prePO contest
Findings & Analysis Report

2023-01-27

Table of contents

- Overview
  - About C4
  - Wardens
- Summary
- Scope
- Severity Criteria
- High Risk Findings (2)
  - [H-01] griefing / blocking / delaying users to withdraw
  - [H-02] A whale user is able to cause freeze of funds of other users by bypassing withdraw limit
- Medium Risk Findings (7)
  - [M-01] Bypass `userWithdrawLimitPerPeriod` check
  - [M-02] The recipient receives free collateral token if an ERC20 token that deducts a fee on transfer used as baseToken
  - [M-03] Frontrunning for unallowed minting of Short and Long tokens
  - [M-04] PrePO NFT holders will not be able to redeem collateral

    - [M-05] `PrePOMarket.setFinalLongPayout()` shouldn’t be called twice.
    - [M-06] Manager can get around min reserves check, draining all funds from Collateral.sol
    - [M-07] Users do not receive owed tokens if `TokenSender` contract cannot cover their owed amount.
- Low Risk and Non-Critical Issues
    - Low Risk Issues Summary
    - L-01 Missing calls to `__ReentrancyGuard_init` functions of inherited contracts
    - L-02 Draft OpenZeppelin Dependencies
    - L-03 There is a risk that the `proxyFee` variable is accidentally initialized to 0 and platform loses money
    - L-04 Owner can renounce Ownership
    - L-05 Missing Event for critical parameters init and change
    - L-06 A single point of failure
    - L-07 Using vulnerable dependency of OpenZeppelin
    - L-09 Loss of precision due to rounding
    - Non-Critical Issues Summary
    - N-01 Critical Address Changes Should Use Two-step Procedure
    - N-02 Initial value check is missing in Set Functions
    - N-03 Use a single file for all system-wide constants
    - N-04 NatSpec comments should be increased in contracts
    - N-05 Function writing that does not comply with the Solidity Style Guide
    - N-06 Add a timelock to critical functions
    - N-07 Use a more recent version of Solidity
    - N-08 Solidity compiler optimizations can be problematic
    - N-09 Include return parameters in NatSpec comments
    - N-10 Omissions in Events
    - N-11 Long lines are not suitable for the Solidity Style Guide

  - N-12 Avoid *shadowing* inherited state variables
  - N-13 Open TODOs
  - N-14 Constant values such as a call to `keccak256()`, should use immutable rather than constant
  - N-15 Mark visibility of initialize(...) functions as external
  - Suggestions
  - S-01 Make the Test Context with Solidity
  - S-02 Project Upgrade and Stop Scenario
  - S-03 Generate perfect code headers every time
- Gas Optimizations
  - Gas Optimizations Summary
  - G-01 Expressions for constant values such as a call to `keccak256()`, should use immutable rather than constant
  - G-02 Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate
  - G-03 State variables should be cached in stack variables rather than re-reading them from storage
  - G-04 Stack variable used as a cheaper cache for a state variable is only used once
  - G-05 Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` statement
  - G-06 `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables
  - G-07 Not using the named return variables when a function returns, wastes deployment gas
  - G-08 Can make the variable outside the loop to save gas
  - G-09 `require()`/`revert()` strings longer than 32 bytes cost extra gas
  - G-10 `require()` or `revert()` statements should be used sorted from cheapest to most expensive
  - G-11 Use a more recent version of Solidity

- G-12 Using `calldata` instead of `memory` for read-only arguments in external functions saves gas
- G-13 Internal functions only called once can be inlined to save gas
- G-14 Public functions not called by the contract should be declared external instead
- G-15 Should use arguments instead of state variable
- G-16 Use assembly to check for `address(0)`
- G-17 Before some functions, we should check some variables for possible gas save
- G-18 Instead of calculating a statevar with `keccak256()` every time the contract is made pre calculate them before and only give the result to a constant

- Disclosures

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the prePO smart contract system written in Solidity. The audit contest took place between December 9—December 12 2022.

Wardens

71 Wardens contributed reports to the prePO contest:

1. 0Kage
2. 0x52

3. 0xAgro
4. 0xNazgul
5. 0xSmartContract
6. 0xTraub
7. 0xdeadbeef0x
8. 0xhacksmithh
9. 8olidity
10. Awesome
11. Aymen0909
12. Bnke0x0
13. Englave
14. HE1M
15. Janio
16. KingNFT
17. Koolex
18. Madalad
19. Mukund
20. Parth
21. RHaO-sec
22. RaymondFam
23. ReyAdmirado
24. Rolezn
25. Sathish9098
26. SmartSek (0xDjango and hake)
27. Tointer
28. Tomio
29. Tricko
30. Trust
31. UdarTeam (ahmedov and tourist)

32. Udsen
33. Zarf
34. adriro
35. ak1
36. aviggiano
37. ayeslick
38. bin2chen
39. caventa
40. cccz
41. chaduke
42. csanuragjain
43. deliriusz
44. dharma09
45. fs0c
46. gz627
47. haku
48. hansfriese
49. hihen
50. idkwhatimdoing
51. imare
52. izhelyazkov
53. joestakey
54. martin
55. mert_eren
56. nadin
57. neumo
58. obront
59. oyc_109
60. pavankv

61. rjs
62. rvierdiiev
63. saneryee
64. shark
65. trustindistrust
66. unforgiven
67. wait
68. yongskiws
69. zaskoh

This contest was judged by Picodes.

Final report assembled by itsmetechjay.

Summary

The C4 analysis yielded an aggregated total of 9 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 7 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 37 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 20 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 prePO contest repository, and is composed of 16 smart contracts written in the Solidity programming language and includes 971 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (2)

[H-01] griefing / blocking / delaying users to withdraw

Submitted by zaskoh, also found by unforgiven, deliriusz, rvierdiiev, and Tricko

To withdraw, a user needs to convert his collateral for the base token. This is done in the **withdraw** function in Collateral.

The WithdrawHook has some security mechanics that can be activated like a global max withdraw in a specific timeframe, also for users to have a withdraw limit for them in a specific timeframe. It also collects the fees.

The check for the user withdraw is wrongly implemented and can lead to an unepexted delay for a user with a position **> userWithdrawLimitPerPeriod**. To withdraw all his funds he needs to be the first in every first new epoch (**lastUserPeriodReset** + **userPeriodLength**) to get his amount out. If he is not the first transaction in the new epoch, he needs to wait for a complete new epoch and depending on the timeframe from **lastUserPeriodReset** + **userPeriodLength** this can get a long delay to get his funds out.

The documentation says, that after every epoch all the user withdraws will be reset and they can withdraw the next set.

```
File: apps/smart-contracts/core/contracts/interfaces/IWithdrawHo
63:   /**
64:    * @notice Sets the length in seconds for which user withd
65:    * be evaluated against. Every time `userPeriodLength` sec
66:    * amount withdrawn for all users will be reset to 0. This
```

But the implementation only resets the amount for the first user that interacts with the contract in the new epoch and leaves all other users with their old limit. This can lead to a delay for every user that is on his limit from a previous epoch until they manage to be the first to interact with the contract in the new epoch.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/WithdrawHook.sol#L66-L72

The following test shows how a user is locked out to withdraw if he's at his limit from a previous epoch and another withdraw is done before him.

apps/smart-contracts/core/test/WithdrawHook.test.ts

```
  describe('user withdraw is delayd', () => {
    beforeEach(async () => {
      await withdrawHook.setCollateral(collateral.address)
      await withdrawHook.connect(deployer).setWithdrawalsAllowed
      await withdrawHook.connect(deployer).setGlobalPeriodLength
      await withdrawHook.connect(deployer).setUserPeriodLength(T
      await withdrawHook.connect(deployer).setGlobalWithdrawLimi
      await withdrawHook.connect(deployer).setUserWithdrawLimitP
      await withdrawHook.connect(deployer).setDepositRecord(depo
      await withdrawHook.connect(deployer).setTreasury(treasury.
      await withdrawHook.connect(deployer).setTokenSender(tokenS
      await testToken.connect(deployer).mint(collateral.address,
      await testToken.connect(deployer).mint(user.address, TEST_
      await testToken.connect(deployer).mint(user2.address, TEST
      await testToken
        .connect(collateralSigner)
        .approve(withdrawHook.address, ethers.constants.MaxUint2
      tokenSender.send.returns()
    })

    it('reverts if user withdraw limit exceeded for period', asy

      // first withdraw with the limit amount for a user
      await withdrawHook.connect(collateralSigner).hook(user.adc
      expect(await withdrawHook.getAmountWithdrawnThisPeriod(use

      // we move to a new epoch in the future
      const previousResetTimestamp = await getLastTimestamp(ethe
      await setNextTimestamp(
        ethers.provider,
        previousResetTimestamp + TEST_USER_PERIOD_LENGTH + 1
      )

      // now another user is the first one to withdraw in this r
      await withdrawHook.connect(collateralSigner).hook(user2.ac
      expect(await withdrawHook.getAmountWithdrawnThisPeriod(use

      // this will revert, because userToAmountWithdrawnThisPeri
      // but it should not revert as it's a new epoch and the us
      await expect(
        withdrawHook.connect(collateralSigner).hook(user.address
      ).to.revertedWith('user withdraw limit exceeded')

    })
  })
```

To get the test running you need to add **let user2: SignerWithAddress** and the user2 in **await ethers.getSigners()**

Recommended Mitigation Steps

The check how the user periods are handled need to be changed. One possible way is to change the lastUserPeriodReset to a mapping like **mapping(address => uint256) private lastUserPeriodReset** to track the time for every user separately.

