PoolTogether V5: Part Deux Findings & Analysis Report

2023-09-07

Table of contents

Overview

About C4

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

A C4 audit 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 V5: Part Deux smart contract system written in Solidity. The audit took place between August 2—August 7 2023.

Wardens

45 Wardens contributed reports to the PoolTogether audit:

1. dirk_y
2. 3agle
3. nadin
4. 0xmystery
5. MohammedRizwan
6. bin2chen
7. 0xStalin
8. Angry_Mustache_Man
9. hals
10. 14si2o_Flint
11. Rolezn
12. 0xbepresent
13. ptsanev
14. 0xSmartContract
15. MatricksDeCoder
16. Giorgio
17. SanketKogekar
18. cartlex_
19. seerether
20. rvierdiiev

21. hunter_w3b
22. K42
23. Aymen0909
24. piyushshukla
25. cholakov
26. DedOhWale
27. josephdara
28. trachev
29. Arz
30. D_Auditor
31. shirochan
32. Jorgect
33. T1MOH
34. Raihan
35. JCK
36. petrichor
37. SY_S
38. dharma09
39. SAQ
40. wahedtalash77
41. 0xta
42. 0xhex
43. ReyAdmirado
44. shamsulhaq123
45. Rageur

This audit was judged by hickuphh3.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 13 unique vulnerabilities. Of these vulnerabilities, 2 received a risk rating in the category of HIGH severity and 11 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 6 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 15 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 V5: Part Deux repository](), and is composed of 17 smart contracts written in the Solidity programming language and includes 1,001 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 (2)

[H-01] Too many rewards are distributed when a draw is closed

Submitted by dirk_y

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngRelayAuction.sol#L178-L184
https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngRelayAuction.sol#L154-L157
https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/26557afa439934afc080eca6165fe3ce5d4b63cd/src/PrizePool.sol#L366
https://github.com/GenerationSoftware/pt-v5-prize-pool/blob/26557afa439934afc080eca6165fe3ce5d4b63cd/src/abstract/TieredLiquidityDistributor.sol#L374

A relayer completes a prize pool draw by calling `rngComplete` in `RngRelayAuction.sol`. This method closes the prize pool draw with the relayed random number and distributes the rewards to the RNG auction recipient and the RNG relay auction recipient. These rewards are calculated based on a fraction of the prize pool reserve rather than an actual value.

However, the current reward calculation mistakenly includes an extra `reserveForOpenDraw` amount just after the draw has been closed. Therefore the fraction over which the rewards are being calculated includes tokens that have not been added to the reserve and will actually only be added to the reserve when the next draw is finalised. As a result, the reward recipients are rewarded too many tokens.

Proof of Concept

Before deciding whether or not to relay an auction result, a bot can call `computeRewards` to calculate how many rewards they'll be getting based on the size of the reserve, the state of the auction and the reward fraction of the RNG auction recipient:

```
    function computeRewards(AuctionResult[] calldata __auctionResu
      uint256 totalReserve = prizePool.reserve() + prizePool.reser
      return _computeRewards(__auctionResults, totalReserve);
    }
```

Here, the total reserve is calculated as the sum of the current reserve and and amount of new tokens that will be added to the reserve once the currently open draw is closed. This method is correct and correctly calculates how many rewards should be distributed when a draw is closed.

A bot can choose to close the draw by calling `rngComplete` (via a relayer), at which point the rewards are calculated and distributed. Below is the interesting part of this method:

```
      uint32 drawId = prizePool.closeDraw(_randomNumber);

      uint256 futureReserve = prizePool.reserve() + prizePool.rese
      uint256[] memory _rewards = RewardLib.rewards(auctionResults
```

As you can see, the draw is first closed and then the future reserve is used to calculate the rewards that should be distributed. However, when `closeDraw` is called on the pool, the `reserveForOpenDraw` for the previously open draw is added to the existing reserves. So `reserve()` is now equal to the `totalReserve` value in the earlier call to `computeRewards`. By including `reserveForOpenDraw()` when computing the actual reward to be distributed we've accidentally counted the tokens that are only going to be added in when the next draw is closed. So now the rewards distribution calculation includes the pending reserves for 2 draws rather than 1.

Recommended Mitigation Steps

When distributing rewards in the call to `rngComplete`, the rewards should not be calculated with the new value of `reserveForOpenDraw` because the previous `reserveForOpenDraw` value has already been added to the reserves when `closeDraw` is called on the prize pool. Below is a suggested diff:

```
  diff --git a/src/RngRelayAuction.sol b/src/RngRelayAuction.sol
```

```
index 8085169..cf3c210 100644
--- a/src/RngRelayAuction.sol
+++ b/src/RngRelayAuction.sol
@@ -153,8 +153,8 @@ contract RngRelayAuction is IRngAuctionRelay

     uint32 drawId = prizePool.closeDraw(_randomNumber);

-    uint256 futureReserve = prizePool.reserve() + prizePool.res
-    uint256[] memory _rewards = RewardLib.rewards(auctionResult
+    uint256 reserve = prizePool.reserve();
+    uint256[] memory _rewards = RewardLib.rewards(auctionResult

     emit RngSequenceCompleted(
       _sequenceId,
```

Assessed type

Math

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

> Nice catch!

[hickuphh3 (judge) commented]():

> Great find!

[H-02] `rngComplete` function should only be called by `rngAuctionRelayer`

Submitted by [Aymen0909](), *also found by* [josephdara](), [trachev](), [Arz](), *seerether (*[1](), [2]()*),* [D_Auditor](), [shirochan](), [dirk_y](), [0xbepresent](), [Jorgect](), [bin2chen](), [0xStalin](), [ptsanev](), [T1MOH](), *and* [rvierdiiev]()

The `rngComplete` function is supposed to be called by the relayer to complete the Rng relay auction and send auction rewards to the recipient, but because the function doesn't have any access control it can be called by anyone, an attacker can call the function before the relayer and give a different `_rewardRecipient` and thus he can collect all the rewards and the true auction reward recipient will not get any.

Proof of Concept

The issue occurs in the `rngComplete` function below:

```
function rngComplete(
    uint256 _randomNumber,
    uint256 _rngCompletedAt,
    address _rewardRecipient, // @audit can set any address
    uint32 _sequenceId,
    AuctionResult calldata _rngAuctionResult
) external returns (bytes32) {
    // @audit should only be callable by rngAuctionRelayer
    if (_sequenceHasCompleted(_sequenceId)) revert SequenceAlrea
    uint64 _auctionElapsedSeconds = uint64(
        block.timestamp < _rngCompletedAt ? 0 : block.timestamp
    );
    if (_auctionElapsedSeconds > (_auctionDurationSeconds - 1))
    // Calculate the reward fraction and set the draw auction re
    UD2x18 rewardFraction = _fractionalReward(_auctionElapsedSec
    _auctionResults.rewardFraction = rewardFraction;
    _auctionResults.recipient = _rewardRecipient;
    _lastSequenceId = _sequenceId;

    AuctionResult[] memory auctionResults = new AuctionResult[](
    auctionResults[0] = _rngAuctionResult;
    auctionResults[1] = AuctionResult({
        rewardFraction: rewardFraction,
        recipient: _rewardRecipient
    });

    uint32 drawId = prizePool.closeDraw(_randomNumber);

    uint256 futureReserve = prizePool.reserve() + prizePool.rese
    uint256[] memory _rewards = RewardLib.rewards(auctionResults

    emit RngSequenceCompleted(
        _sequenceId,
        drawId,
        _rewardRecipient,
        _auctionElapsedSeconds,
        rewardFraction
    );

    for (uint8 i = 0; i < _rewards.length; i++) {
        uint104 _reward = uint104(_rewards[i]);
```

```
            if (_reward > 0) {
                prizePool.withdrawReserve(auctionResults[i].recipier
                emit AuctionRewardDistributed(_sequenceId, auctionRe
            }
        }

        return bytes32(uint(drawId));
    }
```

As we can see the function does not have any access control (modifier or check on the msg.sender), so any user can call it and you can also notice that the `_rewardRecipient` (the address that receives the rewards) is given as argument to the function and there is no check to verify that it is the correct auction reward receiver.

Hence an attacker can call the function before the relayer does, he can thus complete the auction and give another address for `_rewardRecipient` which will receive all the rewards.

The result is in the end that the true auction reward recipient will get his reward stolen by other users.

Recommended Mitigation Steps

Add a check in the `rngComplete` function to make sure that only the relayer can call it, the function can be modified as follows:

```
    function rngComplete(
        uint256 _randomNumber,
        uint256 _rngCompletedAt,
        address _rewardRecipient,
        uint32 _sequenceId,
        AuctionResult calldata _rngAuctionResult
    ) external returns (bytes32) {
        // @audit only called by rngAuctionRelayer
        if (msg.sender != rngAuctionRelayer) revert NotRelayer();
        ...
    }
```

Assessed type

Access Control

[asselstine (PoolTogether) confirmed](#)

Medium Risk Findings (11)

[M-01] `RemoteOwner` circular dependency at deployment time

Submitted by [dirk_y](#)

https://github.com/GenerationSoftware/remote-owner/blob/9c093dbd36c1f18ab7083549d10ac601d91630df/src/RemoteOwner.sol#L58
https://github.com/GenerationSoftware/remote-owner/blob/9c093dbd36c1f18ab7083549d10ac601d91630df/src/RemoteOwner.sol#L120
https://github.com/GenerationSoftware/remote-owner/blob/9c093dbd36c1f18ab7083549d10ac601d91630df/src/RemoteOwner.sol#L96-L99
https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngAuctionRelayerRemoteOwner.sol#L47
https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngAuctionRelayerRemoteOwner.sol#L64

The `RemoteOwner.sol` contract has a security measure that ensures the sender from the remote/origin chain was the origin chain owner (i.e. a `RngAuctionRelayerRemoteOwner.sol` deployment), and this address is set at deployment time in the constructor. The `RngAuctionRelayerRemoteOwner` contract also has a security measure to ensure that messages are only dispatched across chain to the `RemoteOwner` contract deployed in the destination chain, and this address is set at deployment time in the constructor.

Clearly there is a circular dependency here that means the deployment phase will fail. There is a `setOriginChainOwner` method on the `RemoteOwner` contract, however this can only be called by the address on the origin chain specified in the constructor. This method is never called from the origin chain either. In summary, the circular dependency prevents the contracts from being deployed and ever initialised properly.

It is possible that there is an intermediary `__originChainOwner` used in the constructor when deploying `RemoteOwner`, but since I couldn't find any deployment scripts to verify this I have assumed that this is an unintended bug. The severity of this report depends on whether or not this was intended.

