. Current [dYdX default config](#) parameter value in CometBFT is 1GB.
- Maximum block size `MaxBytes` in dYdX is 21 MB [here](#).

## Flaws in Subaccount Determination for Liquidation and Use of the math.rand Package

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	IMPLEMENTATION PROTOCOL
Severity	MEDIUM
Impact	HIGH
Exploitability	LOW
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- </x/clob/keeper/liquidations.go>
- </x/clob/keeper/random.go>

### Description

Current implementation of the liquidations logic:

- Uses `math.rand` Golang package in `LiquidateSubaccountsAgainstOrderbook` to 'randomly' select which subaccounts will be liquidated in case more than `MaxLiquidationAttemptsPerBlock` number of subaccounts are candidates - [here](#):

```
// Get the liquidation order for each subaccount.
// Process at-most `MaxLiquidationAttemptsPerBlock` subaccounts, starting
// from a pseudorandom location
// in the slice. Note `numSubaccounts` is guaranteed to be non-zero at this
// point, so `Intn` shouldn't panic.
pseudoRand := k.GetPseudoRand(ctx)
liquidationOrders := make([]types.LiquidationOrder, 0)
numLiqOrders := lib.Min(numSubaccounts, int(k.Flags.MaxLiquidationAttemptsP
erBlock))
indexOffset := pseudoRand.Intn(numSubaccounts)

startGetLiquidationOrders := time.Now()
for i := 0; i < numLiqOrders; i++ {
    index := (i + indexOffset) % numSubaccounts
    subaccountId := subaccountIds[index]
    liquidationOrder, err := k.MaybeGetLiquidationOrder(ctx, subaccountId)
```

```

        if err != nil {
            // Subaccount might not always be liquidatable since liquidation
            daemon runs
            // in a separate goroutine and is not always in sync with the
            application.
            // Therefore, if subaccount is not liquidatable, continue.
            if errors.Is(err, types.ErrSubaccountNotLiquidatable) {
                telemetry.IncrCounter(1, metrics.MaybeGetLiquidationOrder,
                metrics.SubaccountsNotLiquidatable, metrics.Count)
                continue
            }

            // Return unexpected errors.
            return err
        }

        liquidationOrders = append(liquidationOrders, *liquidationOrder)
    }

```

The `pseudoRand` is of type `Rand` initialized with the following `seed`:

```
k.blockTimeKeeper.GetPreviousBlockInfo(ctx).Timestamp.Unix() .
```

Initializing the seed generator with a fixed value, as explained [here](#) should guarantee the determinism needed in this part of the code, where preparing liquidation orders about to be executed in the following block.

- Enables proposer to propose the subaccounts for liquidation, and subaccount determination is not verified at the consensus level.

## Problem Scenarios

The mentioned `math.rand` package has a [note](#):

*Package rand implements pseudo-random number generators suitable for tasks such as simulation, but it should not be used for security-sensitive work.*

It is not known whether or not this package has been audited. Also introducing a dependency on external libraries that could potentially give different results if different versions of Golang are used is considered risky.

Due to the absence of verification for liquidatable subaccounts at a consensus level, there's a risk that a malicious actor could manipulate the code to select favorable accounts, regardless of the use of the `math.rand` package.

## Recommendation

Consider introducing some other mechanism of selecting which subaccounts should be skipped in processing in order to keep the `MaxLiquidationAttemptsPerBlock` number of attempts.

Simple remainder calculation of current subaccounts provided by the liquidation daemon and

`MaxLiquidationAttemptsPerBlock` could be enough to be calculated as a `index` for first subaccount to be processed.

Following discussions with the dYdX team regarding this discovery, it was acknowledged that the utilization of the `math.rand` package in selecting subaccounts for liquidation indeed introduces an additional attack vector. However, it was also recognized that a malicious actor could choose favorable subaccounts by directly modifying the code, making the exploitation of this vector unnecessary.

The protocol operates under the assumption of validator honesty. However, given the current implementation of liquidation logic, it is an imperative for software deployers to establish monitoring mechanisms to track liquidatable subaccounts reported by the validators. Alternatively, exploring other designs for liquidations could enforce consensus on the selection of subaccounts for liquidation. Consequently, this additional risk was included in the finding and the exploitability was assessed as Low, thereby contributing to an overall severity rating of Medium.

