



PoolTogether Findings & Analysis Report

2023-09-27

Table of contents

- [Overview](#)
 - [About C4](#)
 - [Wardens](#)
- [Summary](#)
- [Scope](#)
- [Severity Criteria](#)
- [High Risk Findings \(9\)](#)
 - [\[H-01\] The `_currentExchangeRate` of the Vault contract can't increase and will always be lower than or equal to `_assetUnit`](#)
 - [\[H-02\] A malicious user can steal other user's deposits from Vault.sol](#)
 - [\[H-03\] `_amountOut` is representing assets and shares at the same time in the `liquidate` function](#)
 - [\[H-04\] `Vault.mintYieldFee` function can be called by anyone to mint Vault Shares to any recipient address](#)
 - [\[H-05\] Delegated amounts can be forcefully removed from anyone in the `TwabController`](#)
 - [\[H-06\] Resetting delegation will result in user funds being lost forever](#)

- [\[H-07\] `_requireVaultCollateralized\(\)` is called at the beginning of the functions `mintYieldFee\(\)` and `liquidate\(\)`](#)
- [\[H-08\] Increasing reserves breaks PrizePool accounting](#)
- [\[H-09\] `Vault` is not compatible with some ERC4626 vaults](#)
- [Medium Risk Findings \(27\)](#)
 - [\[M-01\] If the underlying asset is a fee on transfer token, it could break the internal accounting of the vault](#)
 - [\[M-02\] Unintended or malicious use of prize winners' hooks](#)
 - [\[M-03\] `TwabLib::getTwabBetween` can return inaccurate balances if `_startTime` and `_endTime` aren't safely bound](#)
 - [\[M-04\] The deposit function does not check for the `maxMint` amount](#)
 - [\[M-05\] Balance invariant between the individual and total `twabs` can be broken](#)
 - [\[M-06\] `drawManager` can be set to a malicious address](#)
 - [\[M-07\] In important libraries of PoolTogether, the `pow\(\)` function of PRBMath is used, which exhibits inconsistent return values](#)
 - [\[M-08\] An attacker can front-run `deployVault` to deploy at the same address](#)
 - [\[M-09\] High Prizes might not be claimed](#)
 - [\[M-10\] `PrizePool` -> Winners wouldn't be able to claim prize correctly in `claimPrize` function](#)
 - [\[M-11\] `Vault.mintWithPermit\(\)` can be DoS'd](#)
 - [\[M-12\] The tier odds in `TieredLiquidityDistributor` are incorrect](#)
 - [\[M-13\] Users can manipulate the observation creation](#)
 - [\[M-14\] Number of prize tiers always increases if just 1 canary prize is claimed](#)
 - [\[M-15\] Tiers can be maintained active to give unfair advantage to user through DoS](#)
 - [\[M-16\] The threshold check for adding of new tiers is skipped when `_nextNumberOfTiers` is at the maximum amount](#)

- [\[M-17\] `VaultFactory` allows deployment of vaults with non-authentic `TwabController` and `PrizePool`](#)
- [\[M-18\] The exchange rate change in the case of Lossy Strategy will cause the Vault to be under-collateralized for generic ERC4626 Yield Vaults](#)
- [\[M-19\] Silent overflow could alter computation when calculating the `vaultPortion` in the `PrizePool` contract](#)
- [\[M-20\] Improper handling of cases when withdrawable assets = 0](#)
- [\[M-21\] Vault contribution calculations wrongly include the current round when claiming prizes](#)
- [\[M-22\] Loss of precision leads to under-collateralized](#)
- [\[M-23\] Vault does not conform to ERC4626](#)
- [\[M-24\] `Claimer.claimPrizes` can be front-run in order to make losses for the claim bot](#)
- [\[M-25\] `depositWithPermit` and `mintWithPermit` are allowed to be called by the permit creator only](#)
- [\[M-26\] Transfer of Vault tokens can cause accounting errors in other contracts](#)
- [\[M-27\] Inconsistent behavior for canary claims in the `claimer`](#)
- [Low Risk and Non-Critical Issues](#)
 - [L-01 `vault._mint\(\)` forcing conversion to `uint96` may result in missing values](#)
 - [L-02 `targetOf\(\)` incorrectly uses `_token != _liquidationPair.tokenIn\(\)`](#)
 - [L-03 `setLiquidationPair\(\)` may not be able to set the old `liquidationPair`](#)
 - [L-04 in `_withdraw\(\)`, by calling `_burn\(\)` and then executing `_yieldVault.withdraw\(\)`, it may lead `_updateExchangeRate\(\)` inaccuracy](#)
 - [L-05 `liquidate\(\)` is at risk of a sandwich attack](#)
- [Gas Optimizations](#)

- [Auditor's Disclaimer](#)
- [Note on Gas estimates](#)
- [Tightly pack storage variables/optimize the order of variable declaration \(saves 2.1K gas\)](#)
- [Expensive operation inside a for loop](#)
- [Use `calldata` instead of `memory` for function parameters](#)
- [Repeatedly doing the same operation \(addition\) involving a state variable should be avoided](#)
- [Use the existing memory value to do the subtraction](#)
- [Emitting storage values instead of the memory one.](#)
- [Optimizing the check order for cost efficient function execution](#)
- [Don't cache a state variable if it's used only once](#)
- [Do not cache immutable values](#)
- [Conclusion](#)
- [Audit Analysis](#)
 - [Introduction](#)
 - [Analysis of the Codebase](#)
 - [Architecture Feedback, Centralization & Systemic Risks](#)
 - [Recommendations](#)
 - [Time spent](#)
- [Mitigation Review](#)
 - [Introduction](#)
 - [Overview of Changes](#)
 - [Mitigation Review Scope](#)
 - [Out of Scope](#)
 - [Mitigation Review Summary](#)
 - [H-05 Mitigation error: The `sponsorWithPermit` function allows anyone to provide a valid permit for asset](#)

- [H-09 Not fully mitigated: `_yieldVault.maxWithdraw\(\)` returning a different value than the expected real balance of the caller](#)
- [M-02 Not fully mitigated: A malicious user can still gas grief a claimer by deliberately reverting in a hook](#)
- [M-09 Mitigation error: Auctions run at significantly different speeds for different prize tiers](#)
- [M-16 Mitigation error: Number of prize tiers may never scale due to aggressive new algorithm.](#)
- [Assessed type](#)
- [M-18 Not fully mitigated: `_redeem` method doesn't handle the case where the withdrawal from the underlying yield vault realises losses](#)
- [M-22 Not fully mitigated: deposits could revert if there is a loss of precision of 1 wei in the underlying yield vault](#)
- [Claiming prizes will be bricked if prize periods are not aligned with twab periods](#)
- [Vault will stop participating in draws in the case they deposited maximum assets to the underlying vault](#)
- [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 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 outlined in this document, C4 conducted an analysis of the PoolTogether smart contract system written in Solidity. The audit took place between July 7 - July 14 2023.

Following the C4 audit, 3 wardens ([dirk_y](#), [rvierdiiev](#) and [OxStalin](#)) reviewed the mitigations for all identified issues; the mitigation review report is appended below the

audit report.



Wardens

117 Wardens contributed reports to the PoolTogether:

1. [dirk_y](#)
2. [rvierdiiev](#)
3. [OxStalin](#)
4. [zzzitron](#)
5. [KupiaSec](#)
6. [wangxx2026](#)
7. KIntern_NA ([duc](#) and [TrungOre](#))
8. [pontifex](#)
9. [bin2chen](#)
10. [shaka](#)
11. [volodya](#)
12. [Brenzee](#)
13. [minhtrng](#)
14. [yixxas](#)
15. [Aymen0909](#)
16. [Nyx](#)
17. [GalloDaSballo](#)
18. [ktg](#)
19. [mahyar](#)
20. Ox3e84fa45 ([OxArcturus](#) and [markus_ether](#))
21. [seeques](#)
22. [Jeiwan](#)
23. [RedTiger](#)
24. [DadeKuma](#)
25. [xuwinnie](#)

26. [OxWaitress](#)
27. [ni8mare](#)
28. [markus_ether](#)
29. [Infect3d](#)
30. [degensec](#)
31. [Udsen](#)
32. [catellatech](#)
33. [josephdara](#)
34. [MiniGlome](#)
35. [qpzm](#)
36. [Oxbepresent](#)
37. [Oxkasper](#)
38. [alymurtazamemon](#)
39. [Tripathi](#)
40. [nadin](#)
41. [hals](#)
42. [saneryee](#)
43. [3docSec](#)
44. [c3phas](#)
45. [SAQ](#)
46. [hunter_w3b](#)
47. [petrichor](#)
48. [gzeon](#)
49. [Inspektor](#)
50. [keccak123](#)
51. GREY-HAWK-REACH ([Kose](#), [aswinraj94](#), [dimulski](#), [Oxprinc](#), [aslanbek](#) and [escrow](#))
52. [OxMirce](#)
53. [ptsanev](#)
54. [Breeje](#)

- 55. [CoOnan](#)
- 56. [ABAIKUNANBAEV](#)
- 57. [squeaky_cactus](#)
- 58. [dacian](#)
- 59. [kutugu](#)
- 60. [OxSmartContract](#)
- 61. [peanuts](#)
- 62. [K42](#)
- 63. [wvleak](#)
- 64. [Ox11singh99](#)
- 65. [devOcloo](#)
- 66. [serial-coder](#)
- 67. [Praise](#)
- 68. [naman1778](#)
- 69. [Rolezn](#)
- 70. [grearlake](#)
- 71. [Daniel526](#)
- 72. [btk](#)
- 73. [OxAnah](#)
- 74. [ybansal2403](#)
- 75. [SM3_SS](#)
- 76. [Raihan](#)
- 77. [ReyAdmirado](#)
- 78. [Rageur](#)
- 79. [SAAJ](#)
- 80. [Oxn006e7](#)
- 81. [LeoS](#)
- 82. [koxuan](#)
- 83. [jaraxxus](#)

84. [oxchsystem](#)
85. [neumo](#)
86. [OxPsuedoPandit](#)
87. [namx05](#)
88. [Jorgect](#)
89. [YY](#)
90. [Darwin](#)
91. [lanrebayode77](#)
92. [alexzoid](#)
93. [Kaysoft](#)
94. [banpaleo5](#)
95. [MohammedRizwan](#)
96. [Bauchibred](#)
97. [erebus](#)
98. [Vagner](#)
99. [joaovwfreire](#)
100. [ayden](#)
101. [ArmedGoose](#)
102. [fatherOfBlocks](#)
103. [eyexploit](#)
104. [teawaterwire](#)
105. [alexweb3](#)
106. [LuchoLeonel](#)
107. [Bobface](#)
108. [mahdirostami](#)
109. [John](#)
110. [ravikiranweb3](#)

This audit was judged by [Picodes](#).

Final report assembled by thebrittfactor.



Summary

The C4 analysis yielded an aggregated total of 36 unique vulnerabilities. Of these vulnerabilities, 9 received a risk rating in the category of HIGH severity and 27 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 19 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 PoolTogether repository](#), and is composed of 14 smart contracts written in the Solidity programming language and includes 3324 lines of Solidity code.



Severity Criteria

C4 assesses the severity of disclosed vulnerabilities based on 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

For more information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on [the C4 website](#), specifically our section on [Severity Categorization](#).



High Risk Findings (9)



[H-01] The `_currentExchangeRate` of the Vault contract can't

increase and will always be lower than or equal to

`_assetUnit`

Submitted by [KIntern_NA](#), also found by [KupiaSec](#)

The `_currentExchangeRate` of the Vault contract can not increase and will always be lower than or equal to `_assetUnit`. Therefore, when the vault is under-collateralized (`_currentExchangeRate < _assetUnit`), it can't be further collateralized.



Proof of concept

```
function _currentExchangeRate() internal view returns (uint256)
    uint256 _totalSupplyAmount = _totalSupply();
    uint256 _totalSupplyToAssets = _convertToAssets(
        _totalSupplyAmount,
        _lastRecordedExchangeRate,
        Math.Rounding.Down
    );

    uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address(0));

    if (_withdrawableAssets > _totalSupplyToAssets) {
        _withdrawableAssets = _totalSupplyToAssets;
    }

    if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
        return _withdrawableAssets.mulDiv(_assetUnit, _totalSupplyAmount);
    }

    return _assetUnit;
}
```

The `_totalSupplyAmount != 0 && _withdrawableAssets != 0`, `_currentExchangeRate` function will return a value `_withdrawableAssets * _assetUnit / _totalSupplyAmount`. However, `_withdrawableAssets` can not exceed `_totalSupplyToAssets`, which is equal to `_totalSupplyAmount * _lastRecordedExchangeRate / _assetUnit`. Therefore, `_currentExchangeRate` will always be lower than or equal to `_lastRecordedExchangeRate`.

Add this assert line and run `forge test`; all tests will pass.

```
if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {  
    assert(_withdrawableAssets.mulDiv(_assetUnit, _totalSupplyAmount)  
    return _withdrawableAssets.mulDiv(_assetUnit, _totalSupplyAmount)  
}
```



Recommended Mitigation Steps

Remove the lines of code that limit the `_withdrawableAssets`:

```
if (_withdrawableAssets > _totalSupplyToAssets) {  
    _withdrawableAssets = _withdrawableAssets - (_withdrawableAssets -  
}
```



Assessed type

Context

[asselstine \(PoolTogether\) confirmed and commented:](#)

Hmm, I'm not sure about this one. Will mark as confirmed and figure it out later.

[Picodes \(judge\) commented:](#)

@asselstine - did you figure it out?

It seems to me that because of `if (_withdrawableAssets > _convertToAssets(_totalSupplyAmount, _lastRecordedExchangeRate, Math.Rounding.Down))`, the rate indeed can't increase; which is a huge issue in case of a momentary undercollateralization.

[PierrickGT \(PoolTogether\) commented:](#)

@Picodes - Indeed, the exchange rate should not be greater than 1 because users should not be able to claim the yield that has accrued in the `YieldVault`.

That's why we have the following condition:

```
if (_withdrawableAssets > _totalSupplyToAssets) {  
    _withdrawableAssets = _withdrawableAssets - (_withdrawableAssets - _totalSupplyToAssets)  
}
```

We subtract the yield from the total amount, with the yield being the difference between `_withdrawableAssets` and `_totalSupplyToAssets`.

If the `YieldVault` becomes under-collateralized, users won't be able to deposit anymore, but will be able to withdraw their share of the deposit. Any yield that has accrued and has not been claimed yet will be shared proportionally amongst users.

[PierrickGT \(PoolTogether\) commented:](#)

The code has been improved and clarified in the following PR:
<https://github.com/GenerationSoftware/pt-v5-vault/pull/13>.

[Picodes \(judge\) commented:](#)

Keeping high severity here because of the issue in case of temporary undercollateralization.

[PoolTogether mitigated:](#)

The issue turned out not to be the case; the exchange rate is always ≤ 1 (yield is liquidated). However, comments were added to clarify the behaviour.
PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/13>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[H-02] A malicious user can steal other user's deposits from Vault.sol

Submitted by [zzzitron](#), also found by [pontifex](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L509-L521> <https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L407-L415>



Impact

When the `Vault.withdraw()` function is called, a maximum of `type(uint96).max` shares are burnt subsequently: `Vault.withdraw() -> Vault._withdraw() -> Vault._burn` burns `uint96(_shares)`, see [Vault.sol line 1139](#).

A malicious user can exploit this in the following way:

1. A malicious user deposits, for example, two times the value of `type(uint96).max` underlying assets into the Vault; calling the function `Vault.deposit()` two times. They can't deposit more in a single transaction because `type(uint96).max` is the maximum value to deposit.
2. Then, the malicious user calls `Vault.withdraw()` with a higher value of assets to withdraw more than `type(uint96).max`. For example, they withdraw `(2 * type(uint96).max)`, which is the total amount of assets they deposited before.
3. Now what happens, is the Vault.sol contract only burns `type(uint96).max` shares for the user, but transfers `2 * type(uint96).max` underlying assets to the malicious user, which is the total amount they deposited before.
4. This happens because `Vault._burn()` only burns `uint96(shares)` shares of the malicious users - see Vault.sol line 1155.
5. Now, the malicious user has still vault shares left but they withdrew the total amount of their deposited assets.
6. Now, the vault transferred the total amount of the malicious user's assets back to them and the malicious user still has shares left to withdraw; with even more assets that are now being stolen from assets deposited by other users.
7. Or, if the malicious user was the first depositor, they wait until another user deposits and the malicious user can now withdraw the other users deposited assets since the malicious user still has Vault shares left.

8. Or, if the malicious user is not the first depositor, they use a `flashLoan` or `flashMint` to deposit multiple times `type(uint96).max` assets into the vault. Then, they can withdraw their deposit, pay back the `flashLoan` or `flashMint` and they will still have enough vault shares left to steal all other users assets by withdrawing them.

In this way, other user's deposited assets can be stolen, as explained above.



Proof of Concept

Here is a POC, where the problem is illustrated:

<https://gist.github.com/zzzitron/397790302ca95aa3fbf05694ae1497ab>



Recommended Mitigation Steps

Consider adjusting the `Vault._burn` function to not convert from `uint256` to `uint96` when burning shares.



Assessed type

Math

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Added SafeCast

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/9>

Status: Mitigation confirmed. Full details in reports from [dirk_y](#), [rvierdiev](#) and [OxStalin](#).



[H-03] `_amountOut` is representing assets and shares at the same time in the `liquidate` function

Submitted by [Aymen0909](#), also found by [KupiaSec](#), [wangxx2026](#) ([1](#), [2](#)), and [OxWaitress](#)

In the `liquidate` function from the `Vault` contract, the input argument `_amountOut` is used as if it was representing a value of asset amounts and share amounts at the same time; which is impossible as there is a conversion rate between them. This error will make the `liquidate` function behave in an expected manner, not the one that was intended.

Proof of Concept

The issue is occurring in the `liquidate` function below:

```
function liquidate(
    address _account,
    address _tokenIn,
    uint256 _amountIn,
    address _tokenOut,
    uint256 _amountOut
) public virtual override returns (bool) {
    _requireVaultCollateralized();
    if (msg.sender != address(_liquidationPair))
        revert LiquidationCallerNotLP(msg.sender, address(_liquidationPair));
    if (_tokenIn != address(_prizePool.prizeToken()))
        revert LiquidationTokenInNotPrizeToken(_tokenIn, address(_prizePool.prizeToken()));
    if (_tokenOut != address(this))
        revert LiquidationTokenOutNotVaultShare(_tokenOut, address(this));
    if (_amountOut == 0) revert LiquidationAmountOutZero();

    uint256 _liquidableYield = _liquidatableBalanceOf(_tokenOut);

    // @audit _amountOut compared with _liquidableYield which represents share amounts
    // @audit _amountOut is considered as an asset amount
    if (_amountOut > _liquidableYield)
        revert LiquidationAmountOutGTYield(_amountOut, _liquidableYield);

    _prizePool.contributePrizeTokens(address(this), _amountIn);

    if (_yieldFeePercentage != 0) {
        // @audit _amountOut used to increase fee shares so consider it as share amounts
        _increaseYieldFeeBalance(
            (_amountOut * FEE_PRECISION) / (FEE_PRECISION - _yieldFeePercentage)
        );
    }

    uint256 _vaultAssets = IERC20(asset()).balanceOf(address(this));
```



```

// @audit _amountOut compared with _vaultAssets which represents
// @audit _amountOut is considered as an asset amount
if (_vaultAssets != 0 && _amountOut >= _vaultAssets) {
    _yieldVault.deposit(_vaultAssets, address(this));
}

// @audit _amountOut representing a share amount minted to the
_mint(_account, _amountOut);

return true;
}

```

As you can see from the code above, the value of the argument `_amountOut` is used multiple times in the function logic and each time is representing either an asset amount or a share amount; which is impossible as there is a conversion formula used to transform the asset amount into the share amount (and inversely) with the function `_convertToShares` (or `_convertToAssets`).

From the function comments, I couldn't figure out what the value of `_amountOut` actually represents, but because there is also another argument given to the `liquidate` function, which is `_tokenOut == address(this)`, I'm supposing that `_amountOut` is representing as a share amount; which will mean that all the instances highlighted in the code above, when `_amountOut` is considered as an asset amount, are wrong.



Instance 1:

```

// @audit _amountOut compared with _liquidableYield which represents
if (_amountOut > _liquidableYield)
    revert LiquidationAmountOutGTYield(_amountOut, _liquidableYield);

```



Instance 2:

```

// @audit _amountOut compared with _vaultAssets which represents
if (_vaultAssets != 0 && _amountOut >= _vaultAssets) {
    _yieldVault.deposit(_vaultAssets, address(this));
}

```

And before comparing `_amountOut` to the asset amount values of `_vaultAssets` and `_liquidableYield`, it's value should be converted to an asset amount with the function `_convertToAssets`.

This issue will cause problems for the protocol working, as the `liquidate` function logic will not behave as expected, because it's comparing values that represent different things.

Note: If `_amountOut` is actually representing an asset amount (not a share amount as I supposed), the issue is still valid because `_amountOut` is also used as being a share amount inside the `liquidate` function. In that case, it should first be converted to a share amount with `_convertToShares` in order to get the correct behavior of the `liquidate` function.



Recommended Mitigation Steps

To solve this issue, I recommend to first convert the value of `_amountOut` in the `liquidate` function to an asset amount and store it in a local variable, `_amountOutToAsset`. In the function logic, use the correct variable, either `_amountOut` or `_amountOutToAsset`, when interacting with a share amount or an asset amount.



Assessed type

Error

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Fixed conversion and naming of field.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/6>

Status: Mitigation confirmed. Full details in reports from [dirk_y](#), [rvierdiev](#) and [OxStalin](#).



[H-04] `Vault.mintYieldFee` function can be called by anyone to mint Vault Shares to any recipient address

Submitted by [Udsen](#), also found by [minhtrng](#), [markus_ether](#), [GREY-HAWK-REACH](#), [KupiaSec](#), [serial-coder](#), [Aymen0909](#), [peanuts](#), [teawaterwire](#), [ni8mare](#), [alexweb3](#), [josephdara](#), [zzzitron](#), [Jeiwan](#), [Nyx](#), [keccak123](#), [LuchoLeonel](#), [btk](#), [seeques](#), [OxPsuedoPandit](#), [OxMirce](#), [RedTiger](#), [Praise](#), [bin2chen](#), [ktg](#), [Bobface](#), [rvierdiiev](#), [wangxx2026](#), [Oxbepresent](#), [dirk_y](#), [ptsanev](#), [shaka](#), [dacian](#), [mahdirostami](#), [John](#), [OxStalin](#), [ravikiranweb3](#), and [CoOnan](#)

The `Vault.mintYieldFee` external function is used to mint Vault shares to the yield fee `_recipient`. The function is an external function and can be called by anyone since there is no access control. The function will revert only under following two conditions:

1. If the Vault is under-collateralized.
2. If the `_shares` are greater than the accrued `_yieldFeeTotalSupply`.

The issue with this function is, it allows the caller to set the `_recipient` (address of the yield fee recipient). It does not use the `_yieldFeeRecipient` state variable, which was set in the `Vault.constructor` as the yield fee recipient.

Which means, anyone can steal the available `yield fee` from the vault (as long as the above two revert conditions are not satisfied) by minting shares to their own address or to any address of their choice.



Proof of Concept

```
function mintYieldFee(uint256 _shares, address _recipient) external {
    _requireVaultCollateralized();
    if (_shares > _yieldFeeTotalSupply) revert YieldFeeGTAvailable();

    _yieldFeeTotalSupply -= _shares;
    _mint(_recipient, _shares);

    emit MintYieldFee(msg.sender, _recipient, _shares);
}
```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L394-L402>

Tools Used

VSCode

Recommended Mitigation Steps

Hence, it is recommended to use the `_yieldFeeRecipient` state variable value as the `yield fee recipient` inside the `Vault.mintYieldFee` external function and to remove the input parameter `address _recipient` from the `Vault.mintYieldFee` function; so that the caller will not be able to mint shares to any arbitrary address of their choice and steal the yield fee of the protocol.

The updated function should be as follows:

```
function mintYieldFee(uint256 _shares) external {
    _requireVaultCollateralized();
    if (_shares > _yieldFeeTotalSupply) revert YieldFeeGTAvailab

    _yieldFeeTotalSupply -= _shares;
    _mint(_yieldFeeRecipient, _shares);

    emit MintYieldFee(msg.sender, _recipient, _shares);
}
```

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Removed recipient param.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/7>

Status: Mitigation confirmed. Full details in reports from [rvierdiiev](#), [dirk_y](#) and [OxStalin](#).

