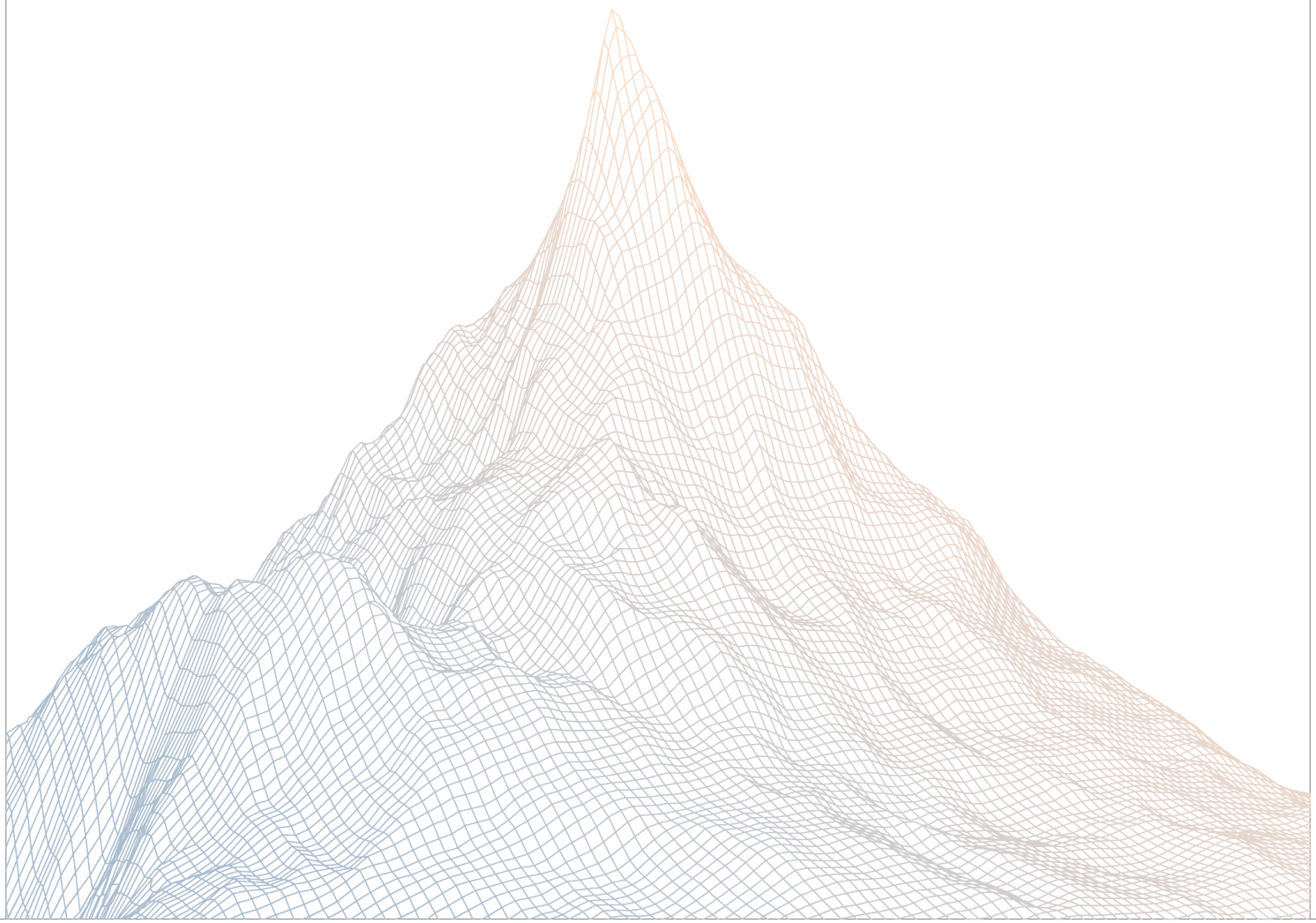
Zenith

Pooltogether

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: February 19th to February 21th, 2025

AUDITED BY: fedebianu
spicymeatball

Contents

1 Introduction

1.1 About Zenith

Zenith is an offering by Code4rena that provides consultative audits from the very best security researchers in the space. We focus on crafting a tailored security team specifically for the needs of your codebase.

