prePO contest
Findings & Analysis Report

2022-05-09

Table of contents

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 March 17—March 19 2022.

Wardens

49 Wardens contributed reports to the prePO contest:

1. kirk-baird
2. cmichel
3. WatchPug (jtp and ming)
4. leastwood
5. csanuragjain
6. Ruhum
7. IllIllI
8. GreyArt (hickuphh3 and itsmeSTYJ)
9. rayn
10. CertoraInc (danb, egjlmn1, OriDabush, ItayG, and shakedwinder)
11. 0xDjango
12. defsec
13. TomFrenchBlockchain
14. Dravee
15. hake
16. TerrierLover
17. robee
18. 0xkatana
19. berndartmueller
20. 0x1f8b

21. Funen
22. kenta
23. 0xNazgul
24. saian
25. 0xwags
26. minhquanym
27. samruna
28. rfa
29. Kenshin
30. oyc_109
31. sorrynotsorry
32. bugwriter001
33. cccz
34. GeekyLumberjack
35. peritoflores
36. remora
37. wuwe1
38. Jujic
39. 0xngndev
40. Tadashi
41. Kiep
42. 0v3rf10w
43. Tomio

This contest was judged by gzeon.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 8 unique vulnerabilities. Of these vulnerabilities, 3 received a risk rating in the category of HIGH severity and 5 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 35 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 29 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 18 smart contracts written in the Solidity programming language and includes 1,207 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 (3)

[H-01] Strategy Migration May Leave Tokens in the Old Strategy Impacting Share Calculations

Submitted by kirk-baird

If a strategy does not have sufficient funds to `withdraw()` for the full amount then it is possible that tokens will be left in this yield contract during `migrate()` .

It is common for withdrawal from a strategy to withdraw less than a user's balance. The reason is that these yield protocols may lend the deposited funds to borrowers, if there is less funds in the pool than the withdrawal amount the withdrawal may succeed but only transfer the funds available rather than the full withdrawal amount.

The impact of tokens remaining in the old strategy is that when we call `StrategyController.totalValue()` this will only account for the tokens deposited in the new strategy and not those stuck in the previous strategy. Therefore `totalValue()` is undervalued.

Thus, when a user calls `Collateral.deposit()` the share calculations `_shares = (_amountToDeposit * totalSupply()) / (_valueBefore);` will be over stated (note: `uint256 _valueBefore = _strategyController.totalValue();`). Hence, the user will receive more shares than they should.

The old tokens may be recovered by calling `migrate()` back to the old strategy. If this is done then `totalValue()` will now include the tokens previously stuck. The recent depositer may now withdraw and will be owed `(_strategyController.totalValue() * _amount) / totalSupply()` . Since `totalValue()` is now includes the previously stuck tokens `_owed` will be overstated and the user will receive more collateral than they should.

The remaining users who had deposited before `migrate()` will lose tokens proportional to their share of the `totalSupply()` .

Proof of Concept

[SingleStrategyController.sol#L51-L72]

```
        function migrate(IStrategy _newStrategy)
            external
            override
            onlyOwner
            nonReentrant
        {
            uint256 _oldStrategyBalance;
            IStrategy _oldStrategy = _strategy;
            _strategy = _newStrategy;
            _baseToken.approve(address(_newStrategy), type(uint256).
            if (address(_oldStrategy) != address(0)) {
                _baseToken.approve(address(_oldStrategy), 0);
                _oldStrategyBalance = _oldStrategy.totalValue();
                _oldStrategy.withdraw(address(this), _oldStrategyBal
                _newStrategy.deposit(_baseToken.balanceOf(address(th
            }
            emit StrategyMigrated(
                address(_oldStrategy),
                address(_newStrategy),
                _oldStrategyBalance
            );
        }
```

Recommended Mitigation Steps

The recommendation is to ensure that `require(_oldStrategy.totalValue() == 0)` after calling `_oldStrategy.withdraw()`. This ensures that no funds are left in the strategy. Consider the code example below.

```
        function migrate(IStrategy _newStrategy)
            external
            override
            onlyOwner
            nonReentrant
        {
            uint256 _oldStrategyBalance;
            IStrategy _oldStrategy = _strategy;
            _strategy = _newStrategy;
            _baseToken.approve(address(_newStrategy), type(uint256).
            if (address(_oldStrategy) != address(0)) {
                _baseToken.approve(address(_oldStrategy), 0);
                _oldStrategyBalance = _oldStrategy.totalValue();
                _oldStrategy.withdraw(address(this), _oldStrategyBal
                require(_oldStrategy.totalValue() == 0)
                _newStrategy.deposit(_baseToken.balanceOf(address(th
            }
            emit StrategyMigrated(
                address(_oldStrategy),
                address(_newStrategy),
                _oldStrategyBalance
            );
        }
```

ramenforbreakfast (prePO) confirmed and commented:

> This is a valid claim, although it is an edge case. I will maintain the severity of this issue as is. Should consider removing a fixed migration procedure altogether as this issue demonstrates one of the many problems that can occur.

gzeon (judge) commented:

> Agree with sponsor.