[H-05] Delegated amounts can be forcefully removed from anyone in the TwabController

Submitted by [Oxkasper](#), also found by [minhtrng](#), [qpzm](#), [Aymen0909](#), [Jeiwan](#), [GREY-HAWK-REACH](#), [OxStalin](#), [Oxbepresent](#), [3docSec](#), and [CoOnan](#)

The `sponsor` function in the `Vault.sol` contract allows anyone to remove another user's delegation by forcing them to delegate to the sponsor address. `_sponsor` will deposit some amount from the caller for the target user and then force a delegation to the sponsor address (`address(1)`).

However, this amount can just be 0 and so it becomes a function to simply force a removal of a delegation. The full delegated power gets removed, because delegations to the sponsor address are not tracked.

As such, it becomes possible to call the `sponsor` function for every user and make the total delegated power supply in the `TwabController` equal to 0. The attacker can then be the only one with some delegated amount that is equal to 100% of the total supply, manipulating the process of the lottery.

Rectifying the delegation requires manual interaction from the user and the exploit can be repeated anytime and continuously, further manipulating the values in the `TwabController`.



Proof of Concept

```
function testPoCDelegateRemoval() public {
    address SPONSORSHIP_ADDRESS = address(1);
    uint96 BALANCE = 100_000_000 ether;

    token.approve(address(vault), BALANCE);
    vault.deposit(BALANCE, address(this));

    assertEq(address(this), twab_controller.delegateOf(address(victim)));
    assertEq(BALANCE, twab_controller.delegateBalanceOf(address(victim)));

    // As attacker, call sponsor with 0 amount and victim address
    vm.prank(address(0xdeadbeef));
    vault.sponsor(0, address(this));
}
```

```
// Delegated balance is now gone
assertEq(SPONSORSHIP_ADDRESS, twab_controller.delegateOf(address(vault))
assertEq(0, twab_controller.delegateBalanceOf(address(vault))
assertEq(0, twab_controller.delegateBalanceOf(address(vault))
}
```



Tools Used

VSCode, Foundry.



Recommended Mitigation Steps

The `sponsor` function should only accept deposits if the receiver has already delegated to the sponsorship address. Otherwise, the deposit is accepted, but the delegation should not be forced.

[asselstine \(PoolTogether\) confirmed via duplicate issue #393](#)

[PierrickGT \(PoolTogether\) commented via duplicate issue #393:](#)

I've removed the `receiver` param in the following PR:

<https://github.com/GenerationSoftware/pt-v5-vault/pull/19>.

This way, only the `msg.sender` can sponsor the Vault by depositing into it and delegating to the sponsorship address, if it is not already the case. If the user wants to deposit on behalf of another user, they can still use the `deposit` function. Funds will then be delegated to any address set by the `receiver`.

[PoolTogether mitigated:](#)

Removed recipient param.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/19>

Status: Mitigation error. Full details in report from [rvierdiiev](#), and in the [Mitigation Review](#) section below.



[H-06] Resetting delegation will result in user funds being lost forever

Submitted by [dirk_y](#), also found by [KupiaSec](#), [Jeiwan](#), [Oxkasper](#), [Oxbepresent](#), [bin2chen](#), [rvierdiiev](#), and [xuwinnie](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/0145eeac23301ee5338c659422dd6d69234f5d50/src/TwabController.sol#L596-L599> <https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/0145eeac23301ee5338c659422dd6d69234f5d50/src/TwabController.sol#L648-L664>



Impact

The default delegate value for a user is `address(0)`, which maps to the user delegating to themselves. If a user had delegated to another address and wanted to reset their delegated balance back to themselves, they would lose all of their funds contributed to the vault.



Proof of Concept

As mentioned above, the default behaviour for a user is that they delegate their balance to themselves, where the actual default value in storage is the `0 address`:

```
function _delegateOf(address _vault, address _user) internal view returns (
    address _userDelegate;

    if (_user != address(0)) {
        _userDelegate = delegates[_vault][_user];

        // If the user has not delegated, then the user is the default
        if (_userDelegate == address(0)) {
            _userDelegate = _user;
        }
    }

    return _userDelegate;
}
```

When a user wants to delegate their balance, they call `delegate` in `TwabController.sol` and specify which vault they want to delegate the balance of and to which address they want to delegate to. This calls `_delegate` under the hood:

```
function _delegate(address _vault, address _from, address _to) internal {
    address _currentDelegate = _delegateOf(_vault, _from);
    if (_to == _currentDelegate) {
        revert SameDelegateAlreadySet(_to);
    }

    delegates[_vault][_from] = _to;

    _transferDelegateBalance(
        _vault,
        _currentDelegate,
        _to,
        uint96(userObservations[_vault][_from].details.balance)
    );

    emit Delegated(_vault, _from, _to);
}
```

If a user wanted to reset the delegation to themselves, they would specify `_to` as `address(0)`. However, the issue with this is that the underlying `_transferDelegateBalance` call will mistakenly move the delegated user funds to the `0` address.

At this point, the user might try to call `delegate` again with their actual address; however, now the `(_to == _currentDelegate)` check will be true and revert because of the behaviour specified earlier. The user also can't delegate to any other address because they don't own their own delegate balance anymore. Their funds are officially lost forever.

Below is a change to the existing test suite that can be executed with `forge test -vvv --match-path test/unit/Vault/Withdraw.t.sol` to demonstrate this issue:

```
diff --git a/test/unit/Vault/Withdraw.t.sol b/test/unit/Vault/Withdraw.t.sol
index 6a15a59..3cec9e3 100644
--- a/test/unit/Vault/Withdraw.t.sol
```



```

+++ b/test/unit/Vault/Withdraw.t.sol
@@ -47,6 +47,36 @@ contract VaultWithdrawTest is UnitBaseSetup {
    vm.stopPrank();
}

+ function testFundsLostForever() external {
+     vm.startPrank(alice);
+     uint256 _amount = 1000e18;
+     underlyingAsset.mint(alice, _amount);
+
+     // Alice deposits as usual
+     _deposit(underlyingAsset, vault, _amount, alice);
+
+     // Alice decides she wants to delegate to bob
+     twabController.delegate(address(vault), bob);
+
+     // Alice now tries to reset her delegation
+     twabController.delegate(address(vault), address(0));
+
+     // At this point the funds are lost!! Alice tries to recover
+
+     // Alice tries to delegate back to herself but can't
+     vm.expectRevert();
+     twabController.delegate(address(vault), alice);
+
+     // Alice also can't delegate to any other address
+     vm.expectRevert();
+     twabController.delegate(address(vault), bob);
+
+     // Alice can't withdraw because her funds have been lost forever
+     // Expecting a revert with "DelegateBalanceLTAmt(0, 1000e18)"
+     vault.withdraw(vault.maxWithdraw(alice), alice, alice);
+     vm.stopPrank();
+ }
+
+ function testWithdrawMoreThanMax() external {
+     vm.startPrank(alice);

```



Tools Used

Foundry



Recommended Mitigation Steps

The simplest way to fix this issue is to prevent delegating back to the `0 address`. If a user delegates away from the default, then they can delegate back to themselves by specifying their own address:

```
diff --git a/src/TwabController.sol b/src/TwabController.sol
index a7e2d51..ae7b9ea 100644
--- a/src/TwabController.sol
+++ b/src/TwabController.sol
@@ -646,6 +646,7 @@ contract TwabController {
    * @param _to the address to delegate to
    */
    function _delegate(address _vault, address _from, address _to
+    require(_to != address(0), "Cannot delegate back to 0 address");
    address _currentDelegate = _delegateOf(_vault, _from);
    if (_to == _currentDelegate) {
        revert SameDelegateAlreadySet(_to);
```



Assessed type

Invalid Validation

[asselstine \(PoolTogether\) confirmed via duplicate issue #293](#)

[Picodes \(judge\) commented via duplicate issue #293:](#)

High severity seems justified because of the lines `if (_userDelegate == address(0)) { _userDelegate = _user;}`. It seems likely that some users are tricked into thinking that delegating back to the `0 address` will cancel their delegation.

[PoolTogether mitigated:](#)

Added check for zero address.

PR: <https://github.com/GenerationSoftware/pt-v5-twab-controller/pull/7>

Status: Mitigation confirmed with comments. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[H-07] `_requireVaultCollateralized()` is called at the beginning of the functions `mintYieldFee()` and `liquidate()`

Submitted by [RedTiger](#), also found by [zzzitron](#) and [wangxx2026](#)

The `liquidate()` and `mintYieldFee()` functions could leave the vaults under-collateralized.

Proof of Concept

`_requireVaultCollateralized()` is called at the beginning of `mintYieldFee()` and `liquidate()`. These two functions change the state and the vault could become under-collateralized at the end of the functions.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L557>

```
function mintYieldFee(uint256 _shares, address _recipient) external {
    _requireVaultCollateralized();
    if (_shares > _yieldFeeTotalSupply) revert YieldFeeGTAvailab.
```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L395>

```
function mintYieldFee(uint256 _shares, address _recipient) external {
    _requireVaultCollateralized();
    if (_shares > _yieldFeeTotalSupply) revert YieldFeeGTAvailab.
```

Recommended Mitigation Steps

Call `_requireVaultCollateralized()` at the end of these functions instead of calling it at the beginning.

Assessed type

Invalid Validation

[Picodes \(judge\) increased severity to High](#)

[asselstine \(PoolTogether\) confirmed via duplicate issue #307](#)

[Picodes \(judge\) commented via duplicate issue #307:](#)

Keeping High severity as this report shows how shares could be minted; although, the vault is in fact under-collateralized leading to a loss of funds for users.

[PierrickGT \(PoolTogether\) commented via duplicate issue #307:](#)

The warden missed the following comment:

```
* @dev - We exclude the amount of yield generated by the Yield
*       Except when the vault is under-collateralized, in this c
```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/a08fe40155aa65aab202c8bda5806dd91eaa1a9a/src/Vault.sol#L1169-L1170>

If the Vault ends up being under-collateralized, any yield that has not been claimed will be shared proportionally between depositors. If we use `_totalShares` instead of `_totalSupply`, we would account for Vault shares that have not been minted yet, since `_yieldFeeTotalSupply` keeps track of the accrued yield fee, but needs to be minted as Vault shares by calling `mintYieldFee`:

<https://github.com/GenerationSoftware/pt-v5-vault/blob/44a6c6b081db5cc5e2acc4757a3c9dbaa6f60943/src/Vault.sol#L395>

I've added the following to test this scenario:

<https://github.com/GenerationSoftware/pt-v5-vault/blob/44a6c6b081db5cc5e2acc4757a3c9dbaa6f60943/test/unit/Vault/Witdraw.t.sol#L128>

[Picodes \(judge\) commented via duplicate issue #307:](#)

@PierrickGT - I do agree with you that the mitigation of this report is incorrect. However, my understanding is that there is still an important issue here; because

`mintYieldFee` could be called even when it would lead to an under-collateralized state.

So we could imagine a scenario where `_yieldFeeTotalSupply` is 4, assets in the vault are 10, and user shares are 10. Then, a call to `mintYieldFee` would mint 4 and would lead to a loss of funds for users, as it'd bring the vault into an under-collateralized state.

[PierrickGT \(PoolTogether\) commented via duplicate issue #307:](#)

@Picodes - Yes, exactly. This issue you are referring to was better explained in [this issue](#).

[PoolTogether mitigated:](#)

Fixed check for partial collateralization.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/13>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[H-08] Increasing reserves breaks PrizePool accounting

Submitted by [dirk_y](#), also found by [Jeiwan](#), [OxStalin](#), and seeques ([1](#), [2](#))



Lines of code

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L498-L502> <https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L743-L746> <https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L312>



Impact

When anyone calls the `increaseReserve` method in `PrizePool.sol` the accounted balance in the prize pool isn't properly updated. This allows a vault to effectively steal

the prize token contribution and this contribution gets distributed during draws; effectively double counting the initial injection into the reserves. The actual prize token balance of the prize pool will be below the accounted balance of the prize pool as time goes on.



Proof of Concept

As mentioned in the audit README:

“the balance of prize tokens held by the contract must always be equal to the sum of the available tier liquidity and the reserve. When contributing liquidity, the prize pool will temporarily hold a balance greater than the accounted balance, but otherwise the two should match”.

Unfortunately, this is broken when anyone contributes directly to the reserve by calling `increaseReserve`.

In the normal audit flow, the reserve is increased when a draw is closed by the draw manager. During calls to `closeDraw`, the next draw is started with a given number of tiers and the contributions for the round are calculated and split across the tiers and the reserve:

```
_nextDraw(_nextNumberOfTiers, uint96(_contributionsForDraw(lastC.
```

Under the hood, this calls `_computeNewDistributions` which calculates the amount to increase the reserves, based on the number of reserve shares and the new prize token liquidity being contributed in this round. During this flow, the actual balance of reward tokens held in the prize pool are equal to the accounted balance.

The break in accounting occurs when calling `increaseReserve`:

```
function increaseReserve(uint104 _amount) external {
    _reserve += _amount;
    prizeToken.safeTransferFrom(msg.sender, address(this), _amount);
    emit IncreaseReserve(msg.sender, _amount);
}
```

As you can see, the prize tokens are transferred into the pool and the reserve increased. But the accounted balance is unchanged:

```
function _accountedBalance() internal view returns (uint256) {
    Observation memory obs = DrawAccumulatorLib.newestObservation();
    return (obs.available + obs.disbursed) - _totalWithdrawn;
}
```

Because the accounted balance is unchanged, any vault can now call `contributePrizeTokens` to effectively steal the funds meant for the reserve:

```
function contributePrizeTokens(address _prizeVault, uint256 _amount) internal {
    uint256 _deltaBalance = prizeToken.balanceOf(address(this)) - _accountedBalance();
    _prizeVault.transferFrom(address(this), _prizeVault, _deltaBalance);
    _reserve += _deltaBalance;
}
```

This increases the relevant vault accumulator and the total accumulator; thereby, effectively double counting the same prize tokens, since we've already increased `_reserve`.



Recommended Mitigation Steps

The accounted balance of the prize pool should be updated when `increaseReserve` is called. I think the easiest way of achieving this is having a tracker for “reserve injections”:

```
diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..3c14476 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -233,6 +233,9 @@ contract PrizePool is TieredLiquidityDistribution {
    /// @notice The total amount of prize tokens that have been contributed
    uint256 internal _totalWithdrawn;

+    /// @notice The total amount of reserve injections that have been made
+    uint256 internal _reserveInjections;
+
    /// @notice The winner random number for the last closed draw
    uint256 internal _winningRandomNumber;

@@ -497,6 +500,7 @@ contract PrizePool is TieredLiquidityDistribution {
```

```

    /// @param _amount The amount of tokens to increase the reserve
    function increaseReserve(uint104 _amount) external {
        _reserve += _amount;
+       _reserveInjections += amount;
        prizeToken.safeTransferFrom(msg.sender, address(this), _amount);
        emit IncreaseReserve(msg.sender, _amount);
    }
@@ -742,7 +746,7 @@ contract PrizePool is TieredLiquidityDistribution
    /// @return The balance of tokens that have been accounted for
    function _accountedBalance() internal view returns (uint256)
        Observation memory obs = DrawAccumulatorLib.newestObservation();
-       return (obs.available + obs.disbursed) - _totalWithdrawn;
+       return (obs.available + obs.disbursed) - _totalWithdrawn + _
    }

    /// @notice Returns the start time of the draw for the next slot

```



Assessed type

Math

[asselstine \(PoolTogether\) confirmed and commented via duplicate issue #200:](#)

Going to add some notes here:

The issue is that `_accountedBalance()` **computes** `(obs.available + obs.disbursed) - _totalWithdrawn`, which basically means the total contributed liquidity minus the withdrawn liquidity.

However, when adding liquidity manually, via `increaseReserve`, it does not increase the contributed liquidity.

This issue points to back-running as the problem, but that's not the case. The case is improper accounting, as in [this issue](#) that was closed as a duplicate.

[PoolTogether mitigated:](#)

Fixed reserve accounting.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/18>

Status: Mitigation confirmed. Full details in reports from [dirk_y](#), [rvierdiiev](#) and [OxStalin](#).



[H-09] Vault is not compatible with some ERC4626 vaults

Submitted by [rvierdiiev](#), also found by [Brenzee](#)

Depositors can lose funds.



Proof of Concept

Anyone can build `Vault` with an underlying vault inside, which should earn yields.

When a user deposits/withdraws then the `_convertToShares` **function is used** to determine amount of shares the user will receive for the provided assets amount. This function **calls** `_currentExchangeRate` to find out current rate.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1168-L1187>

```
function _currentExchangeRate() internal view returns (uint256
    uint256 _totalSupplyAmount = _totalSupply();
    uint256 _totalSupplyToAssets = _convertToAssets(
        _totalSupplyAmount,
        _lastRecordedExchangeRate,
        Math.Rounding.Down
    );

    uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address:

    if (_withdrawableAssets > _totalSupplyToAssets) {
        _withdrawableAssets = _withdrawableAssets - (_withdrawable:
    }

    if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
        return _withdrawableAssets.mulDiv(_assetUnit, _totalSupply:
    }
```

```
    return _assetUnit;  
}
```

As you can see, in order to find the current exchange rate, function `_yieldVault.maxWithdraw(address(this))` is used. In this case, if this value (which is supposed to be a full amount of deposits + yields inside `_yieldVault`) is less than `_totalSupplyAmount` (which is the total supply `_lastRecordedExchangeRate`), then the rate will be decreased. Which means that the vault lost funds and users should receive less when they withdraw. Later, this new `_currentExchangeRate` **will be stored as** `_lastRecordedExchangeRate`.

Now, when I explained how rate is changed, I can also explain the problem with some ERC4626 vaults.

There are some ERC4626 vaults as `DnGmxSeniorVault`, that collect deposited funds and borrow them. When you **call** `maxWithdraw` **for such vaults**, if not enough funds are present (due to some borrowing percentages on the vault), then the amount returned can be less than real balance of caller.

In case a such wrong value is returned, then depositors of the `Pool` vault will face losses, as their exchange rate will be less than 1. `DnGmxSeniorVault` will again have enough balance (when the debt is repaid by `jn` vault), and the exchange rate will be 1.

Another problem ERC4626 vaults can create are vaults that have a withdraw limit. In that case, if the `Pool` vault has balance inside the yield vault that is bigger than the borrow limit, depositors will face the same problem, which leads to a loss of funds.



Tools Used

VsCode



Recommended Mitigation Steps

You need to consider cases where some vaults can't be used as yield vaults and be aware of vault creators for that.



Assessed type

Error

[asselstine \(PoolTogether\) confirmed](#)

[asselstine \(PoolTogether\) commented:](#)

This is more of a general warning about the possible incompatibilities of 4626 vaults.

[PoolTogether mitigated:](#)

Fixed undercollateralized redemption while providing an exit.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/37>

Status: Not fully mitigated. Full details in reports from [OxStalin](#) and [rvierdiiev](#), and in the [Mitigation Review](#) section below.



Medium Risk Findings (27)



[M-01] If the underlying asset is a fee on transfer token, it could break the internal accounting of the vault

Submitted by [Udsen](#), also found by [qpzm](#) and [saneryee](#)

The `Vault._deposit` function is used by the users to deposit `_assets` to the vault and mint vault shares to the `recipient` address. The amount of `_assets` are transferred to the `Vault` as follows:

```
SafeERC20.safeTransferFrom(  
    _asset,  
    _caller,  
    address(this),  
    _assetsDeposit != 0 ? _assetsDeposit : _assets  
);
```

The `Vault.deposit` function uses this `_assets` amount to calculate the number of shares to be minted to the `_recipient` address.

The issue here is if the underlying `_asset` is a fee on transfer token then the actual received amount to the vault will be less than what is referred in the `Vault.deposit` function `_assets` input parameter. But the shares to mint is calculated using the entire `_assets` amount.

This issue could be further aggravated since the `_asset` is again deposited to the `_yieldVault` and when needing to be redeemed, will be withdrawn from the `_yieldVault` as well. These operations will again charge a fee if the `_asset` is a fee on transfer token. Hence, the actual `_asset` amount left for a particular user will be less than the amount they initially transferred in.

Hence, when the user redeems the minted shares back to the `_assets`, the contract will not have enough assets to transfer to the `redeemer`, thus reverting the transaction.



Proof of Concept

```
SafeERC20.safeTransferFrom(
    _asset,
    _caller,
    address(this),
    _assetsDeposit != 0 ? _assetsDeposit : _assets
);
```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L951-L956>

```
_yieldVault.deposit(_assets, address(this));
```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L959>

```
_yieldVault.withdraw(_assets, address(this), address(this));  
SafeERC20.safeTransfer(IERC20(asset()), _receiver, _assets);
```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1026-L1027>



Tools Used

VSCode



Recommended Mitigation Steps

It is recommended to compute the `_assets` amount balance of the contract before and after the `safeTransferFrom` call to get the difference between the two for the “actually transferred” amount to the `Vault`. Then, the “actually transferred” amount can be converted to shares and mint the correct amount of shares to the `recipient`.

```
uint256 balanceBefore = _asset.balanceOf(address(this));  
  
SafeERC20.safeTransferFrom(  
    _asset,  
    _caller,  
    address(this),  
    _assetsDeposit != 0 ? _assetsDeposit : _assets  
);  
  
uint256 balanceAfter = _asset.balanceOf(address(this));  
  
uint256 transferredAmount = balanceAfter - balanceBefore;
```

[asselstine \(PoolTogether\) confirmed](#)



[M-O2] Unintended or malicious use of prize winners' hooks

Submitted by [wvleak](#), also found by [wvleak](#), [serial-coder](#), [DadeKuma](#), [Oxbepresent](#), [dirk_y](#), [keccak123](#), [rvierdiiev](#), [shaka](#), [3docSec](#), [Udsen](#), and [Jeiwan](#)

The `setHooks` function in `Vault.sol` allows users to set arbitrary hooks, potentially enabling them to make external calls with unintended consequences. This vulnerability could lead to various unexpected behaviors, such as unauthorized side transactions with gas paid unbeknownst to the claimer, reentrant calls, or denial-of-service attacks on claiming transactions.

[Vault.sol#L653](#)

```
function setHooks(VaultHooks memory hooks) external {
    _hooks[msg.sender] = hooks;

    emit SetHooks(msg.sender, hooks);
}
```



Proof of Concept

Consider the following side contract and malicious hook implementation:

Side Contract:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;

import "forge-std/console.sol";

contract SideContract {
    uint256 public codex;

    function execute() external {
        codex = 1;
        console.log("Entered side contract");
    }
}
```

Malicious Hook:

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.17;
```

```

import { SideContract } from "../SideContract.sol";

contract ViciousHook {
    SideContract public sideContract = new SideContract();

    function beforeClaimPrize(
        address winner,
        uint8 tier,
        uint32 prizeIndex
    ) external returns (address) {
        sideContract.execute();

        return address(this);
    }
}

```

Modified Test File:

```

diff --git a/test/unit/Vault/Vault.t.sol b/test/unit/Vault/Vault
index d0d6d30..65caca1 100644
--- a/test/unit/Vault/Vault.t.sol
+++ b/test/unit/Vault/Vault.t.sol
@@ -4,6 +4,7 @@ pragma solidity 0.8.17;
    import { UnitBaseSetup, LiquidationPair, PrizePool, TwabControl
    import { IVaultHooks, VaultHooks } from "../../src/interface
    import "src/Vault.sol";
+import { ViciousHook } from "../ViciousHook.sol";

    contract VaultTest is UnitBaseSetup {
        /* ===== Events ===== */
@@ -174,6 +175,25 @@ contract VaultTest is UnitBaseSetup {
        vm.stopPrank();
    }

+    function testClaimPrize_viciousHook() public {
+        ViciousHook viciousHook = new ViciousHook();
+        vm.startPrank(alice);
+        VaultHooks memory hooks = VaultHooks({
+            useBeforeClaimPrize: true,
+            useAfterClaimPrize: false,
+            implementation: IVaultHooks(address(viciousHook))
+        });
+        vault.setHooks(hooks);
+        vm.stopPrank();

```

```

+
+     vm.startPrank(address(claimer));
+
+     mockPrizePoolClaimPrize(uint8(1), alice, 0, address(viciousl
+     claimPrize(uint8(1), alice, 0, 1e18, address(claimer));
+
+     vm.stopPrank();
+ }
+
+     function testClaimPrize_beforeHook() public {
+         vm.startPrank(alice);
+         VaultHooks memory hooks = VaultHooks({

```

When running the test with `forge test --match-test`

`testClaimPrize_viciousHook -vv` the output is:

```

Running 1 test for test/unit/Vault/Vault.t.sol:VaultTest
[PASS] testClaimPrize_viciousHook() (gas: 321545)
Logs:
    Entered side contract

```

```

Test result: okay. 1 passed; 0 failed; finished in 7.29ms

```

This indicates that it is possible for a hook to make an external call and modify the EVM state. With that fact, there are multiple attack vectors.