Learn more about us at https://code4rena.com/zenith.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Pooltogether

PoolTogether is a prize savings protocol, enabling you to win by saving.

1. Deposit USDC for a chance to win
2. Participate in daily prize draws
3. Withdraw your deposit any time - even if you don't win!

Every dollar you deposit gives you a chance to win prizes. The more you save, the higher your odds!

PoolTogether is one of the first and most widely used DeFi (Decentralized Finance) applications and has been live for over four years. Since its inception, the protocol distributed over $10 million in prizes to depositors.

2.2 Scope

The engagement involved a review of the following targets:

Target	pt-v5-prize-pool-twab-rewards
Repository	https://github.com/GenerationSoftware/pt-v5-prize-pool-twab-rewards
Commit Hash	`63d7c8d2a3740d38dd56a2a07d07b6097b475dad`
Files	`PrizePoolTwabRewards.sol`

2.3 Audit Timeline

February 19, 2025	Audit start
February 21, 2025	Audit end
March 10, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	0
Medium Risk	1
Low Risk	2
Informational	2
Total Issues	**5**

3

Findings Summary

ID	Description	Status
M-1	PrizePoolTwabRewards.sol contract doen't support rebasing tokens	Resolved
L-1	Missing epoch validation in getVaultRewardAmount()	Resolved
L-2	Strict balance checks prevents using tokens with rounding issues like stETH	Resolved
I-1	Inefficient error handling and misleading NatSpec in reward claiming	Resolved
I-2	Perform divisions after multiplications	Resolved

4

Findings

4.1 Medium Risk

A total of 1 medium risk findings were identified.

[M-1] `PrizePoolTwabRewards.sol` contract doen't support rebasing tokens

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- PrizePoolTwabRewards.sol

Description:

The contract fails to properly handle rebasing tokens (like stETH) when used as reward tokens, leading to potential fund locking or failed claims. The issue stems from the contract tracking rewards through a fixed `rewardsUnclaimed` variable that doesn't account for rebasing effects.

Two critical scenarios emerge:

1. Positive rebase:

- Alice creates a promotion with 1000 `stETH` as rewards
- The contract records `rewardsUnclaimed = 1000`
- Over time, due to `stETH` rebasing, the actual balance grows to 1050 `stETH`
- Bob and other users claim 500 `stETH` in total
- The contract updates `rewardsUnclaimed = 500`
- Alice ends the promotion and receives only 500 `stETH`
- 50 stETH remain permanently locked in the contract

2. Negative rebase (rare but possible):

- Alice creates a promotion with 1000 `stETH`
- Due to negative rebase, the balance decreases to 950 `stETH` due to slashing events
- Bob and others claim 900 `stETH`
- Charlie tries to claim the last 100 `stETH` but the transaction reverts

- Alternatively, when Alice tries to end the promotion, it reverts due to insufficient balance

Recommendations:

Add logic to handle rebasing tokens or explicitly write in the documentation that the contract doesn't support rebasing tokens as per fee-on-trasfer tokens.

Pooltogether: Resolved with PR-6

Zenith: Verified.

4.2 Low Risk

A total of 2 low risk findings were identified.

[L-1] Missing epoch validation in `getVaultRewardAmount()`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- PrizePoolTwabRewards.sol

Description:

The function `getVaultRewardAmount()` doesn't validate if the epoch is over before calculating rewards, unlike `_calculateRewardAmount()` used in `claimRewards()`.

Recommendations:

Add the same validation in `getVaultRewardAmount()`.

Pooltogether: Resolved with PR-5

Zenith: Verified.

