Prisma Audit

BROKEN WITH ❤ BY nomoi

We reviewed the https://github.com/prisma-fi/prisma-contracts repository at commit d43922b42ffce5502e57e6dcea8b17dbe9b895cd.

***Update**: After the audit was finished, the Prisma team requested us to review some additional changes that consisted in a completely new PriceFeed and the support for duplicated collateral tokens across TroveManagers. No major issues were found in this second part of the audit.*

Introduction

Our team has undertaken a thorough security review of the Prisma protocol, with the objective of independently assessing the security aspects, code quality, and overall functionality of the project's smart contracts. Prisma is a unique entrant in the DeFi space, targeting the effective utilization of Ethereum's liquid staking tokens (LSTs).

The core functionality of Prisma revolves around enabling users to mint an overcollateralized stablecoin (mkUSD) backed by various supported LSTs. Further incentives are provided through platforms such as Curve and Convex Finance, offering users the opportunity to accumulate CRV, CVX, and PRISMA, in addition to their Ethereum staking rewards.

Prisma's codebase is based on Liquity's contracts, which were heavily modified to simplify the code and support multiple collaterals. Additionally, it provides contracts that allow flexibility in governance level decisions concerning collaterals and other protocol parameters. The Prisma DAO manages these governance aspects, encompassing parameters, emissions, and protocol fees.

This audit report's primary recommendations target enhancing the Prisma protocol's security, mainly by fixing issues related to governance and using safer practices when dealing with external protocol integrations.

The smart contracts in scope for this audit were:

```
.
├── core
│   ├── BorrowerOperations.sol
│   ├── DebtToken.sol
│   ├── Factory.sol
│   ├── GasPool.sol
│   ├── LiquidationManager.sol
│   ├── PriceFeed.sol
│   ├── PrismaCore.sol
│   ├── SortedTroves.sol
│   ├── StabilityPool.sol
│   └── TroveManager.sol
├── dao
│   ├── AdminVoting.sol
│   ├── AllocationVesting.sol
│   ├── BoostCalculator.sol
│   ├── EmissionSchedule.sol
│   ├── FeeReceiver.sol
│   ├── IncentiveVoting.sol
│   ├── PrismaToken.sol
│   ├── TokenLocker.sol
│   └── Treasury.sol
├── dependencies
│   ├── DelegatedOps.sol
│   ├── PrismaBase.sol
│   ├── PrismaMath.sol
│   ├── PrismaOwnable.sol
│   └── SystemStart.sol
└── staking
    ├── Convex
    │   ├── ConvexDepositFactory.sol
    │   └── ConvexDepositToken.sol
    └── Curve
        ├── CurveDepositFactory.sol
        ├── CurveDepositToken.sol
        └── CurveProxy.sol
```

Findings

1. Anyone can steal pending rewards from treasury

LIKELIHOOD HIGH IMPACT MEDIUM

The `Treasury.transferAllocatedTokens` function can be called by anyone to steal pending rewards for any account. This can be done by setting the `amount` parameter to `0`, which would bypass the first line of the function and transfer all the `claimant`'s pending rewards to any `receiver`.

***Update**: the Prisma team independently discovered this issue and fixed it in commit `28007d6` by only allowing non-zero `amount`s. This solution still puts some trust in the accounts that have been allocated some funds by the Treasury, as these can obtain the pending rewards of any other account.*

2. Incorrect `require` condition to fetch rewards

LIKELIHOOD HIGH IMPACT LOW

The `ConvexDepositToken.fetchRewards` function includes the following require statement:

```
require(block.timestamp / 1 weeks > periodFinish / 1 weeks, "Can only fetch o
```

This condition, however, presents a discrepancy with the other functions within the same contract, as well as the `CurveDepositToken.fetchRewards` function. These counterparts use a `>=` comparison operator, establishing a slightly different rule, instead of the `>` operator found in the `ConvexDepositToken.fetchRewards` function.

The consequence of this is that it will not be possible to call `fetchRewards` unless a whole period passes and no other operations are performed during that period, presumably defeating the whole purpose of this function.

Recommendation

Consider fixing this condition to be consistent with the rest of the code.

***Update**: As of commit `86fb60a` this issue has been resolved by changing the condition in the `require` statement.*

3. Incorrect account weight is returned under certain conditions

LIKELIHOOD HIGH IMPACT LOW

The `while` loop condition in `TokenLocker.getAccountWeightAt` should compare `accountWeek` against `week`, instead of `systemWeek`. This results in the function returning lower weights than it should when querying the weight for an account that hasn't been updated recently.

