

Security Audit Report

Ojo Node, Price Feeder and Smart Contracts

Delivered: May 1st, 2023



Prepared for Ojo by



Table of Contents

Executive Summary

Goal

Scope

Platform and Contract Features Description

Methodology

Disclaimer

Contract Description and Invariants

Findings

A01: (Price feed contract) No error propagation in bulk operations

A02: (Price feeder) Potential risk of running pingTicker more times than desired

A03: (Ojo node) Denial of service of the node (EndBlocker failure)

A04: (Price feeder) Enforcement of time interval for last candles in TVWAP takes only in consideration the comparison between the candle timestamp and the local clock time minus 5 minutes

A05: (Ojo node) Price manipulation by overpowering validators

A06: (Ojo node) Variable representing the number of maximum valid votes can be lower than the actual value

A07: (Price feeder) Possibility for denial of service in the price feeder

A08: (Price feeder) Filtering of candles delays modification of exchange rates for assets

Informative Findings

B01: (Price feed contract) RELAYERS map values are not used

B02: (Price feed contract) Median data is a vector, which implies that the current median is not stored



B03: (Price feed contract) Relayers are fully trusted by the contract and no validations are performed over relayed rates

B04: (Price feed contract) Median status is ignored when executing `execute_force_relay_historical_median`

B05: (Price feed contract) There are no prevention mechanisms to fully stop the price relaying process

B06: (Price feed contract) No checks on the received resolve time

B07: Outdated and vulnerable dependencies

B08: (Price feeder) Potential cross-site scripting (XSS) vulnerability

B09: (Ojo node) Add the voting power used for converging into an exchange rate

B10: (Ojo node) Median endpoint does not provide a timestamp or block number

B11: (Ojo node) No validations on possible division by zero

B12: (Price feeder) Possibility for price manipulation in the price feeder project

Executive Summary

Ojo engaged [Runtime Verification Inc.](#) to conduct a security audit of their smart contracts. The objective was to review the platform's business logic and implementations in Rust, as well as in Go, and identify any issues that could potentially cause the system to malfunction or to be exploited.

The audit was conducted over the course of 8 calendar weeks (February 28, 2023 through April 14, 2023) focused on analyzing the security of the code related to the contracts of the Ojo platform, which enables users to query information about asset exchange rates. The platform also provides the necessary mechanics to run nodes, validators and price providers for tokens supported on the platform.

The audit led to identifying issues of potential severity for the protocol's health, which have been identified as follows:

- Functional correctness: [A02](#), [A04](#), [A06](#)
- Node/System malfunctions: [A03](#), [A08](#)
- Exchange rate manipulation: [A05](#), [A08](#), [B12](#)

In addition, several informative findings and general recommendations also have been made, including:

- Input validations: [A03](#), [B03](#), [B06](#), [B11](#)
- Best practices: [A01](#), [B01](#), [B02](#), [B04](#), [B06](#), [B07](#), [B08](#), [B10](#), [B12](#)
- Code optimization related particularities: [B01](#)
- Blockchain related particularities: [A05](#), [B03](#), [B05](#), [B09](#)

All of the potentially critical issues have been addressed, while a substantial amount of the informative findings and general recommendations have been incorporated to the contracts' code. All of the remaining findings have been acknowledged by the client and deemed non-threatening to the integrity of the platform, when not intended by design.

Goal

The goal of the audit is twofold:

1. Review the high-level business logic (protocol design) of Ojo's system based on the provided documentation;
2. Review the low-level implementation of the system given as smart contracts in Rust;
3. Review the low-level implementation of the system for the individual Go projects (node and price feeder);
4. Analyze the integration between the multiple modules in scope of the engagement.

The audit focuses on trying to identify issues in the system's logic and its implementation that could potentially render the system vulnerable to attacks or cause it to malfunction. Furthermore, the audit highlights informative findings that could be used to improve safety and efficiency of the implementation.

Platform and Contract Features Description

Aiming to provide oracle services to the Cosmos ecosystem, the Ojo network implements a L1 oracle chain focused on delivering precise pricing of assets using a decentralized, secure strategy for asset price collection.

Through means of validators that fetch asset exchange rates from multiple sources and submit votes to the blockchain nodes, the Ojo network is capable of using a decentralized strategy to collect the values that are going to be distributed and provided at their nodes and contract. This effectively eliminates the need of trust on a centralized source of prices, enhancing security and providing secure and efficient features for blockchain solutions on Cosmos.

All the necessary software to participate in the consensus protocol of this chain is provided by the Ojo team.

Scope

The scope of the audit is limited to the code contained in repositories of three different projects provided by the client. These projects are:

1. **Ojo node (v. 0.1.2)**: implements the functionalities related to the operation of the Ojo node and the core of the oracle protocol, handling the consensus on the voting process for electing exchange rates in interactions with validators;
2. **Price feeder (v. 0.1.1)**: implements the necessary tooling for fetching information about the exchange rate of assets, preprocessing the fetched data, including the output of this process into messages and signing for submission as votes for the Ojo node;
3. **Contract & Relay (v. 0.1.1)**: implements the necessary functionalities for providing the exchange rates obtained in the Ojo node through means of a smart contract, which also includes the mechanics for forwarding exchange rate data to said smart contract.

The comments provided in the code, and a general description of the project, including samples of tests used for interacting with the platform, and also online documentation provided by the client were used as reference material.

The audit is limited in scope to the artifacts listed above. Off-chain, auto-generated or client-side portions of the codebase as well as deployment and upgrade scripts are not in the scope of this engagement.

Commits containing the addressment of the findings presented in this report have also been analyzed to ensure the resolvment of potential issues in the protocol.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in our [Disclaimer](#), we have followed the approaches described below to make our audit as thorough as possible.

First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic. To this end, we carefully analyzed all the proposed features of the platform and actors involved in the lifetime of a deployed contract.