Proof of Concept

In the `RemoteOwner.sol` contract, the origin chain owner is set in the constructor:

```
constructor(
  uint256 originChainId_,
  address executor_,
  address __originChainOwner
) ExecutorAware(executor_) {
  if (originChainId_ == 0) revert OriginChainIdZero();
  _originChainId = originChainId_;
  _setOriginChainOwner(__originChainOwner);
}
```

Any calls to the `RemoteOwner` contract are protected by the `_checkSender` view:

```
function _checkSender() internal view {
  if (!isTrustedExecutor(msg.sender)) revert LocalSenderNotExe
  if (_fromChainId() != _originChainId) revert OriginChainIdUr
  if (_msgSender() != address(_originChainOwner)) revert Origi
}
```

Now, if we have a look at the `RngAuctionRelayerRemoteOwner.sol` contract, we can see that the remote owner address is also specified in the constructor:

```
        constructor(
            RngAuction _rngAuction,
            ISingleMessageDispatcher _messageDispatcher,
            RemoteOwner _remoteOwner,
            uint256 _toChainId
        ) RngAuctionRelayer(_rngAuction) {
            messageDispatcher = _messageDispatcher;
            account = _remoteOwner;
            toChainId = _toChainId;
        }
```

This `account` address is now hard-coded and used with any calls to `relay`:

```
    function relay(
            IRngAuctionRelayListener _remoteRngAuctionRelayListener,
            address rewardRecipient
        ) external returns (bytes32) {
            bytes memory listenerCalldata = encodeCalldata(rewardRec
            bytes32 messageId = messageDispatcher.dispatchMessage(
                toChainId,
                address(account),
                RemoteOwnerCallEncoder.encodeCalldata(address(_remot
            );
            emit RelayedToDispatcher(rewardRecipient, messageId);
            return messageId;
        }
```

There is a circular dependency here due to the reliance on specifying the relevant addresses in the constructor.

Recommended Mitigation Steps

To remove the circular dependency and reliance on a very specific deployment pipeline that requires a specific call from a remote chain address, I would make the following change to the `RemoteOwner` contract:

```
    diff --git a/src/RemoteOwner.sol b/src/RemoteOwner.sol
    index 7c1de6d..a6cb8f1 100644
    --- a/src/RemoteOwner.sol
    +++ b/src/RemoteOwner.sol
```

```
@@ -55,7 +55,6 @@ contract RemoteOwner is ExecutorAware {
   ) ExecutorAware(executor_) {
     if (originChainId_ == 0) revert OriginChainIdZero();
     _originChainId = originChainId_;
-    _setOriginChainOwner(__originChainOwner);
   }

   /* ============ External Functions ============ */
@@ -94,7 +93,7 @@ contract RemoteOwner is ExecutorAware {
    *      If the transaction get front-run at deployment, we ca
    */
   function setOriginChainOwner(address _newOriginChainOwner) ex
-    _checkSender();
+    require(_originChainOwner == address(0), "Already initiali
     _setOriginChainOwner(_newOriginChainOwner);
   }
```

However I can understand how the current deployment pipeline functionality would make it harder to frontrun `setOriginChainOwner` if this was done deliberately, so alternatively you could keep the functionality the same but just provide better comments.

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

> Interesting. We haven't deployed the `RemoteOwner` yet so we didn't run into this issue.

> It looks like the simplest solution is to pass the `RemoteOwner` as an argument to the `RngAuctionRelayerRemoteOwner#relay` call.

> This also makes deploying to new L2s more convenient, because we won't need to redeploy a `RngAuctionRelayerRemoteOwner` contract.

[M-02] PRBMATH `SD59x18.exp()` reverts on hugely negative numbers.

Submitted by [nadin](#)

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L34-L36
https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L64
https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L85-L87
https://github.com/PaulRBerg/prb-math/blob/5959ef59f906d689c2472ed08797872a1cc00644/src/sd59x18/Math.sol#L168-L181

`ContinuousGDA.sol` inherits a version of `PRB Math` that contains a vulnerability in the `SD59x18.exp()` function, which can be reverted on hugely negative numbers. `SD59x18.exp()` is used for calculations in `ContinuousGDA.sol#purchasePrice()`, `ContinuousGDA.sol#purchaseAmount()` and `ContinuousGDA.sol#computeK()`. Recently, the creators of the `PRBMath` have acknowledged this situation. Here is the corresponding link. This issue should be proactively corrected by `PoolTogether` to avoid unexpected results that corrupt the protocol's computation flow.

Proof of Concept

There are 05 instances of this issue: see here

```
File: ContinuousGDA.sol
34:     topE = topE.exp().sub(ONE);
36:     bottomE = bottomE.exp();
64:     SD59x18 exp = _decayConstant.mul(_timeSinceLastAuctionSta
85:     SD59x18 eValue = exponent.exp();
87:     SD59x18 denominator = (_decayConstant.mul(_purchaseAmount
```

Proof of the bug acknowledgment by the creator of the PRBMath.

`SD59x18.exp()` correctly returns 0 for inputs less than (roughly) -41.45e18, however it starts to throw `PRBMath_SD59x18_Exp2_InputTooBig` when the input gets hugely negative. This is because of the unchecked multiplication in `exp()` overflowing into positive values: see here.

```
    function exp(SD59x18 x) pure returns (SD59x18 result) {
        int256 xInt = x.unwrap();

        // This check prevents values greater than 192e18 from being
        if (xInt > uEXP_MAX_INPUT) {
            revert Errors.PRBMath_SD59x18_Exp_InputTooBig(x);
        }

        unchecked {
            // Inline the fixed-point multiplication to save gas.
            int256 doubleUnitProduct = xInt * uLOG2_E;
            result = exp2(wrap(doubleUnitProduct / uUNIT));
        }
    }
```

Tools Used

Manual Review and Proof of the bug acknowledgment by the creator of the PRBMath

Recommended Mitigation Steps

A potential fix would be to compare the input with the smallest (most negative) number that can be safely multiplied by `uLOG2_E`, and return 0 if it's smaller. Alternatively, `exp()` could return 0 for inputs smaller than -41.45e18, which are expected to be truncated to zero by `exp2()` anyway.

Assessed type

Math

asselstine (PoolTogether) confirmed

hickuphh3 (judge) commented:

> Would have been nice to see a concrete example where the input < -41.45e18.

> Given that the pricing formula parameters are determined by the liquidation pair creator, it is a possibility to achieve this condition and have this bug invoked.

[M-03] Missing `deadline` param in `swapExactAmountOut()` allowing outdated slippage and allow pending transaction to be executed unexpectedly

Submitted by SanketKogekar, *also found by* bin2chen, MohammedRizwan, cartlex_, *and* piyushshukla

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationRouter.sol#L63-L80
https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L211-L226

Loss of funds/tokens for the protocol, since block execution is delegated to the block validator without a hard deadline.

Proof of Concept

The function `swapExactAmountOut()` from `LiquidationRouter.sol` and `LiquidationPair.sol` use these methods to swap tokens:

```
source.liquidate(_account, tokenIn, swapAmountIn, tokenOut, _amountOut);
```

and

```
_liquidationPair.swapExactAmountOut(_receiver, _amountOut, _amountInMax);
```

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L211-L226

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationRouter.sol#L63-L80

Both methods make sure to pass slippage (minimum amount out), but miss to provide the deadline which is crucial to avoid unexpected trades/losses for users and protocol.

Without a deadline, the transaction might be left hanging in the mempool and be executed way later than the user wanted.

That could lead to users/protocol getting a worse price, because a validator can just hold onto the transaction. And when it does get around to putting the transaction in a block

One part of this change is that PoS block proposers know ahead of time if they're going to propose the next block. The validators and the entire network know who's up to bat for the current block and the next one.

This means the block proposers are known for at least 6 minutes and 24 seconds and at most 12 minutes and 48 seconds.

Further reading: https://blog.bytes032.xyz/p/why-you-should-stop-using-block-timestamp-as-deadline-in-swaps

Recommended Mitigation Steps

Let users provide a fixed deadline as param, and also never set deadline to `block.timestamp`.

asselstine (PoolTogether) confirmed

hickuphh3 (judge) commented:

> The main argument here is the user will lose out on positive slippage if the exchange rate becomes favourable when the tx is included in a block.

> As to why it's Medium severity: when the value lost is guaranteed to be very large and preventing the mistake has a very low cost, it is reasonable to assign a medium risk rating. The closer the cost/benefit ratio gets to zero, the more likely the issue should be rated QA.

[M-04] Potential Near-Zero Scenarios for `purchasePrice` in the Continuous Gradual Dutch Auction

Submitted by [0xmystery]

[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L211-L226]
[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L294-L319]
[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L16-L44]

The Continuous Gradual Dutch Auction (CGDA) model has potential scenarios where the `purchasePrice` for an amount of tokens could approach near-zero values. This is influenced mainly by two factors: `_emissionRate` and `_timeSinceLastAuctionStart`. If either one or both of these factors (`_emissionRate` specifically more likely) are significantly large, the `purchasePrice` could drastically drop.

This condition could cause undesired economic effects in the auction process. Under this context, participants may acquire tokens (amount of Vault shares) at an extremely low price (very low POOL amount indeed), which could lead to significant chance of winnings.

Proof of Concept

Here is the purchasePrice function within the ContinuousGDA library, which computes the purchase price of tokens based on various parameters.

[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L23-L44]

```
  function purchasePrice(
    SD59x18 _amount,
    SD59x18 _emissionRate,
    SD59x18 _k,
    SD59x18 _decayConstant,
    SD59x18 _timeSinceLastAuctionStart
  ) internal pure returns (SD59x18) {
    if (_amount.unwrap() == 0) {
      return SD59x18.wrap(0);
    }
    SD59x18 topE = _decayConstant.mul(_amount).div(_emissionRate
    topE = topE.exp().sub(ONE);
    SD59x18 bottomE = _decayConstant.mul(_timeSinceLastAuctionSt
    bottomE = bottomE.exp();
    SD59x18 result;
    if (_emissionRate.unwrap() > 1e18) {
      result = _k.div(_emissionRate).mul(topE).div(bottomE);
    } else {
      result = _k.mul(topE.div(_emissionRate.mul(bottomE)));
    }
    return result;
  }
```

One possible scenarios where `purchasePrice` could approach near-zero values is:
_k = 1e18 (the initial price of the CGDA)
_decayConstant = 0.00001 (a very small decay constant)
_amount = 1e18 (the amount of tokens to purchase which has reportedly become a rare commodity)
_emissionRate = 1e20 (a significantly large, but more practical emission rate)
_timeSinceLastAuctionStart = 3600 (equivalent to 1 hour)

The small `_decayConstant` , along with the relatively short `_timeSinceLastAuctionStart` , results in `bottomE` being close to 1. `_emissionRate` is significantly larger than `_amount` , making `topE` also close to 1. As a result, this combination of factors drives the `purchasePrice` towards near-zero values.