With a mapping you can change the condition to:

```
File: apps/smart-contracts/core/contracts/WithdrawHook.sol
18:   mapping(address => uint256) lastUserPeriodReset;
```

```
File: apps/smart-contracts/core/contracts/WithdrawHook.sol
    if (lastUserPeriodReset[_sender] + userPeriodLength < bl
      lastUserPeriodReset[_sender] = block.timestamp;
      userToAmountWithdrawnThisPeriod[_sender] = _amountBefo
    } else {
      require(userToAmountWithdrawnThisPeriod[_sender] + _am
      userToAmountWithdrawnThisPeriod[_sender] += _amountBef
    }
```

With this change, we can change the test to how we would normaly expect the contract to work and see that it is correct.

```
    it('withdraw limit is checked for every use seperatly', asyn

      // first withdraw with the limit amount for a user
      await withdrawHook.connect(collateralSigner).hook(user.adc

      // we move to a new epoch in the future
      const previousResetTimestamp = await getLastTimestamp(ethe
      await setNextTimestamp(
        ethers.provider,
        previousResetTimestamp + TEST_USER_PERIOD_LENGTH + 1
      )

      // now another user is the first one to withdraw in this r
      await withdrawHook.connect(collateralSigner).hook(user2.ac

      // the first user also can withdraw his limit in this epoc
      await withdrawHook.connect(collateralSigner).hook(user.adc

      // we move the time, but stay in the same epoch
      const previousResetTimestamp2 = await getLastTimestamp(eth
      await setNextTimestamp(
        ethers.provider,
        previousResetTimestamp2 + TEST_USER_PERIOD_LENGTH - 1
      )

      // this now will fail as we're in the same epoch
      await expect(
        withdrawHook.connect(collateralSigner).hook(user.address
      ).to.revertedWith('user withdraw limit exceeded')
```

```
        })
```

ramenforbreakfast (prePO) confirmed

[H-02] A whale user is able to cause freeze of funds of other users by bypassing withdraw limit

Submitted by Trust, *also found by* 0Kage, imare, hansfriese, ayeslick, rvierdiiev, bin2chen, fs0c, mert_eren, Parth, cccz, aviggiano, *and* chaduke)

https://github.com/prepo-io/prepo-monorepo/blob/3541bc704ab185a969f300e96e2f744a572a3640/apps/smart-contracts/core/contracts/WithdrawHook.sol#L61

https://github.com/prepo-io/prepo-monorepo/blob/3541bc704ab185a969f300e96e2f744a572a3640/apps/smart-contracts/core/contracts/WithdrawHook.sol#L68

Description

In Collateral.sol, users may withdraw underlying tokens using withdraw. Importantly, the withdrawal must be approved by withdrawHook if set:

```
function withdraw(uint256 _amount) external override nonReentran
  uint256 _baseTokenAmount = (_amount * baseTokenDenominator) /
  uint256 _fee = (_baseTokenAmount * withdrawFee) / FEE_DENOMINA
  if (withdrawFee > 0) { require(_fee > 0, "fee = 0"); }
  else { require(_baseTokenAmount > 0, "amount = 0"); }
  _burn(msg.sender, _amount);
  uint256 _baseTokenAmountAfterFee = _baseTokenAmount - _fee;
  if (address(withdrawHook) != address(0)) {
    baseToken.approve(address(withdrawHook), _fee);
    withdrawHook.hook(msg.sender, _baseTokenAmount, _baseTokenAn
    baseToken.approve(address(withdrawHook), 0);
  }
  baseToken.transfer(msg.sender, _baseTokenAmountAfterFee);
  emit Withdraw(msg.sender, _baseTokenAmountAfterFee, _fee);
}
```

The hook requires that two checks are passed:

```
    if (lastGlobalPeriodReset + globalPeriodLength < block.timestamp
      lastGlobalPeriodReset = block.timestamp;
      globalAmountWithdrawnThisPeriod = _amountBeforeFee;
    } else {
      require(globalAmountWithdrawnThisPeriod + _amountBeforeFee <=
      globalAmountWithdrawnThisPeriod += _amountBeforeFee;
    }
    if (lastUserPeriodReset + userPeriodLength < block.timestamp) {
      lastUserPeriodReset = block.timestamp;
      userToAmountWithdrawnThisPeriod[_sender] = _amountBeforeFee;
    } else {
      require(userToAmountWithdrawnThisPeriod[_sender] + _amountBefo
      userToAmountWithdrawnThisPeriod[_sender] += _amountBeforeFee;
    }
```

If it has been less than “globalPeriodLength” seconds since the global reset, we step into the if block, reset time becomes now and starting amount is the current requested amount. Otherwise, the new amount must not overpass the globalWithdrawLimitPerPeriod. Very similar check is done for “user” variables.

The big issue here is that the limit can be easily bypassed by the first person calling withdraw in each group (“global” and “user”). It will step directly into the if block where no check is done, and fill the variable with any input amount.

As I understand, the withdraw limit is meant to make sure everyone is guaranteed to be able to withdraw the specified amount, so there is no chance of freeze of funds. However, due to the bypassing of this check, a whale user is able to empty the current reserves put in place and cause a freeze of funds for other users, until the Collateral contract is replenished.

Impact

A whale user is able to cause freeze of funds of other users by bypassing withdraw limit.

Proof of Concept

1. Collateral.sol has 10,000 USDC reserve

2. Withdraw limit is 150 USDC per user per period
3. There are 5 users - Alpha with collateral worth 12,000 USDC, and 4 users each with 1,000 USDC
4. Alpha waits for a time when request would create a new lastGlobalPeriodReset **and** new lastUserPeriodReset. He requests a withdraw of 10,000 USDC.
5. The hook is passed and he withdraws the entire collateral reserves.
6. At this point, victim Vic is not able to withdraw their 150 USDC. It is a freeze of funds.

Recommended Mitigation Steps

Add limit checks in the if blocks as well, to make sure the first request does not overflow the limit.

Judge note

I've confirmed with the PrePO team during the contest that withdraw limit bypass is a very serious issue.

ramenforbreakfast (prePO) confirmed

Medium Risk Findings (7)

[M-01] Bypass `userWithdrawLimitPerPeriod` check

Submitted by csanuragjain

User can bypass the `userWithdrawLimitPerPeriod` check by transferring the balance to another account.

Proof of Concept

1. Assume `userWithdrawLimitPerPeriod` is set to `1000`
2. User A has current deposit of amount `2000` and wants to withdraw everything instantly
3. User A calls the withdraw function and takes out the `1000` amount

```
function withdraw(uint256 _amount) external override nonReentran
    uint256 _baseTokenAmount = (_amount * baseTokenDenominator)
    uint256 _fee = (_baseTokenAmount * withdrawFee) / FEE_DENOMI
    if (withdrawFee > 0) { require(_fee > 0, "fee = 0"); }
    else { require(_baseTokenAmount > 0, "amount = 0"); }
    _burn(msg.sender, _amount);
    uint256 _baseTokenAmountAfterFee = _baseTokenAmount - _fee;
    if (address(withdrawHook) != address(0)) {
      baseToken.approve(address(withdrawHook), _fee);
      withdrawHook.hook(msg.sender, _baseTokenAmount, _baseToker
      baseToken.approve(address(withdrawHook), 0);
    }
    baseToken.transfer(msg.sender, _baseTokenAmountAfterFee);
    emit Withdraw(msg.sender, _baseTokenAmountAfterFee, _fee);
  }
```

4. Remaining `1000` amount cannot be withdrawn since `userWithdrawLimitPerPeriod` is reached

```
function hook(
    address _sender,
    uint256 _amountBeforeFee,
    uint256 _amountAfterFee
  ) external override onlyCollateral {
...
require(userToAmountWithdrawnThisPeriod[_sender] + _amountBefore
...
}
```

5. User simply transfers his balance to his other account and withdraw from that account
6. Since withdraw limit is tied to account, this new account will be allowed to make withdrawal thus bypassing `userWithdrawLimitPerPeriod`

Recommended Mitigation Steps

User should only be allowed to transfer leftover limit. For example if User already utilized limit X then he should only be able to transfer `userWithdrawLimitPerPeriod-X`.

ramenforbreakfast (prePO) confirmed

[M-02] The recipient receives free collateral token if an ERC20 token that deducts a fee on transfer used as baseToken

Submitted by Koolex, *also found by* idkwhatimdoing, adriro, haku, SmartSek, KingNFT, *and* pavankv

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/Collateral.sol#L45-L61

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/Collateral.sol#L64-L78

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/DepositHook.sol#L49-L50

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/WithdrawHook.sol#L76-L77

Impact

- There are some ERC20 tokens that deduct a fee on every transfer call. If these tokens are used as baseToken then:

    1. When depositing into the **Collateral** contract, the recipient will receive collateral token more than what they should receive.
    2. The **DepositRecord** contract will track wrong user deposit amounts and wrong globalNetDepositAmount as the added amount to both will be always more than what was actually deposited.
    3. When withdrawing from the **Collateral** contract, the user will receive less baseToken amount than what they should receive.
    4. The treasury will receive less fee and the user will receive more `PPO` tokens that occur in **DepositHook** and **WithdrawHook**.

Proof of Concept

Given:

- baseToken is an ERC20 token that deduct a fee on every transfer call.
- **FoT** is the deducted fee on transfer.
- The user deposits baseToken to the **Collateral** contract by calling `deposit` function passing `\_amount` as 100e18.
- `baseToken.transferFrom` is called to transfer the amount from the user to the contract.
    - https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/Collateral.sol#L49
- The contract receives the _amount - **FoT**. Let's assume the FoT percentage is 1%. Therefore, the actual amount received is 99e18.
- When the **DepositHook** is called. the `\_amount` passed is 100e18 which is wrong as it should be the actual amount 99e18.
    - https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/Collateral.sol#L53
- Calculating **collateralMintAmount** is based on the `\_amount` (100e18- the fee for treasury) which will give the recipient additional collateral token that they shouldn't receive.
    - https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/Collateral.sol#L57

Recommended Mitigation Steps

1. Consider calculating the actual amount by recording the balance before and after.
    - For example:

```
uint256 balanceBefore = baseToken.balanceOf(address(this));
baseToken.transferFrom(msg.sender, address(this), _amount);
uint256 balanceAfter = baseToken.balanceOf(address(this));
```

```
    uint256 actualAmount = balanceAfter - balanceBefore;
```

2. Then use **actualAmount** instead of `\_amount` to perform any further calculations or external calls.

Note: apply the same logic for **DepositHook** and **WithdrawHook** as well at:

- https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/DepositHook.sol#L49-L50
- https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/WithdrawHook.sol#L76-L77

ramenforbreakfast (prePO) confirmed

Picodes (judge) decreased severity to Medium

[M-03] Frontrunning for unallowed minting of Short and Long tokens

Submitted by zaskoh*, also found by* UdarTeam, ak1*, and* 0xTraub

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/PrePOMarket.sol#L68

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/PrePOMarket.sol#L109

Vulnerability details

Unallowed minting of Short and Long tokens

The documentation states that minting of the Short and Long tokens should only be done by the governance.

```
File: apps/smart-contracts/core/contracts/interfaces/IPrePOMarke
73:     * Minting will only be done by the team, and thus relies
74:     * to enforce access controls. This is also why there is r
```

```
75:     * as opposed to `redeem()`.
```

The problem is, that as long as the **_mintHook** is not set via **setMintHook**, everyone can use the mint function and mint short and long tokens. At the moment the **_mintHook** is not set in the contructor of PrePOMarket and so the transaction that will set the **_mintHook** can be front run to mint short and long tokens for the attacker.