Second, we thoroughly reviewed the contract source code to detect any unexpected (and possibly exploitable) behaviors. To facilitate our understanding of the platform's behavior, several higher-level representations of the Rust and Go codebase were created, where the most comprehensive were:

1. A high-level relationship model to describe and correlate the operations performed by the individual contracts of the Ojo platform, which could be used for the analysis of the separate entities of the protocol,
2. Modeled sequences of mathematical operations as equations and, considering the limitations of size and types of variables in the COSMWASM VM and the utilized go modules, checking if all desired properties hold for any possible input value.

This approach enabled us to systematically check consistency between the logic and the provided Rust and Go implementation of the contracts and modules.

Finally, we performed rounds of internal discussion with security experts over the code and platform design with the aid of static analysis tools and generated call graphs, aiming to verify possible exploitation vectors and to identify improvements for the analyzed contracts.

Additionally, given the nascent Cosmos development and auditing community, we reviewed [this list](#) of known Ethereum security vulnerabilities and attack vectors and checked whether they apply to the smart contracts and scripts; if they apply, we checked whether the code is vulnerable to them.



Disclaimer

This report does not constitute legal or investment advice. You understand and agree that this report relates to new and emerging technologies and that there are significant risks inherent in using such technologies that cannot be completely protected against. While this report has been prepared based on data and information that has been provided by you or is otherwise publicly available, there are likely additional unknown risks which otherwise exist. This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system. This report is for informational purposes only and is provided on an "as-is" basis and you acknowledge and agree that you are making use of this report and the information contained herein at your own risk. The preparers of this report make no representations or warranties of any kind, either express or implied, regarding the information in or the use of this report and shall not be liable to you or any third parties for any acts or omissions undertaken by you or any third parties based on the information contained herein.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk.

Finally, the possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Contract Description and Invariants

Ojo is a decentralized oracle network built on the Cosmos SDK. As such, it extends the base implementation from the Cosmos SDK with modules that perform specific functions.

This section describes the module at a high-level and which invariants of its state we expect it to always respect at the end of a contract interaction or a block. In this document, we will use "price" for the price feed of exchange rates, which refers to the token's market value in USD.

Parameters and Variables

The Ojo network relies on validators to periodically vote on current exchange rates. During each voting period, the valid voters (neither jailed nor inactive) will reveal their exchange rate votes for all accepted denominations in USD alongside proof that they had submitted the votes into ballots for each denomination. For each *denom*, a voter's ballot consisting of $\{ExchangeRate: rate, Denom: denom\}$ is recorded. A ballot for the exchange rate must have at least 50% of total voting power. The oracle price for a *denom* is calculated based on the median in ballots. After the votes are tallied, the oracle module will adjust the miss counters for validators, properly penalize validators who have missed more than the penalty threshold, and distribute rewards to ballot winners.

The Oracle module provides the Ojo blockchain with an up-to-date and accurate price of multiple currencies for the leverage module. To facilitate the pricing activities of the oracle, the module keeps track of the important statistics as state variables such as the exchange rate, the miss counter, etc.

We will introduce these parameters and state variables in the following, along with an abstract model of the protocol, to facilitate the reasoning of the oracle logic.

```
type Params struct {
    VotePeriod uint64
    VoteThreshold sdk.dec
    RewardBands []RewardBand
    RewardDistributionWindow uint64
    AcceptList []Denom
```

```

SlashFraction sdk.dec
SlashWindow uint64
MinValidPerWindow sdk.dec
MandatoryList []Denom
HistoricStampPeriod uint64
MedianStampPeriod uint64
MaximumPriceStamps uint64
MaximumMedianStamps uint64
}

```

ExchangeRate records all current active exchange rates are purged from the store.

$$ExchangeRate(BaseDenom) = \frac{ExchangeRate(SymbolDenom)}{10^{Exponent}}$$

FeederDelegation records the account address of the validator operator, which delegated Oracle vote rights.

MissCounter is the variable that tracks the number of vote periods a validator operator missed during the current slashing window. Increase the miss counter if the voter misses a vote at the end of the block. Reset miss counters for all validators at the last block of the slashing window.

AggregateExchangeRatePrevote records a validator aggregated prevote for all denominations for the current voting period.

AggregateExchangeRateVote is a record type of a validator's aggregated prevote for all denominations for the current voting period.

Operations

The oracle consists of n oracles $O = \{O_1, \dots, O_i\}$, which are also called nodes. Each oracle O_i makes time-varying observations of a value such as a price. This is modeled as a value they can read at any time.

Calculate price

At the end of every block, the module O_i checks whether it's the last block of the voting period. If it is, it processes the voting procedure and runs the *CalcPrices* function. At the

the start of each voting period, the oracle queries the current list of votes for the validator for each denomination listed in the *MandatoryList*. If there was a previous prevote for the asset, the oracle checks if the salt and related information is in memory. If so, it will then submit a single transaction containing multiple votes, one for each of the *MandatoryList* denominations.

This function first builds a claim map over all active validators and then performs the operations below.