[H-02] First depositor can break minting of shares

Submitted by GreyArt, also found by 0xDjango, CertoraInc, cmichel, rayn, TomFrenchBlockchain, and WatchPug

Collateral.sol#L82-L91

The attack vector and impact is the same as TOB-YEARN-003, where users may not receive shares in exchange for their deposits if the total asset amount has been manipulated through a large “donation”.

Proof of Concept

- Attacker deposits 2 wei (so that it is greater than min fee) to mint 1 share
- Attacker transfers exorbitant amount to `_strategyController` to greatly inflate the share’s price. Note that the `_strategyController` deposits its entire balance to the strategy when its `deposit()` function is called.

- Subsequent depositors instead have to deposit an equivalent sum to avoid minting 0 shares. Otherwise, their deposits accrue to the attacker who holds the only share.

```
it("will cause 0 share issuance", async () => {
        // 1. first user deposits 2 wei because 1 wei will be de
        let firstDepositAmount = ethers.BigNumber.from(2)
        await transferAndApproveForDeposit(
            user,
            collateral.address,
            firstDepositAmount
        )

        await collateral
            .connect(user)
            .deposit(firstDepositAmount)

        // 2. do huge transfer of 1M to strategy to controller
        // to greatly inflate share price
        await baseToken.transfer(strategyController.address, etl

        // 3. deployer tries to deposit reasonable amount of 10_
        let subsequentDepositAmount = ethers.utils.parseEther("1
        await transferAndApproveForDeposit(
            deployer,
            collateral.address,
            subsequentDepositAmount
        )

        await collateral
            .connect(deployer)
            .deposit(subsequentDepositAmount)

        // receives 0 shares in return
        expect(await collateral.balanceOf(deployer.address)).to.
});
```

Recommended Mitigation Steps

- Uniswap V2 solved this problem by sending the first 1000 LP tokens to the zero address. The same can be done in this case i.e. when `totalSupply() ==`

`0`, send the first min liquidity LP tokens to the zero address to enable share dilution.
- Ensure the number of shares to be minted is non-zero: `require(_shares != 0, "zero shares minted");`
- Create a periphery contract that contains a wrapper function that atomically calls `initialize()` and `deposit()`
- Call `deposit()` once in `initialize()` to achieve the same effect as the suggestion above.

ramenforbreakfast (prePO) confirmed and commented:

> Valid submission, good explanation of the problem and nice to see it being demonstrated via a test case block.

gzeon (judge) commented:

> Agree with sponsor.

[H-03] Withdrawal delay can be circumvented

Submitted by cmichel, also found by IllIIl and leastwood

Collateral.sol#L97

After initiating a withdrawal with `initiateWithdrawal`, it's still possible to transfer the collateral tokens. This can be used to create a second account, transfer the accounts to them and initiate withdrawals at a different time frame such that one of the accounts is always in a valid withdrawal window, no matter what time it is. If the token owner now wants to withdraw they just transfer the funds to the account that is currently in a valid withdrawal window.

Also, note that each account can withdraw the specified `amount`. Creating several accounts and circling & initiating withdrawals with all of them allows withdrawing larger amounts **even at the same block** as they are purchased in the future.

I consider this high severity because it breaks core functionality of the Collateral token.

Proof of Concept

For example, assume the `_delayedWithdrawalExpiry = 20` blocks. Account A owns 1000 collateral tokens, they create a second account B.

- At `block=0`, A calls `initiateWithdrawal(1000)`. They send their balance to account B.
- At `block=10`, B calls `initiateWithdrawal(1000)`. They send their balance to account A.
- They repeat these steps, alternating the withdrawal initiation every 10 blocks.
- One of the accounts is always in a valid withdrawal window (`initiationBlock < block && block <= initiationBlock + 20`). They can withdraw their funds at any time.

Recommended Mitigation Steps

If there's a withdrawal request for the token owner (`_accountToWithdrawalRequest[owner].blockNumber > 0`), disable their transfers for the time.

```
// pseudo-code not tested
beforeTransfer(from, to, amount) {
  super();
  uint256 withdrawalStart =  _accountToWithdrawalRequest[from].b
  if(withdrawalStart > 0 && withdrawalStart + _delayedWithdrawal
    revert(); // still in withdrawal window
  }
}
```

ramenforbreakfast (prePO) commented:

> This is a valid claim.

gzeon (judge) commented:

> Agree with sponsor.