Please note that this is only one of many possible scenarios where the purchase price could approach near-zero values. Other combinations of parameter tweaks could potentially also lead to similar or more likely outcomes.

Recommended Mitigation Steps

1. Introduce checks in the `_computeExactAmountIn` function to ensure that `_emissionRate` doesn't exceed a certain practical limit.
2. Introduce checks in function `swapExactAmountOut` that the scaled ratio of `_amountInForPeriod` to `_amountOutForPeriod` should not fall below a certain threshold.

Assessed type

Context

asselstine (PoolTogether) confirmed and commented:

> I'd like to see a more concrete example, as some of these parameters have been chosen arbitrarily. What is the asset:prize exchange rate? What should the value of _k be after being tuned correctly with computeK? This boundary condition feels a little contrived.

> Regardless of the details, I think it's good that it points out the possibility of a zero-value swap.

> We should assert that `swapAmountIn > 0` in the `swapExactAmountOut` function.

hickuphh3 (judge) commented:

> This issue is affected by `#24` : the near-zero condition presented no longer holds true when the formula has been modified.

> Nevertheless, looking at this issue in isolation, the argument is valid: near-zero / zero price scenarios should be blocked.

asselstine (PoolTogether) commented:

> Fixed in this commit.

[M-05] `RngRelayAuction.rngComplete()` DOS attack

Submitted by bin2chen, also found by 0xbepresent and MatricksDeCoder

If the recipient maliciously enters the blacklist of `priceToken` , it may cause `rngComplete()` to fail to execute successfully.

Proof of Concept

The current implementation of `RngRelayAuction.rngComplete()` immediately transfers the `prizeToken` to the `recipient` .

```
    function rngComplete(
      uint256 _randomNumber,
      uint256 _rngCompletedAt,
      address _rewardRecipient,
      uint32 _sequenceId,
      AuctionResult calldata _rngAuctionResult
    ) external returns (bytes32) {
  ...
      for (uint8 i = 0; i < _rewards.length; i++) {
        uint104 _reward = uint104(_rewards[i]);
        if (_reward > 0) {
@>        prizePool.withdrawReserve(auctionResults[i].recipient, _
          emit AuctionRewardDistributed(_sequenceId, auctionResult
        }
      }
```

```
  contract PrizePool is TieredLiquidityDistributor {
    function withdrawReserve(address _to, uint104 _amount) externa
      if (_amount > _reserve) {
        revert InsufficientReserve(_amount, _reserve);
      }
      _reserve -= _amount;
@>    _transfer(_to, _amount);
      emit WithdrawReserve(_to, _amount);
    }

    function _transfer(address _to, uint256 _amount) internal {
      _totalWithdrawn += _amount;
@>    prizeToken.safeTransfer(_to, _amount);
    }
```

There is a risk that if `prizeToken` is a `token` with a blacklisting mechanism, such as : `USDC` .
Then `recipient` can block `rngComplete()` by maliciously entering the `USDC` blacklist.
Since `RngAuctionRelayer` supports `AddressRemapper` , users can more simply specify blacklisted addresses via `remapTo()`

Recommended Mitigation Steps

Add `claims` mechanism, `rngComplete()` only record `cliamable[token][user]+=rewards` .
Users themselves go to `claims` .

Assessed type

Context

asselstine (PoolTogether) confirmed

[M-06] `_computeAvailable()` the calculations are wrong

Submitted by bin2chen, *also found by* Angry_Mustache_Man, dirk_y, *and* Giorgio

`_computeAvailable()` incorrect calculations that result in a return value greater than the current balance, causing methods such as `liquidate` to fail.

Proof of Concept

`VaultBooster._computeAvailable()` used to count the number of `tokens` currently available.
There are two conditions:

1. `accrue` continuously according to time
2. the final cumulative value cannot be greater than the current contract balance

The code is as follows:

```
    function _computeAvailable(IERC20 _tokenOut) internal view ret
      Boost memory boost = _boosts[_tokenOut];
      uint256 deltaTime = block.timestamp - boost.lastAccruedAt;
      uint256 deltaAmount;
      if (deltaTime == 0) {
        return boost.available;
      }
      if (boost.tokensPerSecond > 0) {
        deltaAmount = boost.tokensPerSecond * deltaTime;
      }
      if (boost.multiplierOfTotalSupplyPerSecond.unwrap() > 0) {
        uint256 totalSupply = twabController.getTotalSupplyTwabBet
        deltaAmount += convert(boost.multiplierOfTotalSupplyPerSec
      }
@>    uint256 availableBalance = _tokenOut.balanceOf(address(this)
@>    deltaAmount = availableBalance > deltaAmount ? deltaAmount :
      return boost.available + deltaAmount;
    }
```

The current implementation code, limiting the maximum value of `deltaAmount` is wrong, using the minimum value compared to the current balance `_tokenOut.balanceOf(address(this))`.

But the current balance includes the previously accumulated `boost.available`, so normally it should be compared to the difference between the current balance and `boost.available`.

So the value returned may be larger than the current balance, and `LiquidationPair.sol` performs `source.liquidatableBalanceOf()` and `source.liquidate()` with too large a number, resulting in a failed transfer.

Recommended Mitigation Steps

The maximum value returned should not exceed the current balance

```
    function _computeAvailable(IERC20 _tokenOut) internal view ret
      Boost memory boost = _boosts[_tokenOut];
      uint256 deltaTime = block.timestamp - boost.lastAccruedAt;
      uint256 deltaAmount;
      if (deltaTime == 0) {
        return boost.available;
      }
      if (boost.tokensPerSecond > 0) {
        deltaAmount = boost.tokensPerSecond * deltaTime;
      }
      if (boost.multiplierOfTotalSupplyPerSecond.unwrap() > 0) {
        uint256 totalSupply = twabController.getTotalSupplyTwabBet
        deltaAmount += convert(boost.multiplierOfTotalSupplyPerSec
      }
      uint256 availableBalance = _tokenOut.balanceOf(address(this)
  -       deltaAmount = availableBalance > deltaAmount ? deltaAmount :
  -       return boost.available + deltaAmount;
  +       uint256 result = boost.available + deltaAmount;
  +       if (result > availableBalance) result = availableBalance;
  +       return result;

    }
```

Assessed type

Context

[asselstine (PoolTogether) confirmed](#)

[M-07] Liquidators can be tricked to operate with `LiquidationPairs` that were deployed using the `LiquidationPairFactory` but they configured the `LiquidationSource` as a fake malicious contract

Submitted by [0xStalin](#)*, also found by* [MohammedRizwan](#)

Users and Bots can be tricked into operating with LiquidationPairs that were deployed using the LiquidationPairFactory but they configured the LiquidationSource as a fake malicious contract that will allow the Liquidator's creator to steal all the POOL tokens that were meant to be used to liquidate the Vault's Yield.

Proof of Concept

- The current implementation of the `LiquidationPairFactory::createPair()` allows the callers to set the LiquidationSource as any arbitrary address with no restrictions.
- The problem is that the address of the `_source` parameter may not necessarily be a real vault contract, and even though the `_source` address is set as a fake malicious contract, the LiquidationPair will be created and added to the `deployedPairs` mapping, and **as stated in the protocol's documentation, any LiquidationPair created by the factory determines if a pair is legitimate or not**

Liquidation Pair Protocol's Documentation

- So, if a LiquidationPair is created by the LiquidationPairFactory may allow malicious users to trick users who want to liquidate the Vault's Yield to operate with a LiquidationPair who'll end up stealing their POOL tokens (tokenIn) when swapping tokens.
- Practical Example of how the LiquidationPair would steal the user's assets, (Keep in mind that `source` is not the address of a Vault, but an arbitrary contract)

  - The fake `source` contract would look like this:

```
contract FakeSource {

  function targetOf(address _token) external view returns (addre
    return <ContractCreatorAddress>;
  }

  function liquidate(
    address _account,
    address _tokenIn,
    uint256 _amountIn,
    address _tokenOut,
    uint256 _amountOut
  ) public virtual override returns (bool) {
    return true;
  }

}
```

1. Calling `LiquidationPair.target()` will return `source.targetOf(tokenIn)`, and the `source` contract can return any address when the `targetOf()` is called, so, let's say that will return the address of its creator.
2. So, `LiquidationPair.target()` will return the address of its creator, instead of returning the expected address of the `PrizePool` contract
3. Calling `LiquidationPair::swapExactAmount()` will do some computations prior to call `source.liquidate()`, and the `source.liquidate()` can just return true not to cause the tx to be reverted.
4. So, `LiquidationPair::swapExactAmount()` will basically do nothing.
5. With the above points in mind, let's see what would be the result of swapping using the LiquidationRouter contract

```
function swapExactAmountOut(
  LiquidationPair _liquidationPair,
  address _receiver,
  uint256 _amountOut,
  uint256 _amountInMax
) external onlyTrustedLiquidationPair(_liquidationPair) returns

  //@audit-issue => The `tokenIn` will be transferred to the adc
  IERC20(_liquidationPair.tokenIn()).safeTransferFrom(
    msg.sender,
    _liquidationPair.target(),
    _liquidationPair.computeExactAmountIn(_amountOut)
  );

  //@audit-issue => This call will basically do nothing, just re
  uint256 amountIn = _liquidationPair.swapExactAmountOut(_receiv

  emit SwappedExactAmountOut(_liquidationPair, _receiver, _amour

  return amountIn;
}
```

Recommended Mitigation Steps

- Use the `deployedVaults` mapping of the `VaultFactory` contract to validate if the inputted address of the `_source` parameter is a valid vault supported by the Protocol.

