Security Audit Report

dYdX 2024 Q2: Slinky Integration, Reduce Only

Authors: Aleksandar Ignjatijevic, Mirel Dalcekovic

Last revised 15 June, 2024

Table of Contents

Audit Overview

The Project

In April 2024., dYdX has engaged Informal Systems to work on partnership and conduct security audit of the following items in dYdX's open source software:

1. Reduce Only orders
2. Slinky Integration

Scope of this report

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

- Reduce Only orders implementation
- Slinky Integration with dYdX, with important areas to audit
    - ABCI methods: VE logic, prepare process proposal, etc.
    - Communication between dYdX protocol's ` `x/prices` ` and Sidecar
    - Sidecar logic for fetching prices with low priority

Audit plan

The audit was conducted between April 4th and May 9th, 2024.

Timeline

- 04.04.2024. Audit start - Informal Systems auditing team is onboarding.
- 09.04.2024. Kick off meeting, featuring scope agreement and a high-level code walkthrough conducted by the dYdX team for the Reduce Only orders and Slinky Integration. dYdX team members emphasized the importance of concentrating on the Slinky integration, which involves modifications to the dYdX protocol code base and utilization of existing logic from the Slinky repository. Pull Requests (PRs) to be reviewed were shared with the auditing team. Furthermore, it has been communicated that the Sidecar implementation, which resides exclusively within Skip's Slinky repository is considered unnecessary for auditing, and is categorized as a lower-priority item for assessment.
- 10.04.2024. - 23.04.2024. Asynchronous communication and syncing with dYdX team representatives over shared slack channel.
    - draft version of system overview and threat models shared and feedback collected
    - draft issues shared
- 23.04.2024. Weekly sync meeting. Sharing progress with dYdX team representatives.
- 09.05.2024. Draft report for 2024 Q2 shared. New revisions of audit reports for 2023 Q4 and 2024 Q1 audit phases generated and shared.
- 13.05.2024. Closure meeting
- 13.05.2024. Final report for 2024 Q2 shared. The revised version includes suitable statuses for the findings and agreed-upon severity levels for the issues.

Conclusions

Upon conducting a thorough examination of the project, it was noted that the audited Reduce only new feature and Slinky integration do not entail a substantial amount of changes in the code base. However the complexity of Slinky integration, execution of hooks and functions whose logic is placed in several repositories (dYdX Cosmos SDK fork, Slinky oracle and dYdX protocol) required extra careful attention and focus. Several minor issues and protocol level flaws were surfaced, as outlined in the Findings section.

Further Increasing Confidence

At the conclusion of the audit, the auditing team dedicated time to assess the current implementation of the Sidecar process, utilizing the dYdX market map designed for the dYdX chain. The analysis focused on the successful execution of the entire flow, encompassing the initialization and operation of the sidecar oracle, initialization of the orchestrator and providers as per the custom-defined dYdX market map, and the retrieval of prices from the providers, followed by the aggregation of these prices by the oracle server.

Due to time constraints, a comprehensive audit of this scope was not feasible. However, it's worth noting that the client did not designate this scope as a mandatory area for auditing. The client expressed their preference for prioritizing other aspects of the project.

The auditing team recommends treating the Skip’s Sidecar implementation as a distinct scope for auditing, extending beyond its current integration within the dYdX system.

Audit Dashboard

Target Summary

- **Type**: Protocol and Implementation
- **Platform**: Go
- **Artifacts** audited over release/protocol/v5.x branch and fixed d866d2b commit hash.
    - Reduce Only orders
        - PRs containing current RO implementation for review:
            - [CLOB-1058] Allow FOK/IOC Reduce Only Orders
            - E2E Tests for Replacement RO Orders, allow all types of RO order replacements
    - Slinky Integration
        - PRs containing full slinky integration for dYdX for review:
            - Main logic changes (high priority to audit)
                - feat: Slinky sidecar integration
                - feat: Proposal logic for Slinky
                - feat: VoteExtension Slinky logic
                - feat: Slinky full integration PR
                - [SKI-21] Bump slinky version to v0.3.1
            - Optimizations (medium priority to audit)
                - feat: Enforce MarketParam.Pair uniqueness constraint
                - perf: optimize slinky_adapter via in-memory cache [SKI-13]
                - [SKI-1] Remove smoothed prices
            - Testing/dev/staging fixes (low priority to audit)
                - test: Fix flaky slinky client test
                - [SKI-16] Slinky Dockerfile Removal
                - [SKI-18] Fix localnet
                - Remove volatile market
            - Logic implemented in Skip's Slinky repo version v0.3.2 and referenced by dYdX code from the above listed PRs.

Engagement Summary

- **Dates**: 04.04.2024. - 09.05.2024.
- **Method**: Manual code review, protocol analysis

Severity Summary

Finding Severity	#
Critical	0
High	0
Medium	1
Low	2
Informational	9

Finding Severity	**#**
Total	12

Finding: Analysis of ABCI++ requirements contains the analysis of all the ABCI ++ requirements.