Medium Risk Findings (5)

[M-01] Duplicate `_tokenNameSuffix` and `_tokenSymbolSuffix` will incorrectly update current Market

Submitted by csanuragjain

PrePOMarketFactory.sol#L42

Impacted Function: `createMarket`.

1. Owner calls createMarket with _tokenNameSuffix S1 and _tokenSymbolSuffix S2 which creates a new market M1 with _deployedMarkets[_salt] pointing to M1. Here salt can be S which is computed using _tokenNameSuffix and _tokenSymbolSuffix
2. This market is now being used
3. After some time owner again mistakenly calls createMarket with _tokenNameSuffix S1 and _tokenSymbolSuffix S2
4. Instead of returning error mentioning that this name and symbol already exists, new market gets created. The problem here is that salt which is computed using _tokenNameSuffix and _tokenSymbolSuffix will again come as S (as in step 1) which means _deployedMarkets[_salt] will now get updated to M2. This means reference to M1 is gone

Recommended Mitigation Steps

Add below check:

```
require(_deployedMarkets[_salt]==address(0), "Market already exi
```

ramenforbreakfast (prePO) confirmed and commented:

> This is a valid submission, and since this would overwrite the existing mapping within `PrePOMarketFactory`, I think it is fair for this to remain as medium severity.

gzeon (judge) commented:

> Agree with sponsor.

[M-02] Market expiry behaviour differs in implementation and documentation

Submitted by GreyArt, also found by leastwood and rayn

prePO Docs: Expiry
PrePOMarket.sol#L145-L156

The docs say that "If a market has not settled by its expiry date, it will automatically settle at the lower bound of its Valuation Range."

However, in the implementation, the expiry date is entirely ignored. The default settlement after expiry is a 1:1 ratio of long and short token for 1 collateral token.

Impact

Should users believe that the market will settle at the lower bound, they would swap and hold long for short tokens instead of at a 1:1 ratio upon expiry. Thereafter, they would incur swap fees from having to swap some short tokens back for long tokens for redemption. User funds are also affected should long tokens are repurchased at a higher price than when they were sold.

Recommended Mitigation Steps

If the market is to settle at the lower valuation after expiry, then the following logic should be added:

```
    // market has expired
    // settle at lower bound
    if (block.timestamp > _expiryTime) {
            uint256 _shortPrice = MAX_PRICE - _floorLongPrice;
```

```
        _collateralOwed =
                (_floorLongPrice * _longAmount + _shortPrice * _
                MAX_PRICE;
    } else if (_finalLongPrice <= MAX_PRICE) {
        ...
    } else {
        ...
    }
```

Otherwise, the documentation should be updated to reflect the default behaviour of 1:1 redemption.

ramenforbreakfast (prePO) disagreed with Medium severity

ramenforbreakfast (prePO) agreed with Medium severity and commented:

> This is a valid submission, no longer disagreeing with severity as we clearly stated that expiry should be enforceable.

> This was a mistake on our part and I think we ended up not using `expiryTime` since the only thing that really mattered was if the `finalLongPrice` was set. We should decide whether to enforce it or remove it altogether.

gzeon (judge) commented:

> Agree with sponsor.

[M-03] `getSharesForAmount` returns wrong value when `totalAssets == 0`

Submitted by cmichel

The `getSharesForAmount` function returns `0` if `totalAssets == 0`.

However, if `totalSupply == 0`, the actual shares that are minted in a `deposit` are `_amount` even if `totalAssets == 0`.

Contracts / frontends that use this function to estimate their deposit when `totalSupply == 0` will return a wrong value.

Recommended Mitigation Steps

```
    function getSharesForAmount(uint256 _amount)
        external
        view
        override
        returns (uint256)
    {
+       // to match the code in `deposit`
+       if (totalSupply() == 0) return _amount;

        uint256 _totalAssets = totalAssets();
        return
            (_totalAssets > 0)
                ? ((_amount * totalSupply()) / _totalAssets)
                : 0; // @audit this should be _amount according to `
    }
```

ramenforbreakfast (prePO) confirmed and commented:

> Valid claim, I believe this is a bug in our code, still need to verify.

gzeon (judge) commented:

> Agree with sponsor.

[M-04] SingleStrategyController doesn't verify that new strategy uses the same base token

Submitted by Ruhum

When migrating from one strategy to another, the controller pulls out the funds of the old strategy and deposits them into the new one. But, it doesn't verify that both strategies use the same base token. If the new one uses a different base token, it won't "know" about the tokens it received on migration. It won't be able to deposit and transfer them. Effectively they would be lost.