The `getAccountWeightAt` function is used by the `AdminVoting` and `BoostCalculator` contracts, so even if it is a `view` function, this issue affects the actual voting power of users and their rewards.

Recommendation

Consider fixing the `while` loop condition so that the function always returns the correct weights.

***Update**: As of commit `86fb60a` this issue has been resolved by changing the `while` condition to use the `week` parameter instead of `systemWeek`.*

4. Incorrect total weight is returned under certain conditions

LIKELIHOOD HIGH IMPACT LOW

The `while` loop condition in `TokenLocker.getTotalWeightAt` should compare `updatedWeek` against `week`, instead of `systemWeek`. This results in the function returning lower weights than it should when querying the total weight for a specific week.

The `getTotalWeightAt` function is used by the `AdminVoting` and `BoostCalculator` contracts, so even if it is a view function, this issue affects the required weight for a proposals to pass and also user rewards.

Recommendation

Consider fixing the `while` loop condition so that the function always returns the correct weights.

***Update**: As of commit `86fb60a` this issue has been resolved by changing the `while` condition to use the `week` parameter instead of `systemWeek`.*

5. `boostDelegate` callback fee is ignored

LIKELIHOOD MEDIUM IMPACT MEDIUM

The `Treasury.claimableRewardAfterBoost` function calls the `boostDelegate`'s callback if the fee is set as `type(uint16).max`. It wraps this call in a `try / catch`, but it doesn't capture the return value. As in this case the `fee` is set to a value larger than the maximum fee, the condition on line `425` will be `true` and the function will return early. Thus, all calls to `claimableRewardAfterBoost` will return `(0, 0)` for `boostDelegate`s that provide a fee through the `getFeePct` function.

Even though this function is not called by other components of the system, other integrations such as frontends and external protocols might rely on this functionality and could break in unexpected ways.

Recommendation

Consider correctly using the value returned by the `getFeePct` function.

***Update**: As of commit `86fb60a` this issue has been resolved by using the return value of the `boostDelegate`'s callback in the `try/catch` block.*

6. Convex LP tokens can get stuck

LIKELIHOOD MEDIUM IMPACT MEDIUM

The `ConvexDepositToken.treasuryClaimReward` function doesn't decrease the `lastCrvBalance` and `lastCvxBalance` state variables before transferring rewards. This can lead to an underflow in the `_fetchRewards` function when subtracting these amounts from the actual token balances.

Recommendation

Consider updating the `lastCrvBalance` and `lastCvxBalance` state variables when transferring CRV and CVX tokens.

***Update**: As of commit `86fb60a` this issue has been resolved by updating the CRV and CVX token balances before transferring them.*

7. Users can withdraw locked tokens before expiration and with no penalties

LIKELIHOOD MEDIUM IMPACT MEDIUM

Users can withdraw their locks before expiration and with no penalties by calling `TokenLocker.withdrawWitPenalty` multiple times, due to rounding errors. It is also possible to lock tokens and withdraw them in the same transaction, which can lead to inconsistent states. This is possible due to the loss of precision when calculating the penalty for a withdrawal.

One example of how this can be abused is to inflate the voting power related to incentive distribution, by "double spending" governance tokens using multiple accounts:

1. Create a lock in the `TokenLocker` contract
2. Call the `IncentiveVoting.registerAccountWeightAndVote` function to vote for the desired receiver
3. Use `withdrawWithPenalty` to withdraw the lock without penalties
4. Transfer the tokens to a different address
5. Repeat steps from 1 to 4

We found two ways of calling `withdrawWithPenalty` without paying the penalty:

In the following descriptions, `lockAmount` is the amount of prisma tokens locked, `amountToWithdraw` the amount of locked prisma tokens to withdraw (passed as a parameter to `withdrawWithPenalty`) and `weeksToUnlock` the amount of weeks left for the lock to expire.

1. Abusing rounding error in line 801

```
uint256 penaltyOnAmount = (lockAmount * weeksToUnlock) / MAX_LOCK_WEEKS
```

Conditions that will set `penaltyOnAmount == 0`:

$$lockAmount = amountToWithdraw$$

$$1 \leq lockAmount * weeksToUnlock < 52$$

2. Abusing rounding error in line 806

```
penaltyOnAmount = (remaining * MAX_LOCK_WEEKS) / (MAX_LOCK_WEEKS - weeksToUnl
```

Conditions that will set `penaltyOnAmount == 0`:

$$lockAmount - \left\lfloor \frac{lockAmount * weeksToUnlock}{52} \right\rfloor > amountToWithdraw$$

$$\left\lfloor \frac{amountToWithdraw * 52}{52 - weeksToUnlock} \right\rfloor = amountToWithdraw$$

Some examples of values that will meet the conditions above:

$$weeksToUnlock = 1$$

$$1 \leq amountToWithdraw \leq 50$$

and

$$amountToWithdraw = 1$$

$$1 \leq weeksToUnlock \leq 25$$

Recommendation

Consider rounding in favor of the protocol or preventing penalties of 0.

