# PoolTogether
# Findings & Analysis Report

2021-09-16

## Table of contents

# Overview

## About C4

Code 432n4 (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 code contest 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 code contest outlined in this document, C4 conducted an analysis of PoolTogether smart contract system written in Solidity. The code contest took place between June 16—June 23.

## Wardens

12 Wardens contributed reports to the PoolTogether code contest:

1. cmichel
2. 0xRajeev
3. shw
4. tensors
5. pauliax

6. • gpersoon
7. • Jmukesh
8. • GalloDaSballo
9. • jvaqa
10. • hrkrshnn
11. • a_delamo
12. • axic

This contest was judged by LSDan (ElasticDAO).

Final report assembled by moneylegobatman and ninek.

## Summary

The C4 analysis yielded an aggregated total of 30 unique vulnerabilities. All of the issues presented here are linked back to their original finding

Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity, 7 received a risk rating in the category of MEDIUM severity, and 18 received a risk rating in the category of LOW severity.

C4 analysis also identified 6 non-critical recommendations and 34 gas optimizations.

## Scope

The code under review can be found within the C4 PoolTogether code contest repository is comprised of 10 smart contracts written in the Solidity programming language and includes 1,395 lines of Solidity code.

## Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

## High Risk Findings (5)

### [H-01] User could lose underlying tokens when redeeming from the `IdleYieldSource`

*Submitted by shw*

The `redeemToken` function in `IdleYieldSource` uses `redeemedShare` instead of `redeemAmount` as the input parameter when calling `redeemIdleToken` of the Idle yield source. As a result, users could get fewer underlying tokens than they should.

When burning users' shares, it is correct to use `redeemedShare` (line 130). However, when redeeming underlying tokens from Idle Finance, `redeemAmount` should be used instead of `redeemedShare` (line 131). Usually, the `tokenPriceWithFee()` is greater than `ONE_IDLE_TOKEN`, and thus `redeemedShare` is less than `redeemAmount`, causing users to get fewer underlying tokens than expected.

Recommend changing `redeemedShare` to `redeemAmount` at line L131.

PierrickGT (PoolTogether) confirmed and patched:

> PR: https://github.com/pooltogether/idle-yield-source/pull/4

## [H-02] `YearnV2YieldSource` wrong subtraction in withdraw

*Submitted by cmichel, also found by 0xRajeev*

When withdrawing from the `vault`, one redeems `yTokens` for `token`s, thus the `token` balance of the contract should increase after withdrawal. But the contract subtracts the `currentBalance` from the `previousBalance`:

```
    uint256 yShares = _tokenToYShares(amount);
    uint256 previousBalance = token.balanceOf(address(this));
    // we accept losses to avoid being locked in the Vault (if losse
    if(maxLosses != 0) {
        vault.withdraw(yShares, address(this), maxLosses);
    } else {
        vault.withdraw(yShares);
    }
    uint256 currentBalance = token.balanceOf(address(this));
    // @audit-issue this seems wrong
    return previousBalance.sub(currentBalance);
```

All vault withdrawals fail due to the integer underflow as the `previousBalance` is less than `currentBalance`. Users won't be able to get back their investment.

Recommend that It should return `currentBalance > previousBalance ? currentBalance - previousBalance : 0`

**kamescg (PoolTogether) confirmed and patched:**

- https://github.com/pooltogether/pooltogether-yearnv2-yield-source/pull/new/fix/90
- https://github.com/pooltogether/pooltogether-yearnv2-yield-source/pull/7

## [H-03] `BadgerYieldSource` `balanceOfToken` share calculation seems wrong

*Submitted by cmichel*

When suppling to the `BadgerYieldSource`, some `amount` of `badger` is deposited to `badgerSett` and one receives `badgerSett` share tokens in return which are stored in the `balances` mapping of the user. So far this is correct.

The `balanceOfToken` function should then return the redeemable balance in `badger` for the user's `badgerSett` balance. It computes it as the pro-rata share of the user balance (compared to the total-supply of `badgerSett`) on the `badger` in the vault:

```
balances[addr].mul(
  badger.balanceOf(address(badgerSett))
).div(
  badgerSett.totalSupply()
)
```

However, `badger.balanceOf(address(badgerSett))` is only a small amount of badger that is deployed in the vault ("Sett") due to most of the capital being deployed to the *strategies*. Therefore, it under-reports the actual balance:

> Typically, a Sett will keep a small portion of deposited funds in reserve to handle small withdrawals cheaply. **Badger Docs**

Any contract or user calling the `balanceOf` function will receive a value that is far lower than the actual balance. Using this value as a basis for computations will lead to further errors in the integrations.

Recommend using `badgerSett.balance()` instead of `badger.balanceOf(address(badgerSett))` to also account for "the balance in the Sett, the Controller, and the Strategy".