The migration is done by the owner. So the owner must make a mistake and migrate to the wrong strategy by accident. In a basic protocol with 1 controller and a single active strategy managing that should be straightforward. There shouldn't be a real risk of that mistake happening. But, if you have multiple controllers running at the same time each with a different base token, it gets increasingly likelier.

According to the `IStrategy` interface, there is a function to retrieve the strategy's base token: `getBaseToken()` . I'd recommend adding a check in the `migrate()` function to verify that the new strategy uses the correct base token to prevent this issue from being possible.

Proof of Concept

[SingleStrategyController.sol#L51-L72]

[IStrategy.sol#L52]

Recommended Mitigation Steps

Add `require(_baseToken == _newStrategy.getBaseToken());` to the beginning of `migrate()` .

[ramenforbreakfast (prePO) confirmed and commented]:

> Valid claim, in addition to H-02, illustrates why we should probably not have a fixed migration function due to the complexity of such an operation.

[gzeon (judge) commented]:

> Agree with sponsor.

[M-05] Wrong formula of `getSharesForAmount()` can potentially cause fund loss when being used to calculate the `shares` to be used in `withdraw()`

Submitted by WatchPug

In `Collateral`, the getter functions `getAmountForShares()` and `getSharesForAmount()` is using `totalAssets()` instead of `_strategyController.totalValue()`, making the results can be different than the actual shares amount needed to `withdraw()` a certain amount of `_baseToken` and the amount of shares expected to get by `deposit()` a certain amount.

Specifically, `totalAssets()` includes the extra amount of `_baseToken.balanceOf(Collateral)`.

[Collateral.sol#L306-L329]()

```
    function getAmountForShares(uint256 _shares)
        external
        view
        override
        returns (uint256)
    {
        if (totalSupply() == 0) {
            return _shares;
        }
        return (_shares * totalAssets()) / totalSupply();
    }

    function getSharesForAmount(uint256 _amount)
        external
        view
        override
        returns (uint256)
    {
        uint256 _totalAssets = totalAssets();
        return
            (_totalAssets > 0)
                ? ((_amount * totalSupply()) / _totalAssets)
                : 0;
    }
```

[Collateral.sol#L339-L343]()

```
    function totalAssets() public view override returns (uint256) {
        return
            _baseToken.balanceOf(address(this)) +
            _strategyController.totalValue();
    }
```

[Collateral.sol#L137-L148]

```
    function withdraw(uint256 _amount)
        external
        override
        nonReentrant
        returns (uint256)
    {
        require(_withdrawalsAllowed, "Withdrawals not allowed");
        if (_delayedWithdrawalExpiry != 0) {
            _processDelayedWithdrawal(msg.sender, _amount);
        }
        uint256 _owed = (_strategyController.totalValue() * _amount) /
            totalSupply();
        ...
```

Proof of Concept

Given:

- `_baseToken.balanceOf(Collateral)` == 90
- `_strategyController.totalValue()` == 110
- totalSupply of shares = 100

`totalAssets()` returns: 200

`getSharesForAmount(100)` returns: 50, while `withdraw(50)` will actual only get: 55.

When `Collateral` is used by another contract that manages many users' funds, and if it's using `getSharesForAmount()` to calculate the amount of shares needed for a certain amount of underlying tokens to be withdrawn.

This issue can potentially cause fund loss to the user of that contract because it will actually send a lesser amount of `_baseToken` than expected.

Recommended Mitigation Steps

Consider changing `Collateral.totalValue()` to:

```
    function totalAssets() public view override returns (uint256) {
        return
            _strategyController.totalValue();
    }
```

ramenforbreakfast (prePO) confirmed and commented:

> Well organized and thorough submission. This is a valid claim.

gzeon (judge) commented:

> Agree with sponsor.

Low Risk and Non-Critical Issues

For this contest, 35 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **defsec** received the top score from the judge.

The following wardens also submitted reports: GreyArt, hake, 0xkatana, berndartmueller, cmichel, csanuragjain, IllIllI, rayn, 0x1f8b, 0xDjango, CertoraInc, robee, sorrynotsorry, WatchPug, 0xNazgul, 0xwags, bugwriter001, cccz, Funen, GeekyLumberjack, kenta, kirk-baird, leastwood, minhquanym, peritoflores, remora, Ruhum, saian, samruna, TerrierLover, Kenshin, oyc_109, rfa, *and* wuwe1.