***Update**: As of commit `86fb60a` this issue has been resolved by clearing the registered weight when users withdraw tokens using the `withdrawWithPenalty` function. The rounding problem was acknowledged but it wont be fixed because it's not economically viable to abuse it.*

8. Boost Delegate fee can change unexpectedly

LIKELIHOOD LOW IMPACT LOW

When calling the `Treasury.batchClaimRewards` function and using a `boostDelegate`, it is possible for the delegate to change its fee right before the `batchClaimRewards` call is executed, potentially resulting in an unexpectedly high fee from the perspective of the caller.

Recommendation

Consider adding a new argument to the `Treasury.batchClaimRewards` function so that the caller can specify the maximum fee that they are willing to accept.

***Update**: As of commit `86fb60a`, this issue has been resolved by including a `maxFeePct` argument in the `Treasury.batchClaimRewards`.*

9. Convex extra rewards could temporarily prevent withdrawals

LIKELIHOOD LOW IMPACT LOW

The `ConvexDepositToken.withdraw` function calls `crvRewards.withdrawAndUnwrap`, passing the `claim` argument as `true` if rewards haven't been fetched for the previous period. The `crvRewards` contract will internally call `crvRewards.getReward`, passing `_claimExtras` as true. This function will then iterate through the `extraRewards` contracts and perform external calls that might fail, reverting the transaction.

The impact of this issue is limited, however, as a call to `fetchRewards` would unlock the funds in this situation.

Recommendation

Consider calling `crvRewards.withdraw(amount, false)` and `crvRewards.getReward(false)` instead of a single call to `withdrawAndUnwrap`. This provides the same functionality but prevents the external calls to the `extraRewards` contracts.

***Update**: As of commit `86fb60a`, this issue has been resolved by preventing external calls to the `extraRewards` contracts.*

10. Inconsistent boost delegation fee validation

LIKELIHOOD LOW IMPACT LOW

The `Treasury.setBoostDelegationParams` function allows setting a fee of `10000` (100%). The `_transferAllocated` function also supports this fee value. However, the `claimableRewardAfterBoost` function only supports a fee up to `9999`, returning an incorrect result of `(0, 0)` if using the maximum fee. This could potentially lead to inconsistencies if external integrations such as the frontend or other contracts rely on the `claimableRewardAfterBoost` function.

Recommendation

Consider fixing the way the `claimableRewardAfterBoost` function handles a fee value of `10000`, by correctly returning the `ajustedAmount` and `feeToDelegate`.

***Update**: As of commit `86fb60a`, this issue has been resolved by correctly handling a fee value of `10000` in `claimableRewardAfterBoost`.*

11. Insufficient sanity check when registering a receiver in the Treasury

LIKELIHOOD LOW IMPACT LOW

The `Treasury.registerReceiver` function notifies the receiver contract by calling the `IEmissionReceiver.notifyRegisteredId` function. A comment before that call indicates that it is also used as a sanity check to "ensure the contract is capable of receiving emissions". However, the `boolean` return value of the `notifyRegisteredId` is not checked, so a future implementation of `IEmissionReceiver` that returns `false` could be registered incorrectly.

Recommendation

Consider always checking the return value of the `notifyRegisteredId` function.

***Update**: Acknowledged, the team considers that the return data length checks performed by the compiler are enough to confirm that a valid receiver is being used.*

12. Prisma tokens can get locked if transferring to Treasury through `transferTokens`

LIKELIHOOD LOW IMPACT LOW

The `Treasury.transferTokens` function will decrease the `unallocatedTotal` state variable and emit an event if Prisma tokens are transferred. However, if the `receiver` is the Treasury itself, the `unallocatedTotal` will still be decreased but the balance will not change, also emitting an event with misleading information.

Note: It is extremely unlikely for this to happen, as `transferTokens` can only be called by the `owner`.

Recommendation

Consider preventing transferring Prisma tokens to the Treasury itself in the `transferTokens` function.