Proof of Concept

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/PrePOMarket.sol#L68

https://github.com/prepo-io/prepo-monorepo/blob/feat/2022-12-prepo/apps/smart-contracts/core/contracts/PrePOMarket.sol#L109

This test shows how an attacker could frontrun the **setMintHook** function:

```
    describe('# mint front run attack', () => {
      let mintHook: FakeContract<Contract>
      beforeEach(async () => {
        prePOMarket = await prePOMarketAttachFixture(await createM
      })
      it('user can frontrun the set setMintHook to mint long and s

        mintHook = await fakeMintHookFixture()
        await collateralToken.connect(deployer).transfer(user.addr
        await collateralToken.connect(user).approve(prePOMarket.ac

        // we expect the mintHook to always revert
        mintHook.hook.reverts()

        // attacker frontruns the setMintHook, even we expect the
        await prePOMarket.connect(user).mint(TEST_MINT_AMOUNT)

        // governance sets the mintHook
        await prePOMarket.connect(treasury).setMintHook(mintHook.a

        // as we expect minthook to revert if not called from trea
        mintHook.hook.reverts()
        await expect(prePOMarket.connect(user).mint(TEST_MINT_AMOU
        // we should now have long and short tokens in the attacke
        const longToken = await LongShortTokenAttachFixture(await
        const shortToken = await LongShortTokenAttachFixture(await
        expect(await longToken.balanceOf(user.address)).to.eq(TEST
        expect(await shortToken.balanceOf(user.address)).to.eq(TES
      })
    })
```

Recommended Mitigation Steps

To prevent the front-running, the `\_mintHook` should be set in the deployment in the PrePOMarketFactory.

You could add one more address to the createMarket that accepts the mintHook address for that deployment and just add the address after the PrePOMarket ist deployed in the Factory.

```
File: apps/smart-contracts/core/contracts/PrePOMarketFactory.sol
46:     PrePOMarket _newMarket = new PrePOMarket{salt: _salt}(_g
47:     deployedMarkets[_salt] = address(_newMarket);
```

Alternatively you could add a default MintHook-Contract address that will always revert until it's changed to a valid one.

ramenforbreakfast (prePO) acknowledged

[M-04] PrePO NFT holders will not be able to redeem collateral

Submitted by 0xdeadbeef0x

The protocol has set a limitation on who can participate in the protocol activities.

1. Users who are included in an allowed list: `_accountList`.
2. Users who own specific NFTs that are supported by NFTScoreRequirement. These NFTs are PrePO NFTs that were minted to accounts that historically participated in PrePO activities.

Users who are #2 that deposited funds into the protocol are not able to redeem collateral tokens and withdraw their profits/funds from the market. (Loss of funds).

Proof of Concept

When a user has deposited, the protocol checks if the user is permitted to participate in the protocol activities by checking #1 and #2 from the Impact section. The check is done in the `hook` function in `DepositHook`: https://github.com/prepo-io/prepo-monorepo/blob/3541bc704ab185a969f300e96e2f744a572a3640/apps/smart-contracts/core/contracts/DepositHook.sol#L45

```
  function hook(address _sender, uint256 _amountBeforeFee, uint2
-----
    if (!_accountList.isIncluded(_sender)) require(_satisfiesScc
-----
  }
```

After a user has deposited and received collateral tokens, he will trade it in uniswap pools to receive Long/Short tokens either manually or through the `depositAndTrade` function.

When the user decided to `redeem` through the market in order to receive the collateral tokens and his funds/profits, the user will not be able to receive it because only users that are in the account list (#1) will pass the checks. Users who participated because they own NFT (#2) will get a revert when calling the function.

`redeem` in `PrePOMarket`: https://github.com/prepo-io/prepo-monorepo/blob/3541bc704ab185a969f300e96e2f744a572a3640/apps/smart-contracts/core/contracts/PrePOMarket.sol#L96

```
  function redeem(uint256 _longAmount, uint256 _shortAmount) ext
-----
      _redeemHook.hook(msg.sender, _collateralAmount, _collatera
-----
  }
```

`hook` function in `RedeemHook`: https://github.com/prepo-io/prepo-monorepo/blob/3541bc704ab185a969f300e96e2f744a572a3640/apps/smart-contracts/core/contracts/RedeemHook.sol#L18

```
  function hook(address sender, uint256 amountBeforeFee, uint256
    require(_accountList.isIncluded(sender), "redeemer not allow
----
  }
```

As you can see above, only users that are in the account list will be able to redeem. NFT holders will receive a revert of “redeemer not allowed”.

Hardhat POC

There is an already implemented test where `hook` will revert if the user is not in the allowed list: https://github.com/prepo-io/prepo-monorepo/blob/3541bc704ab185a969f300e96e2f744a572a3640/apps/smart-contracts/core/test/RedeemHook.test.ts#L128

```
    it('reverts if caller not allowed', async () => {
      msgSendersAllowlist.isIncluded.returns(false)
      expect(await msgSendersAllowlist.isIncluded(user.address))

      await expect(redeemHook.connect(user).hook(user.address, 1
        'msg.sender not allowed'
      )
    })
```

Tools Used

VS Code, Hardhat

Recommended Mitigation Steps

Add an additional check in `DepositHook` to NFT holders through `NFTScoreRequirement`.

ramenforbreakfast (prePO) disputed and commented:

> We should have been more clear with this, but when users `redeem` their positions, we intend to simply deactivate the `AccountList`, since we only want to gate initial deposits in the system, the amount of underlying within the system. If someone has a position they'd like to redeem once AMM pools are closed, they will be able to via `redeem` since we would not have an ACL configured at that point in time.

Picodes (judge) commented:

> @ramenforbreakfast I get your point, but the finding is valid for the codebase and documentation that was audited, right? Anyone obtaining the tokens through a transfer or by minting with a NFT would not be able to redeem with the audited hook.

ramenforbreakfast (prePO) commented:

> I think another problem with this finding, is it assumes that the `AccountList` used for `DepositHook` (for minting `Collateral`) and `RedeemHook` for `PrePOMarket` will be the same. In the diagram provided, each `AccountList` is denoted with a number to clarify that these are different `AccountList` instances.

> The `RedeemHook` `AccountList` is a separate list, that will essentially only allow `governance` to redeem positions via the market contract directly, *until* the UniswapV3 pool is closed and final redemptions are allowed by all users.

> While I agree that this finding would have been valid given we did not document this well, I think the assumption that these lists would be the same and affect one another *was* documented and this issue is incorrectly assuming otherwise.

Picodes (judge) commented:

> I don't think this issue assumes that the two `AccountList` are exactly the same, it just highlights that some users may be allowed to mint but not to redeem, which would lead to a loss of funds.

> They could still trade their tokens with someone whitelisted but the risk that they have to take a loss or are left unable to redeem is high.

> However, you're right to highlight that technically the governance could add every NFT holder in the redeem `AccountList`, so could mitigate this.

> Also, to clarify @ramenforbreakfast, do you intend to deactivate the `AccountList` by removing the `redeemHook`? Or is there a way in `redeemHook` to deactivate the `AccountList`?

[ramenforbreakfast (prePO) commented]():

> For the `redeemHook`, once trading on the UniswapV3Pool is closed, we would most likely replace the `redeemHook` with one that does not have an `AccountList`, therefore allowing with existing positions to redeem them via the `PrePOMarket` contract.

> I understand that this person is highlighting that some people would be allowed to mint, but not redeem, but users do not mint positions anyway. They buy them from UniswapV3 Pools with `Collateral` that they mint. Liquidity on the pools is provided by the team, who are the only ones allowed to mint positions for a `PrePOMarket`.

[Picodes (judge) decreased severity to Medium and commented]():

> My decision will be to accept this finding but downgrade it to Medium, considering that:

- the audited scope didn't contain a version of `redeemHook` without `AccountList`
- if users don't mint but buy using Uniswap, they still cannot redeem if there is some form of whitelisting required
- as highlighted by the sponsor the `AccountList` isn't supposed to be the same as the `mint` one
- funds aren't lost but stuck until the governance updates the `AccountList`

[M-05] `PrePOMarket.setFinalLongPayout()` shouldn't be called twice.

Submitted by hansfriese, also found by Trust, unforgiven, and cccz

If `finalLongPayout` is changed twice by admin fault, the market would be insolvent as it should pay more collateral than it has.

Proof of Concept

If `finalLongPayout` is less than `MAX_PAYOUT`, it means the market is ended and `longToken Price = finalLongPayout, shortToken Price = MAX_PAYOUT - finalLongPayout`.

So when users redeem their long/short tokens, the total amount of collateral tokens will be the same as the amount that users transferred during `mint()`.

Btw in `setFinalLongPayout()`, there is no validation that this function can't be called twice and the below scenario would be possible.

1. Let's assume there is one user `Bob` in the market for simplicity.
2. `Bob` transferred 100 amounts of `collateral` and got 100 long/short tokens. The market has 100 `collateral`.
3. The market admin set `finalLongPayout = 60 * 1e16` and `Bob` redeemed 100 `longToken` and received 60 `collateral`. The market has 40 `collateral` now.
4. After that, the admin realized `finalLongPayout` is too high and changed `finalLongPayout = 40 * 1e16` again.
5. `Bob` tries to redeem 100 `shortToken` and receive 60 `collateral` but the market can't offer as it has 40 `collateral` only.

When there are several users in the market, some users can't redeem their long/short tokens as the market doesn't have enough `collaterals`.