Findings

Title	Type	Severity	Status	Issue
Unclear off-chain removal status in cases of FOK/IOC orders removal	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Keeping dYdX forks Updated with Slinky's Cosmos SDK and CometBFT referenced version	OTHER	INFORMATIONAL	ACKNOWLEDGED	
Unnecessary code execution in ABCI methods on VE enabling height	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Inadequate error propagation in prepareProposal	IMPLEMENTATION PROTOCOL	INFORMATIONAL	ACKNOWLEDGED	
Inconsistent error logging between processProposal and prepareProposal handlers	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Code improvements for Slinky related code changes	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Missing automatic detection of validators not reporting certain CP price updates	PROTOCOL	MEDIUM	ACKNOWLEDGED	

Title	Type	Severity	Status	Issue
Telemetry monitoring and Indexer events created when updating prices in extend vote handler	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Unclear handling in cases when GetValidMarketPriceUpdates returns no valid price updates to be injected to VE	IMPLEMENTATION	INFORMATIONAL	ACKNOWLEDGED	
Slinky's ValidateExtendedCommitAgainstLastCommit function contains invalid checks for non-absent votes	IMPLEMENTATION	LOW	ACKNOWLEDGED	
Analysis of ABCI++ requirements	PROTOCOL	LOW	ACKNOWLEDGED	
Reconsider application panicking instead of working with corrupted data	PROTOCOL	INFORMATIONAL	ACKNOWLEDGED	

Unclear off-chain removal status in cases of FOK/IOC orders removal

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	**IMPLEMENTATION**
Severity	**INFORMATIONAL**
Impact	**NONE**
Exploitability	**NONE**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- dydxprotocol/v4-chain/protocol/x/clob/memclob/memclob.go
- dydxprotocol/v4-chain/protocol/indexer/off_chain_updates/types/off_chain_updates.pb.go

Description

Currently `OrderRemoveV1_ORDER_REMOVAL_STATUS_BEST_EFFORT_CANCELED` status will be sent to the off-chain component (indexer) as an update in cases of FOK/IOC orders removals - e.g. code-ref1 and code-ref2. It seems that removal reasons sent to the indexer could be more specific about why the order was removed in case of FOK or POST_ONLY crossing reasons - removal reasons are more descriptive e.g. here.

Problem Scenarios

The off-chain data and the reasons for removing the immediate execution order might not be clear. It appears that the full information will be obtained from both the `takerOrderStatus` and the off-chain removal status when combined.

Recommendation

Consider including in-line comments for specific removal cases to clarify that `OrderRemoveV1_ORDER_REMOVAL_STATUS_BEST_EFFORT_CANCELED` will be sent as an off-chain update for FOK/IOC order removals here.

Also, consider introducing more detailed off-chain removal order status types if this would improve the quality and speed of information retrieved from the indexer.

Keeping dYdX forks Updated with Slinky's Cosmos SDK and CometBFT referenced version

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	OTHER
Severity	INFORMATIONAL
Impact	HIGH
Exploitability	UNKNOWN
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- dYdX fork Cosmos SDK
- dYdX CometBFT fork

Description

With integration of Slinky oracle, it has become crucial to carefully manage existing CometBFT and Cosmos SDK dYdX forks.

Problem Scenarios

With the custom modifications made to dYdX forks, including alterations not just to function signatures but also internal logic, the functioning of the Slinky Oracle could be significantly affected. These changes could have a substantial impact on critical ABCI++ flows crucial for dYdX chain liveness and correct functioning.

Recommendation

Custom changes made to dYdX forks should be verified and tested with the integrated Slinky oracle solution. It is now essential to ensure the careful transfer of all updates, changes, and security patches from the aforementioned Slinky-referenced forks to dYdX forks, when upgrading the dYdX integration to new Slinky releases.

Unnecessary code execution in ABCI methods on VE enabling height

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	**IMPLEMENTATION**
Severity	**INFORMATIONAL**
Impact	**NONE**
Exploitability	**NONE**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- dydxprotocol/v4-chain/protocol/app/vote_extensions/extend_vote_handler.go
- dydxprotocol/v4-chain/protocol/app/process/slinky_market_price_decoder.go

Description

1. **Prepare Proposal Handler:** For the *vote extensions enabling height* dYdX prepare proposal handler will not process oracle injected tx (since Slinky prepare proposal handler did not inject one) and will not create `MsgUpdateMarketPrices` containing actual updates - code. `MsgUpdateMarketPrices` defined as struct will be empty message, with no slice elements in `MarketPriceUpdates` (empty slice).

```
type MsgUpdateMarketPrices struct {
    MarketPriceUpdates []*MsgUpdateMarketPrices_MarketPrice
`protobuf:"bytes,1,rep,name=market_price_updates,json=marketPriceUpdates,proto3
" json:"market_price_updates,omitempty"`
}
```

   Further dYdX process proposal handler execution will validate there is no price updates for the current block, but the empty message must be present - code.
   It seems that processing of the following code in prepare proposal handler is unnecessary. Since the `GetValidMarketPriceUpdates` function will receive some arbitrary tx placed in the first place, other than oracle injected tx.

```
// Grab the injected VEs from the previous block.
// If VEs are not enabled, no tx will have been injected.
var extCommitBzTx []byte
if len(req.Txs) >= constants.OracleVEInjectedTxs {
    extCommitBzTx = req.Txs[constants.OracleInfoIndex]
}

// get the update market prices tx
msg, err := priceUpdateGenerator.GetValidMarketPriceUpdates(ctx, extCommitBzTx)
if err != nil {
    ctx.Logger().Error(fmt.Sprintf("GetValidMarketPriceUpdates error: %v",
err))
    recordErrorMetricsWithLabel(metrics.PricesTx)
    return &EmptyResponse, nil
}
```

Because of the possibility of propagating non oracle injected tx in `GetValidMarketPriceUpdates` function, there is `!ve.VoteExtensionsEnabled(ctx)` check and the empty `MsgUpdateMarketPrices` will be returned with `GetValidMarketPriceUpdates` - code:

```
if !ve.VoteExtensionsEnabled(ctx) {
        // return a nil MsgUpdateMarketPricesTx w/ no updates
        return &pricestypes.MsgUpdateMarketPrices{}, nil
    }
```

This could be checked right away and skip this code block execution with assigning the empty `MsgUpdateMarketPrices`.