***Update**: As of commit `86fb60a`, this issue has been resolved by preventing the `Treasury` from transferring Prisma tokens to itself.*

13. Reward integrals not updated when claiming through Treasury

The `CurveDepositToken.treasuryClaimReward` doesn't call `_updateIntegrals`. The impact seems to be limited, however, as the user can claim the unaccounted rewards by claiming rewards directly with the `claimReward` function.

Recommendation

Consider calling `_updateIntegrals` at the beginning of the `treasuryClaimRewardFunction`.

***Update**: As of commit `86fb60a`, this issue has been resolved by correctly updating reward integrals when claiming through the Treasury.*

14. Use `staticcall` instead of `call` when fetching exchange rate

The `PriceFeed.fetchPrice` function performs a low-level `call` to the `collateral` contracts to get the share price. This could result in unexpected issues such as the possibility of reentrancy attacks. Even though the current collateral contracts don't present such issues, this could change as the code of some of them is upgradeable, and new collaterals might be added in the future.

Recommendation

Consider using a `staticcall` to prevent unexpected issues when fetching the share price of the collateral contracts.

***Update**: As of commit `86fb60a`, this issue has been resolved by using `staticcall` instead of `call` when fetching collateral exchange rates.*

15. Use `staticcall` instead of `call` when fetching share price (second part)

LIKELIHOOD LOW IMPACT LOW

The new `PriceFeed` implementation performs a low-level `call` to the `_token` contracts to get the share price. This could result in unexpected issues such as the possibility of reentrancy attacks. Even though the current collateral contracts don't present such issues, this could change as the code of some of them is upgradeable, and new collaterals might be added in the future.

Recommendation

Consider using a `staticcall` to prevent unexpected issues when fetching the share price of the collateral contracts.

***Update**: As of commit `f72caf8`, this issue has been resolved by using `staticcall` instead of `call` when fetching collateral exchange rates.*

16. Assume fixed decimals for Chainlink Aggregators (second part)

ENHANCEMENT

The new `PriceFeed` implementation queries the Chainlink Aggregator `decimals` on each call to `fetchPrice`. It is safe to assume that the number of decimals for an Aggregator will not change, so the decimals can be cached when the Aggregator is added.

17. Change `PriceFeed._isPriceChangeAboveMaxDeviation` function state mutability (second part)

ENHANCEMENT

The `PriceFeed._isPriceChangeAboveMaxDeviation` is declared as `view` but it can be restricted to `pure`

18. Documentation Issues

ENHANCEMENT

These are some examples where documentation and naming could be improved:

- The internal functions in the `PrismaMath` library should not begin with an underscore, as they are meant to be used by other contracts
- For the two-parameter variation of the `PrismaMath._computeCR` function, it is important to explicitly denote that the `_coll` parameter must have the price factored in
- Naming issues:
  - In `BorrowerOperations.getTCR`, `amount` should be renamed to `tcr`
  - In `BorrowerOperations.checkRecoveryMode`, `TCR` should be renamed to `tcr`
  - In `TroveManager`, `EtherSent` should be renamed to `CollateralSent`
  - In `DebtToken`, `enableCollateral` should be renamed to `enableTroveManager` (second part of the audit)
- Use UPPER_SNAKE_CASE for constants and immutable:
  - `lockToTokenRatio`
  - `lockToken`
  - `incentiveVoter`
  - `prismaCore`

- Incorrect comments that reference Liquity's codebase:
  - In `StabilityPool._updateG` , the comment refers to the inexistent `CommunityIssuance` contract
  - In `StabilityPool._claimReward` , incorrect comment "Needed only for event log"
  - In `TroveManager._redeemCloseTrove` , "send collateral to account" and "send collateral from Trove Manager to CollSurplus Pool" incorrectly describe the code
- The new `PriceFeed` contract (second part of the audit) contains a link in the comments which points to the incorrect reference implementation.

Consider thoroughly reviewing inline documentation and improving variable, function, and parameter names to boost the codebase's consistency and readability.

19. Magic Values

ENHANCEMENT

There are several instances of hardcoded values throughout the codebase, which reduces the readability of the codebase. Some examples include:

- `TroveManager:L934 : 1e18`
- `TroveManager:L948 : 1e18`
- `IncentiveVoting:L200 : 1e18`
- `Treasury:L329 : type(uint16).max`
- `Treasury:L331 : 10000`

Consider replacing all hardcoded values with constants.

20. Unnecessary comparison

ENHANCEMENT

The `TokenLocker.getAccountWeightAt` function performs a redundant comparison on line 177, as the condition is always `false` .