- Additionally, it could be a good idea to set the `_tokenIn` and `_tokenOut` by pulling the values that are already set up in the vault.

```
function createPair(
  ILiquidationSource _source,
- address _tokenIn,
- address _tokenOut,
  uint32 _periodLength,
  uint32 _periodOffset,
  uint32 _targetFirstSaleTime,
  SD59x18 _decayConstant,
  uint112 _initialAmountIn,
  uint112 _initialAmountOut,
  uint256 _minimumAuctionAmount
) external returns (LiquidationPair) {

+ require(VaultFactory.deployedVaults(address(_source)) == true,
+ address _prizePool = _source.prizePool();
+ address _tokenIn = _prizePool.prizeToken();
+ address _tokenOut = address(_source);

  LiquidationPair _liquidationPair = new LiquidationPair(
    _source,
    _tokenIn,
    _tokenOut,
    _periodLength,
    _periodOffset,
    _targetFirstSaleTime,
    _decayConstant,
    _initialAmountIn,
    _initialAmountOut,
    _minimumAuctionAmount
  );

  allPairs.push(_liquidationPair);
  deployedPairs[_liquidationPair] = true;

  emit PairCreated(
    _liquidationPair,
    _source,
    _tokenIn,
    _tokenOut,
    _periodLength,
    _periodOffset,
    _targetFirstSaleTime,
    _decayConstant,
    _initialAmountIn,
    _initialAmountOut,
    _minimumAuctionAmount
  );

  return _liquidationPair;
}
```

Assessed type

Invalid Validation

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

> This is an interesting report.

> There are two ways we expect liquidations to occur:

- EOAs that interact with the LiquidationRouter. (users, off-chain bots, etc)
- Smart Contracts that interact with the LiquidationPair directly

> Smart contracts that interact with the Pair can easily assert that they received the expected tokens; so a malicious LiquidationSource would not be a problem.

> However, EOAs that interact with a Pair via the LiquidationRouter would be vulnerable to this kind of attack.

> This makes me think that the fix is actually to alter the LiquidationRouter so that:

1. In `swapExactAmountOut` the router makes itself the recipient of the tokens.
2. The router asserts that it has received the expected amount of tokens
3. The router transfers to the tokens to the caller.

> It costs more gas, but this change ensures that anyone who calls the router would receive the expected tokens.

[M-08] An attacker can preemptively block the configuration of boost values or the liquidation pair in `VaultBooster` through front-running

Submitted by 3agle

https://github.com/GenerationSoftware/pt-v5-vault-boost/blob/9d640051ab61a0fdbcc9500814b7f8242db9aec2/src/VaultBooster.sol#L142-L165
https://github.com/GenerationSoftware/pt-v5-vault-boost/blob/9d640051ab61a0fdbcc9500814b7f8242db9aec2/src/VaultBooster.sol#L211-L237

Impact

- This issue is related to the `setBoost()` function in `VaultBooster`, which allows the owner to configure boost parameters for a specific token (`tokenOut`).
- The function includes a check on `_initialAvailable` to ensure it does not exceed the contract's balance.

```
if (_initialAvailable > 0) {
      uint256 balance = _token.balanceOf(address(this));
      if (balance < _initialAvailable) {
        revert InitialAvailableExceedsBalance(_initialAvailable,
}
```

- However, an attacker can front-run the owner's transaction and call `liquidate()` through the Liquidation Pair contract, reducing the contract's balance. As a result, the owner's transaction will revert, preventing the update of the liquidation pair and other boost parameters.
- To initiate this attack, the attacker does not need a large amount of tokens. Even a liquidation amount as small as `1 wei` is sufficient to prevent the owner from configuring the boost parameters for as long as needed. This allows the attacker to maintain control and hinder the owner's ability to update the boost settings.
- The inability to change the values such as `_multiplierOfTotalSupplyPerSecond` and `_tokensPerSecond` when

needed could lead to suboptimal boost strategies, inefficiencies, and missed opportunities for the associated prize vault. Flexibility in adjusting these parameters is crucial for adapting to changing market conditions and maintaining competitiveness in the dynamic DeFi ecosystem.

Proof of Concept

Assembling this PoC will take a little work as the standard tests used only mock addresses instead of actual contracts.

- Create a folder `/2023-08-pooltogether/pt-v5-vault-boost/test/PoC`
- Add the following code to `/2023-08-pooltogether/pt-v5-vault-boost/test/PoC/MockERC20.sol`

```
// SPDX-License-Identifier: MIT
pragma solidity 0.8.19;

import "openzeppelin/token/ERC20/ERC20.sol";

contract MockERC20 is ERC20 {
  constructor(string memory _name, string memory _symbol) ERC20

  function mint(address to, uint256 amount) public {
    _mint(to, amount);
  }
}
```

- Copy `/2023-08-pooltogether/pt-v5-cgda-liquidator/src/libraries/ContinuousGDA.sol` to `/2023-08-pooltogether/pt-v5-vault-boost/test/PoC/ContinuousGDA.sol`
- Copy `/2023-08-pooltogether/pt-v5-cgda-liquidator/src/LiquidationPair.sol` to `/2023-08-pooltogether/pt-v5-vault-boost/test/PoC/LiquidationPair.sol`
- Edit the import of ContinuosGDA at line 8 in `LiquidationPair.sol` as follows:

```
import { ContinuousGDA } from "./ContinuousGDA.sol";
```

- Add the following code to `/2023-08-pooltogether/pt-v5-vault-boost/test/PoC/PoC.t.sol`

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity 0.8.19;

import "forge-std/Test.sol";
import "forge-std/console.sol";

import "./MockERC20.sol";
import "./LiquidationPair.sol";

import { SD1x18, unwrap, UNIT, sd1x18 } from "prb-math/SD1x18.sc
import { UD2x18, ud2x18 } from "prb-math/UD2x18.sol";

import { VaultBooster, Boost, UD60x18, UD2x18, InitialAvailableE
import { PrizePool, TwabController, ConstructorParams, IERC20 }

contract PoC is Test {

  // Tons of params required to setup the whole PoolTogether sys

  ConstructorParams params;
  VaultBooster booster;
  LiquidationPair liquidationPair;
  ILiquidationSource source;
  TwabController twabController;
  PrizePool prizePool;
  MockERC20 boostToken;
  MockERC20 prizeToken;

  address vault;
  SD59x18 decayConstant = wrap(0.001e18);
  uint32 periodLength = 1 days;
  uint32 periodOffset = 1 days;
  uint32 targetFirstSaleTime = 12 hours;
  uint112 initialAmountIn = 1e18;
  uint112 initialAmountOut = 1e18;
  uint256 minimumAuctionAmount = 0;
  uint32 drawPeriodSeconds = 1 days;
  uint64 lastClosedDrawStartedAt = uint64(block.timestamp + 1 da
  uint8 initialNumberOfTiers = 3;
  address drawManager = address(this);
```

```
    function setUp() public {
      //TokenIn
      prizeToken = new MockERC20("PrizeToken", "PT");

      //TokenOut
      boostToken = new MockERC20("BoostToken", "BT");

      //TwabController
      twabController = new TwabController(drawPeriodSeconds, uint3

      //Prize Vault
      vault = makeAddr("vault");

      //Prize Pool
      params = ConstructorParams(
        IERC20(address(prizeToken)),
        twabController,
        drawManager,
        drawPeriodSeconds,
        lastClosedDrawStartedAt,
        initialNumberOfTiers,
        100,
        10,
        10,
        ud2x18(0.9e18),
        sd1x18(0.9e18)
      );
      prizePool = new PrizePool(params);

      //Vault Booster
      booster = new VaultBooster(prizePool, vault, address(this));

      //Liquidation Pair
      source = ILiquidationSource(address(booster));
      liquidationPair = new LiquidationPair(
        source,
        address(prizeToken),
        address(boostToken),
        periodLength,
        periodOffset,
        targetFirstSaleTime,
        decayConstant,
        initialAmountIn,
        initialAmountIn,
        minimumAuctionAmount
      );
    }

    function testFrontRun() public {
      vm.warp(0);

      //Minting 1e18 Boost Tokens to the Booster
      boostToken.mint(address(booster), 1e18);

      //Setting up the Booster to allow liquidation for Boost Toke
      booster.setBoost(boostToken, address(liquidationPair), UD2x1

      //Ensuring VaultBooster is properly configured
      Boost memory boost = booster.getBoost(boostToken);
      assertEq(boost.available, 1e18);

      vm.warp(10);

      //Now the Vault Booster's owner decides to update the boost
      //But attacker front-runs it by doing the following two step

      //1. Attacker sends 100 wei Prize Tokens to Prize Pool
      prizeToken.mint(address(prizePool), 100);
      //2. Attacker calls the liquidation Pair to liquidate 100 we
      vm.prank(address(liquidationPair));
      booster.liquidate(address(this), address(prizeToken), 100, a

      //The transcation to update the boost will revert as `_initi
      vm.expectRevert();
      booster.setBoost(boostToken, address(liquidationPair), UD2x1
    }
  }
```

- Run the following command in `/2023-08-pooltogether/pt-v5-vault-boost/`:

```
forge test --mc "PoC" -vvvv
```

Recommended Mitigation Steps

- Add a pausing functionality on liquidation to allow Vault Booster's owners to update the boost values.

hickuphh3 (judge) decreased severity to Medium and commented:

> Valid griefing concern.

asselstine (PoolTogether) acknowledged

[M-09] Create methods are suspicious of the reorg attack

Submitted by MohammedRizwan*, also found by* MohammedRizwan, nadin*, and* ptsanev

The createVaultBooster() function deploys a new VaultBooster contract using the `create`, where the address derivation depends only on the VaultBoosterFactory nonce.

Re-orgs can happen in all EVM chains and as confirmed the contracts will be deployed on most EVM compatible L2s including Arbitrum, etc. It is also planned to be deployed on ZKSync in future. In ethereum, where this is deployed, Re-orgs has already been happened. For more info, check here.

This issue will increase as some of the chains like Arbitrum and Polygon are suspicious of the reorg attacks.

Polygon re-org reference: click here. This one happened this year in February, 2023.

Polygon blocks forked: check here

The issue would happen when users rely on the address derivation in advance or try to deploy the position clone with the same address on different EVM chains, any funds sent to the `new` contract could potentially be withdrawn by anyone else. All in all, it could lead to the theft of user funds.