Tools Used

Foundry



Recommended Mitigation Steps

To prevent any malicious calls, there are two possible solutions:

1. Change the `IVaultHook.sol` to set the hooks as view functions and prevent EVM state changes:

```

diff --git a/src/interfaces/IVaultHooks.sol b/src/interfaces/IVa
index 15217b8..95482c1 100644
--- a/src/interfaces/IVaultHooks.sol

```



```

+++ b/src/interfaces/IVaultHooks.sol
@@ -23,7 +23,7 @@ interface IVaultHooks {
    address winner,
    uint8 tier,
    uint32 prizeIndex
- ) external returns (address);
+ ) external view returns (address);

    /// @notice Triggered after the prize pool claim prize function
    /// @param winner The user who won the prize and for whom this prize is claimed
@@ -37,5 +37,5 @@ interface IVaultHooks {
    uint32 prizeIndex,
    uint256 payout,
    address recipient
- ) external;
+ ) external view;
}

```

2. As this solution may disturb the business logic, another solution would be to cap the gas used by the hooks. In `Vault.sol`, set a gas limit variable that can be adjusted by the owner of the vault for flexibility:

[Vault.sol#L1053](#)

[Vault.sol#L1068](#)

```

function _claimPrize(
    address _winner,
    uint8 _tier,
    uint32 _prizeIndex,
    uint96 _fee,
    address _feeRecipient
) internal returns (uint256) {
    VaultHooks memory hooks = _hooks[_winner];
    address recipient;
    if (hooks.useBeforeClaimPrize) {
        recipient = hooks.implementation.beforeClaimPrize{gas: gasLimit}(_winner, _tier, _prizeIndex, _fee, _feeRecipient);
    } else {
        recipient = _winner;
    }
    ...

    if (hooks.useAfterClaimPrize) {

```

```

        hooks.implementation.afterClaimPrize{gas: gasLimit}( //@au
            _winner,
            _tier,
            _prizeIndex,
            prizeTotal - _fee,
            recipient
        );
    }

    return prizeTotal;
}

```

asselstine (PoolTogether) confirmed and commented:

Adding an internal gas limit and then catching the revert would be ideal; so the claimer won't be grieved and can detect badly-written hooks.

Picodes (judge) commented:

The issue here, is not the possible DoS, as a claimer can just skip one user, but the possibility of the grieving attack by, for example, front-running by a malicious hook. I'll give partial credit to duplicates if there is too much focus on DoS.

PoolTogether mitigated:

Added hook gas limits and error handling.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/21>

Status: Not fully mitigated. Full details in reports from [dirk_y](#), [rvierdiiev](#) and [OxStalin](#), and in the [Mitigation Review](#) section below.

🔗

[M-03] `TwabLib::getTwabBetween` can return inaccurate balances if `_startTime` and `_endTime` aren't safely bound

Submitted by [Infect3d](#), also found by [Aymen0909](#)

Here's the documentation of the `TwabLib::getTwabBetween` function:

File: twab-controller\src\libraries\TwabLib.sol

```
278:  /**
279:   * @notice Looks up a users TWAB for a time range.
280:   * @dev If the timestamps in the range are not exact match
281:   * @dev Ensure timestamps are safe using isTimeRangeSafe.
282:   * @param _observations The circular buffer of observations
283:   * @param _accountDetails The account details to query with
284:   * @param _startTime The start of the time range
285:   * @param _endTime The end of the time range
286:   * @return twab The TWAB for the time range
287:   */
288:  function getTwabBetween( //@audit - M01: dev comment (is'
289:      uint32 PERIOD_OFFSET,
290:      ObservationLib.Observation[MAX_CARDINALITY] storage _ob:
291:      AccountDetails memory _accountDetails,
292:      uint32 _startTime,
293:      uint32 _endTime
294:  ) internal view returns (uint256) {
```

TwabLib::getTwabBetween -function gets called in. Then,

PrizePool::_getVaultUserBalanceAndTotalSupplyTwab gets called by

PrizePool::claimPrize, returning twab for a specified time range, which finally is called by PrizePool::isWinner in PrizePool::claimPrize.

The thing is, as explained by [the documentation](#), values from balance history aren't guaranteed to be accurate if the timestamp at which time it is queried do not fall between:

- The newest observation in a period.
- The end of that period (period must have ended).

That is because until a period has ended, we cannot be sure the value we are querying hasn't changed in the future (from pov of the timestamp).



Impact

This means, that if a prize is claimed, but timestamps at which the twab is getting requested do not follow these rules, the Prize Pool contract might distribute prizes based on inaccurate data. This could lead to the prizes being distributed to users who should not have won, or not being distributed to users who should have won.

File: prize-pool\src\PrizePool.sol

```
920:     function _getVaultUserBalanceAndTotalSupplyTwab(  
921:         address _vault,  
922:         address _user,  
923:         uint256 _drawDuration  
924:     ) internal view returns (uint256 twab, uint256 twabTotalSupply)  
925:     {  
926:         uint32 _endTimeStamp = uint32(_lastClosedDrawStartedAt + _drawDuration);  
927:         uint32 _startTimeStamp = uint32(_endTimeStamp - _drawDuration);  
928:         // @audit - dev comment for TwabLib::getTwabBetween does not seem to be followed  
929:         twab = twabController.getTwabBetween(_vault, _user, _startTimeStamp, _endTimeStamp);  
930:         twabTotalSupply = twabController.getTotalSupplyTwabBetween(_vault, _user,  
931:             _startTimeStamp, _endTimeStamp);  
932:     }  
933: }
```



Recommended Mitigation Steps

Add a `isTimeRangeSafe` check (will depend on how devs want to handle it, either a `revert` or another error management solution) before `getTwabBetween` to ensure queried balances are accurate; otherwise, prize distribution could be unfair.

```
function _getVaultUserBalanceAndTotalSupplyTwab(  
    address _vault,  
    address _user,  
    uint256 _drawDuration  
) internal view returns (uint256 twab, uint256 twabTotalSupply)  
{  
    uint32 _endTimeStamp = uint32(_lastClosedDrawStartedAt + _drawDuration);  
    uint32 _startTimeStamp = uint32(_endTimeStamp - _drawDuration);  
    // @audit dev comment doesn't seem to be followed in PrizePool  
    // @audit added isTimeRangeSafe before getTwabBetween  
    require(twabController.isTimeRangeSafe(_vault, _user, _startTimeStamp, _endTimeStamp));  
    twab = twabController.getTwabBetween(_vault, _user, _startTimeStamp, _endTimeStamp);  
  
    twabTotalSupply = twabController.getTotalSupplyTwabBetween(  
        _vault,  
        _startTimeStamp,  
        _endTimeStamp  
    );  
}
```

}



Assessed type

Invalid Validation

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Ensure search timestamps are on or at period end timestamps.

PR: <https://github.com/GenerationSoftware/pt-v5-twab-controller/pull/5>

Status: Mitigation confirmed. Full details in reports from [dirk_y](#) and [rvierdiiev](#).



[M-04] The deposit function does not check for the `maxMint` amount

Submitted by [ni8mare](#), also found by [ni8mare](#), [hals](#), and [shaka](#)

It is theoretically possible for the deposit amount to mint shares more than the `maxMint` amount



Proof of Concept

The deposit function has a check for `maxDeposit` and reverts if the deposit value is more than `max(uint96)`. But, it does not check the shares to be less than `maxMint` amount and hence, bypasses the check. Theoretically, if the assets are equal to `max(uint96)` and if the vault is under-collateralized, the ratio of `_assetUnit` to `_exchangeRate` is greater than or equal to 2, then the calculation in `_convertToShares`.

Function `_assets.mulDiv(_assetUnit, _exchangeRate, _rounding);` could return a value more than the `maxMint` amount. This is possible in scenarios where the `_assetUnit` is a big enough number (as there is no limit on the decimals of the underlying asset, and `_assetUnit = 10 ** super.decimals()`) and the Vault is severely under-collateralized.



Recommended Mitigation Steps

Include the `maxMint` check in the deposit function to prevent this problem.



Assessed type

Invalid Validation

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Fixed 4626 compliance.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/11>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-05] Balance invariant between the individual and total `twabs` can be broken

Submitted by [minhtrng](#)

An edge case in the `TwabController._transferBalance` can cause the total balance for a vault account to decrease; although, it did not actually decrease. This will cause the sum of the individual `delegateBalances` for a vault to be greater than the registered total for that vault. This again will skew the odds in favor of winning a price, causing the reserve to be drained faster over the time intended.



Proof of Concept

The `TwabController._transferBalance` function handles the case incorrectly, where `_to` is equal to the `SPONSORSHIP_ADDRESS`. First, assume the `_from` address is an address that has a balance and delegates to itself (the default). Then, the function will decrease the balance and `delegateBalance` of `_from`. It will also decrease the total balances of the vault account (note: that `_toDelegate` will be `SPONSORSHIP_ADDRESS` due to default):

```

if (_from != address(0)) {
    bool _isFromDelegate = _fromDelegate == _from;

    _decreaseBalances(_vault, _from, _amount, _isFromDelegate

if (!_isFromDelegate && _fromDelegate != SPONSORSHIP_ADDRESS)
    _decreaseBalances(_vault, _fromDelegate, 0, _amount);
}

if (
    _to == address(0) ||
    (_toDelegate == SPONSORSHIP_ADDRESS && _fromDelegate != SPONSORSHIP_ADDRESS)
) {
    _decreaseTotalSupplyBalances(
        _vault,
        _to == address(0) ? _amount : 0,
        (_to == address(0) && _fromDelegate != SPONSORSHIP_ADDRESS)
        ? (_toDelegate == SPONSORSHIP_ADDRESS && _fromDelegate
            ? _amount
            : 0
        )
    );
}
}

```

Then, the balance and `delegateBalance` of `SPONSORSHIP_ADDRESS` will be increased by the same amount. This is not supposed to happen in the first place, as this address is not meant to have any balances. However, the issue described is due to the fact that the total balances of the vault account will not be adjusted, because the condition `_toDelegate != SPONSORSHIP_ADDRESS` is not met:

```

if (_to != address(0)) {
    bool _isToDelegate = _toDelegate == _to;

    _increaseBalances(_vault, _to, _amount, _isToDelegate ? _amount : 0);

if (!_isToDelegate && _toDelegate != SPONSORSHIP_ADDRESS)
    _increaseBalances(_vault, _toDelegate, 0, _amount);
}

if (
    _from == address(0) ||
    (_fromDelegate == SPONSORSHIP_ADDRESS && _toDelegate != SPONSORSHIP_ADDRESS)
) {

```

```

        _increaseTotalSupplyBalances(
            _vault,
            _from == address(0) ? _amount : 0,
            (_from == address(0) && _toDelegate != SPONSORSHIP_ADDRESS)
            ? (_fromDelegate == SPONSORSHIP_ADDRESS && _toDelegate == address(0)
            ? _amount
            : 0
            : 0
        );
    }
}

```

The ratio of `_userTwab` and `_vaultTwabTotalSupply` plays a role when determining a winner in `TierCalculationLib.isWinner` and the underlying model assumes the invariant to hold true:

```

uint256 constrainedRandomNumber = _userSpecificRandomNumber %
uint256 winningZone = calculateWinningZone(_userTwab, _vaultTwabTotalSupply);

```

If the sum of the individual `twabs` is higher than the sum, there will be more winners than intended, causing the described drainage of the reserve.



Recommended Mitigation Steps

Disallow `_to` to be `SPONSORSHIP_ADDRESS`.



Assessed type

Invalid Validation

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Reject transfers to the sponsorship address.

PR: <https://github.com/GenerationSoftware/pt-v5-twab-controller/pull/6>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-06] drawManager can be set to a malicious address

Submitted by [Udsen](#), also found by [catellatech](#), [Inspektor](#), [squeaky_cactus](#), [OxPsuedoPandit](#), [Ox11singh99](#), [Tripathi](#), [Nyx](#), [Daniel526](#) (1, 2), [namx05](#), [Jorgect](#), [Praise](#) (1, 2), [xuwinnie](#), [YY](#), and [Darwin](#)

In the `PrizePool.constructor` the `drawManager` state variable is set as follows:

```
drawManager = params.drawManager;
```

But it accepts, even if the `params.drawManager == address(0)`. There is no input validation for the `address(0)` as well.

The `PrizePool.setDrawManager` function is used to set the `drawManager` if not already set. But the problem here, is that there is no access control for this function and anyone can call this. A malicious user can front-run the valid `drawManager` assignment transaction and set a malicious address as the `drawManager`. The other issue, is once the `drawManager` is set, it can not be updated due to the following conditional check. Hence, the contract will have to redeployed.

```
if (drawManager != address(0)) { //@audit-issue - can be front r
    revert DrawManagerAlreadySet();
}
```

The following two functions controlled by the `onlyDrawManager` modifier will be vulnerable to attacks, since the `drawManager` is a malicious address now.

1. `PrizePool.withdrawReserve` can withdraw tokens from the `reserve` to any address given in the `_to` parameter. Hence, this function will be vulnerable.
2. The `PrizePool.closeDraw()` function can close the current open draw. This function is only callable by the `drawManager` function. A malicious user can get control of this function, thus making it vulnerable.



Proof of Concept

```
drawManager = params.drawManager;
if (params.drawManager != address(0)) {
    emit DrawManagerSet(params.drawManager);
}
```

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L278-L281>

```
function setDrawManager(address _drawManager) external {
    if (drawManager != address(0)) {
        revert DrawManagerAlreadySet();
    }
    drawManager = _drawManager;

    emit DrawManagerSet(_drawManager);
}
```

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L299-L306>

```
function closeDraw(uint256 winningRandomNumber_) external only!
```

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L348>

```
function withdrawReserve(address _to, uint104 _amount) external {
    if (_amount > _reserve) {
        revert InsufficientReserve(_amount, _reserve);
    }
    _reserve -= _amount;
    _transfer(_to, _amount);
    emit WithdrawReserve(_to, _amount);
}
```

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L335-L342>



Tools Used

VSCode



Recommended Mitigation Steps

It is recommended to check for `address(0)` in the constructor when `drawManager` is set; or to implement `access control` in the `PrizePool.setDrawManager` function, so that only the admin of the contract can call this function to set the `drawManager`, if it is not set already.



Assessed type

Invalid Validation

PoolTogether confirmed via discord communication

PoolTogether mitigated:

Record deployer and permission draw manager.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/31>

Status: Mitigation confirmed. Full details in reports from [rvierdiiev](#), [dirk_y](#) and [OxStalin](#).



[M-07] In important libraries of PoolTogether, the `pow()` function of `PRBMath` is used, which exhibits inconsistent return values

Submitted by [catellatech](#), also found by [Tripathi](#) and [nadin](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/libraries/DrawAc>

[cumulatorLib.sol#L406-L407 https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/libraries/TierCalculationLib.sol#L26](https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/libraries/TierCalculationLib.sol#L26)



Impact

The `DrawAccumulatorLib.sol` and `TierCalculationLib.sol` libraries inherit a version of `PRBMath` that contains a critical vulnerability in the `pow()` function, which can return inconsistent values. This vulnerability is of great importance to the PoolTogether protocol, as the `pow()` function is used in the computation of `TierCalculationLib.getTierOdds` and `DrawAccumulatorLib.computeC`.

Recently, another protocol has also experienced the same bug and the creators of the `PRBMath` have acknowledged this situation. Here is the corresponding [request](#). Due to time constraints, we were unable to thoroughly address certain rounding errors with the `mul` and `div` functions of `SD59x18`. However, these errors have been corrected in `PRBMath V4`.



Proof of Concept

[Proof of the bug acknowledgment by the creator of the PRBMath](#)

This PR makes four significant changes:

- Upgrades to `PRBMath v4`.
- Bumps all other submodules to their most recent versions.
- Increases the minimum Solidity pragma to `v0.8.19`, since this is a requirement of `PRBMath V4`.



Recommended Mitigation Steps

To mitigate this issue, please update the contracts to `0.8.19` and upgrade the `PRBMath` to version `V4`.



Assessed type

Math

[asselstine \(PoolTogether\) confirmed](#)

Picodes (judge) decreased severity to Medium

PoolTogether mitigated:

Upgrade PRBMath to v4 and Solidity to 0.8.19.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/commit/ba56ea8bac3bce06f1e08ae071a19954dd720b1f>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-08] An attacker can front-run `deployVault` to deploy at the same address

Submitted by [gzeon](#), also found by [Inspektor](#), [Breeje](#), [OxMirce](#), and [ptsanev](#)

Vaults are created from the factory via `CREATE1`. An attacker can front-run `deployVault` to deploy at the same address, but with different config. If the deployed chain reorg, a different vault might also be deployed at the same address.



Proof of Concept

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/VaultFactory.sol#L67-L78>

1. Bob setup a bot to monitor the `mempool` when PT deploys a new vault.
2. Bob's bot saw a deployment by PT at `0x1234` and fires a tx to deposit immediately.
3. Alice front-runs PT's deployment by deploying a malicious vault at `0x1234`.
4. Bob's transaction ended up deposited into Alice's malicious vault.



Recommended Mitigation Steps

Use `CREATE2` and the vault config as salt.



Assessed type

asselstine (PoolTogether) disputed and commented:

The Vault address is a derivative of the (sender address, nonce). I don't see how this scenario is possible?

Picodes (judge) commented:

@asselstine - exactly. Here, it only depends on the nonce of the factory; so in case of reorg, someone could "override" a vault deployment and all following transactions would still be executed.

PoolTogether mitigated:

Used `CREATE2` for `VaultFactory`.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/25>

Status: Mitigation confirmed. Full details in reports from [dirk_y](#), [rvierdiev](#) and [OxStalin](#).



[M-09] High Prizes might not be claimed

Submitted by [markus_ether](#), also found by [josephdara](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/57a381aef690a27c9198f4340747155a71cae753/src/Claimer.sol#L210>
<https://github.com/GenerationSoftware/pt-v5-claimer/blob/57a381aef690a27c9198f4340747155a71cae753/src/Claimer.sol#L173>



Impact

PoolTogether V5 delegates the task of claiming the prizes to a network of `Claimers` that are incentivized to claim prizes for a share of the prize won. The fees the Claimant receives is calculated based on a Dutch Auction, but then limited by the minimum prize of the prize pool.

`fee = min (fee in auction = % maxFeePortionOfPrize * minPrize).`

When the gas costs exceed the `minPrize` , the solver has no incentives to claim the price. Notice that the number of prizes only adjusts *after* very few prizes are claimed.



Proof of Concept

Gas prices will then rise well above the `minPrize` . The highest prize is suddenly drawn that day. The solvers have no incentive to claim the prize and the winner does not monitor the contract. As a result, no one claims the top prize. The protocol suffers a loss of reputation.



Tools Used

VS Code



Recommended Mitigation Steps

The problem is that the incentives are not exactly aligned. The user is willing to pay anything up to their price to receive their price. But the maximum the solver receives is the minimum price. We can align the two incentives by using each user's price as an upper bound.

```
function _computeMaxFee(uint8 _tier, uint8 _numTiers) internal view returns (uint256) {
    uint8 _canaryTier = _numTiers - 1;
    if (_tier != _canaryTier) {
        // canary tier
        return _computeMaxFee(prizePool.getTierPrizeSize(_canaryTier), _numTiers);
    } else {
        return _computeMaxFee(prizePool.getTierPrizeSize(_tier), _numTiers);
    }
}
```

Since we already validate the inputs in `claimPrize` , the attacker can not claim more than the prize is worth.

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Made `maxFee` a function of the tier's prize size, not the smallest prize size.

PR: <https://github.com/GenerationSoftware/pt-v5-claimer/pull/7>

Status: Mitigation error. Full details in report from [dirk_y](#), and in the [Mitigation Review](#) section below.



[M-10] PrizePool -> Winners wouldn't be able to claim prize correctly in `claimPrize` function

Submitted by [mahyar](#)

In `PrizePool.sol` vaults can call the `claimPrize()` function to claim prizes for winners. The `claimPrize` calls `_getTier(_tier, numberOfTiers)` function to get back the `prizeSize` but there is a problem in the `_getTier` function:

```
function _getTier(uint8 _tier, uint8 _numberOfTiers) internal returns (
    Tier memory tier = _tiers[_tier];
    uint16 _lastClosedDrawId = lastClosedDrawId;
    if (tier.drawId != _lastClosedDrawId) {
        tier.drawId = _lastClosedDrawId;
        tier.prizeSize = uint96(
            _computePrizeSize(
                _tier,
                _numberOfTiers,
                fromUD34x4toUD60x18(tier.prizeTokenPerShare),
                fromUD34x4toUD60x18(prizeTokenPerShare)
            )
        );
    }
    return tier;
}
```

As you can see, it calls the `_computePrizeSize` function and downcasts the returned result. Since it returns `uint256` by downcasting it to `uint96`, it's possible to result in the incorrect number in the value overflow. When downcasting from one type to another, Solidity will not revert, but overflows. This means, there is a possibility the `prizeSize` value will result in the incorrect prize size and vaults can't claim the prizes correctly for winners.



Recommended Mitigation Steps

Check whether the result of `prizeSize` exceeds `type(uint96).max` or not, if so, revert the tx. I recommend to use [SafeCast](#) by OpenZeppelin for casting different types safely.

I also found some other downcastings in the codebase, but since they are pegged to vault share values and you already check for overflows, there is no problem.



Assessed type

Under/Overflow

[asselstine \(PoolTogether\) confirmed and commented:](#)

Hmm; looks like the Tier struct has an extra 16 bits available to pack. Will increase `prizeSize` type to `uint112` and add safe casting.

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

Downgrading to Med as there is no proof of impact or discussion about the chances of this occurring in the report.

[asselstine \(PoolTogether\) commented:](#)

Come to think of it, since prizes are in POOL and its supply is 10m with 18 decimals, then prizes can never be more than $\sim 1e25$.

[PoolTogether mitigated:](#)

Improved handling of prize size overflow.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/16>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#) and [dirk_y](#).



[M-11] `Vault.mintWithPermit()` can be DoS'd

Submitted by [KupiaSec](#), also found by [dirk_y](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-vault/tree/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L466-L469> <https://github.com/GenerationSoftware/pt-v5-vault/tree/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L971-L974> <https://github.com/GenerationSoftware/pt-v5-vault/tree/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L882-L887>



Impact

`Vault.mintWithPermit()` uses a signature to approve the underlying asset. But the asset amount can be changed easily, so this method can be reverted and might be DoS'd.



Proof of Concept

`Vault.mintWithPermit()` gets the share amount as an input and calculates the asset amount from the share. Then, it approves the asset amount with `permit` method.

```
uint256 _assets = _beforeMint(_shares, _receiver);

_permit(IERC20Permit(asset()), msg.sender, address(this), _a
_deposit(msg.sender, _receiver, _assets, _shares);
```

The signature is generated using the exact value of the expected asset amount calculated from the share amount, and the resulting asset amount depends on the exchange rate of current vault.

```
function _beforeMint(uint256 _shares, address _receiver) internal
    if (_shares > maxMint(_receiver)) revert MintMoreThanMax(_re
    return _convertToAssets(_shares, Math.Rounding.Up);
}
```

```
function _convertToAssets(
    uint256 _shares,
```

```
Math.Rounding _rounding
) internal view virtual override returns (uint256) {
    return _convertToAssets(_shares, _currentExchangeRate(), _ro
```

The resulting asset amount can be different from the value of transaction start time. Even an adversary can front-run and manipulate the exchange rate. If the resulting asset amount is different from the original one, the signature will not work as expected and `mintWithPermit()` will revert in most cases.



Recommended Mitigation Steps

We can input an upper bound of the asset amount instead of the exact value of the asset amount.



Assessed type

DoS

[asselstine \(PoolTogether\) confirmed](#)

| assestine marked the issue as sponsor confirmed

[PoolTogether mitigated:](#)

| Removed `mintWithPermit` .

| PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/15>

Status: Mitigation confirmed. Full details in reports from [dirk_y](#), [rvierdiev](#) and [OxStalin](#).



[M-12] The tier odds in `TieredLiquidityDistributor` are incorrect

Submitted by [dirk_y](#)

The following points are stated in the docs:

1. The highest standard prize tier is the most common prize: occurring every single draw.
2. The canary tier has odds of occurring daily (as if it were the highest tier).

