



Security Audit Report

dYdX v4

Authors: Mirel Dalcekovic, Andrija Mitrovic, Aleksandar Ljahovic, Ivan Gavran

Last revised 15 September, 2023

Table of Contents

dYdX Audit Overview	1
The Project	1
Scope of this report	1
Audit organization	1
Conducted work	2
Conclusions	3
Further Increasing Confidence	3
Disclaimer	3
dYdX Audit Dashboard	5
Severity Summary	5
Resolution Status Summary	6
Findings	7
Sentinel errors are not registered as recommended by the Cosmos SDK documentation	10
Assets should be created only for valid atomic resolution and denom exponent	12
Validation of the denom string for Assets should be introduced	14
Usage of unsafe arithmetic in x/epochs module begin blocker	16
Various Minor Code Improvements - Audit Phase I	18
x/sending module documentation is significantly out of date	21
The funding mechanism requires clearer and v4 aligned description	23
Liquidation logic should be integrated within the app	25
Prices Documentation issues	26
Exchange internal issues or changes impacts on possibility of calculating the price updates	27
Minor code changes in special messages prevention from unintended usage	29
Possible improvements in functions used for calculations in lib package	30
CheckTx allows addition of message with no handler to the mempool	32
Custom runCheckTxConcurrently() should contain existing handler msgs check	34
Analysis of CometBFT fork changes and CometBFT configuration	36
Performance optimizations in x/clob	38
Minor code changes in x/clob module	40
Validation of certain fields does not exist	42
Minor code changes in Place and Cancel Order Msg flows	43

Deleveraging algorithm needs improvement	45
Unneeded notifyTxAvailable call for v1 mempool	46
Gossiping Short Term order txs not possible due to deletion from the mempool	48
Results of the order matching should be atomically persisted to memclob	50
Optimizing Efficiency of StatefulOrdersTimeSlice KVStore	52
Optimizing Memory Efficiency for OrderId Structure in KVStores	54
Division by Zero Issue in GetBankruptcyPriceInQuoteQuantums Function	56
Missing Single Clob Message Check in RateLimitDecorator AnteHandle	58
Minor code changes in Rate Limits	60
Appendix: Vulnerability Classification	61
Impact Score	61
Exploitability Score	61
Severity Score	62

dYdX Audit Overview

The Project

In May 2023, dYdX has engaged Informal Systems to conduct a security audit of the new v4 open-source software that dYdX has been developing.

dYdX v4 is built on top of the CometBFT (formerly known as Tendermint) and Cosmos SDK as a standalone chain (the “dYdX chain”), which gives it a unique architecture and ecosystem compared to previous versions of the protocol.

Although several custom modules were stable at the moment this audit has started, the engagement with Informal Systems was agreed upon while the dYdX team was still heavily focused on development of certain parts of the system.

This security audit is essential to ensure that the open-source software is secure and can function as intended if users deploy such software, as it prepares for its release in the Cosmos ecosystem.

Scope of this report

Considering the extensive scope of the project, ongoing development and the necessity for further analysis of certain parts of dYdX v4 by the Informal Systems Auditing Team, multiple comprehensive reports will be issued.

Each report will entail a thorough analysis and present the outcomes and tasks conducted by Informal Systems' personnel during the security audit related to specific audited areas that have been deemed complete within the established timeline.

The final version of the report will encompass results from all phases of the audit.

Moreover, Informal Systems will furnish two versions of the audit reports:

1. A more detailed version for internal use by dYdX - which will include specific information about the performed tasks, sync meetings, progress, and agreements.
2. A public version of the audit report - intended for a broader audience, containing the identified findings and relevant information.

Audit organization

Phase I

Organized between May 3rd and June 23rd, 2023 and performed by:

- Andrija Mitrovic
- Mirel Dalcekovic
- Ivan Gavran

The agreed-upon plan for the first phase consisted of the following tasks:

- audit custom build dYdX modules: `x/assets`, `x/perpetuals`, `x/epochs`, `x/sending`, `x/prices`, `x/subaccounts`
- audit of the off chain components - daemons: `liquidation daemon` and `price-feed daemon`
- audit of changes made to CometBFT fork - over commit `dd32f1d`, which contained changes introduced to mempool, some metrics for TTL option and introduced locks over critical sections of execution, due to usage of `async client`.

- audit of custom `AnteHandlers` - as a part of dYdX's built mechanism to prevent special messages to be used in a malicious or unintended way, custom Ante Handlers were also analyzed during the audit for the updated commit hash `44ab121`. Special messages were introduced for `x/prices`, `x/perpetuals` and `x/clob` modules.
- audit of custom Cosmos SDK fork - over commit `f6e7e7a`, which contains changes made in order to optimize `CheckTx`.

II phase: July 24h and August 22nd, 2023

- Mirel Dalcekovic
- Aleksandar Ljahovic

The agreed-upon plan for the second phase consisted of the following tasks:

- audit of the `x/clob` module over commit hash `bc6b73e`.

After the kick off and the walkthrough meetings, it was concluded by Informal Systems' audit team and confirmed with the dYdX team that certain features were partially implemented in the commit hash analyzed and were therefore not auditable:

- **Conditional orders (type of the State-full orders)** were still under development. At this point in the time, dYdX team plans incorporating conditional orders once the dYdX goes live on the Mainnet in October 2023.
- **ReduceOnly type of orders are disabled** - the user can not place this type of order.
- **Replacement of stateful orders is not implemented**, whereas replacement logic exists for short term orders. Stateful order replacements are not going to be implemented before mainnet.
- **Liquidations and deleveraging were still under development**. Happy paths were covered in the version of the code being audited, but the sad paths were not.

During this audit, Informal Systems will report the findings according to their knowledge of the future planned development and already known issues.

It was agreed on the kickoff meeting that the priority flows needed to be audited are:

- Messages Flow - looking at it per ABCI message type. `CheckTx`, `DeliverTx`, `PrepareCheckState` are largest and most important flows for `x/clob`.
- Flows of each of our user messages (`MsgPlaceOrder`, `MsgCancelOrder`) and the messages that are generated by the block proposer `MsgProposedOperations`.
- Spam mitigation.

Conducted work

During the kick-off meeting, representatives from Informal Systems and dYdX agreed on the audit process and identified the need for several onboarding sessions to ensure a smooth and efficient transition for the auditing team to the project. It was also agreed to skip a detailed analysis of the documentation provided due to the significant changes made that were not adequately documented at the time of the audit started. Informal Systems will report any significant documentation issues only if they significantly impact understanding of the system.

The team began by reviewing the shared documentation (Notion pages shared with Informal Systems) and examining the code base to gain a preliminary understanding of the system before conducting code walkthrough sessions with the dYdX development team. The team observed that the code base contained many functions that were not currently utilized in any active system flows and could be either deprecated or used in a different manner. They prepared questions for the upcoming meetings with the dYdX team to gain a high-level overview and walkthrough of the code base. On the first sync meeting it was agreed with the dYdX representatives to first do the code inspection of the code that is actually living, and then just to go over the unused code in order to find some explicit issues. dYdX team has provided explanations, mostly about the current state of the code, planned development, usage of currently unused functions and economic terms on the following walk through meetings.

After establishing a clear understanding of the terminology used and the existing implementation, the auditing team performed a manual code review with a primary focus on code correctness and a critical points analysis of the agreed-upon scope, by phases.

Conclusions

Phase I

In general, we found the codebase to be of high quality. The documentation should be updated or provide additional information for external readers about the future development documented. The code is well-structured and easy to follow and is covered with tests, which could be richer with additional edge test cases.

As a confirmation of high quality code, we did not find any critical or high severity issues in the codebase. Most of the issues we encountered were related to aesthetics and suggestions for improving the current design. Overall, we are impressed with the quality of the codebase.

dYdX Cosmos SDK and CometBFT forks were also analyzed and no critical issues were detected:

- There was an issue detected in vanilla version of Cosmos SDK, communicated to the Cosmos SDK team and fixed during this audit. The solution was suggested to the dYdX team as well.
- CometBFT fork analysis was done, and there was one finding surfaced. The finding was directed towards usage of the v1 version of mempool which will be removed from v 0.38.

Phase II

Overall, based on the present iteration of the `x/clob` module's code, we can affirm its high level of coding quality. Our examination for potential threats did not uncover any critical concerns, with all attack vectors either proven unfeasible or addressed through the implemented measures. However, a number of issues with low to medium severity were identified.

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. This is especially relevant to the analysis of locks in the CometBFT fork.

During the second audit phase, we thoroughly examined the extensive codebase of `x/clob`. We have confirmed with the dYdX team that a particularly complex module was still being developed, with changes being made simultaneously in the main repository. As a result, it became clear that the version of `x/clob` we audited would not be the one deployed on the mainnet.

In light of this, the audit team suggests revisiting the `x/clob` module once more to ensure it aligns with the final code version intended for release.

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 bug free 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.

dYdX Audit Dashboard

Target Summary

- **Type:** Specification and Implementation
- **Platform:** Go
- **Artifacts Phase I:**
 - v4 repository - performed at commit hash [a2b7802](#) and a new commit hash [44ab121](#) for prices and ante handlers.
 - [liquidation and price-feed daemons](#)
 - [x/assets](#)
 - [x/perpetuals](#)
 - [x/epochs](#)
 - [x/prices](#)
 - [x/subaccounts](#)
 - [x/sending](#)
 - CometBFT fork - performed at commit hash [dd32f1d](#).
 - Cosmos SDK fork - performed at commit hash [f6e7e7a](#).

Duration: 12 person weeks

- **Artifacts Phase II:**
 - [x/clob](#) module performed at the agreed hash for the phase: [bc6b73e](#). It was concluded that the impact of the listed artifacts would be once again analyzed, now in the context of the [x/clob](#) module:
 - CometBFT fork on commit hash [dd32f1d](#). - due to filtering depending on the type of the order (short vs state-full) and manipulation with `tx` s containing `MsgPlaceOrder` and `MsgCancelOrder` ;
 - Cosmos SDK fork on commit hash [f6e7e7a](#) - due to Commit phase triggering logic executed in the [x/clob](#) module;
 - [ante handlers](#): due to newly introduced RateLimitDecorator and ClobDecorator, both implemented on the [x/clob](#) .