Recommended Mitigation Steps

We should modify `setFinalLongPayout()` like below so it can't be finalized twice.

```
    function setFinalLongPayout(uint256 _finalLongPayout) external
```

```
        require(finalLongPayout > MAX_PAYOUT, "Finalized already");

        require(_finalLongPayout >= floorLongPayout, "Payout cannot
        require(_finalLongPayout <= ceilingLongPayout, "Payout cannc
        finalLongPayout = _finalLongPayout;
        emit FinalLongPayoutSet(_finalLongPayout);
      }
```

[ramenforbreakfast (prePO) confirmed]

[M-06] Manager can get around min reserves check, draining all funds from Collateral.sol

Submitted by [obront], *also found by* [joestakey], [Trust], [wait], [Madalad], [hansfriese], [deliriusz], [rvierdiiev], [HE1M], [8olidity], [zaskoh], [hihen], [cccz], *and* [csanuragjain]

When a manager withdraws funds from Collateral.sol, there is a check in the `managerWithdrawHook` to confirm that they aren't pushing the contract below the minimum reserve balance.

```
    require(collateral.getReserve() - _amountAfterFee >= getMinReser
```

However, a similar check doesn't happen in the `withdraw()` function.

The manager can use this flaw to get around the reserve balance by making a large deposit, taking a manager withdrawal, and then withdrawing their deposit.

Proof of Concept

Imagine a situation where the token has a balance of 100, deposits of 1000, and a reserve percentage of 10%. In this situation, the manager should not be able to make any withdrawal.

But, with the following series of events, they can:

- Manager calls `deposit()` with 100 additional tokens

- Manager calls `managerWithdraw()` to pull 100 tokens from the contract
- Manager calls `withdraw()` to remove the 100 tokens they added

The result is that they are able to drain the balance of the contract all the way to zero, avoiding the intended restrictions.

Recommended Mitigation Steps

Include a check on the reserves in the `withdraw()` function as well as `managerWithdraw()`.

Picodes (judge) commented:

> From what I understand, although it's not clear from the documentation or the code, this `minReserve` requirement is here to keep some funds in the contract to allow for withdrawals but does not provide any additional safety, and it should be clear for users that a compromised manager would immediately lead to a loss of all funds.

Picodes (judge) commented:

> I'll merge all issues regarding the manager being able to withdraw all funds, regardless of the method, the core issue being that the managerWithdrawHook check is easily bypassable.

ramenforbreakfast (prePO) confirmed

Picodes (judge) decreased severity to Medium and commented:

> Flagging as best for this centralization issue, combined with the other finding by the same warden #255

[M-07] Users do not receive owed tokens if `TokenSender` contract cannot cover their owed amount.

Submitted by trustindistrust, also found by Trust, fs0c, imare, and chaduke

The `TokenSender.send()` method is called during the course of users withdrawing or redeeming tokens from the protocol. The method is called via `DepositHook.hook()`, `RedeemHook.hook()`, and `WithdrawHook.hook()`. These in turn are called in `prePOMarket.redeem()` or `Collateral.deposit()|.withdraw()` `TokenSender.send()` contains some logic to return early without sending any of the "outputToken", such as if the price of the outputToken has fallen below an adjustable lower bound, or if the amount would be 0.However, it also checks its own balance to see if it can cover the required amount. If it cannot, it simply doesn't send tokens. These tokens are intended to be a compensation for fees paid elsewhere in the process, and thus do represent a value loss.

```
function send(address recipient, uint256 unconvertedAmount) exte
    uint256 scaledPrice = (_price.get() * _priceMultiplier) / MU
    if (scaledPrice <= _scaledPriceLowerBound) return;
    uint256 outputAmount = (unconvertedAmount * _outputTokenDeci
    if (outputAmount == 0) return;
    if (outputAmount > _outputToken.balanceOf(address(this))) re
    _outputToken.transfer(recipient, outputAmount);
}
```

The documentation in `ITokenSender.sol` states this is so the protocol doesn't halt the redeem and deposit/withdraw actions.

Impact

The warden agrees that the protocol halting is generally undesirable.

However, there isn't any facility in the code for the user who triggered the overage amount to be able to later receive their tokens when the contract is topped up. They must rely upon governance to send them any owed tokens. This increases centralization risks and isn't necessary.

Since the contract makes no attempt to track the tokens that should have been sent, manually reviewing and verifying owed tokens becomes a non-trivial task if any more than a handful of users were affected.

Since the user did receive their underlying collateral in any case and the loss isn't necessarily permanent, medium seems to be the right severity for this issue.

Proof of Concept

Bob wants to redeem his long and short tokens via `PrePOMarket.redeem()`. However, Alice's redemption prior to his, significantly drained the `TokenSender` contract of its tokens. As a result, Bob's redemption fails to benefit him in the amount of the outputToken he should have received in compensation for the fees paid.

Because the quantity of tokens paid to Bob is partially dependent upon the token's price at the time of redemption, the protocol might shoulder more downside loss (token price dropped compared to when Bob redeemed, must pay out more tokens) or Bob might suffer upside loss (price went up compared to time of redemption, Bob loses the difference).

Bob's recourse is to contact the project administrators and try to have his tokens sent to him manually. Agreeing to a value adds friction to the process.

Recommended Mitigation Steps

The `TokenSender` contract should track users whose balance wasn't covered in a mapping, as well as a function for them to manually claim tokens later on if the contract's balance is topped up.

Such a function might record the price at redemption time, or it might calculate it with the current price.

Picodes (judge) commented:

> In my understanding, if the refunds are all used, they aren't owed anything and pay the regular fees. The design seems to be that there is a way to give temporary discounts or fee refunds but it is not mandatory.

ramenforbreakfast (prePO) disagreed with severity and commented:

> I believe this issue is a duplicate of #311, although this one is more fleshed out and would consider as the primary issue. I agree with lowering this to QA since there is no expectation of rewards, and the frontend would be able to reliably inform the

> user whether they would receive a rebate, nothing is being misrepresented on-chain. Token rewards are only a possible incentive, not an owed liability, to users.

Picodes (judge) commented:

> The front-end cannot properly mitigate the possibility of front running and someone taking all the available rebates before a user transaction. Therefore, in my opinion, due to the lack of safety checks, Medium severity is appropriate as a user could think he'd receive a rebate but won't receive it.

Low Risk and Non-Critical Issues

For this contest, 37 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xSmartContract** received the top score from the judge.

The following wardens also submitted reports: joestakey, Zarf, Aymen0909, 0x52, Udsen, 0Kage, yongskiws, 0xNazgul, caventa, gz627, trustindistrust, neumo, obront, deliriusz, izhelyazkov, wait, idkwhatimdoing, rvierdiiev, 0xAgro, Janio, Rolezn, Awesome, Mukund, shark, Englave, Tointer, oyc_109, Parth, RaymondFam, SmartSek, csanuragjain, UdarTeam, Bnke0x0, chaduke, 0xTraub, *and* 0xhacksmithh.

Low Risk Issues Summary

Number	Issues Details	Context
[L-01]	Missing calls to `__ReentrancyGuard_init` functions of inherited contracts	1
[L-02]	Draft Openzeppelin Dependencies	2
[L-03]	There is a risk that the `proxyFee` variable is accidentally initialized to 0 and platform loses money	4
[L-04]	Owner can renounce Ownership	3
[L-05]	Missing Event for critical parameters init and change	6
[L-06]	A single point of failure	
[L-07]	Using vulnerable dependency of OpenZeppelin	1
[L-08]	Loss of precision due to rounding	1

Total: 8 issues

[L-01] Missing calls to `__ReentrancyGuard_init` functions of inherited contracts

Most contracts use the delegateCall proxy pattern and hence their implementations require the use of `initialize()` functions instead of constructors. This requires derived contracts to call the corresponding init functions of their inherited base contracts. This is done in most places except a few.

Impact

The inherited base classes do not get initialized which may lead to undefined behavior.

Missing call to `__ReentrancyGuard_init` : Collateral.sol#L35-L37

PrePOMarketFactory.sol#L16

Recommended Mitigation Steps

Add missing calls to init functions of inherited contracts.

```
     function initialize(string memory _name, string memory _symbol)
        __SafeAccessControlEnumerable_init();
        __ERC20_init(_name, _symbol);
        __ERC20Permit_init(_name);
+       __ReentrancyGuard_init();
      }
```

[L-02] Draft OpenZeppelin Dependencies

The `Collateral.sol` and `DepositTradeHelper.sol` contracts utilised `draft-IERC20Permit.sol` and `draftERC20PermitUpgradeable.sol` , an OpenZeppelin contracts. These contracts are still a draft and are not considered ready for mainnet use.

OpenZeppelin contracts may be considered draft contracts if they have not received adequate security auditing or are liable to change with future development.

```
2 results - 2 files

/Collateral.sol:
5: import "@openzeppelin/contracts-upgradeable/token/ERC20/exter

/DepositTradeHelper.sol:
  6: import "@openzeppelin/contracts/token/ERC20/extensions/draf
```

[L-03] There is a risk that the `proxyFee` variable is accidentally initialized to 0 and platform loses money

There is a risk that the `fees` variables are accidentally initialized to 0.

Starting `setWithdrawFee` with 0 is an administrative decision, but since there is no information about this in the documentation and NatSpec comments during the audit, we can assume that it will not be 0, also other fee setter functions.

In addition, it is a strong belief that it will not be 0, as it is an issue that will affect the platform revenues.

Although the value initialized with 0 by mistake or forgetting can be changed later by onlyOwner/onlyRole, in the first place it can be exploited by users and cause a huge amount of usage.

`require == 0` should not be made because of the possibility of the platform defining the `0` rate.

```
4 files

/Collateral.sol:
   90:   function setDepositFee(uint256 _newDepositFee) external
   91      require(_newDepositFee <= FEE_LIMIT, "exceeds fee lin

   96:   function setWithdrawFee(uint256 _newWithdrawFee) externa
   97      require(_newWithdrawFee <= FEE_LIMIT, "exceeds fee li
  126:   function setRedemptionFee(uint256 _redemptionFee) exter
  127      require(_redemptionFee <= FEE_LIMIT, "Exceeds fee lin

/TokenSender.sol:
  45:   function setPrice(IUintValue price) external override or
  46      _price = price;
```

[L-04] Owner can renounce Ownership

```
6 results - 6 files

/DepositTradeHelper.sol:
  5: import "prepo-shared-contracts/contracts/SafeOwnable.sol";
  8: contract DepositTradeHelper is IDepositTradeHelper, SafeOwr

/LongShortToken.sol:
  5: import "@openzeppelin/contracts/access/Ownable.sol";
  8: contract LongShortToken is ERC20Burnable, Ownable {

/MintHook.sol:
   8: import "prepo-shared-contracts/contracts/SafeOwnable.sol";
  10: contract MintHook is IMarketHook, AllowedMsgSenders, Accou

/PrePOMarket.sol:
   7: import "@openzeppelin/contracts/access/Ownable.sol";
  10: contract PrePOMarket is IPrePOMarket, Ownable, Reentrancy(

/PrePOMarketFactory.sol:
   8: import "@openzeppelin/contracts-upgradeable/access/Ownable
  12: contract PrePOMarketFactory is IPrePOMarketFactory, Ownabl

/RedeemHook.sol:
   9: import "prepo-shared-contracts/contracts/SafeOwnable.sol";
  11: contract RedeemHook is IMarketHook, AllowedMsgSenders, Acc
```

Typically, the contract's owner is the account that deploys the contract. As a result, the owner is able to perform certain privileged activities.