Consider removing lines 177-179 to simplify the code.

21. Unnecessary state variable update

ENHANCEMENT

The `TroveManager._resetState` function updates the `lastDefaultInterestUpdate` variable to `block.timestamp` and a few lines bellow it updates the variable again to `0`.

Consider updating the variable only once to the correct value.

22. Use same type of errors within each contract (second part)

ENHANCEMENT

The new `PriceFeed` uses custom errors everywhere except for the `require` statement used to check if the share price call fails. Consider using the same type of error for consistency.

23. Optimizations

OPTIMIZATION

The following list is a non-exhaustive list of optimizations that should not affect the functionality of the protocol. We decided to include them as they seem like simple enough improvements which, in many cases, significantly improve gas usage. Even though we reviewed each one of these and believe that implementing them won't result in changes to the actual functionality of the contracts, the team should consider that any change has the risk of introducing unrelated issues to the codebase.

Unnecessary storage operations

There are several places in the codebase where a storage operation is not necessary under certain conditions:

- In `StabilityPool._updateDepositAndSnapshots` the storage write `deposits[_depositor] = _newValue;` can be moved after the `if` statement.

- In `StabilityPool.getDepositorCollateralGain` the `uint80[256] storage depositorGains` variable declaration can be moved after the first conditional return.
- In `StabilityPool.getDepositorCollateralGain` the `uint256 initialDeposit` variable declaration can be moved after the first conditional return.
- In `StabilityPoo.getDepositorCollateralGain` the `uint128 epochSnapshot` and `uint128 scaleSnapshot` variable declaration can be moved after the first conditional return.

Multiple storage reads of the same value

There are a few places in the codebase where a storage is unnecessarily read multiple times. For example:

- The `IncentiveVoting.getAccountCurrentVote` function reads `accountLockData[account]` multiple times.
- The `StabilityPool._accrueDepositorCollateralGain` function reads `depositSnapshots[_depositor]` multiple times.
- The `StabilityPool.getDepositorCollateralGain` function reads `depositSnapshots[_depositor]` multiple times.
- The `AllocationVesting.transferPoints` function reads `allocationPoints[from]` multiple times.
- The `AllocationVesting.transferPoints` function reads `allocationPoints[to]` multiple times.
- The `TroveManager._closeTrove` function reads `sortedTroves` multiple times.
- The `TroveManager._updateStakeAndTotalStakes` function reads `totalStakes` multiple times.
- The `TroveManager._computeNewStake` function reads `totalStakesSnapshot` multiple times.
- The `TroveManager._computeNewStake` function reads `totalCollateralSnapshot` multiple times.
- The `TroveManager._redistributeDebtAndColl` function reads `totalStakes` multiple times.
- The `TroveManager._redistributeDebtAndColl` function reads `L_collateral` multiple times.
- The `TroveManager._redistributeDebtAndColl` function reads `L_debt` multiple times.
- The `TroveManager._updateTroveRewardSnapshots` function reads `L_collateral` multiple times.

- The `TroveManager._updateTroveRewardSnapshots` function reads `L_debt` multiple times.
- The `TroveManager.collectInterests` function reads `interestPayable` multiple times.
- The `TroveManager.getPendingCollAndDebtRewards` function reads `Troves[_borrower]` multiple times.
- The `StabilityPool._vestedEmissions` function reads `lastUpdate` multiple times.
- The `StabilityPool._updateG` function reads `currentScale` multiple times.
- The `StabilityPool._updateG` function reads `currentEpoch` multiple times.
- The `StabilityPool._updateG` function reads `epochToScaleToG[currentEpoch][currentScale]` multiple times.
- The `StabilityPool._updateRewardSumAndProduct` function reads `currentScale` multiple times.
- The `StabilityPool._updateRewardSumAndProduct` function reads `currentEpoch` multiple times.

Others

- The `Treasury.registerReceiver` function computes the system's week on each iteration. Consider storing it in memory.
- The `StabilityPool._triggerRewardIssuance` function is called often. A simple optimization is to not declare the `issuance` variable and pass the result of `_vestedEmissions()` directly as an argument.
- Calculating the `minPrice` and `maxPrice` in the `PriceFeed` contract can be simplified to prevent unnecessary comparisons by writing it as follows:

```
(uint256 minPrice, uint256 maxPrice)  = scaledTellorPrice < scaledChainlinkPr
```

- (second part) The `PriceFeed._scalePriceByDigits` function can be optimized by reordering the branches of the `if` so that the most common branch is checked first.