**asselstine (PoolTogether) confirmed**

## [H-04] withdraw timelock can be circumvented

*Submitted by cmichel*

One can withdraw the entire `PrizePool` deposit by circumventing the timelock. Assume the user has no credits for ease of computation:

- user calls `withdrawWithTimelockFrom(user, amount=userBalance)` with their entire balance. This "mints" an equivalent `amount` of `timelock` and resets `_unlockTimestamps[user] = timestamp = blockTime + lockDuration`.
- user calls `withdrawWithTimelockFrom(user, amount=0)` again but this time withdrawing `0` amount. This will return a `lockDuration` of `0` and thus `unlockTimestamp = blockTime`. The inner `_mintTimelock` now resets `_unlockTimestamps[user] = unlockTimestamp`
- As `if (timestamp <= _currentTime())` is true, the full users amount is now transferred out to the user in the `_sweepTimelockBalances` call.

Users don't need to wait for their deposit to contribute their fair share to the prize pool. They can join before the awards and leave right after without a penalty which leads to significant issues for the protocol. It's the superior strategy but it leads to no investments in the strategy to earn the actual interest.

Recommend that the unlock timestamp should be increased by duration each time, instead of being reset to the duration.

**asselstine (PoolTogether) confirmed**:

> Mitigation:
>
> If a user's timelock balance is non-zero, the prize strategy rejects the ticket burn.

## [H-05] `IdleYieldSource` doesn't use mantissa calculations

*Submitted by tensors*

Because mantissa calculations are not used in this case to account for decimals, the arithmetic can zero out the number of shares or tokens that should be given.

For example, say I deposit 1 token, expecting 1 share in return. On L95, if the `totalUnderlyingAssets` is increased to be larger than the number of total shares,

then the division would output 0 and I wouldn't get any shares.

Recommend implementing mantissa calculations like in the contract for the AAVE yield.

PierrickGT (PoolTogether) confirmed and patched:

> PR: https://github.com/pooltogether/idle-yield-source/pull/5

## Medium Risk Findings (7)

### [M-01] `safeApprove()` for Yearn Vault may revert preventing deposits causing DoS

*Submitted by 0xRajeev, also found by pauliax*

The `_depositInVault()` function for Yearn yield source uses ERC20 `safeApprove()` from OpenZeppelin's SafeERC20 library to give maximum allowance to the Yearn Vault address if the current allowance is less than contract's token balance.

However, the `safeApprove` function prevents changing an allowance between non-zero values to mitigate a possible front-running attack. It reverts if that is the case. Instead, the `safeIncreaseAllowance` and `safeDecreaseAllowance` functions should be used. Comment from the OZ library for this function:

> "// `safeApprove` should only be called when setting an initial allowance, // or when resetting it to zero. To increase and decrease it, use // 'safeIncreaseAllowance' and 'safeDecreaseAllowance'"

If the existing allowance is non-zero (say, for e.g., previously the entire balance was not deposited due to vault balance limit resulting in the allowance being reduced but not made 0), then `safeApprove()` will revert causing the user's token deposits to fail leading to denial-of-service. The condition predicate indicates that this scenario is possible. See similar Medium-severity finding M03.

Recommend using `safeIncreaseAllowance()` function instead of `safeApprove()`.

kamescg (PoolTogether) confirmed and patched:

- https://github.com/pooltogether/pooltogether-yearnv2-yield-source/pull/new/fix/71
- https://github.com/jmonteer/pooltogether-yearnv2-yield-source/pull/6

## [M-02] Return values of ERC20 `transfer` and `transferFrom` are unchecked

*Submitted by shw, gpersoon, JMukesh, also found by a*delamo and cmichel_

In the contracts `BadgerYieldSource` and `SushiYieldSource`, the return values of ERC20 `transfer` and `transferFrom` are not checked to be `true`, which could be `false` if the transferred tokens are not ERC20-compliant (e.g., `BADGER`). In that case, the transfer fails without being noticed by the calling contract.

If warden's understanding of the `BadgerYieldSource` is correct, the `badger` variable should be the `BADGER` token at address `0x3472a5a71965499acd81997a54bba8d852c6e53d`. However, this implementation of `BADGER` is not ERC20-compliant, which returns `false` when the sender does not have enough token to transfer (both for `transfer` and `transferFrom`). See the source code on Etherscan (at line 226) for more details.

Recommend using the `SafeERC20` library implementation from Openzeppelin and call `safeTransfer` or `safeTransferFrom` when transferring ERC20 tokens.

kamescg (PoolTogether) confirmed and patched:

> Sushi

- https://github.com/pooltogether/sushi-pooltogether/pull/new/fix/112
- https://github.com/pooltogether/sushi-pooltogether/pull/11

> Badger

- https://github.com/pooltogether/badger-yield-source/pull/new/fix/112
- https://github.com/pooltogether/badger-yield-source/pull/2

dmvt (judge) commented:

> Sponsor has repeatedly stated in duplicate issues that: "It's more of a 1 (Low Risk) because the subsequent deposit calls will fail. There is no advantage to be gained; the logic is simply poor."

> I disagree with this assessment. The function(s) in question do not immediately call deposit or another function that would cause a revert. In fact the balances are updated:

```
balances[msg.sender] = balances[msg.sender].sub(requiredShares
badger.transfer(msg.sender, badgerBalanceDiff);
return (badgerBalanceDiff);
```

> The impact that this would have on the rest of the system is substantial, including causing incorrect balances to be returned and potentially lost funds.