[L-01] `transferOwnership` should be two step process

All contracts are inherited from OpenZeppelin's Ownable and OwnableUpgradable contract which enables the onlyOwner role to transfer ownership to another address. It's possible that the onlyOwner role mistakenly transfers ownership to the wrong address, resulting in a loss of the onlyOwner role. The current ownership transfer process involves the current owner calling Unlock.transferOwnership(). This function checks the new owner is not the zero address and proceeds to write the new owner's address into the owner's state variable. If the nominated EOA account is not a valid account, it is entirely possible the owner may accidentally transfer ownership to an uncontrolled account, breaking all functions with the onlyOwner() modifier. Lack of two-step procedure for critical operations leaves them error-prone if the address is incorrect, the new address will take on the functionality of the new role immediately

for Ex : -Alice deploys a new version of the whitehack group address. When she invokes the whitehack group address setter to replace the address, she accidentally enters the wrong address. The new address now has access to the role immediately and is too late to revert

Proof of Concept

1. Navigate to [PrePOMarket.sol#L10]().
2. The contracts have many onlyOwner function.
3. The contract is inherited from the Ownable which includes transferOwnership.

Recommended Mitigation Steps

Implement zero address check and Consider implementing a two step process where the owner nominates an account and the nominated account needs to call an acceptOwnership() function for the transfer of ownership to fully succeed. This ensures the nominated EOA account is a valid and active account.

[L-02] The Contract Should Approve(0) first

Some tokens (like USDT L199) do not work when changing the allowance from an existing non-zero allowance value. They must first be approved by zero and then the actual allowance must be approved.

```
    IERC20(token).approve(address(operator), 0);
    IERC20(token).approve(address(operator), amount);
```

Proof of Concept

Navigate to the following contract functions:

Collateral.sol#L76
SingleStrategyController.sol#L60
SingleStrategyController.sol#L62

Recommended Mitigation Steps

Approve with a zero amount first before setting the actual amount.

[L-03] Front-runnable Initializers

All contract **initializers** were missing access controls, allowing any user to initialize the contract. By front-running the contract deployers to initialize the contract, the incorrect parameters may be supplied, leaving the contract needing to be redeployed.

Proof of Concept

1. Navigate to the following contracts: Collateral.sol#L38.
2. Initialize functions does not have access control. They are vulnerable to front-running.

Recommended Mitigation Steps

While the code that can be run in contract constructors is limited, setting the owner in the contract's constructor to the `msg.sender` and adding the `onlyOwner` modifier to all **initializers** would be a sufficient level of access control.

[L-04] Use safeTransfer/safeTransferFrom consistently instead of transfer/transferFrom

It is good to add a require() statement that checks the return value of token transfers or to use something like OpenZeppelin's safeTransfer/safeTransferFrom unless one is sure the given token reverts in case of a failure. Failure to do so will cause silent failures of transfers and affect token accounting in contract.

Reference: This similar medium-severity finding from Consensys Diligence Audit of Fei Protocol.

Proof of Concept

1. Navigate to the following contract.
2. transfer/transferFrom functions are used instead of safe transfer/transferFrom on the following contracts.

PrePOMarket.sol#L121
PrePOMarket.sol#L123

Recommended Mitigation Steps

Consider using safeTransfer/safeTransferFrom or require() consistently.

[L-05] USE SAFEERC20.SAFEAPPROVE INSTEAD OF APPROVE

This is probably an oversight since SafeERC20 was imported and safeTransfer() was used for ERC20 token transfers. Nevertheless, note that approve() will fail for certain token implementations that do not return a boolean value (). Hence it is recommend to use safeApprove().

Proof of Concept

1. Navigate to SingleStrategyController.sol#L60 and SingleStrategyController.sol#L62.
2. Preview approveUnderlying function on the contract.
3. Approve function has been used instead of SafeApprove.

Recommended Mitigation Steps

Update to _token.safeApprove(spender, type(uint256).max) in the function.

[N-01] Missing zero-address check in constructors and the setter functions

Missing checks for zero-addresses may lead to infunctional protocol, if the variable addresses are updated incorrectly.

Proof of Concept

1. Navigate to the following all contract functions: PrePOMarket.sol#L44.

Recommended Mitigation Steps

Consider adding zero-address checks in the discussed constructors: require(newAddr != address(0));.

[N-02] Consider making contracts Pausable

There are many external risks so my suggestion is that you should consider making the contracts pausable, so in case of an unexpected event, the admin can pause transfers.

Recommended Mitigation Steps

Consider making contracts Pausable (OpenZeppelin/Pausable.sol).

[N-03] Missing Vault Input Validation

During the code review, It has been observed that vault is not validated. The address check should be added into the function.

Proof Of Concept

SingleStrategyController.sol#L75

Recommended Mitigation Steps

Add address check on the function.

ramenforbreakfast (prePO) disputed and commented:

> **[L-01]** Two step ownership is unnecessarily complex considering we are using timelocked execution and proposed execution can be withdrawn.
> **[N-02]** This issue is too vague to be of any use.

gzeon (judge) commented:

> I think **[L-01]** is valid because it ensures someone can sign from the new owner address.