2. **Extend Vote Handler:** During `ExtendVote` phase, validators should inject Vote extensions. However, they can't consider the current block's `MsgUpdateMarketPrices` for this because it doesn't contain any updates at the *vote extensions enabling height*. The updates for price currency pairs can be prepared only considering current values in the blocks state at height.
It seems that the following code block, after the `DecodeUpdateMarketPricesTx` could also be skipped in case of `VoteExtensionsEnableHeight = ctx.BlockHeight()`, since all the code would be execute over empty `MsgUpdateMarketPrices`:

```
// Decode the x/prices txn in the current block
        updatePricesTx, err :=
e.PricesTxDecoder.DecodeUpdateMarketPricesTx(ctx, req.Txs)
        if err != nil {
            return nil, fmt.Errorf("DecodeMarketPricesTx failure %w", err)
        }

        // ensure that the proposed MsgUpdateMarketPrices is valid in
accordance w/ stateful information
        // this check is equivalent to the check in ProcessProposal
(indexPriceCache has not been updated)
        err = updatePricesTx.Validate()
        if err != nil {
            return nil, fmt.Errorf("updatePricesTx.Validate failure %w", err)
        }

        // Update the market prices in the PricesKeeper, so that the
GetValidMarketPriceUpdates
        // function returns the latest available market prices.
        updateMarketPricesMsg, ok := updatePricesTx.GetMsg().
(*prices.MsgUpdateMarketPrices)
        if !ok {
            return nil, fmt.Errorf("expected %s, got %T",
"MsgUpdateMarketPrices", updateMarketPricesMsg)
        }

        // Update the market prices in the PricesKeeper
        err = e.PricesKeeper.UpdateMarketPrices(ctx,
updateMarketPricesMsg.MarketPriceUpdates)
        if err != nil {
            return nil, fmt.Errorf("failed to update market prices in extend
vote handler pre-slinky invocation %w", err)
        }
```

Recommendation

Streamline the code as per suggestions given in the description, except if the current version of the code was a trade-off involving omitting height checks, that lead to redundant code execution during a specific corner case - the height when vote extensions are enabled. If this was the intention, include inline code comments for better code readability.

Inadequate error propagation in prepareProposal

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	IMPLEMENTATION PROTOCOL
Severity	INFORMATIONAL
Impact	HIGH
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- slinky-mev/slinky/abci/proposals/proposals.go
- dydxprotocol/v4-chain/protocol/app/prepare/prepare_proposal.go

Description

This particular threat revolves around the propagation of errors between dYdX's `prepare_proposal.go` and Slinky's `proposals.go`. Threat inspection yielded that dYdX `PrepareProposalHandler` keeps sending `nil` instead of various kinds of `err` (example code snippet).

```
pricesTxResp, err := EncodeMarketPriceUpdates(txConfig, msg)
if err != nil {
    ctx.Logger().Error(fmt.Sprintf("GetUpdateMarketPricesTx error: %v", err))
    recordErrorMetricsWithLabel(metrics.PricesTx)
    return &abci.ResponsePrepareProposal{Txs: [][]byte{}}, nil
}
```

If `nil` is returned as `err`, this leads to `err` not being caught in the `if` statement and thus resulting in `injectAndResize` being called on `Txs: [][]byte{}`. The code snippet below illustrates said behavior (link to code snippet).

```
wrappedPrepareProposalStartTime := time.Now()
resp, err = h.prepareProposalHandler(ctx, req)
wrappedPrepareProposalLatency = time.Since(wrappedPrepareProposalStartTime)
if err != nil {
    h.logger.Error("failed to prepare proposal", "err", err)
    err = slinkyabci.WrappedHandlerError{
        Handler: servicemetrics.PrepareProposal,
        Err:     err,
    }
    return &cometabci.ResponsePrepareProposal{Txs: make([][]byte, 0)}, err
}

// Inject our VE Tx ( if extInfoBz is non-empty), and resize our response Txs to
respect req.MaxTxBytes
resp.Txs = h.injectAndResize(resp.Txs, extInfoBz,
req.MaxTxBytes+int64(len(extInfoBz)))
```

If `injectAndResize` is called on said value, because of the checks in `if` statement within that function, it will inject VE tx as the only tx in `returnedTxs` struct. Execution of `PrepareProposalHandler` finishes as if nothing went wrong and the proposal is prepared with only VE transaction.

This kind of block will cause `ProcessProposalHandler` to crash further in the future (code).

Problem Scenarios

Any proposal that causes dYdX `PrepareProposalHandler` to crash will produce a block with only the VotingExtension transaction included. This leads prepareProposal round to pass, but that doesn't matter because the block will be rejected when processing the proposal.

Recommendation

Our recommendation is to propagate the error from dYdX `PrepareProposalHandler` to the Slinky's `PrepareProposalHandler` wrapper.