## Deferred Liquidation for the Most Underwater Subaccount

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	PROTOCOL IMPLEMENTATION
Severity	MEDIUM
Impact	MEDIUM
Exploitability	MEDIUM
Status	ACKNOWLEDGED
Issue	

### Involved artifacts

- </x/clob/keeper/liquidations.go>

### Description

To restrict the processing of subaccounts identified as liquidation candidates, a 'filtering logic' has been implemented to ensure that attempts to liquidate are capped at the defined

`MaxLiquidationAttemptsPerBlock` number.

Here, we observe that the algorithm calculates the starting index for subaccounts in a 'pseudo-random' manner, as shown:

```
// Get the liquidation order for each subaccount.
// Process at-most `MaxLiquidationAttemptsPerBlock` subaccounts, starting from a
pseudorandom location
// in the slice. Note `numSubaccounts` is guaranteed to be non-zero at this
point, so `Intn` shouldn't panic.
pseudoRand := k.GetPseudoRand(ctx)
liquidationOrders := make([]types.LiquidationOrder, 0)
numLiqOrders := lib.Min(numSubaccounts, int(k.Flags.MaxLiquidationAttemptsPerBloc
k))
indexOffset := pseudoRand.Intn(numSubaccounts)

startGetLiquidationOrders := time.Now()
for i := 0; i < numLiqOrders; i++ {
    index := (i + indexOffset) % numSubaccounts
    subaccountId := subaccountIds[index]
    liquidationOrder, err := k.MaybeGetLiquidationOrder(ctx, subaccountId)
    if err != nil {
        // Subaccount might not always be liquidatable since liquidation daemon
runs
```



```

        // in a separate goroutine and is not always in sync with the
        application.
        // Therefore, if subaccount is not liquidatable, continue.
        if errors.Is(err, types.ErrSubaccountNotLiquidatable) {
            telemetry.IncrCounter(1, metrics.MaybeGetLiquidationOrder,
metrics.SubaccountsNotLiquidatable, metrics.Count)
            continue
        }

        // Return unexpected errors.
        return err
    }

    liquidationOrders = append(liquidationOrders, *liquidationOrder)
}

```

With this algorithm:

- The number of processing subaccounts is constrained by `numLiqOrders` (minimum of `numSubaccounts` and `MaxLiquidationAttemptsPerBlock`).
- If an error is returned for `MaybeGetLiquidationOrder`, indicating that a subaccount is not liquidatable, it is counted as a liquidation attempt. As a result, we continue looping, and due to the failed creation of a `liquidationOrder`, some other liquidatable accounts might be overlooked.
- Even if all `numLiqOrders` attempts in creating the liquidation orders succeed, it remains uncertain whether the most underwater subaccount will be processed or filtered out.
- The next step involves [sorting the liquidation orders](#) so that the most underwater accounts can be prioritized for liquidation

## Problem Scenarios

With current algorithm processing the `subaccountIds` for liquidation, there is no guarantee that the most problematic subaccount will be liquidated.

The timely liquidation of underwater accounts is essential for maintaining the stability, security, and fairness of decentralized exchanges and other DeFi platforms. It helps prevent the accumulation of unsustainable debt, protects other participants, and contributes to the overall integrity of the decentralized financial system.

## Recommendation

We suggest changing the algorithm:

- go through all the `subaccountIds` and attempt to create liquidation orders
- sort liquidation orders
- perform bounded further processing, depending on the most underwater subaccount (not on the subaccount order and pseudo-rand determined index).

Following discussions with the dYdX team regarding this finding, it was shared that this is a known issue. The suggested approach, aimed at ensuring the liquidation of the most underwater subaccounts, has led to so-called "starvation issues" (as experienced during one of the testnet outages, as shared by the dYdX team).

A "starvation issue" occurs when a subaccount is marked for liquidation, but there is not enough liquidity on the book to close its positions. Despite this, the subaccount still counts toward the per-block limit and would prevent

other liquidations from occurring. For instance, if there were 50 such subaccounts, no other liquidations would occur (resulting in starvation).