The OpenZeppelin's Ownable used in this project contract implements renounceOwnership. This can represent a certain risk if the ownership is renounced for any other reason than by design.

Renouncing ownership will leave the contract without an owner, thereby removing any functionality that is only available to the owner.

`onlyOwner` functions;

```
13 results - 5 files

/LongShortToken.sol:
  10
  11:   function mint(address _recipient, uint256 _amount) exter
  12  }

/MintHook.sol:
  17
  18:   function setAllowedMsgSenders(IAccountList allowedMsgSer
  19
  20:   function setAccountList(IAccountList accountList) public
  21  }

/PrePOMarket.sol:
  108
  109:   function setMintHook(IMarketHook mintHook) external ove
  110      _mintHook = mintHook;

  113
  114:   function setRedeemHook(IMarketHook redeemHook) external
  115      _redeemHook = redeemHook;

  118
  119:   function setFinalLongPayout(uint256 _finalLongPayout) e
  120      require(_finalLongPayout >= floorLongPayout, "Payout

  125
  126:   function setRedemptionFee(uint256 _redemptionFee) exter
  127      require(_redemptionFee <= FEE_LIMIT, "Exceeds fee lin

/PrePOMarketFactory.sol:
  21
  22:   function createMarket(string memory _tokenNameSuffix, st
  23     require(validCollateral[_collateral], "Invalid collate

  35
  36:   function setCollateralValidity(address _collateral, bool
  37      validCollateral[_collateral] = _validity;

/RedeemHook.sol:
  25
  26:   function setAllowedMsgSenders(IAccountList allowedMsgSen
  27
  28:   function setAccountList(IAccountList accountList) public
  29
  30:   function setTreasury(address _treasury) public override
  31
  32:   function setTokenSender(ITokenSender tokenSender) public
  33  }
```

Recommended Mitigation Steps

We recommend either reimplementing the function to disable it or to clearly specify if it is part of the contract design.

[L-05] Missing Event for critical parameters init and change

```
6 results - 6 files

/Collateral.sol:
  28
  29:   constructor(IERC20 _newBaseToken, uint256 _newBaseTokenD
  30      baseToken = _newBaseToken;

/DepositRecord.sol:
  22
  23:   constructor(uint256 _newGlobalNetDepositCap, uint256 _ne
  24      globalNetDepositCap = _newGlobalNetDepositCap;

/DepositTradeHelper.sol:
  13
  14:   constructor(ICollateral collateral, ISwapRouter swapRout
  15      _collateral = collateral;

/LongShortToken.sol:
  8  contract LongShortToken is ERC20Burnable, Ownable {
  9:   constructor(string memory name_, string memory symbol_) E
  10
```

```
/PrePOMarket.sol:
  41     */
  42:   constructor(address _governance, address _collateral, II
  43       require(_ceilingLongPayout > _floorLongPayout, "Ceilir

/TokenSender.sol:
  30
  31:   constructor(IERC20Metadata outputToken) {
  32       _outputToken = outputToken;
```

Events help non-contract tools to track changes, and events prevent users from being surprised by changes

Recommended Mitigation Steps

Add Event-Emit.

[L-06] A single point of failure

```
6 results - 6 files

/DepositTradeHelper.sol:
  5: import "prepo-shared-contracts/contracts/SafeOwnable.sol";
  8: contract DepositTradeHelper is IDepositTradeHelper, SafeOwr

/LongShortToken.sol:
  5: import "@openzeppelin/contracts/access/Ownable.sol";
  8: contract LongShortToken is ERC20Burnable, Ownable {

/MintHook.sol:
   8: import "prepo-shared-contracts/contracts/SafeOwnable.sol";
  10: contract MintHook is IMarketHook, AllowedMsgSenders, Accou

/PrePOMarket.sol:
   7: import "@openzeppelin/contracts/access/Ownable.sol";
  10: contract PrePOMarket is IPrePOMarket, Ownable, Reentrancy(

/PrePOMarketFactory.sol:
   8: import "@openzeppelin/contracts-upgradeable/access/Ownable
  12: contract PrePOMarketFactory is IPrePOMarketFactory, Ownabl
```

```
/RedeemHook.sol:
   9: import "prepo-shared-contracts/contracts/SafeOwnable.sol";
  11: contract RedeemHook is IMarketHook, AllowedMsgSenders, Acc
```

Impact

The `owner` role has a single point of failure and `onlyOwner` can use critical a few functions.

`owner` role in the project:

Owner is not behind a multisig and changes are not behind a timelock.

Even if protocol admins/developers are not malicious there is still a chance for Owner keys to be stolen. In such a case, the attacker can cause serious damage to the project due to important functions. In such a case, users who have invested in project will suffer high financial losses.

`onlyOwner` functions;

```
13 results - 5 files

/LongShortToken.sol:
  10
  11:   function mint(address _recipient, uint256 _amount) exter
  12  }

/MintHook.sol:
  17
  18:   function setAllowedMsgSenders(IAccountList allowedMsgSer
  19
  20:   function setAccountList(IAccountList accountList) public
  21  }

/PrePOMarket.sol:
  108
  109:   function setMintHook(IMarketHook mintHook) external ove
  110       _mintHook = mintHook;

  113
  114:   function setRedeemHook(IMarketHook redeemHook) external
  115        _redeemHook = redeemHook;

  118
  119:    function setFinalLongPayout(uint256 _finalLongPayout) e
  120       require(_finalLongPayout >= floorLongPayout, "Payout

  125
  126:    function setRedemptionFee(uint256 _redemptionFee) exter
  127       require(_redemptionFee <= FEE_LIMIT, "Exceeds fee lin

/PrePOMarketFactory.sol:
  21
  22:   function createMarket(string memory _tokenNameSuffix, st
  23      require(validCollateral[_collateral], "Invalid collate

  35
  36:   function setCollateralValidity(address _collateral, bool
  37      validCollateral[_collateral] = _validity;

/RedeemHook.sol:
  25
  26:   function setAllowedMsgSenders(IAccountList allowedMsgSer
  27
  28:   function setAccountList(IAccountList accountList) public
  29
  30:   function setTreasury(address _treasury) public override
  31
  32:   function setTokenSender(ITokenSender tokenSender) public
  33  }
```

This increases the risk of `A single point of failure`

Recommended Mitigation Steps

Add a time lock to critical functions. Admin-only functions that change critical parameters should emit events and have timelocks.

Events allow capturing the changed parameters so that off-chain tools/interfaces can register such changes with timelocks that allow users to evaluate them and consider if they would like to engage/exit based on how they perceive the changes as affecting the trustworthiness of the protocol or profitability of the implemented financial services.

Allow only multi-signature wallets to call the function to reduce the likelihood of an attack.

https://twitter.com/danielvf/status/1572963475101556738?s=20&t=V1kvzfJlsx-D2hfnG0OmuQ

Also detail them in documentation and NatSpec comments.

[L-07] Using vulnerable dependency of OpenZeppelin

The package.json configuration file says that the project is using 4.7.3 of OZ which has a not last update version:

```
1 result - 1 file

packages/prepo-shared-contracts/package.json:
  26    },
  27:   "dependencies": {
  28:     "@openzeppelin/contracts": "4.7.3",
  29:     "@openzeppelin/contracts-upgradeable": "4.7.3",
```

Recommended Mitigation Steps

Use patched versions.

Latest non vulnerable version 4.8.0.

[L-09] Loss of precision due to rounding

```
/Collateral.sol:
45:   function deposit(address _recipient, uint256 _amount) exte
  46:     uint256 _fee = (_amount * depositFee) / FEE_DENOMINAT
```

Non-Critical Issues Summary

Number	Issues Details	Context
[N-01]	Critical Address Changes Should Use Two-step Procedure	5
[N-02]	Initial value check is missing in Set Functions	4
[N-03]	Use a single file for all system-wide constants	43
[N-04]	NatSpec comments should be increased in contracts	All Contracts
[N-05]	Function writing that does not comply with the Solidity Style Guide	All Contracts
[N-06]	Add a timelock to critical functions	5
[N-07]	Use a more recent version of Solidity	All Contracts
[N-08]	Solidity compiler optimizations can be problematic	
[N-09]	Include return parameters in NatSpec comments	All Contracts
[N-10]	Omissions in Events	11
[N-11]	Long lines are not suitable for the Solidity Style Guide	6
[N-12]	Avoid *shadowing* inherited state variables	1
[N-13]	Open TODOs	1
[N-14]	Constant values such as a call to keccak256(), should used to immutable rather than constant	35
[N-15]	Mark visibility of initialize(...) functions as external	2

Total: 15 issues

[N-01] Critical Address Changes Should Use Two-step Procedure

The critical procedures should be a two-step process.

```
5 results
```

```
/Collateral.sol:
   84
   85:    function setManager(address _newManager) external overr
   86       manager = _newManager;

/ManagerWithdrawHook.sol:
  18
  19:    function setCollateral(ICollateral _newCollateral) exter
  20       collateral = _newCollateral;

  24:    function setDepositRecord(IDepositRecord _newDepositReco
  25       depositRecord = _newDepositRecord;

/TokenSenderCaller.sol:
  11:    function setTreasury(address treasury) public virtual ov
  12       _treasury = treasury;


/WithdrawHook.sol:
  116:    function setTreasury(address _treasury) public override
  117       super.setTreasury(_treasury);
```

Recommended Mitigation Steps

Lack of two-step procedure for critical operations leaves them error-prone. Consider adding a two- step procedure on the critical functions.