Vote Target

Denominations that require votes should be on the accept list. *voteTargetDenoms* contains all *denom* to vote on. After deciding the vote target of denominations, clear all existing *ExchangeRate* of the O_i from the store.

```
voteTargetDenoms := make([]string, 0)
for _, v := range params.AcceptList {
    voteTargetDenoms = append(voteTargetDenoms, v.BaseDenom)
}

k.ClearExchangeRates(ctx)
```

Organize Ballot

In a voting period, the function *OrganizeBallotByDenom* collects all oracle votes categorized by the votes' *denom* parameter. *aggregateHandler* collects aggregate votes and organizes ballots for all the valid voters.

```
func (k Keeper) OrganizeBallotByDenom(
    ctx sdk.Context,
    validatorClaimMap map[string]types.Claim,
) []types.BallotDenom {
    votes := map[string]types.ExchangeRateBallot{}

    // collect aggregate votes
    aggregateHandler := func(voterAddr sdk.ValAddress,
                             vote types.AggregateExchangeRateVote)
                             bool {
        // organize ballot only for the active validators
        claim, ok := validatorClaimMap[vote.Voter]
```

```

    if ok {
        power := claim.Power

        for _, decCoin := range vote.ExchangeRates {
            tmpPower := power
            votes[decCoin.Denom] = append(
                votes[decCoin.Denom],
                types.NewVoteForTally(decCoin.Amount,
                                    decCoin.Denom,
                                    voterAddr,
                                    tmpPower),
            )
        }

        return false
    }

    k.IterateAggregateExchangeRateVotes(ctx, aggregateHandler)

    return types.BallotMapToSlice(votes)
}

```

When *claim = ok*, for $(_, decCoin) \in vote.ExchangeRates$, records aggregate votes *votes[denom]*, iterates rate over prevotes in the store, and returns an array of sorted exchange rate ballots. Otherwise, returns an empty array.

```

func (k Keeper) IterateAggregateExchangeRateVotes(
    ctx sdk.Context,
    handler IterateExchangeRateVote,
) {
    store := ctx.KVStore(k.storeKey)

    iter := sdk.KVStorePrefixIterator(store,
types.KeyPrefixAggregateExchangeRateVote)
    defer iter.Close()

    for ; iter.Valid(); iter.Next() {
        voterAddr := sdk.ValAddress(iter.Key()[2:])

        var aggregateVote types.AggregateExchangeRateVote
    }
}

```

```

        k.cdc.MustUnmarshal(iter.Value(), &aggregateVote)

        if handler(voterAddr, aggregateVote) {
            break
        }
    }
}

```

Update Exchange Rate

The function below iterates through ballots and updates exchange rates, dropping if not enough votes have been achieved.

```

for _, ballotDenom := range ballotDenomSlice {
    // Convert ballot power to a percentage to compare with
    // VoteThreshold param
    if sdk.NewDecWithPrec(ballotDenom.Ballot.Power(), 2)
        .LTE(k.VoteThreshold(ctx)) {
        ctx.Logger().Info("Ballot voting power is under vote
                           threshold, dropping ballot", "denom",
                           ballotDenom)

        continue
    }

    [...]
}

```

Assume *ballot* is an organized ballot that has been filtered out inactive or jailed validators, the above function iterates through when $(_, ballotDenom) \in ballot \wedge ballotDenom.Power \leq VoteThreshold$.

Update Miss Counter

```

for _, claim := range claimSlice {
    misses := uint64(voteTargetsLen - int(claim.MandatoryWinCount))
    if misses == 0 {
        continue
    }

    // Increase miss counter
}

```

```

    k.SetMissCounter(ctx,
                     claim.Recipient,
                     k.GetMissCounter(ctx, claim.Recipient)+misses)
}

```

Let *MandatoryWinCount* be the win count of denominations in *MandatoryList*, for each claim of *claimSlice*, if *voteTargetLen* \neq *MandatoryWinCount*, we have $MissCounter[v]_{i+1} = MissCounter[v]_i + len(MandatoryList) - MandatoryWinCount$.

Increase the number of vote periods missed in the current Oracle slash window at the end of the block. Reset miss counters for all validators at the last block of the slashed window.

Ballot Reward

The reward portion is determined by the *missCounter* amount for the voted on exchange rates accrued by each validator in the *SlashWindow*, where the validator with the smallest *missCounter* collects the most reward and the rest are rewarded logarithmically favoring fewer miss counts. Here *v* denotes validator.

```

func (k Keeper) RewardBallotWinners(
    ctx sdk.Context,
    votePeriod int64,
    rewardDistributionWindow int64,
    voteTargets []string,
    ballotWinners []types.Claim,
) {
    if len(ballotWinners) == 0 {
        return
    }

    distributionRatio := sdk.NewDec(votePeriod)
                           .QuoInt64(rewardDistributionWindow)

```

$$DistributionRatio = \frac{votePeriod}{rewardDistributionWindow}$$

```

if rewardPool.IsZero() {
    continue
}

periodRewards = periodRewards.Add(sdk.NewDecCoinFromDec(
    denom,
    sdk.NewDecFromInt(rewardPool.Amount).Mul(distributionRatio),
))

```

Assume *rewardPool* is not empty,
 $periodRewards[denom] = rewardPool[denom].amount * DistributionRatio$, return if
rewardPool doesn't have rewards to give out.

```

maxMissCount := int64(len(voteTargets) *
    (int(k.SlashWindow(ctx) / k.VotePeriod(ctx))))

```

The Maximum possible *MissCounter* is calculated based on the formula below:

$$MaxMissCount = \lfloor number\ of\ target\ votes * \frac{SlashWindow}{VotePeriod} \rfloor$$

```

func CalculateRewardFactor(missCount, m, s int64) float64 {
    logBase := float64(m-s) + 1
    logKey := float64(missCount-s) + 1
    rewardFactor := 1 - (math.Log(logKey) / math.Log(logBase))
    if math.IsNaN(rewardFactor) || math.IsInf(rewardFactor, 0) {
        rewardFactor = 0
    }

    return rewardFactor
}

```

$$RewardFactor[v] = 1 - \log_{m-s+1} MissCounter - s + 1 \text{ where}$$

- $missCounter[v]$ denote the current miss count of validator v .
- m denote $MaxMissCount[v]$, and

- s denote the smallest miss count.

```
rewardCoins, _ := periodRewards.MulDec(rewardDec
                                         .QuoInt64(ballotLength))
                                         .TruncateDecimal()

if rewardCoins.IsZero() {
    continue
}
```

$$Reward[v] = \lfloor PeriodRewards[v] * \frac{RewardFactor[v]}{Number\ of\ BallotWinners[o_i]} \rfloor^1$$

$$= RewardPool[denom].amount * DistributionRatio * \frac{\lfloor 1 - \log_{m-s+1} MissCounter - s + 1 \rfloor}{\lfloor Number\ of\ BallotWinners[o_i] \rfloor}$$

Reward Band

After the votes are tallied, the winners of the ballots are determined with *tally()*. Voters that have managed to vote within a narrowband around the median are rewarded with a portion of the collected seigniorage.