Gas Optimizations

For this contest, 29 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: TerrierLover, robee, WatchPug, CertoraInc, IllIllI, Jujic, 0xngndev, defsec, 0x1f8b, Tadashi, Funen, Kiep, 0xkatana, kenta, 0v3rf10w, GreyArt, berndartmueller, 0xNazgul, saian, rfa, Kenshin, Tomio, oyc_109, rayn, 0xwags, minhquanym, samruna, *and* hake.

Foreword

- **Storage-reading optimizations**

> The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). In the paragraphs below, please see the `@audit-issue` tags in the pieces of code's comments for more information about SLOADs that could be saved by caching the mentioned **storage** variables in **memory** variables.

- **Unchecking arithmetics operations that can't underflow/overflow**

> Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation, or the operation doesn't depend on user input), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

- `@audit` **tags**

> The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

File: PrePOMarket.sol

Tightly pack storage variables

Here, storage variables can be tightly packed from:

```
File: PrePOMarket.sol
    address private _treasury; //@audit 20 bytes

    IERC20 private immutable _collateral;
    ILongShortToken private immutable _longToken;
    ILongShortToken private immutable _shortToken;

    uint256 private immutable _floorLongPrice;
    uint256 private immutable _ceilingLongPrice;
    uint256 private _finalLongPrice;//@audit 32 bytes

    uint256 private immutable _floorValuation;
    uint256 private immutable _ceilingValuation;

    uint256 private _mintingFee;
    uint256 private _redemptionFee;//@audit 32 bytes

    uint256 private immutable _expiryTime;

    bool private _publicMinting; //@audit 1 byte, but takin
```

to

```
File: PrePOMarket.sol
    address private _treasury; //@audit 20 bytes

    bool private _publicMinting; //@audit 1 byte

    IERC20 private immutable _collateral;
    ILongShortToken private immutable _longToken;
    ILongShortToken private immutable _shortToken;

    uint256 private immutable _floorLongPrice;
    uint256 private immutable _ceilingLongPrice;
    uint256 private _finalLongPrice;//@audit 32 bytes

    uint256 private immutable _floorValuation;
    uint256 private immutable _ceilingValuation;

    uint256 private _mintingFee;
    uint256 private _redemptionFee;//@audit 32 bytes

    uint256 private immutable _expiryTime;
```

Which would save 1 storage slot.

function redeem()

Cache `_finalLongPrice`

Caching this in memory can save around 2 SLOADs (around 200 gas)

```
File: PrePOMarket.sol
        if (_finalLongPrice <= MAX_PRICE) { //@audit _final
            uint256 _shortPrice = MAX_PRICE - _finalLongPri
            _collateralOwed =
                (_finalLongPrice * _longAmount + _shortPric
                MAX_PRICE;
```

File: CollateralDepositRecord.sol

function recordDeposit()

```
File: CollateralDepositRecord.sol
    function recordDeposit(address _sender, uint256 _amount)
        external
        override
        onlyAllowedHooks
    {
        require(
            _amount + _globalDepositAmount <= _globalDeposit
            "Global deposit cap exceeded"
        );
        require(
            _amount + _accountToNetDeposit[_sender] <= _accc
            "Account deposit cap exceeded"
        );
        _globalDepositAmount += _amount; //@audit 2nd _amour
        _accountToNetDeposit[_sender] += _amount; //@audit 2
    }
```

Cache `_amount + _globalDepositAmount`

Cache `_amount + _accountToNetDeposit[_sender]`

Optimized code:

```
File: CollateralDepositRecord.sol
    function recordDeposit(address _sender, uint256 _amount)
        external
        override
        onlyAllowedHooks
    {
        uint256 _newGlobalDepositAmount = _amount + _globalI
        uint256 _newAccountToNetDeposit = _amount + _account
        require(
            _newGlobalDepositAmount <= _globalDepositCap,
            "Global deposit cap exceeded"
        );
        require(
            _newAccountToNetDeposit <= _accountDepositCap,
            "Account deposit cap exceeded"
        );
        _globalDepositAmount = _newGlobalDepositAmount;
        _accountToNetDeposit[_sender] = _newAccountToNetDepo
    }
```

function recordWithdrawal()

```
    function recordWithdrawal(address _sender, uint256 _amou
        external
        override
        onlyAllowedHooks
    {
        if (_globalDepositAmount > _amount) { //@audit _glok
```

```
            _globalDepositAmount -= _amount; //@audit _globa
        } else {
            _globalDepositAmount = 0;
        }
        if (_accountToNetDeposit[_sender] > _amount) { //@au
            _accountToNetDeposit[_sender] -= _amount; //@auc
        } else {
            _accountToNetDeposit[_sender] = 0;
        }
    }
```