Duration: 6 person weeks

Engagement Summary

Phase I

- **Dates:** 06.05.2023. to 23.06.2023.
- **Method:** Manual code review & protocol analysis
- **Employees Engaged:** 3

Phase II

- **Dates:** 24.07.2023. to 22.08.2023.
- **Method:** Manual code review & protocol analysis
- **Employees Engaged:** 2

Severity Summary

Finding Severity	#
Critical	0

Finding Severity	#
High	0
Medium	1
Low	12
Informational	15
Total	28

Resolution Status Summary

Resolution Status	#
Resolved	0
Risk-accepted	25
Functioning as Designed	3
Total	28

Findings

Title	Type	Severity	Issue
Usage of ABCI++ in dydx: risk scenarios	PROTOCOL	2 MEDIUM	
Gossiping Short Term order txs not possible due to deletion from the mempool	PROTOCOL DOCUMENTATION	2 MEDIUM	
CheckTx allows addition of message with no handler to the mempool	IMPLEMENTATION	1 LOW	https://github.com/dydxprotocol/cosmos-sdk/issues/20
Custom runCheckTxConcurrently() should contain existing handler msgs check	IMPLEMENTATION	1 LOW	https://github.com/dydxprotocol/cosmos-sdk/issues/20
Analysis of CometBFT fork changes and CometBFT configuration	IMPLEMENTATION	1 LOW	
Performance optimizations in x/clob	IMPLEMENTATION	1 LOW	
Validation of certain fields does not exist	IMPLEMENTATION	1 LOW	
Deleveraging algorithm needs improvement	IMPLEMENTATION	1 LOW	
Unneeded notifyTxsAvailable call for v1 mempool	IMPLEMENTATION	1 LOW	
Results of the order matching should be atomically persisted to memclob	IMPLEMENTATION	1 LOW	
Optimizing Efficiency of StatefulOrdersTimeSlice KVStore	IMPLEMENTATION	1 LOW	
Optimizing Memory Efficiency for OrderId Structure in KVStores	IMPLEMENTATION	1 LOW	
Missing Single Clob Message Check in RateLimitDecorator AnteHandle	IMPLEMENTATION	1 LOW	

Title	Type	Severity	Issue
Sentinel errors are not registered as recommended by the Cosmos SDK documentation	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1278
Assets should be created only for valid atomic resolution and denom exponent	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1279
Validation of the denom string for Assets should be introduced	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1280
Usage of unsafe arithmetic in x/epochs module begin blocker	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1281
Various Minor Code Improvements - Audit Phase I	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1316
x/sending module documentation is significantly out of date	DOCUMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1295
The funding mechanism requires clearer and v4 aligned description	DOCUMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1296
Liquidation logic should be integrated within the app	PROTOCOL	0 INFORMATIONAL	
Prices Documentation issues	DOCUMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1321
Exchange internal issues or changes impacts on possibility of calculating the price updates	PROTOCOL	0 INFORMATIONAL	
Minor code changes in special messages prevention from unintended usage	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1451

Title	Type	Severity	Issue
Possible improvements in functions used for calculations in lib package	IMPLEMENTATION	0 INFORMATIONAL	https://github.com/dydxprotocol/v4/issues/1452
Minor code changes in Place and Cancel Order Msg flows	IMPLEMENTATION	0 INFORMATIONAL	
Division by Zero Issue in GetBankruptcyPriceInQuoteQuantums Function	IMPLEMENTATION	0 INFORMATIONAL	
Minor code changes in x/clob module	IMPLEMENTATION	0 INFORMATIONAL	
Minor code changes in Rate Limits	IMPLEMENTATION	0 INFORMATIONAL	

Sentinel errors are not registered as recommended by the Cosmos SDK documentation

Title	Sentinel errors are not registered as recommended by the Cosmos SDK documentation
Project	dYdX v4
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1278

Involved artifacts

- [x/perpetuals/types/errors.go](#)
- [x/prices/types/errors.go](#)
- [x/sending/types/errors.go](#)
- [x/assets/types/errors.go](#)
- [x/subaccounts/types/errors.go](#)
- [app/process/errors.go](#)

Description

During the analysis of the possibility of potential threats realization, it was noticed that sentinel errors registered for the custom modules were not following general suggestions from the Cosmos SDK documentation provided on the link.

The following recommendations were not followed:

1. App developers should import new [cosmossdk.io/errors](#) package.
In the current implementation, dYdX custom modules are using deprecated package [errors](#).

```
var sdkerrors.Register func(codespace string, code uint32, description string) *errors.Error
```


Example of the import and usage could be seen [here](#).
2. Error codes must be greater than one, as a code value of one is reserved for internal errors.
This is explained in the Cosmos SDK Building modules documentation, on the [link](#).
We noticed that all the modules are using error codes < 2 for registering sentinel errors. Also, there are some modules that have error codes = 0. Zero is reserved for a successful execution.
 - Process Proposal: `ErrDecodingTxBytes` - [code](#)
 - Assets: `ErrAssetDoesNotExist` - [code](#)
 - Prices: `ErrInvalidInput` - [code](#)

- Sending: `ErrSenderSameAsRecipient` - code
- Perpetual: `ErrPerpetualDoesNotExist` - code
- Subaccounts uses error codes = 0 for `ErrIntegerOverflow` - code, which is used in `x/clob` module, not in subaccounts.

Problem Scenarios

There is a possibility of potential clashing with existing system-level errors, since error code 1 is being reserved for errors utilized by the SDK itself.

Recommendation

To ensure clarity and avoid potential clashing with existing system-level errors, it is advised to adhere to the convention of using custom error codes starting from 2 or higher.

Remediation Plan

Risk accepted: The dYdX team accepted the risk of this issue. They plan on resolving it in 2024.

Assets should be created only for valid atomic resolution and denom exponent

Title	Assets should be created only for valid atomic resolution and denom exponent
Project	dYdX v4
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1279

Involved artifacts

- [x/assets/types/genesis.go](#)
- [x/assets/keeper/asset.go](#)

Description

`CreateAsset` function is used to create assets for the dYdX chain. Currently it is called only on the initialization of the chain and only for `USDC` asset, which will be the only existing asset in the dYdX chain (at least upon the release).

There are two asset's parameters that should be either validated upon the creation of the Asset in `CreateAsset` or the validation should be removed in another `ConvertAssetToCoin` function.

Those asset's parameters are:

- `atomic_resolution`, type: `int32` - the exponent for converting 1 quantum to a full coin.
- `denom_exponent`, type: `int32` - the exponent for converting one unit of `denom` to a full coin.

Currently, absolute values of these parameters are expected to be smaller than `MaxAssetUnitExponentAbs`, defined in asset's module `constants` because `ConvertAssetToCoin` will not work due to `checks` if this values are not valid.

```
MaxAssetUnitExponentAbs = 32
```

Problem Scenarios

Currently this is not high risk, since there will be only one asset (USDC) defined for the dYdX chain in genesis file, where the values are `hardcoded`, so we expect these values to be within expected value span, but the validations of this fields should be present even in this case.

This could become an issue if another ways of creating assets are introduced latter during the development. Currently transfer functions from `x/subaccounts` module are using the `ConvertAssetToCoin` function and will be aborted due to invalid values.

Recommendation

If it is expected for these exponent values not to be larger than number 32. Both can be represented with another type (e.g. int8 with range: -128 through 127.)

There should be validation in `CreateAssets` function added or the existing one removed from the `ConvertAssetToCoin`. Also, expected validation and the reasons for the maximum number should be explained in the documentation. We could not find anything in the shared Notion pages.

If validation is not necessary, for any reason (needed to be explained why this validation exists and how it is determined in the first place) it should be removed from `ConvertAssetToCoin`.

Also, `Validate` function, called from the `ValidateGenesis` should also validate this fields. Currently, this is not the case.

Remediation Plan

Risk accepted: The dYdX team accepted the risk of this issue. They plan on resolving it in 2024.

Validation of the denom string for Assets should be introduced

Title	Validation of the denom string for Assets should be introduced
Project	dYdX v4
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	1 LOW
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1280

Involved artifacts

- <x/assets/keeper/asset.go>

Description

There is no validation for Assets denom field, which will contain string value representing

“The name of base denomination unit of the `Asset` (e.g. `udydx`, `uatom`, `ibc/xxxxx`). Must be unique and match the `denom` used in the `sdk.Coin` type in the `x/bank` module.” - [source](#)

We should have the following validations:

Since it is necessary to perform conversion from the dYdX exchange presentation of `Asset.Denom` string to bank module's `Coin.Denom` string, in order to make transfers and place deposits in subaccounts, It seems that validation of the Asset's denom field should be the same as one existing for Coin.Denom defined with regular expression [here](#):

```
// 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}`
```

Also, if dYdX will be working only with IBC denoms, there should be some basic validation introduced because all denominations arriving over IBC, should have a specific structure, with the prefix `ibc/` followed by a bounded HEX string.

Problem Scenarios

There should be no “invalid” assets - ones that can not be converted to Coin.Denom in the system, because those assets will be unusable.

Currently, there will be only one asset in the system, defined upon initialization of the system from genesis file ([USDC](#)) but nevertheless, validations for expected format should be existing.

Recommendation

Introduce validations.

Remediation Plan

Risk accepted: The dYdX team accepted the risk of this issue. They plan on resolving it in 2024.

Usage of unsafe arithmetic in x/epochs module begin blocker

Title	Usage of unsafe arithmetic in x/epochs module begin blocker
Project	dYdX v4
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	3 HIGH
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1281

Involved artifacts

- [x/epochs/keeper/epoch_info.go](#)

Description

In the Begin Blocker code for Epochs module, there are several places where explicit conversion of int64 values into uint32 values is performed. This code contains logic for checking if new epoch should be triggered and this is depending on:

- several parameters defined for the epoch in epoch info: next_tick, duration
- and from the current block timestamp

Places in code:

- When starting the next epoch `MaybeStartNextEpoch` function with:

```
// Starts next epoch.
currentTick := epoch.NextTick

epoch.NextTick = epoch.NextTick + epoch.Duration
epoch.CurrentEpoch++
epoch.CurrentEpochStartBlock =
lib.MustConvertIntegerToUint32(ctx.BlockHeight())
k.setEpochInfo(ctx, epoch)
```

- when calculating the `nextTick`: two `uint32` values are added up, and could/will at a certain point in time, overflow and we will “lose” expected duration of an epoch, because next tick will be smaller than it should have been, due to the value being out of range.

Example: `next_tick = 1000`; `duration = maxUint32`;
on initialization of the epoch, we will have: `block_time = 1001`;
`duration_Multiplier = (1001-1000)/maxUint32 + 1 = 1`

```

next_tick = 1000 + maxUInt32*1 = 999, after initialization:
next_tick = 999+maxUInt32 = 998 ...
next_tick = 1 + maxUInt32 = 0
next_tick = 0 + maxUInt32 = 4294967295

```

- when determining the `CurrentEpochStartBlock` we are defining the next epoch with the number of the current block height, which is `int64`. This number is explicitly converted to `uint32` with `MustConvertIntegerToUInt32` function, which will `panic` in case when block height is larger than maximum `uint32`.
- There are several places where similar casts are used, in other epoch module code places:
 - `NumBlocksSinceEpochStart` function - [here](#), used from perpetuals code, from End Blocker

Problem Scenarios

All the cases listed above are pretty far away or not possible due to current implementation where epochs will be defined with the genesis file, but these code lines represent bad code practice.

Recommendation

Probably the `CurrentEpochStartBlock` should be of type `uint64` or this code should be returning some error containing information that the maximum height was reached - if this is the expectation at a certain point in the future.

Also, it could be reasonable to determine some maximal valid value for the `duration` of one epoch.

Remediation Plan

Risk accepted: The dYdX team accepted the risk of this issue. They plan on resolving it in 2024.