Therefore, both the highest standard tier and the canary tier should have odds of 1; however, only the canary tier has odds of 1. The result is that the protocol is not distributing prizes as intended.



Proof of Concept

The issue with the odds calculations can be seen in the

`TieredLiquidityDistributor.sol` contract. Below is a short snippet for the tier odds when there are 3 tiers:

```
SD59x18 internal constant TIER_ODDS_0_3 = SD59x18.wrap(27397260  
SD59x18 internal constant TIER_ODDS_1_3 = SD59x18.wrap(52342390  
SD59x18 internal constant TIER_ODDS_2_3 = SD59x18.wrap(10000000
```

The canary tier odds are 1 as intended; however, the highest normal prize tier odds are incorrect. If the tier odds were being calculated correctly, we should see the last two odds for each number of tiers being 1 (i.e. `10000000000000000000`).



Recommended Mitigation Steps

Since the `TierCalculationLib.getTierOdds` function only seems to be used by the `GenerateConstants` script, I suggest modifying the script used to generate the constants that are placed in `TieredLiquidityDistributor.sol`:

```
diff --git a/script/generateConstants.s.sol b/script/generateConstants.s.sol  
index fe1d4ed..fb9f90a 100644  
--- a/script/generateConstants.s.sol  
+++ b/script/generateConstants.s.sol  
@@ -38,16 +38,24 @@ contract GenerateConstants is Script {  
    MAX_TIERS = 15;  
    // Precompute the odds for each tier  
    for (uint8 numTiers = MIN_TIERS; numTiers <= MAX_TIERS; numTiers++) {  
-        for (uint8 tier = 0; tier < numTiers; tier++) {  
+        for (uint8 tier = 0; tier < numTiers - 1; tier++) {  
            console.log(
```

```

        "SD59x18 internal constant TIER_ODDS_%d_%d = SD59x18.1
        uint256(tier),
        uint256(numTiers),
        uint256(
-            SD59x18.unwrap(TierCalculationLib.getTierOdds(tier,
+            SD59x18.unwrap(TierCalculationLib.getTierOdds(tier,
        )
        );
    }
+    console.log(
+        "SD59x18 internal constant TIER_ODDS_%d_%d = SD59x18.1
+        uint256(numTiers - 1),
+        uint256(numTiers),
+        uint256(
+            SD59x18.unwrap(TierCalculationLib.getTierOdds(numTi
+        )
+    );
    }
}
}

```



Assessed type

Math

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Recomputed Tier odds, reduced the number of tiers and made grand prize period draws configurable.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/24>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-13] Users can manipulate the observation creation

Submitted by [Nyx](#)

Users can alter the record of their balances.



Proof of Concept

If the users `delegateBalance` has changed, observations will be recorded. If there is a need for a new observation, the new observation will be created; otherwise, it will be overwritten.

A new observation creation will be decided in the `_getNextObservationIndex()` function.

```

if (currentPeriod == 0 || currentPeriod > newestObservationPeriod)
    return (
        uint16(RingBufferLib.wrap(_accountDetails.nextObservationIndex,
        newestObservation,
        true
    ));
}

```

`newestObservationPeriod` is the last observations period. `currentPeriod` is a period that calculated with `uint32 currentTime = uint32(block.timestamp)`.

```

uint32 currentPeriod = _getTimestampPeriod(PERIOD_LENGTH, PERIOD_OFFSET,
uint32 newestObservationPeriod = _getTimestampPeriod(
    PERIOD_LENGTH,
    PERIOD_OFFSET,
    newestObservation.timestamp
);

```

```

function _getTimestampPeriod(
    uint32 PERIOD_LENGTH,
    uint32 PERIOD_OFFSET,
    uint32 _timestamp
) private pure returns (uint32 period) {
    if (_timestamp <= PERIOD_OFFSET) {
        return 0;
    }
    // Shrink by 1 to ensure periods end on a multiple of PERIOD_LENGTH.
    // Increase by 1 to start periods at # 1.
    return ((_timestamp - PERIOD_OFFSET - 1) / PERIOD_LENGTH) + 1;
}

```

The problem is, if someone deposits a small amount frequently, `currentPeriod` and `newestObservationPeriod` will always be the same and a new observation won't be created. Attackers can keep doing this until `closeDraw` and manipulate their balances.

According to the docs, if a draw were to start and end within a period, a user would be able to alter the record of their balance for that draw by overwriting an Observation.

It is important to note that due to Observation overwriting, average balances for a period are not finalized until a period ends. Therefore, if a draw ends but a period has not, a user would be able to manipulate their average balance for that final period of time after the draw ends. This would result in an inaccurate record of their balance held during the draw.

[asselstine \(PoolTogether\) confirmed](#)

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

If someone deposits a small amount frequently, `currentPeriod` and `newestObservationPeriod` will always be the same and a new observation won't be created This only works for small deposit within the same `PERIOD_LENGTH`; otherwise, timestamp roundings will differ.

This report shows how by leveraging the fact that new observations are not created if the previous observations falls within the same period, an attacker could modify its average balance for the final period if a draw ends within a period.

[PoolTogether mitigated:](#)

Aligned twab queries on period boundaries.

PR: <https://github.com/GenerationSoftware/pt-v5-twab-controller/pull/5>

Status: Mitigation confirmed with comments. Full details in reports from [dirk_y](#), [OxStalin](#) and [rvierdiiev](#).

[M-14] Number of prize tiers always increases if just 1 canary prize is claimed

Submitted by [dirk_y](#), also found by [bin2chen](#), [xuwinnie](#), and [volodya](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L361> <https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L446-L448> <https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L781-L810>



Impact

As mentioned in the docs:

“The Canary Tier is a special prize tier that receives a smaller amount of prize liquidity. The Canary Tier informs the Prize Pool as to whether it’s worth increasing the number of prize tiers. The canary tier’s role is to let us know whether it’s worth offering $n+1$ tiers of prizes”.

The intended behaviour is that the number of tiers only increases if a high enough portion of both the highest non-canary tier prizes and the canary tier prizes are claimed, so that prizes scale with demand and liquidity. However, there is a bug that causes the number of tiers to increase if at least 1 canary prize is claimed. Therefore, it is highly likely that it will take just 12 draws to reach the cap of 15 prize tiers (or an attacker could force the situation by claiming canary prizes); at which point the prize sizes will be broken, based on the liquidity provided and the protocol will become almost unusable.



Proof of Concept

When a draw has finished/elapsed, the draw manager closes the draw by calling `closeDraw`. During this closing process the next number of tiers is computed:

```
_nextNumberOfTiers = _computeNextNumberOfTiers(_numTiers);
```


Before we dive into this computation, it's worth exploring the `claimPrize` method. Whenever a prize is claimed, there is a `largestTierClaimed` state variable that is set to the highest tier that has been claimed for; which can include the canary tier:

```
if (largestTierClaimed < _tier) {
    largestTierClaimed = _tier;
}
```

Now, the issue titled in this report exists because of how the next number of tiers is calculated in the `_computeNextNumberOfTiers` function:

```
function _computeNextNumberOfTiers(uint8 _numTiers) internal view
    UD2x18 _claimExpansionThreshold = claimExpansionThreshold;

    uint8 _nextNumberOfTiers = largestTierClaimed + 2; // canary
    _nextNumberOfTiers = _nextNumberOfTiers > MINIMUM_NUMBER_OF_TIERS
        ? _nextNumberOfTiers
        : MINIMUM_NUMBER_OF_TIERS;

    if (_nextNumberOfTiers >= MAXIMUM_NUMBER_OF_TIERS) {
        return MAXIMUM_NUMBER_OF_TIERS;
    }

    // check to see if we need to expand the number of tiers
    if (
        _nextNumberOfTiers >= _numTiers &&
        canaryClaimCount >=
            fromUD60x18(
                intoUD60x18(_claimExpansionThreshold).mul(_canaryPrizeCount)
            ) &&
        claimCount >=
            fromUD60x18(
                intoUD60x18(_claimExpansionThreshold).mul(toUD60x18(_estimatedClaimCount))
            )
    ) {
        // increase the number of tiers to include a new tier
        _nextNumberOfTiers = _numTiers + 1;
    }

    return _nextNumberOfTiers;
}
```

For the sake of argument, let's say the prize pool had 3 tiers, so the canary tier is tier 2. If there is at least 1 canary tier prize that has been claimed for the previous draw, then `largestTierClaimed = 2` ; therefore, `_nextNumberOfTiers = 4` . What's interesting here, is that even if the check to expand the number of tiers fails, we still return `_nextNumberOfTiers` , which has just been set to 4. So we're expanding the number of tiers even if the claim count isn't higher than the claim expansion thresholds.

I have made a tiny change to the existing test suite to demonstrate that the number of tiers will increase with just 1 canary prize claim. It can be executed with `forge test -vvv --match-path test/PrizePool.t.sol` :

```
diff --git a/test/PrizePool.t.sol b/test/PrizePool.t.sol
index 45d8606..36063eb 100644
--- a/test/PrizePool.t.sol
+++ b/test/PrizePool.t.sol
@@ -561,25 +561,12 @@ contract PrizePoolTest is Test {
    function testCloseAndOpenNextDraw_expandingTiers() public {
        contribute(1e18);
        closeDraw(1234);
-       mockTwab(address(this), address(this), 0);
-       claimPrize(address(this), 0, 0);
-       mockTwab(address(this), sender1, 1);
-       claimPrize(sender1, 1, 0);
-       mockTwab(address(this), sender2, 1);
-       claimPrize(sender2, 1, 0);
-       mockTwab(address(this), sender3, 1);
-       claimPrize(sender3, 1, 0);
-       mockTwab(address(this), sender4, 1);
-       claimPrize(sender4, 1, 0);
+
+       // Just 1 canary prize tier claim increases the number of t.

        // canary tiers
        mockTwab(address(this), sender5, 2);
        claimPrize(sender5, 2, 0);
-       mockTwab(address(this), sender6, 2);
-       claimPrize(sender6, 2, 0);
-
-       vm.expectEmit();
-       emit DrawClosed(2, 245, 3, 4, 7997159090909503, UD34x4.wrap

        closeDraw(245);
```

```
assertEq(prizePool.numberOfTiers(), 4);
```



Tools Used

Foundry



Recommended Mitigation Steps

The `_computeNextNumberOfTiers` logic should be updated to the below:

```
diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..016124a 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -791,19 +791,22 @@ contract PrizePool is TieredLiquidityDistr
     }

    // check to see if we need to expand the number of tiers
-   if (
-       _nextNumberOfTiers >= _numTiers &&
-       canaryClaimCount >=
-       fromUD60x18(
-           intoUD60x18(_claimExpansionThreshold).mul(_canaryPrizeCo
-       ) &&
-       claimCount >=
-       fromUD60x18(
-           intoUD60x18(_claimExpansionThreshold).mul(toUD60x18(_es
-       )
-   ) {
-       // increase the number of tiers to include a new tier
-       _nextNumberOfTiers = _numTiers + 1;
+   if (_nextNumberOfTiers > _numTiers) {
+       if (canaryClaimCount >=
+           fromUD60x18(
+               intoUD60x18(_claimExpansionThreshold).mul(_canaryPrize
+           ) &&
+           claimCount >=
+           fromUD60x18(
+               intoUD60x18(_claimExpansionThreshold).mul(toUD60x18(_
+           )
+       ) {
+           // increase the number of tiers to include a new tier
+           return _numTiers + 1;
+       }
+   }
```

```
+         else {  
+             return _numTiers;  
+         }  
+     }  
  
    return _nextNumberOfTiers;
```



Assessed type

Math

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Superseceded by the simplification of tier expansion logic.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/17>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-15] Tiers can be maintained active to give unfair advantage to user through DoS

Submitted by [Ox3e84fa45](#)

Tiers can be maintained active by claiming a single prize of the last tier at a loss. The protocol will close tiers of the next draw based on the [largest tier claimed](#) of the draw being closed, which is [registered during the claiming of the prizes](#).

This is a symptom that liquidity is being spread too thin on the last tier, and is no longer profitable for bots to claim prizes for users since their gas costs are over the claiming fees they earn.

The closing of a tier is meant to increase `maxFee`, which is the highest price a claiming fee can achieve in the reverse Dutch Auction mechanism; which in the current draw, isn't enough for the bot to be profitable.

The `maxFee` is computed based on the last active tier's prize size, and by lowering the amount of tiers, the liquidity is going to get concentrated in less tiers and in less prizes (since the number of prizes is $4^{\text{tierNumber}}$); thus, making the bots profitable again and incentivizing them to claim prizes for users.

During the draw where there are no incentives for bots to claim prizes, prizes won't get claimed; which means that if someone won a high prize from a low tier, they would have to manually claim it. Otherwise, the liquidity of that prize will continue onto the next draw, leaving the user without a possibility of claiming their prize once the draw has passed.

A malicious actor that has deposited liquidity in the vaults that contribute to the prize pool could take advantage of this mechanic.

By having a bot that in this closing tier state claim their prizes when available and claim a single prize from the highest tier, they are able to maintain the tier active; which will cause following draws to fall again into the state where bots don't claim prizes. This hugely decreases the probability of winning for users that don't have a claiming bot while giving the malicious actor an unfair advantage.



Proof of Concept

`Draw 0` : `closeDraw` has just been executed and the remaining liquidity from the previous draw with the newly added liquidity for this draw computes a `maxFee` that isn't enough to cover gas costs for bots to claim prizes. Therefore, the last tier is supposed to be closed. A malicious actor can claim a random single prize from the last tier at a loss to maintain the tier active, also claiming any prizes they may have won.

`Draw 1` : the new liquidity is added to the system, which will make the last tier's prize size increase, and subsequently, the `maxFee` will increase. Depending on how much liquidity is added, `maxFee` will or won't be enough for bots to start claiming prizes, since the last tier has accumulated the liquidity of two draws.

It could be the case that the added liquidity isn't enough and creates a draw state similar to `draw 0`. To continue with the example, let's assume that the accumulated liquidity of two draws computes a `maxFee` that is enough to incentivize bots to claim prizes.

`Draw 1` behaves as expected, which means that the last tier's liquidity would get claimed, and the following draw would rely mostly on the newly added liquidity. "Mostly" since there will be a remanent part of liquidity from the prizes that haven't been won.

`Draw 2` : The situation of `draw 0` repeats itself and the added liquidity on this draw is not concentrated enough to incentivize bots to claim prizes. The malicious actor claims a single prize at a loss again, to keep the tier active, also claiming any prizes they may have won.

This results in a situation where, depending on how much liquidity is added to each draw, many draws could result as inactive. Meaning, that unless users claim their own prizes, no one is going to claim them; while the malicious actor has an advantage, since their bot will claim their prizes, apart from maintaining the highest tier active.



Recommended Mitigation Steps

Change the mechanism by which `largestTierClaimed` is set, so that it's resistant to a single user performing a claim.



Assessed type

DoS

[asselstine \(PoolTogether\) confirmed](#)

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

It seems to me that Medium severity is more appropriate here, as the profitability and the damage done by this attack scenario are hypothetical. It assumes that it is worth claiming prizes at a loss of, say `maxFeePortionOfPrize * prize`, to increase the attacker's chances of winning. The damage is that users not claiming their prize (which requires multiple winners, as the attacker is claiming one price) will lose it.

[PoolTogether mitigated:](#)

Improved tier shrinking, retention, and expansion logic.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/17>

Status: Mitigation confirmed. Full details in reports from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-16] The threshold check for adding of new tiers is skipped when `_nextNumberOfTiers` is at the maximum amount

Submitted by [yixxas](#)

The wrong implementation allows a new tier to be added as the threshold check is not done when we currently have 14 tiers. This means, that with just a single canary claim, a new tier will be added when `number of tiers = 14`.



Proof of Concept

In order for the prize pool to add new tiers, the normal prize claim count and canary tier claim count must pass a certain threshold. This check is done by the below code snippet:

[PrizePool.sol#L794-L804](#)

```
if (
    _nextNumberOfTiers >= _numTiers &&
    canaryClaimCount >=
        fromUD60x18(
            intoUD60x18(_claimExpansionThreshold).mul(_canaryPrizeCo
        ) &&
    claimCount >=
        fromUD60x18(
            intoUD60x18(_claimExpansionThreshold).mul(toUD60x18(_est
        )
    ) {
```

However, in the case where `_nextNumberOfTiers >= MAXIMUM_NUMBER_OF_TIERS`, `MAXIMUM_NUMBER_OF_TIERS` this is instantly returned, skipping this check.

[PrizePool.sol#L784C1-L791C6](#)

```
uint8 _nextNumberOfTiers = largestTierClaimed + 2; // canary
```

```

_nextNumberOfTiers = _nextNumberOfTiers > MINIMUM_NUMBER_OF_
? _nextNumberOfTiers
: MINIMUM_NUMBER_OF_TIERS;

if (_nextNumberOfTiers >= MAXIMUM_NUMBER_OF_TIERS) {
    return MAXIMUM_NUMBER_OF_TIERS;
}

```

As seen from the above code, if we currently have 14 tiers and the canary tier is claimed in this draw, hence `largestTierClaimed = 13`, then we have

`_nextNumberOfTiers = 15`. Since `MAXIMUM_NUMBER_OF_TIERS == 15`, we return `MAXIMUM_NUMBER_OF_TIERS` and the new number of tiers is going to be 15, even if the number of claims for this draw does not pass the threshold check.



Recommended Mitigation Steps

Make sure that we do the threshold check specifically, in the case where `_numTiers == 14` and `canaryClaimCount >= 1`.



Assessed type

Invalid Validation

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Simplified tier expansion logic.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/17>

Status: Mitigation error. Full details in report from [dirk_y](#) and in the [Mitigation Review](#) section below.



[M-17] VaultFactory allows deployment of vaults with non-authentic TwabController and PrizePool

Submitted by [Jeiwan](#), also found by devOcloo (1, 2), jaraxxus, OxSmartContract, oxchsystem, OxStalin (1, 2, 3), keccak123, dirk_y, btk, neumo, grearlake, 3docSec,

[rvierdiiev](#), and [ABAIKUNANBAEV](#)

A malicious vault can be deployed via the official `VaultFactory` contract. Users may lose funds while interacting with such vaults.



Proof of Concept

The [VaultFactory.deployVault](#) function allows deploying `Vault` contracts. The factory contract maintains a [mapping to verify that a vault has been deployed via the factory](#). This allows users to check the authenticity of a vault to ensure the that implementation of a vault is authentic (i.e. not altered/malicious).

However, the business logic of vaults is split into multiple contracts: [Vault](#), [TwabController](#), and [PrizePool](#). `TwabController` tracks the historical balances of users to determine their chances of winning prizes. `PrizePool` runs regular draws and distributes prizes among winners. Thus, it's critical that in every authentic `Vault` contract, the implementations of `TwabController` and `PrizePool` are also authentic. Otherwise, a malicious actor could deploy an authentic vault via the official `VaultFactory` and provide malicious `TwabController` and `PrizePool` contracts; which can incorrectly determine user balances, favors some specific addresses when determining winners, or steals the prize token. In the current implementation, users are be able to check the authenticity of a vault contract, but not the authenticity of the `TwabController` and the `PrizePool` contracts that a vault integrates with.



Recommended Mitigation Steps

Consider implementing `TwabController` and `PrizePool` factory contracts. In the contracts, consider tracking the addresses of the deployed `TwabController` and `PrizePool` contracts. In the `VaultFactory.deployVault` function, consider checking that the passed `_twabController` and `_prizePool` address were deployed via the respective factory contracts.

[Picodes \(judge\) commented:](#)



See [Issue #324](#).

[asselstine \(PoolTogether\) acknowledged and commented via duplicate issue #324:](#)

This the point of V5: we are replacing protocol gatekeeping with curation by front-ends.

The responsibility of curation will rest on front-ends, who will curate which vaults they show their users.



[M-18] The exchange rate change in the case of Lossy Strategy will cause the Vault to be under-collateralized for generic ERC4626 Yield Vaults

Submitted by [GalloDaSballo](#)

For Yield Vaults that use Lossy Strategies, the PT Vault will burn more Yield Vault Shares than intended when processing a withdrawal, socializing a loss at the advantage of early withdrawers.

`withdraw` calls `_convertToShares`:

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L517>

```
uint256 _shares = _convertToShares(_assets, Math.Rounding.Up
```

Which uses the `_currentExchangeRate()`. This in turn, computes the

`withdrawableAssets` by fetching `yieldVault.maxWithdraw` and then mapping the principal vs. the `totalSupply`.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1169C1-L1187C4>

```
uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address:  
  
if (_withdrawableAssets > _totalSupplyToAssets) {  
    _withdrawableAssets = _withdrawableAssets - (_withdrawable:  
}
```

```

if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
    return _withdrawableAssets.mulDiv(_assetUnit, _totalSupply);
}

```

This calculation is based on `maxWithdraw`, which is a `view` function.

Most implementations of `maxWithdraw` will simply map the available tokens to the shares owned by the owner. See OZ for example:

<https://github.com/OpenZeppelin/openzeppelin-contracts/blob/121be5dd09caa2f7ce731f0806b0e14761023bd6/contracts/token/ERC20/extensions/ERC4626.sol#L141-L143>

```

function maxWithdraw(address owner) public view virtual returns (uint256) {
    return _convertToAssets(balanceOf(owner), Math.Rounding.Down);
}

```

Or similarly, [Yearn V3](#).

Due to the implementation, losses can be applied during `YieldVault.withdraw`, causing the burning of more shares than intended.

Because the PT Vault computes the shares to burn before accounting for a potential loss:

- No loss will be accounted for in `maxWithdraw` (since it will use static value for assets in the vault and assets in the strategy).
- The loss will be locked during `_withdraw`, but it will not be checked. The specifics of the loss are that it will cause burning of more shares in order to receive the intended `assets`.
- This will cause the user to pay `shares` from PT Vault.
- But it will cause PT Vault to pay more `yieldVault Shares` than intended, effectively diluting everyone else and socializing the losses on the laggards.



Proof Of Concept

- Call `withdraw`.
- This will compute the shares to burn to the user via `_convertToShares`.
- PT Vault will call `YieldVault.withdraw` with the amount necessary to withdraw.
- The Vault will take a loss. The loss will cause more shares from PT Vault to be burned, socializing a loss to other depositors.



Mitigation

Ensures that the PPFS of the Yield Vault doesn't decrease, or add functions to handle lossy withdrawals.



Assessed type

ERC4626

[asselstine \(PoolTogether\) confirmed](#)

[PierrickGT \(PoolTogether\) commented:](#)

I don't entirely understand the issue here; to me it looks like this is the intended behavior.

Users knowingly deposit in a Vault relying on a `YieldVault` that employs Lossy Strategies to generate yield. Let's say for example, an option protocol, the position of the Vault loses value so all shares are backed by less underlying assets. When withdrawing, people will only be able to withdraw their shares of deposits, which indeed socializes the loss.

I think for this kind of YieldVaults, we will need to develop custom Vaults, because it's a pretty specific use case.

[PoolTogether mitigated:](#)

Redeemed underlying liquidity using shares, not assets directly. This socializes losses, if any.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/27>

Status: Not fully mitigated. Full details in report from [dirk_y](#) and in the [Mitigation Review](#) section below.

[PierrickGT \(PoolTogether\)](#) commented:

Working PR here: <https://github.com/GenerationSoftware/pt-v5-vault/pull/37>.



[M-19] Silent overflow could alter computation when calculating the `vaultPortion` in the `PrizePool` contract

Submitted by [OxStalin](#)

If an overflow happens, the computed value of the vault portion will be a totally different value than what should really be; which would end up causing the rewards to disperse and be lower for the draw that was used to compute the vault portion.

Note: Even though [the bot reported issues about downcasting variables](#), it didn't mention this specific unsafe casting; which if an overflow occurs, could cause a huge impact on the calculation of the vault portions.



Proof of Concept

When computing the `vaultPortion` to the `PrizePool` over a specific duration in draw, the values of the `vaultContributed` & `totalContributed` variables are computed on the `DrawAccumulatorLib` library, and [they are computed and returned as `uint256` values](#).

The issue is, that in the `PrizePool::_getVaultPortion()` the `vaultContributed` & `totalContributed` variables are unsafely cast from `uint256` to `int256`, which could lead to a silent overflow if any of the two original values don't fit on an `int256`.

As a result of a silent overflow, the computed value of the vault portion will be a totally different value than what should really be; which would end up causing the rewards to disperse and be lower for the draw that was used to compute the vault portion.



Recommended Mitigation Steps

Make sure to implement a safe cast that checks if overflows occur to prevent computing a totally different value than what it should really be. Use `0Z safeCast` library for this type of operation.