> That said, I do not think this is very likely and so high severity seems excessive here. Im adjusting all of these reports to Medium Risk given that lower likelihood.

## [M-03] `SafeMath` not completely used in yield source contracts

*Submitted by shw, also found by cmichel*

`SafeMath` is not completely used at the following lines of yield source contracts, which could potentially cause arithmetic underflow and overflow:

1. line 78 in `SushiYieldSource`
2. line 67 in `BadgerYieldSource`
3. line 91 and 98 in `IdleYieldSource`

Recommend using the `SafeMath` library functions in the above lines.

asselstine (PoolTogether) confirmed and disagreed with severity:

> While the arithmetic ceiling is quite high, if an overflow occurred this would significantly disrupt the yield sources. I'd qualify this issue higher as `2 (Med Risk)`.

dmvt (judge) commented:

> I agree with the sponsor's risk evaluation. Increasing to medium.

## [M-04] The assumption that operator == to (user) may not hold leading to failed timelock deposits

*Submitted by 0xRajeev*

The contract uses `_msgSender()` to denote an operator who is operating on behalf of the user. This is typically used for meta-transactions where the operator is an intermediary/relayer who may facilitate gas-less transactions on behalf of the user. They may be the same address but it is safer to assume that they may not be.

While the code handles this separation of role in most cases, it misses doing so in `timelockDepositTo()` function where it accounts the `_timelockBalances` to the operator address instead of the user specified `to` address. It assumes they are the same. The corresponding usage in `_mintTimelock()` which is called from `withdrawWithTimelockFrom()` uses the user specified 'from' address and not the `_msgSender()`. Therefore the corresponding usage in `timelockDepositTo()` should be the same.

In the scenario where the operator address != user specified from/to addresses, i.e. meta-transactions, the timelock deposits and withdrawals are made to/from different addresses and so the deposits of timelocked tokens will fail because the operator's address does not have the required amount of `_timelockBalances`.

Recommend changing `operator` to `from` on L281 of `timelockDepositTo()` and specifying the scenarios where the role of the operator is applicable and document/implement those accordingly.

asselstine (PoolTogether) disputed:

> In the function `timelockDepositTo()` the msg.sender is using their timelocked funds to re-enter the pool. They can only spend their own funds; they should not be able to spend other user's funds.

> The warden is saying the `timelockDepositTo` should be callable by anyone and allow them to transfer other user's funds from the timelock back into tickets. This actually introduces an attack vector.

dmvt (judge) commented:

> I think sponsor is misunderstanding warden's concern here. The issue is not that `msg.sender` is being checked, but that `_msgSender` is being checked. Happy to discuss this more if sponsor still disagrees, but I think the concern raised is valid.

## [M-05] Actual yield source check on address will succeed for non-existent contract

*Submitted by 0xRajeev*

Low-level calls `call` / `delegatecall` / `staticcall` return true even if the account called is non-existent (per EVM design). Solidity documentation warns:

> "The low-level functions call, delegatecall and staticcall return true as their first return value if the account called is non-existent, as part of the design of the EVM. Account existence must be checked prior to calling if needed."

The `staticcall` here will return True even if the `_yieldSource` contract doesn't exist at any incorrect-but-not-zero address, e.g. EOA address, used during initialization by accident.

The hack, as commented, to check if it's an actual yield source contract, will fail if the address is indeed a contract account which doesn't implement the `depositToken` function. However, if the address is that of an EOA account, the check will pass here but will revert in all future calls to the yield source forcing contract redeployment after the pool is active. Users will not be able to interact with the pool and abandon it.

Recommend that a contract existence check should be performed on `_yieldSource` prior to the depositToken function existence hack for determining yield source contract.

asselstine (PoolTogether) confirmed

## [M-06] `YieldSourcePrizePool_canAwardExternal` does not work

*Submitted by cmichel*

The idea of `YieldSourcePrizePool_canAwardExternal` seems to be to disallow awarding the interest-bearing token of the yield source, like aTokens, cTokens, yTokens.

> "@dev Different yield sources will hold the deposits as another kind of token: such a Compound's cToken. The prize strategy should not be allowed to move those tokens."

However, the code checks `_externalToken != address(yieldSource)` where `yieldSource` is the actual yield strategy contract and not the strategy's interest-bearing token. Note that the `yieldSource` is usually not even a token contract except for `ATokenYieldSource` and `YearnV2YieldSource`.

The `_canAwardExternal` does not work as expected. It might be possible to award the interest-bearing token which would lead to errors and loss of funds when trying to redeem underlying.

There doesn't seem to be a function to return the interest-bearing token. It needs to be added, similar to `depositToken()` which retrieves the underlying token.

asselstine (PoolTogether) acknowledged:

> This is an interesting one:
>
> - the yield source interface does not require the deposit be tokenized; the implementation is entirely up to the yield source.

- the _canAwardExternal is a legacy of older code. Since it had to be included it was set to assume the yield source was tokenized.