[N-02] Initial value check is missing in Set Functions

```
4 results

/Collateral.sol:
   90:    function setDepositFee(uint256 _newDepositFee) external
   91       require(_newDepositFee <= FEE_LIMIT, "exceeds fee lin

   96:    function setWithdrawFee(uint256 _newWithdrawFee) extern
   97       require(_newWithdrawFee <= FEE_LIMIT, "exceeds fee li

  126:    function setRedemptionFee(uint256 _redemptionFee) exter
  127       require(_redemptionFee <= FEE_LIMIT, "Exceeds fee lin

/TokenSender.sol:
  45:   function setPrice(IUintValue price) external override or
  46      _price = price;
```

Checking whether the current value and the new value are the same should be added.

[N-03] Use a single file for all system-wide constants

There are many addresses and constants used in the system. It is recommended to put the most used ones in one file (for example constants.sol, use inheritance to access these values):

This will help with readability and easier maintenance for future changes.

constants.sol

Use and import this file in contracts that require access to these values. This is just a suggestion, in some use cases this may result in higher gas usage in the distribution

```
43 results - 8 files

/Collateral.sol:
  18
  19:   uint256 public constant FEE_DENOMINATOR = 1000000;
  20:   uint256 public constant FEE_LIMIT = 100000;
  21:   bytes32 public constant MANAGER_WITHDRAW_ROLE = keccak25
  22:   bytes32 public constant SET_MANAGER_ROLE = keccak256("Co
  23:   bytes32 public constant SET_DEPOSIT_FEE_ROLE = keccak256
  24:   bytes32 public constant SET_WITHDRAW_FEE_ROLE = keccak25
  25:   bytes32 public constant SET_DEPOSIT_HOOK_ROLE = keccak25
  26:   bytes32 public constant SET_WITHDRAW_HOOK_ROLE = keccak2
  27:   bytes32 public constant SET_MANAGER_WITHDRAW_HOOK_ROLE =
  28

/DepositHook.sol:
  16
  17:   bytes32 public constant SET_COLLATERAL_ROLE = keccak256(
  18:   bytes32 public constant SET_DEPOSIT_RECORD_ROLE = keccak
  19:   bytes32 public constant SET_DEPOSITS_ALLOWED_ROLE = kecc
  20:   bytes32 public constant SET_ACCOUNT_LIST_ROLE = keccak25
  21:   bytes32 public constant SET_REQUIRED_SCORE_ROLE = keccak
  22:   bytes32 public constant SET_COLLECTION_SCORES_ROLE = kec
  23:   bytes32 public constant REMOVE_COLLECTIONS_ROLE = keccak
  24:   bytes32 public constant SET_TREASURY_ROLE = keccak256("I
  25:   bytes32 public constant SET_TOKEN_SENDER_ROLE = keccak25
  26

/DepositRecord.sol:
  13
  14:   bytes32 public constant SET_GLOBAL_NET_DEPOSIT_CAP_ROLE
  15:   bytes32 public constant SET_USER_DEPOSIT_CAP_ROLE = kecc
  16:   bytes32 public constant SET_ALLOWED_HOOK_ROLE = keccak25
  17

/DepositTradeHelper.sol:
  11    ISwapRouter private immutable _swapRouter;
  12:   uint24 public constant override POOL_FEE_TIER = 10000;
  13

/ManagerWithdrawHook.sol:
  11
  12:   uint256 public constant PERCENT_DENOMINATOR = 1000000;
  13:   bytes32 public constant SET_COLLATERAL_ROLE = keccak256(
  14:   bytes32 public constant SET_DEPOSIT_RECORD_ROLE = keccak
  15:   bytes32 public constant SET_MIN_RESERVE_PERCENTAGE_ROLE
  16

/PrePOMarket.sol:
  28
  29:   uint256 private constant MAX_PAYOUT = 1e18;
  30:   uint256 private constant FEE_DENOMINATOR = 1000000;
  31:   uint256 private constant FEE_LIMIT = 100000;
  32

/TokenSender.sol:
  24
  25:   uint256 public constant MULTIPLIER_DENOMINATOR = 10000;
  26:   bytes32 public constant SET_PRICE_ROLE = keccak256("Toke
  27:   bytes32 public constant SET_PRICE_MULTIPLIER_ROLE = kecc
  28:   bytes32 public constant SET_SCALED_PRICE_LOWER_BOUND_ROI
  29:   bytes32 public constant SET_ALLOWED_MSG_SENDERS_ROLE = k
  30

/WithdrawHook.sol:
  21
  22:   bytes32 public constant SET_COLLATERAL_ROLE = keccak256(
  23:   bytes32 public constant SET_DEPOSIT_RECORD_ROLE = keccak
```

```
bytes32 public constant SET_WITHDRAWALS_ALLOWED_ROLE = k
bytes32 public constant SET_GLOBAL_PERIOD_LENGTH_ROLE =
bytes32 public constant SET_USER_PERIOD_LENGTH_ROLE = ke
bytes32 public constant SET_GLOBAL_WITHDRAW_LIMIT_PER_PE
bytes32 public constant SET_USER_WITHDRAW_LIMIT_PER_PERI
bytes32 public constant SET_TREASURY_ROLE = keccak256("W
bytes32 public constant SET_TOKEN_SENDER_ROLE = keccak25

```

[N-04] NatSpec comments should be increased in contracts

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation.

In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommended Mitigation Steps

NatSpec comments should be increased in contracts.

[N-05] Function writing that does not comply with the Solidity Style Guide

Order of Functions; ordering helps readers identify which functions they can call and to find the constructor and fallback definitions easier. But there are contracts in the project that do not comply with this.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html

Functions should be grouped according to their visibility and ordered:

- constructor
- receive function (if exists)
- fallback function (if exists)

- external
- public
- internal
- private
- within a grouping, place the view and pure functions last

[N-06] Add a timelock to critical functions

It is a good practice to give time for users to react and adjust to critical changes. A timelock provides more guarantees and reduces the level of trust required, thus decreasing risk for users. It also indicates that the project is legitimate (less risk of a malicious owner making a sandwich attack on a user).

Consider adding a timelock to:

```
5 results

/Collateral.sol:
   84
   85:    function setManager(address _newManager) external overr
   86       manager = _newManager;

/ManagerWithdrawHook.sol:
  18
  19:    function setCollateral(ICollateral _newCollateral) exter
  20       collateral = _newCollateral;

  23
  24:    function setDepositRecord(IDepositRecord _newDepositReco
  25       depositRecord = _newDepositRecord;

/TokenSenderCaller.sol:
  11:    function setTreasury(address treasury) public virtual ov
  12       _treasury = treasury;


/WithdrawHook.sol:
  116:    function setTreasury(address _treasury) public override
  117       super.setTreasury(_treasury);
```

[N-07] Use a more recent version of Solidity

For security, it is best practice to use the latest Solidity version.

For the security fix list in the versions:
https://github.com/ethereum/solidity/blob/develop/Changelog.md

Recommended Mitigation Steps

Old version of Solidity is used, newer version can be used `(0.8.17)` .

[N-08] Solidity compiler optimizations can be problematic

```
packages/prepo-shared-contracts/hardhat.config.ts:

const config: HardhatUserConfig = {
  ...hardhatConfig,
  solidity: {
    compilers: [
      {
        version: '0.8.7',
        settings: {
          optimizer: {
            enabled: true,
            runs: 99999,
          },
        },
```

Protocol has enabled optional compiler optimizations in Solidity.

There have been several optimization bugs with security implications. Moreover, optimizations are actively being developed. Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them.

Therefore, it is unclear how well they are being tested and exercised.

High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten-generated solc-js compiler used by Truffle and

Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG.

Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. More recently, another bug due to the incorrect caching of keccak256 was reported. A compiler audit of Solidity from November 2018 concluded that the optional optimizations may not be safe.

It is likely that there are latent bugs related to optimization and that new bugs will be introduced due to future optimizations.

Exploit Scenario

A latent or future bug in Solidity compiler optimizations—or in the Emscripten transpilation to solc-js—causes a security vulnerability in the contracts.

Recommended Mitigation Steps

Short term, measure the gas savings from optimizations and carefully weigh them against the possibility of an optimization-related bug.

Long term, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

[N-09] Include return parameters in NatSpec comments

It is recommended that Solidity contracts are fully annotated using NatSpec for all public interfaces (everything in the ABI). It is clearly stated in the Solidity official documentation. In complex projects such as Defi, the interpretation of all functions and their arguments and returns is important for code readability and auditability.

https://docs.soliditylang.org/en/v0.8.15/natspec-format.html

Recommended Mitigation Steps

Include return parameters in NatSpec comments:

Recommendation Code Style: (from Uniswap3)

```
/// @notice Adds liquidity for the given recipient/tickLowe
/// @dev The caller of this method receives a callback in tl
/// in which they must pay any token0 or token1 owed for the
/// on tickLower, tickUpper, the amount of liquidity, and tl
/// @param recipient The address for which the liquidity wil
/// @param tickLower The lower tick of the position in which
/// @param tickUpper The upper tick of the position in which
/// @param amount The amount of liquidity to mint
/// @param data Any data that should be passed through to tl
/// @return amount0 The amount of token0 that was paid to mi
/// @return amount1 The amount of token1 that was paid to mi
function mint(
    address recipient,
    int24 tickLower,
    int24 tickUpper,
    uint128 amount,
    bytes calldata data
) external returns (uint256 amount0, uint256 amount1);
```

[N-10] Omissions in Events

Throughout the codebase, events are generally emitted when sensitive changes are made to the contracts. However, some events are missing important parameters.