Assessed type

Under/Overflow

[asselstine \(PoolTogether\) confirmed and commented:](#)

The `DrawAccumulator` stores the available balance as `uint96` , but we could definitely tighten up the data types and make casting safe.

[PoolTogether mitigated:](#)

Added SafeCast.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/22>

Status: Mitigation confirmed. Full details in report from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-20] Improper handling of cases when withdrawable assets = 0

Submitted by [ktg](#)

- Improper handling of cases when withdrawable assets = 0.
- The vault will not function correctly.



Proof of Concept

The functions `_currentExchangeRate` and `_isVaultCollateralized` of Vault are implemented as follows:

```
function _currentExchangeRate() internal view returns (uint256)
    uint256 _totalSupplyAmount = _totalSupply();
    uint256 _totalSupplyToAssets = _convertToAssets(
        _totalSupplyAmount,
```

```

        _lastRecordedExchangeRate,
        Math.Rounding.Down
    );

    uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address:

    if (_withdrawableAssets > _totalSupplyToAssets) {
        _withdrawableAssets = _withdrawableAssets - (_withdrawableAssets - _totalSupplyToAssets);
    }

    if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
        return _withdrawableAssets.mulDiv(_assetUnit, _totalSupplyAmount);
    }

    return _assetUnit;
}

function _isVaultCollateralized() internal view returns (bool) {
    return _currentExchangeRate() >= _assetUnit;
}

```

The function calculates the exchange rate between the amount of assets withdrawable from the `YieldVault` and the amount of shares minted by this Vault.

However, if `_withdrawableAssets` is 0, then the function returns `_assetUnit` (which means 1-1 ratio). This means, that even when the `vault` has no withdrawable assets from `_yieldVault`, it's still considered collateralized.

To illustrate the oddity of this special case, consider when `_withdrawableAssets = 1` and `_totalSupplyAmount > 0`; in this scenario, `_currentExchangeRate` returns 0 and the vault is considered under-collateralized (since `1 < _assetUnit`). However, if `_withdrawableAssets = 0`, the vault is considered collateralized.

This case has profound impact since a lot of vault logic is based on the vault's collateralized status.

Below is a POC for the above example. For ease of testing, let's place these 2 test cases in file `vault/test/unit/Vault/Deposit.t.sol` under contract `VaultDepositTest`, then test them using commands:

- `forge test --match-path test/unit/Vault/Deposit.t.sol --match-test testOneWithdrawableAmount -vvvv`
- `forge test --match-path test/unit/Vault/Deposit.t.sol --match-test testZeroWithdrawableAmount -vvvv`

`testOneWithdrawableAmount` is to demonstrate when `_withdrawableAssets = 1` and the vault is considered not collateralized. `testZeroWithdrawableAmount` is to demonstrate when `_withdrawableAssets = 0` and the vault is considered collateralized.

```
function testZeroWithdrawableAmount() public {
    vm.startPrank(alice);
    uint256 _amount = 1000e18;
    underlyingAsset.mint(alice, _amount);
    underlyingAsset.approve(address(vault), type(uint256).max);

    vault.deposit(_amount, alice);

    // Now make the withdrawable asset = 0
    // Burn the balance of yieldVault
    uint256 yieldVaultAsset = underlyingAsset.balanceOf(address(yieldVault));
    underlyingAsset.burn(address(yieldVault), yieldVaultAsset);

    assertEq(underlyingAsset.balanceOf(address(yieldVault)), 0);

    // Although the vault has no asset withdrawable in yieldVault
    // the exchange rate is 10**18 = assetUnit and the vault is
    assertEq(yieldVault.maxWithdraw(address(vault)), 0);
    assertEq(vault.exchangeRate(), 10**18);
    assertEq(vault.isVaultCollateralized(), true);
}
```

```
function testOneWithdrawableAmount() public {
    vm.startPrank(alice);
    uint256 _amount = 1000e18;
    underlyingAsset.mint(alice, _amount);
    underlyingAsset.approve(address(vault), type(uint256).max);

    vault.deposit(_amount, alice);

    // Now make the withdrawable asset = 0
```



```

// Burn the balance of yieldVault
uint256 yieldVaultAsset = underlyingAsset.balanceOf(address(yieldVault));
underlyingAsset.burn(address(yieldVault), yieldVaultAsset - 1);

assertEq(underlyingAsset.balanceOf(address(yieldVault)), 1);

// vault only has 1 asset token withdrawable, and the exchangeRate is 1
// the vault is not collateralized
assertEq(yieldVault.maxWithdraw(address(vault)), 1);
assertEq(vault.exchangeRate(), 0);
assertEq(vault.isVaultCollateralized(), false);
}

```



Recommended Mitigation Steps

Since `_withdrawableAssets` is the dividend, there seems to be no harm in removing the check if `_withdrawableAssets == 0`. Therefore, I recommend removing it from the condition.



Assessed type

Invalid Validation

[asselstine \(PoolTogether\) confirmed](#)

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

There is definitely a bug here, but I don't see why it should be high severity. "This case has a profound impact since a lot of vault logic is based on the vault's collateralized status" is a bit light to pass the burden or proof. At first sight, as there are no funds in the vault anymore, there is no loss of funds. Then there is the case where assets are re-added to the vault for some reason; but this would be a Medium severity scenario, considering it's very unlikely and not described by the report.

[ktg9 \(warden\) commented:](#)

Hi @Picodes, thank you for your response!

Sorry I didn't make it clearer. As you can see in the report, the affected function is `_currentExchangeRate`, so not only is the `Vault` incorrectly specified as

collateralized, the `currentExchangeRate` also = 1 (`assetUnit`). These 2 functions affect many other functions, for example:

- Function `mintYieldFee` requires that vault is collateralized, so a user can mint the yield fee even when it's not.
- Function `liquidate` requires that vault is collateralized, so a user can liquidate asset even when vault's withdrawable asset = 0 .
- For `deposit` function, first `deposit` will calculate `maxDeposit` to limit the input tokens:

```
function maxDeposit(address) public view virtual override returns (uint)
return _isVaultCollateralized() ? `type(uint96).max` : 0;
}
```

If the Vault is under-collateralized, a user can't deposit since `maxDeposit` will return 0; however, a user can still deposit in this case:

- User calls `exchangeRate` and receives 1, this can lead them to bad decisions.

To conclude, I agree that this will not lead to direct loss of funds for the vault, but will greatly impact the user, making them send tokens to the vault (through liquidate or deposit) when it's not collateralized and will potentially suffer economically. I think it should be marked as High severity.

[Picodes \(judge\) commented:](#)

@ktg9 - I still think that the original reports perfectly fit with the definition of Med severity:

“leak value with a hypothetical attack path with stated assumptions, but external requirements”.

The described scenarios require that we are in a situation with `withdrawable assets` = 0 and that some users behave incorrectly. Also note that I can't consider scenarios that are not described in the original report.

[PierrickGT \(PoolTogether\) commented:](#)

This issue has been fixed and the exchange rate between the Vault and YieldVault is now 1 to 1. We've replaced the exchange rate function by a collateral one, which simplifies the conversions from shares to assets and assets to shares.

If the Vault is collateralized, meaning the amount of withdrawable assets from the YieldVault is greater than the amount of underlying assets supplied to the YieldVault, then users can only withdraw the amount they deposited. Otherwise, any remaining collateral within the YieldVault is available and distributed proportionally among depositors.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/8985bead1be85ae6822cd329933f5a53de05c237/src/Vault.sol#L1208>

PoolTogether mitigated:

Vault shares are now 1:1 with an underlying asset, and the exchange logic has been simplified.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/18>

Status: Mitigation confirmed. Full details in report from [dirk_y](#), [rvierdiiev](#) and [OxStalin](#).



[M-21] Vault contribution calculations wrongly include the current round when claiming prizes

Submitted by [dirk_y](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L366> <https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L421> <https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L903-L908>



Impact

When claiming prizes, the higher the contribution percentage of the given vault, the higher the odds of a user contributing to that vault receiving a prize. Because the vault contributions are wrongly calculated, a vault gets credit for distributions that haven't actually been made yet. A vault can swing the odds in its favour by contributing prize tokens to the prize pool in a recent round before claiming prizes (based on a normal value of alpha). This is particularly important when considering prize tiers that occur frequently (the highest tiers).



Proof of Concept

A draw is closed by the draw manager by calling `closeDraw`. During this process, new liquidity is distributed among the tiers to be available for prize claims:

```
_nextDraw(_nextNumberOfTiers, uint96(_contributionsForDraw(lastC.
```

In the above line, the contributions are calculated based on `lastClosedDrawId + 1` because `lastClosedDrawId` has not yet been incremented. It is incremented at the end of the `_nextDraw` method.

A prize is claimed when a vault calls (via a claimer) `claimPrize`. In this process, the contribution percentage of the vault is calculated because it affects the winning region (i.e. odds) for a user:

```
(SD59x18 _vaultPortion, SD59x18 _tierOdds, uint16 _drawDuration)
    msg.sender,
    _tier,
    numberOfTiers,
    lastClosedDrawId
);
```

As can be seen above, the vault portion is calculated based on `lastClosedDrawId`, which is the value which was incremented when closing the draw previously. This method makes the following underlying call:

```
vaultPortion = _getVaultPortion(
    _vault,
```

```

        uint16(drawDuration > _lastClosedDrawId ? 0 : _lastClosedD
        _lastClosedDrawId + 1,
        smoothing.intoSD59x18())
    );

```

This calculates the value dispersed between a start and end draw INCLUSIVE. Therefore, this is actually calculating the vault contributions based on the current in-progress round as well, which hasn't yet been dispersed to the tiers. In fact, it also wrongly increments the start round ID.



Recommended Mitigation Steps

The `_computeVaultTierDetails` method should not increment the start and end draw IDs:

```

diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..34548c9 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -902,8 +902,8 @@ contract PrizePool is TieredLiquidityDistrib
     drawDuration = uint16(TierCalculationLib.estimatePrizeFreque
     vaultPortion = _getVaultPortion(
         _vault,
-        uint16(drawDuration > _lastClosedDrawId ? 0 : _lastClosedD
-        _lastClosedDrawId + 1,
+        uint16(drawDuration > _lastClosedDrawId ? 0 : _lastClosedD
+        _lastClosedDrawId,
         smoothing.intoSD59x18())
     );
}

```



Assessed type

Math

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Draw query ends on closed draw.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/10>

Status: Mitigation confirmed. Full details in report from [dirk_y](#), [rvierdiiev](#) and [OxStalin](#).



[M-22] Loss of precision leads to under-collateralized

Submitted by [bin2chen](#)

Since `_yieldVault` mostly calculates shares and uses round down when depositing, there is often a `1 wei` loss of precision, which can cause the vault to go into under-collateralized mode by mistake.



Proof of Concept

When a user deposits an asset, we update `_lastRecordedExchangeRate` ; the calculation is done by method `_currentExchangeRate()` .

The code is as follows:

```
function _currentExchangeRate() internal view returns (uint256
    uint256 _totalSupplyAmount = _totalSupply();
    uint256 _totalSupplyToAssets = _convertToAssets(
        _totalSupplyAmount,
        _lastRecordedExchangeRate,
        Math.Rounding.Down
    );

    @> uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address:

    if (_withdrawableAssets > _totalSupplyToAssets) {
        _withdrawableAssets = _withdrawableAssets - (_withdrawable:
    }

    if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
        return _withdrawableAssets.mulDiv(_assetUnit, _totalSupply:
    }

    return _assetUnit;
}
```

This method takes `_yieldVault.maxWithdraw(address(this))` as the maximum value to calculate the current exchange rate. If the exchange rate is lower than `_assetUnit`, then it goes into an under-collateralized mode, where it can only be withdrawn, not deposited. So if `_yieldVault` is losing money, it goes into under-collateralized.

But there is one missing consideration here:

As long as `_yieldVault` is not exclusive, there will be precision loss issues. After `_yieldVault.deposit()`, `maxWithdraw()` will lose precision because most vaults will “round down” the shares calculations. For example, depositing `10000000000`; however, it can only withdraw `9999999999`.

This leads to the problem that when the first deposit is made, it is likely to go into an under-collateralized mode immediately due to the `1 wei` loss.

The following code demonstrates that when a non-exclusive `_yieldVault`, once Alice first deposits normally, it immediately enters an under-collateralized mode.

Add to `Deposit.t.sol`:

```
function testLossPrecision() external {
    vm.startPrank(alice);

    //0.Constructing a yieldVault that is already profitable
    uint256 _amount = 333e18;
    underlyingAsset.mint(alice, _amount);
    underlyingAsset.approve(address(yieldVault), type(uint256).max);
    yieldVault.deposit(_amount, alice);
    //profitable 0.1e18
    underlyingAsset.mint(address(yieldVault), 0.1e18);

    //1.alice deposit
    _amount = 100e18;
    underlyingAsset.mint(alice, _amount);
    underlyingAsset.approve(address(vault), type(uint256).max);
    console2.log("deposit:", _amount);
    vault.deposit(_amount, alice);
    console2.log("maxWithdraw:", yieldVault.maxWithdraw(address(vault)));
    console2.log("loss :", _amount - yieldVault.maxWithdraw(address(vault)));
}
```

```

        console2.log("isVaultCollateralized:", vault.isVaultCollatera
    return;
}

```

```
$ forge test --match-test testLossPrecision -vvv
```

```
[PASS] testLossPrecision() (gas: 402881)
```

```
Logs:
```

```

deposit: 1000000000000000000000
maxWithdraw: 99999999999999999999
loss : 1
isVaultCollateralized: false

```

This small loss of precision should not be treated as a loss. We can avoid it by adding `1wei` when calculating the exchange rate.



Recommended Mitigation Steps

Have `_yieldVault.maxWithdraw() + 1` to avoid loss of precision into an under-collateralized mode.

```

function _currentExchangeRate() internal view returns (uint256
    uint256 _totalSupplyAmount = _totalSupply();
    uint256 _totalSupplyToAssets = _convertToAssets(
        _totalSupplyAmount,
        _lastRecordedExchangeRate,
        Math.Rounding.Down
    );

-   uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address
+   uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address

    if (_withdrawableAssets > _totalSupplyToAssets) {
        _withdrawableAssets = _withdrawableAssets - (_withdrawable

    if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
        return _withdrawableAssets.mulDiv(_assetUnit, _totalSupply

    return _assetUnit;

```


}



Assessed type

Decimal

[asselstine \(PoolTogether\) confirmed](#)

[PierrickGT \(PoolTogether\) commented:](#)

Nice catch. Since we don't have any control over the YieldVault exchange rate, if it gets manipulated at any time, we should revert any deposits where the amount of withdrawable assets from the YieldVault is lower than the expected amount.

[PoolTogether mitigated:](#)

Improved vault share / asset calculation.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/18>

Status: Not fully mitigated. Full details in report from [dirk_y](#) and [OxStalin](#), and in the [Mitigation Review](#) section below.



[M-23] Vault does not conform to ERC4626

Submitted by [degensec](#), also found by [pontifex](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L375-L377> <https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L383-L385>



Impact

Vault does not conform to ERC4626 which may break external integrations.



Proof of Concept

The [ERC4626 specification](#) states that `maxDeposit` *MUST* return the maximum amount of assets `deposit` would allow to be deposited for receiver and not cause a revert, which *MUST NOT* be higher than the actual maximum that would be accepted.

Similarly, `maxMint` *MUST* return the maximum amount of shares `mint` would allow to be deposited to receiver and not cause a revert, which *MUST NOT* be higher than the actual maximum that would be accepted.

The PoolTogether V5 `Vault` connects to an external ERC4626-compliant Vault (`_yieldVault`) and deposits incoming assets in it. This means, that `maxDeposit` and `maxMint` of the PoolTogether Vault must be constrained by the `maxDeposit` and `maxMint` of the external Vault.



Tools Used

[ERC-4626: Tokenized Vaults](#)



Recommended Mitigation Steps

Replace the implementation of `maxDeposit`:

```
function maxDeposit(address) public view virtual override retur
    return _isVaultCollateralized() ? `type(uint96).max` : 0;
}
```

with:

```
function maxDeposit(address receiver) public view virtual over
    if (!_isVaultCollateralized()) return 0;

    uint256 yvMaxDeposit = _yieldVault.maxDeposit(receiver);
    return yvMaxDeposit < `type(uint96).max` ? yvMaxDeposit : `t
}
```

Analogously, change the implementation of `maxMint` from:

```
function maxMint(address) public view virtual override returns
    return _isVaultCollateralized() ? `type(uint96).max` : 0;
}
```

to:

```
function maxMint(address receiver) public view virtual override
    if(!_isVaultCollateralized()) return 0;

    uint256 yvMaxMint = _yieldVault.maxDeposit(receiver);
    return yvMaxMint < `type(uint96).max` ? yvMaxMint : `type(uii
}
```



Assessed type

ERC4626

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Fixed compliance with 4626.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/11>

Status: Mitigation confirmed. Full details in report from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-24] `claimer.claimPrizes` can be front-runned in order to make losses for the claim bot

Submitted by [rvierdiiev](#), also found by [MiniGlome](#)



Proof of Concept

All prizes in the system should be claimed during the draw period. Anyone can call the `claimPrizes` function for any winner and prize. But the idea of protocol is to

incentivize claim bots to do all claims instead of users. These should benefit from batching claims together in 1 call.

Also the `Claimer` contract is using `vrgda` which can increase/decrease fee for the claimer, depending of amount of already claimed prizes. That actually means, that in case not enough prizes are claimed in some point of time, then the fee will be increased to incentivize more bots.

So as bots will try to batch a lot of prize claims together, that means that they will spent a lot on gas. A malicious actor can see which prizes the bot is going to claim and front-run them with the last prize in the list. As a result, [claiming will fail](#) and the bot will face losses.

Another reason why a malicious actor can do that is to make `vrgda` to provide bigger fees. They can create a bot that will block claims of other bots with a big amount of claims in the batch, in order to wait and get better prices and claim those prize by themself and receive fees.



Tools Used

VsCode



Recommended Mitigation Steps

In the case of if a price is already claimed, then you can return early from the `PrizePool.claimPrize` function and emit some event. In this case, bots will not receive fee for the already claimed prize and their tx will continue with other prizes in the batch.



Assessed type

Error

[asselstine \(PoolTogether\) confirmed](#)

[Picodes \(judge\) commented:](#)

I'll keep medium severity, as the sponsor confirmed with their label that it was valuable to them. I think we could also justify this being low. As in such cases, it is up to the bot to use a private rpc or some other solution to avoid being front-ran.

PoolTogether mitigated:

Added `minVrgdaFee` and allowed silent failure of claims.

PRs: <https://github.com/GenerationSoftware/pt-v5-claimer/pull/8>

<https://github.com/GenerationSoftware/pt-v5-vault/pull/22>

<https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/23>

Status: Mitigation confirmed. Full details in report from [rvierdiiev](#), [dirk_y](#) and [OxStalin](#).



[M-25] `depositWithPermit` and `mintWithPermit` are allowed to be called by the permit creator only

Submitted by [rvierdiiev](#), also found by [xuwinnie](#)

Functions `depositWithPermit` and `mintWithPermit` are allowed to be called by the permit creator only. Not any other contracts will be able to execute these function on behalf of signer.



Proof of Concept

The `depositWithPermit` function is allowed to provide a signed permit in order to receive, approve and deposit funds into the vault.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L427-L437>

```
function depositWithPermit(
    uint256 _assets,
    address _receiver,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) external returns (uint256) {
    _permit(IERC20Permit(asset()), msg.sender, address(this), _a:
    return deposit(_assets, _receiver);
```

```
}
```

This function calls `_permit` and pass `msg.sender` as `_owner` to that function.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1093-L1104>

```
function _permit(
    IERC20Permit _asset,
    address _owner,
    address _spender,
    uint256 _assets,
    uint256 _deadline,
    uint8 _v,
    bytes32 _r,
    bytes32 _s
) internal {
    _asset.permit(_owner, _spender, _assets, _deadline, _v, _r, _s)
}
```

This means, that the signer can be only the same person that called `depositWithPermit` function.

However, the purpose of the permit is to allow someone to sign the approve signature, so that this signature can be used by another contract to call some function on behalf of signer.

In this case, anyone should be able to sign permit for the vault, and vault should check that `_receiver` is same who signed permit.



Tools Used

VsCode



Recommended Mitigation Steps

Use `_receiver` instead of `msg.sender`.

```
function depositWithPermit(  
    uint256 _assets,  
    address _receiver,  
    uint256 _deadline,  
    uint8 _v,  
    bytes32 _r,  
    bytes32 _s  
) external returns (uint256) {  
    _permit(IERC20Permit(asset()), _receiver, address(this), _as:  
    return deposit(_assets, _receiver);  
}
```



Assessed type

Error

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Fixed permit implementations.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/20>

Status: Mitigation confirmed. Full details in report from [dirk_y](#), [rvierdiev](#) and [OxStalin](#).



[M-26] Transfer of Vault tokens can cause accounting errors in other contracts

Submitted by [shaka](#)



Lines of code

[https://github.com/GenerationSoftware/pt-v5-](https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1155)

[vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1155](https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1155)

[https://github.com/GenerationSoftware/pt-v5-twab-](https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/0145eeac23301ee5338c659422dd6d69234f5d50/src/libraries/TwabLib.sol#L48)

[controller/blob/0145eeac23301ee5338c659422dd6d69234f5d50/src/libraries/TwabLib.sol#L48](https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/0145eeac23301ee5338c659422dd6d69234f5d50/src/libraries/TwabLib.sol#L48)



Vulnerability details

`Vault::_transfer` overrides the `ERC20::_transfer` function so the transfer is performed using the `TwabController` contract. In the implementation, `_shares` is downcasted to `uint96` and this can result in an underflow silently.

There are nat spec comments indicating that the balance in `TwabController` is stored in `uint96`, which would mean that there is no real risk of underflow. However, the `TwabController` contract uses `uint112` to store the balance, so the risk of underflow is real.

File: `TwabLib.sol`

```
47 struct AccountDetails {
48     uint112 balance; // <== not uint96
49     uint112 delegateBalance;
50     uint16 nextObservationIndex;
51     uint16 cardinality;
52 }
```



Impact

Integrations of other contracts or protocols with the `Vault` contract may result in accounting errors, due to the amount of shares transferred being less than expected due to silent underflow.



Proof of Concept

Foundry test:

```
mapping(address => uint256) userToBalance;

function testTransferUnderflow() external {
    vm.startPrank(alice);

    uint256 _amount = type(uint112).max;
    underlyingAsset.mint(alice, _amount);
    underlyingAsset.approve(address(vault), type(uint256).max);

    vault.deposit(`type(uint96).max`, alice);
    vault.deposit(`type(uint96).max`, alice);
```



```

vm.stopPrank();
assertEq(vault.balanceOf(alice), uint256(`type(uint96).max`))

// Alice has now a balance of `type(uint96).max` * 2
// Let's imagine some integration of another contract with the vault
// that performs the following operations:

// 1. Alice approves all the Vault shares to the contract
vm.prank(alice);
vault.approve(address(this), type(uint256).max);

// 2. The contract transfers all the Vault shares from Alice
// in in a mapping
uint256 aliceBalance = vault.balanceOf(alice);
vault.transferFrom(alice, address(this), aliceBalance);
userToBalance[alice] = aliceBalance;

// 3. Only `type(uint96).max` shares are transferred, but the contract
// is registered as transfer underflows silently. So alice can't get her
// remaining balance from the vault.
vm.prank(alice);
vault.transfer(bob, `type(uint96).max`);
}

```



Recommended Mitigation Steps

Use OpenZeppelin's `SafeCast` library to convert `uint256` to `uint96`.



Assessed type

Under/Overflow

[asselstine \(PoolTogether\) confirmed](#)

[PierrickGT \(PoolTogether\) commented:](#)

Regarding the `TwabController`, we need to figure out if we want to keep `uint112` for balances or if we want to use `uint96`.

[PoolTogether mitigated:](#)

Added SafeCast.

PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/9>

Status: Mitigation confirmed. Full details in report from [OxStalin](#), [dirk_y](#) and [rvierdiiev](#).