> Since yield sources are audited and analyzed, I think this is a pretty low risk. Additionally, not all of the yield sources are tokenized (Badger and Sushi are not), so it isn't a risk for them.

> We could have `canAwardExternal` on the yield source itself, but it would add gas overhead.

**aodhgan (PoolTogether) commented**:

> Could we add an check - `function _canAwardExternal(address _externalToken) internal override view returns (bool) { return _externalToken != address(yieldSource) && _externalToken != address(yieldSource.depositToken()) }`

**asselstine (PoolTogether) commented**:

> We could add another check, but it's still arbitrary. The point is that the yield source knows what token the prize pool may or may not hold, so without asking the yield source it's just a guess.

> Let's leave it as-is

## [M-07] Using `transferFrom` on ERC721 tokens

*Submitted by shw*

In the function `awardExternalERC721` of contract `PrizePool`, when awarding external ERC721 tokens to the winners, the `transferFrom` keyword is used instead of `safeTransferFrom`. If any winner is a contract and is not aware of incoming ERC721 tokens, the sent tokens could be locked.

Recommend consider changing `transferFrom` to `safeTransferFrom` at line 602. However, it could introduce a DoS attack vector if any winner maliciously rejects the received ERC721 tokens to make the others unable to get their awards. Possible mitigations are to use a `try/catch` statement to handle error cases separately or

provide a function for the pool owner to remove malicious winners manually if this happens.

**asselstine (PoolTogether) confirmed and disagreed with severity:**

> This issue poses no risk to the Prize Pool, so it's more of a `1 (Low Risk` IMO.

> This is just about triggering a callback on the ERC721 recipient. We omitted it originally because we didn't want a revert on the callback to DoS the prize pool.

> However, to respect the interface it makes sense to implement it fully. That being said, if it does throw we must ignore it to prevent DoS attacks.

**dmvt (judge) commented:**

> I agree with the medium risk rating provided by the warden.

## Low Risk Findings (18)

### [L-01] no check for `_stakeToken` !=0

*Submitted by gpersoon*

The `initializeYieldSourcePrizePool` function of `YieldSourcePrizePool.sol` has a check to make sure `_yieldSource` !=0. However, the `initialize` function of the comparable `StakePrizePool.sol` doesn't do this check.

Although unlikely this will introduce problems, it is more consistent to check for 0.

`YieldSourcePrizePool.sol` **L24**

```
    function initializeYieldSourcePrizePool (... IYieldSource _yie
..
    require(address(_yieldSource) != address(0), "YieldSourcePrize
    PrizePool.initialize(
```

`StakePrizePool.sol` **L20**

```
function initialize ( ..  IERC20Upgradeable _stakeToken)...  {
  PrizePool.initialize(
```

Recommend adding something like the following in the initialize function of `StakePrizePool.sol`:

```
    require(address(_stakeToken) != address(0), "StakePrizePool/st
```

## [L-02] Lack of `nonReentrant` modifier in yield source contracts

_Submitted by shw, also found by gpersoon, 0xRajeev and pauliax__

The `YearnV2YieldSource` contract prevents the `supplyTokenTo`, `redeemToken`, and `sponsor` functions from being reentered by applying a `nonReentrant` modifier. Since these contracts share a similar logic, adding a `nonReentrant` modifier to these functions in all of the yield source contracts is reasonable. However, the same protection is not seen in other yield source contracts.

A `nonReentrant` modifier in the following functions is missing:

1. The `sponsor` function of `ATokenYieldSource`
2. The `supplyTokenTo` and `redeemToken` function of `BadgerYieldSource`
3. The `sponsor` function of `IdleYieldSource`
4. The `supplyTokenTo` and `redeemToken` function of `SushiYieldSource`

Recommend adding a `nonReentrant` modifier to these functions. For `BadgerYieldSource` and `SushiYieldSource` contracts, make them inherit from Openzeppelin's `ReentrancyGuardUpgradeable` to use the `nonReentrant` modifier.

**kamescg (PoolTogether) confirmed and patched**:

> ATokenYieldSource: https://github.com/pooltogether/aave-yield-source/tree/fix/119 SushiYieldSource: https://github.com/pooltogether/sushi-

> pooltogether/pull/new/fix/119 BadgerYieldSource: https://github.com/pooltogether/badger-yield-source/pull/new/fix/119 IdleYieldSource: https://github.com/pooltogether/idle-yield-source/pull/new/fix/119

## [L-03] What is default duration when `creditRateMantissa` is not set

*Submitted by gpersoon*

In `PrizePool.sol`, if the value of `_tokenCreditPlans[_controlledToken].creditRateMantissa` isn't set (yet), then the function `_estimateCreditAccrualTime` returns 0. This means the `TimelockDuration` is 0 and funds can be withdrawn immediately, defeating the entire timelock mechanism.

Recommend perhaps a different default would be useful.