Inconsistent error logging between processProposal and prepareProposal handlers

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	**IMPLEMENTATION**
Severity	**INFORMATIONAL**
Impact	**NONE**
Exploitability	**NONE**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- slinky-mev/slinky/abci/proposals/proposals.go
- dydxprotocol/v4-chain/protocol/app/process/process_proposal.go
- dydxprotocol/v4-chain/protocol/app/prepare/prepare_proposal.go

Description

This particular threat revolves around error logging and is closely connected with the threat Inadequate error propagation in prepareProposal .

The first inconsistency that concerned us revolved around different ways to log errors.

Errors that have occurred within `PrepareProposalHandler` (code) are being logged in the way shown in the snippet below.

```
ctx.Logger().Error(fmt.Sprintf("AddOtherTxs error: %v", err))
```

While within `ProcessProposalHandler` errors are logged by calling the wrapper function `LogErrorWithOptionalContext` (here).

```
error_lib.LogErrorWithOptionalContext(ctx, "DecodeProcessProposalTxs failed", err)
```

`LogErrorWithOptionalContext` implementation can seen in the snippet below (code).

```
func LogErrorWithOptionalContext(
    ctx sdk.Context,
    msg string,
    err error,
) {
    logger := ctx.Logger()
    var logContextualizer LogContextualizer
    if ok := errors.As(err, &logContextualizer); ok {
        logger = logContextualizer.AddLoggingContext(logger)
        // Log the original error.
        err = logContextualizer.Unwrap()
    }

    logger.Error(msg, "error", err)
}
```

Inconsistencies within code are never a good practice. If there is already an implemented wrapper function, it should be used instead of calling functions raw, to keep the code consistent.

The concern we had regarding error propagation mentioned in Inadequate error propagation in prepareProposal also impacts the missing log from the line. Besides that missing log, logs on lines (ref1, ref2) might give a deceiving image that everything is okay during monitoring, when in fact dYdX's `PreaprePrоposalHandler` returned an error.

That kind of log is also missing from Slinky's `ProcessProposalHandler` implementation, which again leads to inconsistency.

The general concern we had is that **no errors will be returned to CosmosSDK**. CosmosSDK expects from PrepareProposal and ProcessProposal response and an error. If an error occurs, it will be logged and returned as `nil` to CometBFT in order not to crash the CometBFT node. The code snippets showing said behavior is shown in the snippets below (code ref1, ref2).

```
resp, err = app.processProposal(app.processProposalState.Context(), req)
if err != nil {
    app.logger.Error("failed to process proposal", "height", req.Height, "time",
req.Time, "hash", fmt.Sprintf("%X", req.Hash), "err", err)
    return &abci.ResponseProcessProposal{Status:
abci.ResponseProcessProposal_REJECT}, nil
}
```

```
resp, err = app.prepareProposal(app.prepareProposalState.Context(), req)
if err != nil {
    app.logger.Error("failed to prepare proposal", "height", req.Height, "time",
req.Time, "err", err)
    return &abci.ResponsePrepareProposal{Txs: req.Txs}, nil
}
```

If `nil` is returned to CosmosSDK, logs (ref1, ref2) will be missing completely and thus the logging will be incomplete.

Problem Scenarios

Logs will be inconsistent and incomplete.

Recommendation

Keep the code consistent. Use implemented wrappers and stick to certain logging standards.

Code improvements for Slinky related code changes

Project	dYdX 2024 Q2: Slinky integration, Reduce Only
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- dydxprotocol/v4-chain/protocol/x/prices/keeper/update_price.go
- dydxprotocol/v4-chain/protocol/daemons/server/types/pricefeed/exchange_to_price.go
- dydxprotocol/v4-chain/protocol/x/prices/keeper/slinky_adapter.go
- dydxprotocol/v4-chain/protocol/app/prepare/prepare_proposal.go
- dydxprotocol/v4-chain//protocol/daemons/slinky/client/price_fetcher.go

Description

Here is the list of aesthetic and minor code improvements and issues around logging found during the code inspection of the audited scope. 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.

- After removing smoothing prices, there is a bunch of generally unclear and suboptimal code that could be rewritten here and here. It could be thought over again. If there is an intention to at some point implement some other reasons for not proposing a price, it's worth keeping the code, otherwise, refactoring is advised for general clarity.
- Inconsistency between returning `EmptyResponse` and `abci.ResponsePrepareProposal{Txs: [][]byte{}}` within PrepareProposalHandler (code ref1, ref2). Stick to the best practices. `EmptyResponse` is of the same type as `abci.ResponsePrepareProposal{Txs: [][]byte{}}`, so returning only one of them would keep the code cleaner
- There is no need to create a variable here. Pass an already existing one to the function.
- There is no need to create a variable here. Assign `types.NewPriceTimestamp()` to `etp.exchangeToPriceTimestamp[exchangeId]`.
- There will be no distinction between the price fetcher not working right or not working at all. In both cases, the error will **not** be logged and will be just returned back to the Fetcher process. There will be no distinct logs that would suggest that is the case that failed. Both kinds of error will just log this (shown in the snippet below). If looked more carefully, it can be seen that the error will be logged in the following code snippet.

```
c.logger.Error("Failed to run fetch prices for slinky daemon", "error", err)
c.ReportFailure(errors.Wrap(err, "failed to run PriceFetcher for slinky daemon")
)
```

We would suggest using logs to indicate which kind of failure has occurred here and here.

- The same problem as the one mentioned above could be found in the `FetchIdMappings` function.
- Price fetching executed on ticks. If there were no valid market price updates the `FetchPrices` function will return `nil`. This will prompt the `RunPriceFetcher` function to assume that there were no errors with fetching prices and call `c.ReportSuccess()`. This is rather misleading. Obviously, there was an error and hence there were no price updates. Consider changing this to return the error back to the `RunPriceFetcher` function.