Cache `_globalDepositAmount`

Cache `_accountToNetDeposit`

Optimized code:

```
File: CollateralDepositRecord.sol
    function recordWithdrawal(address _sender, uint256 _amou
        external
        override
        onlyAllowedHooks
    {
        uint256 __globalDepositAmount = _globalDepositAmount
        if (__globalDepositAmount > _amount) {
            _globalDepositAmount = __globalDepositAmount - _
        } else {
            _globalDepositAmount = 0;
        }
        uint256 __accountToNetDeposit = _accountToNetDeposit
        if (__accountToNetDeposit > _amount) {
            _accountToNetDeposit[_sender] = __accountToNetDe
        } else {
            _accountToNetDeposit[_sender] = 0;
        }
    }
```

General recommendations

Storage Variables

Emitting storage values instead of caching of function arguments

Emitting a storage value can be avoided to save a SLOAD at these places by using the function argument instead:

```
contracts/core/AccountAccessController.sol:
  96          _root = _newRoot;
  97:         emit RootChanged(_root); //@audit should emit _new

contracts/core/Collateral.sol:
  191          _strategyController = _newStrategyController;
  192:         emit StrategyControllerChanged(address(_strategy(

  200          _delayedWithdrawalExpiry = _newDelayedWithdrawalE
  201:         emit DelayedWithdrawalExpiryChanged(_delayedWithd

  210          _mintingFee = _newMintingFee;
  211:         emit MintingFeeChanged(_mintingFee); //@audit sho

  220          _redemptionFee = _newRedemptionFee;
  221:         emit RedemptionFeeChanged(_redemptionFee); //@aud

  229          _depositHook = _newDepositHook;
  230:         emit DepositHookChanged(address(_depositHook)); /

  238          _withdrawHook = _newWithdrawHook;
  239:         emit WithdrawHookChanged(address(_withdrawHook));

contracts/core/CollateralDepositRecord.sol:
  63          _globalDepositCap = _newGlobalDepositCap;
  64:         emit GlobalDepositCapChanged(_globalDepositCap); /
```

Incrementing before emitting a storage value instead of emitting the increment's result

These can be optimized:

```
contracts/core/AccountAccessController.sol:
   35          _blockedAccountsIndex++;
   36:         emit BlockedAccountsCleared(_blockedAccountsIndex

  101          _allowedAccountsIndex++;
  102:         emit AllowedAccountsCleared(_allowedAccountsIndex
```

to

```
contracts/core/AccountAccessController.sol:
   35:         emit BlockedAccountsCleared(++_blockedAccountsInc

  101:         emit AllowedAccountsCleared(++_allowedAccountsInc
```

Some storage variables should be immutable

Marking these as immutable (as they never change outside the constructor) would avoid them taking space in the storage:

```
contracts/core/Collateral.sol:
  23:     address private _treasury;//@audit should be immutable
  26:     IERC20Upgradeable private _baseToken; //@audit should

contracts/core/DepositHook.sol:
  11:     IAccountAccessController private _accountAccessControl
  12:     ICollateralDepositRecord private _depositRecord; //@au

contracts/core/WithdrawHook.sol:
  10:     ICollateralDepositRecord private _depositRecord; //@au
```

Help the optimizer by declaring a storage variable instead of repeatedly fetching the value

To help the optimizer, go from:

```
File: AccountAccessController.sol
61:     function allowSelf(bytes32[] calldata _proof) external c
62:         require(
63:             _allowedAccounts[_allowedAccountsIndex][msg.senc
...
69:         _allowedAccounts[_allowedAccountsIndex][msg.sender]
```

to

```
File: AccountAccessController.sol
     function allowSelf(bytes32[] calldata _proof) external over
         bool storage _accountAllowed = _allowedAccounts[_allowe
         require(
             _accountAllowed == false,
...
         _accountAllowed = true;
```

Also here, use `require(!_allowedAccounts[_allowedAccountsIndex][msg.sender])` at L63 to avoid the cost of a comparison to a constant.

For-Loops

An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
core/AccountAccessController.sol:44:        for (uint256 _i = 0;
core/AccountAccessController.sol:55:        for (uint256 _i = 0;
```

`++i` costs less gas compared to `i++`

`++i` costs less gas compared to `i++` for unsigned integer, as pre-increment is cheaper (about 5 gas per iteration)

`i++` increments `i` and returns the initial value of `i`. Which means:

```
uint i = 1;
i++; // == 1 but i == 2
```

But `++i` returns the actual incremented value:

```
uint i = 1;
++i; // == 2 and i == 2 too, so no need for a temporary variable
```

In the first case, the compiler has to create a temporary variable (when used) for returning `1` instead of `2`

Instances include:

```
core/AccountAccessController.sol:35:        _blockedAccountsInde
core/AccountAccessController.sol:44:        for (uint256 _i = 0;
core/AccountAccessController.sol:55:        for (uint256 _i = 0;
core/AccountAccessController.sol:101:        _allowedAccountsInc
```

I suggest using `++i` instead of `i++` to increment the value of an uint variable.

Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Instances include:

```
core/AccountAccessController.sol:44:        for (uint256 _i = 0;
core/AccountAccessController.sol:55:        for (uint256 _i = 0;
```

The code would go from:

```
for (uint256 i; i < numIterations; i++) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for a `uint256` here.