Let σ be the standard deviation of the votes in the ballot. The reward band around the median is set to be $\epsilon = \max(\sigma, \frac{rewardBand}{2})$. All valid (i.e., bonded and non-jailed) validators that submitted an exchange rate vote in the interval $[Median - \epsilon, Median + \epsilon]$ should be included in the set of winners.

```
rewardSpread := median.Mul(rewardBand.QuoInt64(2))
rewardSpread = sdk.MaxDec(rewardSpread, standard
```

rewardSpread is the maximum number of *median* * $\frac{rewardBand}{2}$ and standard deviation.

¹ $\lfloor \cdot \rfloor$ is a floor function, and later the $\lceil \cdot \rceil$ is a ceiling function. Both claim the rounding directions.

Slashing Misbehavior

Validators who fail to submit 50% of valid votes over a time window will be slashed 0.01% of their stake.

$$\text{possible wins} = \frac{\text{slashWindow}}{\text{votePeriod}} * \text{number of accepted assets}$$

```
k.IterateMissCounters(ctx, func(operator sdk.ValAddress,
                                missCounter uint64) bool {

    validVotes := sdk.NewInt(possibleWinsPerSlashWindow
                             - int64(missCounter))

    validVoteRate := sdk.NewDecFromInt(validVotes)
                    .QuoInt64(possibleWinsPerSlashWindow)

    [...]

})
```

$$\text{validVoteRate} = \frac{\text{possible wins} - \text{missCounter}}{\text{PossibleWins}}$$

Prune Price

After completing the voting procedure, delete all historical prices, medians, and median deviations outside the pruning period. The pruning period is determined by the stamp period and the maximum stamps.

Let $isLastBlock_h$ be the last block of $HistoricStampPeriod$, $isLastBlock_m$ be the last block of $MedianStampPeriod$, $prune_h$ be $pruneHistoricPeriod$ and $prune_m$ be $pruneMedianPeriod$. We have

$$prune_h = HistoricStampPeriod * MaximumPriceStamps$$

And

$$prune_m = MedianStampPeriod * MaximumMedianStamps$$

Condition 1: $isLastBlock_h \wedge prune_h < blockHeight$

Condition 2: $isLastBlock_h \wedge lastBlock_m \wedge prune_m < blockHeight$

When Condition 1 is satisfied, delete all historic prices before the block, and if Condition 2 is satisfied, delete all medians and median deviations before the block.

Observations

1. A valid voter should be delegated from a validator that is currently in the active set.
2. A valid ballot for rate must satisfy the following properties:
 - denominations that accept votes should be in the accept list.
 - a voter should be an active validator who is neither jailed nor inactive.
 - a voter should submit prices for all accepted denominations in the previous block.
 - the prices a voter submitted for all accepted denominations in the voting period must be the same as the prices they submitted in the previous block.
 - ballot for rate should have at least 50% of total vote power.
3. Oracle rewards will not be distributed when the Oracle module does not have enough coins to cover the rewards.
4. At the end of each block:
 - increase the miss counter if a voter misses a vote.
 - exchange rates should be cleared at the last block of the voting period:
 - at the last block of the slashed window,
 - if the valid vote rate is smaller than the minimum threshold, slash, and jail the validator (bonded and active).
 - reset miss counters for all validators.
 - prune all prices.

Invariants

The protocol's core logic is expected to satisfy the following invariants. However, no formal proof is conducted to ensure they are satisfied with the implementation during this audit.

I01: *missCounter* never decreases, only if it's at the last block of the slashing window.



I02: Amount of rewards earned by all ballot winners equals the total outstanding reward.

Findings

Findings presented in this section are issues that can cause the system to fail, malfunction, and/or be exploited, and should be properly addressed.

All findings have a severity level and an execution difficulty level, ranging from low to high, as well as categories in which it fits.

A01: (Price feed contract) No error propagation in bulk operations


[Severity: Low | Difficulty: Medium | Category: Best practices]

Description

The price feed contract, intended to be one of the sources of data regarding the exchange rate of assets, is also capable of providing the rate of exchange between two tokens, the median of prices, and the price deviation for a token. All these operations can also be performed for a list of token identifiers provided by the users, which are referred to as *bulk* operations. These bulk operations iterate over the sequence of identifiers provided by the caller executing function calls that provide the required data for each of the identifiers received. A sample bulk operation can be seen as follows:

```
                Querying the exchange rate between two tokens
                (ojo/cosmwasm/contracts/price-feed/contract.rs module)

399  fn query_reference_data_bulk(
400      deps: Deps,
401      symbol_pairs: &[(String, String)],
402  ) -> StdResult<Vec<ReferenceData>> {
403      symbol_pairs
404          .iter()
405          .map(|pair| query_reference_data(deps, pair))
406          .collect()
407  }
```



The above piece of code displays that, for the bulk querying operations, a mapping of individual queried symbols (or pair of symbols) to their respective queried values is returned. What becomes an issue is the call for the function returning the individual asset information (which, in this case, is `query_reference_data_bulk`). While the called functions properly propagate the potential errors triggered by the function execution, the call for bulk operations does not do so, causing triggered errors to be mishandled by the contract.

Recommendation

Properly use the error propagation functionality provided by the Rust programming language using the '?' operator in order to handle function calls inside bulk operations.

Status

This finding has been acknowledged.