Problem Scenarios

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

Recommendation

As explained in the Description section.

Missing automatic detection of validators not reporting certain CP price updates

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	**PROTOCOL**
Severity	**MEDIUM**
Impact	**HIGH**
Exploitability	**LOW**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- price updates injection in VE per validator design

Description

With current design for injection of price updates per validator, validators could skip providing update of certain currency pair and make dYdX protocol work with stale prices without being slashed. There is no mechanism implemented with purpose of incentivize honest behavior and discourage malicious actions that could harm the integrity and security of the blockchain network.

Problem Scenarios

If several validators join in skipping the price update injection for a currency pair it is possible that the price update will not reach 2/3 +1 threshold needed (especially considering current dYdX validator power distribution). In this case, dYdX could continue trading with stale prices.

Recommendation

There are several potential solutions:

- dYdX deployers or Skip Slinky team should set up monitoring in place detecting validators skipping providing certain currency pairs. Perhaps these monitoring is already agreed with Skip team and in place.
- dYdX could include Skip's slashing mechanisms (x/sla) in place once implemented.

Telemetry monitoring and Indexer events created when updating prices in extend vote handler

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /dydxprotocol/v4-chain/protocol/app/vote_extensions/extend_vote_handler.go
- /dydxprotocol/v4-chain/protocol/x/prices/keeper/market_price.go

Description

In dYdX `ExtendVoteHandler` prices are updated (currently, for the vote extension context) with UpdateMarketPrices function, in order for `SlinkyExtendVoteHandler` to inject correct price updates calculated against the latest prices.
The actual price update for the app state will be performed during the `DeliverTx` of the `MsgUpdateMarketPrices` created with prepare proposal for the current block.

Problem Scenarios

When updating the prices from dYdX `ExtendVoteHandler`:

- telemetry monitors new updated prices info - code
- indexer events are generated - code

This way telemetry and indexer could contain misleading information, in best case if the current block is voted for there will be duplicate information for the same price update.

Recommendation

Do not emit a telemetry metric or generate indexer event in case of `UpdateMarketPrices` being executed from `ExtendVoteHandler`.

Unclear handling in cases when GetValidMarketPriceUpdates returns no valid price updates to be injected to VE

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	IMPLEMENTATION
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Involved artifacts

- /skip-mev/slinky/abci/ve/vote_extension.go
- /dydxprotocol/v4-chain/protocol/app/vote_extensions/oracle_client.go

Description

Current implementation is confusing from the aspects of logging the actual vote extensions content in cases when there were no valid prices collected with `GetValidMarketPriceUpdates` function:

- Updates would be initialized as empty slice and returned from the function as such.
- Prices function expects the response to be nil in case that there was no valid prices. Only in this case, the appropriate log would be logged and nil value would be returned as indicator for the wrapped Slinky `VoteExtensionHandler` to create empty VE. This is actually the same result - empty vote extensions would be injected in both cases, but current implementation will contain misleading logs.
- Timestamp gets updated, but it's not used anywhere in the Slinky `VoteExtensionHandler`. Currently, it's not a problem, but it's worth questioning whether the timestamp should be updated if there are no price updates.
- Slinky also expects the response to be nil in order to log the appropriate message.

Problem Scenarios

Logs could be misleading. Questionable need for updating the `Timestamp` in `QueryPricesResponse` in cases of no valid price updates.

Recommendation

Follow the expected nil values returned from `GetValidMarketPriceUpdates` and `Prices` to the functions above in the call stack (Slinky `VoteExtensionHandler`) in order to have the appropriate logs. If deemed necessary by the dYdX team, distinguish between the logs for two scenarios: when the oracle fails to provide any price updates and when it provides invalid prices in the dYdX code.

Slinky's ValidateExtendedCommitAgainstLastCommit function contains invalid checks for non-absent votes

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	**IMPLEMENTATION**
Severity	**LOW**
Impact	**LOW**
Exploitability	**MEDIUM**
Status	**ACKNOWLEDGED**
Issue	

Involved artifacts

- /skip-mev/slinky/abci/ve/utils.go
- /skip-mev/slinky/blob/abci/proposals/validate.go

Description

During **prepare proposal**, proposer performs:

1. PruneAndValidateExtendedCommitInfo - filters out all invalid VE - ones failing the validateVoteExtension execution. Filtering of invalid VEs is done with marking those VE are marked as: (code)

   ```
   // failed to validate this vote-extension, mark it as absent in the original
   commit
   vote.BlockIdFlag = cometproto.BlockIDFlagAbsent
   vote.ExtensionSignature = nil
   vote.VoteExtension = nil
   extendedCommitInfo.Votes[i] = vote
   ```

   So, the content of the vote would be replaced as if the vote extension was not injected.
2. Validation of **pruned** VEs in extended commit with `validateVoteExtensionsFn` - where ValidateVoteExtensions is executed. This function compares:
   a. Extended commit and last commit and detects mismatches - ValidateExtendedCommitAgainstLastCommit.
   In this function, there is one additional check performed where the intention was to check mismatches in extend commit and last commit for the `BlockIdFlag` values for the non-absent votes (`vote.BlockIdFlag != cmtproto.BlockIDFlagAbsent`) The absent votes could be marked via pruning in the prepare proposal function as explained in 1. - code:

```
// only check non-absent votes (these could have been modified via pruning
in prepare proposal)
if !(vote.BlockIdFlag == cmtproto.BlockIDFlagAbsent &&
len(vote.VoteExtension) == 0 && len(vote.ExtensionSignature) == 0) {
    if int32(vote.BlockIdFlag) != int32(lcVote.GetBlockIDFlag()) {
        return fmt.Errorf("mismatched block ID flag between extended
commit vote %d and last proposed commit %d", int32(vote.BlockIdFlag),
int32(lcVote.GetBlockIDFlag()))
    }
}
```