`PrizePool.sol` **L783**

```
    function _estimateCreditAccrualTime( address _controlledToken,ui
      uint256 accruedPerSecond = FixedPoint.multiplyUintByMantissa(_
      if (accruedPerSecond == 0) {
        return 0;
      }
      return _interest.div(accruedPerSecond);
    }
```

`PrizePool.sol` **L710**

```
    function _calculateTimelockDuration( address from, address contr
    ...
      uint256 duration = _estimateCreditAccrualTime(controlledToken,
      if (duration > maxTimelockDuration) {
        duration = maxTimelockDuration;
      }
      return (duration, _burnedCredit);
    }
```

Recommend considering the default duration for the case `_tokenCreditPlans[_controlledToken].creditRateMantissa` isn't set.

## [L-04] `staticCall` to `yieldSource.depositToken` doesn't provide any security guarantees

*Submitted by GalloDaSballo*

The assumption that a yield source is valid, just because it has the method `depositToken`, is not a security guarantee. I could create any random contract with that function but that is not a guarantee that the contract will behave as intended.

I believe a better solution would be to have a registry, controlled by governance, that accepts the valid yield sources. A valid registry ensures the the yield sources are properly maintained.

In summary: There is no security difference between having the check and not having the check, because the check can be sidelined without any effort and doesn't truly provide any guarantee of the contract being valid. Having no checks would save you gas. While having a governance registry would guarantee that the yield sources usable are exclusively the community vetted ones.

**asselstine (PoolTogether) acknowledged:**

> It's possible for a malicious developer to fork our code and create a pool with a rugging yield source. That can't really be helped either way.

> We decide which pools to display on https://app.pooltogether.com, so we can vet pools already.

## [L-05] Switch modifier order to consistently place the non-reentrant modifier as the first one

*Submitted by 0xRajeev*

If a function has multiple modifiers they are executed in the order specified. If checks or logic of modifiers depend on other modifiers this has to be considered in their ordering. `PrizePool` has functions with multiple modifiers with one of them

being non-reentrant which prevents reentrancy on the functions. This should ideally be the first one to prevent even the execution of other modifiers in case of re-entrancies.

While there is no obvious vulnerability currently with non-reentrant being the last modifier in the list, it is safer to place it in the first. This is of slight concern with the deposit functions which have the `canAddLiquidity()` modifier (before non-reentrant) that makes external calls to get `totalSupply` of controlled tokens.

For reference, see similar finding in Consensys's audit of Balancer.

Recommend switching modifier order to consistently place the non-reentrant modifier as the first one to run so that all other modifiers are executed only if the call is non-reentrant.

kamescg (PoolTogether) confirmed and patched:

- https://github.com/pooltogether/pooltogether-pool-contracts/pull/new/fix/50
- https://github.com/pooltogether/pooltogether-pool-contracts/pull/308

## [L-06] Missing modifier `onlyControlledToken` may result in undefined/exceptional behavior

*Submitted by 0xRajeev*

The modifier `onlyControlledToken` is used for functions that allow the `controlledToken` address as a parameter to ensure that only whitelisted tokens (ticket and sponsorship) are provided. This is used in all functions except `calculateEarlyExitFee()`.

The use of a non-whitelisted `controlledToken` will result in calls to potentially malicious token contract and cause undefined behavior for the `from` user address specified in the call.

Recommend adding missing modifier `onlyControlledToken` to `calculateEarlyExitFee()`.

kamescg (PoolTogether) confirmed:

> This would likely break assumptions made by other contracts when used to get the early exit fee.

> For example in Pods to calculate the exit fee. Plus this is called statically from JS frontends to get the fee.

```
/**
  * @notice Calculate the cost of withdrawing from the Pod if th
  * @param amount Amount of tokens to withdraw when calculating
  * @dev Based of the Pod's total token/ticket balance and total
*/
function getEarlyExitFee(uint256 amount) external returns (uint2
    uint256 tokenBalance = _podTokenBalance();
    if (amount <= tokenBalance) {
        return 0;
    } else {
        // Calculate Early Exit Fee
        (uint256 exitFee, ) =
            _prizePool.calculateEarlyExitFee(
                address(this),
                address(ticket),
                amount.sub(tokenBalance)
            );
        // Early Exit Fee
        return exitFee;
    }
}
```

asselstine (PoolTogether) commented:

> @kamescg Rajeev is suggesting to add the modifier `onlyControlledToken` to `calculateEarlyExitFee()`

> That means it would revert on invalid controlled tokens. It would still be a static call!

> That being said this isn't a deal breaker. We can skip this one and it wouldn't hurt.

## [L-07] Missing calls to `init` functions of inherited contracts

*Submitted by 0xRajeev, also found by shw*

Most contracts use the `delegateCall` proxy pattern and hence their implementations require the use of `initialize()` functions instead of constructors. This requires derived contracts to call the corresponding `init` functions of their inherited base contracts. This is done in most places except a few.

The inherited base classes do not get initialized which may lead to undefined behavior.

- Missing call to `__ReentrancyGuard_init` in `ATokenYieldSource.sol` L99-L102 and L59-L61
- Missing call to `__ERC20_init` in `ATokenYieldSource.sol` L59-L61 and L83-L86

Recommend adding missing calls to init functions of inherited contracts.