A02: (Price feeder) Potential risk of running pingTicker more times than desired

[Severity: Low | Difficulty: High | Category: Functional correctness]


Description

When utilizing Go's select implementation, the executed code will enable the system to wait on several communication operations simultaneously, blocking further execution of the function where the select statement is located until one of the operations successfully returns, proceeding to the execution of the case.

This statement is used in the price feeder module in order to periodically ping servers being communicated with, and its implementation can be seen in the code snippet below:

```
                                pingLoop function
                                (ojo/price-feeder/oracle/provider/Websocket_controller.go)

206  func (conn *WebsocketConnection) pingLoop() {
207      if conn.pingDuration == disabledPingDuration {
208          return // disable ping loop if disabledPingDuration
209      }
210      pingTicker := time.NewTicker(conn.pingDuration)
211      defer pingTicker.Stop()
212
213      for {
214          err := conn.ping()
215          if err != nil {
216              return
217          }
218          select {
219              case <-conn.websocketCtx.Done():
220                  return
221              case <-pingTicker.C:
222                  continue
223          }
224      }
225  }
```



When both *pingTicker* and *conn.websocketCtx.Done()* (Line 219 and 221: *Websocket_controller.go*) are written simultaneously, the nondeterministic-select randomly picks a case to execute. Thus the *pingTicker.C* may run multiple times.

Recommendation

It is best to avoid non-deterministic behaviors in safety-critical systems. In this scenario, such behavior can cause starvation for the *conn.websocketCtx.Done()* branch of execution for this select statement.

Status

This finding has been acknowledged.

A03: (Ojo node) Denial of service of the node (EndBlocker failure)

[Severity: High | Difficulty: Low | Category: Node malfunctions, Input validations]

Description


The *EndBlocker* function, executed at the end of every block at the Ojo node, performs a series of actions accounting the exchange rates provided in the users' votes, as well as, at the end of the voting and slashing periods, the necessary operations aiming to slash misbehaving validators and rewarding the ones that properly participated in the consensus of the protocol.

Regarding the end of voting periods more specifically, the node iterates through the ballots of validators and evaluates if the suggested exchange rates fall within a certain deviation range. One of the variables used to obtain the deviation range is titled *rewardBand*, and it is assigned as follows:

Obtaining rewardBand and checking for related errors (ojo/ojo/x/oracle/abci.go module)

```
066 // Get the current denom's reward band
067 rewardBand, err := params
                                .RewardBands
                                .GetBandFromDenom(ballotDenom.Denom)
068 if err != nil {
069     return err
070 }
```

In its current version, the *EndBlocker* function will fail every iteration where the voting period ends if a validator submits a vote for a token denomination that does not exist or is simply not included in the *RewardBands* list.. This causes the *params.RewardBands.GetBandFromDenom* function to fail and return an error. This error is propagated through the *CalcPrices* and *EndBlocker* functions, effectively terminating the *EndBlocker* without distributing rewards, setting the prices, clearing the ballots, and updating the miss counters. This also means that the *RewardBand* list is doing aggressive enforcement of what *AcceptedList*, which is used to filter token denominations that can be processed by the node, should/could do, thus it is necessary



to at least validate if all entries in *AcceptedList* have an entry in *RewardBands* as to prevent this issue from manifesting for valid ballots.

Recommendation

A naive approach for handling the above is to replace the *return err* in line 69 of the *abci.go* file by a *continue* statement. This would prevent the denial of service but would force all valid ballots to be directed to tokens that are included in the *RewardBands* list. This would effectively prevent users that try to add exchange rates to upcoming tokens to the Ojo oracle. Furthermore, an effective approach for solving this issue that strikes the balance between new token voting permissiveness and enforcement of the accepted token list still must be designed.

Status

This finding has been addressed.

A04: (Price feeder) Enforcement of time interval for last candles in TVWAP takes only in consideration the comparison between the candle timestamp and the local clock time minus 5 minutes

[Severity: Low | Difficulty: High | Category: Functional correctness]

Description


The *ComputeTVWAP* function (oracle/util.go) has the purpose of obtaining the TVWAP using a set of candles that fit within the last 5 minutes range, but it might not be the case if we consider it to be running on a machine with a deviated clock (either by misconfiguration or maliciously compromised), or also if the deviated clock is on the provider side.

There is a conditional on line 113 of the *ojo/price-feeder/oracle/util.go* module which checks if the times of the candles are greater than the “5 minutes ago” timestamp according to the machine’s clock. If said candles are considered to be in the future (validator’s clock time < provider’s clock time), they will be computed as a part of the TVWAP. This might result in problematic consequences as the higher the clock deviation is, the more candles can be added to the TVWAP calculation and thus reduce the precision of the calculated value.

Furthermore, in a scenario where you have candles that fit both in the last 5 minutes intervals and “in the future”, the VWAP value would be skewed towards the later candles, as *timeDiff*, the variable used for representing the delta between the machines clock and the candle timestamp, would be negative and the individual candle volume will be expressed as *candle.Volume * (weightUnit * (period - timeDiff) + minimumTimeWeight)*.

Recommendation

A cost-effective approach to handle this is simply to not include candles that have a timestamp greater than the local machine’s current timestamp, together with the check that is already in place. This would lay the responsibility of having a correctly configured clock on the validator’s side. Alternatively, it is possible to count the average number of candles that have been received or expect to receive within the 5 minutes range. For instance, if the provider sends you a candle every minute, then there is no reason to



include more than the 5 newest candles within the TVWAP calculation. Hypothetically, if a provider sends candles representing the span of seconds instead of minutes, it is possible to obtain the average number of candles received per minute A and multiply it by the N minutes range of the TVWAP. This value A will be used to limit the number of candles included in the TVWAP calculation. It is important to highlight that approaches to solving this issue, such as the one previously mentioned, may lead the code to be less maintainable.

Status

This finding has been addressed.