The events should include the new value and old value where possible:

```
11 results

/AccountListCaller.sol:
   9
  10:   function setAccountList(IAccountList accountList) public
  11:     _accountList = accountList;
  12:     emit AccountListChange(accountList);
  13:   }

/AllowedMsgSenders.sol:
  14
  15:   function setAllowedMsgSenders(IAccountList allowedMsgSer
  16:     _allowedMsgSenders = allowedMsgSenders;
  17:     emit AllowedMsgSendersChange(allowedMsgSenders);
  18:   }
```

```
/Collateral.sol:

  function setManager(address _newManager) external overr
    manager = _newManager;
    emit ManagerChange(_newManager);
  }

  function setDepositFee(uint256 _newDepositFee) external
    require(_newDepositFee <= FEE_LIMIT, "exceeds fee lin
    depositFee = _newDepositFee;
    emit DepositFeeChange(_newDepositFee);
  }

  function setWithdrawFee(uint256 _newWithdrawFee) extern
    require(_newWithdrawFee <= FEE_LIMIT, "exceeds fee li
    withdrawFee = _newWithdrawFee;
    emit WithdrawFeeChange(_newWithdrawFee);
  }

  function setDepositHook(ICollateralHook _newDepositHook
    depositHook = _newDepositHook;
    emit DepositHookChange(address(_newDepositHook));
  }

  function setWithdrawHook(ICollateralHook _newWithdrawHo
    withdrawHook = _newWithdrawHook;
    emit WithdrawHookChange(address(_newWithdrawHook));
  }

  function setManagerWithdrawHook(ICollateralHook _newMar
    managerWithdrawHook = _newManagerWithdrawHook;
    emit ManagerWithdrawHookChange(address(_newManagerWit
  }


/DepositHook.sol:

  function setCollateral(ICollateral _newCollateral) exter
    collateral = _newCollateral;
    emit CollateralChange(address(_newCollateral));
  }

  function setDepositRecord(IDepositRecord _newDepositRecc
    depositRecord = _newDepositRecord;
    emit DepositRecordChange(address(_newDepositRecord));
  }
  function setDepositsAllowed(bool _newDepositsAllowed) ex
    depositsAllowed = _newDepositsAllowed;
    emit DepositsAllowedChange(_newDepositsAllowed);
  }
```

[N-11] Long lines are not suitable for the Solidity Style Guide

Collateral.sol#L9

Collateral.sol#L112

DepositHook.sol#L69-L79

DepositRecord.sol#L40

DepositTradeHelper.sol#L22-L32

RedeemHook.sol#L26

It is generally recommended that lines in the source code should not exceed 80-120 characters. Today's screens are much larger, so in some cases it makes sense to expand that. The lines above should be split when they reach that length, as the files will most likely be on GitHub and GitHub always uses a scrollbar when the length is more than 164 characters.

See why-is-80-characters-the-standard-limit-for-code-width.

Recommended Mitigation Steps

Multiline output parameters and return statements should follow the same style recommended for wrapping long lines found in the Maximum Line Length section.

https://docs.soliditylang.org/en/v0.8.17/style-guide.html#introduction

```
thisFunctionCallIsReallyLong(
    longArgument1,
    longArgument2,
    longArgument3
);
```

[N-12] Avoid *shadowing* inherited state variables

```
/DepositHook.sol:
79:   function setTokenSender(ITokenSender _tokenSender) public

/TokenSenderCaller.sol:
  7: contract TokenSenderCaller is ITokenSenderCaller {
  9:   ITokenSender internal _tokenSender;
```

`_tokenSender` is shadowed.

Recommended Mitigation Steps

Avoid using variables with the same name, including inherited in the same contract. If used, it must be specified in the NatSpec comments.

[N-13] Open TODOs

```
/TokenSender.sol:
  14    AllowedMsgSenders,
  15:   WithdrawERC20, // TODO: Access control when WithdrawERC2
```

Recommended Mitigation Steps

Use temporary TODOs as you work on a feature, but make sure to treat them before merging. Either add a link to a proper issue in your TODO, or remove it from the code.

[N-14] Constant values such as a call to `keccak256()` , should use immutable rather than constant

There is a difference between constant variables and immutable variables, and they should each be used in their appropriate contexts.

While it doesn't save any gas because the compiler knows that developers often make this mistake, it's still best to use the right tool for the task at hand.

Constants should be used for literal values written into the code, and immutable variables should be used for expressions, or values calculated in, or passed into the constructor.

```
35 results - 6 files

/Collateral.sol:
  21:   bytes32 public constant MANAGER_WITHDRAW_ROLE = keccak25
  22:   bytes32 public constant SET_MANAGER_ROLE = keccak256("Co
  23:   bytes32 public constant SET_DEPOSIT_FEE_ROLE = keccak256
  24:   bytes32 public constant SET_WITHDRAW_FEE_ROLE = keccak25
  25:   bytes32 public constant SET_DEPOSIT_HOOK_ROLE = keccak25
  26:   bytes32 public constant SET_WITHDRAW_HOOK_ROLE = keccak2
  27:   bytes32 public constant SET_MANAGER_WITHDRAW_HOOK_ROLE =

/DepositHook.sol:
  17:   bytes32 public constant SET_COLLATERAL_ROLE = keccak256(
  18:   bytes32 public constant SET_DEPOSIT_RECORD_ROLE = keccak
  19:   bytes32 public constant SET_DEPOSITS_ALLOWED_ROLE = kecc
  20:   bytes32 public constant SET_ACCOUNT_LIST_ROLE = keccak25
  21:   bytes32 public constant SET_REQUIRED_SCORE_ROLE = keccak
  22:   bytes32 public constant SET_COLLECTION_SCORES_ROLE = kec
  23:   bytes32 public constant REMOVE_COLLECTIONS_ROLE = keccak
  24:   bytes32 public constant SET_TREASURY_ROLE = keccak256("I
  25:   bytes32 public constant SET_TOKEN_SENDER_ROLE = keccak25

/DepositRecord.sol:
  14:   bytes32 public constant SET_GLOBAL_NET_DEPOSIT_CAP_ROLE
  15:   bytes32 public constant SET_USER_DEPOSIT_CAP_ROLE = kecc
  16:   bytes32 public constant SET_ALLOWED_HOOK_ROLE = keccak25

/ManagerWithdrawHook.sol:
  13:   bytes32 public constant SET_COLLATERAL_ROLE = keccak256(
  14:   bytes32 public constant SET_DEPOSIT_RECORD_ROLE = keccak
  15:   bytes32 public constant SET_MIN_RESERVE_PERCENTAGE_ROLE

/TokenSender.sol:
  26:   bytes32 public constant SET_PRICE_ROLE = keccak256("Toke
  27:   bytes32 public constant SET_PRICE_MULTIPLIER_ROLE = kecc
  28:   bytes32 public constant SET_SCALED_PRICE_LOWER_BOUND_ROI
  29:   bytes32 public constant SET_ALLOWED_MSG_SENDERS_ROLE = k
```

```
/WithdrawHook.sol:
    bytes32 public constant SET_COLLATERAL_ROLE = keccak256
    bytes32 public constant SET_DEPOSIT_RECORD_ROLE = keccak
    bytes32 public constant SET_WITHDRAWALS_ALLOWED_ROLE = k
    bytes32 public constant SET_GLOBAL_PERIOD_LENGTH_ROLE =
    bytes32 public constant SET_USER_PERIOD_LENGTH_ROLE = ke
    bytes32 public constant SET_GLOBAL_WITHDRAW_LIMIT_PER_PE
    bytes32 public constant SET_USER_WITHDRAW_LIMIT_PER_PERI
    bytes32 public constant SET_TREASURY_ROLE = keccak256("W
    bytes32 public constant SET_TOKEN_SENDER_ROLE = keccak25
```

[N-15] Mark visibility of initialize(...) functions as external

```
/Collateral.sol:

    function initialize(string memory _name, string memory _
      __SafeAccessControlEnumerable_init();
```

```
/PrePOMarketFactory.sol:

    function initialize() public initializer { OwnableUpgrad

```

If someone wants to extend via inheritance, it might make more sense that the overridden `initialize(...)` function calls the `internal {...}_init` function, not the parent public `initialize(...)` function.

External instead of public would give more sense of the `initialize(...)` functions to behave like a constructor (only called on deployment, so should only be called externally).

From a security point of view, it might be safer so that it cannot be called internally by accident in the child contract.

It might cost a bit less gas to use external over public.

It is possible to override a function from external to public (= “opening it up”) ✅ but it is not possible to override a function from public to external (= “narrow it down”). ❌

For the above reasons, you can change `initialize(...)` to external:

https://github.com/OpenZeppelin/openzeppelin-contracts/issues/3750

Suggestions

Number	Suggestion Details
[S-01]	Make the Test Context with Solidity
[S-02]	Project Upgrade and Stop Scenario
[S-03]	Generate perfect code headers every time

Total: 3 suggestions

[S-01] Make the Test Context with Solidity

It’s crucial to write tests with possibly 100% coverage for smart contract systems.

It is recommended to write appropriate tests for all possible code streams and especially for extreme cases.

But the other important point is the test context.

Tests written with Solidity are safer, so it is recommended to focus on tests with Foundry.

[S-02] Project Upgrade and Stop Scenario

At the start of the project, the system may need to be stopped or upgraded. I suggest you have a script beforehand and add it to the documentation.

This can also be called an “EMERGENCY STOP (CIRCUIT BREAKER) PATTERN “.

https://github.com/maxwoe/solidity_patterns/blob/master/security/EmergencyStop.sol

[S-03] Generate perfect code headers every time

I recommend using header for Solidity code layout and readability:

https://github.com/transmissions11/headers

```
/*//////////////////////////////////////////////////////////////
                           TESTING 123
//////////////////////////////////////////////////////////////*/
```

ramenforbreakfast (prePO) commented:

> The following are things I spotted while reviewing:

- **L-07:** I am not sure why that is here given that even in our architecture diagram, ownership is behind a governance multi-sig? We already plan to govern via a multi-sig.
- **N-06:** We plan to use a timelock in production.
- **N-12:** I don't think I caught this one, but our repo does follow < 80 char width guidelines, the formatting was changed to > 80 specifically for the audit repo.

Picodes (judge) commented:

- **L-07:** Report is valid (there is a centralization risk), although it is not stated in the repo that owner role isn't behind a multi-sig so this can be amended for the report.
- **N-12:** Per the sponsor comments seems invalid on their main repo, but valid on the audit repo.

Gas Optimizations

For this contest, 20 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **ReyAdmirado** received the top score from the judge.