Various Minor Code Improvements - Audit Phase I

Title	Various Minor Code Improvements - Audit Phase I
Project	dYdX v4
Type	IMPLEMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1316

Involved artifacts

- [x/subaccounts/types/subaccount.go](#)
- [x/subaccounts/keeper/transfer.go](#)
- [x/subaccounts/keeper/subaccount_helper.go](#)
- [x/subaccounts/keeper/transfer.go](#)
- [x/epochs/keeper/epoch_info.go](#)
- [daemons/pricefeed/client/sub_task_runner.go](#)

Description

Here is the list of aesthetic and minor code improvements and issues around logging found during the code inspection of the dYdX audit scope in the phase I. They do not pose a security threat nor do they introduce an issue, but the following suggestions are shared to improve the code readability, keep consistency, optimize, and improve logging.

- This comment is not fulfilled in the function. `SetUsdcAssetPosition` does not return an error in any case. → Remove the comment or add the check and return the error.
- The function `applyUpdatesToPositions` could be changed to immediately return the same positions if the updates are empty, to optimize the execution.
- Should `else` statement be used [here](#) because `pp` is a reference to something that might be deleted in the if statement?
- Try to keep consistency in the `transfer.go` functions:
 - For the `TransferFundsFromSubaccountToModule` if `applyValidSubaccountUpdateForTransfer` fails, there will be `panic` explicitly thrown and error will be logged.
 - In other functions from this file, like for e.g.: `TransferFundsFromModuleToSubaccount`, at the end of the function, where `applyValidSubaccountUpdateForTransfer` is executed, there is no panicking and no logging of error - [here](#). In that case coins were transferred from the sending module to the subaccount module but subaccount did not get updated. This is handled differently in the previous function where the flow of coins was opposite.

- This [part](#) could execute after this [check](#), to skip validation if epoch already exists.
- This [map](#) is not well defined, even though at the keys are of uint32 type - it is confusing to leave it like this:

```
marketPriceUpdateMap :=
```

```
make(map[types.ExchangeFeedId]*api.MarketPriceUpdate)
```

The key for this map is `types.ExchangeFeedId`, while further down the code an `MarketId` is used for a key:

```
marketPriceUpdate, exists :=
marketPriceUpdateMap[marketPriceTimestamp.MarketId]
// Add key with empty `api.MarketPriceUpdate` if entry does not exist.
if !exists {
    marketPriceUpdate = &api.MarketPriceUpdate{
        MarketId:      marketPriceTimestamp.MarketId,
        ExchangePrices: []*api.ExchangePrice{},
    }
    marketPriceUpdateMap[marketPriceTimestamp.MarketId] = marketPriceUpdate
```

Both are of type uint32 so it won't make any problems but it is confusing.

- These [checks statements](#) should be immediately after the line where [index price is acquired](#) from the map.
- This [part of the code](#) should be under the else statement of the previous `if`. Interpolation between two same values is meaningless.
- The function `GetSubaccount` should be renamed/refactored to something like `GetOrInitSubaccount` and should add the key/value to the store right away because by current implementation if subaccount does not exist `GetSubaccount` returns the key/value as it exists: [code](#). This can be confusing.
- This [comment](#) is wrong: instead of saying `if 0 <= cPpm <= 1_000_000`, it should be `if not 0 <= cPpm <= 1_000_000`
- `GetRemoveSampleTailsFunc`: with division [here](#), `topRemoval` and `bottomRemoval` could be of different values, should the `topRemoval:end` slice be returned [here](#).

```
premiums := []int32{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13}
prem := premiums[5:10]
fmt.Println(prem) → {5, 6, 7, 8, 9}
e.g.
```

```
totalRemoval = 9,
bottomRemoval = 4, topRemoval=5
END = length - topRemoval = 14 - 5 = 9
Return premiums[bottomRemoval:end]
prem := premiums [4:9] → {4, 5, 6, 7, 8} )
```

The algorithm will remove 4 elements from the start and 5 from the bottom, like `topRemoval` is 4, `bottomRemoval` is 5? This is small, but maybe it should be changes to be aligned with the expectations:

```
end = length - bottomRemoval
return premiums[topRemoval:end]
```

- `lenPadding` doesn't have to be int64 [here](#). Or some calculations could go wrong? In that case, int32 is enough, but error must be returned if negative value is calculated?
- Logging of errors [here](#) in `DepositFundsFromAccountToSubaccount` and [here](#) in `WithdrawFundsFromSubaccountToAccount` could be improved to provide the information about the exact invalid negative quantum amount, like [here](#) in `TransferFeesToFeeCollectorModule`.
- `x/sending` module implementation has some unused code left behind after removing pruning of transfers from `endblocker` logic, `goodtilblock` field on transfers, and execution of transfers in `endblocker` :
 - key prefixes defined [here](#) are not needed anymore since nothing is stored in the state
 - Errors defined [here](#) probably remained from removed features. Also: `ErrMissingFields` is not used.

Given that the auditing team has been notified about the absence of any further planned enhancements for the `x/sending` module, except for the possibility of enabling multiple assets within the chain, our suggestion is to refine the code and update the documentation accordingly. Additionally, there is a separate finding concerning inconsistencies found in the technical documentation provided to us.

- It would be more intuitive to not use the “index price” variable name when calculating the premiums [here](#) since it is calculated taking the underlying assets price into account, but the “consensus” price - not the index price received from the daemons in a validators node. We can see this [here](#) in the perpetuals module code when reaching out to the `x/clob` module for the exact value of the premium.
- Perpetual’s keys contain defined constant for `FundingSamplesKeyPrefix` [here](#) - which is not used, and has probably remained from the previous implementation of the funding mechanism. Together with the function name `MaybeProcessNewFundingSampleEpoch` it lead us to believe that this store is also being used to store processed premium votes (but actually premium samples are stored). These should be removed.
- These `AssetsKeeper`, `BankKeeper` expected keepers in subaccounts are currently not used. Should they be removed.

Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements.

Recommendation

As explained in the Description section.

Remediation Plan

Risk accepted: The dYdX team accepted the risk of issues listed in this finding. They plan on resolving most of them in 2024.

x/sending module documentation is significantly out of date

Title	x/sending module documentation is significantly out of date
Project	dYdX v4
Type	DOCUMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1295

Involved artifacts

- Notion [page](#) containing x/sending module's technical design

Description

Existing protocol spec for the x/sending module is significantly out of the date. Notion [page](#) containing the explanation of the business logic shared with the auditing team was misleading and once the team dived into the code, we realized that module is designed and implemented differently.

Here is a list of things we noticed and that should be removed from the documentation:

- All the sections containing the explanation of the `endblocker` 's pruning of expired transfers
- Mentioning `goodtilblock` parameter and validations for this value should be removed from the documentation, also `TransgerBlockExpiration` and `ShortBlockWindow`
- Message `MsgSubaccountTransfer` is renamed to `MsgCreateTransfer` . There are some notes about future refactoring, that should probably be removed, since it seems this is done, but in a different way than described in the document.
- The documentation says that transfers are done in the `endblocker` , which indicates that all transfers created during the deliver tx will be stored somehow in the state and being executed in the `endblocker` - which is not the case. Upon the creation of the transfer, sending is being executed within the deliver tx.

We have found that pruning was indeed initially implemented, but at a certain point removed, and that the transfer changed its [structure](#) and fields. Also, comments that exist on that page regarding this feature were misleading at the moment of onboarding to the project and writing the system overview and threat modeling.

Recommendation

Considering that there are no significant development plans for the `x/sending` module, we recommend conducting a thorough review of the existing documentation.

Remediation Plan

Risk accepted: The dYdX team plans on updating the documentation before the mainnet release of v4 open-source software.

The funding mechanism requires clearer and v4 aligned description

Title	The funding mechanism requires clearer and v4 aligned description
Project	dYdX v4
Type	DOCUMENTATION
Severity	0 INFORMATIONAL
Impact	0 NONE
Exploitability	0 NONE
Issue	https://github.com/dydxprotocol/v4/issues/1296

Involved artifacts

- Notion [v4 Funding Spec](#)
- Notion [v4 Funding Premium Sampling](#)
- Notion [x/perpetual](#)

Description

There are several details that were misleading during our audit and should be probably removed or updated in the listed documents:

- `MsgAddFundingSamples` is renamed and contains different fields:

- ```
// FundingPremium represents a funding premium value for a perpetual
// market. Can be used to represent a premium vote or a premium sample.
message FundingPremium {
 // The id of the perpetual market.
 uint32 perpetual_id = 1;
 // The sampled premium rate. In parts-per-million.
 int32 premium_ppm = 2;
}

// MsgAddPremiumVotes is a request type for the AddPremiumVotes method.
message MsgAddPremiumVotes {
 string proposer = 1 [(cosmos_proto.scalar) = "cosmos.AddressString"];
 repeated FundingPremium votes = 2 [(gogoproto.nullable) = false];
}
```

- [Note about funding](#) should be rephrased to correctly explain the prepare proposal. Interest rate was slightly confusing, since we could not find anything that is being added up by the proposer only, and is not set by

governance. As understood from information about the future development in x/perpetuals module - only ability to create new perpetuals using `x/gov` module will be added.

- `expectedNumSamples` explained [here](#) are misleading - we could not find any expectations regarding the minimum number of samples or that epochs are depending on this parameter.
- [Explanation](#) about funding premiums being retrieved from the oracle daemon and compared with `defaultFunding` on perpetual - we were not sure if premium votes are/were also received somehow from the pricing or some other daemon? Also, it is clear that there is no in memory data structure containing Funding Premiums.
- Governance messages do not exist. Will they be added for some parameters and which ones? We could not find anything about emergency parameters, but we saw that those are added recently, so it is possible that this is planned as a future development.
- Will there be interest rate in v4 - since we concluded [here](#) it is taken from v3 version's technical documentation, and we could not find it in the current implementation.
- It is unclear why the term "index price" is used when calculating the premium, since the market price is actually read from the prices store? So it is calculated regarding the underlying assets price, but the price agreed with the consensus. We can see this [here](#) in the perpetuals module code, when reaching out to the `x/clob` module for the exact value of the premium.
- One overall suggestion regarding the funding mechanism is to review the existing documentation and revise the term "funding sample" to "premium sample" for greater clarity and consistency. Alternatively, you could include a note clarifying that these terms are interchangeable and should be treated as such. This update will help ensure that the terminology used throughout the documentation aligns with the intended meaning and avoids any confusion.

## Recommendation

We recommend conducting a thorough review of the existing documentation, update it and align the implementation, if needed.

## Remediation Plan

**Risk accepted:** The dYdX team plans on updating the documentation before the mainnet release of v4 open-source software or shortly after.