A05: (Ojo node) Price manipulation by overpowering validators

[Severity: High | Difficulty: High | Category: Exchange rate manipulation, Blockchain related particularities]

Description

While a validator's claim contains his voting power as well as in the instance of their vote on the ballots, evaluated by the Ojo node in the *EndBlocker*, the only part of the logic for calculation of the exchange rate where the voting power is used is to drop a ballot in case the voting power is below a threshold.

This means that all validators' votes have the same weight regardless of how much each individual stakes. If a malicious user wants to attempt to manipulate the value of an asset, they can, for instance, instantiate multiple validators with the minimal voting power needed to vote while providing their desired token prices.

For instance, if only a single token unit is required to become a validator and there are 15 honest validators staking 100 tokens in total in the protocol, a malicious user can attempt to create 15 validators to overpower the consensus, needing only 15 tokens to do so.

It is possible that whoever performs this attack will be at a severe loss of assets, given the mechanics of rewarding and slashing validators, but this will also depend on the number of validators providing prices for a specific asset and the minimum amount of tokens that have to be staked for becoming a validator. Even so, by manipulating the prices provided by an oracle, it is likely that the profit made by an attacker by exploring this issue will outweigh their costs, and that the barrier of entry to perform this manipulation attack has to be raised.

Recommendation

When evaluating the votes of validators, add the voting power as one of the weights for the calculation of an asset's exchange rate.

Status

This finding has been addressed.

A06: (Ojo node) Variable representing the number of maximum valid votes can be lower than the actual value

[Severity: Low | Difficulty: Medium | Category: Functional correctness, Blockchain related particularities]

Description

In the Ojo node, a validator is awarded a “win” if their exchange rate for the voted asset falls within the deviation range calculated by the protocol mechanisms. The maximum number of wins that a validator can have is calculated in the *PossibleWinsPerSlashWindow* function, in the *ojo/x/oracle/keeper/slash.go* module. The following equation describes the calculated value:

$$\text{possible wins} = \text{int}\left(\frac{\text{slash window duration}}{\text{voting period duration}}\right) \cdot \text{number of accepted assets}$$


The logic of the slashing operation is that firstly, we obtain the expected values for a possible amount of wins within a slash window and the minimum rate of wins; the first value is used for calculating the valid votes count of the validator (*validator valid vote rate = (possible number of wins - validator misses)/possible number of wins*). The worrying detail is how the possible number of wins is calculated, as it is the outcome of *int(slash window duration / voting period duration)*, where the truncated floating part of the division is discarded. This lowers the bar for the number of votes needed to be slashed as now the possible number of wins per slash window is smaller than the actual number of wins that a validator can have within the slashing period.

Scenario

For a better clarification of this finding, let's consider the following example:

- slashWindow = 8;
- votePeriod = 5;
- numberOfAssets = 10.

You can have at least 1 vote period within the slash window and at most 2. This means that you can have either 10 or 20 possible wins per slash window. The way that it is



implemented right now, you will always have 10. This can help an individual that is erroneously reporting prices not be slashed.

Recommendation

Enforce that the slash window duration is a multiple of the voting window duration, or do not truncate the returned division of these two values.

Status

This finding has been addressed.

A07: (Price feeder) Possibility for denial of service in the price feeder

[Severity: Medium | Difficulty: Medium | Category: System malfunction]

Description


Implemented in the price feeder project, the *ConvertCandlesToUSD* function in the *ojo/price-feeder/oracle/convert.go* module is responsible for converting candles that are not quoted in USD to USD through means of intermediary assets.

The logic between lines 58 to 78 (inside the *ConvertCandlesToUSD* function) aims to find assets that can be intermediary assets for, from the origin token, reaching USD with one hop. Having the origin token as Token A, which has candle information in a pair with Token B, to reach USD, the first thing that is done is to search for providers that have exchange rates from Token B to USD. The following step is to find candles from the identified providers that have Token B as its quote, adding these specific candles to a list of valid candles to be used for price conversion. This last step has no validation regarding if the base of that candle is actually USD, meaning that candles with the exchange rate of assets other than USD can be used for calculating the price of Token A.

According to the configuration of assets fetched by the price feeder, in the *ConvertCandlesToUSD* function, a map of valid candles that can be used for calculating the USD prices of a token through intermediary tokens is built. The problem arises from the fact that if no candles are found providing an exchange rate from the intermediary token to USD, an error will be returned, halting the price fetching process for all candles.

To perform the conversion of the exchange rates, the implementation strategy of *ConvertCandlesToUSD* attempts to iterate through the list of candles provided as parameters, performing the necessary preparations for the exchange rate conversion. The error highlighted above is being triggered within this loop and is immediately returned, which, in turn, is detected in the *GetComputedPrices* function in the *ojo/price-feeder/oracle.go* module and returned immediately as well.

Such an issue will prevent the price-feeder from submitting prices, also implying risking the possibility of slashing a validator. The behavior of the implementation assumes a high-impact response for a low-severity issue, as the process of price conversion can



be continued while ignoring the candles where the exchange rate failed to be converted. Still, it is interesting to enforce that there are always paths of conversion between assets and USD during the instantiation of the price-feeder, notifying the user of any impossibilities to generate USD prices for assets.

Recommendation

If no valid candles are found for the asset exchange rate conversion, then instead of returning an error, the asset should be ignored and the calculation of prices for the remaining assets should be performed.

Status

This finding has been addressed.

A08: (Price feeder) Filtering of candles delays modification of exchange rates for assets

[Severity: Low | Difficulty: High | Category: Exchange rate manipulation]

Description

Akin to findings [A07](#), this finding also considers aspects of the implementation of the *ConvertCandlesToUSD* function.

When converting the prices of candles to USD, candles that will be used for the exchange rate conversion are filtered prior to the asset price calculation. This is done by obtaining the TVWAP of the token price for a specific provider and removing candles with values outside of a calculated deviation range, by default 1 standard deviation, aiming to remove the influence of possibly incorrect candle values.