Arithmetics

Uncheck calculations to save gas when an overflow/underflow is impossible

Instances include:

```
contracts/core/Collateral.sol:
   71          require(_amountToDeposit > _fee, "Deposit amount
   72          _baseToken.safeTransfer(_treasury, _fee);
   73:         _amountToDeposit -= _fee; //@audit uncheck (see I

  169          require(_amountWithdrawn > _fee, "Withdrawal amou
  170          _baseToken.safeTransfer(_treasury, _fee);
  171:         _amountWithdrawn -= _fee; //@audit uncheck (see I

contracts/core/CollateralDepositRecord.sol:
  46          if (_globalDepositAmount > _amount) {
  47:             _globalDepositAmount -= _amount; //@audit unch

  51          if (_accountToNetDeposit[_sender] > _amount) {
  52:             _accountToNetDeposit[_sender] -= _amount; //@a

contracts/core/PrePOMarket.sol:
  120         require(_amount > _fee, "Minting amount too small'
  121          _collateral.transferFrom(msg.sender, _treasury, _
  122:         _amount -= _fee; //@audit uncheck (see L120)

  167          require(_collateralOwed > _fee, "Redemption amour
  168          _collateral.transfer(_treasury, _fee);
  169:         _collateralOwed -= _fee; //@audit uncheck (see L1
```

Errors

Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Instances include:

```
core/AccountAccessController.sol:62:        require(
core/AccountAccessController.sol:68:        require(MerkleProof.
core/Collateral.sol:58:        require(_depositsAllowed, "Deposi
core/Collateral.sol:71:        require(_amountToDeposit > _fee,
core/Collateral.sol:101:        require(balanceOf(msg.sender) >=
core/Collateral.sol:118:        require(
core/Collateral.sol:124:        require(
core/Collateral.sol:128:        require(
core/Collateral.sol:143:        require(_withdrawalsAllowed, "Wi
core/Collateral.sol:169:        require(_amountWithdrawn > _fee,
core/Collateral.sol:209:        require(_newMintingFee <= FEE_LI
core/Collateral.sol:219:        require(_newRedemptionFee <= FEE
core/CollateralDepositRecord.sol:15:        require(_allowedHook
core/CollateralDepositRecord.sol:29:        require(
core/CollateralDepositRecord.sol:33:        require(
core/DepositHook.sol:22:        require(msg.sender == _vault, "C
core/DepositHook.sol:31:        require(
core/PrePOMarket.sol:58:        require(
core/PrePOMarket.sol:62:        require(_newExpiryTime > block.t
core/PrePOMarket.sol:63:        require(_newMintingFee <= FEE_LI
core/PrePOMarket.sol:64:        require(_newRedemptionFee <= FEE
core/PrePOMarket.sol:65:        require(_newCeilingLongPrice <=
core/PrePOMarket.sol:108:            require(_publicMinting, "Pu
core/PrePOMarket.sol:110:        require(_finalLongPrice > MAX_I
core/PrePOMarket.sol:111:        require(
core/PrePOMarket.sol:120:        require(_amount > _fee, "Mintir
core/PrePOMarket.sol:135:        require(
core/PrePOMarket.sol:139:        require(
core/PrePOMarket.sol:151:            require(
core/PrePOMarket.sol:167:        require(_collateralOwed > _fee,
core/PrePOMarket.sol:185:        require(
core/PrePOMarket.sol:189:        require(
core/PrePOMarket.sol:202:        require(_newMintingFee <= FEE_I
core/PrePOMarket.sol:212:        require(_newRedemptionFee <= FE
core/PrePOMarketFactory.sol:55:        require(_validCollateral[
core/SingleStrategyController.sol:22:        require(msg.sender
core/SingleStrategyController.sol:27:        require(address(_to
core/WithdrawHook.sol:17:        require(msg.sender == _vault, '
```

I suggest replacing revert strings with custom errors.

ramenforbreakfast (prePO) commented:

> This one is a high quality submission that is well organized and identifies nearly every single case where such an optimization could be applied.

> The thoroughness of this submission exceeds others, but does not mention the specific amount saved.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smart contract software should be used at the sole risk and responsibility of users.

Top