## Liquidation logic should be integrated within the app

|                       |                                                       |
|-----------------------|-------------------------------------------------------|
| <b>Title</b>          | Liquidation logic should be integrated within the app |
| <b>Project</b>        | dYdX v4                                               |
| <b>Type</b>           | PROTOCOL                                              |
| <b>Severity</b>       | 0 INFORMATIONAL                                       |
| <b>Impact</b>         | 0 NONE                                                |
| <b>Exploitability</b> | 0 NONE                                                |
| <b>Issue</b>          |                                                       |

### Involved artifacts

- [v4/daemons/liquidation](#)

### Description

In the current version of dYdX v4 application the liquidation process is lead by a daemon process called liquidation daemon. Through documentation and code analysis it has been concluded that liquidation daemon is only used to establish communication between subaccounts, perpetuals and clob modules regarding filtering subaccounts ready for liquidation and filling in the internal structure with such subaccounts. Thus we found it unnecessary to be a separate process which is started as a go-routine in the main app thus increasing the chance of app failure if the daemon fails. Even though unnecessarily separating it from the modules leads to one more “moving part” that makes it more difficult for maintaining and development.

### Problem Scenarios

Unnecessary of chain part to be maintained. It is started as a go-routine in the main app, that could lead to app failure if the daemon fails.

### Recommendation

Liquidation process should be a part of the app itself. We will not recommend a concrete implementation plan, but for an example it could be a part of one of the modules used in the liquidation process.

### Remediation Plan

**Risk accepted:** The dYdX team accepts this finding and is aware of the design flaw. This might be prioritized in 2024.

## Prices Documentation issues

|                       |                                                                                                             |
|-----------------------|-------------------------------------------------------------------------------------------------------------|
| <b>Title</b>          | Prices Documentation issues                                                                                 |
| <b>Project</b>        | dYdX v4                                                                                                     |
| <b>Type</b>           | DOCUMENTATION                                                                                               |
| <b>Severity</b>       | 0 INFORMATIONAL                                                                                             |
| <b>Impact</b>         | 1 LOW                                                                                                       |
| <b>Exploitability</b> | 0 NONE                                                                                                      |
| <b>Issue</b>          | <a href="https://github.com/dydxprotocol/v4/issues/1321">https://github.com/dydxprotocol/v4/issues/1321</a> |

## Involved artifacts

- [notion/dydx/x-prices](#)
- [notion/dydx/Acceptance-Criteria-for-Price-Changes](#)

## Description

1. In the [docs on prices](#), it is said that “During `ProcessProposal` step, validators check that the followings are true. Otherwise, reject the proposed block [...]” and then two criteria are listed. However, the block is accepted as soon as **any** of the criteria is met. This is spelled out in the attached document on [Acceptance criteria for price changes](#), though, so it is more an issue of being stale than being wrong.
2. In the [Acceptance criteria for price changes](#), there is a typo in the condition: `(idx <= new <= old) || (old >= new >= idx)` should be replaced by `(idx <= new <= old) || (old <= new <= idx)`. The code is correct.

## Problem Scenarios

Documentation is misleading.

## Recommendation

Noted in description.

## Remediation Plan

**Risk accepted:** The dYdX team plans on updating the documentation shortly after the mainnet release of v4 open-source software.

## Exchange internal issues or changes impacts on possibility of calculating the price updates

|                       |                                                                                             |
|-----------------------|---------------------------------------------------------------------------------------------|
| <b>Title</b>          | Exchange internal issues or changes impacts on possibility of calculating the price updates |
| <b>Project</b>        | dYdX v4                                                                                     |
| <b>Type</b>           | PROTOCOL                                                                                    |
| <b>Severity</b>       | 0 INFORMATIONAL                                                                             |
| <b>Impact</b>         | 1 LOW                                                                                       |
| <b>Exploitability</b> | 0 NONE                                                                                      |
| <b>Issue</b>          |                                                                                             |

### Involved artifacts

- [x/prices/keeper/market.go](#)
- [daemons/server/types/pricfeed/market\\_to\\_exchange\\_prices.go](#)
- [daemons/pricfeed/client/price\\_fetcher.go](#)

### Description

Since dYdX chain is connected and depends on the data retrieved from external services and systems, that is exchanges like *Binance*, *BinanceUs*, *Bitfinex*, *Kraken* it is crucial for smooth running of dYdX chain to take into account and gracefully process the possibility of exchanges' downtime.

Exchanges could be upgraded, API could be changed and even the exchanges name could be changed. Also, there could be some internal errors, than need to be processed in a way that it does not impact the system.

All the exchanges are defined at genesis as well as the minimum number of exchanges `min_exchanges` parameter for a market. This would insure that number of live exchanges being looped for the price updates is always higher.

```
// Market defines the price configuration for a single market relative to
// quoteCurrency.
message Market {

 // Unique, sequentially-generated value.
 uint32 id = 1;
 ...
 // The list of exchanges to query to determine the price.
 repeated uint32 exchanges = 4;

 // The minimum number of exchanges that should be reporting a live price for
 // a price update to be considered valid.
```

```
uint32 min_exchanges = 5;

...

}
```

There are two places that actually check the value of number of exchanges:

- upon creating markets on genesis [here](#) and
- when calculating the median price [here](#).

In case of inability to query the data from an exchange [here](#), the place where the system would catch this is when calculating the median price. If the number of valid prices retrieved from the exchanges is lower than `min_exchanges` price update will be skipped for that market.

## Problem Scenarios

The dYdX team will include a way for updating which exchanges need to be queried for a certain market, but as this will be part of `x/gov` it will take some time to make the update.

The realistic values for `exchanges` and `min_exchanges` will differ in count by ~50%. For example the number of exchanges that is queried in v3 is typically 5-8 exchanges per market, with a minimum of 3-4 (depending on the size of the market). This means having issues with one exchange is not critical for getting the price update. At the time being for v4, there are only 3 or 4 exchanges and the minimum is 2-3 and if it stays this way until release the risk will be far greater comparing to the numbers in v3.

However, if dYdX would like to develop the open-source v4 software so that it can support more transactions for which it would be convenient to execute instantly, due to fixing all kinds of situations that could occur and introduce instability or issues in the system, we would suggest the dYdX team to consider introducing permission roles for special system maintenance transactions.

## Recommendation

There could be a chain management account in the dYdX chain that could hold the special `chain_management` role. This would mean that this account needs to be a part of the state, and that it needs to be updated if necessary (this could be implemented through the governance proposal). Also, additional ante handler should be introduced, in order to check the signers of the special, restricted messages such as the one for updating the number or list of exchanges (for example).

There are public solutions in Cosmos ecosystem that introduced `x/permission` module and are using permission roles to execute chain management transactions for a quick response in cases of unwanted, critical situations.

## Remediation Plan

**Risk accepted:** The dYdX Team accepted the risk of this issue. They are considering solutions (including the one listed here) and will resolve this in 2024.

## Minor code changes in special messages prevention from unintended usage

|                       |                                                                                                             |
|-----------------------|-------------------------------------------------------------------------------------------------------------|
| <b>Title</b>          | Minor code changes in special messages prevention from unintended usage                                     |
| <b>Project</b>        | dYdX v4                                                                                                     |
| <b>Type</b>           | IMPLEMENTATION                                                                                              |
| <b>Severity</b>       | 0 INFORMATIONAL                                                                                             |
| <b>Impact</b>         | 0 NONE                                                                                                      |
| <b>Exploitability</b> | 0 NONE                                                                                                      |
| <b>Issue</b>          | <a href="https://github.com/dydxprotocol/v4/issues/1451">https://github.com/dydxprotocol/v4/issues/1451</a> |

### Involved artifacts

- [app/messages/normal\\_msgs.go](#)
- [app/ante/msg\\_type.go](#)
- [app/ante.go](#)

### Description

- [These](#) messages shouldn't be amongst the normal messages. Since they will be [deprecated](#) - they should be within unsupported. All the other software upgrade types of messages are usually in internal messages, since they can not be sent by external user.
- [Here](#) we need to change the error message: "msgs cannot be empty" to "tx should contain messages".
- Since in case of injected message we shouldn't be checking the signers, it seems that [ValidateSigCountDecorator](#) should be wrapped with `NewAppInjectedMsgAnteWrapper`. There is no need for tx to be signed, or to contain the appropriate number of signatures. This could be one of the ante handlers skipped as well and it could even speed up the CheckTx, which is very important for dYdX.

### Recommendation

As explained above.

### Remediation Plan

**Risk accepted:** The dYdX team accepted the risk of this issue. Since this is of low priority for them, they plan on maybe resolving it in 2024.



## Possible improvements in functions used for calculations in lib package

|                       |                                                                                                             |
|-----------------------|-------------------------------------------------------------------------------------------------------------|
| <b>Title</b>          | Possible improvements in functions used for calculations in lib package                                     |
| <b>Project</b>        | dYdX v4                                                                                                     |
| <b>Type</b>           | IMPLEMENTATION                                                                                              |
| <b>Severity</b>       | 0 INFORMATIONAL                                                                                             |
| <b>Impact</b>         | 0 NONE                                                                                                      |
| <b>Exploitability</b> | 0 NONE                                                                                                      |
| <b>Issue</b>          | <a href="https://github.com/dydxprotocol/v4/issues/1452">https://github.com/dydxprotocol/v4/issues/1452</a> |

### Involved artifacts

- [v4/lib/math.go](#)

### Description

- `DivisionUint32RoundUp()` [here](#).

Function is currently used only in one place, for calculating the number of minimum samples required for premium rate and it is working with epochs duration numbers, but since this is a function from lib package which, in our understanding should present a custom built general set of functions, the functions in math.go should consider invalid or unpredictable inputs as well.

Suggestion: Error handling should be implemented in case this function is in future used in some other places as well: division by zero would lead to a runtime panic, so this case needs to be handled in a desired way: either by returning an error or a special value.

- Usage of float numbers could introduce precision loss in `ChangeRateUint64()` [here](#).

This function contains possible issues with precision loss when:

casting from `uint64` to `big.Float` [here](#) - the default precision for `big.Float` (if not explicitly specified) will be 53 bits (same as for the `float64` )

casting to `float32` [here](#), since the values could be large, the precision of the `float32` approximation may not be able to capture the small difference rate between `bigOriginalV` and `bigNewV` . The result may be rounded to the nearest representable `float32` value, leading to precision loss.

Since this function is currently used only for calculating the change rate of prices for telemetry, then the impact is not critical at all.

No changes in the implementation needs to be done, but we will share out usual suggestions when it comes to the

usage of float numbers.

For similar calculations, since we can not not predict what is the possible range of the price values in the future, we suggest using the Dec numbers instead - defining the precision wanted and handling the rounding of the numbers in the desired way.

As you can see with the go playground [example](#), even some integer numbers cannot be adequately represented in `float32` starting from some point (for numbers not in  $[-2^{24}, 2^{24}]$ ). For `float32`, we take the example of,  $16,777,216 = 2^{23} \times 2$ . Since the mantissa of a `float32` number is 23 bits long, float32 with this number we are starting to lose precision.

- The `float64` type has a wider range of integer numbers being computed without errors. But in this case casting the result back to `float32` could introduce issues, as well.
- The values that would be correctly represented with floats are limited by the aforementioned ranges ( $[-2^{24}, 2^{24}]$  for `float32`,  $[-2^{53}, 2^{53}]$  for `float64`).

Here we have one more go playground [example](#) showing how the final results are different when casting the Quo result back to the `float32`.

Prices have a high impact through the entire system, so it is important to handle them right in case the results are used for further calculations.

## Recommendation

As stated above.