PierrickGT (PoolTogether) confirmed and patched:

> ATokenYieldSource PR: https://github.com/pooltogether/aave-yield-source/pull/18

PierrickGT (PoolTogether) commented:

> Has been fixed already for IdleYieldSource: https://github.com/pooltogether/idle-yield-source/blob/master/contracts/IdleYieldSource.sol#L60-#62

PierrickGT (PoolTogether) commented:

> YearnV2YieldSource: https://github.com/pooltogether/pooltogether-yearnv2-yield-source/pull/8

## [L-08] Unlocked pragma used in multiple contracts

*Submitted by shw, also found by 0xRajeev and JMukesh*

Some contracts (e.g., `PrizePool`) use an unlocked pragma (e.g., `pragma solidity >=0.6.0 <0.7.0;`) which is not fixed to a specific Solidity version. Locking the pragma helps ensure that contracts do not accidentally get deployed using a different compiler version with which they have been tested the most.

Please use `grep -R pragma .` to find the unlocked pragma statements.

Recommend locking pragmas to a specific Solidity version. Consider the compiler bugs in the following lists and ensure the contracts are not affected by them. It is also recommended to use the latest version of Solidity when deploying contracts (see Solidity docs).

Solidity compiler bugs: Solidity repo - known bugs Solidity repo - bugs by version

kamescg (PoolTogether) confirmed:

> PrizePool

- https://github.com/pooltogether/pooltogether-pool-contracts/pull/new/fix/109
- https://github.com/pooltogether/pooltogether-pool-contracts/pull/303

> Remaining Contracts

- https://github.com/pooltogether/pooltogether-pool-contracts/pull/new/fix/109-remaining-contracts
- https://github.com/pooltogether/pooltogether-pool-contracts/pull/304

## [L-09] Missing zero-address checks

*Submitted by 0xRajeev*

Checking addresses against zero-address during initialization or during setting is a security best-practice. However, such checks are missing in all address variable initializations.

Allowing zero-addresses will lead to contract reverts and force redeployments if there are no setters for such address variables.

Recommend adding zero-address checks for all initializations of address state variables.

kamescg (PoolTogether) commented:

> Core

- https://github.com/pooltogether/pooltogether-pool-contracts/pull/new/fix/65
- https://github.com/pooltogether/pooltogether-pool-contracts/pull/306

> Aave

- https://github.com/pooltogether/aave-yield-source/pull/new/fix/65
- https://github.com/pooltogether/aave-yield-source/pull/22

> Sushi

- https://github.com/pooltogether/sushi-pooltogether/pull/new/fix/65
- https://github.com/steffenix/sushi-pooltogether/pull/21

> Idle

- https://github.com/pooltogether/idle-yield-source/pull/new/fix/65
- https://github.com/pooltogether/idle-yield-source/pull/3

> Badger

- https://github.com/pooltogether/badger-yield-source/pull/new/fix/65
- https://github.com/pooltogether/badger-yield-source/pull/6

## [L-10] Overly permissive threshold check allows high yield loss

*Submitted by0xRajeev*

The Yearn yield source defines `maxLosses` as: “Max % of losses that the Yield Source will accept from the Vault in BPS” and uses a setter `setMaxLosses()` to allow owner to set this value. However, the threshold check implemented only

checks if this value is less than 10_000 or 100%, which is a good sanity check but allows loss of even 100%. The buffer for the loss is to avoid funds being locked in the Yearn vault in any emergency situation.

If the losses are really high for some reason, it will impact the interest and the prizes.

Perform a tighter upper threshold check to allow a more acceptable max loss value in `setMaxLosses()`

**asselstine (PoolTogether) acknowledged:**

> Yield sources are controlled by governance, so this isn't a concern

## [L-11] Ignored return values may lead to undefined behavior

*Submitted by 0xRajeev*

The `_depositInVault()` returns the value returned from its call to the Yearn vault's `deposit()` function. However, the return value is ignored at both call sites in `supplyTokenTo()` and `sponsor()`.

It is unclear what the intended usage is and how, if any, the return value should be checked. This should perhaps check how much of the full balance was indeed deposited/rejected in the vault by comparing the return value of issued vault shares as commented: "The actual amount of shares that were received for the deposited tokens" because "if deposit limit is reached, tokens will remain in the Yield Source and they will be queued for retries in subsequent deposits."

Recommend checking return value appropriately or if not, document why this is not necessary.

**asselstine (PoolTogether) disputed:**

> Regardless of how much of the deposit made it into the underlying vault, the depositor will hold the correct number shares. It doesn't matter if only a portion of the funds made it into the vault.

**dmvt (judge) commented:**

> I think this is a reasonable finding by the warden. If the return value isn't needed, it should be removed or at least documented that it's there for no reason. If there is a reason to have the return value, the return value should be considered by the calling functions.

## [L-12] Using `memory[]` parameter without checking its length

*Submitted by JMukesh*

Using memory array parameters (e.g. `uint[]` memory) as function parameters can be tricky in Solidity, because an attack is possible with a very large array which will overlap with other parts of the memory.