What causes this to be problematic is the fact that any candles with outlier values will be removed from the exchange rate calculation and, in a scenario of drastic price fluctuations, the newest and valid asset prices will not be used for future calculations, delaying the time taken to update the asset exchange rate to its correct value.

Scenario

Token X is very volatile, but has been quite stable for the last TVWAP window (default being 5 minutes). As news of a rug pull/large collaboration emerges the price takes a sharp and immediate drop/boost, far outside the standard deviation of the latest prices. As a result, all new incoming prices get filtered out. Instead of the oracle TVWAP price converging on the new actual price, the price remains frozen for the entire window. Only when a full window has passed since the price change will the price incidentally get corrected, since there is no longer a reference price history to refer to, and any new incoming prices will be automatically accepted. The initial oracle prices may be more volatile than usual due to lack of historical candles against which to calculate a TVWAP.

Recommendation

Do not perform candle filtering prior to the exchange rate calculation or use a less aggressive filter. Ideally, all reported candles are kept for future calculations of standard deviations and means before filtering. The filter only applies before calculating the TVWAP. The candles are still kept around for calculating the mean and standard



deviation for the filter that is applied before the TVWAP calculation. This way, the real price deviation drives the mean towards the new price and increases the standard deviation gradually, until the new candles get fully included.

Status

This finding has been acknowledged.

Informative Findings

Findings presented in this section do not necessarily represent any flaw in the code itself. However, they indicate areas where the code may need some external support or where the code deviates from best practices. We have also included information on potential code size reductions and remarks on the operational perspective of the contract.

B01: (Price feed contract) RELAYERS map values are not used

[Severity: Informative | Difficulty: - | Category: Best practices, Code optimization]

Description

Whenever a relayer is to be considered trustworthy, an entry is added to the RELAYERS map with a key-value pair composed of: the relayer address as a key, and the boolean value true representing its status, according to the comment above the map instantiation.

The value mapped by the relayer address is never used as the validation process that identifies an address as a validator checks only if the address has an entry in the RELAYERS map.

Recommendation

To implement the RELAYERS entity, either a list or a set would be more appropriate.

Status

This finding has been acknowledged.

B02: (Price feed contract) Median data is a vector, which implies that the current median is not stored

[Severity: Informative | Difficulty: - | Category: Best practices]

Description

When providing the median from a set of values, it is expected that the closest value to the midpoint of the ordered set will be returned. In Ojo's price-feed contracts, all values from the set are returned.

Recommendation

The contract's functions and variables' nomenclature should be adapted to reflect the operation performed and the data returned, or the implementation should be modified so that the returned value actually respects the functions and variables' nomenclature.

Status

This finding has been acknowledged.

B03: (Price feed contract) Relayers are fully trusted by the contract and no validations are performed over relayed rates

[Severity: Informative | Difficulty: - | Category: Blockchain related particularities]

Description

Relayers are able to perform relays to the price-feed contract either gracefully or forcefully. In both cases, no validations over the actual exchange rate provided by the relayer are performed. The value is simply placed in the storage of the price feed contract.

Recommendation

For oracle contracts, it is common to have mechanisms for preventing aggressive price manipulation attacks, preventing the price from being updated (or disabling a specific query functionality) when there is an attempt to relay a price with a deviation considered too high.

Status

This finding has been acknowledged.

B04: (Price feed contract) Median status is ignored when executing `execute_force_relay_historical_median`

[Severity: Informative | Difficulty: - | Category: Best practices]

Description

All requests to update or query the historical median data will fail if the median status is false, indicating that this endpoint is disabled in the contract. The `execute_force_relay_historical_median` is still able to update the historical median value if the median status is false, which does not have much of a purpose, given that whatever value that is being pushed will not be used due to the disabled queries.

Recommendation

It is recommended to verify if the historical median is enabled prior to forcefully updating its value.

Status

This finding has been acknowledged.

B05: (Price feed contract) There are no prevention mechanisms to fully stop the price relaying process

[Severity: Informative | Difficulty: - | Category: Blockchain related particularities]

Description

Only the median data can be halted by changing the MEDIANSTATUS boolean variable. Deviation, reference, and ref data still can be queried. Assuming the possibility of a corrupted price data source or a rogue relayer, while probably unknown to the parties involved in the protocol management during the beginning of the anomalous event, it is advisable to enable the admin to change the status of all types of queries to false (disabling it).

Recommendation

Following a standard for operations of similar purposes is a best practices recommendation. With this in mind, it is recommended to also have statuses controlled by admins that will define if it is possible to query and update values in other endpoints of the contract.

Status

This finding has been acknowledged.

B06: (Price feed contract) No checks on the received resolve time

[Severity: Informative | Difficulty: - | Category: Best practices, Input validation]

Description

General guidelines on smart contracts development indicate that contracts should be self-contained regarding the checks on received parameters. With this in mind, assuming the possibility of a compromised relayer, the contract could receive a price/rate to be relayed with a resolve time far in the future, preventing other non-compromised relayers from normally pushing rates to the contract. A forced overwrite would be required to prevent malicious manipulation of the price feed contract.

Recommendation

It is recommended to check if the timestamp is from a moment in time that is equals to or prior to the current blockchain timestamp, possibly adding a time window for protections against relay delays or other time synchronicity issues.

Status

This finding has been acknowledged.