## Remediation Plan

**Risk accepted:** The dYdX team accepted the risk of this issue. They plan on resolving it in 2024.

## CheckTx allows addition of message with no handler to the mempool

|                       |                                                                                                                         |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------|
| <b>Title</b>          | CheckTx allows addition of message with no handler to the mempool                                                       |
| <b>Project</b>        | dYdX v4                                                                                                                 |
| <b>Type</b>           | IMPLEMENTATION                                                                                                          |
| <b>Severity</b>       | 1 LOW                                                                                                                   |
| <b>Impact</b>         | 2 MEDIUM                                                                                                                |
| <b>Exploitability</b> | 1 LOW                                                                                                                   |
| <b>Issue</b>          | <a href="https://github.com/dydxprotocol/cosmos-sdk/issues/20">https://github.com/dydxprotocol/cosmos-sdk/issues/20</a> |

### Involved artifacts

- Cosmos SDK:
  - [cosmos-sdk/baseapp/baseapp.go](https://github.com/cosmos/cosmos-sdk/blob/master/baseapp/baseapp.go)
- dYdX Cosmos SDK:
  - [cosmos-sdk/baseapp/baseapp.go](https://github.com/dydxprotocol/cosmos-sdk/blob/master/baseapp/baseapp.go)

### Description

The transaction containing a message that has no message handler (and eventually will not be possible to execute in the DeliverTx mode) should not be added to the mempool during CheckTx.

This is documented [here](#) when explaining the CheckState transitions and [here](#) - when explaining what parts of `RunTx()` are executed in different modes.

The actual check should be a part of `RunMsgs()` function, being called from `RunTx()`, once again with the appropriate mode. For CheckTx this would be: `runTxModeCheck`. This is stated in the documentation [here](#):

`RunMsgs` is called from `RunTx` with `runTxModeCheck` as parameter to check the existence of a route for each message the transaction, and with `runTxModeDeliver` to actually process the `sdk.Msg`s. First, it retrieves the `sdk.Msg`'s fully-qualified type name, by checking the `type_url` of the Protobuf `Any` representing the `sdk.Msg`. Then, using the application's `msgServiceRouter`, it checks for the existence of `Msg` service method related to that `type_url`. At this point, if `mode == runTxModeCheck`, `RunMsgs` returns. Otherwise, if `mode == runTxModeDeliver`, the `Msg` service RPC is executed, before `RunMsgs` returns.

Since the issue was noticed for the v0.47 of Cosmos SDK, links to the code and the documentation are for this version. Still, we did check the state on the main branch of Cosmos SDK and we have found the same [issue](#).

## Problem Scenarios

The missing check described above allows invalid messages to be inserted into the mempool during the CheckTx process and impacts the optimization of the blocks creation. Although these blocks may be accepted, the slots in the block are wasted and could have been used for transactions actually containing the applicable state changes. A simple filtering out in advance would guarantee the production of more useful blocks.

Once the block is committed, those invalid transactions will be removed from the mempool - which makes this issue less severe.

In cases when custom ante handlers are made or the existing ones are being skipped, we could be in a situation that special messages could be added to the mempool completely free (fee deduction and gas consumption ante handlers checks are skipped ) and signers are not checked at all.

Due to ABCI++ possibilities, proposers could propose blocks with injected messages like in dYdX project where those injected messages are not being charged.

This and similar scenarios, open the possibility of abusing this oversight even more.

## Recommendation

Since this issue was communicated with Cosmos SDK team, the [solution](#) was provided and applied to the Cosmos SDK main branch during the ongoing dYdX audit.

The missing check for confirming that the message handler exists for each of the messages in the tx, was placed in `runTx()` after the validate basic tx check.

```
for _, msg := range msgs {
 handler := app.msgServiceRouter.Handler(msg)
 if handler == nil {
 return sdk.GasInfo{}, nil, nil, errorsmod.Wrapf(sdkerrors.ErrUnknownRequest,
 "no message handler found for %T", msg)
 }
}
```

## Remediation Plan

**Risk accepted:** The Cosmos SDK team resolved this [issue](#). The dYdX team will consider backporting this issue before the mainnet release of v4 open-source software.

## Custom `runCheckTxConcurrently()` should contain existing handler msgs check

|                       |                                                                                                                         |
|-----------------------|-------------------------------------------------------------------------------------------------------------------------|
| <b>Title</b>          | Custom <code>runCheckTxConcurrently()</code> should contain existing handler msgs check                                 |
| <b>Project</b>        | dYdX v4                                                                                                                 |
| <b>Type</b>           | IMPLEMENTATION                                                                                                          |
| <b>Severity</b>       | 1 LOW                                                                                                                   |
| <b>Impact</b>         | 2 MEDIUM                                                                                                                |
| <b>Exploitability</b> | 1 LOW                                                                                                                   |
| <b>Issue</b>          | <a href="https://github.com/dydxprotocol/cosmos-sdk/issues/20">https://github.com/dydxprotocol/cosmos-sdk/issues/20</a> |

### Involved artifacts

- dYdX Cosmos SDK fork:
  - [cosmos-sdk/baseapp/baseapp.go](https://github.com/dydxprotocol/cosmos-sdk)

### Description

A thorough analysis of Cosmos SDK's `RunTx()` for modes: `runTxModeCheck`, `runTxModeReCheck` and newly added dYdX Cosmos SDK's fork `runCheckTxConcurrently()` functions is performed in order to confirm that all necessary checks are being executed prior to adding the transaction to the mempool.

It has been noticed that `RunMsgs()` is missing a check whether the application's `msgServiceRouter` contains a handler for each of the messages contained in the tx. This finding is explained [here](#). Since this check:

```
handler := app.msgServiceRouter.Handler(msg)
if handler == nil {
 return nil, sdkerrors.Wrapf(sdkerrors.ErrUnknownRequest, "can't route message %v", msg)
}
```

should be a part of the flow executing in case of the `runTxModeCheck`, it should be a part of the custom dYdX `runCheckTxConcurrently()` which is now called instead of the `RunTx()`.

### Problem Scenarios

Without the check listed above, there is a possibility of invalid (non-executable) transactions from the mempool reaching to the proposed blocks.

## Recommendation

`runCheckTxConcurrently` () should contain the following check:

```
for i, msg := range msgs {
 handler := app.msgServiceRouter.Handler(msg)
 if handler == nil {
 err = sdkerrors.Wrapf(sdkerrors.ErrUnknownRequest, "can't route message %+v",
msg)
 }
}
```

The check could be placed above the critical section part [here](#), which would be inline with the solution Cosmos SDK team applied [here](#).

## Remediation Plan

**Risk accepted:** The dYdX team accepted the risk of this issue for now. They will apply the fix before the mainnet release of v4 open-source software.

## Analysis of CometBFT fork changes and CometBFT configuration

|                       |                                                              |
|-----------------------|--------------------------------------------------------------|
| <b>Title</b>          | Analysis of CometBFT fork changes and CometBFT configuration |
| <b>Project</b>        | dYdX v4                                                      |
| <b>Type</b>           | IMPLEMENTATION                                               |
| <b>Severity</b>       | 1 LOW                                                        |
| <b>Impact</b>         | 1 LOW                                                        |
| <b>Exploitability</b> | 1 LOW                                                        |
| <b>Issue</b>          |                                                              |

### Involved artifacts

- [v4/cmd/dydxprotocol/cmd/config.go](#)
- [cometbft/config/config.go](#)
- [cometbft/mempool/dydx\\_helpers.go](#)
- [cometbft/mempool/v1/mempool.go](#)
- [cometbft/mempool/metrics.go](#)

### Description

- dYdX team uses `v1` version of CometBFT mempool - priority mempool. It is important to note that:
  - this version is deprecated in `0.37.x` and will be removed in the `0.38.x`.
  - also it seems that it had a concurrency issue, see [this comment in the PR](#).

The CometBFT team did not work on resolving possible issues, due to this mempool version will be completely removed.

Auditing team concluded that the only reason for dYdX's usage of v1 mempool was due to TTL option. There is the possibility of defining the number of blocks with CometBFT configuration parameter `TTLNumBlocks` to keep the transaction in the mempool in case it is still not included in the block.

Since the `PrepareProposal` introduces the possibility of custom logic during proposing blocks, this feature could be implemented with custom prepare proposal logic.

The known issue, shared above is not impacting the dYdX since [local client connection which FlushSync is a no-op](#). This is impacting the critical section in a way that [the unlock and lock are pushed after this line in the code](#).

- Also, we noticed that `KeepInvalidTxnInCache` configuration parameter is set to true, when initializing dYdX app. This parameter makes it impossible for a once rejected transaction to be accepted upon being examined the second time.

We could not see any issues arising from this, but we want to flag it nonetheless given that we did not yet examine the full system. It would be beneficial to document the reasons for setting

`KeepInvalidTxInCache`.

- The usage of `async client` is understandable to us, since there is a need for concurrently performing parts of the `CheckTx()` to gain speed. The locks placed in the dYdX CometBFT fork are analyzed and the critical section under lock in `runCheckTxConcurrently()` is considered, as well. Due to possible writes that could happen to the `memclob` during `CheckTx()` it is understandable that this part should not happen in parallel.

Auditing team suggests to once again review the possible impact of `x/clob` module to this part of the system, as noted in recommendation and Further Increasing Confidence paragraph.

## Problem Scenarios

CometBFT team will no longer provide support for `v1` version of the mempool and will remove it from `v0.38.x`. This would mean that dYdX team needs to maintain `v1` version, and follow the changes made in `v0` version of the mempool. This would, in our opinion, made the entire development process much more complicated. Due to not being able to guarantee the correctness of the `v1` version of the mempool (there could be more potential issues), but since the CometBFT team plans to maintain `v0.38.x` at least until the end of 2024, we marked this issue as Low severity.

## Recommendation

Consider replacing `v1` mempool with `v0` mempool and adding custom logic to `PrepareProposal` in order to gain TTL options feature.

Document all the reasons behind the specifics about CometBFT configuration.

Perform one more review of the `x/clob` module impact on `CheckTx()` and specifics behind dYdX CometBFT fork once the `x/clob` development and refactoring is finalized.

## Remediation Plan

**Risk accepted:** There are no “known issues” in `v1` mempool that are impacting dYdX v4 open-source software.

Since no *official* EOL date is known at the moment for `v0.37.x` (it will be after the end of 2024), the dYdX team decided to continue with using the `v1` mempool. They will revisit this topic timely and potentially shift to supported version of the mempool in some future release.



