PoolTogether Audit

JANUARY 16, 2020 | IN SECURITY AUDITS | BY OPENZEPPELIN SECURITY

PoolTogether is a protocol that allows users to join a trust-minimized no-loss lottery on the Ethereum network.

The team asked us to review and audit the system. We looked at the code and now publish our results.

The audited commit is `78ac6863f4616269f7d04a0ddd1d60bdfc454937` and the contracts included in the scope were:

- `BasePool`
- `DrawManager`
- `ERC777Pool`
- `MCDAwarePool`
- `Pool`
- `RecipientWhitelistERC777Pool`
- `UniformRandomNumber`

All external code and contract dependencies were assumed to work correctly. Additionally, during this audit we assumed that the administrators are available, honest, and not compromised.

Update: *All issues have been addressed or accepted by the PoolTogether team. Our analysis of the mitigations assumes the pull requests will be merged, but disregards any other potential changes to the code base. Note that PR#3 introduces the* `Blocklock` *contract and renames the* `ERC777Pool` *contract to* `PoolToken` *as part of a structural change. These new contracts are also in scope.*

Here we present our findings.

Summary

Overall, we are happy with the security posture of the team and the health of the codebase. We are pleased to see the use of small, encapsulated functions and contracts that are mostly well documented. We have some reservations about the privileged roles but we are glad to find that the team has considered the implications of their threat model with an intention to upgrade the design where appropriate.

System Overview

The system is a pool contract that accepts ERC20 tokens and deposits them into Compound Finance to earn interest, which is credited to the contract (not to the depositors). At any point in time, assuming Compound has available liquidity, users can withdraw their original ERC20 deposit to recover their initial value (at most, forfeiting the opportunity cost associated with the funds).

The deposits are grouped into time windows, known as *draws*. Before a new draw is opened, a lottery is created and any interest held by the contract is assigned to the winner. Each user's probability of winning is proportional to their total deposits in previous draws. It should be noted that the deposits in the current open draw are not eligible, so users cannot simply make a large deposit immediately before the transition in the hope of winning the lottery and then withdraw it immediately after. Users can also optionally deposit into a pool (increasing the interest

earned by the contract) without entering the lottery. This last method is known as *sponsorship*.

Additionally, users are assigned a new ERC777 token representing their deposits in committed draws (that is, draws that are eligible for the current lottery). This makes it possible to transfer all or part of their stake in a pool. Of course, they still have the option of redeeming these tokens for the equivalent number of underlying ERC20 tokens if desired.

To accommodate the ongoing transition from single-collateral DAI to multi-collateral DAI in the broader Ethereum ecosystem, the DAI pool contract also contains a mechanism for users to easily exchange their SAI Pool tokens for DAI Pool tokens.

Privileged Roles

The pool contract is managed by administrators with wide-ranging powers. These powers include the ability to upgrade the contracts with completely new functionality. Naturally, in the hands of a malicious or compromised administrator, this includes the ability to freeze or steal the funds held by the pool contract.

Additionally, administrators are involved in the regular operation of the system. For instance, the process to create a new draw is triggered by an administrator at a time of their choosing. They also choose (and pre-commit) the entropy that is used in the lottery. In the event that they forget the entropy, they have the option of opening a new draw without running a lottery, in which case the accrued interest is simply rolled into the subsequent lottery. An administrator can also pause the pool contract, which prevents new deposits but does not prevent token transfers or withdrawals. Lastly, the possible ERC777 token recipients are currently restricted to a whitelist, which is managed entirely by the administrators.

The PoolTogether team intends to progressively decentralize many of these powers.

Here we present our findings

Critical Severity

None. 🙂

High Severity

[H01] Users can influence the lottery winner

When the administrator calls `reward` or `rewardAndOpenNextDraw`, the secret and salt that will determine the lottery winner is revealed.

However, the selected address will depend on the distribution of committed draws, which can be influenced by sending pool tokens to another address, burning pool tokens, or withdrawing an address' entire balance. It can also be influenced by sending SAI pool tokens to the DAI pool contract.