This an example to show the exploit based on `Exploit.sol` :

```
pragma solidity ^0.4.24; // only works with low solidity versior

contract test{
    struct Overlap {
        uint field0;
    }
    event log(uint);

  function mint(uint[] memory amounts) public  returns (uint) {
       Overlap memory v;
       v.field0 = 1234;
       emit log(amounts[0]); // would expect to be 0 however is
       return 1;
     }

  function go() public { // this part requires the low solidity
      uint x=0x8000000000000000000000000000000000000000000000(
      bytes memory payload = abi.encodeWithSelector(this.mint.se
      bool success=address(this).call(payload);
  }
}
```

Recommend checking the array length before using it

**asselstine (PoolTogether) confirmed**:

> We're going to remove the timelock functions. The initializer I'm not concerned about.

## [L-13] Uneven use of events

*Submitted by JMukesh*

To track off-chain data it is necessary to use events

In `ATokenYieldSource.sol`, `IdleYieldSource.sol`, and `yearnV2yieldsource`, events are emmitted in `supplyTokenTo()`, `redeemToken()`, and `sponsor()`, but not in `BadgerYieldsource.sol` and `shushiyieldsource.sol`

Recommend using events.

asselstine (PoolTogether) confirmed

## [L-14] Missing parameter validation

*Submitted by cmichel*

Some parameters of functions are not checked for invalid values:

- `StakePrizePool.initialize: address _stakeToken` not checked for non-zero or contract
- `ControlledToken.initialize: address controller` not checked for non-zero or contract
- `PrizePool.withdrawReserve: address to` not checked for non-zero, funds will be lost when sending to zero address
- `ATokenYieldSource.initialize: address _aToken, _lendingPoolAddressesProviderRegistry` not checked for non-zero or contract
- `BadgerYieldSource.initialize: address badgerSettAddr, badgerAddr` not checked for non-zero or contract
- `SushiYieldSource.constructor: address _sushiBar, _sushiAddr` not checked for non-zero or contract

Wrong user input or wallets defaulting to the zero addresses for a missing input can lead to the contract needing to redeploy or wasted gas.

Recommend validating the parameters.

**PierrickGT (PoolTogether) confirmed and patched:**

> ATokenYieldSource PR: https://github.com/pooltogether/aave-yield-source/pull/19

**PierrickGT (PoolTogether) commented:**

> BadgerYieldSource PR: https://github.com/pooltogether/badger-yield-source/pull/6

**PierrickGT (PoolTogether) commented:**

> SushiYieldSource PR: https://github.com/pooltogether/sushi-pooltogether/pull/16

**PierrickGT (PoolTogether) commented:**

> ControlledToken PR: https://github.com/pooltogether/pooltogether-pool-contracts/pull/306

**PierrickGT (PoolTogether) commented:**

> StakePrizePool PR: https://github.com/pooltogether/pooltogether-pool-contracts/pull/314

**PierrickGT (PoolTogether) commented:**

> @asselstine (PoolTogether) I'm not sure we want to check for non zero address in the PrizePool `withdrawReserve` function since this function is only callable by the Reserve and the owner of the Reserve contract.
> https://github.com/pooltogether/pooltogether-pool-contracts/blob/192429c808ad9714e9e05821386eb926150a009f/contracts/reserve/Reserve.sol#L32 https://github.com/pooltogether/pooltogether-pool-

> contracts/blob/4449bb2e4216511b7187b1ab420118c30af39eb7/contracts/prize-pool/PrizePool.sol#L473

asselstine (PoolTogether) commented:

> Yeah @PierrickGT (PoolTogether) I don't think the `withdrawReserve` needs to do the check. Many tokens reject on transfer to zero anyway.

kamescg (PoolTogether) commented:

> LGTM

### [L-15] `ATokenYieldSource` mixes aTokens and underlying when redeeming

*Submitted by cmichel*

The `ATokenYieldSource.redeemToken` function burns `aTokens` and sends out underlying; however, it's used in a reverse way in the code: The `balanceDiff` is used as the `depositToken` that is transferred out but it's computed on the **aTokens** that were burned instead of on the `depositToken` received.

It should not directly lead to issues as aTokens are 1-to-1 with their underlying but we still recommend doing it correctly to make the code more robust against any possible rounding issues.

Recommend computing `balanceDiff` on the underyling balance ( `depositToken` ), not on the aToken. Then, subtract the actual burned aTokens from the user shares.

PierrickGT (PoolTogether) confirmed:

> I agree that we should compute `balanceDiff` on the underlying balance. Regarding the burn of the user's shares, we should keep it as is to verify first that the user has enough shares. This way if he doesn't, the code execution will revert before funds are withdrawn from Aave.

PierrickGT (PoolTogether) commented:

> PR: https://github.com/pooltogether/aave-yield-source/pull/17

## [L-16] `BadgerYieldSource` and `SushiYieldSource` are not upgradeable

*Submitted by shw*

The contracts `BadgerYieldSource` and `SushiYieldSource` are not upgradeable since they do not inherit from any Openzeppelin's upgradeable contract (e.g., ERC20Upgradeable) as the other yield source contracts.

Recommend making `BadgerYieldSource` and `SushiYieldSource` upgradable.

**asselstine (PoolTogether) disputed:**

> We don't want them to be upgradeable! It's a feature not a bug.