[L-2] Strict balance checks prevents using tokens with rounding issues like `stETH`

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- PrizePoolTwabRewards.sol

Description:

The `PrizePoolTwabRewards` contract contains a vulnerability in its balance check during promotion creation or extension when used with tokens like Lido `stETH`, that has a know rounding down problem as stated in their docs. Specifically, in `createPromotion()`, the contract verifies received tokens with a strict balance comparison:

```
if (_afterBalance < _beforeBalance + unclaimedRewards) {
    revert TokensReceivedLessThanExpected(_afterBalance - _beforeBalance,
    unclaimedRewards);
}
```

This check fails to account for the documented rounding issue in `stETH` where a 1-2 wei difference can occur during transfer operations. With the current implementation, when attempting to create a promotion using `stETH` as the reward token, the transaction will consistently fail due to the strict equality check.

Recommendations:

Modify the balance check to incorporate a small tolerance (e.g. 10 wei) that accounts for this kind of rounding issues.

```
// Add a tolerance to handle stETH corner case
if (_afterBalance + TOLERANCE < _beforeBalance + unclaimedRewards) {
    revert TokensReceivedLessThanExpected(_afterBalance - _beforeBalance,
    unclaimedRewards);
}
```

Pooltogether: Resolved with PR-9

Zenith: Verified.

4.3 Informational

A total of 2 informational findings were identified.

[I-1] Inefficient error handling and misleading NatSpec in reward claiming

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- PrizePoolTwabRewards.sol

Description:

`_claimRewards()` contains inefficient error handling and misleading NatSpec:

1. The function reverts with `RewardsAlreadyClaimed` after checking `_isClaimedEpoch(_userClaimedEpochs, uint8(index))`, which is inefficient as it could simply `continue` to skip already claimed epochs
2. The parameter `_epochClaimFlags` is documented as "Word representing which epochs were claimed" but actually represents which epochs should be claimed in this transaction
3. The function `_isClaimedEpoch()` is used for two different semantic purposes, checking which epochs to claim and checking which epochs have already been claimed by the user

Recommendations:

1. Remove the `if` block and insert this code in `claimRewards()` before calling `_claimRewards()`:

```
bytes32 _userClaimedEpochs = claimedEpochs[_promotionId][_vault][_user];
// exclude epochs already claimed by the user
_epochClaimFlags = _epochClaimFlags & ~_userClaimedEpochs;
```

2. Update the NatSpec documentation to clarify the parameter's purpose:

```
* @param _epochClaimFlags Word representing which epochs to claim
```

3. Rename the `_isClaimedEpoch()` function to something more generic like `_isBitSet()` to reflect its actual function, since it's used for different semantic purposes:

```
function _isBitSet(
  bytes32 _bitMap,
  uint8 _bitIndex
) internal pure returns (bool) {
  return (uint256(_bitMap) >> _bitIndex) & uint256(1) == 1;
}
```

Or add another function with the correct semantic `_isClaimingEpoch()`:

```
function _isClaimingEpoch(
  bytes32 _claimingEpochs,
  uint8 _epochId
) internal pure returns (bool) {
  return (uint256(_claimingEpochs) >> _epochId) & uint256(1) == 1;
}
```

Pooltogether: Resolved with PR-7

Zenith: Verified

[I-2] Perform divisions after multiplications

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- PrizePoolTwabRewards.sol

Description:

`_calculateRewardAmount()` performs a division before a multiplication:

```
uint256 numerator = ((_promotion.tokensPerEpoch * _userAverage)
    / uint256(vaultEpochCache.totalSupply))
    * uint256(vaultEpochCache.contributed);
uint256 denominator = (uint256(epochCache.totalContributed));
return numerator / denominator;
```

Even if, in this particular scenario, this will not cause any issue it is best practice to perform divisions after multiplications.

Recommendations:

As a best practice, consider reordering the arithmetic operations to perform all multiplications before divisions:

```
uint256 numerator = _promotion.tokensPerEpoch * _userAverage
    * uint256(vaultEpochCache.contributed);
uint256 denominator = uint256(vaultEpochCache.totalSupply)
    * uint256(epochCache.totalContributed);
return numerator / denominator;
```

Pooltogether: Resolved with PR-8

Zenith: Verified.