This gives users an opportunity to front-run the administrator transaction (by setting higher gas prices or mining the block themselves) in order to control the pool distribution and ensure an address they control will win the lottery. Consider freezing the committed distribution before revealing the lottery secret.

Update: *Fixed in PR#3. There is a new administrator function that can temporarily freeze all Pool Token balances, which should be called before the lottery secret is revealed. Additionally, it includes a cool down period, set during the Pool initialization, to prevent the administrator from repeatedly calling this function and keeping the balances frozen indefinitely. Naturally, this restriction can be changed or bypassed if the contract is upgraded.*

[H02] Winners can stall the system

Before each new draw is created, the previous one must be rewarded. In the reward process, the `awardWinnings` function of the `ERC777Pool` contract, mints the new Pool tokens for the winner. Since the Pool Tokens are an ERC777, they first call the `tokensReceived` hook for the winner's address, if it exists.

If the winner's `tokensReceived` hook reverts, it will prevent the reward from being applied, stalling the whole system. They could also use this capability as leverage to extract resources (for instance, by writing a hook that will only succeed after receiving a payment).

In the current version, the administrator can still bypass the reward step using the `rollover` feature. Naturally, this should not be relied upon as a mitigation since it introduces a new discretionary role for the administrator, and the feature will eventually be removed.

As detailed in "**[M01] Double counting rewards**", assigning the reward should not be treated as a Pool Token minting event. Consider removing the `awardWinnings` function in the `ERC777Pool` contract, and instead relying on the overridden function in the `BasePool` contract.

Update: *Fixed in PR#3. The* `awardWinnings` *function has been removed.*

Medium Severity

[M01] Double counting rewards

After each draw with a winner, the `awardWinnings` function is called. This updates the `balances` mapping, adds the reward to the current open draw on the winner's behalf and emits the `Minted` and `Transfer` events. However, at this point in the process, the new Pool Tokens have not been created (since the deposit is in the open draw).

When the draw is subsequently committed, the balance of the draw becomes active and the corresponding events are emitted. This means that the `Minted` and `Transfer` events associated with the reward are emitted twice: first sending the prize to the winner address and then implicitly when the open supply is sent to the contract. This will cause a mismatch between the total supply created and the `Minted` events.

Consider removing the `awardWinnings` function in the `ERC777Pool` contract, and instead relying on the overridden function in the `BasePool` contract.

Note: this issue is related to "**[H02] Winners can stall the system**" and any mitigation should consider both simultaneously.

Update: *Fixed in PR#3. The* `awardWinnings` *function has been removed. This pull request actually removes the* `Minted` *and* `Transfer` *events entirely as part of a broader code refactoring, but they are reintroduced in PR#4*

[M02] Bypassing token events

Pool Tokens can be redeemed by calling the `burn` function on the `ERC777Pool` contract. This will emit the `Burned` and `Transfer` events.

However, users can also call the `withdraw` function, which does not emit the events, to redeem their full balance of underlying tokens.

This will prevent users from reacting to these state changes from the ERC777 events (although if they are aware of the code structure they could respond to the `Withdrawn` event). It also means that the `Minted` and `Burned` events will not track the total token supply. Note that the `Withdrawn` event does not compensate for this because it does not distinguish between committed balances, open draw balances, sponsorship balances, and fees. Consider either preventing the `withdraw` function from applying to committed

deposits (that have corresponding Pool tokens), or otherwise modifying it to emit the appropriate events.

Note: this issue is related to "**[L05] Conflated balances**" and any mitigation should consider them both simultaneously.

Update: *Fixed in PR#4. The* `Minted` *and* `Burned` *events are emitted when committed balances are withdrawn from the pool.*

Low severity

[L01] Deviation from ERC777 specification

Pool Tokens are created in a non-conventional way. Whenever users deposit assets into the system, they are internally accounted for but the new balances are not accessible to the ERC777 token functions. At the end of the draw, when the balances become available, it is no longer practical to create the corresponding `Minted` and `Transfer` events for each user. Instead, these events are emitted once for all users, with the recipient set to the Pool contract.