The following wardens also submitted reports: 0xSmartContract, Aymen0909, rjs, gz627, dharma09, RHaO-sec, Mukund, Tomio, Rolezn, Englave, martin, nadin, saneryee, RaymondFam, 0xTraub, Sathish9098, UdarTeam, pavankv, *and* chaduke *.*

Gas Optimizations Summary

	issue
G-01	Expressions for constant values such as a call to `keccak256()`, should use immutable rather than constant
G-02	Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate
G-03	State variables should be cached in stack variables rather than re-reading them from storage
G-04	Stack variable used as a cheaper cache for a state variable is only used once
G-05	Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` statement
G-06	`<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables
G-07	Not using the named return variables when a function returns, wastes deployment gas
G-08	Can make the variable outside the loop to save gas
G-09	`require()/revert()` strings longer than 32 bytes cost extra gas
G-10	`require() or revert()` statements that check input arguments should be at the top of the function]
G-11	Use a more recent version of solidity]
G-12	Using `calldata` instead of `memory` for read-only arguments in external functions saves gas
G-13	Internal functions only called once can be inlined to save gas
G-1	Public functions not called by the contract should be declared external instead

	issue
4	
G-15	Should use arguments instead of state variable
G-16	Use assembly to check for address(0)
G-17	Before some functions we should check some variables for possible gas save
G-18	Instead of calculating a statevar with `keccak256()` every time the contract is made pre calculate them before and only give the result to a constant

[G-01] Expressions for constant values such as a call to `keccak256()`, should use immutable rather than constant

- TokenSender.sol#L26
- TokenSender.sol#L27
- TokenSender.sol#L28
- TokenSender.sol#L29
- WithdrawHook.sol#L22
- WithdrawHook.sol#L23
- WithdrawHook.sol#L24
- WithdrawHook.sol#L25
- WithdrawHook.sol#L26
- WithdrawHook.sol#L27
- WithdrawHook.sol#L28
- WithdrawHook.sol#L29
- WithdrawHook.sol#L30
- DepositRecord.sol#L14
- DepositRecord.sol#L15
- DepositRecord.sol#L16
- Collateral.sol#L21
- Collateral.sol#L22

- Collateral.sol#L23
- Collateral.sol#L24
- Collateral.sol#L25
- Collateral.sol#L26
- Collateral.sol#L27
- DepositHook.sol#L17
- DepositHook.sol#L18
- DepositHook.sol#L19
- DepositHook.sol#L20
- DepositHook.sol#L21
- DepositHook.sol#L22
- DepositHook.sol#L23
- DepositHook.sol#L24
- DepositHook.sol#L25
- ManagerWithdrawHook.sol#L13
- ManagerWithdrawHook.sol#L14
- ManagerWithdrawHook.sol#L15

[G-02] Multiple address/ID mappings can be combined into a single mapping of an address/ID to a struct, where appropriate

Saves a storage slot for the mapping. Depending on the circumstances and sizes of types, can avoid a Gsset (20000 gas) per mapping combined. Reads and subsequent writes can also be cheaper when a function requires both values and they both fit in the same storage slot. Finally, if both fields are accessed in the same function, can save ~42 gas per access due to not having to recalculate the key's keccak256 hash (Gkeccak256 - 30 gas) and that calculation's associated stack operations.

`validCollateral` and `deployedMarkets` used together in a function once and can be placed in a single slot:

- [PrePOMarketFactory.sol#L13-L14]()

[G-03] State variables should be cached in stack variables rather than re-reading them from storage

The instances below point to the second+ access of a state variable within a function. Caching of a state variable replace each Gwarmaccess (100 gas) with a much cheaper stack read. Other less obvious fixes/optimizations include having local memory caches of state variable structs, or having local caches of state variable contracts/addresses.

Most of the times this if statement will be true and we will save 100 gas at a small possibility of 3 gas loss , so cache `_mintHook` before the if statement:

- [PrePOMarket.sol#L68]()

`finalLongPayout`

- [PrePOMarket.sol#L82]()

`redemptionFee`

- [PrePOMarket.sol#L90]()

`_redeemHook` cache it before the if statement because its gonna usually be true (the value changes further in function should take actions accordingly)

- [PrePOMarket.sol#L93]()

`collateral`

- [WithdrawHook.sol#L76]()

`depositFee` cache before #L46

- [Collateral.sol#L46]()

`depositHook` cache before if statement

- [Collateral.sol#L51]()

`withdrawFee` cache before #L66

- [Collateral.sol#L66]()

`withdrawHook` cache before if statement

- [Collateral.sol#L71]()

`managerWithdrawHook`

- [Collateral.sol#L81]()

`collateral`

- [DepositHook.sol#L49]()

if `_requiredScore` is not equal to 0, `_requiredScore` will be read twice from storage consider caching it before the line

- [NFTScoreRequirement.sol#L14]()

[G-04] Stack variable used as a cheaper cache for a state variable is only used once

If the variable is only accessed once, it's cheaper to use the state variable directly that one time, and save the 3 gas the extra stack assignment would spend:

`_shortPayout`

- [PrePOMarket.sol#L82]()
- [PrePOMarketFactory.sol#L42-L45]()

[G-05] Add `unchecked {}` for subtractions where the operands cannot underflow because of a previous `require()` or `if` statement

`require(a <= b); x = b - a => require(a <= b); unchecked { x = b - a } if(a <= b); x = b - a => if(a <= b); unchecked { x = b - a }`

This will stop the check for overflow and underflow so it will save gas.

This is checked in the if statement #L81

- [PrePOMarket.sol#L82]

This can underflow because its withdraw and can not be higher than the original value

- [WithdrawHook.sol#L76]
- [Collateral.sol#L50]
- [Collateral.sol#L70]

[G-06] `<x> += <y>` costs more gas than `<x> = <x> + <y>` for state variables

Using the addition operator instead of plus-equals saves gas

- [WithdrawHook.sol#L64]
- [WithdrawHook.sol#L71]
- [DepositRecord.sol#L31]
- [DepositRecord.sol#L32]
- [DepositRecord.sol#L36]

[G-07] Not using the named return variables when a function returns, wastes deployment gas

Do not use return at the end of the function:

- [PrePOMarketFactory.sol#L41]

[G-08] Can make the variable outside the loop to save gas

Consider making the stack variables before the loop which gonna save gas

2 instances here `collection` and `collectionScore`

- NFTScoreRequirement.sol#L59

[G-09] `require()/revert()` strings longer than 32 bytes cost extra gas

Each extra memory word of bytes past the original 32 incurs an MSTORE which costs 3 gas

- NFTScoreRequirement.sol#L23

[G-10] `require()` or `revert()` statements should be used sorted from cheapest to most expensive

Checks that involve constants should come before checks that involve state variables, function calls, and calculations. By doing these checks first, the function is able to revert before wasting a Gcoldsload (2100 gas*) in a function that may ultimately revert in the unhappy case.

Swap the position of these 2 requires for possible gas save:

- PrePOMarket.sol#L77-L78

Swap the condition of `if and else statements` which will result in checking the cheaper condition first, possibly save gas because we check a argument instead of a state var(100 gas save)

- Collateral.sol#L47-L48

Same logic as the last one

- Collateral.sol#L67-L68

[G-11] Use a more recent version of Solidity

Use a Solidity version of at least 0.8.10 to have `external` calls skip contract existence checks if the external call has a return value.

Use a Solidity version of at least 0.8.12 to get `string.concat()` to be used instead of `abi.encodePacked(<str>,<str>)`.

Use a solidity version of at least 0.8.13 to get the ability to use `using for` with a list of free functions.

[G-12] Using `calldata` instead of `memory` for read-only arguments in external functions saves gas

When a function with a memory array is called externally, the `abi.decode()` step has to use a for-loop to copy each index of the calldata to the memory index. Each iteration of this for-loop costs at least 60 gas (i.e. 60 * <mem_array>.length). Using calldata directly, obliviates the need for such a loop in the contract code and runtime execution.

2 instances in this line

- PrePOMarketFactory.sol#L22

```
setCollectionScores
```

- NFTScoreRequirement.sol#L22

```
removeCollections
```

- NFTScoreRequirement.sol#L35

```
removeCollections
```

- DepositHook.sol#L75

```
setCollectionScores
```

- DepositHook.sol#L73

[G-13] Internal functions only called once can be inlined to save gas

Not inlining costs 20 to 40 gas because of two extra JUMP instructions and additional stack operations needed for function calls.

```
_createPairTokens
```

- PrePOMarketFactory.sol#L41

[G-14] Public functions not called by the contract should be declared external instead

Contracts are allowed to override their parents' functions and change the visibility from external to public and can save gas by doing so.

`setAllowedMsgSenders`

- AllowedMsgSenders.sol#L15

`setAccountList`

- AccountListCaller.sol#L10

`setRequiredScore`

- NFTScoreRequirement.sol#L17

`setCollectionScores`

- NFTScoreRequirement.sol#L22

`removeCollections`

- NFTScoreRequirement.sol#L35

[G-15] Should use arguments instead of state variable

This will save near 97 gas

- DepositRecord.sol#L42

[G-16] Use assembly to check for `address(0)`

Saves 6 gas per instance:

- PrePOMarket.sol#L68

- PrePOMarket.sol#L93
- Collateral.sol#L51
- Collateral.sol#L71
- Collateral.sol#L81

[G-17] Before some functions, we should check some variables for possible gas save

Before transfer, we should check for amount being 0 so the function doesnt run when its not gonna do anything:

Check `_amount`

- PrePOMarket.sol#L69

Check `_collateralAfterFee`

- PrePOMarket.sol#L104

`baseTokenAmount`

- DepositTradeHelper.sol#L26

`_collateralAmountMinted`

- DepositTradeHelper.sol#L31

`_amount`

- Collateral.sol#L82

[G-18] Instead of calculating a statevar with `keccak256()` every time the contract is made pre calculate them before and only give the result to a constant

- TokenSender.sol#L26
- TokenSender.sol#L27

- TokenSender.sol#L28
- TokenSender.sol#L29
- WithdrawHook.sol#L22
- WithdrawHook.sol#L23
- WithdrawHook.sol#L24
- WithdrawHook.sol#L25
- WithdrawHook.sol#L26
- WithdrawHook.sol#L27
- WithdrawHook.sol#L28
- WithdrawHook.sol#L29
- WithdrawHook.sol#L30
- DepositRecord.sol#L14
- DepositRecord.sol#L15
- DepositRecord.sol#L16
- Collateral.sol#L21
- Collateral.sol#L22
- Collateral.sol#L23
- Collateral.sol#L24
- Collateral.sol#L25
- Collateral.sol#L26
- Collateral.sol#L27
- DepositHook.sol#L17
- DepositHook.sol#L18
- DepositHook.sol#L19
- DepositHook.sol#L20
- DepositHook.sol#L21
- DepositHook.sol#L22
- DepositHook.sol#L23
- DepositHook.sol#L24

- DepositHook.sol#L25
- ManagerWithdrawHook.sol#L13
- ManagerWithdrawHook.sol#L14
- ManagerWithdrawHook.sol#L15

Disclosures