## Performance optimizations in x/clob

|                       |                                     |
|-----------------------|-------------------------------------|
| <b>Title</b>          | Performance optimizations in x/clob |
| <b>Project</b>        | dYdX v4                             |
| <b>Type</b>           | IMPLEMENTATION                      |
| <b>Severity</b>       | 1 LOW                               |
| <b>Impact</b>         | 1 LOW                               |
| <b>Exploitability</b> | 1 LOW                               |
| <b>Issue</b>          |                                     |

### Involved artifacts

- [x/clob/types/operation.go](#)
- [x/clob/keeper/liquidations.go](#)

### Description

- [GetInternalOperationsQueueTextString\(\)](#) may be optimized by using `strings.Builder`
- `big.Int` with value 2 is creating new objects in the for loop at three places ([place1](#), [place2](#), [place3](#)) unnecessarily. This object should be created before the loop since it has a constant value. Also, [place1](#) and [place3](#) have the same calculation, so `validatorVolumeQuoteQuantumsPerMarket[clobPairId]` may get value from `validatorVolumeQuoteQuantums`
- Index could be used for adding values in the `messages` in the [GetMessages\(\)](#) function:

```
for i, message := range om.Messages {
 messages[i] = message.Message
}
```

- All comparisons and calculations in the [GetMaxLiquidatableNotionalAndInsuranceLost\(\)](#) can simply be done with `uint64` variables and only convert the final value to `big.Int`.

```
/ Calculate the maximum notional amount that the given subaccount can liquidate in
this block.
liqConfigMaxNotionalLiquidated :=
liquidationConfig.SubaccountBlockLimits.MaxNotionalLiquidated
maxNotionalLiquidatable := liqConfigMaxNotionalLiquidated -
subaccountLiquidationInfo.NotionalLiquidated
if maxNotionalLiquidatable > liqConfigMaxNotionalLiquidated {
 panic(types.ErrLiquidationExceedsSubaccountMaxNotionalLiquidated)
}
```

```
// Calculate the maximum insurance fund payout amount for the given subaccount in
// this block.
liqConfigMaxQuantumsInsuranceLost :=
liquidationConfig.SubaccountBlockLimits.MaxQuantumsInsuranceLost
maxQuantumsInsuranceLost := liqConfigMaxQuantumsInsuranceLost -
subaccountLiquidationInfo.QuantumsInsuranceLost
if maxQuantumsInsuranceLost > liqConfigMaxQuantumsInsuranceLost {
 panic(types.ErrLiquidationExceedsSubaccountMaxInsuranceLost)
}

return new(big.Int).SetUint64(maxNotionalLiquidatable), new(big.Int).SetUint64(maxQuantumsInsuranceLost), nil
```

New big.Int or big.Rat object is created several times unnecessarily:

- New big.Int or big.Rat object is created several times unnecessarily: `GetBankruptcyPriceInQuoteQuantums()`, `GetFillablePrice()`. All of these operations could use one object for execution. For example, `GetBankruptcyPriceInQuoteQuantums()` can create one `big.Int` object instead of four:

```
deltaQuoteQuantums := new(big.Int).Sub(pmmrBig, pmmradBig)
if deltaQuoteQuantums.Sign() == -1 {
 panic("GetBankruptcyPriceInQuoteQuantums: abs(DMMR) is negative")
}

deltaQuoteQuantums.Mul(tncBig, deltaQuoteQuantums)
deltaQuoteQuantums.Div(deltaQuoteQuantums, tmmrBig)
deltaQuoteQuantums.Sub(negDnnvBig, deltaQuoteQuantums)
```

## Problem Scenarios

It was concluded that the code changes in places above were not critical for gaining significant difference in speed, but suggested optimizations could slightly improve performance.

## Recommendation

As explained in the Description section.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.

## Minor code changes in x/clob module

|                       |                                     |
|-----------------------|-------------------------------------|
| <b>Title</b>          | Minor code changes in x/clob module |
| <b>Project</b>        | dYdX v4                             |
| <b>Type</b>           | IMPLEMENTATION                      |
| <b>Severity</b>       | 0 INFORMATIONAL                     |
| <b>Impact</b>         | 0 NONE                              |
| <b>Exploitability</b> | 0 NONE                              |
| <b>Issue</b>          |                                     |

### Involved artifacts

- [x/clob/memclob/memclob.go](#)
- [x/clob/keeper/order\\_state.go](#)
- [x/clob/ante/clob.go](#)
- [lib/constants.go](#)
- [x/clob/keeper/orders.go](#)
- [x/clob/types/quantums.go](#)
- [x/clob/types/orderbook.go](#)

### Description

- Not used functions: [GetCancelOrder](#), [GetAllOrderFillStates](#), [GetOrdersFilledDuringLatestBlock](#) -> [Key](#)
- Use [BigRat0\(\)](#) instead of [big.NewRat\(0, 1\)](#) [here](#).
- [This](#) part of the code may be simplified in the following way:

```
resultRounded := lib.BigRatRound(result, result.Sign() <= 0)
```

- [FillAmountToQuoteQuantums\(\)](#) function could be slightly optimized and simplified, e.q.:

FillAmountToQuoteQuantums() function could be slightly optimized and simplified, e.g.:

```
bigNotional := new(big.Int).Mul(bigSubticks, bigBaseQuantums)
...
if exponent < 0 {
 // `1 / 10^exponent` is an integer.
 return bigNotional.Div(bigNotional, bigExponentValue)
}

// `10^exponent` is an integer.
return bigNotional.Mul(bigNotional, bigExponentValue)
```

- `TotalOpenOrders uint` [def here](#) is used only for observability purposes and there are [decrementing](#) and [incrementing](#) operations over this field. Add checks and log in case maximum uint number is reached or in case an orderbook contains no orders.
- **Suggestions:**
  - As *I* represents [the number of supported assets](#) (for now it's only USDC), the constant for this value should be made.
  - Values comparing and checking value sign for goLang's math/big variables are not very readable. It may be considered to create a wrapper for this logic.
    - Sign checks: `.Sign() == -1/0/1` could be wrapped with `IsNegative()`, `IsZero()`, `IsPositive()`, ...;
    - Compares: `.Cmp() == -1/0/1` could be wrapped with `Equals()`, `GT()`, `GTE()`, `LT()`, `LTE()`.
- **Inline code comments and logging suggestions:**
  - Wrong [comment](#): *No need to process short term order **cancelations** on `ReCheckTx`.*  
-> **placements** instead of **cancelations**
  - Logging of placing and cancelling of short/stateful orders should be done in a same way if it is important to log the type of the order and for consistency. For cancelling is done only for receiving the cancel order [here](#) vs for placing order it is done by type and additional info: [here](#) and [here](#)

## Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements as well for some additional logging and inline code improvements, for easier understanding and readability of the code.

## Recommendation

As explained in the Description section.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.

## Validation of certain fields does not exist

|                       |                                             |
|-----------------------|---------------------------------------------|
| <b>Title</b>          | Validation of certain fields does not exist |
| <b>Project</b>        | dYdX v4                                     |
| <b>Type</b>           | IMPLEMENTATION                              |
| <b>Severity</b>       | 1 LOW                                       |
| <b>Impact</b>         | 1 LOW                                       |
| <b>Exploitability</b> | 1 LOW                                       |
| <b>Issue</b>          |                                             |

### Involved artifacts

- [x/clob/types/message\\_place\\_order.go](#)
- [x/clob/keeper/clob\\_pair.go](#)

### Description

- `MsgPlaceOrder()` - `TimeInForce` - currently there is only check for `Order_SIDE_UNSPECIFIED` in `ValidateBasic`. Checks should be added in `ValidateBasic()` to validate that field has the expected values. Also, there are missing tests that would cover different values of this field.
- Validations `validateClobPair()` - missing `clobPair.Status` validation. What if value is > 4? [here](#). Also, there are missing tests that would cover different values of this field.

### Problem Scenarios

Currently in the code, there are no assumptions that this fields hold valid values. In case of further development, we could easily come in a situation, specially with the `MsgPlaceOrder ValidateBasic()` function to rely that `TimeInForce` was validated and to introduce new code that relies on this field having only valid values.

`ValidateClobPair()` is called only from `InitGenesis()` but it should certainly check if this field holds valid value.

### Recommendation

As described above.

### Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.

## Minor code changes in Place and Cancel Order Msg flows

|                       |                                                        |
|-----------------------|--------------------------------------------------------|
| <b>Title</b>          | Minor code changes in Place and Cancel Order Msg flows |
| <b>Project</b>        | dYdX v4                                                |
| <b>Type</b>           | IMPLEMENTATION                                         |
| <b>Severity</b>       | 0 INFORMATIONAL                                        |
| <b>Impact</b>         | 0 NONE                                                 |
| <b>Exploitability</b> | 0 NONE                                                 |
| <b>Issue</b>          |                                                        |

### Involved artifacts

- [x/clob/keeper/orders.go](#)
- [x/clob/memclob/memclob.go](#)
- [x/clob/keeper/process\\_single\\_match.go](#)
- [x/clob/types/operation.go](#)

### Description

- `func (k Keeper) placeOrder()` is called only from `PlaceShortTermOrder()`. Suggestion is to rename this function to avoid confusion.
- This call of `PerformStatefulValidation()` should not have the `isPreexistingStatefulOrder = true` value of the argument.
- Define function for validating the GTBT values for placing and cancelling stateful orders: [here](#) and [here](#), we could simplify the code; for short term [here](#) and [here](#).
- Checks performed in `matchOrder` could be a part of `mustPerformTakerOrderMatching`, since only error of certain type is assigned to matching error [here](#) and [here](#). By moving the invariant check when the matching cycle is done, some unneeded further execution could be skipped.
- There is a duplication of the code: [here](#) and [here](#). For easier readability define function for retrieving the `fillAmount` and call it for non liquidation taker and maker orders, respectfully.
- Add validation of type of order placed in `decodeOperationRawShortTermOrderPlacementBytes`. Confirm that decoded order is of short term type.
- `AddOrderToOrderbookCollatCheck()`:
  - As written in the `TODO`, there is no need for a map, and accordingly, it is enough to return only `UpdateResult` instead of a map;
  - `clobPairId` equality check is unnecessary, even the `clobPairId` parameter is redundant because it is always passed the same value as in `PendingOpenOrder`;
  - Since `success` is redundant with `successPerUpdate[SubaccountId].IsSuccess()`, and is not used in functions that call it, it can be deleted.

- `ConvertToUpdates()`:
  - This line can be moved above the `assetUpdates` initialization and used for `len()` ;
  - Since `len(assetUpdates) == len(pendingAssetUpdates)` this should be checked at the beginning of the loop.

## Problem Scenarios

Findings listed above could not introduce any issues, they are suggestions for code improvements as well for some additional logging and inline code improvements, for easier understanding and readability of the code.

## Recommendation

As described above.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.

Regarding the minor issue that follows below, dYdX team agreed that it is not necessary for

- `PerformStatefulValidation()` to have the `isPreexistingStatefulOrder = true` value,

but since it is unused anyways maybe just add a comment to avoid the confusion or ignore the observation listed in this finding.

## Deleveraging algorithm needs improvement

|                       |                                          |
|-----------------------|------------------------------------------|
| <b>Title</b>          | Deleveraging algorithm needs improvement |
| <b>Project</b>        | dYdX v4                                  |
| <b>Type</b>           | IMPLEMENTATION                           |
| <b>Severity</b>       | 1 LOW                                    |
| <b>Impact</b>         | 1 LOW                                    |
| <b>Exploitability</b> | 0 NONE                                   |
| <b>Issue</b>          |                                          |

### Involved artifacts

- [x/clob/keeper/liquidations\\_deleveraging.go](https://x/clob/keeper/liquidations_deleveraging.go)

### Description

When Deleveraging the subaccounts for liquidation - current algorithm in `OffsetSubaccountPerpetualPosition()` will always **select the first subaccount** with an open different side position to offset the liquidated subaccount. There is no selection criteria so the first user's subaccount we reach in the loop, when reading data from the store, will always be the one we are offsetting the liquidated subaccounts against for the same positions.

