PoolTogether – Pods Audit

OPENZEPPELIN SECURITY | APRIL 21, 2020

Security Audits

PoolTogether is a protocol that allows users to join a trust-minimized no-loss lottery on the Ethereum network. Pods is a new feature that will allow users to pool their lottery tickets and share the rewards.

Having previously audited the main system, the PoolTogether team asked us to review and audit the new Pods feature. We looked at the code and now publish our results.

The audited commit is `8041b3dc72efd02b94d49fb37b9b308603af5ce` and the contracts included in the scope were:
– `ExchangeRateTracker`,
– `FixedPoint`,
– `ScheduledBalance`,
– `Pod`

Additionally, we reviewed the `callRewarded function` in the original system, which is required to support the Pods feature.

All external code and contract dependencies were assumed to work as documented.

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.*

Here we present our findings.

with the original audit, we are pleased to see the use of small, encapsulated functions and well-documented contracts. Our only concern is the lack of access controls. Not limiting who is able to join a pod may undermine the intention of the feature (see **“[N01] Fragile Use Case”**).

System Overview

Our original audit report contains a description of the main system.

The Pods feature extends the system to introduce an optional intermediate contract between users and a particular pool. This intermediate contract aggregates user funds and deposits them into the PoolTogether system on behalf of the users. In exchange, users receive Pod shares in the form of a new ERC777 token.

If the Pod contract wins a lottery, it updates the internal exchange rate between the Pod tokens and the Pool tokens. This effectively distributes the winnings to all shareholders in the pod.

Critical severity

[C01] Supply is manipulable

When a user deposits collateral, the total supply and their individual balance are scheduled to be updated in the next draw. However, if the collateral is withdrawn before the next round, the user’s individual balance is updated but the supply is not.

When the supply is consolidated, additional Pod tokens will be minted that are not assigned to any user.

Subsequently, when the pod wins a lottery, the new Pool tokens will be spread evenly over all Pod tokens, even the ones that are unassigned. This means that users will receive less than their fair share of the winnings.

Consider updating the scheduled supply when withdrawing a pending deposit. More generally, consider abstracting the interaction with scheduled user balances and supply so they are both updated with the same call.

Update: *Fixed in PR#2. The supply is updated when withdrawing a pending deposit.*

The `_deposit` function of the Pod contract attempts to take collateral from the `operator` but credits it to the `from` address.

If successful, this effectively transfers funds from the operator to the `from` address. On the other hand, if the operator has insufficient funds or the Pod contract does not have approval, the `operatorDeposit` function will revert. Either scenario is undesirable.

Consider updating the `transferFrom` arguments to retrieve collateral from the `from` address.

Update: *Fixed in PR#3. Collateral is correctly taken from the `from` address.*

Medium severity

[M01] Incomplete ERC777 functionality

When a user deposits funds into a pod, they are scheduled to receive Pod tokens after the current draw. Conceptually, when the draw is over, they should have full access to the ERC777 functionality of their Pod tokens. In practice, they only receive the tokens once consolidateBalanceOf is called (which occurs on any subsequent state-changing interaction with the Pod-specific functions).

In the mean time, the `send`, `transfer` and `transferFrom` functions do not account for the new tokens.

Consider extending these methods to call `consolidateBalanceOf`. Alternatively, consider making `consolidateBalanceOf` a public function so users can manually consolidate their tokens.

Update: *Fixed in PR#4. The `Pod` contract extends `send`, `operatorSend`, `transfer` and `transferFrom` to call `consolidateBalanceOf` on the relevant address before calling the corresponding parent method.*

Low severity

return anything. Additionally, its return value is not checked in the the only place it is used.

Consider removing the return value from the function signature.