[M-27] Inconsistent behavior for canary claims in the `claimer`

Submitted by [volodya](#)



Proof of Concept

Whenever a `claimer` claims prizes they compute fees per `claim` without considering canary claims.

```
function claimPrizes(
    Vault vault,
    uint8 tier,
    address[] calldata winners,
    uint32[][] calldata prizeIndices,
    address _feeRecipient
) external returns (uint256 totalFees) {
    uint256 claimCount;
    for (uint i = 0; i < winners.length; i++) {
        claimCount += prizeIndices[i].length;
    }

    uint96 feePerClaim = uint96(
        _computeFeePerClaim(
            _computeMaxFee(tier, prizePool.numberOfTiers()),
            claimCount,
            prizePool.claimCount()
        )
    );

    vault.claimPrizes(tier, winners, prizeIndices, feePerClaim, _

    return feePerClaim * claimCount;
}
```

[src/Claimer.sol#L76](#)



Recommended Mitigation Steps

Consider canary as a `claim`.

```

uint96 feePerClaim = uint96(
    _computeFeePerClaim(
        _computeMaxFee(tier, prizePool.numberOfTiers()),
        claimCount,
-       prizePool.claimCount()
+       prizePool.claimCount() + prizePool.canaryClaimCount()
    )
);

```



Assessed type

Error

[asselstine \(PoolTogether\) confirmed](#)

[PoolTogether mitigated:](#)

Unified standard and canary tier.

PR: <https://github.com/GenerationSoftware/pt-v5-prize-pool/pull/17>

Status: Mitigation confirmed. Full details in report from [rvierdiiev](#), [dirk_y](#) and [OxStalin](#).



Low Risk and Non-Critical Issues

For this audit, 37 reports were submitted by wardens detailing low risk and non-critical issues. The [report highlighted below](#) by [bin2chen](#) received the top score from the judge.

The following wardens also submitted reports: [kutugu](#), [markus_ether](#), [catellatech](#), [alymurtazamemon](#), [yixxas](#), [Jeiwan](#), [rvierdiiev](#), [volodya](#), [dacian](#), [naman1778](#), [lanrebayode77](#), [alexzoid](#), [GREY-HAWK-REACH](#), [Kaysoft](#), [Oxbepresent](#), [banpaleo5](#), [Ox1lsingh99](#), [MohammedRizwan](#), [nadin](#), [Inspektor](#), [Bauchibred](#), [squeaky_cactus](#), [erebus](#), [Vagner](#), [Rolezn](#), [DadeKuma](#), [GalloDaSballo](#), [keccak123](#), [joaovwfreire](#),

[grearlake](#), [ayden](#), [ArmedGoose](#), [fatherOfBlocks](#), [eyexploit](#), [ABAIKUNANBAEV](#), and [OxWaitress](#).

🔗

[L-01] vault._mint() forcing conversion to uint96 may result in missing values

The current protocol Vault.sol has uint256 for shares, but uint96 is used in TwabController.sol. So in mint/burn/transfer, it is forced to convert to uint96. This causes the value to exceed type(uint96).max if asset.decimals()>18, resulting in a loss of value.

Suggesting to add a method like SafeToUint96() to revert if it is larger than uint96:

```
function _burn(address _owner, uint256 _shares) internal virtual {
-   _twabController.burn(_owner, uint96(_shares));
+   _twabController.burn(_owner, _shares.SafeToUint96());
    _updateExchangeRate();

    emit Transfer(_owner, address(0), _shares);
}
```

Note: mint() and transfer() have the same problem.

🔗

[L-02] targetOf() incorrectly uses _token != _liquidationPair.tokenIn()

Currently, targetOf() uses _liquidationPair.tokenIn() but the actual comparison in liquidate() is _tokenIn != address(_prizePool.prizeToken()). So it's _prizePool.prizeToken() that should be used instead.

```
function targetOf(address _token) external view returns (address) {
-   if (_token != _liquidationPair.tokenIn()) revert TargetTokenInvalid();
+   if (_tokenIn != address(_prizePool.prizeToken())) revert TargetTokenInvalid();
    return address(_prizePool);
}
```



[L-03] setLiquidationPair() may not be able to set the old liquidationPair

Use the `safeApprove()` method in `setLiquidationPair()` to set the allowance.

The `safeApprove()` method requires the current allowance to be 0 for the setting to succeed. If the `liquidationPair_` has been used before, this is the second time it is used and there are residual allowances, it may not be possible to set it.

I suggest to clear the allowance to 0 before setting.

```
function setLiquidationPair(
    LiquidationPair liquidationPair_
) external onlyOwner returns (address) {
    if (address(liquidationPair_) == address(0)) revert LPZeroAd

    IERC20 _asset = IERC20(asset());
    address _previousLiquidationPair = address(_liquidationPair)

    if (_previousLiquidationPair != address(0)) {
        _asset.safeApprove(_previousLiquidationPair, 0);
    }

+   _asset.safeApprove(address(liquidationPair_), 0);
    _asset.safeApprove(address(liquidationPair_), type(uint256).1

    _liquidationPair = liquidationPair_;

    emit LiquidationPairSet(liquidationPair_);
    return address(liquidationPair_);
}
```



[L-04] in `_withdraw()` , by calling `_burn()` and then executing `_yieldVault.withdraw()` , it may lead `_updateExchangeRate()` inaccuracy

In `withdraw()` , execute `_burn()` first in `_yieldVault.withdraw()` .

Code as follows:

```

function _withdraw(
    address _caller,
    address _receiver,
    address _owner,
    uint256 _assets,
    uint256 _shares
) internal virtual override {
....

@> _burn(_owner, _shares);

@> _yieldVault.withdraw(_assets, address(this), address(this));
    SafeERC20.safeTransfer(IERC20(asset()), _receiver, _assets);

    emit Withdraw(_caller, _receiver, _owner, _assets, _shares);
}

```

In `_burn()` , the method will execute `_updateExchangeRate()` :

```

function _burn(address _owner, uint256 _shares) internal virtual
@> _twabController.burn(_owner, uint96(_shares));
@> _updateExchangeRate();

    emit Transfer(_owner, address(0), _shares);
}

```

Then, `_currentExchangeRate()` will read `_yieldVault.maxWithdraw()` :

```

function _currentExchangeRate() internal view returns (uint256)
    uint256 _totalSupplyAmount = _totalSupply();
    uint256 _totalSupplyToAssets = _convertToAssets(
        _totalSupplyAmount,
        _lastRecordedExchangeRate,
        Math.Rounding.Down
    );

@> uint256 _withdrawableAssets = _yieldVault.maxWithdraw(address(
    0));

    if (_withdrawableAssets > _totalSupplyToAssets) {
        _withdrawableAssets = _totalSupplyToAssets;
    }
}

```

```

        if (_totalSupplyAmount != 0 && _withdrawableAssets != 0) {
            return _withdrawableAssets.mulDiv(_assetUnit, _totalSupply);
        }

        return _assetUnit;
    }
}

```

This results in a wrong order:

1. `vault.burn()` **reduces** shares.
2. **Read** shares **and** total assets **calculate** the exchange rate.
3. `_yieldVault.withdraw()` **reduces** total assets.

The result is that the second step of reading total assets is not correct and the third step should be put before the second step.

Suggestion:

```

function _withdraw(
    address _caller,
    address _receiver,
    address _owner,
    uint256 _assets,
    uint256 _shares
) internal virtual override {
    ....

+   _yieldVault.withdraw(_assets, address(this), address(this));

    _burn(_owner, _shares);

-   _yieldVault.withdraw(_assets, address(this), address(this));
    SafeERC20.safeTransfer(IERC20(asset()), _receiver, _assets);

    emit Withdraw(_caller, _receiver, _owner, _assets, _shares);
}

```

Currently `liquidate()` requires `_amountOut >= _vaultAssets` to execute `_yieldVault.deposit()` if there is a residual balance in the contract. This presents some risk for a sandwich attack.

Example:

1. A malicious user monitors `memory pool`, `_liquidationPair` transfer assets to the contract, and executes `liquidate()`.
2. Front-runs the transaction in step 1, transferring an `asset` larger than `_amountOut`.
3. Waits until the first transaction, because `_amountOut >= _vaultAssets`, will not execute `_yieldVault.deposit()`, the funds are still in the contract.
4. A malicious user uses the balance left in the contract by executing `vault.deposit()`.

Suggesting to remove the `_amountOut >= _vaultAssets` restriction.

```
function liquidate(
    address _account,
    address _tokenIn,
    uint256 _amountIn,
    address _tokenOut,
    uint256 _amountOut
) public virtual override returns (bool) {
    ...

    uint256 _vaultAssets = IERC20(asset()).balanceOf(address(this));

-   if (_vaultAssets != 0 && _amountOut >= _vaultAssets) {
+   if (_vaultAssets != 0) {
        _yieldVault.deposit(_vaultAssets, address(this));
    }

    _mint(_account, _amountOut);

    return true;
}
```


[PierrickGT \(PoolTogether\) commented:](#)

L-01: has been fixed.

L-02 and 03: have been removed.

L-04 and 05: has been fixed.



Gas Optimizations

For this audit, 19 reports were submitted by wardens detailing gas optimizations. The [report highlighted below](#) by [c3phas](#) received the top score from the judge.

The following wardens also submitted reports: [SAQ](#), [Udsen](#), [hunter_w3b](#), [alymurtazamemon](#), [petrichor](#), [OxAnah](#), [naman1778](#), [ybansal2403](#), [SM3_SS](#), [Ox1Isingh99](#), [Raihan](#), [ReyAdmirado](#), [Rolezn](#), [Rageur](#), [SAAJ](#), [Oxn006e7](#), [LeoS](#), and [koxuan](#).



Auditor's Disclaimer

While we try our best to maintain readability in the provided code snippets, some functions have been truncated to highlight the affected portions.

It's important to note that during the implementation of these suggested changes, developers must exercise caution to avoid introducing vulnerabilities. Although the optimizations have been tested prior to these recommendations, it is the responsibility of the developers to conduct thorough testing again.

Code reviews and additional testing are strongly advised to minimize any potential risks associated with the refactoring process.



Note on Gas estimates

I've tried to give the exact amount of gas being saved from running the included tests. Whenever the function is within the test coverage, the average gas before and after will be included, and often a diff of the code will also accompany this.

Some functions are not covered by the test cases or are internal/private functions. In this case, the gas can be estimated by looking at the opcodes involved or simply benchmark any function that calls our function of interest.



Tightly pack storage variables/optimize the order of variable declaration (saves 2.1K gas)

The EVM works with 32 byte words. Variables less than 32 bytes can be declared next to each other in storage and this will pack the values together into a single 32 byte storage slot (if the values combined are ≤ 32 bytes).

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L215-L224>



We can pack `_liquidationPair` with `_yieldFeePercentage` by reducing size of `_yieldFeePercentage` to `uint64`

```
File: /src/Vault.sol
215:  LiquidationPair private _liquidationPair;

218:  uint256 private _assetUnit;

221:  uint256 private _lastRecordedExchangeRate;

224:  uint256 private _yieldFeePercentage;
```

When setting the value of `_yieldFeePercentage` we have some bounds that constrain the max value. The assigned value can be:

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L1218-L1223>

```
1218:  function _setYieldFeePercentage(uint256 yieldFeePercentage)
1219:      if (yieldFeePercentage_ > FEE_PRECISION) {
1220:          revert YieldFeePercentageGTPrecision(yieldFeePercentage);
1221:      }
1222:      _yieldFeePercentage = yieldFeePercentage_;
1223:  }
```

From the above setter, it is evident the value of `_yieldFeePercentage` cannot be greater than `FEE_PRECISION`, lest we revert. `FEE_PRECISION` is a constant defined as shown below:

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L232-L233>

```
232:  /// @notice Fee precision denominated in 9 decimal places :
233:  uint256 private constant FEE_PRECISION = 1e9;
```

This means, the value assigned to `_yieldFeePercentage` cannot be greater than `1e9`. Using `uint64`, we get a max of `18446744073709551615` which should be more than enough.

As such, we can refactor and pack the variables as shown below:

```
diff --git a/src/Vault.sol b/src/Vault.sol
index e92ecaf..4c21912 100644
--- a/src/Vault.sol
+++ b/src/Vault.sol
@@ -213,6 +213,9 @@ contract Vault is ERC4626, ERC20Permit, ILiquidityProvider {

    /// @notice Address of the LiquidationPair used to liquidate :
    LiquidationPair private _liquidationPair;
+
+    /// @notice Yield fee percentage represented in integer format
+    uint64 private _yieldFeePercentage;

    /// @notice Underlying asset unit (i.e. 10 ** 18 for DAI).
    uint256 private _assetUnit;
@@ -220,9 +223,6 @@ contract Vault is ERC4626, ERC20Permit, ILiquidityProvider {
    /// @notice Most recent exchange rate recorded when burning of
    uint256 private _lastRecordedExchangeRate;

-    /// @notice Yield fee percentage represented in integer format
-    uint256 private _yieldFeePercentage;
-
    /// @notice Address of the yield fee recipient that receives
    address private _yieldFeeRecipient;
```

Note: We would need to refactor all places this value is used to change the type to `uint64`.



Expensive operation inside a for loop

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/abstract/TieredLiquidityDistributor.sol#L361-L369>

	Min	Average	Median	Max	
Before	1901	59662	67737	79917	
After	1901	59649	67723	79903	

```
File: /src/abstract/TieredLiquidityDistributor.sol
361:     for (uint8 i = start; i < end; i++) {
362:         _tiers[i] = Tier({
363:             drawId: closedDrawId,
364:             prizeTokenPerShare: prizeTokenPerShare,
365:             prizeSize: uint96(
366:                 _computePrizeSize(i, _nextNumberOfTiers, _prizeTokenPerShare)
367:             )
368:         });
369:     }
```

```
diff --git a/src/abstract/TieredLiquidityDistributor.sol b/src/abstract/TieredLiquidityDistributor.sol
index 8238103..ba0587b 100644
--- a/src/abstract/TieredLiquidityDistributor.sol
+++ b/src/abstract/TieredLiquidityDistributor.sol
@@ -358,10 +358,11 @@ contract TieredLiquidityDistributor {
     start = _nextNumberOfTiers - 1;
     end = _nextNumberOfTiers;
 }
+
+ UD34x4 prizeTokenPerShare_ = prizeTokenPerShare ;
for (uint8 i = start; i < end; i++) {
    _tiers[i] = Tier({
        drawId: closedDrawId,
-       prizeTokenPerShare: prizeTokenPerShare,
+       prizeTokenPerShare: prizeTokenPerShare_,
        prizeSize: uint96(
            _computePrizeSize(i, _nextNumberOfTiers, _prizeTokenPerShare_)
```



Use calldata instead of memory for function parameters

If a reference type function parameter is read-only, it is cheaper in gas to use calldata instead of memory. Calldata is a non-modifiable, non-persistent area where function arguments are stored, and behaves mostly like memory.

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/VaultFactory.sol#L55-L66>



Use calldata on _name and _symbol (saves 524 gas on average)

	Min	Average	Median	Max	
Before	3737309	3737309	3737309	3737309	
After	3736785	3736785	3736785	3736785	

```
File: /src/VaultFactory.sol
55:  function deployVault(
56:      IERC20 _asset,
57:      string memory _name,
58:      string memory _symbol,
59:      TwabController _twabController,
60:      IERC4626 _yieldVault,
61:      PrizePool _prizePool,
62:      address _claimer,
63:      address _yieldFeeRecipient,
64:      uint256 _yieldFeePercentage,
65:      address _owner
66:  ) external returns (address) {
```

```
diff --git a/src/VaultFactory.sol b/src/VaultFactory.sol
index 559fd90..2b85cba 100644
--- a/src/VaultFactory.sol
+++ b/src/VaultFactory.sol
@@ -54,8 +54,8 @@ contract VaultFactory {
    */
    function deployVault(
```

```

IERC20 _asset,
-   string memory _name,
-   string memory _symbol,
+   string calldata _name,
+   string calldata _symbol,
TwabController _twabController,
IERC4626 _yieldVault,
PrizePool _prizePool,

```



Repeatedly doing the same operation (addition) involving a state variable should be avoided

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L316-L328>

Note this was not found by the bot under caching state variables. Gas saved on average: 548

	Min	Average	Median	Max	
Before	17709	181028	190311	226423	
After	17709	180480	189753	225865	

```

File: /src/PrizePool.sol
316:     DrawAccumulatorLib.add(
317:         vaultAccumulator[_prizeVault],
318:         _amount,
319:         lastClosedDrawId + 1,
320:         smoothing.intoSD59x18()
321:     );
322:     DrawAccumulatorLib.add(
323:         totalAccumulator,
324:         _amount,
325:         lastClosedDrawId + 1,
326:         smoothing.intoSD59x18()
327:     );
328:     emit ContributePrizeTokens(_prizeVault, lastClosedDrawId

```

In the function above, the operation `lastClosedDrawId + 1` is being done repeatedly. As `lastClosedDrawId` is a state variable, the amount of gas becomes too much. We should do the operation and cache the result in a `memory` variable.

```
diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..6a850ae 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -313,19 +313,20 @@ contract PrizePool is TieredLiquidityDistr
    if (_deltaBalance < _amount) {
        revert ContributionGTDeltaBalance(_amount, _deltaBalance)
    }
+   uint16 _lastClosedDrawIdPlusOne = lastClosedDrawId + 1;
    DrawAccumulatorLib.add(
        vaultAccumulator[_prizeVault],
        _amount,
-       lastClosedDrawId + 1,
+       _lastClosedDrawIdPlusOne,
        smoothing.intoSD59x18()
    );
    DrawAccumulatorLib.add(
        totalAccumulator,
        _amount,
-       lastClosedDrawId + 1,
+       _lastClosedDrawIdPlusOne,
        smoothing.intoSD59x18()
    );
-   emit ContributePrizeTokens(_prizeVault, lastClosedDrawId + 1);
+   emit ContributePrizeTokens(_prizeVault, _lastClosedDrawIdPlusOne);
    return _deltaBalance;
}
```



Use the existing memory value to do the subtraction

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L483-L493>

Average Gas Saved: 77

	Min	Average	Median	Max	
Before	2750	18704	23717	29646	
After	2750	18627	23615	29518	

```
File: /src/PrizePool.sol
483:  function withdrawClaimRewards(address _to, uint256 _amount
484:      uint256 _available = claimerRewards[msg.sender];

486:      if (_amount > _available) {
487:          revert InsufficientRewardsError(_amount, _available);
488:      }

490:      claimerRewards[msg.sender] -= _amount;
491:      _transfer(_to, _amount);
492:      emit WithdrawClaimRewards(_to, _amount, _available);
493:  }
```

```
diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..596fa1c 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -487,7 +487,7 @@ contract PrizePool is TieredLiquidityDistrib
     revert InsufficientRewardsError(_amount, _available);
 }

-    claimerRewards[msg.sender] -= _amount;
+    claimerRewards[msg.sender] = _available - _amount;
     _transfer(_to, _amount);
     emit WithdrawClaimRewards(_to, _amount, _available);
 }
```



Emitting storage values instead of the memory one.

Here, the values emitted shouldn't be read from storage. The existing memory values should be used instead:

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L372-L385>

	Min	Average	Median	Max	
Before	1901	59662	67737	79917	
After	1901	59649	67723	79903	

```
File: /src/PrizePool.sol
```

```
372:     _lastClosedDrawStartedAt = openDrawStartedAt_;
373:     _lastClosedDrawAwardedAt = uint64(block.timestamp);

375:     emit DrawClosed(
376:         lastClosedDrawId,
377:         winningRandomNumber_,
378:         _numTiers,
379:         _nextNumberOfTiers,
380:         _reserve,
381:         prizeTokenPerShare,
382:         _lastClosedDrawStartedAt
383:     );
```

```
diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..74d3a25 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -379,7 +379,7 @@ contract PrizePool is TieredLiquidityDistribution {
     _nextNumberOfTiers,
     _reserve,
     prizeTokenPerShare,
-    _lastClosedDrawStartedAt
+    openDrawStartedAt_
    );

    return lastClosedDrawId;
```



Optimizing the check order for cost efficient function execution

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 Gcoldload (2100 gas) in a function that may ultimately revert, in the unhappy case.

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/abstract/TieredLiquidityDistributor.sol#L235-L326>



Validate parameters before doing any `stores` (more than 2200 gas could be saved in case of a revert)

```
File: /src/abstract/TieredLiquidityDistributor.sol
235:  constructor(uint8 _numberOfTiers, uint8 _tierShares, uint8
236:      numberOfTiers = _numberOfTiers;
237:      tierShares = _tierShares;
238:      canaryShares = _canaryShares;
239:      reserveShares = _reserveShares;

320:  if (_numberOfTiers < MINIMUM_NUMBER_OF_TIERS) {
321:      revert NumberOfTiersLessThanMinimum(_numberOfTiers);
322:  }
323:  if (_numberOfTiers > MAXIMUM_NUMBER_OF_TIERS) {
324:      revert NumberOfTiersGreaterThanMaximum(_numberOfTiers)
325:  }
326: }
```

We are assigning some Storage variables a function parameter on top of doing so other minor operations. We then check if the parameters passed to the constructor meet the required specification; i.e.:

```
if (_numberOfTiers < MINIMUM_NUMBER_OF_TIERS) {
    revert NumberOfTiersLessThanMinimum(_numberOfTiers);
}
if (_numberOfTiers > MAXIMUM_NUMBER_OF_TIERS) {
    revert NumberOfTiersGreaterThanMaximum(_numberOfTiers);
}
```

If the above checks fail, we end up reverting the whole transaction, meaning the gas spent writing to storage would go to waste.

```
diff --git a/src/abstract/TieredLiquidityDistributor.sol b/src/a
index 8238103..34c2050 100644
--- a/src/abstract/TieredLiquidityDistributor.sol
```

```

+++ b/src/abstract/TieredLiquidityDistributor.sol
@@ -233,6 +233,12 @@ contract TieredLiquidityDistributor {
    * @param _reserveShares The number of shares to allocate to
    */
    constructor(uint8 _numberOfTiers, uint8 _tierShares, uint8 _c
+   if (_numberOfTiers < MINIMUM_NUMBER_OF_TIERS) {
+       revert NumberOfTiersLessThanMinimum(_numberOfTiers);
+   }
+   if (_numberOfTiers > MAXIMUM_NUMBER_OF_TIERS) {
+       revert NumberOfTiersGreaterThanMaximum(_numberOfTiers);
+   }
    numberOfTiers = _numberOfTiers;
    tierShares = _tierShares;
    canaryShares = _canaryShares;
@@ -317,12 +323,6 @@ contract TieredLiquidityDistributor {
    _tierShares
    );

-   if (_numberOfTiers < MINIMUM_NUMBER_OF_TIERS) {
-       revert NumberOfTiersLessThanMinimum(_numberOfTiers);
-   }
-   if (_numberOfTiers > MAXIMUM_NUMBER_OF_TIERS) {
-       revert NumberOfTiersGreaterThanMaximum(_numberOfTiers);
-   }
}

```

<https://github.com/GenerationSoftware/pt-v5-vault/blob/b1deb5d494c25f885c34c83f014c8a855c5e2749/src/Vault.sol#L550-L568>



Reorder the checks to have the cheaper validations come first

```

File: /src/Vault.sol
550: function liquidate(
551:     address _account,
552:     address _tokenIn,
553:     uint256 _amountIn,
554:     address _tokenOut,
555:     uint256 _amountOut
556: ) public virtual override returns (bool) {
557:     _requireVaultCollateralized();
558:     if (msg.sender != address(_liquidationPair))
559:         revert LiquidationCallerNotLP(msg.sender, address(_liq

```

```

560:     if (_tokenIn != address(_prizePool.prizeToken()))
561:         revert LiquidationTokenInNotPrizeToken(_tokenIn, address(_prizePool.prizeToken()));
562:     if (_tokenOut != address(this))
563:         revert LiquidationTokenOutNotVaultShare(_tokenOut, address(this));
564:     if (_amountOut == 0) revert LiquidationAmountOutZero();

566:     uint256 _liquidableYield = _liquidatableBalanceOf(_tokenOut);
567:     if (_amountOut > _liquidableYield)
568:         revert LiquidationAmountOutGTYield(_amountOut, _liquidableYield);

```

The first few checks are either reading from storage or making some external calls. These operations are quite expensive in terms of gas. We also have some checks that validate some function parameters. Reading function parameters is very cheap. As it is, in case we revert on the function parameter check, the gas consumed doing external calls or state reads would be wasted. We should reorder the checks to reduce gas consumed in case of a revert. Cheaper checks should be done first as shown below:

```

diff --git a/src/Vault.sol b/src/Vault.sol
index e92ecaf..7652d7a 100644
--- a/src/Vault.sol
+++ b/src/Vault.sol
@@ -554,19 +554,23 @@ contract Vault is ERC4626, ERC20Permit, ILiquidatable {
     address _tokenOut,
     uint256 _amountOut
 ) public virtual override returns (bool) {
-    _requireVaultCollateralized();
-    if (msg.sender != address(_liquidationPair))
-        revert LiquidationCallerNotLP(msg.sender, address(_liquidationPair));
-    if (_tokenIn != address(_prizePool.prizeToken()))
-        revert LiquidationTokenInNotPrizeToken(_tokenIn, address(_prizePool.prizeToken()));
+    if (_amountOut == 0) revert LiquidationAmountOutZero();
+
+    if (_tokenOut != address(this))
+        revert LiquidationTokenOutNotVaultShare(_tokenOut, address(this));
-    if (_amountOut == 0) revert LiquidationAmountOutZero();

    uint256 _liquidableYield = _liquidatableBalanceOf(_tokenOut);
    if (_amountOut > _liquidableYield)
        revert LiquidationAmountOutGTYield(_amountOut, _liquidableYield);

+    _requireVaultCollateralized();
+

```

```

+     if (msg.sender != address(_liquidationPair))
+         revert LiquidationCallerNotLP(msg.sender, address(_liquidationPair));
+
+     if (_tokenIn != address(_prizePool.prizeToken()))
+         revert LiquidationTokenInNotPrizeToken(_tokenIn, address(_prizePool.prizeToken()));
+
+     _prizePool.contributePrizeTokens(address(this), _amountIn);
+
+     if (_yieldFeePercentage != 0) {

```



Don't cache a state variable if it's used only once

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L851-L877>

Gas benchmarks are based on function `isWinner` that calls this internal function.