B07: Outdated and vulnerable dependencies

[Severity: Informative | Difficulty: - | Category: Best practices]

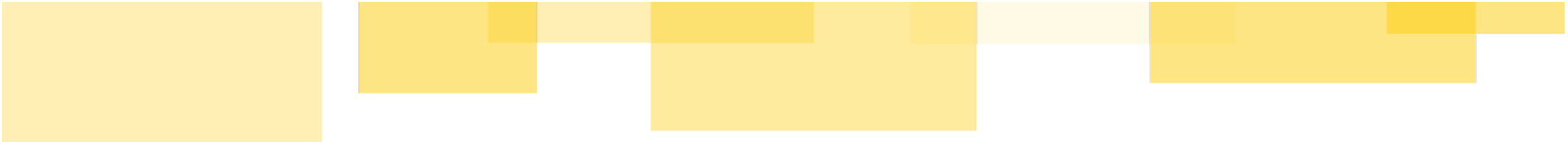
Description

Ojo node, price feeder, and smart contracts rely on outdated and vulnerable dependencies. The table below lists the problematic packages used by Ojo dependencies and the suggested patched versions.

Package	Vulnerabilities	gomod	Patched Version
github.com/aws/aws-sdk-go@1.44.122	CVE-2020-8911 CVE-2020-8912	Ojo, cw-relayer, and price-feeder	github.com/aws/aws-sdk-go-v2@1.2.3
github.com/btcsuite/btcd@0.22.1	CVE-2022-44797	Ojo, cw-relayer, and price-feeder	v0.23.2
golang.org/x/net@0.5.0	CVE-2022-41723	Ojo, cw-relayer, and price-feeder	v0.7.0
github.com/hashicorp/go-getter@1.6.1	CVE-2023-0475	price-feeder and cw-relayer	v1.7.0
golang.org/x/crypto@0.0.0-20220112180741-5e0467b6c7ce	CVE-2022-27191	Ojo testnet	v0.0.0-20220314234659-1baeb1ce4c0b
github.com/mholt/archiver@3.1.1+incompatible	CVE-2019-10743	Ojo testnet	v3.3.2+incompatible

Recommendation

Update the outdated and vulnerable dependencies, even if they do not currently affect Ojo network.



Status

This finding has been addressed.

B08: (Price feeder) Potential cross-site scripting (XSS) vulnerability

[Severity: Informative | Difficulty: - | Category: Best practices]

Description

The *html/template* is used in the *router/v1/router.go*, which would cause the Ojo network under a potential XSS attack. XSS is a computer security vulnerability typically found in web applications that renders user-generated content. Properly escaping any HTML content is the primary means to prevent XSS attacks.

Recommendation

To mitigate this risk, it is recommended to use the Go *html/template* instead of the *text/template* package. The *html/template* provides built-in functionality for contextual autoescaping, which means that when your templates are parsed, the library detects in which context you are putting strings, and it will pick a chain of escaping functions to encode it properly.

Status

This issue has been addressed.

B09: (Ojo node) Add the voting power used for converging into an exchange rate

[Severity: Informative | Difficulty: - | Category: Blockchain related particularities]

Description

Currently, not only the voting power is not used to calculate the exchange rate of an asset, but the user has no way of knowing how much commitment there was at the end of the validators for providing the values stored in the contract.

Recommendation

A proposed layer of protection for consumers of the Ojo node data is the addition of voting power information to the endpoints that provide token prices. That way, a user can not only know the exchange rate of a token but also how much commitment of validators was put into obtaining the token price, which would help prevent issues of price manipulation if a small number of validators dictate the exchange rate of a token.

Status

This finding has been acknowledged.

B10: (Ojo node) Median endpoint does not provide a timestamp or block number

[Severity: Informative | Difficulty: - | Category: Best practices]

Description

The median endpoints of the Ojo node does not provide information about the timing of the provided medians. This may be a source of issues given the possibility of gaps in between entries of the median list.

In conditions where a specific token has no ballots proposing exchange rates due to, for instance, the lack of voting power of the validators submitting votes for that asset, no exchange rate data will be stored for that specific timestamp. Therefore, a user that queries the median list will unknowingly query a list of medians with a gap during specific time windows where there was no vote.

Recommendation

Include a timestamp or block number for the medians provided in the median endpoints.

Status

This finding has been addressed.

B11: (Ojo node) No validations on possible division by zero

[Severity: Informative | Difficulty: - | Category: Input validation]

Description

Akin to finding [A06](#), this finding also considers aspects of the implementation of the maximum possible wins of a validator calculation during a voting period.

Following the concept that modules should be self-contained regarding validations of their vital parameters, there is no check to ensure that the slashing window duration is greater or equal to voting period duration. If we run into the case where the first is smaller than the second, the number of possible wins of a validator will be calculated as 0 as this is what you would get from truncating the division of slash window duration by voting period duration. The outcome of this would be a division by 0 in this same slash module, crashing the execution of the *EndBlocker*.

Recommendation

This is not a critical issue, given that the validation of these values is performed in the governance module and so the issue will not manifest, but it is best practice to have checks like these to prevent unexpected behaviors and possible crashes in the module itself, as well as to notify admins in future versions of the node where the governance implementation may be modified.

Status

This finding has been acknowledged.

B12: (Price feeder) Possibility for price manipulation in the price feeder project

[Severity: Informative | Difficulty: - | Category: Exchange rate manipulation, Best practices]

Description

Akin to findings [A07](#), this finding also considers aspects of the implementation of the *ConvertCandlesToUSD* function.

There is a validation in the price feeder configuration validation file, but, following the concept of self-contained modules, validation of potentially critical attributes of the module must be performed in the module itself. If modifications in future versions of the protocol affect the configuration validation step, price manipulation strategies will become viable for malicious users.

While there are filters to protect the price based on the deviation of the retrieved values, the presence of values that are outside of the deviation range may affect the output of the conversion function, potentially being a cause for the user to be slashed. Also, there are situations where the intermediary token price may be close to the dollar value, falling within the valid deviation and being of influence to the final asset exchange rate.

Recommendation

When searching for hops that will be used to convert the asset exchange rate to USD, it should be enforced that the asset at the end of the chain of hops is indeed USD.

Status

This finding has been acknowledged.