This is a deviation from the ERC777 specification and makes it impossible to track balances using the event logs. This is already acknowledged and documented by the PoolTogether team, but we believe it should be stated in this report anyway for the sake of transparency and community awareness.

[L02] Only direct deposits are pausable

The `BasePool` contract implements a mechanism to allow an administrator to pause the pool contract. However, only the direct deposit functions (`depositPool` and `depositSponsorship`) are affected. In particular, it is arguable that indirect deposits of DAI through the SAI migration mechanism should also be paused for consistency.

Depending on the intended uses of the pause functionality, it may be desirable to permit the use of the other functions anyway. Nevertheless, it is surprising that the balances and contract state can change while the contract is paused. Consider documenting this decision and the corresponding rationale.

Update: *Fixed in PR#7. The names and documentation have been updated to clarify that the intention is to pause deposits into deprecated pools. Paused contracts are now also prevented from accepting indirect deposits and converting any unexpected token balance into a sponsorship.*

[L03] Double spending ERC20 allowance

Like all compliant ERC20 tokens, the `ERC777Pool` contract is vulnerable to the allowance double spending attack. Briefly, an authorized spender could spend both allowances by front running an allowance-changing transaction. Consider implementing OpenZeppelin's decreaseAllowance and increaseAllowance functions to help mitigate this.

Update: *Fixed in PR#16. The functions were implemented.*

[L04] Unexpected Side Effects

Each Pool Token is a claim on an equivalent amount of the underlying token. The `burn` and `operatorBurn` functions of the `ERC777Pool` contract destroy the Pool Tokens, redeem the equivalent value of cTokens in exchange for the underlying asset from Compound, and then return the underlying asset to the token holder. This is the standard mechanism for exiting the PoolTogether system.

However, the conventionally understood definition of burning ERC20 or ERC777 tokens means sacrificing the token values by sending them to the zero address. As

it turns out, this is one step in the redeem functionality, but there are other side effects as well.

Consider adding `redeem` and `operatorRedeem` functions to handle the standard withdrawal mechanism. The `burn` and `operatorBurn` functions should simply destroy tokens (and they may also prevent some or all users from burning tokens).

Update: *Fixed in PR#8. The functionality to exchange pool tokens for underlying tokens is now known as "redeeming". The* `burn` *and* `operatorBurn` *functions revert.*

[L05] Conflated balances

The comments on the `totalBalanceOf` function suggest that the user's total balance is comprised of the underlying token balance in open and committed draws. In fact, their underlying balance could also increase when receiving fees or when choosing to sponsor the lottery.

Depending on the intention of the `totalBalanceOf` function, either the code or the comments should be updated for consistency.

Additionally, since these increases never emit `Minted` events, update the committed supply, or effect the `balanceOf` function, they aren't and won't be tokenized into Pool Tokens. This means there is no mechanism to withdraw them individually. Instead users must call the `withdraw` function to redeem their full balances across all draws.

Consider allowing partial withdraws in the `withdraw` function or providing another mechanism to retrieve balances that are outside all draws.

Note: this issue is related to "**[M02] Bypassing token events**" and any mitigation should consider them both simultaneously.

Update: *Fixed in PR#4. The* `totalBalanceOf` *function comments have been updated. Additional events and functions have been created to support withdrawing from the different user balances. Note that the* `Withdrawn` *event no longer applies to token transfers between users.*

[L06] Misleading comments and variable names

Since the purpose of the Ethereum Natural Specification (NatSpec) is to describe the code to end users, misleading statements should be considered a violation of the public API.

- The `@return` comment describing `BasePool.getDraw` only describes 4 of the 8 return values.
- The `@notice` function of `BasePool.balanceOf` states that it returns the user's total balance but it only returns the committed balance.
- The `rewardAndOpenNextDraw` function and the `reward` function of `BasePool` do not have a `@param` comment for the salt.
- The `BasePool` contract returns the error message "could not transfer winnings" even though it applies to all balances.
- The `@notice` function of `DrawManager.draw` does not describe the case where are no participants.