This check contains a fairly complex condition, which as we concluded does not perform correct validation. The consequences are explained in the Problem Scenarios section.

b. Validates the filtered VE with ensuring extensions checks, calculates the "vote-with-VE" power threshold (`vote.BlockIdFlag == cmtproto.BlockIDFlagCommit`) and returns error if any of the signatures is invalid or signatures found are less than 2/3+1 received.

During **process proposal**, all the nodes perform:

Checks over the extended commit info - ValidateExtendedCommitInfo in order to confirm that extended commit was not altered by the proposer - except for the pruning. It is necessary to perform the same validation steps as in prepare proposal phase, so:

1. validation of **received** VEs in extended commit with `validateVoteExtensionsFn` - where ValidateVoteExtensions is executed (ValidateExtendedCommitAgainstLastCommit, calculation of vote-with-VE power theshold …
2. each vote's vote extension is validated with validateVoteExtension.

Problem Scenarios

There are two possible issues:

1. **Validation is not performed correctly:** The conditions for checking the non absent votes, result in not validating absent votes for empty `VoteExtensions` and `ExtensionSignature`.
   Example that will not be validated with the current implementation:

> *vote: BlockIDFlagAbsent, ext = existingVE, sig = existingSig*
> *lcvote: BlockIDFlagAbsent, ext = nil, sig = nil*

here, we have an invalid vote with VE and signature existing, but this will not be catched with the validation peformed in the ValidateExtendedCommitAgainstLastCommit.

2. **Potential Exploitation of BlockIdFlag in Oracle Transactions In Proposal:**
   Malicious proposer could:
   a. alter the: `VoteExtension`, `ExtensionSignature` data for last commit votes with `blockIdFlag` marked `BlockIDFlagAbsent`
   b. when this altered extended commit reaches process proposal, the ValidateExtendedCommitInfo validation would be executed:
      i. ValidateExtendedCommitAgainstLastCommit - check will not return mismatching error here since:
      `extVote.BlockIdFlag` = `BlockIDFlagAbsent`, `VoteExtension` is != nil, `ExtensionSignature` != nil and `extVote.BlockIdFlag` is equal to `lcVote.BlockIdFlag`.
      ii. `extCommit.Votes` could actually contain vote with `BlockIdFlag` equal to `BlockIDFlagAbsent`, so the

`totalVP += vote.Validator.Power` would include power of the votes that were initially without attached vote extensions and extension signature. - code

iii. since code would make the loop continue with next vote:

```
// Only check + include power if the vote is a commit vote. There
must be super-majority, otherwise the
// previous block (the block vote is for) could not have been
committed.
if vote.BlockIdFlag != cmtproto.BlockIDFlagCommit {
    continue
}
```

no signature verification for the vote extension would be executed.

iv. if power threshold has been reached, extended commit is considered valid (containing non verified VE injected by a malicious proposer).

v. next is the validation of each vote extension with code

The consequence is:

1. Malicious proposer could inject vote extension that can not be validated, making the process proposal for the proposed block to fail, leading to new round and new proposer proposing the block.
The malicious proposer activity results in the rejection of the proposed block, but one step later. Block could be rejected during the ValidateExtendedCommitAgainstLastCommit function, this causes all nodes to expend computational resources in vain, thereby delaying the creation of the next block.

2. Injected data being considered in the median weighted calculations of final price updates - this should not be an issue, since the validator power should be bigger than the 50% of total voting power in order to skew the final price and this is not possible but it is not correct to work with maliciously injected data. They would also increase the block size for maximum of maxCp price updates.

Recommendation

Change the problematic check to something like:

```
if vote.BlockIdFlag == cmtproto.BlockIDFlagAbsent {
  if len(vote.VoteExtension) != 0 || len(vote.ExtensionSignature) != 0) {
    return fmt.Errorf("non empty vote extension or signature for BlockIDFlagAbsent")
  }
  return nil
}

// we know now that vote.BlockIdFlag != cmtproto.BlockIDFlagAbsent
if int32(vote.BlockIdFlag) != int32(lcVote.GetBlockIDFlag()) {
    return fmt.Errorf("mismatched block ID flag between extended commit vote %d and
last proposed commit %d", int32(vote.BlockIdFlag), int32(lcVote.GetBlockIDFlag()))
}
```

The suggested logic will confirm that:

- in cases of `BlockIDFlagAbsent` we need to have empty VE and `ExtensionSignature`, due to pruning.
- remaining votes with non `BlockIDFlagAbsent` will be compared with last commit vote `BlockIdFlag`.