Update: *Fixed in PR#5. The return value was removed from the* `clearConsolidated` *function signature.*

[L02] Misleading comments

Some of the code comments could be clearer.
– In line 85 of FixedPoint.sol: `fixed point 18 number` should be `fixed point 18 mantissa` to be consistent with the convention used in the code base.
– In line 106 and line 119 of `ExchangeRateTracker.sol`: the phrase `in the past` should be `at the specified timestamp`.
– In line 80 and line 91 of `Pod.sol`: `burned` should be `redeemed`
– In line 113 of Pod.sol: `debited` should be `credited`

Update: *Fixed in PR#6. The comments were updated appropriately.*

[L03] Complicated Code

There are cases where the data structure leads to unnecessarily complicated code. In particular the scheduled balances track the `previousBalance` and `lastBalance` separately. However, the only time the previous balance is updated occurs immediately after the balance is consolidated, and only when the `currentTimestamp` is after the `lastTimestamp`. In this case, the `previousBalance` will be increased by zero. This implies it will always be zero.

This fact is not obvious, but it is implicitly assumed when the scheduled balances are converted to tokens (since all balances use the same exchange rate, no matter when they were added). While correct, it makes the code harder to reason about.

Consider simplifying the scheduled balance data structure to record a single balance with its timestamp.

Notes & Additional Information

[N01] Fragile Use Case

Instead of simply depositing funds in Compound Finance directly and receiving small steady interest payments, users of the PoolTogether system receive large sporadic payoffs.

Pods allow users to choose an intermediate point on this spectrum, where the probability of winning a particular lottery scales with the size of the pod, but each user's share of the winnings is reduced accordingly.

It seems that the usefulness of this feature depends on each user's ability to choose the size of the pod that they would like to join, and hence where they fall on the payoff spectrum. They may also want to join a pod with specific participants. However, the lack of access control or lock-in undermines this goal, since a given user may find themselves in a significantly larger or smaller pod than they anticipated, due to the actions of other users.

Consider introducing access control or lock-in requirements, or at least documenting the reason for their absence.

Update: *The PoolTogether team have indicated that this is the intended behavior, since they prefer to launch this feature with minimal initial restrictions.*

[N02] Magic constant

When a pod wins a lottery, the `rewarded` function is executed with a stipend of 200000 gas. The reason for this decision was explained to us during the audit but we believe it should also be documented in the contract. Consider documenting the reason for the limit, as well as why the particular value of 200000 was chosen.

Update: *Fixed in PoolTogether PR#25. The* `BasePool.callRewarded` *comment includes an explanation of the stipend.*

[N03] Duplicated Code

Similarly, the `tokenToCollateralValue` function and the `collateralToTokenValue` function could reuse the `tokenToCollateralValueAt` and `collateralToTokenValueAt` functions respectively (perhaps after a minor refactor).

Update: *Partially fixed in PR#8. The `currentExchangeRateMantissa` function was removed. The conversion functions were retained for convenience.*

[N04] Reinitialize exchange tracker

The `ExchangeRateTracker` data structure can be reinitialized, effectively clearing its contents.

To improve predictability, consider preventing `initialize` from being called on an initialized tracker. If desired, a `reset` function can be used instead.

Update: *Fixed in PR#9. There is a check to ensure `initialize` can only be called once.*

[N05] Unnecessary addition

When calculating the unconsolidated balance, the result may be calculated by adding zero to the desired value. Consider setting the `result` directly without the unnecessary addition.

Update: *This issue is obsolete because the function was removed when addressing **"[L03] Complicated Code"**.*

[N06] Typographical errors

Here are some typographical errors within the codebase:
* In line 25 of ExchangeRateTracker.sol: `it's backing` should be `its backing`.
* In lines 54-56 of ScheduledBalance.sol:
* `deposit are consolidated` should be `deposits are consolidated`
* `last deposit self` should be `last deposit in self`
* In line 123 of ScheduledBalance.sol: `timeslot` should be `timestamp`
* In line 153 of ScheduledBalance.sol: `give the current timestamp` should be `given the current timestamp`

Update: *Fixed in PR#10.*

Conclusion

One critical and one high severity issue was found. Some changes were proposed to follow best practices and reduce potential attack surface.

Related Posts

Beefy Zap Audit

BeefyZapRouter serves as a versatile intermediary designed to execute users' orders through routes...

Security Audits

OpenBrush Contracts Library Security Review

OpenBrush is an open-source smart contract library written in the Rust programming language and the...

Security Audits

Linea Bridge Audit

Linea is a ZK-rollup deployed on top of Ethereum. It is designed to be EVM-compatible and aims to...

Security Audits

OpenZeppelin