```
File: src/VaultBoosterFactory.sol

    function createVaultBooster(PrizePool _prizePool, address _v
>>        VaultBooster booster = new VaultBooster(_prizePool, _v
        emit CreatedVaultBooster(booster, _prizePool, _vault, _c

        return booster;
    }
```

Optimistic rollups (Optimism/Arbitrum) are also suspect to reorgs since if someone finds a fraud the blocks will be reverted, even though the user receives a confirmation.

Attack Scenario
Imagine that Alice deploys a new VaultBooster, and then sends funds to it. Bob sees that the network block reorg happens and calls createVaultBooster. Thus, it creates VaultBooster with an address to which Alice sends funds. Then Alices' transactions are executed and Alice transfers funds to Bob's controlled VaultBooster.

This is a Medium severity issue that has been referenced from below Code4rena reports:
https://code4rena.com/reports/2023-01-rabbithole/#m-01-questfactory-is-suspicious-of-the-reorg-attack
https://code4rena.com/reports/2023-04-frankencoin#m-14-re-org-attack-in-factory

Proof of Concept

https://github.com/GenerationSoftware/pt-v5-vault-boost/blob/9d640051ab61a0fdbcc9500814b7f8242db9aec2/src/VaultBoosterFactory.sol#L29

Recommended Mitigation Steps

Deploy such contracts via `create2` with `salt` that includes `msg.sender`.

asselstine (PoolTogether) confirmed via duplicate issue `#169`

hickuphh3 (judge) commented:

> Accepting because of this:

> *"Imagine that Alice deploys a new VaultBooster, and then sends funds to it. Bob sees that the network block reorg happens and calls createVaultBooster. Thus, it creates VaultBooster with an address to which Alice sends funds. Then Alices' transactions are executed and Alice transfers funds to Bob's controlled VaultBooster."*

> Again, the scenario should have been more explicit: stating how the vault could be different, how funds are transferred & possibly exploited.

[M-10] The `ContinuousGDA` implementation is incorrect leading to liquidation auctions running at the wrong price

Submitted by [dirk_y](), also found by [Angry_Mustache_Man]()

[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L39-L41]()
[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L65]()
[https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L86]()

The `LiquidationPair` contract facilitates Periodic Continuous Gradual Dutch Auctions for yield. This uses the underlying `ContinuousGDA.sol` library in order to correctly price the auctions.

However this library incorrectly implements the formula, using the emission rate in a few places where it should use the decay constant. Since the decay constant is usually less than the emission rate (as can also be seen from the test suite), this means that the `purchasePrice` calculation is lower than it should be, meaning that liquidations are over-incentivised.

Proof of Concept

This is difficult to demonstrate given the issue is basically just that the formula in [https://www.paradigm.xyz/2022/04/gda]() has been wrongly implemented. However

I'll point out a few issues in the code:

```
function purchasePrice(
  SD59x18 _amount,
  SD59x18 _emissionRate,
  SD59x18 _k,
  SD59x18 _decayConstant,
  SD59x18 _timeSinceLastAuctionStart
) internal pure returns (SD59x18) {
  if (_amount.unwrap() == 0) {
    return SD59x18.wrap(0);
  }
  SD59x18 topE = _decayConstant.mul(_amount).div(_emissionRate
  topE = topE.exp().sub(ONE);
  SD59x18 bottomE = _decayConstant.mul(_timeSinceLastAuctionSt
  bottomE = bottomE.exp();
  SD59x18 result;
  if (_emissionRate.unwrap() > 1e18) {
    result = _k.div(_emissionRate).mul(topE).div(bottomE);
  } else {
    result = _k.mul(topE.div(_emissionRate.mul(bottomE)));
  }
  return result;
}
```

In the `result` calculation you can see that `_k` is divided by `_emissionRate`. However, according to the proper formula, `_k` should be divided by `_decayConstant`.

Another issue occurs in `purchaseAmount` where `_k` is added to `lnParam` instead of ONE and `price` is multiplied by `_emissionRate` instead of `_decayConstant`:

```
function purchaseAmount(
  SD59x18 _price,
  SD59x18 _emissionRate,
  SD59x18 _k,
  SD59x18 _decayConstant,
  SD59x18 _timeSinceLastAuctionStart
) internal pure returns (SD59x18) {
  if (_price.unwrap() == 0) {
    return SD59x18.wrap(0);
  }
  SD59x18 exp = _decayConstant.mul(_timeSinceLastAuctionStart)
  SD59x18 lnParam = _k.add(_price.mul(_emissionRate).mul(exp))
  SD59x18 numerator = _emissionRate.mul(lnParam.ln());
  SD59x18 amount = numerator.div(_decayConstant);
  return amount;
}
```

I would suggest double checking the formula and the derivation of the complementary formulas to calculate amount/k. The correct implementation is show in the diff below.

Recommended Mitigation Steps

The implementation should be updated to correctly calculate the price for a continuous GDA. I have made the required fixes in the diff below:

```
diff --git a/src/libraries/ContinuousGDA.sol b/src/libraries/Con
index 721d626..7e2bb61 100644
--- a/src/libraries/ContinuousGDA.sol
+++ b/src/libraries/ContinuousGDA.sol
@@ -36,9 +36,9 @@ library ContinuousGDA {
     bottomE = bottomE.exp();
     SD59x18 result;
     if (_emissionRate.unwrap() > 1e18) {
-      result = _k.div(_emissionRate).mul(topE).div(bottomE);
+      result = _k.div(_decayConstant).mul(topE).div(bottomE);
     } else {
-      result = _k.mul(topE.div(_emissionRate.mul(bottomE)));
+      result = _k.mul(topE.div(_decayConstant.mul(bottomE)));
     }
     return result;
   }
@@ -62,7 +62,7 @@ library ContinuousGDA {
       return SD59x18.wrap(0);
     }
     SD59x18 exp = _decayConstant.mul(_timeSinceLastAuctionStart
-    SD59x18 lnParam = _k.add(_price.mul(_emissionRate).mul(exp)
+    SD59x18 lnParam = ONE.add(_price.mul(_decayConstant).mul(ex
     SD59x18 numerator = _emissionRate.mul(lnParam.ln());
     SD59x18 amount = numerator.div(_decayConstant);
     return amount;
@@ -83,7 +83,7 @@ library ContinuousGDA {
   ) internal pure returns (SD59x18) {
     SD59x18 exponent = _decayConstant.mul(_targetFirstSaleTime)
     SD59x18 eValue = exponent.exp();
-     SD59x18 multiplier = _emissionRate.mul(_price);
+     SD59x18 multiplier = _decayConstant.mul(_price);
     SD59x18 denominator = (_decayConstant.mul(_purchaseAmount).
     SD59x18 result = eValue.div(denominator);
     return result.mul(multiplier);
```

Assessed type

Math

RaymondFam (Lookout) commented:

> The formula in the doc is just one of many different price functions the protocol can work on.

asselstine (PoolTogether) confirmed

hickuphh3 (judge) commented:

> I agree that the pricing formula is up to the project to figure out and implement; it doesn't necessarily have to be the one Paradigm laid out in their blog post.

> Nevertheless, the sponsor confirmed the issue, so I will leave it as that.

> IMO the issue's severity can be raised to High, but I'd have liked to see further proof of price deviations to justify the higher severity. =)

> I suggest explicitly stating out what the intended pricing formula is, so that wardens can verify the implementation's correctness. Right now, it's guesswork.

[M-11] `VaultBooster` : users tokens will be stuck if they deposited with unsupported boost tokens

Submitted by hals, *also found by* 14si2o_Flint

Impact

- In `VaultBooster` contract : users can deposite their tokens to participate in boosting the chances of a vault winning.
- But in `deposit` function: users can deposit **any** ERC20 tokens without verifying if the token is supported or not (has a boost set to it; registered in `_boosts[_token]`).
- As there's no mechanism implemented in the contract for users to retreive their deposited tokens if these tokens are not supported; then they will lose them unless withdrawn by the `VaultBooster` owner; and then the owner transfers these tokens back to the users.
- If this behaviour is intended by design; then there must be a machanism to save unsupported deposited tokens with user address and amount; and another withdraw function accessible by the owner that enables transferring stuck tokens to their owners; or simply make `deposit` function reverts if the token is unsupported.

Proof of Concept

Code: Line 171-176

```
File: pt-v5-vault-boost/src/VaultBooster.sol
Line 171-176:
  function deposit(IERC20 _token, uint256 _amount) external {
    _accrue(_token);
    _token.safeTransferFrom(msg.sender, address(this), _amount);

    emit Deposited(_token, msg.sender, _amount);
  }
```

- Foundry PoC:
- A `MockERC20.t.sol` contract is added to the test folder to simulate the user experience (miting/approving..),

```
pragma solidity 0.8.19;

import "@openzeppelin/contracts/token/ERC20/ERC20.sol";
```

```
contract MockERC20 is ERC20 {
  constructor() ERC20("MockToken", "MT") {}

  function mint(address account, uint256 amount) public returns
    _mint(account, amount);
    return true;
  }
}
```

add remappings to the `foundry.toml`:
remappings = [“@openzeppelin/=lib/openzeppelin-contracts/“]

2. This test is set in `VaultBooster.t.sol` file, where a user deposits unsuppotred token (basically no boosts were set),tries to liquidate but the call will revert, then the unsupported tokens can be withdrawn by the owner only (and not transferred to the original depositor):
   Add this import line at the top of the test file:

```
import "./MockERC20.t.sol";
```

Add this test `testDepositWithUnsupportedToken()` to the `VaultBooster.t.sol` file:

```
  function testDepositWithUnsupportedToken() public {
    //0- minting unsupportedToken to the user
    address user = address(0x2);
    uint256 userBalance = 1e18;
    MockERC20 unsupportedToken = new MockERC20();
    vm.startPrank(user);
    unsupportedToken.mint(user, userBalance);
    assertEq(unsupportedToken.balanceOf(user), userBalance);

    //1.the user deposits unsupported token (mainly there's no b
    unsupportedToken.approve(address(booster), userBalance);

    vm.expectEmit(true, true, true, true);
    emit Deposited(unsupportedToken, user, userBalance);
    booster.deposit(unsupportedToken, userBalance);
    assertEq(unsupportedToken.balanceOf(user), 0);
    assertEq(unsupportedToken.balanceOf(address(booster)), userB
    //2. assertions that the deposited unsupportedToken doesn't
    Boost memory boost = booster.getBoost(unsupportedToken);
    assertEq(boost.liquidationPair, address(0));
    assertEq(boost.multiplierOfTotalSupplyPerSecond.unwrap(), 0,
    assertEq(boost.tokensPerSecond, 0, "tokensPerSecond");
    assertEq(boost.lastAccruedAt, block.timestamp); //as the dep

    //3. the user tries to call liquidate to get back his tokens
    vm.expectRevert(abi.encodeWithSelector(OnlyLiquidationPair.s
    booster.liquidate(user, address(prizeToken), 0, address(unsu
    vm.stopPrank();

    //4. unless the owner tries  withdraws the user stuck tokens
    assertEq(unsupportedToken.balanceOf(address(this)), 0);
    assertEq(unsupportedToken.balanceOf(address(booster)), userB
    booster.withdraw(unsupportedToken, userBalance);
    assertEq(unsupportedToken.balanceOf(address(this)), userBala
    assertEq(unsupportedToken.balanceOf(address(booster)), 0);
  }
```

3. Test result:

```
$ forge test --match-test testDepositWithUnsupportedToken
Running 1 test for test/VaultBooster.t.sol:VaultBoosterTest
[PASS] testDepositWithUnsupportedToken() (gas: 636179)
Test result: ok. 1 passed; 0 failed; finished in 3.36ms
```

Tools Used

Manual Testing & Foundry.

Recommended Mitigation Steps

Update `deposit` function to revert if the user tries to deposit unsuppoerted tokens (that doesn't have a boost set):

```
  function deposit(IERC20 _token, uint256 _amount) external {
+    if(_boosts[_token].liquidationPair==address(0)) revert();
    _accrue(_token);
    _token.safeTransferFrom(msg.sender, address(this), _amount);
```

```
        emit Deposited(_token, msg.sender, _amount);
    }
```

Assessed type

Token-Transfer

asselstine (PoolTogether) confirmed

Low Risk and Non-Critical Issues

For this audit, 6 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **0xmystery** received the top score from the judge.

The following wardens also submitted reports: Rolezn, 0xbepresent, MohammedRizwan, bin2chen, and rvierdiiev.

[01] Zero `_emissionRate` checks

`_emissionRate` is zero if the minimum fund is not met as shown in the code logic below.

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L274-L283

```
    function _computeEmissionRate() internal returns (SD59x18) {
      uint256 amount = source.liquidatableBalanceOf(tokenOut);
      // console2.log("_computeEmissionRate amount", amount);
      if (amount < minimumAuctionAmount) {
        // do not release funds if the minimum is not met
        amount = 0;
        // console2.log("AMOUNT IS ZERO");
      }
      return convert(int256(amount)).div(convert(int32(int(periodI
    }
```

As such, consider introducing checks on functions calls dependent on it when `_emissionRate == 0`. For example, it will be good if a check is implemented in function `swapExactAmountOut` to prevent division by zero in the function logic.

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L211-L226

```
    function swapExactAmountOut(
      address _account,
      uint256 _amountOut,
      uint256 _amountInMax
    ) external returns (uint256) {
      _checkUpdateAuction();
      uint swapAmountIn = _computeExactAmountIn(_amountOut);
      if (swapAmountIn > _amountInMax) {
        revert SwapExceedsMax(_amountInMax, swapAmountIn);
      }
+     if (_emissionRate == 0) {
+       revert EmissionIsZero(_emissionRate);
+     }
      _amountInForPeriod += uint96(swapAmountIn);
      _amountOutForPeriod += uint96(_amountOut);
      _lastAuctionTime += uint48(uint256(convert(convert(int256(_a
      source.liquidate(_account, tokenIn, swapAmountIn, tokenOut,
      return swapAmountIn;
    }
```

[02] Custom error misleading name

The name of the following custom error should be renamed as follows to be more in line of its intended purpose:

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L12

```
- error TargetFirstSaleTimeLtPeriodLength(uint passedTargetSaleT
+ error TargetFirstSaleTimeGtPeriodLength(uint passedTargetSaleT
```

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L118-L120

```
        if (targetFirstSaleTime >= periodLength) {
-           revert TargetFirstSaleTimeLtPeriodLength(targetFirstSaleT
+           revert TargetFirstSaleTimeGtPeriodLength(targetFirstSaleT
        }
```

[03] Inconsistent typecasting

The typecast below should be corrected as follows since `Conversions.convert` takes in `int256` parameter. Additionally, down-casting `periodLength` of `int` or `int256` to `int32` could lead to overflow issue.

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L282

```
-     return convert(int256(amount)).div(convert(int32(int(perioc
+     return convert(int256(amount)).div(convert(int256(periodLer
```

[04] Avoid caching global variables

There is no gas benefit caching a global variable like, `msg.sender`, `block.timestamp` etc. Moreover, the cached `timestamp` is only used once in either the `if` block or the last line of `return`, which further defeat the purpose of variable caching.

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationPair.sol#L375-L383

```
      /// @notice Computes the current auction period
      /// @return the current period
      function _computePeriod() internal view returns (uint256) {
```

```
-    uint256 _timestamp = block.timestamp;
-    if (_timestamp < periodOffset) {
+    if (block.timestamp < periodOffset) {
       return 0;
     }
-    return (_timestamp - periodOffset) / periodLength;
+    return (block.timestamp - periodOffset) / periodLength;
   }
```

[05] Missing constructor check for `AuctionDurationGteSequencePeriod()`

The error `RngAuction.AuctionDurationGteSequencePeriod` has been declared but never used. It should be utilized to implement the following missing check.

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngAuction.sol#L140-L156

```
   constructor(
     RNGInterface rng_,
     address owner_,
     uint64 sequencePeriod_,
     uint64 sequenceOffset_,
     uint64 auctionDurationSeconds_,
     uint64 auctionTargetTime_
   ) Ownable(owner_) {
     if (sequencePeriod_ == 0) revert SequencePeriodZero();
     if (auctionTargetTime_ > auctionDurationSeconds_) revert Auc
+    if (auctionDurationSeconds_ > sequencePeriod_) revert Aucti
     sequencePeriod = sequencePeriod_;
     sequenceOffset = sequenceOffset_;
     auctionDuration = auctionDurationSeconds_;
     auctionTargetTime = auctionTargetTime_;
     _auctionTargetTimeFraction = intoUD2x18(convert(uint(auctior
     _setNextRngService(rng_);
   }
```

[06] Grammatical and spelling errors

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationRouter.sol#L60

```
          @ audit `exactly` --> `exact`
      /// @param _amountOut The `exactly` amount of output tokens ex
```

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngRelayAuction.sol#L74

```
          @ audit `wil` --> `will`
      /// @notice The PrizePool whose draw `wil` be closed.
```

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngRelayAuction.sol#L48

```
          @ audit `Not` --> `Note`
      /// @dev `Not` that the reward fractions compound
```

[07] Possible reverse order of `if else` logic

The order of the `if else` logic seems to have been reversed. Depending on the values of `_emissionRate.unwrap()` and `_emissionRate` involved, this could be crucial enough to make the function truncate to zero when `_emissionRate` was more than one (1e18) and used as the divisor. On the other hand, the function could also overflow when `_emissionRate` was less than one and used as the divisor. If that's the case, I suggest removing the `if` clause and keep only the `else` clause (with one of the parentheses removed) that will cater to both circumstances.

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/libraries/ContinuousGDA.sol#L38-L42

```
-     if (_emissionRate.unwrap() > 1e18) {
-       result = _k.div(_emissionRate).mul(topE).div(bottomE);
-     } else {
-       result = _k.mul(topE.div(_emissionRate.mul(bottomE)));
+       result = _k.mul.topE.div(_emissionRate.mul(bottomE));
-     }
```

[08] Avoid caching state variables that will only be used once

Caching state variables that will only be used once incurs more gas and is not recommended.

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngAuction.sol#L244-L255

```
    function getLastAuctionResult()
      external
      view
      returns (AuctionResult memory)
    {
-     address recipient = _lastAuction.recipient;
-     UD2x18 rewardFraction = _lastAuction.rewardFraction;
      return AuctionResult({
-       recipient: recipient,
+       recipient: _lastAuction.recipient,
-       rewardFraction: rewardFraction
+       rewardFraction: _lastAuction.rewardFraction
      });
    }
```

[09] Consider using descriptive constants when passing zero as a function argument

Passing zero as a function argument can sometimes result in a security issue (e.g. passing zero as the slippage parameter, fees, token amounts ...). Consider using a constant variable with a descriptive name, so it’s clear that the argument is intentionally being used, and for the right reasons.

Here is one specific instance found.

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/RngAuctionRelayerRemoteOwner.sol#L65

```
                RemoteOwnerCallEncoder.encodeCalldata(address(_remot
```

[10] Enhanced reward distribution to safeguard against reserve depletion in auction calculations

The `rewards` function in the provided `RewardLib` library calculates the rewards for a series of auctions based on a given reserve. A potential issue is that midway through the calculations, the `remainingReserve` might become insufficient to provide the required reward for an auction, potentially causing underflows or incorrect behavior. To address this concern, it's recommended to implement a check to ensure that each reward doesn't exceed the `remainingReserve` , and if it does, the function should gracefully exit the loop to avoid unexpected results.

Here's a suggested fix:

https://github.com/GenerationSoftware/pt-v5-draw-auction/blob/f1c6d14a1772d6609de1870f8713fb79977d51c1/src/libraries/RewardLib.sol#L58-L70

```
      function rewards(
        AuctionResult[] memory _auctionResults,
        uint256 _reserve
      ) internal pure returns (uint256[] memory) {
        uint256 remainingReserve = _reserve;
        uint256 _auctionResultsLength = _auctionResults.length;
        uint256[] memory _rewards = new uint256[](_auctionResultsLen
        for (uint256 i; i < _auctionResultsLength; i++) {
-         _rewards[i] = reward(_auctionResults[i], remainingReserve
-         remainingReserve = remainingReserve - _rewards[i];

+         uint256 calculatedReward = reward(_auctionResults[i], rem
+         if (calculatedReward > remainingReserve) {
```

```
+             break; // Stop if there's not enough remaining rese
+         }
+       _rewards[i] = calculatedReward;
+       remainingReserve = remainingReserve - calculatedReward;
      }
      return _rewards;
    }
```

[11] Code efficiency

In `LiquidationRouter.swapExactAmountOut`, `IERC20(_liquidationPair.tokenIn()).safeTransferFrom()` should be moved to `LiquidationPair.swapExactAmountOut` to avoid calling `computeExactAmountIn()` twice.

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/LiquidationRouter.sol#L63-L80

```
    function swapExactAmountOut(
      LiquidationPair _liquidationPair,
      address _receiver,
      uint256 _amountOut,
      uint256 _amountInMax
    ) external onlyTrustedLiquidationPair(_liquidationPair) returr
-      IERC20(_liquidationPair.tokenIn()).safeTransferFrom(
-        msg.sender,
-        _liquidationPair.target(),
-        _liquidationPair.computeExactAmountIn(_amountOut)
-      );

      uint256 amountIn = _liquidationPair.swapExactAmountOut(_rece

      emit SwappedExactAmountOut(_liquidationPair, _receiver, _amc

      return amountIn;
    }
```

https://github.com/GenerationSoftware/pt-v5-cgda-liquidator/blob/7f95bcacd4a566c2becb98d55c1886cadbaa8897/src/Liquidation

Pair.sol#L211-L226

```
      function swapExactAmountOut(
        address _account,
        uint256 _amountOut,
        uint256 _amountInMax
      ) external returns (uint256) {
        _checkUpdateAuction();
        uint swapAmountIn = _computeExactAmountIn(_amountOut);
        if (swapAmountIn > _amountInMax) {
          revert SwapExceedsMax(_amountInMax, swapAmountIn);
        }
+        tokenIn.safeTransferFrom(tx.origin, target(), swapAmountIn)
        _amountInForPeriod += uint96(swapAmountIn);
        _amountOutForPeriod += uint96(_amountOut);
        _lastAuctionTime += uint48(uint256(convert(convert(int256(_a
        source.liquidate(_account, tokenIn, swapAmountIn, tokenOut,
        return swapAmountIn;
      }
```

[12] Activate the optimizer

Before deploying your contract, activate the optimizer when compiling using `solc --optimize --bin sourceFile.sol`. By default, the optimizer will optimize the contract assuming it is called 200 times across its lifetime. If you want the initial contract deployment to be cheaper and the later function executions to be more expensive, set it to `--optimize-runs=1`. Conversely, if you expect many transactions and do not care for higher deployment cost and output size, set `--optimize-runs` to a high number.