Analysis of ABCI++ requirements

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	PROTOCOL
Severity	LOW
Impact	LOW
Exploitability	LOW
Status	ACKNOWLEDGED
Issue	

Description

We have split this analysis into two parts:

- Confirming all requirements listed in the CometBFT documentation regarding **Vote Extensions** are fulfilled
- Re-checking **R1**, **R2**, **R3**, **R4** and **R5** regarding **Prepare** and **Process proposals**

Point #1 - Confirm all requirements listed in the CometBFT documentation regarding Vote Extensions are fulfilled

- **R6: On those vote extensions prepared by the correct node, any other correct node `VerifyVoteExtension` will return true as well.** ✅
  - During the threat inspection, we have identified a few points of failure for this particular requirement. This requirement expands to verification being the same while creating VE as well as during its verification.
  - While inspecting the `VerifyVoteExtension` function from the Slinky codebase, we can see several points where VE might not be verified.
    - Decoding the VE, which will not fail for correct nodes
    - Validating VE
  - Validating has a few points of failure in the function `ValidateOracleVoteExtension` used in `VerifyVoteExtension` and it is crucial that failure does not happen here for any reason. Potential failures include Sidecar providing number of currency pairs that is higher than the maximum possible or that the prices bytes are longer than they should be.

More currency pairs than possible

`maxNumCP` is the value that is defined by the following code snippet here. `GetMaxNumCP` calls `GetPrevBlockCPCounter` from `oracleKeeper` (here). Because in the `app.go`, `oracleKeeper` is defined with `PriceKeeper`'s implementation by dYdX. That means that the previously mentioned `GetPrevBlockCPCounter` calls implementation from dYdX (here). This means that if there is `x` number of currency pairs on the chain, this check in the `VerifyVoteExtensionHandler` will always pass. There are **two edge** scenarios. Either there will be added new market or the old one will be closed.

Scenario A: A new market is added

> This would increase the maximum number of currency pairs, but as Vote Extensions for that block would already be included there would be no collision. In the next block, the maximum number of currency pairs would be increased and thus VEs would work with the new maximum number and, again, the check would pass.

Scenario B: An old market is removed

> Closing the market would remove it from the state, but as the VEs for that particular block and the check passed with the previous number, this is okay. When VEs are injected for the next block, they would take the current number of currency pairs and the check will pass again.

The currency pair is bigger than it should be

This revolves around node packing currency pair that is bigger than expected. The maximum price size is 33 bytes. This is part of the Slinky code, and it is supposed to be that size. Not prone to manipulation.

- **R7: `VerifyVoteExtension` is a deterministic function of the current state, the vote extension received, and the prepared proposal that the extension refers to** ✅
  - For the same block ID and the current application state, all nodes should either verify or fail verification of the vote extension. The code should be deterministic and provide the same output if the block ID from the Prepare proposal and the current state do not change.
- **R8: For any two correct processes the output of their `VerifyVoteExtension` on an arbitrary vote extension is equivalent.** ✅
  - There isn't any nondeterminism present within `VerifyVoteExtension`. `VerifyVoteExtension` revolves only around Vote Extensions.
- **R9: all calls to `PrepareProposal`, `ProcessProposal`, `ExtendVote`, and `VerifyVoteExtension` at height *h* do not modify $s_{p,h-1}$.** ✅
  - As there will only be price updates that will affect future blocks, there will be no state modification for the previous blocks.
- **R10: for any correct process *p*, and any vote extension *e* that *p* received at height *h*, the computation of $s_{p,h}$ does not depend on *e*.** ✅
  - `FinalizeBlock` doesn't depend on injected VE on the current height.

Point #2 - re-checking R1, R2, R3, R4 and R5 regarding Prepare and Process proposals.

- **R1: prepare proposal does not take too much time** ✅
  - `for` loops are within limits, so there will not be unexpected execution time that could potentially be too long.
- **R2: prepare proposal does not take too much space** ✅
  - Slinky prepare proposal handler and dYdX prepare proposal handlers both ensure that `ResponsePrepareProposal.Txs` returned from the proposal handlers will not be larger than `RequestPrepareProposal.max_tx_bytes` - Slinky proposal code: ref1 (reserve bytes for oracle injected tx), ref2 (propagate reduced `maxTxBytes` to dYdX prepare proposal handler), ref3 (make sure the returned size of `ResponsePrepareProposal.Txs` respects `req.MaxTxBytes`; dYdX proposal handler code: ref1 (creates prepare proposal struct containing injected txs and `maxBytes` and `usedBytes` - ensuring the upper limit is not passed) , ref2 (making sure that injected and added txs to the proposal struct do not exceed the upper limit).

- **R3: On those blocks prepared by a correct node, any other correct node's `ProcessProposal` will return true as well.** ❌
    - For this requirement to be respected, every block prepared by the correct node needs to be accepted during the Process proposal phase. That means that, if the proposer is honest, there can not be any reason for the Process proposal to return `cometabci.ResponseProcessProposal_REJECT`. However, if the size of `extInfoBz` is larger or equal to `req.MaxTxBytes` (here) then it will return `&cometabci.ResponsePrepareProposal{Txs: make([][]byte, 0)}, err`. There is a possibility that the proposer was honest, and just using VEs injected from the last block. If we blindly rely on the R3, there has to be a way to handle this scenario in the Process proposal, and not REJECT the proposal. However, that is not the case. If `processProposalHandler` receives an empty array of transactions, Slinky's `processProposalHandler` will REJECT the proposal and return `&cometabci.ResponseProcessProposal{Status: cometabci.ResponseProcessProposal_REJECT}, err`.
- **R4: `ProcessProposal` is a deterministic function of the application's current state and the proposed block.** ✅
- **R5: For any two correct processes, the output of their `ProcessProposal` on an arbitrary block is equivalent.** ✅

With current **VE** utilization and deterministic decisions of the final price update **R4**, and **R5** are considered fulfilled. There aren't any nondeterministic operations and malicious proposals will not be accepted.