	Min	Average	Median	Max	
Before	1046	26982	20177	55866	
After	1046	27014	20232	55921	

```

File: /src/PrizePool.sol
851:     uint8 _numberOfTiers = numberOfTiers;
852:     uint32 tierPrizeCount = _getTierPrizeCount(_tier, _numberOfTiers);
853:
854:     if (_prizeIndex >= tierPrizeCount) {
855:         revert InvalidPrizeIndex(_prizeIndex, tierPrizeCount, _numberOfTiers);
856:     }
857:
858:     uint256 userSpecificRandomNumber = TierCalculationLib.calculateRandomNumber(
859:         _user,
860:         _tier,
861:         _prizeIndex,
862:         _winningRandomNumber
863:     );
864:     (uint256 _userTwab, uint256 _vaultTwabTotalSupply) = _getTwabs(
865:         _vault,
866:         _user,
867:         _drawDuration
868:     );

```

```

870:     return
871:     TierCalculationLib.isWinner(
872:         userSpecificRandomNumber,
873:         uint128(_userTwab),
874:         uint128(_vaultTwabTotalSupply),
875:         _vaultPortion,
876:         _tierOdds
877:     );

```

```

diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..9407b1c 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -848,8 +848,7 @@ contract PrizePool is TieredLiquidityDistribution {
    SD59x18 _tierOdds,
    uint16 _drawDuration
) internal view returns (bool) {
-    uint8 _numberOfTiers = numberOfTiers;
-    uint32 tierPrizeCount = _getTierPrizeCount(_tier, _numberOfTiers);
+    uint32 tierPrizeCount = _getTierPrizeCount(_tier, numberOfTiers);

    if (_prizeIndex >= tierPrizeCount) {
        revert InvalidPrizeIndex(_prizeIndex, tierPrizeCount, _tier);
    }
}

```



Do not cache immutable values

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/4bc8a12b857856828c018510b5500d722b79ca3a/src/PrizePool.sol#L781-L810>

Benchmark is based on the function `nextNumberOfTiers`, which calls our internal function.

	Min	Average	Median	Max
Before	765	765	765	765
After	757	757	757	757

File: /src/PrizePool.sol

```

781:     function _computeNextNumberOfTiers(uint8 _numTiers) internal
782:         UD2x18 _claimExpansionThreshold = claimExpansionThreshold;

793:     // check to see if we need to expand the number of tiers
794:     if (
795:         _nextNumberOfTiers >= _numTiers &&
796:         canaryClaimCount >=
797:             fromUD60x18(
798: intoUD60x18(_claimExpansionThreshold).mul(_canaryPrizeCountF
799:         ) &&
800:         claimCount >=
801:             fromUD60x18(
802: intoUD60x18(_claimExpansionThreshold).mul(toUD60x18(_est
803:         )

```

```

diff --git a/src/PrizePool.sol b/src/PrizePool.sol
index a42a27e..accf0a8 100644
--- a/src/PrizePool.sol
+++ b/src/PrizePool.sol
@@ -779,7 +779,6 @@ contract PrizePool is TieredLiquidityDistrib
    /// @param _numTiers The current number of tiers
    /// @return The number of tiers for the next draw
    function _computeNextNumberOfTiers(uint8 _numTiers) internal
-    UD2x18 _claimExpansionThreshold = claimExpansionThreshold;

    uint8 _nextNumberOfTiers = largestTierClaimed + 2; // canary
    _nextNumberOfTiers = _nextNumberOfTiers > MINIMUM_NUMBER_OF
@@ -795,11 +794,11 @@ contract PrizePool is TieredLiquidityDistrib
    _nextNumberOfTiers >= _numTiers &&
    canaryClaimCount >=
    fromUD60x18(
-        intoUD60x18(_claimExpansionThreshold).mul(_canaryPrizeCo
+        intoUD60x18(claimExpansionThreshold).mul(_canaryPrizeCo
    ) &&
    claimCount >=
    fromUD60x18(
-        intoUD60x18(_claimExpansionThreshold).mul(toUD60x18(_es
+        intoUD60x18(claimExpansionThreshold).mul(toUD60x18(_est
    )
    ) {
        // increase the number of tiers to include a new tier

```

It is important to emphasize that the provided recommendations aim to enhance the efficiency of the code without compromising its readability. We understand the value of maintainable and easily understandable code to both developers and auditors.

As you proceed with implementing the suggested optimizations, please exercise caution and be diligent in conducting thorough testing. It is crucial to ensure that the changes are not introducing any new vulnerabilities and that the desired performance improvements are achieved. Review code changes, and perform thorough testing to validate the effectiveness and security of the refactored code.

Should you have any questions or need further assistance, please don't hesitate to reach out.

[asselstine \(PoolTogether\) confirmed](#)

[PierrickGT \(PoolTogether\) commented:](#)

1 - Fixed in this commit: <https://github.com/GenerationSoftware/pt-v5-vault/pull/45/commits/8ea87b645abc72443efbc2696092950f7751e9b7>.

2, 4, 5, 6, 7, 8, 9, 10: have been fixed.

3: does not save gas anymore now that we use `CREATE2` .



Audit Analysis

For this audit, 13 analysis reports were submitted by wardens. An analysis report examines the codebase as a whole, providing observations and advice on such topics as architecture, mechanism, or approach. The [report highlighted below](#) by OxStalin received the top score from the judge.

The following wardens also submitted reports: [DadeKuma](#), [peanuts](#), [OxSmartContract](#), [catellatech](#), [dirk_y](#), [squeaky_cactus](#), [K42](#), [keccak123](#), [Jeiwan](#), [ABAIKUNANBAEV](#), [3docSec](#), and [OxWaitress](#).



Introduction

The PoolTogether protocol proposition is to gamify the yields earned by depositing assets into vaults. It is an interesting proposition to allow users that are attracted to all-

or-nothing types of games because the PoolTogether vaults allow vault depositors to play at random the possibility to win everyone's yields.

The protocol aims to make the vaults creations fully autonomous and in a permissionless way. This will allow anybody to create vaults that offer the gamification feature that the PoolTogether protocol is building.



Analysis of the Codebase

The codebase of this audit included 4 components of the PoolTogether protocol, the `Vault`, the `PrizePool`, the `TwabController` & the `Claimer`. The 4 components are highly interconnected and dependent on each other.

The Vault can be thought of as the core component because it interacts with all the others.

- It is in charge of updating the `PrizePool` accounting and sending and reading data from the `TwabController` where all the user's balances are tracked.
- The `Claimer` is a more optional component because it is up to the `Vault`'s owner to set a claimer using a deployed `Claimer` contract or not.



Relationship Diagram

There is a Vault Factory that is in charge of deploying new Vaults.

- The deployment of new vaults is fully decentralized. It is not required to ask permission from the protocol to deploy a new Vault, or belong to a whitelist or a restriction of any type; it is fully autonomous and decentralized.

The `YieldVault` is not exactly a contract coded by the Protocol team, but it is expected to be an ERC4626-compliant contract.

- Even though there is not much documentation about the `YieldVault`, I consider it to be a key component of the protocol because all the user's assets will end up being deposited into the `YieldVaults`.

I think that this protocol uniqueness feature is how they've accomplished generating on-chain randomness to match the pseudo-randomness values with past values that have been generated by the user's & vault's balances during a specific period of time.

- Using past values makes it near to impossible to try to compute & force a specific account to be the winner. The reason is, that even though the pseudorandom value can be computed before the tx is actually executed, the past values can't be modified and forced to match the pseudorandom value.
- The Math & Logic involved in this process is quite complex, it is really an artwork.
- Also, the proposition to gamify the yields and allow users to “gamble” their yields in exchange for potentially earning the yields of all the other depositors is quite interesting because most of the crypto users are investors of high-risk profiles, which I'm pretty sure more than 1 will be interested in this offer.



Architecture Feedback, Centralization & Systemic Risks

The protocol is aiming to achieve a fully autonomous and decentralized architecture, but with decentralization comes risks.

The Asset Flows when depositing and withdrawing to/from the PoolTogether vaults are similar to the process explained in the below diagrams.

As we can see, **all the user assets are really deposited in the `YieldVaults`**, which is why ensuring that the `YieldVaults` are real and safe vaults is a critical task that the protocol should take care of.



Deposit Assets Flow



Withdraw Assets Flow

The decentralization part is about allowing anybody to create new vaults by using the `VaultFactory` that is created by the Protocol. Problems can arise if the protocol doesn't enforce the newly created vaults to use the expected contracts for the accounting and holding of the user's assets

- Malicious users could abuse and deploy fake contracts and set them as the `PrizePool`, `TwabController`, and also create fake `YieldVaults`. This is a critical risk for users' safeness because all the vaults created by the `VaultFactory` should indicate to the users that these vaults are safe to interact with.

That is a critical problem that arises with decentralization, because the creation of the vaults is decentralized, but the ownership of each vault isn't, there is a vault owner who could abuse its power if the contracts don't limitate it.

- Limitate the vault owner's power is a task of the protocol because the protocol is the one who is allowing anybody to create new vaults using their VaultFactory.

About the Liquidity Distribution fully relies on some advanced math formulas, which basically compute the number of liquidity to be disbursed across the different draws and in each draw computes how much liquidity to distribute among the different tiers.

- Tiers are flexible, meaning, they can grow or shrink if required.

 Liquidity Distribution

 Draw's Liquidity Distribution

The winners are determined based on pseudorandom-generated values and matching values that have been generated in the past between a specific period of time

- To achieve this, the protocol does some computation between the `PrizePool` and `TwabController` contracts.

 determining the winner

The `Claimer` is only one in charge to claim the prizes of the winners (all the prizes are sent to the corresponding winner). The `Claimer` earns a fee in exchange for claiming the prizes on behalf of the winners.



Recommendations

Always have in mind the fact that if something goes wrong in a vault that was deployed using the `VaultFactory`, most users will blame the protocol, not the vault owner. In a normal user's mind, the protocol is responsible to ensure that each individual vault is safe and sound.

That being said, even though the creation of vaults will be fully decentralized, make sure that the contracts limitate the vault's owners to the minimum privileges that they require to manage the vault.

Also, make sure that the contracts enforce the vaults to use the right and expected contracts. Most of the time allowing users to set arbitrary addresses for contracts that interact with the protocol opens up some attack vectors.

Always try to mitigate the number of open doors that could be exploited or abused by the vault owners.



Time spent

40 hours

I spent 6 days on this audit, each day I put in around 5-8 hours of work. In total would be around 40 hours.

- 3 days were to analyze and fully understand the codebase.
- 2 days were dedicated only to bug hunting. I didn't follow the recognition pattern; I was mostly focused on identifying critical issues caused by incorrect assumptions and bugs in the code.
- The last day was dedicated to filling out the reports and writing this analysis, as well as creating the diagrams.

[asselstine \(PoolTogether\) confirmed](#)



Mitigation Review



Introduction

Following the C4 audit, 3 wardens ([dirk_y](#), [rvierdiiev](#) and [OxStalin](#)) reviewed the mitigations for all identified issues. Additional details can be found within the [PoolTogether Mitigation Review repository](#).



Overview of Changes

- Twab Controller now checks query timestamps for safety. When requesting a twab, timestamps:
 1. are required to be finalized; i.e. cannot occur during an overwrite period

2. are aligned to period boundaries.

- Tier adjustment algorithm

1. The canary tier has been simplified, so that the prize size is adjusted instead of the number of prizes. Now the unified claim count across normal + canary tiers is used to determine the next number of tiers.

- Vault exchange rate

1. The Vault now assumes shares are 1:1 with underlying assets, unless the vault is undercollateralized. No exchange rate is stored.



Mitigation Review Scope



Branch

- [pt-v5-prize-pool](#)
- [pt-v5-twab-controller](#)
- [pt-v5-vault](#)
- [pt-v5-claimer](#)

(all PRs have been merged into main)



Individual PRs

URL	Mitigation of	Purpose
VAULT-PR-13	H-01	The issue turned out not to be the case; the exchange rate is always ≤ 1 (yield is liquidated). However, comments were added to clarify the behaviour
VAULT-PR-9	H-02	Added SafeCast
VAULT-PR-6	H-03	Fixed conversion and naming of field
VAULT-PR-7	H-04	Removed recipient param
VAULT-PR-19	H-05	Removed recipient param
TWAB-PR-7	H-06	Added check for zero address
VAULT-PR-13	H-07	Fixed check for partial collateralization
PRIZE-PR-18	H-08	Fixed reserve accounting

URL	Mitiga tion of	Purpose
VAULT-PR-37	H-09	Fixed undercollateralized redemption while providing an exit
VAULT-PR-21	M-02	Added hook gas limits and error handling
TWAB-PR-5	M-03	Ensure search timestamps are on or at period end timestamps
VAULT-PR-11	M-04	Fixed 4626 compliance
TWAB-PR-6	M-05	Reject transfers to the sponsorship address
PRIZE-PR-31	M-06	Record deployer and permission draw manager
PRIZE-COMMIT-ba56ea8b	M-07	Upgrade PRBMath to v4 and Solidity to 0.8.19
VAULT-PR-25	M-08	Used CREATE2 for VaultFactory
CLAIMER-PR-7	M-09	Made max fee a function of the tier's prize size, not the smallest prize size
PRIZE-PR-16	M-10	Improved handling of prize size overflow
VAULT-PR-15	M-11	Removed <code>mintWithPermit</code>
PRIZE-PR-24	M-12	Recomputed Tier odds, reduced number of tiers, made grand prize period draws configurable
TWAB-PR-5	M-13	Aligned twab queries on period boundaries
PRIZE-PR-17	M-14	Superseceded by the simplification of tier expansion logic
PRIZE-PR-17	M-15	Improved tier shrinking, retention, and expansion logic
PRIZE-PR-17	M-16	Simplified tier expansion logic
VAULT-PR-27	M-18	Redeemed underlying liquidity using shares, not assets directly. This socializes losses, if any.
PRIZE-PR-22	M-19	Added SafeCast
VAULT-PR-17	M-20	Vault shares are now 1:1 with underlying asset, and the exchange logic has been simplified
PRIZE-PR-10	M-21	Draw query ends on closed draw
VAULT-PR-18	M-22	Improved vault share / asset calculation
VAULT-PR-11	M-23	Fixed compliance with 4626
CLAIMER-PR-8 VAULT-PR-22 PRIZE-PR-23	M-24	Added minVrgdaFee and allowed silent failure of claims
VAULT-PR-20	M-25	Fixed permit implementations

URL	Mitiga tion of	Purpose
VAULT-PR-9	M-26	Added SafeCast
PRIZE-PR-17	M-27	Unified standard and canary tier



Out of Scope

Issue	Reason	
M-01	Acknowledged issue but it will not be fixed.	
M-17	Permissionless nature of vaults means that anyone can create custom vaults.	



Mitigation Review Summary

Original Issue	Status	Full Details
H-01	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
H-02	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
H-03	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
H-04	Mitigation confirmed	Reports from rvierdiiev , dirk_y and OxStalin
H-05	Mitigation error	Report from rvierdiiev - details also shared below
H-06	Mitigation confirmed with comments	Reports from OxStalin , dirk_y and rvierdiiev
H-07	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
H-08	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
H-09	Not fully mitigated	Reports from OxStalin and rvierdiiev - details also shared below
M-02	Not fully mitigated	Reports from dirk_y , rvierdiiev and OxStalin - details also shared below
M-03	Mitigation confirmed	Reports from dirk_y and rvierdiiev
M-04	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-05	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-06	Mitigation confirmed	Reports from rvierdiiev , dirk_y and OxStalin
M-07	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev

Original Issue	Status	Full Details
M-08	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
M-09	Mitigation error	Report from dirk_y - details also shared below
M-10	Mitigation confirmed	Reports from OxStalin and dirk_y
M-11	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
M-12	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-13	Mitigation confirmed with comments	Reports from dirk_y , OxStalin and rvierdiiev
M-14	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-15	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-16	Mitigation error	Report from dirk_y - details also shared below
M-18	Not fully mitigated	Report from dirk_y - details also shared below
M-19	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-20	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
M-21	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
M-22	Not fully mitigated	Reports from dirk_y and OxStalin - details also shared below
M-23	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-24	Mitigation confirmed	Reports from rvierdiiev , dirk_y and OxStalin
M-25	Mitigation confirmed	Reports from dirk_y , rvierdiiev and OxStalin
M-26	Mitigation confirmed	Reports from OxStalin , dirk_y and rvierdiiev
M-27	Mitigation confirmed	Reports from rvierdiiev , dirk_y and OxStalin

The wardens determined that 4 findings from the original audit were not fully mitigated. They also surfaced 4 mitigation errors (1 High and 3 Medium severity) and 1 new Medium severity issue.



H-05 Mitigation error: The `sponsorWithPermit` function allows anyone to provide a valid permit for asset

Submitted by [rvierdiiev](#)

Severity: Medium

The `sponsor` function allows a caller to delegate their shares to the special address. In this case, the caller loses the ability to win prizes. The previous version of code had the `sponsor` function, which allowed the deposit of funds on behalf of the owner and delegate all user's funds to the `SPONSORSHIP_ADDRESS`. As result, an attacker had the ability to deposit small amount of funds (or 0) on behalf of user and make them delegate to the `SPONSORSHIP_ADDRESS`.



Solution

The PoolTogether team has fixed that issue by removing [this function](#). So now, the attacker can't directly call `sponsor` for any address.

However, there is another function, `sponsorWithPermit`. This function allows anyone to provide a valid permit for asset and then do the sponsoring. It's possible that a user will create a permit for other purposes (to just run the `depositWithPermit` function). In this case, an attacker (which can be some other web portal) can reuse this permit and make the user delegate all of their balance to the `SPONSORSHIP_ADDRESS`, instead of just depositing.

This issue stands, because currently the user can't be sure how their signed message (permit) will be used. A user can sign the permit to deposit, but it will be used to deposit and delegate to the sponsorship address.

[PierrickGT \(PoolTogether\) commented:](#)

Nice find, indeed. I've removed the `sponsorWithPermit` function in the following PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/40>.

Note: For full discussion, see [here](#).



H-09 Not fully mitigated: `_yieldVault.maxWithdraw()` returning a different value than the expected real balance of the caller

Submitted by [OxStalin](#), also found by [rvierdiiev](#)

Original Issue

H-09: Vault is not compatible with some ERC4626 vaults

Details

The issue is about the Vault contract not being compatible with some ERC4626 vaults, such as `DnGmxSeniorVault`.

- The warning is about vaults that don't stick to the ERC4626 standard, and their `maxWithdraw()` function returns a different value than the expected real balance of the caller.

Mitigation

The mitigation implemented new logic to determine if the Vault is collateralized or not. Now, all shares are backed 1:1 with deposited assets.

When withdrawing and redeeming, the new implementation checks if the vault is collateralized or not; if it's collateralized, the shares:asset ratio is 1:1, if it's under-collateralized, the ratio will be different, and the functions will compute how many shares are required to withdraw the desired amount of assets.

- If the vault is under-collateralized:
 - The `withdraw()` function will compute the required amount of shares that are required to get the desired amount of assets.
 - The `redeem()` function will compute how many assets the specified amount of shares can get.

The new implementation mitigates socializing losses when the vault is under-collateralized. Now, each user can withdraw/redeem up to their `maxWithdraw()` / `maxRedeem()` amounts of assets, which those values are determined based on the amount of shares that the owner has.

Conclusion

I consider that the new implementation mitigates issues that can arise because of ERC4626 vaults not implementing the expected ERC4626 standard.

In regards to the exact issue pointed out in the original finding about the `_yieldVault.maxWithdraw()` returning a different value than the expected real balance of the caller, I do agree with the sponsor's comment that the issue is more of warning about possible incompatibilities with some ERC4626 vaults. For that reason, I think that there is not such a fix for that exact "issue", that is more a thing at the administration level when creating new vaults. The vault's creator should do their due diligence to verify that the ERC4626 Vault follows in the most part the standard.

[dirk_y \(warden\) commented via issue #35:](#)

Sounds like we've all kind of come to the same conclusion here (see [#68](#) and [#97](#)). Basically, the only real resolution is discouraging certain types of underlying yield vaults

[OxStalin \(warden\) commented via issue #35:](#)

@dirk_y - I agree with your comment. I think this can be classified as a Q&A, since this is more of an administration thing. It is up to the vault's creator to select the underlying vault, and it is up to the users to decide if they want to put their assets in a PT Vault using an underlying vault that doesn't stick to the standard and/or implements lossy strategies.

[asselstine \(PoolTogether\) commented via issue #35:](#)

Our mitigation is that we will declare that vault compatibility needs to be checked before integrating, and that yield vaults that round down on deposit are problematic.



M-02 Not fully mitigated: A malicious user can still gas grief a claimer by deliberately reverting in a hook

Submitted by [dirk_y](#), also found by [rvierdiev](#) and [OxStalin](#)



Original Issue

[M-02: Unintended or malicious use of prize winners' hooks](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-vault/blob/main/src/Vault.sol#L1397>

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/main/src/Claimer.sol#L163>



Comments

In the previous implementation, a malicious user could set arbitrary vault hooks for `afterClaimPrize` and `beforeClaimPrize` that could be used to gas grief the claimer; or cause other claims in the same call to fail by deliberately reverting.



Mitigation

The referenced PR does solve the original issue by capping the gas sent to external calls and safely catching reverts. I was slightly concerned that enough gas was being passed to the hooks for a single reentrant call to front-run a prize claim; however, this has been fixed by another change where previously claimed prizes safely return 0. However, this mitigation is unresolved by another change.



Persisting issue

There was another change made to the repo where claiming logic was simplified and mainly moved out of the Vault contract. However, with this change any reverts are not being safely caught:

```
function _claim(
    Vault _vault,
    uint8 _tier,
    address[] calldata _winners,
    uint32[][] calldata _prizeIndices,
    address _feeRecipient,
    uint96 _feePerClaim
) internal returns (uint256) {
    uint256 actualClaimCount;
    uint256 winnersLength = _winners.length;
    for (uint256 w = 0; w < winnersLength; w++) {
        uint256 prizeIndicesLength = _prizeIndices[w].length;
        for (uint256 p = 0; p < prizeIndicesLength; p++) {
            if (0 != _vault.claimPrize(
                _winners[w],
                _tier,
                _prizeIndices[w][p],
                _feePerClaim,
```

```

        _feeRecipient
    )) {
        actualClaimCount++;
    } else {
        emit AlreadyClaimed(_winners[w], _tier, _prizeIndices[w],
    }
}
}
return actualClaimCount;
}

```

This is a problem because if the hook reverts, this is caught and then another revert is thrown:

```

function claimPrize(
    address _winner,
    uint8 _tier,
    uint32 _prizeIndex,
    uint96 _fee,
    address _feeRecipient
) external onlyClaimer returns (uint256) {
    VaultHooks memory hooks = _hooks[_winner];
    address recipient;

    if (hooks.useBeforeClaimPrize) {
        try
            hooks.implementation.beforeClaimPrize{ gas: HOOK_GAS }(
                _winner,
                _tier,
                _prizeIndex,
                _fee,
                _feeRecipient
            )
        returns (address result) {
            recipient = result;
        } catch (bytes memory reason) {
            revert BeforeClaimPrizeFailed(reason);
        }
    } else {
        recipient = _winner;
    }
}

```

As a result, a malicious user can still gas grief a claimer by deliberately reverting in a hook.