```
    module.exports = {
    solidity: {
    version: "0.8.19",
    settings: {
      optimizer: {
        enabled: true,
        runs: 1000,
      },
    },
    },
    };
```

Kindly visit the following site for further information:

https://docs.soliditylang.org/en/v0.5.4/using-the-compiler.html#using-the-commandline-compiler

Here is one particular example of instance on opcode comparison that delineates the gas saving mechanism:

```
    for !=0 before optimization
    PUSH1 0x00
    DUP2
    EQ
    ISZERO
    PUSH1 [cont offset]
    JUMPI

    after optimization
    DUP1
    PUSH1 [revert offset]
    JUMPI
```

Disclaimer from the warden: There have been several bugs with security implications related to optimizations. For this reason, Solidity compiler optimizations are disabled by default, and it is unclear how many contracts in the wild actually use them. Therefore, it is unclear how well they are being tested and exercised. High-severity security issues due to optimization bugs have occurred in the past. A high-severity bug in the emscripten -generated solc-js compiler used by Truffle and Remix persisted until late 2018. The fix for this bug was not reported in the Solidity CHANGELOG. Another high-severity optimization bug resulting in incorrect bit shift results was patched in Solidity 0.5.6. Please measure the gas savings from optimizations, and carefully weigh them against the possibility of an optimization-related bug. Also, monitor the development and adoption of Solidity compiler optimizations to assess their maturity.

asselstine (PoolTogether) confirmed

hickuphh3 (judge) commented:

> There are some gas optimization suggestions thrown into the report, but overall, decent recommendations and issues raised. None in particular that I disagree with.

Gas Optimizations

For this audit, 15 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Rolezn** received the top score from the judge.

The following wardens also submitted reports: Raihan, JCK, petrichor, SY_S, dharma09, SAQ, wahedtalash77, hunter_w3b, 0xta, 0xhex, ReyAdmirado, shamsulhaq123, Rageur, *and* K42.

Summary

	Issue	Contexts	Estimated Gas Saved
[G-01]	Avoid emitting event on every iteration	1	375
[G-02]	Counting down in `for` statements is more gas efficient	2	514
[G-03]	Use `assembly` to write address storage values	11	814
[G-04]	Multiple accesses of a mapping/array should use a local variable cache	8	640
[G-05]	Remove `forge-std` import	1	100
[G-06]	The result of a function call should be cached rather than re-calling the function	5	250
[G-07]	Use do while loops instead of for loops	2	8
[G-08]	Use nested `if` and avoid multiple check combinations	2	12
[G-09]	Using XOR (^) and AND (&) bitwise equivalents	6	78

Total: 44 contexts over 9 issues

[G-01] Avoid emitting event on every iteration

Expensive operations should always try to be avoided within loops. Such operations include: reading/writing to storage, heavy calculations, external calls, and emitting events. In this instance, an event is being emitted every iteration. Events have a base cost of `Glog (375 gas)` per emit and `Glogdata (8 gas) * number of bytes in event`. We can avoid incurring those costs each iteration by emitting the event outside of the loop.

Proof Of Concept

```
File: RngRelayAuction.sol

167: for (uint8 i = 0; i < _rewards.length; i++) {
      uint104 _reward = uint104(_rewards[i]);
      if (_reward > 0) {
        prizePool.withdrawReserve(auctionResults[i].recipient, _
        emit AuctionRewardDistributed(_sequenceId, auctionResult
      }
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngRelayAuction.sol#L167

[G-02] Counting down in `for` statements is more gas efficient

Counting down is more gas efficient than counting up because neither we are making zero variable to non-zero variable and also we will get gas refund in the last transaction when making non-zero to zero variable.

Proof Of Concept

```
File: RngRelayAuction.sol

167: for (uint8 i = 0; i < _rewards.length; i++) {
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngRelayAuction.sol#L167

```
File: RewardLib.sol

65: for (uint256 i; i < _auctionResultsLength; i++) {
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/libraries/RewardLib.sol#L65

Test Code

```
contract GasTest is DSTest {
    Contract0 c0;
    Contract1 c1;
    function setUp() public {
        c0 = new Contract0();
        c1 = new Contract1();
    }
    function testGas() public {
        c0.AddNum();
        c1.AddNum();
    }
}

contract Contract0 {
    uint256 num = 3;
    function AddNum() public {
        uint256 _num = num;
        for(uint i=0;i<=9;i++){
            _num = _num +1;
        }
        num = _num;
    }
}

contract Contract1 {
    uint256 num = 3;
    function AddNum() public {
        uint256 _num = num;
        for(uint i=9;i>=0;i--){
            _num = _num +1;
        }
        num = _num;
    }
}
```

Gas Test Report

Contract0 contract					
Deployment Cost	Deployment Size				
77011	311				
Function Name	min	avg	median	max	# calls
AddNum	7040	7040	7040	7040	1

Contract1 contract					
Deployment Cost	Deployment Size				
73811	295				
Function Name	min	avg	median	max	# calls
AddNum	3819	3819	3819	3819	1

[G-03] Use `assembly` to write address storage values

Proof Of Concept

▶ Details

[G-04] Multiple accesses of a mapping/array should use a local variable cache

Caching a mapping's value in a local storage or calldata variable when the value is accessed multiple times saves ~42 gas per access due to not having to perform the same offset calculation every time. Help the Optimizer by saving a storage variable's reference instead of repeatedly fetching it.

To help the optimizer,declare a storage type variable and use it instead of repeatedly fetching the reference in a map or an array. As an example, instead of repeatedly calling `someMap[someIndex]`, save its reference like this: `SomeStruct storage someStruct = someMap[someIndex]` and use it.

Proof Of Concept

```
File: RngRelayAuction.sol

170: prizePool.withdrawReserve(auctionResults[i].recipient, _rew
171: emit AuctionRewardDistributed(_sequenceId, auctionResults[i
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngRelayAuction.sol#L170-L171

```
File: AddressRemapper.sol

33: if (_destinationAddress[_addr] == address(0)) {
36: return _destinationAddress[_addr];
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/abstract/AddressRemapper.sol#L33

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/abstract/AddressRemapper.sol#L36

```
File: VaultBooster.sol

223: _boosts[IERC20(_tokenOut)].available = amountAvailable.toUi
224: _boosts[IERC20(_tokenOut)].lastAccruedAt = uint48(block.tim
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-vault-boost/src/VaultBooster.sol#L223-L224

```
File: VaultBooster.sol

251: _boosts[_tokenOut].available = available.toUint144();
```

```
252: _boosts[_tokenOut].lastAccruedAt = uint48(block.timestamp);
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-vault-boost/src/VaultBooster.sol#L251-L252

[G-05] Remove import `forge-std`

It’s used to print the values of variables while running tests to help debug and see what’s happening inside your contracts but since it’s a development tool, it serves no purpose on mainnet.

Also, the remember to remove the usage of calls that use `forge-std` when removing of the import of `forge-std`.

Proof Of Concept

```
File: VaultBooster.sol

4: import "forge-std/console2.sol";
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-vault-boost/src/VaultBooster.sol#L4

[G-06] The result of a function call should be cached rather than re-calling the function

External calls are expensive. Results of external function calls should be cached rather than call them multiple times. Consider caching the following:

Proof Of Concept

```
File: RngAuction.sol

370: uint64 currentTime = _currentTime();
382: uint64 currentTime = _currentTime();
```

```
386: return (_currentTime() - sequenceOffset) % sequencePeriod;
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngAuction.sol#L370

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngAuction.sol#L382

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngAuction.sol#L386

```
File: RemoteOwner.sol

119: if (_fromChainId() != _originChainId) revert OriginChainIdU
120: if (_msgSender() != address(_originChainOwner)) revert Orig
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/remote-owner/src/RemoteOwner.sol#L119

https://github.com/code-423n4/2023-08-pooltogether/tree/main/remote-owner/src/RemoteOwner.sol#L120

[G-07] Use do while loops instead of for loops

A do while loop will cost less gas since the condition is not being checked for the first iteration.

Proof Of Concept

```
File: RngRelayAuction.sol

131: function rngComplete

for (uint8 i = 0; i < _rewards.length; i++) {
      uint104 _reward = uint104(_rewards[i]);
      if (_reward > 0) {
        prizePool.withdrawReserve(auctionResults[i].recipient, _
        emit AuctionRewardDistributed(_sequenceId, auctionResult
        }
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngRelayAuction.sol#L132

```
File: RewardLib.sol

58: function rewards

for (uint256 i; i < _auctionResultsLength; i++) {
      _rewards[i] = reward(_auctionResults[i], remainingReserve)
      remainingReserve = remainingReserve - _rewards[i];
    }
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/libraries/RewardLib.sol#L59

[G-08] Use nested `if` and avoid multiple check combinations

Using nested `if`, is cheaper than using `&&` multiple check combinations. There are more advantages, such as easier to read code and better coverage reports.

Proof Of Concept

```
File: LiquidationPair.sol

332: if (_amountInForPeriod > 0 && _amountOutForPeriod > 0) {
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-cgda-liquidator/src/LiquidationPair.sol#L332

```
File: RngAuction.sol

179: if (_feeToken != address(0) && _requestFee > 0) {
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/RngAuction.sol#L179

Test Code

```
contract GasTest is DSTest {
    Contract0 c0;
    Contract1 c1;

    function setUp() public {
        c0 = new Contract0();
        c1 = new Contract1();
    }

    function testGas() public {
        c0.checkAge(19);
        c1.checkAgeOptimized(19);
    }
}

contract Contract0 {

    function checkAge(uint8 _age) public returns(string memory){
        if(_age>18 && _age<22){
            return "Eligible";
        }
    }