In addition, the following internal documentation could be clarified:

- The `MAX_LEAVES` constant does not constrain the number of leaves in the sortition tree. It should be a synonym of `MAX_BRANCHES_PER_NODE` or `DEGREE_OF_TREE`. It is also missing its comment.
- Many of the BasePool functions are not documented.
- The `emitCommitted` functions in the `BasePool` contract and the `ERC777Pool` contract claim to

commit the current draw. In fact, they simply emit events. The relevant state changes occur when the new draw is opened.

- The comments describing `ERC777Pool._callTokensReceived` do not include the last parameter.
- The `requireOpenDraw` and `onlyNonZero` modifiers are missing their comments.
- The `RecipientWhitelistERC777Pool` contract and most of its functions are not commented.

Update: *Fixed in PR#21. These suggestions were implemented and the documentation has been significantly expanded.*

[L07] Excessive code coupling

During a transfer of Pool Tokens, the balance gets added to the recipient's committed draw tree. If the recipient does not have any committed balance, it is added to a newly created balance associated with the previous draw.

However, if the pool is currently in the first draw, which starts at index 1, this will associate the new balance with the invalid zero draw, and will also leave the user's first draw index at zero. This is an inconsistent state that would prevent the recipient from withdrawing, transferring or receiving awards based on their balance.

Fortunately, the overall semantics of the system prevent this scenario. In particular, no user should have any Pool Tokens during the first draw, so the situation could not arise.

Nevertheless, it is bad practice to rely on global properties of the system to prevent local edge cases and it makes the code fragile to unrelated changes (for example, if a version of the code that pre-minted tokens was released, it would reintroduce this vulnerability).

Consider confirming that the first draw is committed before assigning deposits to the previous draw.

Update: *Fixed in PR#9. The* `depositCommitted` *and* `withdrawCommitted` *functions now require at least one draw to be committed.*

[L08] Unchecked casting from uint256 to int256

The `BasePool` contract uses the `FixidityLib` to perform fixed point arithmetic with protection against overflow.

The `newFixed` function of the library accepts an `int256` as the parameter so the `uint256` variables `_feeFraction` and `_grossWinnings` first need to be cast into `int256` values.

If one of those parameters is higher than the maximum `int256`, the cast will overflow. This realistically should not occur but it is nevertheless good coding practice to explicitly check any assumptions.

Consider ensuring that neither parameter exceeds the maximum `int256`.

Update: *Fixed in PR#12. The* `_grossWinnings` *variable is now capped at the maximum safe value. The* `_feeFraction` *was already restricted by the contract logic to be less than 1e18 so it could not cause an overflow.*

Notes

[N01] Unrestricted token ownership

Whenever a pool token is transferred, the `RecipientWhitelistERC777Pool` contract restricts the possible recipients to an administrator-defined white list. It should be noted that this does not prevent

addresses from receiving tokens in exchange for deposits or winning them in a lottery.

Update: *This is the expected behavior*

[N02] Inconsistent imports

The code base imports contracts from the OpenZeppelin `contracts` package as well as `contracts-ethereum-package`. This is unnecessary and may cause issues if there is a name collision with imported contracts across both packages (or the contracts they depend on). In this case there is no collision, but it does introduce unnecessary fragility.

Consider using `contracts-ethereum-package` exclusively, which is a copy of `contracts` that is consistent with the OpenZeppelin `upgrades` package.

Update: *Fixed in PR#10.*

[N03] Default Visibility

Throughout the code base, some of the contract variables use default visibility. For readability, consider explicitly declaring the visibility of all state variables.

Update: *Fixed in PR#13.*

[N04] Reimplementing Pausable

The `BasePool` contract allows an administrator to pause and resume some functions. The functionality is already part of OpenZeppelin contracts, which has been audited and is constantly reviewed by the community.

Consider inheriting from the OpenZeppelin `Pausable` contract to benefit from bug fixes to be applied in future releases and to reduce the code's attack surface.

Update: *Accepted. PoolTogether would prefer not to adopt this suggestion since it would change the storage layout of an existing contract.*

[N05] Repeated code

The `RecipientWhitelistERC777Pool` contract overrides the `_callTokensToSend` function to restrict the possible recipients. However, the rest of the function is identical. For simplicity, consider invoking the overridden function to execute the `tokensToSend` hook.