Recommendation

The reverts should be safely caught in the Claimer contract, similarly to the initial change (and linked PR) that fixed this issue originally.



Assessed type

Loop

[asselstine \(PoolTogether\) commented:](#)

Handled here <https://github.com/GenerationSoftware/pt-v5-claimer/pull/13>.



M-09 Mitigation error: Auctions run at significantly different speeds for different prize tiers

Submitted by [dirk_y](#)

Severity: Medium



Lines of code

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/main/src/Claimer.sol#L76-L78>

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/main/src/Claimer.sol#L136>

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/main/src/Claimer.sol#L262-L264>

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/main/src/Claimer.sol#L223-L250>

<https://github.com/GenerationSoftware/pt-v5-claimer/blob/main/src/Claimer.sol#L289>



Comments

The V5 implementation delegates the task of claiming prizes to a network of claimers. The fees received by a claimer are calculated based on a dutch auction and limited based on the prize size of the highest tier (the smallest prize). As a result, it is possible that the gas price could exceed the fee received by claimers, leading to prizes not being claimed. If any high value prizes happen to be drawn during this period then they will go unclaimed.

🔗 Mitigation

The new implementation only computes the `maxFee` size based on the prize tier that is being claimed for. As a result, the fees received by claimers are now larger for larger prize tiers; thereby, incentivising lower tiers (i.e. those with higher prizes) to be claimed first and resulting in more fees paid to claimers.

🔗 New issue

Because all the tiers run on the same auction, each auction will now run at a completely different speed.

🔗 Impact

If the `_maximumFee` parameter specified in the constructor is relatively small, then the `maxFee` for the lower prize tiers (higher prizes) will never be reached anyway. If the `_maximumFee` is relatively large to give sufficient range for the auctions, the auctions for higher tiers (lower prizes) will ramp up very quickly to the max limit based on the prize tier. The real impact of this is that auctions are now running inefficiently, where fees are likely to be higher than they could be for the higher tiers (i.e. the bots are getting more fees than they would be willing to accept).

🔗 Proof of Concept

Based on the updated implementation, the maximum fee to be paid is now a function of the tier being claimed for, not the total number of active tiers:

```
function _computeMaxFee(uint8 _tier) internal view returns (uint)
    return UD60x18.unwrap(maxFeePortionOfPrize.intoUD60x18()).mul(
    }
```

The return value of this call is used as the first parameter for calls to

`_computeFeePerClaim` and in turn the last parameter of

`_computeFeeForNextClaim`:

```
function _computeFeeForNextClaim(
    uint256 _minimumFee,
    SD59x18 _decayConstant,
    SD59x18 _perTimeUnit,
    uint256 _elapsed,
    uint256 _sold,
    uint256 _maxFee
) internal pure returns (uint256) {
    uint256 fee = LinearVRGDALib.getVRGDAPrice(
        _minimumFee,
        _elapsed,
        _sold,
        _perTimeUnit,
        _decayConstant
    );
    return fee > _maxFee ? _maxFee : fee;
}
```

As you can see, the fee is capped based on the `maxFee` for the prize tier being claimed for. This seems to be doing what was intended; however, there is only one auction for all the tiers, and thus the auction decay constant is consistent across all the tiers:

```
constructor(
    PrizePool _prizePool,
    uint256 _minimumFee,
    uint256 _maximumFee,
    uint256 _timeToReachMaxFee,
    UD2x18 _maxFeePortionOfPrize
) {
    if (_minimumFee >= _maximumFee) {
        revert MinFeeGeMax(_minimumFee, _maximumFee);
    }
    prizePool = _prizePool;
    maxFeePortionOfPrize = _maxFeePortionOfPrize;
    decayConstant = LinearVRGDALib.getDecayConstant(
        LinearVRGDALib.getMaximumPriceDeltaScale(_minimumFee, _max.
    );
};
```



```
    minimumFee = _minimumFee;  
    timeToReachMaxFee = _timeToReachMaxFee;  
}
```

Whichever is the lowest of the auction `_maximumFee` and the tier `maxFee` is the max fee that could possibly be collected. In order to allow all prize tiers to be claimed it is likely that the `_maximumFee` in the constructor will be increased; however, this now means the decay constant is greater and therefore, the auctions ramp up faster. For tiers with a lower max fee, this means the `maxFee` is reached significantly faster, leading to inefficient auctions.



Recommendation

Potentially, there is another angle to resolve the issue reported in the original audit by having a mechanism that allows the `maxFee` to be increased if not enough prizes have been redeemed in the last draw, although this introduces additional complexity.

Alternatively there needs to be an auction for each tier with its own decay constant (i.e. min and max fees) to ensure that all the auctions are ramping up in a similar (but not necessarily identical) manner. In my opinion, this option makes the most sense and is the least complex and therefore, also the hardest to manipulate (if at all).

[asselstine \(PoolTogether\) acknowledged](#)

Note: For full discussion, see [here](#).



M-16 Mitigation error: Number of prize tiers may never scale due to aggressive new algorithm.

Submitted by [dirk_y](#)

Severity: Medium



Lines of code

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/main/src/PrizePool.sol#L369>

<https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/main/src/PrizePool.sol#L369>

[pool/blob/main/src/PrizePool.sol#L807-L811](#)

[https://github.com/GenerationSoftware/pt-v5-prize-](https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/main/src/abstract/TieredLiquidityDistributor.sol#L602-L619)

[pool/blob/main/src/abstract/TieredLiquidityDistributor.sol#L602-L619](#)

[https://github.com/GenerationSoftware/pt-v5-prize-](https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/main/src/abstract/TieredLiquidityDistributor.sol#L156)

[pool/blob/main/src/abstract/TieredLiquidityDistributor.sol#L156](#)

[https://github.com/GenerationSoftware/pt-v5-prize-](https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/main/src/libraries/TierCalculationLib.sol#L134-L147)

[pool/blob/main/src/libraries/TierCalculationLib.sol#L134-L147](#)



Comments

This issue is very similar to M-14 but covers another edge case where the threshold check is not performed when there are currently 14 prize tiers and at least 1 canary tier is claimed. This is due to an early return of `MAXIMUM_NUMBER_OF_TIERS`.



Mitigation

The updated implementation has significantly changed the tier expansion logic so that it only depends on the total number of prize claims. The current number of tiers and the number of canary tier claims has no impact on the tier expansion logic and therefore, the original issue has been resolved. However, the updated logic has introduced a new issue.

Note: The same PR fixes multiple issues, but I have arbitrarily chosen to link this new issue with M-16.



New issue

With the new tier expansion algorithm, it is highly likely that the number of prize tiers will stagnate since the percentage of prizes that need to be claimed for a tier expansion increases as the number of tiers increase.



Proof of Concept

When a draw is closed, the next number of tiers is computed based on the total number of claims with a call to `_computeNextNumberOfTiers`:

```
function _computeNextNumberOfTiers(uint32 _claimCount) internal  
    // claimCount is expected to be the estimated number of claim  
    uint8 numTiers = _estimateNumberOfTiersUsingPrizeCountPerDraw  
    return numTiers > MAXIMUM_NUMBER_OF_TIERS ? MAXIMUM_NUMBER_OF
```

```
}
```

Where the first part of the underlying

`_estimateNumberOfTiersUsingPrizeCountPerDraw` method looks like:

```
function _estimateNumberOfTiersUsingPrizeCountPerDraw(uint32 _prizeCount)
    if (_prizeCount < ESTIMATED_PRIZES_PER_DRAW_FOR_4_TIERS) {
        return 3;
    } else if (_prizeCount < ESTIMATED_PRIZES_PER_DRAW_FOR_5_TIERS) {
        return 4;
    }
}
```

Thus, for the number of tiers to increase from 3 to 4, the number of prize claims must be greater than or equal to `ESTIMATED_PRIZES_PER_DRAW_FOR_4_TIERS`:

```
ESTIMATED_PRIZES_PER_DRAW_FOR_4_TIERS = TierCalculationLib.estimatePrizeCount(3, 16, 1)
```

Here, the estimated claim count is the sum of the number of prizes for a tier multiplied by the tier odds for all the tiers:

```
function estimatedClaimCount(
    uint8 _numberOfTiers,
    uint24 _grandPrizePeriod
) internal pure returns (uint32) {
    uint32 count = 0;
    for (uint8 i = 0; i < _numberOfTiers; i++) {
        count += uint32(
            uint256(
                unwrap(sd(int256(prizeCount(i))).mul(getTierOdds(i, _numberOfTiers)))
            )
        );
    }
    return count;
}
```

If we consider the actual number of prizes available for 3 tiers it is: $1 * 1/365 + 4 * 1 + 16 * 1 = 20 + 1/365 \approx 20$

The return value of `ESTIMATED_PRIZES_PER_DRAW_FOR_4_TIERS` is: $1 * 1/365 + 4 * 1/\sqrt{365} + 16 * 1 \approx 16$

This difference is due to the fact that the `getTierOdds` in `TierCalculationLib.sol` doesn't return 1 for both the highest normal prize tier and the canary tier (it just returns 1 for the canary tier).

In this case, in order for the number of prize tiers to increase, at least 16 prizes (i.e. 80%) need to be claimed in the previous claim window. This doesn't sound too bad, but because the number of prizes per tier scales exponentially, the percentage of prizes that need to be claimed increases as the number of prize tiers increase.

The effect is that as a higher number of tiers are reached, a higher percentage of canary prizes need to be claimed in order to increase the number of prize tiers. The end result is that it is likely the number of prize tiers will stagnate despite there being enough liquidity available to increase the number of prize tiers.



Recommendation

I recommend applying a reverse exponential weighting to the number of prize claims required for a tier expansion based on the number of tiers. By this, I mean that since the number of prizes per tier is 4^i , a similar weighting should be applied in the reverse direction to try to keep the percentage of prize claims required for a tier expansion relatively stable.

Alternatively, I would update `_estimateNumberOfTiersUsingPrizeCountPerDraw` to apply a fixed percentage to the actual number of prizes available (based on tiers odds of 1 for both the canary prize tier and the highest standard prize tier). I feel like this is the easier and also more logical fix.



Assessed type

Math

[asselstine \(PoolTogether\) confirmed and commented:](#)



The solution we propose for this is to adjust two things:

1. `estimatedClaimCount` now includes canary tiers; this means that each of the constants `ESTIMATED_PRIZES_PER_DRAW_FOR_*_TIERS` will include canary prizes.
2. `_estimateNumberOfTiersUsingPrizeCountPerDraw` now *doubles* the given prize count before comparing against the `ESTIMATED` prize values. This is because each tier is approx 4x the previous, so by only doubling the prize count we create a safe buffer.

[dirk_y \(warden\)](#) commented:

So just to confirm the intended behaviour of the suggested change, you want the number of tiers to increase if >50% of the estimated prizes in a tier are claimed?

I agree that sounds safe to me. It's not possible to scale up by more than 1 tier at a time this way.



M-18 Not fully mitigated: `_redeem` method doesn't handle the case where the withdrawal from the underlying yield vault realises losses

Submitted by [dirk_y](#)



Original Issue

[M-18: Exchange rate change in case of Lossy Strategy will cause the Vault to under-collateralized for generic ERC4626 Yield Vaults](#)



Comments

The original implementation didn't handle lossy withdrawals from underlying ERC-4626 yield vaults properly. During withdrawals from the V5 vault, the calculation of amount to withdraw/shares to burn is based on a `maxWithdraw` view function in the underlying yield vault. However, when the actual withdraw is performed from the yield vault, it is possible for the vault to apply losses. Based on the original implementation, this could cause the V5 vault to pay more shares (of the underlying yield vault) than intended, effectively diluting other vault depositors and causing losses for the last users who withdraw.



Mitigation

The updated implementation in the linked diff is further updated based on the changes for H-09. The key changes related to the original issue is that now V5 vault shares can be converted into the underlying yield vault shares through methods like `_convertAssetsToYVAssets` (for asset conversions in this instance) to ensure that withdrawals burn the correct number of V5 vault shares in exchange for the underlying yield vault shares when the vault is in an under-collateralised state.

However, technically the updated implementation doesn't solve the original issue because the `_redeem` method still doesn't handle the case where the withdrawal from the underlying yield vault realises losses. The vault is still considered collateralised based on the underlying `maxWithdraw` call, so it is possible that withdrawing a given number of assets actually burns more of the underlying yield vault shares owned by the V5 vault than intended; this still impacts those that withdraw/redeem later from the V5 vault.



Conclusion

Although the mitigation is technically unsuccessful, I personally agree with [this comment](#) from the sponsor on the original issue.

Although I do disagree with: “When withdrawing, people will only be able to withdraw their shares of deposits which indeed socializes the loss.”

The losses only impact those that try to withdraw last from the V5 vault since there aren't enough of the underlying yield vault shares left to redeem.

To summarise, I think this is quite a specific use case that either needs to be explicitly supported, or the use of underlying lossy strategies should be discouraged. My personal opinion is that despite the mitigation not being resolved, I wouldn't suggest any further changes; just wanted to be explicit that the original mitigation technically isn't resolved. I would suggest that vaults employing underlying lossy strategies should be discouraged instead.



Assessed type

ERC4626



M-22 Not fully mitigated: deposits could revert if there is a loss of precision of 1 wei in the underlying yield vault

Submitted by [dirk_y](#), also found by [OxStalin](#)



Original Issue

[M-22: Loss of precision leads to under-collateralized](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-vault/blob/main/src/Vault.sol#L1183-L1184>



Comments

The underlying yield vaults used by the V5 vaults usually round down shares received when depositing. As a result, if the Vault deposits to an underlying yield vault that has already issued shares, it is possible that a deposit could be rounded down to a lower number of shares. As a result, user deposits could fail because the vault will be considered not-collateralised due to the 1 wei difference in expected withdrawable assets from the yield vault.



Mitigation

The updated implementation now compares the withdrawable assets after a deposit with the expected withdrawal amount based on the size of the deposit and reverts if the former is less than the latter. This ensures that the vault isn't inadvertently becoming undercollateralised during a deposit.



Mitigation not resolved

The new implementation doesn't necessarily solve the original issue because it just reverts if the updated withdrawable assets from the underlying yield vault is less than the assets we've just deposited; yes, it prevents the vault from being under-collateralised, but now deposits could revert if there is a loss of precision of 1 wei in the underlying yield vault. If we wanted to protect against a rounding issue we should add in a 1 wei tolerance to the updated deposit computation. Even though the model OZ implementation rounds up for conversion in one direction and then rounds down for for conversion in the other direction, we cannot guarantee that all underlying yield vaults will do this. Thus, we should still have a small wei buffer to account for any

rounding issues; otherwise, we might inadvertently end up bricking deposits for users in certain conditions where the deposit amount paired with the underlying vault balance leads to a rounding discrepancy.



Suggestion

In order to prevent a user from deliberately depositing assets that will result in a loss of precision in the underlying yield vault (and therefore, lead to undercollateralisation with the first deposit), I'd recommend allowing the V5 vault owner to specify a minimum deposit value that is then enforced in the deposit and mint methods. Any further loss of precision should be guarded against as suggested in the original report.

Below is a suggested diff:

```
diff --git a/src/Vault.sol b/src/Vault.sol
index 4da1d00..b784477 100644
--- a/src/Vault.sol
+++ b/src/Vault.sol
@@ -1180,7 +1180,7 @@ contract Vault is IERC4626, ERC20Permit, IYVault {
    uint256 _expectedWithdrawableAssets = _withdrawableAssets +
    uint256 _withdrawableAssetsAfter = _totalAssets();

-    if (_withdrawableAssetsAfter < _expectedWithdrawableAssets)
+    if (_withdrawableAssetsAfter + 1 < _expectedWithdrawableAssets)
        revert YVWithdrawableAssetsLTExpected(_withdrawableAssetsAfter,
        _expectedWithdrawableAssets);

    _mint(_receiver, _assets);
```



Assessed type

Math

PierrickGT (PoolTogether) commented:

The suggestion proposed is not really fixing the issue though, no? If I deposit 10000000000 but this deposit is only worth 9999999999, when I will withdraw, I will only receive back 9999999999 ?

So the protocol would not be no loss anymore.

[dirk_y \(warden\) commented:](#)

That's true, the "no loss" claim then technically isn't guaranteed, although a 1 wei loss is ~0 in practically every token, so I'm not sure this should hold too much weight.

The proposed suggestion was more around ensuring deposits wouldn't be reverted if the amount happened to cause a rounding issue based on the balance of the underlying vault. Since you don't have control of the underlying yield vault, you have to either revert on loss of precision (which you're doing now) or absorb the 1 wei loss; I'd argue the former has a greater impact on the user since the gas used is worth more than 1 wei of a token.

[asselstine \(PoolTogether\) commented:](#)

Our mitigation is that we will declare that vault compatibility needs to be checked before integrating, and that yield vaults that round down on deposit are problematic.

*Note: for full discussion, see [here](#).



Claiming prizes will be bricked if prize periods are not aligned with twab periods

Submitted by [dirk_y](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/main/src/TwabController.sol#L281-L299>

<https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/main/src/libraries/TwabLib.sol#L244-L251>

<https://github.com/GenerationSoftware/pt-v5-twab-controller/blob/main/src/libraries/TwabLib.sol#L650-L658>



Comments

The previous implementation allowed a malicious user to keep updating their balances, provided if the previous observation fell within the same period. As such, if a

draw ends part of the way through a period, the user would be able to manipulate their average balance for period and therefore, increase their odds of winning in that draw despite the draw already being closed.

Note: I have arbitrarily chosen M-13 for this report rather than M-03 since both share the same PR for their respective mitigations.



Mitigation

The updated implementation overlaps with M-03 and ensures that twab queries are aligned on twab period boundaries. It also ensures that any Twab queries have an end time that is finalised; i.e. the period that is being queried for has ended. This ensures that a user can't manipulate their Twab balance between the time that a draw period ends and a Twab period ends. The original issue has been resolved, but a new issue has been introduced.



New issue

Claiming prizes is bricked if prize periods are not aligned with twab periods



Proof Of Concept

Although the updated logic solves the original issue about ensuring that the observations are safe (i.e. finalised) and observations can't be manipulated, the change has introduced a new issue where prize claiming is bricked if the twab periods are not completely aligned or are not significantly shorter than the prize pool periods (claiming is still bricked, but for a shorter duration). This was not the case before and in fact the only requirement was that "It is imperative that PoolTogether uses periods that are smaller than a single draw". With the change, this is now only partially true. Either the Twab period needs to be significantly smaller than a prize draw period, or alternatively the period needs to be exactly aligned with the draw period. For example, if a draw is 1 day but the period for the accumulator is 20 hours, it is possible that there will be a 20 hour period during which any claims will revert due to the end time not yet being finalised. This could result in two possible issues: either prizes will go unclaimed altogether, or more fees are paid to claimers since the price of the auction is increasing during this bricked period. Claiming prizes will be bricked up to the maximum period length of the accumulator, assuming the twab period started 1 second before the draw period ended and the twab period was the same length as the draw period.

To illustrate this issue further, let's explore the updated implementation. When prizes are claimed, the winning odds are calculated, which includes a call to `getTwabBetween` in the `TwabController.sol` contract:

```
function getTwabBetween(
    address vault,
    address user,
    uint32 startTime,
    uint32 endTime
) external view returns (uint256) {
    TwabLib.Account storage _account = userObservations[vault][user];
    // We snap the timestamps to the period end on or after the
    // if two users update during a period, then the total supply
    return
        TwabLib.getTwabBetween(
            PERIOD_LENGTH,
            PERIOD_OFFSET,
            _account.observations,
            _account.details,
            _periodEndOnOrAfter(startTime),
            _periodEndOnOrAfter(endTime)
        );
}
```

The start and end times passed as arguments corresponds to the start and end times of the prize tier draw frequency (i.e. daily for the highest prize tier). These times are then aligned to Twab period boundaries with the underlying `_periodEndOnOrAfter` call.

The key point now is that the underlying `getTwabBetween` call requires the end time to be finalised:

```
function getTwabBetween(
    uint32 PERIOD_LENGTH,
    uint32 PERIOD_OFFSET,
    ObservationLib.Observation[MAX_CARDINALITY] storage _observations,
    AccountDetails memory _accountDetails,
    uint32 _startTime,
    uint32 _endTime
) internal view requireFinalized(PERIOD_LENGTH, PERIOD_OFFSET,
```

Where `requireFinalized` looks like:

```
modifier requireFinalized(uint32 PERIOD_LENGTH, uint32 PERIOD_OFFSET) {
    // The current period can still be changed; so the start of the current period is
    uint32 overwritePeriodStartTime = _currentOverwritePeriodStartTime;
    // timestamp == overwritePeriodStartTime doesn't matter, because if
    if (_timestamp > overwritePeriodStartTime) {
        revert TimestampNotFinalized(_timestamp, overwritePeriodStartTime);
    }
    _;
}
```

For the sake of argument, let's say that a claimer is trying to claim a prize for the highest tier that is drawn daily, and the twab period ends 1 hour before the draw period. Now, when a claimer tries to claim the prize, the initial `getTwabBetween` call will align the start and end times to twab period boundaries. So the start time will align to the end of the last period, and the end time will align to the end of the current period. However, the current period hasn't finalised yet! The current period finalises 23 hours after the draw has completed, which means that any prize claims for the tier will revert. During this time, the auction is still running, so the fee is perpetually increasing. After the 23 hours have elapsed, the claimers now only have 1 hour to claim the prizes, leading to either of the two scenarios mentioned above.



Recommendation

I recommend aligning the shortest prize period to the twab period and enforcing this in the constructor of the Prize Pool contract.



Assessed type

Timing

[asselstine \(PoolTogether\) confirmed](#)

[OxStalin \(warden\) commented:](#)

This is a good finding. Although, it looks like there needs to be a couple of onchain conditions for the two scenarios to occur. I think this should be classified as a medium sev because of the number of onchain requirements for the issue to occur.

[dirk_y \(warden\) commented:](#)

There is only one condition that needs to be true: twab periods are not aligned with prize pool periods. And in fact, this behaviour would have been guaranteed based on the [official documentation](#).

[Picodes \(judge\) commented:](#)

On first analysis, I do agree that the external conditions are minimal so it fits High severity.



Vault will stop participating in draws in the case they deposited maximum assets to the underlying vault

Submitted by [rvierdiiev](#)



Lines of code

<https://github.com/GenerationSoftware/pt-v5-vault/blob/main/src/Vault.sol#L1399>



Proof of Concept

Vault contract [has a `maxMint` function](#). This function first checks allowed amount to mint in the `PtVault` and then also [checks the amount allowed in underlying vault](#). It is very likely that this value will be less than what is in `PtVault`, as some vaults can have restriction for 1 staker.

This means, that once such amount of shares is minted, then [it's not allowed to mint anymore](#), so each deposit/mint will revert.

In order to contribute earned yield to the prize pool, the liquidation pair should call the `liquidate` function. It's the only way to send earned yield to the prize pool.

The next process is `liquidator` will provide POOL tokens on behalf of `Vault` and `Vault` [should mint the corresponding amount of shares](#) for `liquidator`. In this case, the call will be done after the max amount is minted. Then, it will revert, which means that it will be not possible to send the earned yield to the pool and earn prizes.

In this moment, the main purpose of vault stopped working; users earned yield to participate in prize distributions, but they can't do that.

Of course, when someone will withdraw, it will be possible to liquidate again, but that will lead to bad a user experience and also someone can DoS pool by depositing again to not allow send contributions to the prize pool.

Another thing that is related to this issue, is that in the case `maxMint` was reached, then `mintYieldFee` can't be called as well, as it also mints shares.

Tools Used

VsCode

Recommended Mitigation Steps

Maybe it will be best to not check for `maxMint` for liquidations and fee minting.

Assessed type

Error

[asselstine \(PoolTogether\) confirmed](#)

[PierrickGT \(PoolTogether\) commented:](#)

| Interesting reasoning.

| Vault shares must be backed 1:1 by underlying assets deposited into the Yield Vault. If we mint more Vault shares than underlying assets withdrawable, we will enter in an under-collateralized state.

| That being said, the yield that accumulated in the Yield Vault is liquidated in exchange of Prize Tokens by minting more Vault shares. So even if `maxMint` is reached, we should be able to liquidate this yield.

| Fixed in this PR: <https://github.com/GenerationSoftware/pt-v5-vault/pull/43>.

Note: for full discussion, see [here](#).



Disclosures

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

C4 Audits incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Audit 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

An open organization | [Twitter](#) | [Discord](#) | [GitHub](#) | [Medium](#) | [Newsletter](#) | [Media kit](#) | [Careers](#) |
[code4rena.eth](#)