}

contract Contract1 {

    function checkAgeOptimized(uint8 _age) public returns(strinc
        if(_age>18){
            if(_age<22){
                return "Eligible";
            }
        }
    }
}
```

Gas Test Report

Contract0 contract					
Deployment Cost	Deployment Size				
76923	416				
Function Name	min	avg	median	max	# calls
checkAge	651	651	651	651	1

Contract1 contract					
Deployment Cost	Deployment Size				
76323	413				
Function Name	min	avg	median	max	# calls
checkAgeOptimized	645	645	645	645	1

[G-09] Using XOR (^) and AND (&) bitwise equivalents

Given 4 variables a, b, c and d represented as such:

```
0 0 0 0 0 1 1 0 <- a
0 1 1 0 0 1 1 0 <- b
0 0 0 0 0 0 0 0 <- c
1 1 1 1 1 1 1 1 <- d
```

To have `a == b` means that every 0 and 1 match on both variables. Meaning that a XOR (operator ^) would evaluate to 0 (`(a ^ b) == 0`), as it excludes by definition any equalities. Now, if `a != b` , this means that there's at least somewhere a 1 and a 0 not matching between a and b, making `(a ^ b) != 0` .Both formulas are logically equivalent and using the XOR bitwise operator costs actually the same amount of gas.However, it is much cheaper to use the bitwise OR operator (`|`) than comparing the truthy or falsy values.These are logically equivalent too, as the OR bitwise operator (`|`) would result in a 1 somewhere if any value is not 0 between the XOR (^) statements, meaning if any XOR (^) statement verifies that its arguments are different.

Proof Of Concept

```
File: LiquidationPair.sol

298: if (_amountOut == 0) {
314: if (purchasePrice == 0) {
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-cgda-liquidator/src/LiquidationPair.sol#L298

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-cgda-liquidator/src/LiquidationPair.sol#L314

```
File: RewardLib.sol

83: if (_auctionResult.recipient == address(0)) return 0;
84: if (_reserve == 0) return 0;
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-draw-auction/src/libraries/RewardLib.sol#L83-L84

```
File: VaultBooster.sol

266: if (deltaTime == 0) {
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/pt-v5-vault-boost/src/VaultBooster.sol#L266

```
File: RemoteOwner.sol

104: if (_newOriginChainOwner == address(0)) revert OriginChain(
```

https://github.com/code-423n4/2023-08-pooltogether/tree/main/remote-owner/src/RemoteOwner.sol#L104

Test Code

```
contract GasTest is DSTest {
    Contract0 c0;
    Contract1 c1;
    function setUp() public {
        c0 = new Contract0();
        c1 = new Contract1();
    }
    function testGas() public {
        c0.not_optimized(1,2);
        c1.optimized(1,2);
    }
}

contract Contract0 {
    function not_optimized(uint8 a,uint8 b) public returns(bool)
        return ((a==1) || (b==1));
    }
}

contract Contract1 {
    function optimized(uint8 a,uint8 b) public returns(bool){
        return ((a ^ 1) & (b ^ 1)) == 0;
    }
}
```

Gas Test Report

Contract0 contract					
Deployment Cost	Deployment Size				
46099	261				
Function Name	min	avg	median	max	# calls
not_optimized	456	456	456	456	1

Contract1 contract					
Deployment Cost	Deployment Size				
42493	243				
Function Name	min	avg	median	max	# calls
optimized	430	430	430	430	1

asselstine (PoolTogether) confirmed

Audit Analysis

For this audit, 7 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 **3agle** received the top score from the judge.

The following wardens also submitted reports: 0xSmartContract, 0xmystery, hunter_w3b, cholakov, DedOhWale, *and* K42.

Codebase quality

The overall quality of the codebase for PoolTogether can be classified as "*Good*".

Strengths

- Natspec was really helpful and detailed.
- It tries to achieve complete decentraliztion. It's remarkable they can do this without any governance.

Weaknesses

- Majority of the tests were using mock addresses instead of actual contracts. This is not recommended.
- Almost zero documentation for 50% of the contracts in-scope.

Mechanism Review

- PoolTogether is a decentralized finance protocol that combines savings and lottery, allowing users to deposit funds and have a chance to win rewards in daily prize draws, while maintaining the ability to withdraw their initial deposits at any time.
- Following are the major parts of Pooltogether system:

    - **Prize Pool (OOS):** The Prize Pool receives POOL tokens from Vaults, and releases the tokens as prizes in daily Draws.

- **Prize Vaults (or Vaults) (OOS):** Users deposit tokens into Vaults in order to be eligible to win prizes. Vaults generate yield, liquidate the yield for POOL tokens, and contribute the POOL to the Prize Pool. The amount of contributed POOL determines the Vault's portion of the odds.
- **TwabController (OOS):** A system that keeps track of users token balances, their historic balances and their average balances of historic time periods.
- **Prize Claimer (OOS):** Instead of users claiming their prize, the system incentivizes bots to claim the user's prizes for them.
- ****Draw Auction:**** A system that leverages an incentivisation mechanism to encourage competition among third parties to complete the draws in a timely manner while maximizing cost efficiency.
- **Vault Boosters:** Allows to boost a vault winning chance by providing more liquidity to be exchanged for Pool Tokens.
- **Liquidation Router:** End-users (liquidators) use this contract to swap vault shares or ERC20 tokens (like USDC) for POOL Tokens from the Vault or Vault Boosters. More sophisticated liquidators can directly call the Liquidation Pair.
- **Liquidation Pair**: The call of Liquidation Router goes through a liquidation pair. It calculates how much amount should be swapped for the provided tokens. Vaults only accept call from their assigned liquidation pair to liquidate the tokens/shares.

- **Deposit & Withdrawal Flow**
  Note: See full details and chart in warden's original submission.

  - You have `1000 USDC`. You deposit that into the Pooltogether's USDC Prize Vault .
  - This vault sends the USDC amount to a yield vault (e.g. AAVE) and in-turn recieves yield vault's shares. Then the prize vault will mint shares to the user.
  - All of this happens in the same transaction.
  - The withdrawal process is vice-versa.
  - So, multiple users deposit their USDC into the prize vault which is then deposited into yield vault. This vault recieves yield generated from the Yield Vault. This yield is auctioned off for POOL tokens which are deposited in Prize Pool to increase the vault's chances of winning.

    - *__Note__: Minting and burning of shares of all vaults happen through a singleton contract - `TWABController`. It is omitted for simplicity.*

- **Liquidation Flow**
  Note: See full details and chart in warden's original submission.

    - When a liquidator initiates the liquidation process, they call `swapExactAmountOut` on the LiquidationRouter. The router then transfers the POOL tokens from the liquidator and to the Prize Pool on behalf of the vault. The router also checks that the liquidation pair beinng called is deployed by the liquidation factory.
    - It then calls the `swapExactAmountOut` on Liquidation Pair associated with the vault. In the LiquidationPair, there are two tokens involved:

        - `tokenIn` : POOL token
        - `tokenOut` : Vault shares

      The liquidation pair then calls `liquidate` on the Vault.

    - The vault then calls `contributePrizeTokens` on the PrizePool to register the vault's contribution. Then, it mints the vault shares to the liquidator.
    - *__Note__: The liquidation pair uses Continuous Gradual Dutch Auction system to sell the vault shares or ERC20 tokens (in case of Vault Boosters).*

- **Vault Boosters**
  Note: See full details and chart in warden's original submission.

    - A prize vault can increase its chances of winning by using a Vault Booster.
    - Here's how it works: When a vault is created, a corresponding Vault Booster is deployed and linked to the vault's address (set in the constructor). The owner of the Vault Booster sets up corresponding Liquidation Pair for all the supported tokens.
    - Liquidators can then use the `swapExactAmountOut` function on the LiquidationRouter, providing the address of the Vault Booster's liquidation pair.
    - In this case, instead of receiving vault shares, the liquidator will receive ERC20 tokens, such as USDC or WETH. Additionally, the router will transfer

POOL tokens from the liquidator to the Prize Pool as a contribution for the vault.

- Hence, with the help of vault boosters, a vault can improve its odds of winning by offering additional ERC20 tokens to liquidators in return for Pool Tokens.

Centralization risks

- The protocol has made significant progress towards decentralization from V4 to V5, and the efforts to achieve this are commendable.
- Regarding the auctions used to obtain a random number from a third party for the draw, it is crucial to carefully assess this aspect. There might be a centralization risk if the third party wins the auction and attempts to manipulate the Prize Pool draws by providing a non-random number.

Systemic Risks

- Chainlink VRF is critical for fair prize draws. Any issues or unavailability with Chainlink VRF could impact the integrity of the draws.
- Like any smart contract-based system, PoolTogether is exposed to potential coding bugs or vulnerabilities. Exploiting these issues could result in the loss of funds or manipulation of the protocol.
- The continuous gradual Dutch auction (CGDA) mechanism is sensitive to market dynamics and potential manipulation. Fluctuations in token prices and the CGDA can influence prize distributions and introduce economic uncertainties.

Architecture Recommendations

- I recommend rewriting the tests in the codebase for this audit to use the actual contracts instead of mock addresses. This will offer greater confidence during system deployment.
- Unfortunately, due to time constraints, I was unable to do so and preparing the PoC took longer because the tests lacked actual contracts. Since most of the PoolTogether system (PrizePool, TwabController, etc.) is out-of-scope, it was challenging to create a comprehensive integration test.

Approach

- During this audit, my main focus was on examining the Liquidation system and the Vault Boosters in the Pooltogether V5 protocol.
- **Day 1:** I spent time understanding the overall working of the Pooltogether V5 system and getting an overview of the codebase.
- **Day 2:** I conducted a detailed exploration of the Liquidation and Vault Booster mechanisms.
- **Day 3:** I identified potential attack vectors and edge cases in the Liquidation flow and Vault Boosters. I also created PoC to demonstrate the issue found.
- **Day 4:** I dedicated this day to preparing the final report and analysis, summarizing the findings and recommendations.

Learnings

- PoolTogether is a unique protocol that was new to me during this audit. It introduced me to the concept of a prize savings account, where users can securely deposit their funds and have the opportunity to win rewards in return.
- To be honest, PoolTogether is a fascinating protocol that stands out due to its innovative approach in both daily prize draws and the underlying continuous gradual Dutch auction math and the RNG system. The combination of these features makes it an exciting and captivating platform for users and participants in the DeFi ecosystem.

Time spent:

28 hours

asselstine (PoolTogether) acknowledged

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