Update: *Fixed in PR#4.*

[N06] Random upper bound of zero

The `uniform` function of the `UniformRandomNumber` library returns zero whenever the specified upper bound is zero. This contradicts the Ethereum Natural Specification comment and is inconsistent with the usual behavior of returning a value strictly less than the upper bound.

Consider requiring the upper bound to be non-zero, or updating the comment accordingly.

Update: *Fixed in PR#14. The bound is now required to be greater than zero. The edge case is handled in the calling function.*

[N07] Semantic Overloading

The pool contract identifies if a particular draw has been rewarded by checking if the entropy is non zero. This works because the winner is rewarded in the same function that the entropy is revealed, and it is highly unlikely to be zero.

However, this is an example of semantic overloading. It also necessitates an arbitrary fake entropy value to be used whenever the administrator cannot reveal the entropy. We did not identify any vulnerabilities arising from this pattern, but it does make the code more fragile.

Consider including an explicit contract variable that tracks if the committed draw has been rewarded.

Update: *Acccepted. Since the rollover mechanism and entropy source will both be updated, PoolTogether would prefer not to introduce new state that will need to be deprecated.*

[N08] Unnecessary casting of drawIndex

During the `draw` function in the `DrawManager` library, the relevant tree index is obtained with the `draw` method of the `SortionSumTreeFactory`, which returns a `bytes32` value. It is then cast to a `uint256` and saved in the `drawIndex` variable. However, `drawIndex` is used twice to reference the selected tree, where it has to be cast back to a `bytes32` value each time.

Consider removing the redundant cast into a `uint256` type.

Update: *Fixed in PR#17.*

[N09] Unnecessary SafeMath sum operation

In the `committedBalanceOf` function from the `DrawManager` contract, a `balance` variable is created to add the funds deposited under the `firstDrawIndex` and `secondDrawIndex`.

When the funds under the `firstDrawIndex` are added to the `balance`, `balance` always equals zero, making the addition unnecessary.

For simplicity and clarity, consider changing the `SafeMath` addition into a simple assignment.

Update: *Fixed in PR#18.*

[N10] Instances of uint

Throughout the code base, some variables are declared with type `uint`. To favor explicitness, consider changing all instances of `uint` to `uint256`.

Update: *Fixed in PR#19.*

[N11] Naming

To favor explicitness and readability, several parts of the contracts may benefit from better naming.
Our suggestions are:

In `DrawManager.sol`:

- `usersFirstDrawIndex` to `consolidatedDrawIndex`
- `usersSecondDrawIndex` to `latestDrawIndex`

In `BasePool.sol`:

- `Opened` to `DrawOpened`
- `Committed` to `DrawCommitted`
- `Rewarded` to `DrawConcluded`
- `Paused` to `PoolPaused`
- `Unpaused` to `PoolUnpaused`
- `open` to `openDraw`

Update: *Partially fixed in PR#20. The event names remain unchanged to maintain consistency with the deployed contract.*

Conclusion

No critical and two high severity issues were found. Some changes were proposed to follow best practices and reduce potential attack surface.

Security Audits

- If you are interested in smart contract security, you can continue the discussion in our forum, or even better, join the team 🚀
- If you are building a project of your own and would like to request a security audit, please do so here.

RELATED POSTS

SECURITY AUDITS

Freeverse Audit

The Freeverse team asked us to review and audit their NFT marketplace. We looked at the code and...

READ MORE

SECURITY AUDITS

Celo Contracts Audit – Release 7 – Part 1

The Celo team asked us to review and audit Release 7 smart contracts. We looked at the code and now...

READ MORE

SECURITY AUDITS

Celo Contracts Audit – Release 7 – Part 2

Release 7 – Part 2

The Celo team asked us to review and audit Release 7 smart contracts. We looked at the code and now...

READ MORE

OpenZeppelin

Email*

Get our monthly news roundup

SIGN UP

Products

Contracts
Defender

Security

Security Audits

Learn

Docs
Forum
Ethernaut

Company

Website
About
Jobs
Logo Kit

Privacy | Terms of Service