Recommendation

ABCI++ requirements must be followed so therefore implementing changes that respect requirement R3 is something we recommend.

Reconsider application panicking instead of working with corrupted data

Project	**dYdX 2024 Q2: Slinky integration, Reduce Only**
Type	PROTOCOL
Severity	INFORMATIONAL
Impact	NONE
Exploitability	NONE
Status	ACKNOWLEDGED
Issue	

Description

We have noticed a pattern of propagating errors from Slinky to Cosmos SDK handlers or logging errors and returning nil instead of error from dYdX Prepare and Process proposals handlers to the Slinky wrappers. Those errors are returned in various cases, such as `Encoding` the `ExtendCommitInfo`, `ExtendCommitInfo` size being bigger than `req.MaxTxBytes`, pruning VEs etc. However, after consultations with CometBFT team, we have reached a stance that this kind of implementation could be reconsidered.

Problem Scenarios

In the scenarios that the development team is sure will not happen naturally and would cause data to be corrupted further in the chain, it is generally advised to panic the node. Panicking the node will leave a stack trace that would be used for easier debugging. Sometimes it could be better to halt the chain with the panic and fix the problem quickly using the stack trace left behind because of panic than to keep going with the corrupt data that could ruin the chain's reputation in the future.

Recommendation

We advise to reconsider causing `panic` instead of returning an error or nil in the cases that are not likely to happen but are erroneous.

Appendix: Vulnerability Classification

For classifying vulnerabilities identified in the findings of this report, we employ the simplified version of Common Vulnerability Scoring System (CVSS) v3.1, which is an industry standard vulnerability metric. For each identified vulnerability we assess the scores from the *Base Metric Group*, the Impact score, and the Exploitability score. The *Exploitability score* reflects the ease and technical means by which the vulnerability can be exploited. That is, it represents characteristics of the *thing that is vulnerable*, which we refer to formally as the *vulnerable component*. The *Impact score* reflects the direct consequence of a successful exploit, and represents the consequence to the *thing that suffers the impact*, which we refer to formally as the *impacted component*. In order to ease score understanding, we employ CVSS Qualitative Severity Rating Scale, and abstract numerical scores into the textual representation; we construct the final *Severity score* based on the combination of the Impact and Exploitability sub-scores.

As blockchains are a fast evolving field, we evaluate the scores not only for the present state of the system, but also for the state that deems achievable within 1 year of projected system evolution. E.g., if at present the system interacts with 1-2 other blockchains, but plans to expand interaction to 10-20 within the next year, we evaluate the impact, exploitability, and severity scores wrt. the latter state, in order to give the system designers better understanding of the vulnerabilities that need to be addressed in the near future.

Impact Score

The Impact score captures the effects of a successfully exploited vulnerability on the component that suffers the worst outcome that is most directly and predictably associated with the attack.

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

Exploitability Score

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

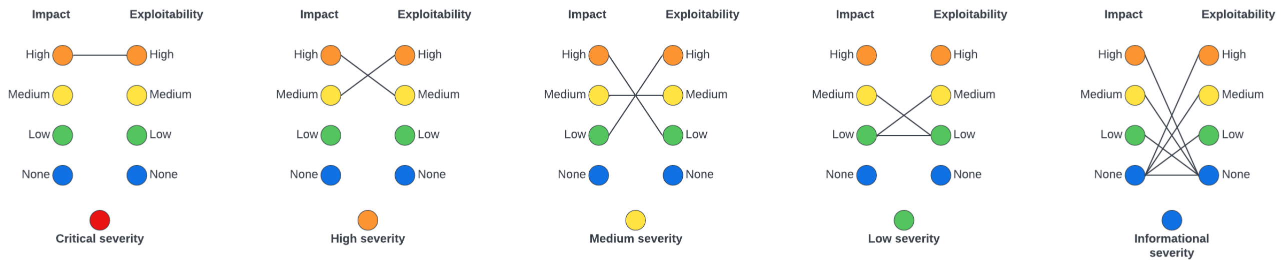
- *Actors* can be any entity that interacts with the system: other blockchains, system users, validators, relayers, but also uncontrollable phenomena (e.g. network delays or partitions).
- *Actions* can be

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

Exploitability Score	Examples
🟠 **High**	illegitimate actions taken by a small group of actors; possibly coordinated with legitimate actions taken by a medium group of actors
🟡 **Medium**	illegitimate actions taken by a medium group of actors; possibly coordinated with legitimate actions taken by a large group of actors
🟢 **Low**	illegitimate actions taken by a large group of actors; possibly coordinated with legitimate actions taken by all actors
🔵 **None**	illegitimate actions taken in a coordinated fashion by all actors

Severity Score

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



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

Severity Score	Examples
🔴 **Critical**	Halting of chain via a submission of a specially crafted transaction

Severity Score	**Examples**
🟠 **High**	Permanent loss of user funds via a combination of submitting a specially crafted transaction with delaying of certain messages by a large portion of relayers
🟡 **Medium**	Substantial unexpected delays in processing user requests via a combination of delaying of certain messages by a large group of relayers with coordinated withdrawal of funds by a large group of users
🟢 **Low**	2x increase in node computational requirements via coordinated withdrawal of all user tokens
🔵 **Informational**	Code inefficiencies; bad code practices; lack/incompleteness of tests; lack/incompleteness of documentation; any exploit for which a coordinated illegitimate action of all actors is necessary

Disclaimer