### Problem Scenarios

For the same positions, same account will be deleveraged each time.

### Recommendation

It seems that the optimal algorithm would be to include the size of the position each subaccount has and deleverage the one with biggest size.

### Remediation Plan

**Risk accepted:** The dYdX team accepts this finding. While no immediate improvements to the algorithm are planned upon the generation of this report, there is an open pull request (PR) that introduces a potential solution for addressing this discovery. Randomly selected subaccount will be picked, seeded on last committed block time (we assume this would be done by the proposer) for offsetting the liquidated positions against. This will be a slight improvement over selecting the same subaccount ordering.



## Unneeded notifyTxsAvailable call for v1 mempool

|                       |                                                 |
|-----------------------|-------------------------------------------------|
| <b>Title</b>          | Unneeded notifyTxsAvailable call for v1 mempool |
| <b>Project</b>        | dYdX v4                                         |
| <b>Type</b>           | IMPLEMENTATION                                  |
| <b>Severity</b>       | 1 LOW                                           |
| <b>Impact</b>         | 1 LOW                                           |
| <b>Exploitability</b> | 1 LOW                                           |
| <b>Issue</b>          |                                                 |

### Involved artifacts

- [mempool/v1/mempool.go](#)

### Description

In v1 - priority mempool, for the inserted short order Msg tx (place & cancel Msgs) mempool will be [notifying](#) the consensus that there are transactions present.

While in v0 mempool's `resCbFirstTime()` there is a check added [here](#):

```
if mempool.IsShortTermClobOrderTransaction(memTx.tx, mem.logger) {
 return
}
mem.notifyTxsAvailable()
```

in v1 priority mempool's `addNewTransaction()`, mempool will [ping](#) consensus in case of any tx being placed in the mempool, including Short Term Order Msg Tx (Place and Cancel Order).

### Problem Scenarios

"This does not cause issues, but [pings consensus](#) even with only short-term orders in the mempool, potentially leading to empty blocks - if no other transactions are present or if no injected messages from the proposing validator are added. According to v3 stats, a significant influx of short-term orders is expected: 212 Place Order Msgs/sec, with the majority being short-term (99.95%), and only a small percentage likely to match (fill to order is 1/125 across all markets, 1/5 for ETH)."

## Recommendation

Skip notifying the consensus if Short Term Order message tx has been added to the v1 mempool.

Monitor the content of the created blocks if a third party deploys dYdX v4 open-source software and how often will the prices be updated, premiums added, shot term orders placed...

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.

## Gossiping Short Term order txs not possible due to deletion from the mempool

|                       |                                                                              |
|-----------------------|------------------------------------------------------------------------------|
| <b>Title</b>          | Gossiping Short Term order txs not possible due to deletion from the mempool |
| <b>Project</b>        | dYdX v4                                                                      |
| <b>Type</b>           | PROTOCOL DOCUMENTATION                                                       |
| <b>Severity</b>       | 2 MEDIUM                                                                     |
| <b>Impact</b>         | 2 MEDIUM                                                                     |
| <b>Exploitability</b> | 04 UNKNOWN                                                                   |
| <b>Issue</b>          |                                                                              |

### Involved artifacts

- [mempool/v1/mempool.go](#)

### Description

The broadcasting routine could still be ongoing, when the `ReCheckTx` phase is triggered.

Since during the `ReCheckTx`, Short Term Order Msg `tx` is removed from the mempool, the `tx` could be deleted from the mempool, prior to being gossiped from the node. Further analysis will explain the processing order and how this is possible.

When `BlockExecutor` is committing the block - mempool is updated with `Update()` function.

When updating the mempool, this is done under lock, so nothing can happen with mempool in parallel. `Txs` that were part of the committed block are deleted from the memclob, and for the others we will trigger the `Recheck` [here](#), since the `Recheck` config parameter is set to true [here](#) (no override [here](#)) - `Recheck` phase is active. Prior to actually calling the execution of `recheck` for each of the `txs` left in the mempool on the app level, we will remove that `tx` [here](#), in `recheckTransactions()`.

So, on the app level there will be nothing going on for `Recheck()` of short order txs [here](#), but they will be removed from the mempool after committing the block and prior to rechecking the remaining txs in the mempool.

### Problem Scenarios

When a `ReCheckTx` removes a Short Term Order `tx`, nodes that received it through gossiping routine will keep the `tx` and if they are proposers they can eventually propose it.

Nodes that did not reach by the time most of the nodes finished the consensus and executed `ReCheckTx` will not get this `tx` unless via the proposal.

So the only way of getting this transaction to the node is by:

1. gossiping it or
2. receiving it over prepare proposal.

The propagation of the transaction across all nodes follows a probabilistic pattern.

In certain scenarios, a transaction might successfully propagate to a specific number of full nodes. However, variations in sentry node arrangements among different validators and network latency, could potentially lead to instances where the transaction fails to reach any validator.

**Mempool connection requirements** are broken, specifically referring to property 13 which states:

*Requirement 13 ensures that a transaction will eventually stop oscillating between `CheckTx` success and failure if it stays in  $p$ 's mempool for long enough. This condition on the Application's behavior allows the mempool to ensure that a transaction will leave the mempool of all full nodes, **either because it is expunged everywhere due to failing `CheckTx` calls, or because it stays valid long enough to be gossiped, proposed and decided.***

*Although Requirement 13 defines a global  $h_{stable}$ , application developers can consider such stabilization height as local to process  $p$  ( $h_{p,stable}$ ), without loss for generality. In contrast, the value of  $b$  MUST be the same across all processes.*

## Recommendation

Since the placement of short term orders is always in a *best effort* and most of them will not be filled (they will be canceled, removed due to GTB height ...) it will be hard to distinct whether a short term order was "captured" on a full node in its mempool or it just expired due to GTB (expiration window).

It is understood that the reliability of short term order `txs` is not a priority, but the speed of processing them and storing such transactions in the state exclusively upon confirmation of an order match.

However, we are reporting this issue since the basic property of ABCI++ application requirements is broken.

If there is no way of guaranteeing this property, one can be redefined in the dYdX spec due to the specifics of the dYdX chain. Possible situation can be explained as acceptable and expected. Existing app-level mechanism for pruning expired orders could also be explained as a way of pruning the `txs` "locked" in full nodes.

## Remediation Plan

**Functioning as Designed:** as concluded by the auditing team.

The dYdX team agrees that this finding is valid, and to keep the severity as set by the auditing team. A fix would increase the reliability of short term orders but they are considered to be best effort currently, so the severity doesn't need to be high.

## Results of the order matching should be atomically persisted to memclob

|                       |                                                                         |
|-----------------------|-------------------------------------------------------------------------|
| <b>Title</b>          | Results of the order matching should be atomically persisted to memclob |
| <b>Project</b>        | dYdX v4                                                                 |
| <b>Type</b>           | IMPLEMENTATION                                                          |
| <b>Severity</b>       | 1 LOW                                                                   |
| <b>Impact</b>         | 1 LOW                                                                   |
| <b>Exploitability</b> | 1 LOW                                                                   |
| <b>Issue</b>          |                                                                         |

### Involved artifacts

- [x/clob/memclob/memclob.go](https://x/clob/memclob/memclob.go)

### Description

Persisting the matching results in the memclob after the matching cycle should be considered as an atomic operation. That is, if one taker order produces one maker order removal (for some reason), some filled matches and at the end the matching is not considered valid, all the changes should be persisted to the memclob - or none.

The matching results will not be saved in the memclob in cases when FOK/Post-only [invariants are broken](#), but the removal of the order will be done.

E.g. this can happen in cases of “self-trading” that is, if the subaccount has a maker order open on the position and then places a new taker order. This is processed [here](#), so that the maker order will be removed with the removal reason: `OrderRemoval_REMOVAL_REASON_INVALID_SELF_TRADE`.

In cases when taker order has been matched with other maker order(s), but the results are not valid (processed in the code [here](#)) the results of the matching will not be persisted in memclob and `operationsToPropose`.

### Problem Scenarios

Since the orders will be removed from the memclob and orderbook, it is not possible for this order to be matched even though the taker order initiating the removal was also not matched.

There could be the possibility of filling the removed order with some new taker orders arriving into the memclob.

### Recommendation

Reconsider how the self trading situations should be resolved.

Both removal orders and filled orders should be processed only in cases when there is no invalid result produced during matching, since matching per order is considered as an atomic operation.

## Remediation Plan

**Functions As Designed:** At the moment of producing this report, dYdX team has agreed with this finding, but shared that both current implementation (non atomically) and proposed way (atomically) are both valid ways to match orders, as long as we make it clear to users.

The dYdX team will double check current implementation does not impact correctness. If concluded that changes are needed, this finding should be marked as Medium severity.

- Currently, the `mustPerformTakerOrderMatching` function modifies values in state. So the reason current implementation doesn't atomically persist results of order matching to memclob is because memclob needs to reflect state values.
- If implementation will atomically persist order matching results to memclob, then it needs to be able to modify how it manipulates with state as well.

## Optimizing Efficiency of StatefulOrdersTimeSlice KVStore

|                       |                                                          |
|-----------------------|----------------------------------------------------------|
| <b>Title</b>          | Optimizing Efficiency of StatefulOrdersTimeSlice KVStore |
| <b>Project</b>        | dYdX v4                                                  |
| <b>Type</b>           | IMPLEMENTATION                                           |
| <b>Severity</b>       | 1 LOW                                                    |
| <b>Impact</b>         | 1 LOW                                                    |
| <b>Exploitability</b> | 1 LOW                                                    |
| <b>Issue</b>          |                                                          |

### Involved artifacts

- [x/clob/keeper/stateful\\_order\\_state.go](#)

### Description

The existing structure of the `StatefulOrdersTimeSlice` KVStore, which uses the key as `StatefulOrdersTimeSlicePrefix + GTBT` and the value as `[]OrderId`, presents opportunities for optimization in terms of the efficiency of reading from this store. The current utilization of this store could be enhanced by revising its key structure and adopting more efficient access patterns.

### Problem Scenarios

Several key operations within the codebase are linked to the `StatefulOrdersTimeSlice` KVStore, which involves adding, removing, and managing stateful orders based on the `GoodTilBlockTime` (GTBT). Here are the specific scenarios:

1. **Adding Orders:** During the addition of a new order to the store, `orderIds` are retrieved based on the appropriate `GoodTilBlockTime`. Following this, a `check` is performed to determine if the given `orderId` already exists within the store.
2. **Removing Orders:** When removing a stateful order from the store, all `orderIds` associated with the relevant `GoodTilBlockTime` are supplied. Then, a new slice is generated, excluding the `orderId` of the order being removed. This new slice is then saved back into the store.
3. **Deleting Expired Orders:** Process of deleting expired stateful orders based on their `GoodTilBlockTime`.