To address this issue, the dYdX team introduced pseudo-randomness. This change ensures that while we prioritize processing the riskiest accounts first, we also guarantee that at least some liquidations would proceed. Even if the most underwater accounts are selected within one block, a different subset of subaccounts would be chosen in the next block. Thus, there shouldn't be any starvation issue, as liquidations would progress.

Due to this reasoning, the impact and exploitability of this issue were assessed as Medium, thereby contributing to an overall severity rating of Medium.

The dYdX team has acknowledged this issue and shared the possibility of deployers implementing more rigorous monitoring. They also indicated their ongoing efforts to explore further optimizations for this solution.

## Potential Chain Halt due to Panic in ABCI Phases

Project	dYdX 2023 Q4: dYdX Modules and Conditional Orders
Type	IMPLEMENTATION
Severity	CRITICAL
Impact	HIGH
Exploitability	HIGH
Status	RESOLVED
Issue	<a href="https://github.com/dydxprotocol/v4-chain/pull/1342">https://github.com/dydxprotocol/v4-chain/pull/1342</a>

### Involved artifacts

- [x/clob/keeper](#)
- [x/clob/memclob](#)

### Description

In ABCI phases such as BeginBlocker, EndBlocker, and Commit, the occurrence of panic due to unexpected states in certain collections can cause the chain to stop. Although this is usually a good approach, in certain cases, such as [untriggered\\_conditional\\_orders.go#L133-L140](#), [memclob.go#L1190-L1199](#) and [memclob.go#L1213-L1222](#), panic may not be a good solution.

### Problem Scenarios

In the current implementation, certain unexpected states trigger a panic response. However, in the context of certain processes, where it is certain that the orders are deleted and have no further impact on the chain execution, halting the chain due to panic could cause more damage than handling the error gracefully.

For instance, scenarios where the error is isolated, and the execution can continue without those specific deleted orders, could be handled more appropriately by logging the error and continuing the execution rather than causing a chain halt.

### Recommendation

1. **Error Logging:** Instead of triggering a panic in cases like [untriggered\\_conditional\\_orders.go#L133-L140](#), [memclob.go#L1190-L1199](#) and [memclob.go#L1213-L1222](#), implement an error logging mechanism. Log the error that indicates the unexpected state and continue the chain execution.
2. **Monitoring and Handling:** dYdX deployers should develop a monitoring system that can detect and handle these specific errors during runtime, providing a mechanism to address these errors without halting the entire chain.
3. **Testing and Validation:** Ensure that the new error handling mechanism is thoroughly tested to guarantee it functions as expected under various scenarios without affecting the overall software's stability.

By adopting a more controlled and responsive approach to handling errors in specific scenarios, the system can prevent unnecessary chain halts and maintain smoother operation even in the presence of unexpected states or issues in certain collections.

After discussions with the dYdX team regarding this finding, it was determined that the scenario leading to the manifestation of this issue would require an order cancellation to be placed within the same block it expires. This situation could arise from:

1. A malicious validator within the system inserting a cancellation transaction in the "expiration" block, leading to a chain halt.
2. Regular usage where an external user manages to place a cancellation order within the same block as the order expires, either intentionally or accidentally. Although the likelihood may be low, it remains a possibility.

Based on this assessment, the impact and exploitability of this issue were deemed to be High, contributing to an overall severity rating of Critical.

The solution remains straightforward, as previously suggested.





## 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:

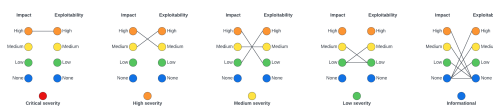
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

- *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
<span style="color: orange;">●</span> <b>High</b>	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
<span style="color: gold;">●</span> <b>Medium</b>	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
<span style="color: green;">●</span> <b>Low</b>	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
<span style="color: blue;">●</span> <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
<span style="color: red;">●</span> <b>Critical</b>	Halting of chain via a submission of a specially crafted transaction
<span style="color: orange;">●</span> <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
<span style="color: gold;">●</span> <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

Severity Score	Examples
 <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 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.