## [L-17] `onERC721Received` not implemented in `PrizePool`

*Submitted by shw*

The `PrizePool` contract does not implement the `onERC721Received` function, which is considered a best practice to transfer ERC721 tokens from contracts to contracts. The absence of this function could prevent `PrizePool` from receiving ERC721 tokens from other contracts via `safeTransferFrom`.

Consider adding an implementation of the `onERC721Received` function in `PrizePool`.

**kamescg (PoolTogether) confirmed**:

> https://github.com/pooltogether/pooltogether-pool-contracts/pull/new/fix/118

> https://github.com/pooltogether/pooltogether-pool-contracts/pull/300

## [L-18] Lack of event emission after critical `initialize()` functions

*Submitted by 0xRajeev*

Most contracts use `initialize()` functions instead of constructor given the `delegatecall` proxy pattern. While most of them emit an event in the critical `initialize()` functions to record the init parameters for off-chain monitoring and transparency reasons, `Ticket.sol` nor its base class `ControlledToken.sol` emit such an event in their `initialize()` functions.

These contracts are initialized but their critical init parameters (name, symbol, decimals and controller address) are not logged for any off-chain monitoring.

See similar Medium-severity Finding M01 in OpenZeppelin's audit of UMA protocol.

Recommend emitting an initialized event in `Ticket.sol` and `ControlledToken.sol` logging their init parameters.

asselstine (PoolTogether) confirmed but disagree with severity:

> This is just event emission; it's severity is `0 (Non-critical)`.

kamescg (PoolTogether) patched:

- https://github.com/pooltogether/pooltogether-pool-contracts/pull/new/fix/68
- https://github.com/pooltogether/pooltogether-pool-contracts/pull/305

dmvt (judge) commented:

> I'm going to split the difference here. Events are important for various reasons. In this case, due to the proxy pattern, the creation of the contract in the initialize function happen at the same time, making it trivial for a user to go back and look at the initialization parameters in the creation transaction.

## Non-Critical findings (6)

- [N-01] Use immutable keyword
- [N-02] uint256(-1)
- [N-03] function sponsor not all ways present

- [N-04] `PrizePool.beforeTokenTransfer()` incorrectly uses msg.sender in seven places instead of `_msgSender()`
- [N-05] Named return values are never used in favor of explicit returns
- [N-06] setPrizeStrategy check for Interface Supported in PrizePool.sol doesn't guarantee that the new prize strategy is valid

## Gas Optimizations (34)

- [G-01] cache and reuse _vault.apiVersion() result
- [G-02] Caching badger and badgerSett can save 400 gas in `supplyTokenTo()`
- [G-03] currentTime() outside of loop
- [G-04] SushiYieldSource save gas with pre-approval
- [G-05] _accrueCredit -> _updateCreditBalance
- [G-06] modifier canAddLiquidity and function _canAddLiquidity
- [G-07] function _getRefferalCode() can be refactored to a constant variable
- [G-08] Gas Optimization: PrizePool._calculateCreditBalance.creditBalance is incorrectly passed by reference rather than passed by value, causing unnecessary SLOADs instead of MLOADs
- [G-09] Upgrading the solc compiler to >=0.8 may save gas
- [G-10] Avoid use of state variables in event emissions to save gas
- [G-11] Simplifying extensible but expensive modifier may save gas
- [G-12] Gas savings of 300 by caching _currentAwardBalance in captureAwardBalance()
- [G-13] Using access lists can save gas due to EIP-2930 post-Berlin hard fork
- [G-14] Gas savings of 100 per user by caching _timelockBalances[user] in _sweepTimelockBalances()
- [G-15] Gas savings of 100 by caching maxTimelockDuration in _calculateTimelockDuration()
- [G-16] Unnecessary indirection to access block.timestamp value
- [G-17] Preventing zero-address controlled tokens from being added can avoid checks later

- [G-18] Gas savings of (100*loop-iteration-count) by caching _tokens.end() in _tokenTotalSupply()
- [G-19] _depositToAave always returns 0
- [G-20] Zero-address check unnecessary due to the initializer modifier
- [G-21] Using function parameter in `initialize()` instead of state variable saves 100 gas
- [G-22] token can be cached in a local variable to save 200 gas in `_depositInVault()`
- [G-23] token can be cached in a local variable to save 100 gas in _withdrawFromVault()
- [G-24] `maxLosses` can be cached in a local variable to save 100 gas in _withdrawFromVault()
- [G-25] Caching sushiAddr and sushiBar in local variables to save 200 gas in `supplyTokenTo()`
- [G-26] Various gas optimizations
- [G-27] ATokenYieldSource save gas with pre-approval
- [G-28] Credit accrual is done twice in `award`
- [G-29] CreditBurned event emitted even on zero tokens burned
- [G-30] Gas savings on uninitialized variables.
- [G-31] Declare functions as `external` to save gas
- [G-32] Gas optimization on `_depositToAave`
- [G-33] Gas optimization on `redeemToken` of `ATokenYieldSource`
- [G-34] Use ERC-165 instead of homebrew staticcall-based check

## Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest 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