### Recommendation

To optimize the efficiency of the `StatefulOrdersTimeSlice` KVStore, we suggest considering the following modifications:

- **Key Restructuring:** Alter the key structure so that it consists of `Prefix + GTBT + OrderId`. This adjustment can significantly improve data access efficiency.
- **Use of Complete Key:** In scenarios 1 and 2 (addition and deletion of orders), utilize a complete key for fast existence checks and retrieval of the corresponding records.
- **Iterator Utilization:** In the process of deleting expired orders (scenario 3), iterate through keys without the `OrderId` part, i.e., using `Prefix + GTBT`. This way, you can efficiently retrieve all `OrderIds` that have expired by the given GTBT.
- **Value Optimization:** Given that value retrieval isn't necessary for this context, set a minimal value size, perhaps 1 byte, to further optimize storage usage.

By incorporating these recommendations, you can significantly enhance the efficiency and performance of the `StatefulOrdersTimeSlice` KVStore, leading to faster read speeds and improved overall execution of the code.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.



## Optimizing Memory Efficiency for OrderId Structure in KVStores

|                       |                                                                |
|-----------------------|----------------------------------------------------------------|
| <b>Title</b>          | Optimizing Memory Efficiency for OrderId Structure in KVStores |
| <b>Project</b>        | dYdX v4                                                        |
| <b>Type</b>           | IMPLEMENTATION                                                 |
| <b>Severity</b>       | 1 LOW                                                          |
| <b>Impact</b>         | 1 LOW                                                          |
| <b>Exploitability</b> | 1 LOW                                                          |
| <b>Issue</b>          |                                                                |

### Involved artifacts

- [x/clob/keeper/order\\_state.go](#)
- [x/clob/keeper/stores.go](#)
- [x/clob/keeper/stateful\\_order\\_state.go](#)

### Description

The current implementation utilizes the `OrderId` structure, which is composed of a `SubaccountId` structure and additional fields. This structure is utilized across various KVStores as either a part of the key or as part of the value. However, the use of `OrderId` in this manner could potentially lead to memory inefficiencies within these KVStores.

### Problem Scenarios

The inefficiency becomes apparent in several cases:

1. **Key and Value Usage:** The `OrderId` structure is used as a part of the key and/or value within several KVStores. For instance, it's used in the `OrderAmountFilled` store as part of the key, in `BlockHeightToPotentiallyPrunableOrders` and `StatefulOrdersTimeSlice` as part of the value ( `[]OrderId` ), and in `StatefulOrderPlacement` as both part of the key and part of the value.
2. **Memory Overhead:** Since the `OrderId` structure holds a `SubaccountId` structure and additional fields, it can lead to memory bloat within these KVStores, especially considering that these structures might be stored across multiple KVStores.

### Recommendation

To address the memory inefficiencies and optimize memory usage in these KVStores, we recommend implementing the following changes:

- **Introduce a Unique Index:** Consider introducing a unique index, such as a `uint64`, that corresponds to each `OrderId`. This index would serve as an identifier for each `OrderId` entry.
- **Index-based Storage:** Instead of storing the entire `OrderId` structure, store only the unique index in the KVStores. This would significantly reduce memory usage within the KVStores.
- **Mapping KVStore:** Create a new KVStore specifically for mapping the `OrderId` to its corresponding unique index. This store would serve as a lookup table, facilitating efficient retrieval of `OrderId` entries based on the unique index.

By implementing these recommendations, you can significantly relieve memory usage within the KVStores, optimize the storage of `OrderId` entries, and improve the overall efficiency and performance of the codebase.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future or maybe at all.

The dYdX team has downgraded the severity of this issue from Medium to Low, a decision that has been endorsed by the auditing team. This adjustment is attributed to the finding's focus on performance enhancement rather than affecting correctness. Notably, the auditing team has not conducted benchmark tests to ascertain the extent of the optimization's value.

The migration to having a unique key -> `orderId` would be non-trivial, as dYdX team concludes.

## Division by Zero Issue in GetBankruptcyPriceInQuoteQuantums Function

|                       |                                                                      |
|-----------------------|----------------------------------------------------------------------|
| <b>Title</b>          | Division by Zero Issue in GetBankruptcyPriceInQuoteQuantums Function |
| <b>Project</b>        | dYdX v4                                                              |
| <b>Type</b>           | IMPLEMENTATION                                                       |
| <b>Severity</b>       | 0 INFORMATIONAL                                                      |
| <b>Impact</b>         | 3 HIGH                                                               |
| <b>Exploitability</b> | 0 NONE                                                               |
| <b>Issue</b>          |                                                                      |

### Involved artifacts

- [x/clob/keeper/liquidations.go](#)
- [x/subaccounts/keeper/subaccount.go](#)

### Description

The codebase contains a critical issue within the function `GetBankruptcyPriceInQuoteQuantums`. In cases where the variable `tmmrBig` holds a value of 0, a [division by zero](#) panic occurs. The value of `tmmrBig` is obtained from the function `subaccountsKeeper.GetNetCollateralAndMarginRequirements()`. This function internally calls `internalGetNetCollateralAndMarginRequirements()`, where the calculation for `tmmrBig` (`bigMaintenanceMargin`) takes place. This finding highlights the scenarios under which the value of `tmmrBig` can be zero, leading to potential critical failures in subsequent call paths.

### Problem Scenarios

The panic due to division by zero in the `GetBankruptcyPriceInQuoteQuantums` function can be triggered under [specific circumstances](#):

1. If both `assetSizes` and `perpetualSizes` collections are empty, or
2. If the function `GetMarginRequirements` consistently returns 0 for `bigMaintenanceMarginRequirements`.

Given the potential critical call paths of the `GetBankruptcyPriceInQuoteQuantums` function, such as from `FullNodeProcessProposalHandler`, `ProcessProposalHandler`, `PrepareCheckState`, and similar, a panic in this function could lead to chain halts.

## Recommendation

It is necessary to check that `tmmrBig` is not equal to zero and handle such a case in an appropriate way.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but did not agree with the severity. Additional check could be useful, but it was concluded by both teams - auditing and dYdX, that it is likely impossible for `tmmrBig` to be zero if the `subaccount` is `liquidatable` (so by definition they have non-zero MMR)? We [skip constructing the liquidation order](#) if that's the case. The finding's severity is downgraded to Informational. No immediate improvements to the algorithm are planned upon the generation of this report.

## Missing Single Clob Message Check in RateLimitDecorator AnteHandle

|                       |                                                                    |
|-----------------------|--------------------------------------------------------------------|
| <b>Title</b>          | Missing Single Clob Message Check in RateLimitDecorator AnteHandle |
| <b>Project</b>        | dYdX v4                                                            |
| <b>Type</b>           | IMPLEMENTATION                                                     |
| <b>Severity</b>       | 1 LOW                                                              |
| <b>Impact</b>         | 1 LOW                                                              |
| <b>Exploitability</b> | 1 LOW                                                              |
| <b>Issue</b>          |                                                                    |

### Involved artifacts

- [x/clob/ante/clob.go](#)
- [app/ante.go](#)

### Description

The current implementation of the `RateLimitDecorator AnteHandle` lacks a check for a single clob message per transaction, similar to what is present in the `ClobDecorator`. As a result, unnecessary rate limit checks are performed even for transactions that are already known to be invalid. To address this inefficiency and improve code optimization, it's recommended to introduce a mechanism to validate the single clob message per transaction before proceeding with rate limit checks.

### Problem Scenarios

The issue primarily arises due to the absence of a check for single clob messages within the `RateLimitDecorator AnteHandle`. Here's the specific scenario:

1. Transactions containing multiple clob messages can lead to rate limit checks being executed for each clob message, even when the transaction is invalid. This results in additional overhead and impacts performance.

### Recommendation

One suggestion is to wrap `RateLimitDecorator` with `SingleMsgClobTxAnteWrapper` as done with `SetUpContextDecorator`. Also, the `ClobDecorator` could be wrapped with `SingleMsgClobTxAnteWrapper` instead of checking inside the `AnteHandle` itself.

## Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.

## Minor code changes in Rate Limits

|                       |                                   |
|-----------------------|-----------------------------------|
| <b>Title</b>          | Minor code changes in Rate Limits |
| <b>Project</b>        | dYdX v4                           |
| <b>Type</b>           | IMPLEMENTATION                    |
| <b>Severity</b>       | 0 INFORMATIONAL                   |
| <b>Impact</b>         | 0 NONE                            |
| <b>Exploitability</b> | 0 NONE                            |
| <b>Issue</b>          |                                   |

### Involved artifacts

- [x/clob/rate\\_limit/order\\_rate\\_limiter.go](#)
- [x/clob/rate\\_limit/multi\\_block\\_rate\\_limiter.go](#)

### Description

- Wrong value of string "MaxShortTermOrderCancellationsPerMarketPerNBlocks" for context on creation of the RateLimiter for Cancel orders ([single block](#), [multi block](#)).
- Defining the variable `count` is redundant in [this](#) part of the code.

```
perBlockCounts[blockHeight] += 1
```

- Since `len(r.config)` is equal to `len(perRateLimitCounts)` then the `perRateLimitCounts` increment can be performed in the `for` loop [below](#) (before the `if` condition). [This](#) `for` loop is redundant.

### Problem Scenarios

Findings listed above would not introduce any issues, they are suggestions for code improvements as well for some additional logging and inline code improvements, for easier understanding and readability of the code.

### Recommendation

As explained in the Description section.

### Remediation Plan

**Risk accepted:** The dYdX team accepts this finding but due to other higher priority development plans, this issue will not be resolved in the near future.


## 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

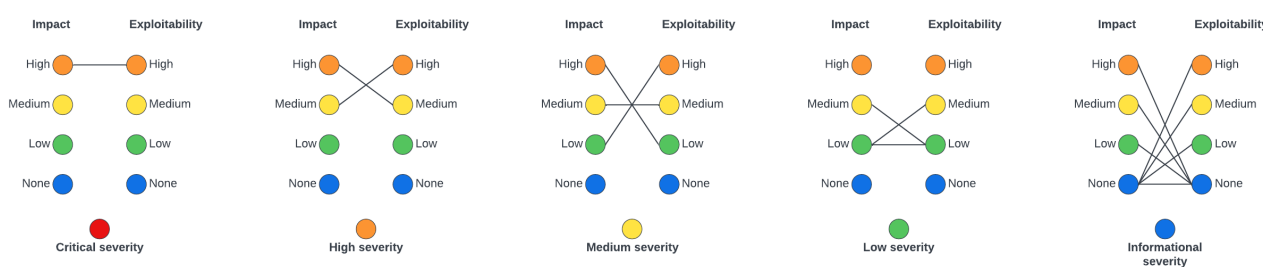


- *legitimate*, e.g. submission of a transaction that follows protocol rules by a user; delegation/redelegation/bonding/unbonding; validator downtime; validator voting on a single, but alternative block; delays in relaying certain messages, or speeding up relaying other messages;
- *illegitimate*, e.g. 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         |